Motionamp
A Spatiotemporal Video Amplifier

Akashnil Dutta, Rishi Patel, Pranav Kaundinya

I. Introduction

Visualizing small changes that are often indiscernible to the human eye is an interesting challenge in video processing. Given that modern cameras have the ability to detect minute motions, processing video to amplify these motions can give significant insight. In order to understand what types of information we can extract we must first define what we mean by video amplification; there are two main classes of video amplification in this paper. The first is temporal amplification which amplifies the changes in time at each pixel of an input video. For example, in a video, if a dim light gets brighter by only a small amount, the amplified output will show a much brighter light. We distinguish this from spatial amplification, which exaggerates the translational motion of objects in a video. However, in reality, both of these types of amplification go hand in hand - temporal amplification is one component of spatial amplification.

Spatiotemporal video amplification has a wide range of applications. For instance, in biological imaging of cells, it enables scientists to visualize tiny movements as seen through a microscope. Other interesting applications include medical imaging, experimental physics and even non-scientific applications such as sports. With intensity amplification, one can see the changes in blood-flow as amplified changes in facial color and measure heart-rate. In physics experiments, spatiotemporal amplification can allow weak and barely noticeable events to be detected and analyzed. In real-time sports analysis, it is possible to magnify tiny motions of a ball or object hitting a target.

This project implements a real-time video-processing system that enhances small changes in videos. We decided to implement this system in hardware (on an FPGA) because real-time video processing is computationally intensive and often impossible in software. Our system implements discrete-time equations to obtain and amplify the effect of translational and temporal changes on intensity. We demonstrate spatial and temporal amplification on the same system, so that the user can switch between the amplification type according to his or her needs.

Our report is organized as follows: We begin with an explanation of the theory behind our project and equations we implemented, highlighting how temporal changes coupled with knowledge of spatial changes can let us amplify motion. Next, we present software simulations from MATLAB that we used to verify our theoretical framework. We then describe the hardware implementation followed by some of the challenges we faced, and possible extensions.

II. Theory

In this section we'll discuss the theoritical basis of amplitude magnification. The main idea is to compute the rate of change of intensity or the position with respect to time and add it back to the original signal multiplied by a suitable gain factor.

Assume that the video input is represented as a continuously differentiable ($\mathbf{C}^1$) function

$$I : [0, X] \times [0, Y] \times [0, T] \to V$$

such that $I(x, y, t)$ represents the intensity of pixel (x, y) at time t. V can be either a scalar field for a grayscale video or in general a vector space with RGB components for a colored video. We want to obtain a function $J(x, y, t)$ from I which amplifies the small changes in I with respect to time.

First consider the problem of magnitude amplification. In this case, we can amplify the temporal variation of each point independently. We just add a gain of μ to rate of change of I with respect to time.

$$J(x, y, t) = I(x, y, t) + \mu \frac{\partial I}{\partial t}(x, y, t)$$

To extend this to motion amplification, we will first consider I in an implicit form, $f(x, y, z, t) = I(x, y, t) - z$. The video stream is precisely the space of solutions of the equation $f = 0$. In general we want to map each point (x, y) and magnitude z to another point and magnitude (x', y') and a corresponding magnitude z' so that

$$x' = x + \lambda \frac{dx}{dt}, y' = y + \lambda \frac{dy}{dt}, z' = z + \mu \frac{dz}{dt}$$

Here, λ is the translational gain factor. The derivatives are obtained by implicit differentiation of the constraint relation $f(x, y, z, t) = 0$, i.e.

$$\left.\frac{dx}{dt}\right|_{f=0} = -\frac{\partial f/\partial t}{\partial f/\partial x}$$

etc. Let the partial derivatives $\frac{\partial I}{\partial x}, \frac{\partial I}{\partial y}, \frac{\partial I}{\partial t}$ by I_x, I_y, I_z respectively. We can simplify the above as

$$\frac{dx}{dt} = -\frac{I_t}{I_x}, \frac{dy}{dt} = -\frac{I_t}{I_y}, \frac{dz}{dt} = I_t$$

Hence we get

$$\begin{aligned} x' &= x - \lambda I_t/I_x \\ y' &= y - \lambda I_t/I_y \\ z' &= z + \mu I_t \end{aligned}$$

To produce an output video from this mapping, we need to find the intensity z' of an arbitrary point (x', y') in the proposed image. To do that, we first find pre-image point (x, y) in the original image from (x', y') of the proposed image, then take $z = I(x, y, t)$ and use the forward mapping to get proposed magnitude z' from z.

