Optimisation of a colour segmentation and tracking algorithm for real-time FPGA implementation

C.T. Johnston, D.G. Bailey, K.T. Gribbon

Institute of Information Sciences and Technology, Massey University, Palmerston North.
Email: c.t.johnston@massey.ac.nz, d.g.bailey@massey.ac.nz, k.t.gribbon@massey.ac.nz

Abstract

This paper concentrates on the optimisations and tradeoffs required to implement a colour segmentation based object tracking algorithm on a small Field Programmable Gate Array (FPGA). The algorithm consists of colour conversion, segmentation and labelling, morphological filtering, and bounding box based object recognition, and is able to track up to 4 independent coloured targets. By optimising and adapting the algorithm, the object tracking module occupies less than 10% of a Spartan 2 FPGA (XC2S200) and operates at a clock frequency of 27 MHz.

Keywords: FPGA, object tracking, image processing, real-time systems

1 Introduction

Object tracking often involves the use of a camera to provide scene data from which the motion of real-world objects is mapped to system controls [1]. Object tracking for control-based applications usually requires the use of a real-time system as sensing delays in the input can cause instability in closed-loop control. This is particularly important if the user must receive sensory feedback from the system.

While image processing at video rates can be achieved on a serial processor such as a desktop computer, the required hardware is quite cumbersome. Furthermore, as the number of objects that need to be detected and reliably tracked increases, the real-time processing capabilities of even the fastest desktop computer can be challenged.

This is due to several factors such as the large data set represented by a captured image, and the complex operations which may need to be performed on an image. At real-time video rates of 25 frames per second a single operation performed on every pixel of a 768 by 576 colour image (PAL frame) requires 33 million operations per second. This does not take into account the overhead of storing and retrieving pixel values. Tracking algorithms require several operations to be performed on each pixel in the image resulting in a large number of operations per second.

Field programmable gate arrays (FPGAs) provide an alternative to using serial processors. Continual advances in the size and functionality of FPGAs over recent years has resulted in an increasing interest in their use as implementation platforms for real-time video processing [2].

An FPGA consists of a matrix of logic blocks that are connected by a switching network. Both the logic blocks and the switching network are reprogrammable allowing application specific hardware to be constructed, while at the same time maintaining the ability to change the functionality of the system with ease. Performance gains are obtained from bypassing the fetch-decode-execute overhead of serial processors and by using the inherent parallelism of digital hardware to exploit concurrency within the algorithm. As a result, FPGAs may achieve the same computational throughput with operating clock frequencies of an order of magnitude lower than high-end serial processors, lowering power consumption.

In a previous paper [3] we provided an overview of the object tracking algorithm, focusing on low cost real-time signal processing on an FPGA. This paper outlines the tradeoffs and optimisations required for implementing the algorithm on a small FPGA. Design emphasis has been placed on minimising the logic and resource utilisation of the implementation, leaving resources for additional functionality that will use the tracking information in the desired application. Design decisions to reduce hardware requirements place a limit on the type of object tracking algorithms that can be implemented [4].

To demonstrate the usefulness of the tracking algorithm, we have implemented a simple interactive game that uses arm movements as input.

Section two discusses in detail the elements which make up the vision system and tracking algorithm. By considering the purpose of each step within the algorithm, optimisations and tradeoffs for FPGA implementation are discussed. Section three presents the resource utilisation of the final tracking system that was implemented.

2 Image processing system

Two major forms of parallelism exist in low-level image processing [5]: spatial parallelism, in which the image is divided in to multiple sections and processed concurrently, and temporal parallelism, where the algorithm may be represented as a time sequence of simple concurrent operations. Our algorithm exploits temporal parallelism by building a pipeline of concurrent operations that feed data from one operation to the next. This removes the need to have multiple access or copies of the image in memory and makes it possible to operate directly from the input stream provided by the video ADC. The stream-based processing also limits the type of algorithms that can be easily implemented [4].

Several object tracking techniques could be used in this application. Direct, motion based algorithms work on differences between successive frames. By detecting the differences between frames, the motion of an object may be inferred directly. Such motion based methods require frame buffering and were not considered in this application for that reason.

