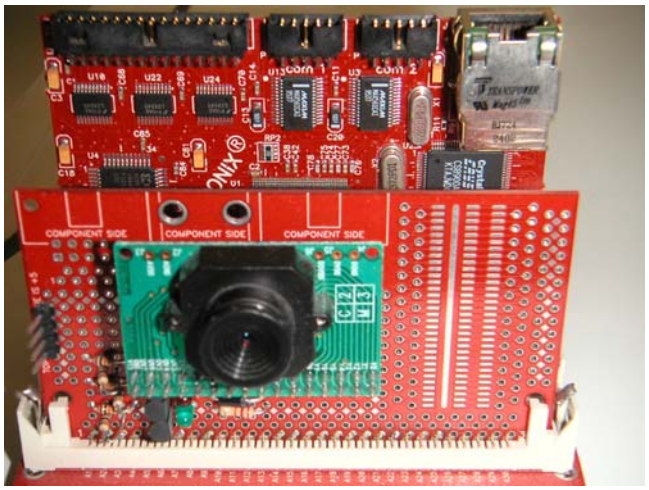
Programmable Java Camera

Report Java Camera Project 27-01-04



Students :
Reena Mahabir
9935584
rkmahabi@science.uva.nl

Sebastian Eigenmann
0061476
eigenman@science.uva.nl

Gerben de Vries
0033383
gkdvries@science.uva.nl

Tim van Oosterhout
0021490
tim.vanoosterhout@student.uva.nl

Roald Hopman
0017957
rhopman@science.uva.nl

Project Leader :
Peter van Lith
peter@lithp.nl

Teacher :
Arnoud Visser
arnoud@science.uva.nl

Abstract

Using a Java chip and a simple camera, a hardware camera can be created which is completely programmable in Java. This is beneficial because in many education and developer environments, Java is the language of choice. In this paper we will describe the software simulation of the hardware JavaCam and some algorithms we developed to demonstrate the usage of it.

Introduction

There is a growing interest in the application of small low-cost programmable camera systems for a variety of applications. Most camera systems depend on a frame grabber and software that runs on a PC. Developers of embedded applications are interested in using the same technology. Because the power of an embedded system is much lower than that of desktop or notebook PCs alternatives need to be found to allow a fair amount of processing power on such small systems.
Image processing usually involves computational intensive algorithms. They are therefore programmed in low-level languages. On first look the Java programming language isn't fit for this kind of application. The introduction of dedicated hardware Java chips however changes this.

When the project was started, there was a hardware prototype of the JavaCam. However, this prototype is still under development, so we had to work on the software simulation of the JavaCam. The simulation of the JavaCam uses a standard Webcam on a PC and a Java Framework to simulate the hardware.

The main goal of our project was to get a deeper understanding of the architecture, organization, operation and possible problem areas of developing image analysis software for a variety of applications.
To get familiar with the system, we first implemented some basis algorithms like Motion Detection en Edge Detection.
After we got acquainted with the software simulation we could implement more advanced algorithms and could extend and improve the existing system. Because of the short time we had for this project we succeeded only in implementing Optical Flow.

Program Structure

Our project was to develop image processing demo applications that were to run in a given framework. This framework was created to visualise the processed images and provide a way to test the algorithms without using the actual JavaCam. Instead, a local USB video capture device can be used as a video source. The framework was implemented in Java using the open source software development environment Eclipse, and used some custom graphics packages. This implementation was functional, but not well structured and contained some bugs. We decided to restructure the software without straying from the framework so that our new GUI and system would still meet the original criterion that the image processing tasks would be able to run unmodified on both a pc and the hardware JavaCam (Figure 1). This meant that we had to remember that certain parts should operate more or less independently, and communicate in a restricted manner. The hardware contains an FPGA (Field Programmable Gate Array) and a Java Chip that do not share the same memory, so we could not pass objects between the different parts, only bytes. Since the JavaCam is still in its development phase and moreover contains a programmable FPGA we still had some freedom in design and synchronisation. We tried to make the design more modular. For example, we created a Camera interface instead of a camera object, so that the video functionality could easily be extended to read other types of cameras as well. Other video sources might also be possible, such as a noise generator or and file reader. We also defined the roles of the individual objects more strict. The result is a more stable and flexible core and two versions of the GUI that can be used side by side.