Then we get that the inverse of (x', y') is approximately $(x' + \lambda I_t/I_x, y' + \lambda I_t/I_y)$.[1] The corresponding z for (x, y) is then given by $I(x' + \lambda I_t/I_x, y' + \lambda I_t/I_y)$. The image of this under the map is $z + \mu I_t$. Hence the final function is given by

$$J(x, y, t) =$$

$$I\left(x + \lambda \frac{\partial I/\partial t}{\partial I/\partial x}, y + \lambda \frac{\partial I/\partial t}{\partial I/\partial y}, t\right) + \mu \frac{\partial I}{\partial t} \quad (1)$$

In the discrete domain this formula is equivalent to

$$\begin{aligned} J(x, y) = I(x + \lambda &\frac{I(x, y, t) - I(x, y, t-k)}{I(x, y, t) - I(x-k, y, t)}, \\ y + \lambda &\frac{I(x, y, t) - I(x, y, t-k)}{I(x, y, t) - I(x, y-k, t)}, t) \\ + \mu &\frac{I(x, y, t) - I(x, y, t-k)}{k} \end{aligned} \quad (2)$$

Here k is a parameter which specifies the window length for calculating the derivative. Its value depends on the noise characteristics. It is also possible to correct for noise with multiple data points using Richardson Extrapolation for instance.

[1] Here we are making an assumption that I_x, I_y, I_t values are the same at (x', y', t') as (x, y, t). This assumption does not work well for large λ, but makes the processing simpler in our implementation. However, it is not necessary to assume this for the theoritical treatment. We can invert the mapping exactly to do this as well, at the cost of a more complex hardware.

III. Prototyping

Before designing our hardware implementation, we needed to verify our ideas with software prototyping. First we verified that our temporal amplification algorithm would amplify changes in intensity in an input video. Our test case consisted of a video of a computer desktop background rapidly changing intensity after a few seconds (from a darker to lighter shade of gray). To verify that the output is as we expect, we played our input and output (processed) videos together on the computer screen. While the unprocessed video showed a barely discernible change of shading (from dark to light gray), the amplified output showed a saturated transition. That is, the region on the screen where the change occurred saturated to white (maximum pixel value) thereby making the change much easier to see. The implementation consisted of taking differences of input pixel intensities at two different time steps, multiplying this difference by a constant gain, and adding to the current pixel intensity. One can see in the plots shown in figure 2 that the intensity step is magnified in the output image, converging back to original with time. This verified that as expected, temporal amplification would work in principle.

Likewise, we tested our spatial amplification algorithm (this time on synthetic data instead of real video data). Here, we generated a moving rectangle with smooth edges (by applying a blurring function that replaced the value of each pixel with the average of intensities of those around it). Our code calculated the quantity $\frac{\partial x}{\partial t}$ for each pixel by taking the ratio of $\frac{\partial I}{\partial t}$ to $\frac{\partial I}{\partial x}$. We replaced each point in the input image, by a translated point $I(x - \lambda\frac{\partial x}{\partial t}, y - \lambda\frac{\partial x}{\partial t}, t)$ where the lambda is our amplification factor. We discovered that the output rectangle had extra apparent displacement in the direction of motion at each time-step. The dimensions of the rectangle were conserved. Interestingly, the amplification was not a linear function of the gain lambda, and we noticed that blurring was extremely important for acceptable results. The higher the blurring in the input image, the larger the maximum possible amplified output; this tradeoff is due to an approximation in our theory. Ultimately, seeing an amplified translation output in the simulation code gave us confidence to move forward with the hardware implementation.