An alternative is to use a segmentation based approach, where the target objects are segmented from the rest of the scene in the captured image. The object is then tracked by considering the change of position in successive frames.

Segmentation partitions the captured image into several disjoint object regions based on common uniform feature characteristics [6]. One simple method is to segment the image based on colour by applying thresholds to each pixel. This is ideal for stream processing because thresholding is a point operation which can be implemented easily on the FPGA. Colour based segmentation requires care to be taken in the environmental setup as discussed in the next section. A block diagram of the complete system is shown below in Figure 1.

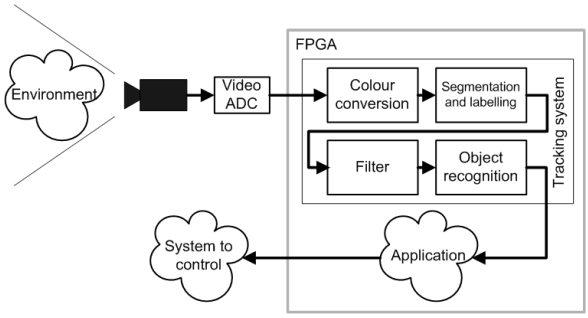

Figure 1: Block diagram of system

The tracking algorithm is broken into four stages: colour conversion, segmentation and region labelling, morphological filtering, and object detection. The pixel stream from the image capture sub-system is converted to a YUV colour space to remove the inherent interdependence between luminance and chrominance in the RGB colour space. Segmentation is performed using colour thresholding to associate each pixel with a particular colour class. A morphological filter is then used to remove any noise pixels that are not part of object regions. Finally, objects are detected by constructing a bounding box which encloses all of the pixels within a colour class. For each object, region parameters such as position, size, orientation and other useful information may be determined.

A strict timing constraint is imposed with pixels arriving at a rate of 13.5MHz, allowing approximately 74 ns per pixel to perform all four operations in Figure 1 . This inevitably requires applying both coarse-grain (between operations in the processing chain) and fine-grain pipelining (within operations) to ensure meeting of timing constraints.

2.1 Environment

Our aim is to work in a relatively unstructured environment. An unstructured environment (such as no blue/green screen) provides greater system flexibility and portability but can make reliable segmentation more difficult because of the need to distinguish the objects of interest from any other objects that may be present within the image. This limitation may be overcome by restricting the target objects to saturated and distinctive colours to enable them to be distinguished from the unstructured background.

Augmenting the unstructured environment with structured colour in this way is a compromise that enables a much simpler segmentation algorithm to be used. In the context of the game, the user is required to wear or hold fluorescent (highly saturated and intense) colour “markers”. Each marker has a different colour to allow the tracking algorithm to differentiate between them. Each object to be identified and tracked must have a unique uniform colour associated with it; this is called its colour class.

Detecting and segmenting purely by colour introduces a number of interacting variables that must be correctly adjusted. In addition to aperture adjustment of the camera and tuning of the colour thresholds for each target object, lighting within the environment also plays an important role. The colour of the lighting and placement within the environment has a dramatic affect on the reliability of the algorithm. Lights positioned within the camera field of view can introduce noise pixels into the image. Frontal spot lighting can cause the pixels of the target objects to saturate so that they appear white. Thus diffuse lighting is preferred.

2.2 Image capture

Image capture is performed using a video camera and ADC converter (decoder chip) which digitises the analogue signal from the camera into a stream of 16-

bit RGB (5:6:5) pixels. The stream is interlaced with successive fields providing the odd and even lines of the PAL frame.

The algorithm operates on each field of the interlaced frame. The effective image size is therefore reduced to 768 by 288 pixels. In our implementation the decrease in spatial resolution is justified because tracked objects are large in relation to the size of the frame. Processing each field independently increases the temporal sampling frequency to 50 samples per second. It also avoids "tearing" resulting from the rapid movement of objects between successive fields.