Figure 1 – Hardware prototype of the JavaCam

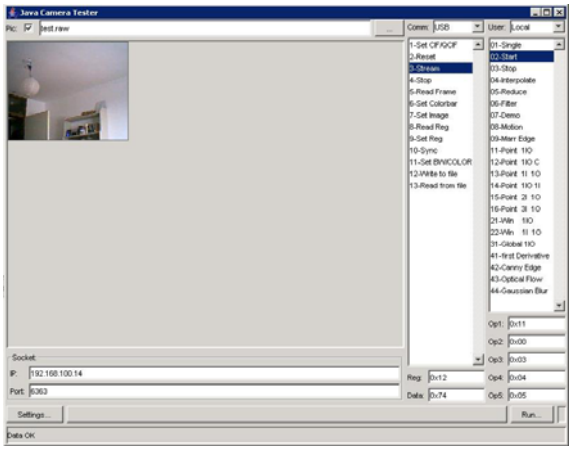


Figure 2a – Swing GUI Interface

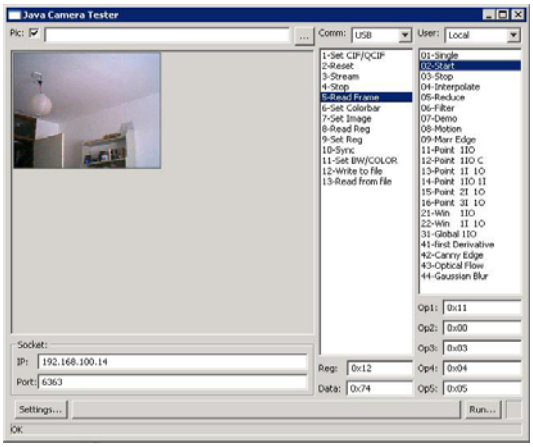


Figure 2b – Swt GUI Interface

The GUI

The original GUI was implemented using the *org.eclipse.swt* packages, which are provided with the Eclipse development environment. These packages provide a framework for developing applications that interact closely with the underlying operating system and hardware. To ensure stability, there are a number of restrictions on the operations you can perform on these objects. For instance, a thread that creates a Display object (*org.eclipse.swt.widgets.Display*) automatically becomes the user-interface thread. This thread is not allowed to make another Display object until it disposes the first one, and no thread is allowed to interact with the Display but the user-interface thread. This means synchronising the display with other threads such as timers or data collectors is disallowed. Extending a Display is also not allowed. Restrictions like these give us less freedom in our design.
A more pressing issue to move away from the Eclipse packages was that it seemed to contain bugs. One annoying bug is in the Scrollable object (*org.eclipse.swt.widgets.Scrollable*). We don't use this object directly, but an extension, namely a Canvas object. This object is responsible for drawing the processed camera images. The bug is that it keeps accumulating new buffer data without releasing the old data. As a result, the application may take up several hundred megabytes of memory if it is run for a while. Some other minor issues also made the program less stable.
One last reason to move away from the Eclipse packages and use the *javax.swing* package instead is that it is a standard package provided with most distributions of the java SDK and runtime environment. Therefore, apart from the necessary JMF package if you want to capture video, no additional packages need to be installed.
Moreover, developers could still use the Eclipse environment, but also have the choice to use their own familiar tools and they look quite the same (Figure 2a-2b).

The System

The underlying system has been reworked in an object oriented structure, with proper data hiding and a minimal amount of public functions. Most of the old code is still in the software, but some methods have been moved to other objects to make the structure more intuitive. Some pieces of code were removed, notably duplicate functions with different names, others were modified, for instance functions that directly access variables that are now hidden. The final product still takes into account the hardware design of the JavaCam, for instance only byte instructions are passed from and to the section that is to run in the FPGA. Below is a top-down description of the structure.

The GUI (*mainForm* or *mainFormSwing*) handles all the user interaction and drawing of the processed images. It maintains a *JPB* object, which models the Java Programmable Board. A virtual machine runs on the hardware board that can communicate with the pc over a socket. If a local camera is used, we need to communicate with that instead of handling the socket. To implement this functionality we created a *JPBvm* interface that is implemented by *SocketHandler* and *LocalJPBvm*.