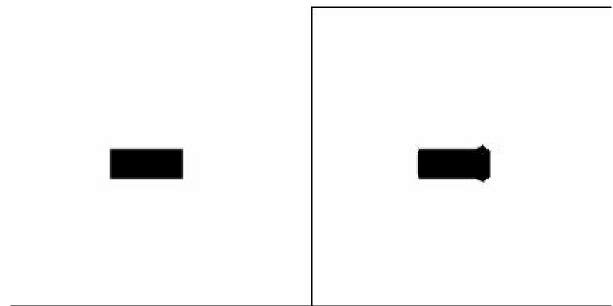

Figure 1: *Spatial Amplification Results on Software Testing.*

IV. Implementation

The two most important impacts of our approach to this problem on implementation are the requirement to store multiple frames for processing, and the fact that each pixel needs to individually compute its amplified intensity. Processing the video in real time put heavy constraints on timing and memory, as discussed in the section on challenges. Keeping this in mind, our hardware design targets computational efficiency. It is also flexible and modular, so that the design can be easily implemented with better hardware (for example, it can be easily modified to work with different memory).

The hardware implementation consists of an input stage, a spatial amplifier, a temporal amplifier, and the output stage. Our overall system architecture works by taking input from a camera, storing video frames in memory, and processing the frames for output. We store three frames in a circular frame buffer - two that are used for processing, and one which is being written into. Our frame buffer has been implemented on Block RAM within the FPGA. The processing modules access this buffer, and the output frame is written to a ZBT RAM. The output unit displays the frame in the ZBT RAM on a monitor. The various modules in our implementation are proposed in more detail below.

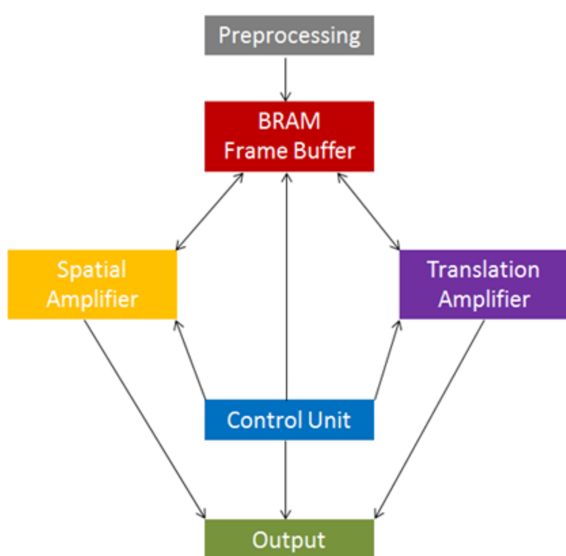

Figure 2: *Overview*

4.1 Input and Storage

4.1.1 Preprocessing

This stage consists of the video decoder and the YCrCb to RGB modules which produce the raw NTSC data.

4.1.2 NTSC to bram

This module writes incoming image data to the internal BRAM on the FPGA. Each group of four pixels is assigned an address, the higher-order bits of which signify the frame-number. The frame number is incremented on the start of signalframe which signals when an entire frame had been written to memory (ie. when both even and odd fields from the ntsc camera input have been written). The BRAM address is 16 bits long, the first 2 bits store the frame number. The next 8 store the y coordinate (a number between 0 and 255) and the final 6 store the higher order bits of the x coordinate (bits 7 through 2). The lower order bits of x are ignored because the data-words are 32 bits long, enough to store four pixels before asserting a write enable. While the data is shifted to the output register (taking 4 cycles), the address of x remains constant. Once three frames have been written, the writing overwrites previous frames and cycles through again (that is, although framenum is a two bit value it is forced to zero if it exceeds 2). The addressing scheme we use is such that we store a cropped image of only 255 by 255 pixels. The cropping was achieved by truncating the higher order bits of x and y in the address, originally intended for a 1028 x 764 image, and using the higher order bits to control the we signal. This was required to ensure that we could store at least three frames.

