Computer Vision Pipeline For Object Recognition

Kevin Zhang and Felipe Hofmann
December 13th, 2017

Abstract

We propose to implement a computer vision pipeline for object recognition. This pipeline will read input from a camera, detect and recognize objects, and display an annotated image on the VGA monitor. The basic user experience is as follows: the user presses a button and a box appears on the screen - the user holds an arbitrary object in front of the camera so that it shows up in the box. When the button is released, the FPGA learns to recognize the object and highlights it on the screen.

Motivation

State of the art computer vision systems achieve above-human levels of accuracy on object recognition tasks such as ImageNet, Cifar, and CoCo. However, state-of-the-art models such as ResNet are virtually impossible to run in realtime, and even the least resource intensive models which still achieve human level accuracy can consume >375 watts, which translates to roughly 40 joules per frame processed.

We want to see how far we can get with a cheap, low-resolution camera - without expensive depth sensors or shortcuts such as infrared LED tags - and a Virtex-II FPGA which consumes <50 watts yet achieves a significantly higher frame rate, coming out to an average of less than 1 joule per frame processed.

Design

We plan to attach a CCX-Z11 camera to the labkit through the s-video port, decode the video signal with the onboard ADV7185 decoder chip, process the frames in real time, and write to a VGA monitor with the onboard ADV7125 video encoder. The high-level block diagram for our implementation is shown below.[1]

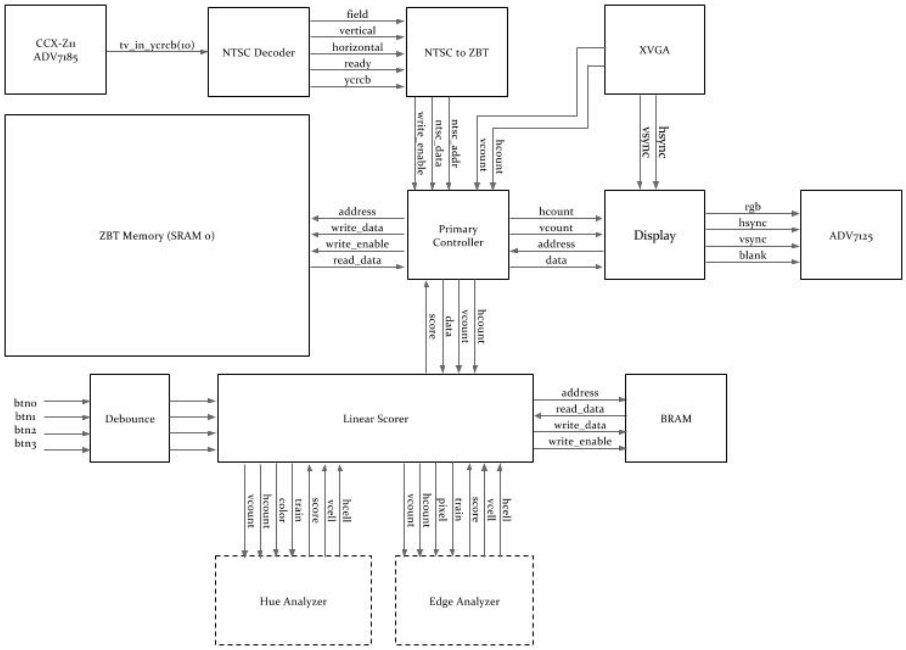


Figure 1: High-level block diagram showing the camera-to-vga pipeline.

Due to the limited memory size of the BRAMs, we will be using one of the two 512kx36 ZBT SRAMs soldered onto the labkit to store the frame for processing. Furthermore, because of the 2 clock cycle delay from the read/write signal to retrieval/execution, special care needs to be taken when retrieving pixel values.

The hue and edge analyzers are discussed in greater detail in later subsections.

[1] The bus widths are omitted from this diagram as they will be discussed along with implementation details in a later section.

Grids and Cells

In our initial proposal, we suggested a fully-convolutional model which stacks layers of convolution operations on top of each other, analogous to a convolutional neural network with hand-crafted input features. After implementing our modules and rigorously testing each component separately, we discovered unexpected integration issues.

