**Hardware-software object tracking system using a moving camera**

**Michał Daniłowicz**

**michaldanilow@gmail.com**

**Submitted for the 2019 Digilent Design Contest Europe**

**03.05.2019**

**Advisor: Tomasz Kryjak, PhD**

**AGH University of Science and Technology**

**Kraków, Poland**

**{Photo of the project}**

**{Photo of the team members}**

(These can be combined, so long as everything is clearly identifiable)

# Introduction

Tracking problem is finding the tracked object in each frame of video sequence. Object’s velocity can also be calculated. Video tracking system can be used in many applications:

* Security – tracking vehicles and people in video surveillance systems,
* Entertainment – player controls in VR games via body movements,
* Autonomous vehicles – park assist, road signs and pedestrians tracking,
* Military – scout drones, aim assist

**Abstract**

Vision tracking algorithm was implemented in programmable logic of Zynq SoC. Used IP camera is mounted on moving head with pan/tilt position adjustments. The system controls camera’s position via TCP connection so that the tracked object is positioned in the center of each video frame.

**Objectives**

* Controlling camera’s position with HTTP requests,
* Implementation of skin color tracking algorithm in FPGA,
* Sending HTTP requests via Zybo processing system,
* Connecting FPGA with processing system via AXI Lite interface,
* Implementation of KLT tracking algorithm,
* Implementation of pyramidal version of KLT algorithm

**Project Summary**

The project can be divided into two main parts, vision tracking algorithm implemented in FPGA and camera control system implemented in processing system of Zybo (ARM microcontroller). The camera’s response time is short enough for tracking moving objects like moving people or body parts. Implemented KLT algorithm tracks one point of video stream. The design can be used in many applications, as the camera can be controlled via internet connection.

**Digilent Products Required**

Zybo Z7-20 development board.

**Tools Required**

IP camera with HTTP pan/tilt position control,

Vivado/SDK environment,

Ethernet switch and cables

**Design Status**

Point tracking was successfully implemented, but for robust tracking, Harris corner detector should be implemented for finding best points candidates. Point clustering should also be implemented in order to track actual object, not just points in video sequence.

# Background

**Why This Project?**

Insert text here.

**Reference Material**

Pyramidal Implementation of the Lucas Kanade Feature Tracker Description of the algorithm - <https://web.stanford.edu/class/cs231m/references/pyr-lucas-kanade-feature-tracker-bouget.pdf>

## Design

**Features and Specifications**

Insert text here.

**Design Overview**

Project’s block diagram is shown on Figure 1. The processing system is responsible for sending requests on TCP port 80 (HTTP) that control camera’s pan/tilt position and for that purpose, Xilinx TCP client project example was used. The network code was extended by features needed for the design, like manual reconnection with TCP server, manual disconnection and HTTP request generation. Two P-type regulators were also implemented in the processing system, one for “x” axis and one for “y” axis. The programmable logic (FPGA) implements video tracking algorithm and uses HDMI port as input. Communication between ARM and FPGA is done via AXI4 interface.