4.2 Frame Buffer

4.2.1 Circular Frame Buffer

Here we store the last 2 image frames for processing. This uses the block RAM on the FPGA.

The initial steps in working with the camera interface involved storing multiple frames to the ZBT memory. This required using the higher-order address bit to label our address with the frame number, and disabling writes to memory upon pressing a pausebutton. We were able to disable writing, and then select between the two frames with a frame-select button that toggled between two high-order bits in the read address generated by vram_display. We realized however, that storing input frames in BRAM would be much more convenient for our processing. Unlike ZBT, BRAM is a dual port memory that returns data with single clock cycle read and write latency.

4.3 Temporal Amplifier

This module is responsible for amplifying changes in intensity over time. This module is instantiated in the top-level module and is independent. It contains two submodules - the dI_dt_amp module and a get_point module. The dI_dt_amp module requests the pixel intensity values at particular times and outputs control signals, which are fed into get_point, the module which actually interfaces with the frame buffer and obtains the required intensity values.

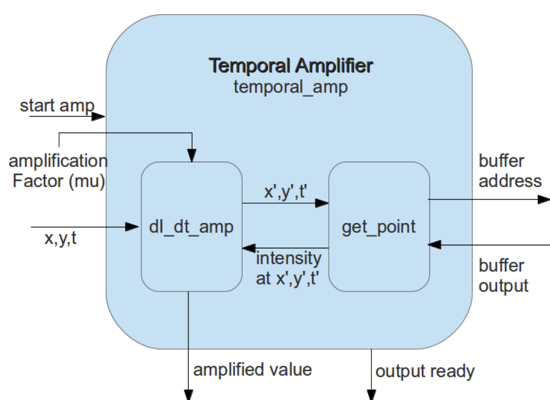

Figure 3: *Temporal Amplification module*

4.3.1 dI_dt_amp

This module actually computes the amplified intensity value at a given point. It starts its calculation on receiving a start signal from its parent module. It also accepts the (x, y) pixel coordinates at which it needs to calculate the amplified value. This module requests the intensity at the given coordinate in the current and previous frame. Each memory access takes a single cycle and the value from the get_point module is read into a register when the got_value signal arrives from the get_point module. Then, the current intensity is subtracted from the old intensity, multiplied by the amplification factor mu, and added to the old intensity. Since pixel intensity must be between 0 and 255, the final value obtained is capped to be within this range. All of this computation is done in combinational logic, so the module just waits for some clock cycles and then signals got_intensity.

4.4 Spatial Amplifier

This module is responsible for amplifying motion in the video. As explained in the theory section, the change in the position of a pixel from one frame to the next is not readily available. Our entire algorithm is local in nature, and each pixel individually computes its amplified intensity. We do not estimate the motion of an object in the video. Therefore, we use the change in intensity of a given pixel through time and the change in intensity of pixels across a single frame and divide the two to obtain the change in the position of a pixel from one frame to the next (ie. $\frac{ds}{dt} = \frac{dI/dt}{dI/dx)}$.