2.3 Colour space transformation

The simplest form of colour segmentation is to independently threshold the red, green and blue components of each pixel. Those pixels that are within all three ranges are classified as belonging to the corresponding colour class. Unfortunately, such segmentation is frequently unreliable because any change in the intensity of a colour results in a diagonal movement within RGB space, with a significant effect on all of the components [7]. To allow for this, the colour region has to be quite large with the significant likelihood of background pixels being misclassified into each colour class. This intensity interdependence problem may be overcome by transforming the image to another colour space such as HSI or YUV. Each of these separates the colour components from the intensity or luminance. The transformation from RGB to YUV colour space consists of a coordinate rotation to map the RGB cube onto the *Y*, *U* and *V* axes. Transforming from RGB to HSI involves mapping the RGB cube to a cone and is computationally more expensive, although it gives better intensity independence [8].

The YUV colour space is widely used in video and broadcasting [9]. The standard RGB to YUV transformation matrix involves several floating point multiplication operations which are computationally expensive for an FPGA implementation:

$$\begin{bmatrix} Y \\ U \\ V \end{bmatrix} = \begin{bmatrix} 0.299 & 0.587 & 0.114 \\ -0.169 & -0.331 & 0.500 \\ 0.500 & -0.419 & -0.081 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix} \quad (1)$$

A more efficient alternative is to replace equation (1) with the modified YUV colour space transform proposed in [7]. This removes the multiplications, replacing them with simple addition, subtraction and shift operations:

$$\begin{bmatrix} Y' \\ U' \\ V' \end{bmatrix} = \begin{bmatrix} \frac{1}{4} & \frac{1}{2} & \frac{1}{4} \\ \frac{1}{4} & -\frac{1}{2} & \frac{1}{4} \\ \frac{1}{2} & 0 & -\frac{1}{2} \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix} \quad (2)$$

Note that this transformation is only similar to the standard YUV transform in that it separates the luminance and chrominance components, and the two chrominance components are orthogonal. However in this application, we do not need true YUV, so this simplification is valid. Although the *R*, *G*, and *B* components have different precisions, as far as equations (1) and (2) are concerned, they can be treated as fractions between 0 and 1. Equation (2) will give 7 bits of precision for each component.

As equation (2) is a coordinate rotation, any change to the intensity will also affect the values of U' and V'. Therefore, to make the U' and V' less sensitive to illumination, we want to normalise them by the intensity, Y'. This caused problems because for the saturated colours chosen for the markers, the value of Y' can be lower than that of V'. This can shift the normalised values outside the range -1 to 1. This may be avoided by normalising by

$$Y'' = \max(R, G, B) \quad (3)$$

instead of Y'. Y'' would have 6 bits precision.

One consequence of the normalisation is that the decreased dependence on light intensity, comes at the cost of colour specificity. Normalisation has the tendency to detect all colours with similar hue. Thus, there is a trade off between intensity insensitivity and colour selectivity.

The FPGA implementation calculates Y'' from equation (3), and U' and V' from equation (2) in parallel for each pixel in the input stream. This makes it possible for the whole conversion to occur in one clock cycle.

2.4 Colour thresholding

For each colour class, the normalised components are independently thresholded using

$$Y'' > Y''_{\min},\ U'_{\min} < \frac{U'}{Y''} < U'_{\max} \text{ and } V'_{\min} < \frac{V'}{Y''} < V'_{\max} \quad (4)$$

Only a single threshold is required for Y'' because the target colours are bright. A pixel belongs to a colour class only if it is within all three Y'', U', and V' ranges, and is labelled with a unique ID corresponding to that particular class. As each colour class corresponds to an object being tracked, all pixels within the image are labelled as part of an object region or as part of the background. This reduces the raw pixel data to a unique ID that depends on the number of tracking objects defined. For N colour classes there will be $N+1$ (to include the background label) unique IDs.

Equation (4) includes a division operation which is costly to implement in hardware, using a large amount of FPGA resources and introducing long combinatorial delays if not pipelined [4]. The division can be removed algebraically by multiplying the U',

and V' components of equation (4) through by Y'' to give

$$Y''U'_{\min} < U' < Y''U'_{\max} \text{ and } Y''V'_{\min} < V' < Y''V'_{\max} \quad (5)$$

The resulting multiplications are less expensive than the divisions. However, as is shown in the next section, even these multiplications may be eliminated.