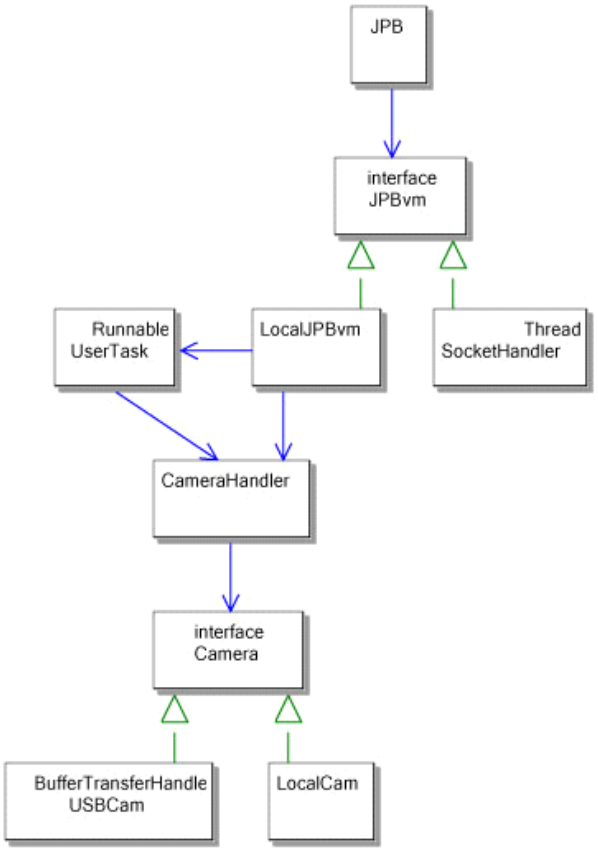


Figure 3 – UML diagram of the software JavaCam

The SocketHandler has not been properly implemented yet, because we have no means of testing it with an actual java camera. However, the byte instructions are defined in the interface and used by the LocalJPBvm as well, so that consistent behaviour should easily be achieved. The LocalJPBvm keeps a reference to a *CameraHandler*, which is responsible for maintaining several buffers for the algorithms to work on. An algorithm specifies a buffer it wants to use for input and which one to use for output. Optionally, a third buffer can be specified if the algorithm needs to store in between result. Because of this, algorithms can re-use certain operations and even complete algorithms could be chained. New camera images are fed in a pre defined buffer and another is used to retrieve processed images. So the starting and ending buffer are pre defined, but these buffers are available to use in any way the algorithm sees fit while it runs. The actual processing is done by setting the X and Y position and then calling a function that performs the actual pixel transfer and manipulation on a relevant block of pixels instead of manipulating an entire buffer.

In hardware, the CameraHandler runs on the FPGA and so has access to the working memory for the buffers and to the camera input. The local version of the CameraHandler has a reference to an object that implements the *Camera* interface. Through this interface, different types of camera can be accessed in a uniform way. At this point we have implemented a *LocalCam*, which generates random coloured noise, and a *USBCam*, which accesses a video capture device through the Java Media Framework. Another possible camera would be a camera model that reads video files. On the JavaCam, a special Camera would have to be created to access the JavaCam hardware.

The LocalJPBvm also maintains a *UserTask*, which is a thread that is designed to run asynchronously from the main application. A UserTask specifies the complete algorithms, by specifying per algorithm the aforementioned input and output buffers and calling the manipulation functions in CameraHandler. Which algorithm is executed gets passed to the UserTask on creation by the LocalJPBvm, which receives this instruction in the form of a byte specified in the JPBvm interface. The byte instructions are passed by the mainForm as an argument to a function call. For the remote version using SocketHandler, this function would, being unable to create the remote UserTask itself, send this byte over the socket to the JavaCam, which would receive it in its own JPBvm which would in turn then be able to create the UserTask. Theoretically, several UserTask could be running at the same time, but for them to perform useful tasks would require algorithms that would be specifically designed for this, since they operate on the same buffers. A UserTask always performs its algorithm only once, and it is the JPBvm's task to make sure that the UserTask runs every time a new frame is requested.
Finally, the USBCam handles a video capture device through use of the Java Media Framework. On creation the USBCam searches for an available camera and captures it. If it is not being used anymore its *dispose()* method is called to release the camera and make it available again to other applications. Once it has acquired a camera, USBCam registers itself to be notified anytime a new frame is available. It holds a reference to a frame, the contents of which are copied to a buffer, which will be returned when a new image is requested, so that the contents of the frame can be safely changed while maintaining the integrity of the returned buffer. In the process of copying, the contents of the frame are locked. On notification, USBCam waits for the frame to be unlocked (which it usually already is, since normally more frames become available than are requested) and updates it to contain the new frame, regardless whether a new frame is requested or not, so that it always contains the latest available frame.