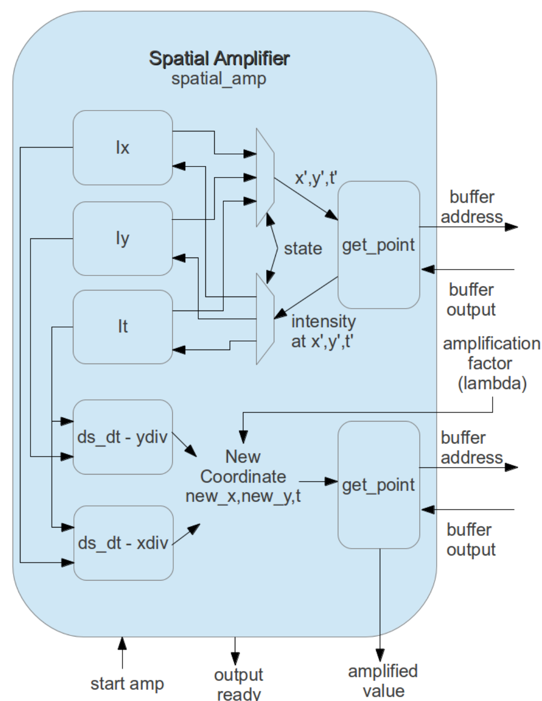

Figure 4: *Spatial Amplification module*

This module implements an FSM which controls memory accesses and coordinates the flow of data between the modules. The states can be divided into 3 main parts - states 0-2 involve obtaining the intensity derivatives with respect to x,y and t, ie. I_x, I_y and It. State 3 involves dividing the time derivative (I_t) by the spatial derivatives (I_x and I_y) to obtain the position derivatives ($\frac{dx}{dt}$ and $\frac{dy}{dt}$). Then we compute the new coordinate (new_x,new_y), whose intensity in the input frame, needs to be reproduced at the current point (x,y) on the output frame. State 4 involves obtaining the intensity at this new point and outputting it. The spatial amplifier contains two get_point modules. One of them is used by the I_x, I_y and I_t modules to obtain intensities at desired coordinates. The other is used to obtain the intensity value at the new coordinate calculated. Since each state has only a single module accessing the memory, there are no contention issues. The state

decides which of the submodules is connected to the get_point module.

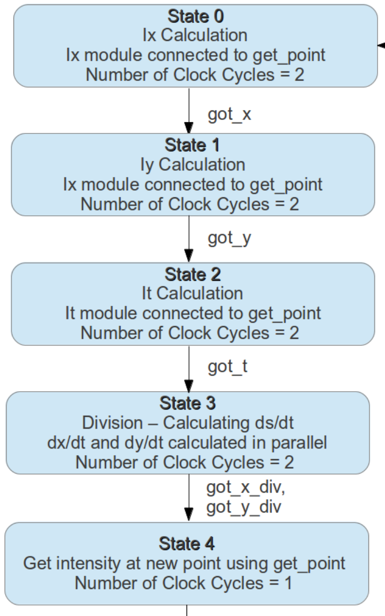

Figure 5: *State Transition Diagram*

4.4.1 I_x, I_y and I_t

These three modules and dI_dt_amp are very similar. Like the dI_dt_amp module, these three modules request intensity values at two different points. The difference is that the final value returned is just the difference of the two values obtained. In the Ix module, the points between which the intensity difference is calculated is (x, y, t) and $(x-1, y, t)$. Similarly for Iy, the points are (x, y, t) and $(x, y, t-1)$. For It, the points are (x, y, t) and $(x, y, t-1)$. Like the dI_dt_amp module, the calculation is done in combinational logic, and the module waits for 2 cycles before signaling got_intensity.

4.4.2 ds_dt

This module acts as a wrapper module for the divider, providing the control signals for the divider. Since the divider is implemented using combinational logic, this module holds the inputs of the divider steady for two clock cycles, and then returns the output of the divider, signaling a got_quotient.

4.4.2.1 divider

We needed integer division to calculate the rate of change of position with respect to time, which we use to compute the shift in x-coordinate and y-coordinate for spatial amplification. We considered several ways to implement this divider. We considered using a standard pipelined 8-bit 8-bit divider, but opted to implement our own because we had a specific purpose for the divider, and our requirements only required small outputs, since the shifts are supposed to be small. So we did not need a general purpose 8-bit 8-bit divider. Instead we focused on optimizing it for the required purpose. Our first implementation required a variable number of clock cycles per division. On the q-th cycle we checked whether $0 \leq$ numerator - denominator $\cdot q <$ denominator. the output quotient q is asserted whenever the condition is met. Since there is no control on the worst case time consumed, we introduced a cap on the output. But the timing constraints forced this cap to be too small and we did not get desirable results.