2.5 Lookup table optimisation

To perform segmentation and labelling of N colour classes, $4N$ multiplications and $5N$ comparisons must be made on each pixel after performing the colour transformation. For real-time operation, each colour class must have separate hardware because all N sets of comparisons must be performed per clock cycle. These operations can therefore consume significant resources on the FPGA if performed in parallel.

Due to the proliferation of small dedicated on-chip RAM resources on many of today's FPGAs (Xilinx calls this Block RAM), a lookup table (LUT) can be used to perform the thresholding, normalisation and labelling in a single step. This saves on fabric resource utilisation, combinatorial logic delays and pipeline latency.

The LUT method involves pre-calculating the result of equation (5) for all valid input values. On initialisation, the resulting values are loaded into local memory on the FPGA (in Block RAM). All of the tests of equations (4) and (5) could be performed in one step, requiring 2^{20} entries in the LUT table (6 address bits for Y'' and 7 each for U' and V'). However, since the U' and V' thresholds are independent, they may be separated, reducing lookup to 2 tables of 2^{13} entries each. Each entry would produce a single bit result indicating whether or not the colour is within the thresholded range.

Each Block RAM on the target FPGA has 2^{12} bits, therefore 4 Block RAMs would be required for each colour class. Again this is expensive in terms of resources. By reducing the precision of Y'', U' and V', a compromise can be made between precision and resource utilisation. As the RGB inputs only have 5 or 6 bits precision, reducing the precision of U' and V' to 6 bits is reasonable. The Y'' is used primarily for normalisation, and some normalisation will take place even when relatively low precision is used. Therefore the Y'' component was reduced to 3 bits. The net effect is that the $Y''U'$ and $Y''V'$ tables requires 2^9 bits each.

Multiple colours may be tested simultaneously because several tables in parallel would use the same address (the actual colour of the pixel being classified), but have 1 bit for each colour class being tested. This arrangement is illustrated in Figure 2.

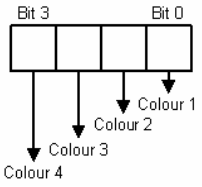

Figure 2: Representation within a LUT element

In our implementation, we test 4 colours simultaneously in this way. Both $Y''U'$ and $Y''V'$ for 4 colours may be combined into a single Block RAM. The RAM is dual-ported, allowing both tables to be accessed simultaneously. This configuration means the first half of the Block RAM stores the $Y''U'$ table and the second half stores $Y''V'$. The first bit of the index thus specifies the table to be accessed. The next 3 bits of the index correspond to the Y'' value and the final 6 bits are either the U' or V' value of the transformed pixel.

The two 4-bit colour class mask results are bit-wise ANDed together. If the result is 0000 the pixel does not belong to any of the colour classes and is considered as a background pixel. If there is a single 1 in any bit position, then the position of the 1 corresponds to the class of the object. The case of multiple ones is by definition invalid, because a pixel may belong to at most one class.

The LUT approach to colour classification works with any number of colour classes (subject to resource limitations), and has a constant processing time of 1 clock cycle.

2.6 Object recognition

A bounding box [9] provides a simple method for calculating the position, size and aspect ratio of a labelled region. With a little additional processing, the bounding box can also be employed to give the orientation of a non-symmetric object.

The bounding box is defined by the topmost, bottommost, leftmost and rightmost pixels belonging to a particular colour class. The method works by recording the co-ordinates of the first pixel of the labelled region in a raster scan. This gives the top of the box. Then with each successive line the x co-ordinate of any pixel in the labelled region is compared with the recorded co-ordinates of the current left (min) and right sides (max) of the box. These are updated if the detected pixel is outside the present bounds. Finally the last pixel in the labelled region is used to give the y co-ordinate for the bottom of the box.

Each target object and thus colour class has its own bounding box. Thus all bounding boxes must be calculated in parallel during the processing of the field. Separate hardware and a local data structure can be built to keep track of each bounding box. However, as each pixel is labelled as part of a single colour class

(overlapping classes are invalid) only a single bounding box is adjusted for each pixel. Therefore, the use of duplicated hardware is inefficient as only one instance will be used at a time.