Algorithms

The algorithms we implemented are Motion Detection, Edge Detection and Optical Flow. When dealing with image processing Convolution is necessary in most of the previous mentioned algorithms. In the following sections we will describe in detail the algorithms we implemented.

Convolution

Convolution is a function (usually) on a small coefficient scheme (kernel) and a larger coefficient scheme (image). A general convolution is described by equation 1.1.

$$(f * g)(i,j) = \sum_k \sum_l f(i-k, j-l) g(k,l)$$

Equation 1 - Convolution, from [5]

A convolution however leads to a problem: what to do with the edge pixels? You can choose to not do a convolution there. Or you can give the imaginary pixels (outside the

edge pixel), which you need, some value. We chose to not convolve edge pixels. Therefore our convoluted images generally have a black border.

The problem we have with doing the convolution is to create good kernels. Due to hardware demands our kernels need to be represented by bytes. But this means that most kernels would increase the intensity of an image, since they sum up to far more than one. So we need a good 'normalization factor' to divide the computed pixel value with. When we don't do this our computed images would end up with values far out of the range of 0 to 255 (due to the byte size). We can't normalize an image afterwards, because this would mean saving an image as integers (which can take far bigger values) first. This is not hardware friendly.
Our convolution algorithm has the option to either convolve with the RGB or the black and white version of the image. Black and white is however recommendable as this is faster and most algorithms are designed for black and white images.

Motion Detection
To calculate and visualize motion in a real time environment at least two consecutive frames are necessary. The basic idea behind motion detection is to compare two frames and visualize the difference. The first step in our approach was to average the three valued R, G and B images (with size M-by-N) of the color input into a one valued grayscale image (with size M-by-N).

$$I(x,y) = \sum_{x=1}^{M}\sum_{y=1}^{N}\frac{1}{3}(I(x_R, y_R) + I(x_G, y_G) + I(x_B, y_B))$$

Equation 2 - RGB to black and white taken from [5]

This was done to get a hold on all the information of the camera sensors (R, G and B) and to average out possible fluctuations in one of the three color channels. The second step in our motion detection algorithm is to subtract two consecutive grayscale frames I_1 and I_2 into a new frame I_{Dif} which contains the absolute values of the differences between I_1 and I_2.

$$I_{Dif}(x,y) = \sum_{x=1}^{M}\sum_{y=1}^{N} abs(I_2(x,y) - I_1(x,y))$$

Equation 3 - absolute difference between to frames, taken from [1]

The absolute values are taken because the difference can take on negative values depending on when in time the difference occurred. This information is irrelevant because we are interested in difference and not when it occurred.
The next step that is taken in our algorithm is to deal with noise. To achieve a robust algorithm that is stable against image noise. To this end we constructed a 3-by-3 averaging kernel which is convolved with I_{Dif} to I_{avgDif}.

$$avgKernel = \frac{1}{9}\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

Equation 4 - averaging kernel, taken from [2]

This new image is then compared per pixel against a threshold (which is found experimentally). Our algorithm contains at the moment to thresholds one for "lighter" motion and one for "heavier" motion. If a certain pixel in the I_{avgDif} image falls into the "lighter" motion threshold this pixel will be colored green and for the "heavier" motion it will be colored in red. If the pixel in question is below these thresholds it will get the original R, G and B values from image I_2.