Next, we made a asynchronous divider with combinational logic which takes the numerator and denominator as input, and outputs a quotient between 0 and 15. We implemented this in a binary search procedure to get the right value of q. This makes sure the propagation delay is only the delay of $\log 16 = 4$ multiplications. We only multiply variables with parameters, so they are mapped to lookup tables by the compiler. The net propagation delay was found to be 14ns, which is slightly less than one cycle of the system clock. The details of its implementation is available in the code.

4.5 Control Unit

4.5.1 next_point

This module coordinates the processing with the input and the display units. When the input unit signals that a frame is ready (frame ready), it provides a start signal to the amplifier (start amp), along with the first coordinate to be processed (0,0). Once

the amplifier signals output_ready, it increments the point to be processed and again signals start_amp. It continues this until all the coordinates have been processed. Then, it waits till the next frame_ready signal, and repeats the process.

4.5.2 get_point

This module is used by both the temporal and spatial amplifiers and acts as an interface between the calculation modules and the frame buffer. It accepts the (x, y, t) coordinates of the point at which the intensity is desired. It uses an address_mapper to obtain the buffer address of the desired coordinate. Then, it outputs this address to the frame buffer. Since the frame buffer is a BRAM on the FPGA, the output is available after a single clock cycle. The output from the buffer is again fed into the address mapper, which parses the output and and returns the desired intensity value. The get_point module then outputs this intensity value. This module takes 1 clock cycle to output the desired value, since 1 clock cycle is required for the memory access.

4.5.2.1 address_mapper

This module abstracts away the memory from the rest of the modules. It maps (x,y,t) coordinates to addresses which can be used to obtain the intensity at the coordinate. It also parses the output of the buffer and extracts the desired intensity value. This makes our design flexible and scalable, because changing the size or type of memory or changing the addressing scheme only involves changing the mapping in this module. Other modules would be unaffected by changing the memory structure.

4.6 Output

The general goal of the camera/display modules is to store input frames in memory for processing, and to store the resulting output in the zbt RAM for display. We stored three frames in our input frame buffer, two frames for performing our computations, and one frame for writing the incoming image-stream. (This scheme avoids the memory contention issue which would arise with a two frame buffer). Our output frame buffers store one frame each. The overall dataflow of our image storage is shown in the figure below. We utilize one BRAM and two ZBT ram banks, for processed and unprocessed outputs.

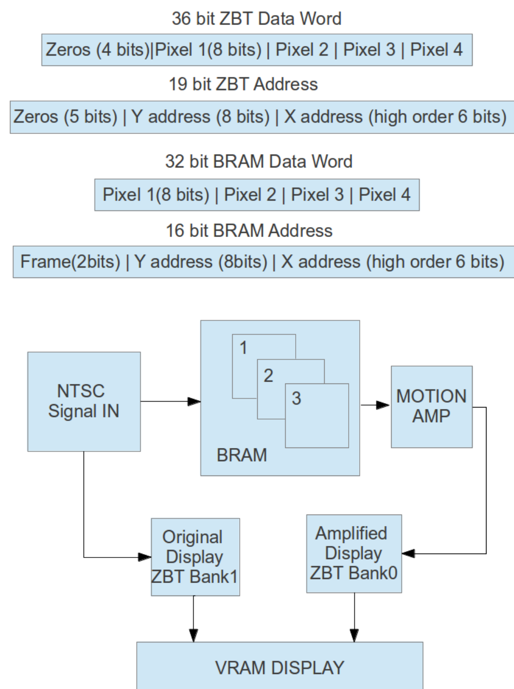

Figure 6: *Display Modules*

4.6.1 Display Buffer

This is a ZBT RAM bank that serves as a single frame buffer for constructing the output image on the VGA display.

4.6.2 display_zbt

This module writes our processed image data to the first ZBT rambank. ZBT requires a 19 bit address, so we padded the higher order bits with 5 zeros followed by the 8 bit y address and bits 7:2 of the x address. As new data from the amplifier became available (on the output_ready signal) we shifted each 8 bit pixel into into our output data register. When 4 pixels have been accumulated a write enable is asserted, and the output data register is padded with 4 leading zeros (to create a 36 bit data word).