Although our modules satisfied all the memory and timing limitation, as planned, we ran into routing problems due to the large number of highly connected wires in our original design.

Therefore, we simplified our model to make use of a fixed grid, where each cell is only connected to the ones immediately surrounding it, significantly reducing resource usage.

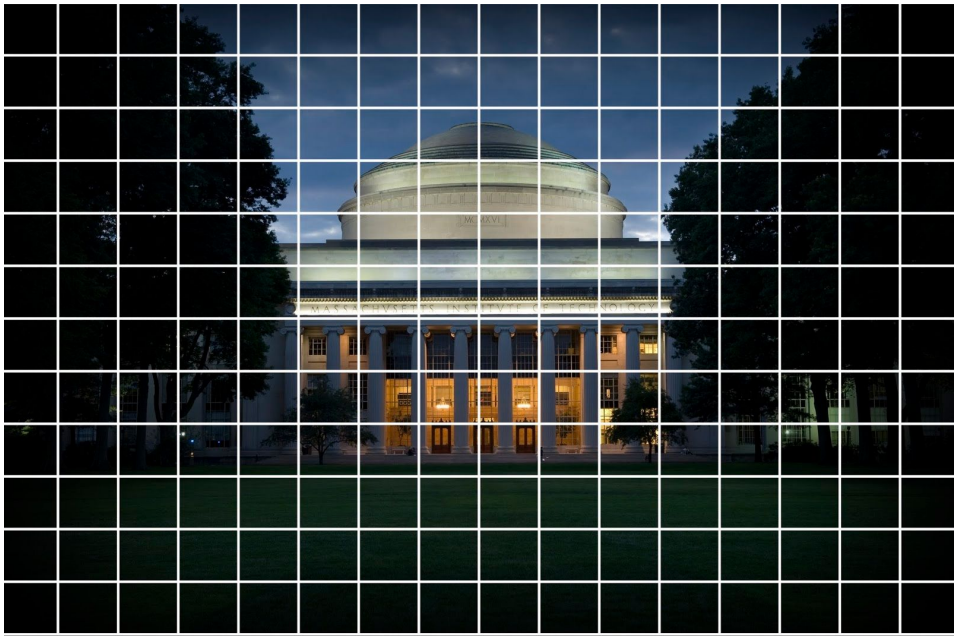


Figure 2: A picture of the great dome divided into the 16x12 grid.

The VGA display resolution is 1024x768, which we divide into a 16x12 grid. The standard variable names used to index into the raw display pixels are *hcount* and *vcount*, and we use the 4 highest bits of *hcount* and *vcount* to produce *hcell* and *vcell* which can be used to index into our grid.

We aggregate the edge and hue features within each cell into histograms to further reduce the computational complexity. When attempting to detect objects, we look at the features for the target cell as well as cells immediately left and right of the target, which allows our model to learn positional bias.

Note that this fixed grid is equivalent to our fully-convolutional proposal when the cells are of size 1; we explored the tradeoff between the size of the grid and the resource consumption by trying cells of size 1, 32, and 64 and find that 64 is the smallest cell size which can be synthesized in a reasonable amount of time.

Hue Analyzer[2]

At a high level, the hue analyzer ingests a heavily quantized 2-bit color value, computes a histogram for the distribution over possible colors for each cell, and then finds a linear decision boundary between the "correct" distribution of the training sample and the other distributions.

[2] Writing and figures by Kevin Zhang, design and implementation by Felipe Hofmann.

Because the NTSC camera operates in the YCrCb color space, we choose to quantize the colors directly in the CrCb space (as opposed to converting to HSV) to avoid using multipliers in this stage. We experiment with a 2-bit quantization and 3-bit quantization and find that they offer similar performance.