$$I_{out}(x,y) = \begin{cases} green\ if\ 20 < I_{avgDif}(x,y) < 30 \\ red\ if\ I_{avgDif}(x,y) > 30 \\ I_2(x,y)\ if\ I_{avgDif} < 20 \end{cases}$$

Equation 5 - threshold function

Edge Detection

Edge detection comes in many flavors. In our project we chose to start with edge detection based on the first derivative of an image. The variant we implemented is known as Sobel's edge detector [7].
The Sobel edge detector uses two masks to calculate an approximation of the first derivative in the X-direction and in the Y-direction by convolving with those masks. After calculating these, the norm of the X-derivative and the Y-derivative is taken, this is called the magnitude. A magnitude above a certain threshold indicates an edge pixel.

$$s_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \quad s_y = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

Equation 6 - Sobel masks, taken from [7]

$$M = \sqrt{s_x(x,y)^2 + s_y(x,y)^2}$$

Equation 7 – Magnitude, taken from [7]

This edge detector however has problems locating the edges. To locate edges we need to take a look at more advanced edge detectors, who take into account the second order derivative of an image.

The first candidate for this is the Marr edge detector. This detector is based on the Laplacian of an image [5]. The Laplacian gives an approximation of the second order derivative of an image. It is however very sensitive to noise. Therefore it is a good idea to first blur your image using a Gaussian Kernel. Luckily this can even be done in one step, by convolving a Laplacian kernel with a Gaussian and then convolving the image with this kernel, which is known as the Laplacian of the Gaussian.

0	−1	−2	−1	0
−1	0	2	0	−1
−2	2	8	2	−2
−1	0	2	0	−1
0	−1	−2	−1	0

Equation 8 - Laplacian of Gaussian, taken from [8]

In the convolved image we have to look for the zero-crossings, this is where the second derivative changes sign. A zero crossing indicates a peak in the first derivative. This way we better localize the position of an edge. Finding a zero crossing is done by comparing a pixel *p* with each of its neighbors, including diagonals. If one of them is below zero and the other is above zero, then we have a possible zero crossing. We also need to check if *p* is closer to zero then the neighbor we found, if this is the case then *p* is a zero crossing. This makes sure that the edge we are going to find best approximates the real, continuous, edge. Next we look at *p* in the first derivative (calculated using for instance Sobel's mask's or Gaussians as with the Canny edge detector) to see if the magnitude of the gradient is above a certain threshold. This filters out weak edges, which are mostly due to noise.

A better edge detector is the Canny edge detector [5]. Canny's edge detector localizes the edges at the zero-crossings of the f_{ww}.

$$f_{ww} = f_x^2 f_{xx} + 2 f_x f_y f_{xy} + f_y^2 f_{yy}$$

Equation 9 - taken from [5]

The various *f*'s are calculated by convolving with derivatives of the Gaussian kernel. We need G_x, G_y, G_{xy}, G_{xx} and G_{yy}. For better performance, the entire image is first convolved with a normal *G*, this blurs it, and that way removes noise. The zero crossings are detected the same way as in the Marr edge detector.

$$G(x, y) = \sigma^2 e^{-\left(\frac{x^2+y^2}{2\sigma^2}\right)}$$

Equation 10 - two-dimensional Gaussian function, taken from [9]

Since the Canny and Marr edge detectors make use of Gaussian kernels we have the standard deviation σ parameter. A standard deviation of $\sigma=1$ detects pixel wide edges. A bigger σ would detect wider edges, so more global image features. A complete edge detector would use different values of σ on the same image and integrate these values into one edge detected image.

The implementation of Sobel's Edge Detector was rather straight forward. It is a matter of convolving every pixel with the two masks, and then calculates its magnitude. The selection of a good threshold is difficult. This depends on the camera and the lighting available.