Instead, we can share a single instance of the bounding box hardware. This is multiplexed between the data structures containing information for each bounding box, depending on the colour class. By configuring logic cells to act as on-chip memory, an array of bounding box structures may be created to store the required data, with the multiplexing performed implicitly though the address port. As the clock frequency of our design is twice that of the pixel data rate, the RAM can be single ported. This allows a read and comparison of the bounding box data with the pixel position on the first clock cycle, with the results to be written back to RAM on the second clock cycle.

After scanning through the entire field each bounding box will have been successfully constructed. Useful information about the bounding box can be extracted during the vertical blanking period and be used to control any application that can map this information to some useful function. For example:

- The centre of the bounding box will approximate the centre and thus position of the target object. A simple method for calculating this is to add the top and bottom and left and right coordinates, shifting each result one bit to the right to divide by two.
- The size is given by the width and the height (difference between the left and right, and the top and bottom). From this the distance from the camera to the object may be inferred.
- The aspect ratio is given as the ratio of the width to the height. The user can twist the colour markers to change the aspect ratio, and this may be used in a similar manner to a mouse click.
- The orientation of an elongated object may be determined by augmenting the bounding box. For each row within the image, if the leftmost boundary is adjusted an accumulator is decremented, and if the rightmost boundary is adjusted the accumulator is incremented. For a rod-like object at an angle, one side will be extended more frequently than the other and this is reflected in the sign of the accumulator.

The bounding box method of object recognition is sensitive to noise on the boundary pixels which can also introduce jitter into the detected position. These introduce an uncertainty or noise into the tracking parameters. More accurate information can be derived by smoothing tracking information over a series of fields, at the expense of additional logic resources.

2.7 Morphological filtering

After testing the initial algorithm, we found that even with adequate tuning there were isolated noise pixels associated with each colour class. Any stray pixel will cause the box to encompass the noise pixel, leading to the erroneous calculation of bounding boxes and consequently the derived tracking information. Pixels can be labelled incorrectly for several reasons including: objects of similar colour to the target in the environment, noise introduced by the image capture sub-system, specular reflections, and high contrast edges [10]. To remove these mislabelled pixels we employ a morphological erosion filter with a two by two structuring element.

This filter will remove any pixel labels that are not part of a group of pixels in a square four element window. This was found to be sufficient to remove most of the noise as long as the environment lighting, image capture sub-system (including aperture) and colour thresholds have been set up appropriately. Filtering cannot remove mislabelled pixels as a result of an object of similar colour to the target being within the captured environment.

The two by two filter is separable which means that filtering can be performed independently using a two element window in each of the horizontal and vertical directions. Thus the current pixel is first compared with the previous pixel (its left neighbour), by using a bitwise AND of the colour class masks. If the previous pixel belongs to the same class, it will be unchanged, but if the class is different, the result will be 0. The result is stored into the corresponding location in a line buffer. The result is also compared with the class label of the pixel from the previous line, obtained from the line buffer. This results in the filter using the bottom line from the previous frame when operating on the top line of the frame and the right most pixel from the previous line when operating on the left most pixel.

A single Block RAM is used for the line buffer, allowing a maximum line length of 1024 pixels (for a 4 bit colour class mask). Single-ported RAM is used, resulting in a two clock cycle pipeline to perform the filtering. In the first cycle the horizontal filter is applied, and the value from the previous row is read from the line buffer. The second is used to do the vertical filtering and write the horizontal filter result to the line buffer. The use of the colour class mask enables all classes to be filtered in parallel with simple bitwise AND operations in constant time.

In some instances this filter does not remove all noise; one way to reduce such noise would be to increase the window size of the morphological filter. Alternatively, a more complex region labelling sub-system could be implemented such as one based on connected component labelling. However both of these would come at the cost of increased on-chip memory and logic usage. We found that in most situations our system worked satisfactorily after careful adjustments to the lighting and adequate tuning.

3 Results

The resource utilisation for the tracking algorithm is shown below in Table 1. The target device is a XC2S200, a low cost and small sized FPGA. The resource requirements of each step were estimated by removing each part of the algorithm and calculating the difference in size that was given by the hardware generation tools.