4.6.3 ntsc_to_zbt

We wanted to have a live, unprocessed display for our user-interface. This requires writing the incoming ntsc signal from the camera to a zbt ram bank. The data addressing scheme here is identical to ntsc_to_bram. This makes it straightforward to switch between the two displays.

4.6.4 vram_display

This module reads data from our ZBT ram banks, and displays an image output on the VGA display. The module generates a 19 bit read address that accesses ZBT. To avoid displaying multiple copies of the stored image, we checked to see if the high-order bits of y and x were equal to appropriate constants.

4.7 Timing

One of the central points that we had to keep in mind throughout the processing implementation was cycles per pixel. Since we are doing real time video processing, we needed to make sure the following constraints are met -

cycles per pixel * number of pixels * system clock period * frame rate < 1

Because of memory limitations we were forced to reduce number of pixels from the maximum to resolution of the camera to a $256\times$ 256 field. We had to keep in mind that the system clock was 65 mhz, about double of the camera clock frequency, which is 27mhz. We calculated that we could afford 28 cycles per pixel on average. This was not any issue in the temporal amplification processing, but we found this a limitation in case of spatial processing. Timing is one of the reasons we opted to use the bram rather than the zbt for processing buffer, since bram has a lower latency than the zbt.

The bottlenecks in our processing stage were:

1. Ram latency
2. Divider Propagation delay
3. Gain multiplication

Our strategy for utilizing the maximum ram throughput was to use the computation for I_x, I_y, I_t in parallel while queueing on the read_enable signal at the get_point module. Parts of the code was pipelined and we successively saved a few cycles from the initial implementation.

4.8 User Interface

Our system has a very simple and minimalistic user interface. It consists of the NTSC video camera, a set of buttons and switches on the Labkit, a 16 character LED display, and a monitor. The LED display shows the mode that the system is in - either temporal or spatial. It also displays the current value of the amplification factor. The mode can be changed by flipping a switch, and the amplification factor can be changed using two buttons (for increasing and decreasing the value). There is also a reset button, which resets the system. The monitor displays both the original, unprocessed video, and the amplified video next to each other. This allows the user to compare the videos and distinctly observe the amplification.

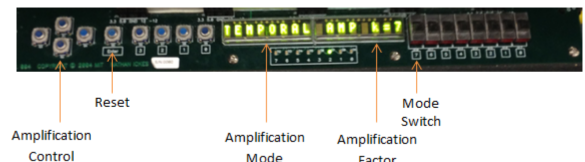

Figure 7: *User Interface*

V. Challenges

The main challenge we faced in our project was limited memory. We had only 3MBits of memory to store in the bram, which forced us to crop the frames to a 256×256 region. Although the processing would have worked very easily with color video, there was not enough room to store 3x amount of data for RGB pixels.

The second problem was the noise from camera input that always got in the way of computing the delta, because actual difference in intensity was often indistinguishable from fluctuations due to noise. This was particularly a problem in the division step of the spatial amplification, because of Catastrophic Precision Loss during the subtraction. We used a threshold value to prevent this, but it meant that

the processing was slightly insensitive to small gradients of intensity.

The noise problem could have been mitigated to a good extent if we had enough capacity to store more than 3 frames, then we would have taken the delta between pixels separated by more than one time step and more than one coordinate position to increase the precision.

Thirdly, we found some systematic imperfections in the output video which lined up vertically in infrequent but irregular fashion. We tested all parts of our implementation individually and made sure that correct data is written to the output_zbt. The problem seems to be occur reading from the zbt. We suspect that this is caused by the clock skew, since the zbt is located further away from the clock generator.