Marr's Edge Detector is implemented by convolving with a 5 by 5 Laplacian of Gaussian Kernel, as given in equation 8. The zero crossings are calculated as described. However, the first derivative isn't used to check for a peak. We implement a threshold on the absolute difference between the pixel and the neighbor with which it has a zero crossing. This difference needs to be above a certain level, which is the case when the first derivative is big. So the behavior is essentially the same, it however saves us from computing the first derivative, which is computationally heavier. This way the Marr Edge detector is even faster then the Sobel edge detector.
For the canny edge detector we need to convolve with a lot of different Gaussian kernels. Fortunately all except one of them can be done one dimensionally. This saves on computation. The blurring 5 by 5 kernel G, can be split up in a horizontal and vertical kernel. G_x, G_y, G_{xx} and G_{yy} are all 5 by 1 or 1 by 5 kernels respectively. Only the G_{xy} kernel is 5 by 5. When we have all the different convolutions done on a pixel we can calculate the f_{ww} using equation 9. The same zero crossings detection is then used as for the Marr Edge Detector.
We have only computed edges with the Marr and Canny Edge Detectors using $\sigma=1$. Doing computations for different σ would be to computationally expensive. It is however very easy to expand the program with kernels with other σ.

Optical Flow
After implementing Motion Detection, we wanted to know the direction and speed of a pixel within a small time interval. This is what the Optical Flow algorithm does.
First, we assume that the color conservation of a pixel over a small time interval holds true. Color conservation means that when looked at a small time interval, the color of the pixel remains the same.

Normalized rgb introduces two independent quantities to represent color properties of a spectrum, for example rgb uses r and g (or any other pair of the r,g and b quantities). In the normalized rgb system, the two independent quantities representing color properties of a spectrum are defined as different ratios of the RGB quantities. The quantities we use are r and g, so the color conservation assumption implies

$$\frac{\partial r}{\partial x}u + \frac{\partial r}{\partial y}v + \frac{\partial r}{\partial t} = 0$$
$$\frac{\partial g}{\partial x}u + \frac{\partial g}{\partial y}v + \frac{\partial g}{\partial t} = 0,$$

Equation 11 – color conservation assumption, taken from [1]

where $\frac{\partial g}{\partial x}$, $\frac{\partial r}{\partial x}$ denote the derivative of the g and r components in the x direction and $\frac{\partial g}{\partial y}$, $\frac{\partial r}{\partial y}$ denote the derivative of the g and r components in the y direction. $\frac{\partial g}{\partial t}$, $\frac{\partial r}{\partial t}$ denote the derivative in time of the g and r components.

This system is well determined and its solution provides an estimate of the image flow where *u* represents the estimated image flow in the x direction and *v* the estimated image flow in the y direction.

First we have to normalize the RGB values into rgb using the following equations:

$$I_r(x,y) = \frac{I_R(x,y)}{I_R(x,y) + I_B(x,y) + I_G(x,y)},$$
$$I_g(x,y) = \frac{I_G(x,y)}{I_R(x,y) + I_B(x,y) + I_G(x,y)}$$

Equation 12 - RGB conversion to rgb, taken from [6]

The first derivative in the x, y and t direction can be calculated with the convolution using the following kernels:

$$x\_derivativeKernel = \begin{array}{|c|c|c|}\hline 1 & 2 & 1 \\ \hline 0 & \underline{0} & 0 \\ \hline -1 & -2 & -1 \\ \hline \end{array}$$

$$y\_derivativeKernel = \begin{array}{|c|c|c|}\hline 1 & 0 & -1 \\ \hline 2 & \underline{0} & -2 \\ \hline 1 & 0 & -1 \\ \hline \end{array}$$

$$t\_derivativeKernel = \frac{1}{9}\begin{array}{|c|c|c|}\hline 1 & 1 & 1 \\ \hline 1 & \underline{1} & 1 \\ \hline 1 & 1 & 1 \\ \hline \end{array}$$

Equation 13 - kernels, taken from [2]

To calculate the optical flow we need two consecutive frames with a small time interval.

$$I_{x\_derivative\_r}(x,y) = (I_{1\_r}(x,y) + I_{2\_r}(x,y)) * x\_derivativeKernel$$
$$I_{y\_derivative\_r}(x,y) = (I_{1\_r}(x,y) + I_{2\_r}(x,y)) * y\_derivativeKernel$$
$$I_{t\_derivative\_r}(x,y) = (Abs((I_{2\_r}(x,y) - I_{1\_r}(x,y))) * t\_derivativeKernel$$

Equation 14 – derivatives, taken from [2]

The same calculations are used for the derivative of the g component.
Now that we have the values of the derivatives in *x*,*y* and *t* direction, we can calculate *u* and *v* by solving the linear system of equation 11. *u* and *v* can now be used to visualize the optical flow.