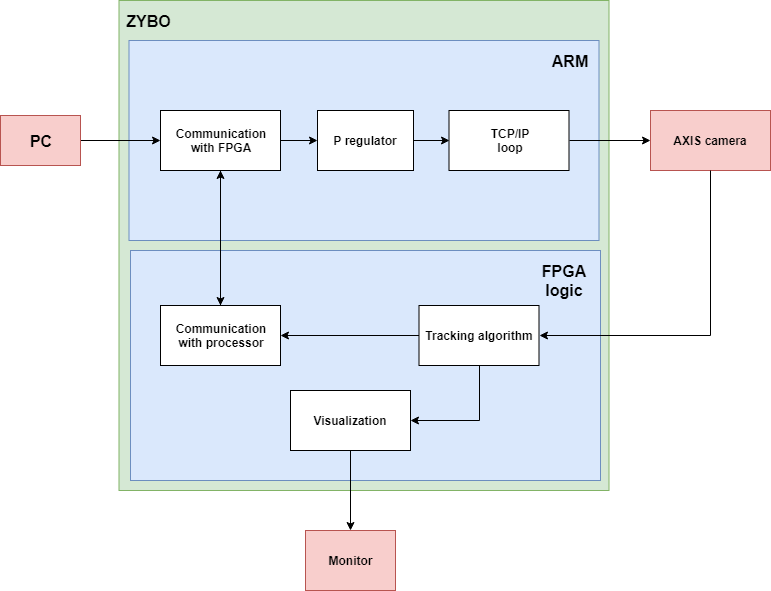


Figure 1. System diagram

**Detailed Design Description**

In following design description, we consider pixel coordinate as row index and pixel coordinate as column index, starting at [1, 1] for the upper left corner pixel. We will also use coordinates for a pixel, and in that case , .

**Controlling camera position**

Camera’s server accepts HTTP requests with following structure:

<camera\_ip\_address>/axis-cgi/com/ptz.cgi?<parameter>=<value>

For example, to set x-axis velocity to 40% and y-axis velocity to -20% of a camera with IP address 192.168.1.7, the following request should be sent:

192.168.1.7/axis-cgi/com/ptz.cgi?continuouspantiltmove=40,-20

**Skin color tracking**

For testing camera’s response time, a relatively simple skin color tracking algorithm was implemented in programmable logic of the SoC. General block diagram is shown on Figure 2.

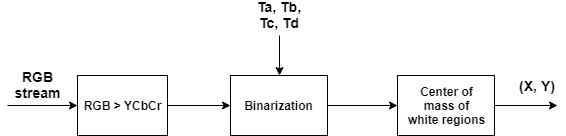
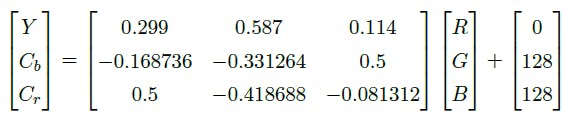


Figure 2. Skin color tracking

Video processing begins with conversion to color space. Input RGB pixel is converted as follows:



Skin color can be defined as some rectangle in two-dimensional space:

255 pixel value is considered a skin color pixel and values are found experimentally. Notice that we are not taking component (brightness) into account, because we want this classification to be independent (to some extent) from lighting. After classification, center coordinates of skin regions can be calculated:

Where:

is class value for pixel coordinates.

The most interesting part of skin color tracking implementation in FPGA is calculating coordinates of skin color center of mass. The general block diagram of the module that does the work is shown on Figure 3. The module’s name in project files is .

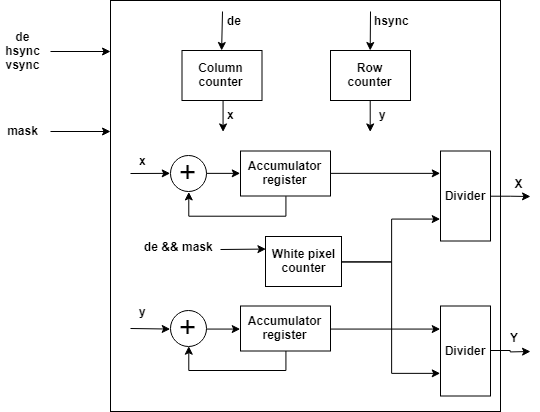
****

Figure 3. Centroid module

**KLT tracking algorithm (klt\_tracker.v)**

The KLT tracker uses grayscale frame of the video sequence and a general idea of the algorithm is to find 2x2 matrix and 2-dimensional vector :

Where:

* stands for and means pixels of the tracked area of the video frame.
* are directional image derivatives.
* difference in given pixel brightness between current and previous frame.

and together define linear equations system:

The solution is disposition of the tracked region relative to the previous frame. For more information about KLT algorithm and its more advanced, pyramidal version, please refer to given reference material.

Block diagram of FPGA implementation of the algorithm is shown on Figure 4.