VI. Related Work

We were motivated to take up this problem for our project by the work of Freeman et. al. from MIT CSAIL (*Eulerian Video Magnification for Revealing Subtle Changes in the World*)

VII. Summary

We have implemented a real-time hardware digital processing system that amplifies temporal and spatial changes in videos. Our temporal amplifier shows promising results, and has a very clear amplified output. Our spatial amplifier shows translational amplification but requires further work on noise reduction and thresholding.

VIII. Future work

We were primarily limited by the image quality of our camera and memory constraints. An interesting future direction given more memory would be to allow frequency selective amplification. Here for example, we look only at a frame one time step in the past for our time derivative calculations. If we looked further back instead we would be sensitive to lower frequency changes. By making this lookback-time a parameter, it would thus be possible to look at and amplify frequency selective changes. Other future directions include enabling color display, which is again only limited by memory; the processing details remain exactly the same.

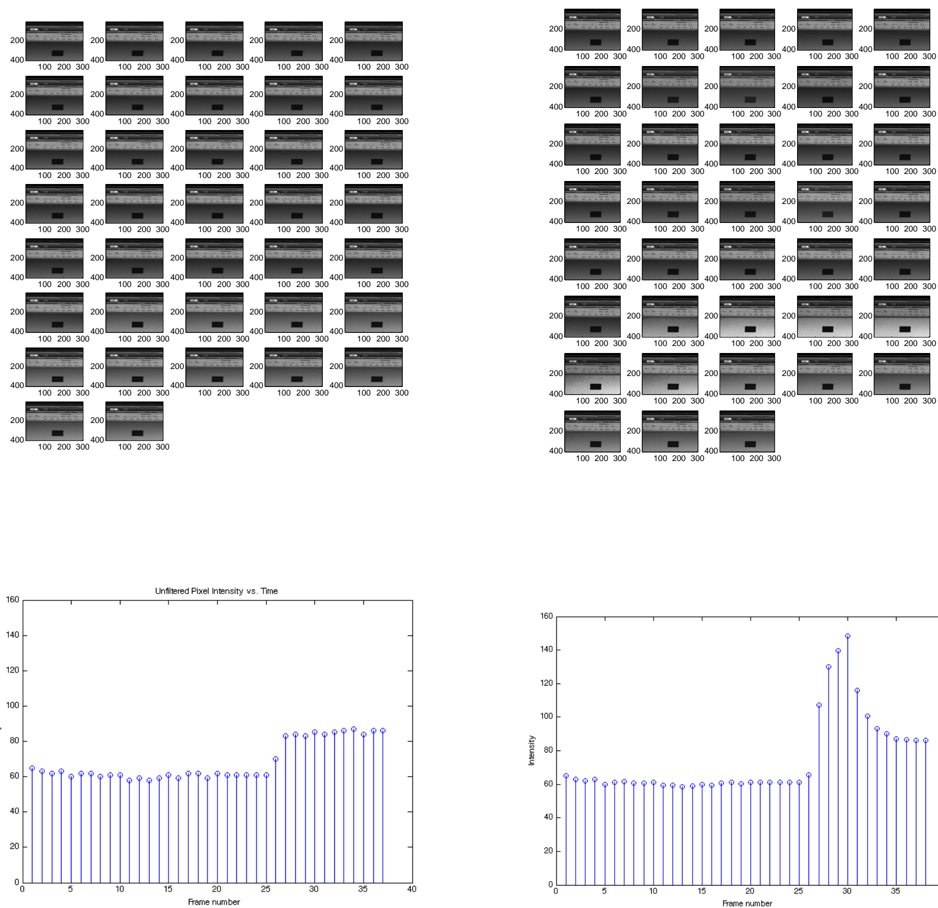

Figure 8: *In the top two figures the results from our matlab experiment are shown on a greyscale video stream. The left set of frames is the original video with small change in brightness. The signal is temporarily magnified in the corresponding frames of the right panel. The plots in the bottom depict the variation in intensity of a sample pixel.*

BK PRECISION
VOLT.
AMP
OUTPUT
POWER
VOLTAGE
CURRENT
Tektronix
TDS 2024
DELL
6.111 Labkit

DELL