Evaluation

Motion Detection
After having implemented the basic steps of the motion detection algorithm the thresholds had to be found manually by trial and error. Although it works by this method a better idea would be to have an automatically detection of the threshold. At the moment there are only two thresholds which can be extended into as many as are needed in the future. The next step could be to implement different algorithms and not only the one mentioned above. However, this simple algorithm is quite stable and robust against image and randomly introduced noise.

Figure 4 – motion detection

Edge detection
On first look the output of the simplest edge detection algorithm, the Sobel Edge Detector seems to be the best. All the edges we expect to see are there, if we use an adequate threshold. The big problem however is the width of the detected edges.
Detecting the precise location of edges is what the Marr and Canny Edge Detectors where designed for. Although a first comparison of their output with a Sobel Edge Detector seems to be in the favor of the latter. When we look at the edges determined by both Marr

and Canny we see that they are only one pixel wide, and in good alignment with the actual edges, this is what we need, if we want to do, for instance, Hough Transforms on this output. Looking at our example pictures (Figure 5a-5c) we see that each of the algorithms have more or less their own edge type that they can handle well.
The problem described earlier with our convolution function, the fact that the kernels are in bytes, is perhaps of influence on the quality of the Canny and Marr Edge Detectors. It is hard to make sure that the end result of the convolutions maps nicely between 0 and 255. This is however necessary to have good results. Tweaking the normalize factor parameter is the only solution.
Both the Marr and the Canny Edge Detector have a problem with noise. Selecting a good threshold is therefore paramount to successful edge detection. Making sure that the image is blurred helps as well. Both algorithms do this in their own way, as described earlier.
Comparing the Marr and Canny Edge Detectors is difficult. From the literature [5] it is known that the Canny Edge Detector should perform better in detecting corners. We tend to notice this as well. The Marr Edge Detector is however the clear computational winner, doing only one convolution instead of the five for the Canny Edge Detector.

Figure 5a - Sobel

Figure 5b - Marr

Figure 5c - Canny

Optical Flow

After we had implemented optical flow, the visualization was not very intuitive and mostly incorrect because of difficulties which are involved when visualizing the motion. We tried two kinds of visualization. One with arrows and one with a color coded system which gave every pixel a different color depending on the estimated image flow direction. Our assumption was that the color coded system would give better results, but as we can see in Figure 6b, you cannot see the object that moved. In the visualization with the arrows (Figure 6a), we can the object, but the arrows point in all directions.
Because of the short time and the extensive literature around the subject of optical flow we could not look for other algorithms for a better visualization. Although we think that the calculations are correct future work could be invested into better visualization (Blur and better thresholds).

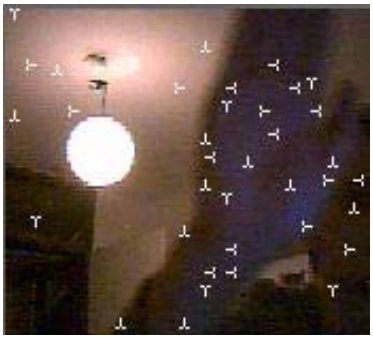


Figure 6a – optical flow with arrows

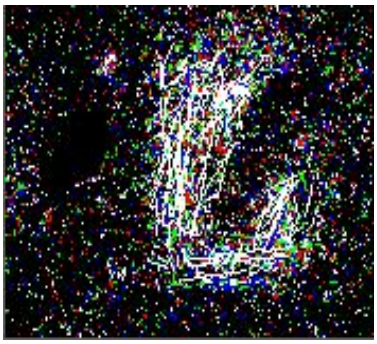


Figure 6b – optical flow by color

Running times (average over 100 runs) *

Algorithm	**Average time/run (ms)**	**Frame rate (fps)**
Motion Detection:	21	47,6
Optical Flow:	46	21,7
Sobel edge:	39	25,6
Marr edge:	46	21,7
Canny edge:	130	7,7
*Test -PC hardware: Cpu: Athlon XP2500+ Mem: 512 MB dual channel DDR (333 MHz) HDD: 120 GB WD 8MB cache ultra-ata/100, 7200 rpm Video: Club 3D 9600 pro 128 MB DDR		

Table 1 – Running Times

On first look these running times (Table 1) are good for this PC project. But they are too slow for the actual JavaCam, which runs at approximately 1/50 of the speed of an ordinary today's PC. This would mean that Motion Detection would run at a 1 fps frame rate.