Table 1: Resource utilisation of target device (Xilinx Spartan-II XC2S200)

	LUT RAMs	Logic LUT	Flip Flops	Block RAM
Video decoder		111	78	
Colour conversion		25	15	
Segmentation and region labelling		124	147	1
Morphological filter		5	12	1
Bounding box	39	76	123	
Calculating position and aspect ratio		19	10	
Total	39	360	385	2
Available	4704		4704	14

The implementation uses approximately 10% of the available logic resources of the target device. The logic used to implement the segmentation and region labelling appears high but thresholding itself accounts for only 5 logic LUTs and 4 flip flops with the remaining hardware needed to initialise the lookup table from off-chip flash memory.

The FPGA implementation operates at only 27 MHz with 90% area left for control applications. This meets our aim of a small footprint system, leaving sufficient room for applications using the tracking information to be implemented.

4 Summary

Object identification and tracking frequently requires the use of a real-time image processing system. Although real-time processing is achievable on serial processors, it can be beneficial to take advantage of the parallelism, low cost, and low power consumption offered by FPGAs.

To implement a compact design, it is necessary to balance algorithm optimisations and their effect on the target environment. All such optimisations rely on assumptions made about the environment and may restrict the operation of the system.

To reduce resource and logic utilisation the tracking algorithm uses several optimisations. The colour transformation has been simplified to remove the multiplications found in the standard YUV transform. The LUT-based segmentation and region labelling combine two costly operations into one step, eliminating the need for large numbers of parallel comparators. We have also made trade-offs in the type of algorithm used to implement this system, simple colour based segmentation, and the use of a bounding box over more complex connected component based region labelling.

The resulting optimised implementation can fit on a small and low-end FPGA, such as the Xilinx Spartan-II XC2S200, with sufficient resources still available for an application to make use of the derived tracking information. We have demonstrated this by designing a simple interactive arcade game where the user's movement of coloured markers are used as an input into the game.

5 Acknowledgements

The authors would like to acknowledge the Celoxica University Programme for generously providing the DK3 Design Suite.

6 References

[1] L. Baumela and D. Maravall, "Real-time target tracking," *Aerospace and Electronic Systems Magazine, IEEE*, vol. 10, no. 7, pp. 4-7, 1995.

[2] J. Villasenor and B. Hutchings, "The flexibility of configurable computing," *IEEE Signal Processing Magazine*, vol. 15, no. 5, pp. 67-84, 1998.

[3] C. T. Johnston, K. T. Gribbon, and D. G. Bailey, "FPGA based Remote Object Tracking for Real-time Control," to appear in *Proceedings of International Conference on Sensing Technology*, Palmerston North, New Zealand, pp. 2005.

[4] C. T. Johnston, K.T. Gribbon, and D. G. Bailey, "Implementing Image Processing Algorithms on FPGAs," *Proceedings of the Eleventh Electronics New Zealand Conference, ENZCon'04*, Palmerston North, pp. 118-123, November 2004.

[5] A. Downton and D. Crookes, "Parallel architectures for image processing," *Electronics & Communication Engineering Journal*, vol. 10, no. 3, pp. 139-151, 1998.

[6] K. R. Castleman, *Digital Image Processing*, 1 ed. New Jersey: Prentice-Hall, 1996.

[7] G. Sen Gupta and D. G. Bailey, "A new colour-space for efficient and robust segmentation," *Proceedings of Image and Vision conference New Zealand*, Akaroa, N.Z., pp. 315-320, Nov. 2004.

[8] A. Van Dam and J. D. Foley, *Fundamentals of Interactive Computer Graphics*. Reading, Massachusetts: Addison-Wesley, 1982.

[9] J. C. Russ, *The Image Processing Handbook*, Fourth ed. Boca Raton: CRC Press, 2002.

[10] K.T. Gribbon, D.G.Bailey, and C. T. Johnston, "Colour edge enhancement," *Proceedings of Image and Vision conference New Zealand*, Akaroa, N.Z. ,pp. 297-302, Nov. 2004.