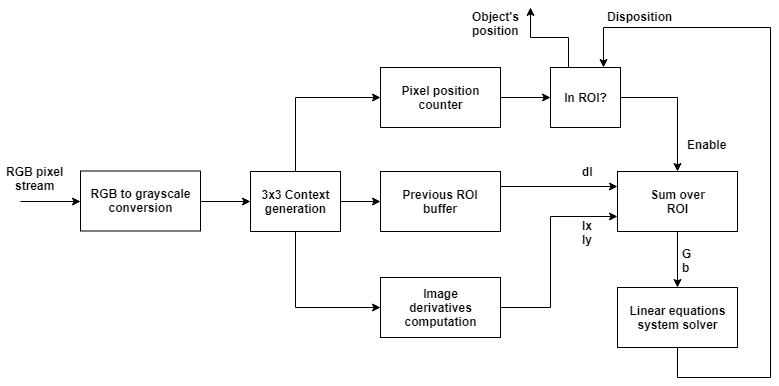
****

Figure 4. KLT implementation block diagram

**RGB to grayscale conversion**

For RGB to grayscale conversion, component from converter was used from skin color tracking algorithm.

**3x3 context generation (context\_3x3.v)**

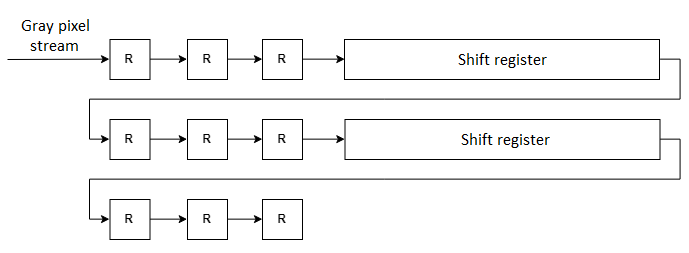
****

Figure 5. Context generation module

In order to compute image derivative at point , pixel neighbours are required and for that purpose, this module was created. Each register has 11-bit length to store 8-bit brightness, as well as synchronization flags for that pixel (de, hsync, vsync). Shift register was created with one port block RAM memory (BRAM) and a simple address controller. Write depths of those shift registers are equal to , where is the number of clock cycles in each horizontal line of the video stream (for 1080p image, ).

**Previous ROI buffer (point\_tracker.v)**

This module is responsible for storing ROI from previous frame and read a pixel from it at the moment when it is needed by module. Notice that due to ROI position change from frame to frame, a larger region (called extended ROI in this project) must be written to the memory. Figure 6 shows this relationship.

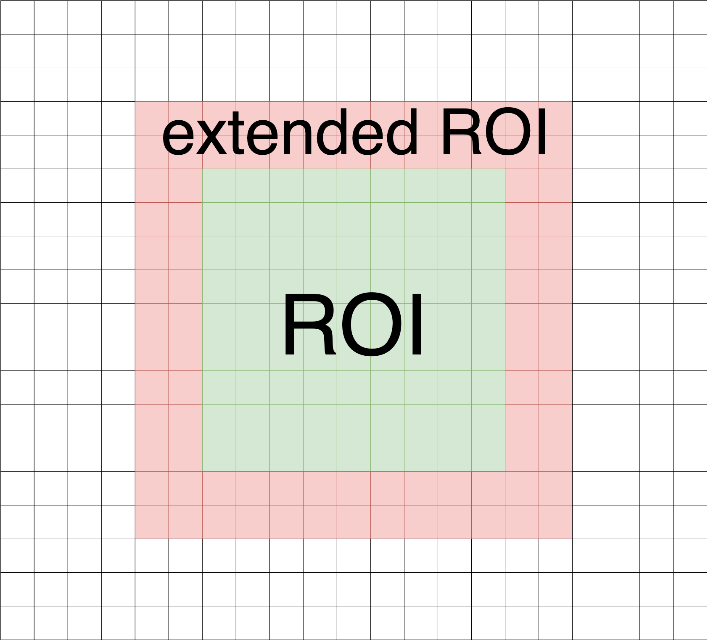
****

Figure 6. Extended ROI

Figure 7 shows general block diagram of the module. Signal is given by module and is set when center pixel of the 3x3 context is in extended ROI. Write address generator is basically clock cycle counter that is counting when is set. Notice that in the case of stationary ROI, read address equal to write address would be enough to properly read pixel from previous frame. Unfortunately, that is not the case and read address should be modified:

The last problem shows up when ROI changes position in y-axis. In that case, memory cells containing pixels from previous frame can be overwritten with current frame pixels too early (before previous frame pixels are needed). To avoid that, two BRAMs with multiplexed write flags were used. When current pixels are written to one BRAM, previous frame pixels are safely read from another and roles are exchanged after each frame.

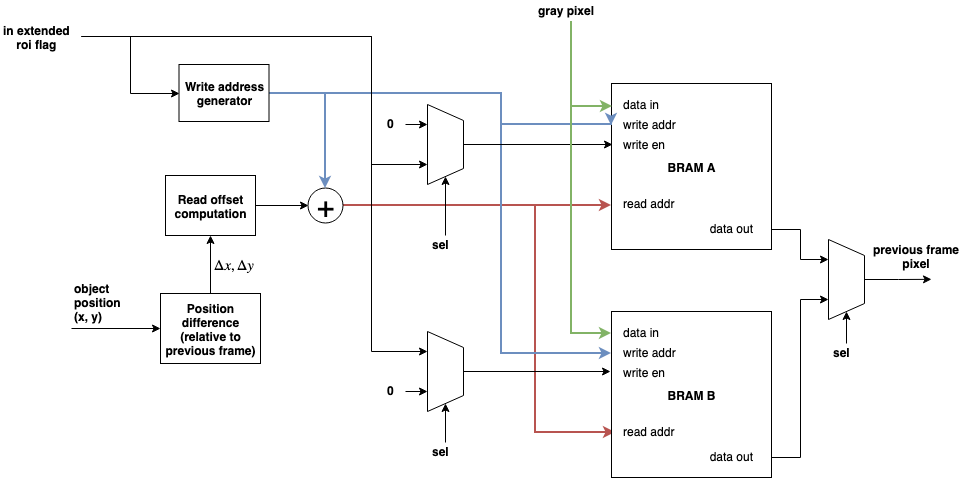
****

Figure 7. Previous ROI buffer diagram

**Linear equations system solver (linsolve.v)**

The system is defined by matrix and vector:

The solution is calculated in implemented design using Cramer’s rule:

**Pyramidal KLT tracking**

Standard KLT tracking algorithm works only for relatively small object disposition (about 1-2 pixels). For faster moving objects, a pyramidal version of this algorithm is proposed. The idea is to create scaled down representations of original video frame. Level 0 is the original frame, level 1 is scaled version with two times shorter width and height and level 2 is scaled down level 1 image by the same factor. On each level, starting from level 2 (smallest image), standard KLT algorithm is applied, but the solution (called pyramidal guess) is propagated to lower levels as initial starting point for the algorithm. On level 0, final disposition is calculated and object position is updated.