One solution would be to have the algorithms work on averaged blocks of pixels to effectively reduce the resolution of the processed image. For motion this would probably be an adequate solution. However, averaging the pixels would destroy a lot of the edge information.

The Marr Edge Detector and the Motion Detector seem well suited to implement in low level instructions, i.e. the FPGA connected to the JavaCam. This is because they only involve relatively simple computations (addition, subtraction, multiply and division) on bytes and integers.

The Optical Flow algorithm makes use of floating point calculations, so this will require a bit of conversion. This is probably doable. For instance multiply with 100 and clip the decimals.

Both the Sobel and Canny Edge Detector use a square root in their computation. This is more difficult to implement fast. However, exact values for the square root are not necessary, so perhaps it is not as much a problem as we think.

Suggestions

Other methods/algorithms that could be implemented are for example: People or Object Counting, Object Recognition, Object Tracking, Line Follower, Distance Estimation, Particle Filters, Aggression Detection.
We also suggest Face Recognition and Pedestrian Detection. The first is relevant for security systems. Pedestrian Detection can be build into cars.
Aggression Detection is often tried with image recognition alone. This is proven to be very difficult. Therefore we suggest using sound as well. By combining visual and auditory input we create better conditions for detecting aggression. Loud yelling combined with speedy motion by some skin-colored object is perhaps a way to detect aggression. More microphones and cameras would allow to check whether the location of the sound and the image match up.

Conclusion

The software simulation of the JavaCam is a good way to develop image processing algorithms in general and more specifically for the hardware Camera. Because in general, programming in Java is easier, development is faster and less expensive.
The implementation of different algorithms has succeeded according to our expectation although the running times indicate that on the hardware camera not very high frames will be achievable. The resulting application is robust and extendible. Implementing new algorithms will not be difficult. Lots can still be done, but the basis is there.

References

[1] Golland, P. and Bruckstein (1997), A.M., Motion from Color, *to appear in*: *CVGIP, Computer Vision and Image Understanding*

[2] Baltes, J. (2003), Optical Flow Algorithms, *Lecture notes, www.cs.uminatoba.ca/~jacky*

[3] Hedlund, M., Jonsson, F. & Alberg, D. (2002), SMD151 – Multimedia Systems, lab #1 motion detector, *lab course report*

[4] Trucco, A. and Verri, A. (1998), Introductory Techniques for 3-d Computer Vision, *Prentice Hall*

[5] Van den Boomgaard, R. & Dorst, L. (2001), Machine Vision an introduction for computer scientists, *Syllabus for Machine Perception Course Universiteit van Amsterdam*

[6] Gevers, Th. (2003), Multimedia Information Retrieval, *Syllabus for Multimedia Information Retrieval Course Universiteit van Amsterdam*

[7] http://www.ii.metu.edu.tr/~ion528/demo/lectures/6/1/index.html

[8] http://www.efg2.com/Lab/Library/UseNet/1999/0929b.txt

[9] http://www.cpsc.ucalgary.ca/Research/vision/501/edgedetect.pdf

[10]http://www.sgi.com/software/opengl/advanced97/notes/node152.html#SECTION0001435000000000000000

[11] http://www.dai.ed.ac.uk/HIPR2/log.htm

Project url's :

Our project website: http://kreng7.homelinux.org/cgi-bin/wiki/wiki.pl
Project course website: http://www.science.uva.nl/~arnoud/education/DOAS/

Detailed assignment:
http://www.science.uva.nl/~arnoud/education/DOAS/Project2004/CameraTeamProject.doc

Installation Documentation of the JavaCam software:
http://kreng7.homelinux.org/~gerben/InstallationJavaCam.doc