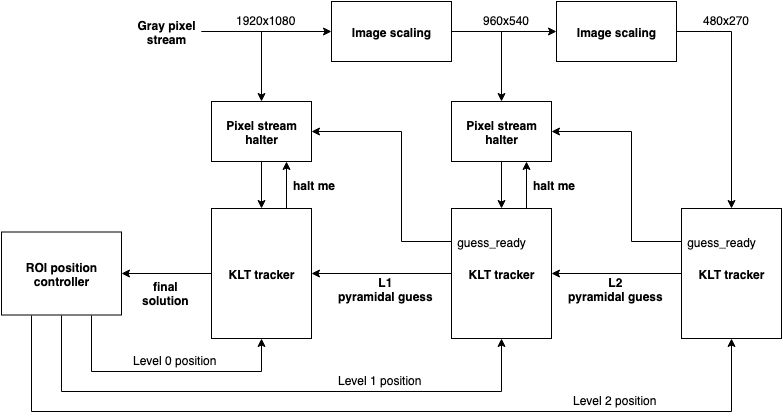
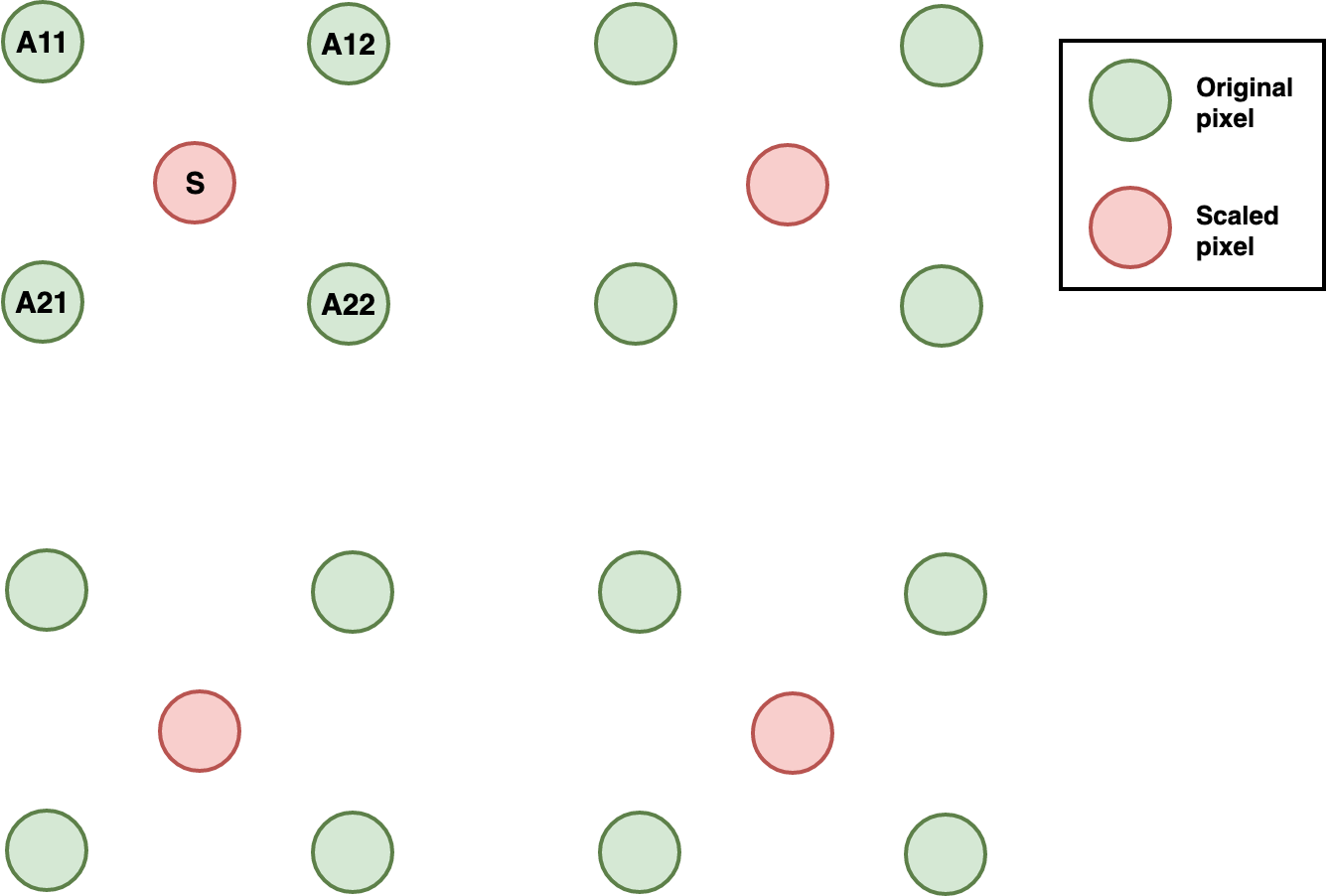
****

Figure 8. Top level diagram of pyramidal tracker

**Image scaling (scale2x.v)**

In general case, bilinear or bicubic interpolation is used for image scaling, but the problem drastically simplifies if the image is scaled by a factor of two.

****

In this project, scaled pixel value is just arithmetic mean of surrounding pixel neighbours.

The main challenge of this module is to generate valid output synchronization signals (i.e flags). On Figure 9, general block design of this module is shown.

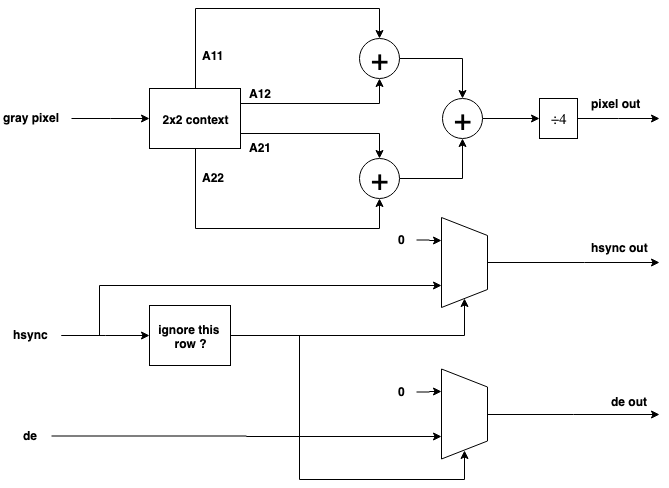
****

Figure 9. Image scaling module

Pixel neighbours are generated with 2x2 context module, which is essentially smaller version of 3x3 context module. Notice that every second context in a video horizontal line should be ignored and this is achieved by generating two times lower output clock frequency. Output horizontal synchronization signal (hsync out) and pixel valid flag (de out) are generated by ignoring corresponding input flags in every odd line of input pixel stream.

**Pixel stream halter (vision\_stream\_halter.v)**

The main problem of pyramidal tracker implementation is due to the fact, that lower level modules get to the ROI faster than higher levels, but computations must be started at the highest level. To solve this problem, was implemented. The module feeds unchanged pixel stream to the KLT tracker until the stream is just before the ROI, at which moment, the stream is being written to FIFO memory. Pixel transfer is halted from KLT tracker perspective until higher level of the pyramid confirms that pyramidal guess is ready and pixel data starts being read from the FIFO.

## Discussion

**Problems Encountered**

* TCP server in used IP camera ignored every HTTP request after successfully processing the first one. This was solved by manually reconnecting with the camera after sending each request.
* During early hardware tests, second video output of PC was used. Even though the display driver stated that output video signal was 1280x720 and the algorithm was parameterized for that resolution, the tracking did not work correctly. Horizontal line clock cycles counter was implemented and it yielded unexpected results. The actual line length was 2200 clock cycles, as in 1920x1080 resolution.
* Fixed point divider was used in module. Many hours were lost forever due to the fact that real number radix in Vivado waveform window was not displaying Q59.29 fixed point number correctly. Tests with MATLAB Fixed-Point Designer showed that the results were actually valid.

**Engineering Resources Used**

About 350 hours.

## References

AXIS V5915 PTZ Network Camera. AXIS communications

<https://www.axis.com/files/datasheet/ds_v5915_t10136145_en_1904.pdf>

VAPIX Pan Tilt Zoom API. AXIS communications

<https://www.axis.com/files/manuals/vapix_ptz_52933_en_1307.pdf>

Zynq-7000 SoC Data Sheet. Xilinx

<https://www.xilinx.com/support/documentation/data_sheets/ds190-Zynq-7000-Overview.pdf>

Pyramidal Implementation of the Lucas Kanade Feature Tracker Description

of the algorithm

<https://web.stanford.edu/class/cs231m/references/pyr-lucas-kanade-feature-tracker-bouget.pdf>