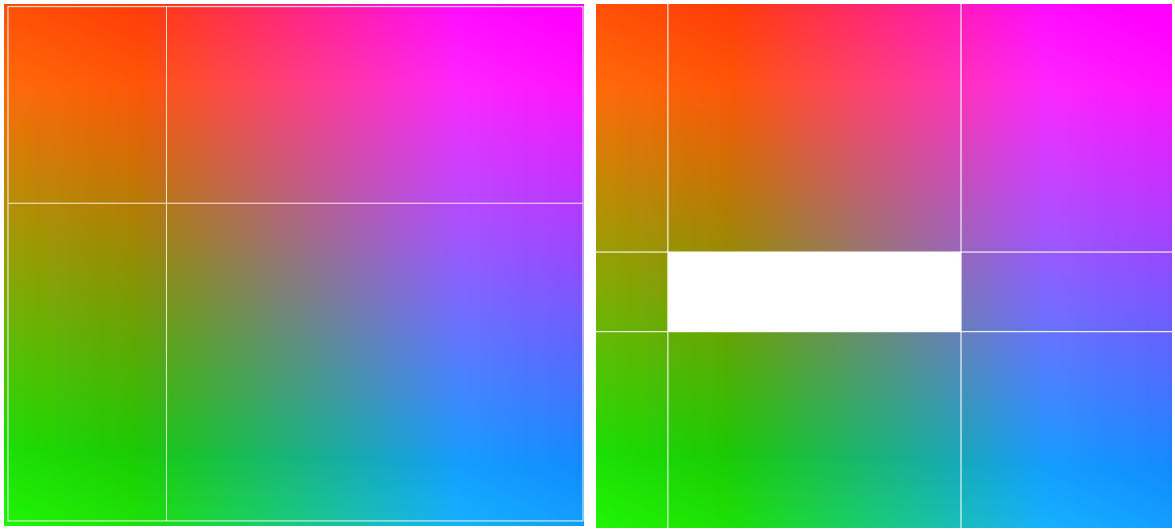


Figure 3: The 2-bit and 3-bit quantization operations - note the deliberately non-uniform spacing. The white cell indicates a set of colors which is ignored in the 3-bit quantization mode.

However, the 3-bit quantization results in more noise in the NTSC module - the lack of modularity due to sensitive timing constraints in the provided camera code will be discussed in a later section - so we choose to proceed with the 2-bit quantization.

The below figure shows the result of the 2-bit quantization operation, with each of the 4 quantized colors mapped to a new point in RGB space for easy visualization.

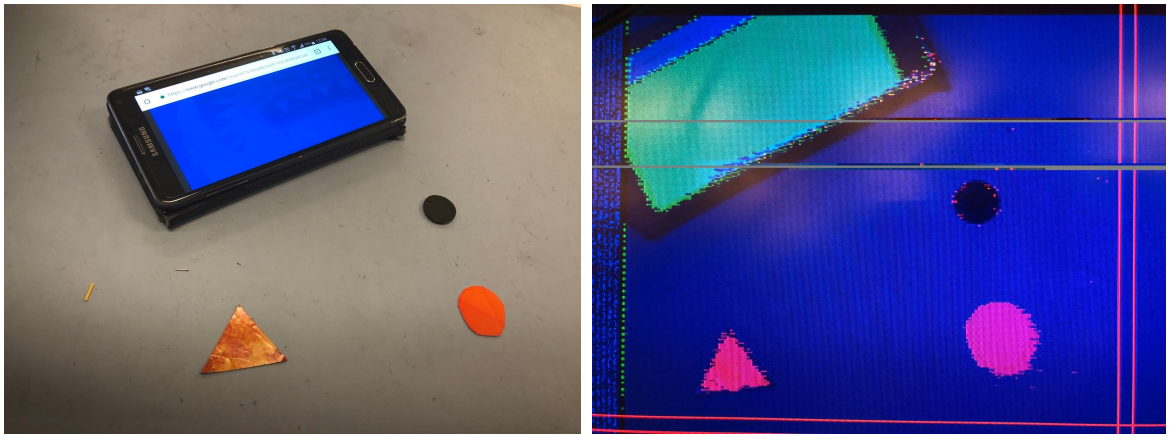


Figure 4: Color quantization.

The above figure shows the result of the 2-bit quantization operation, with each of the 4 quantized colors mapped to a new point in RGB space for easy visualization.

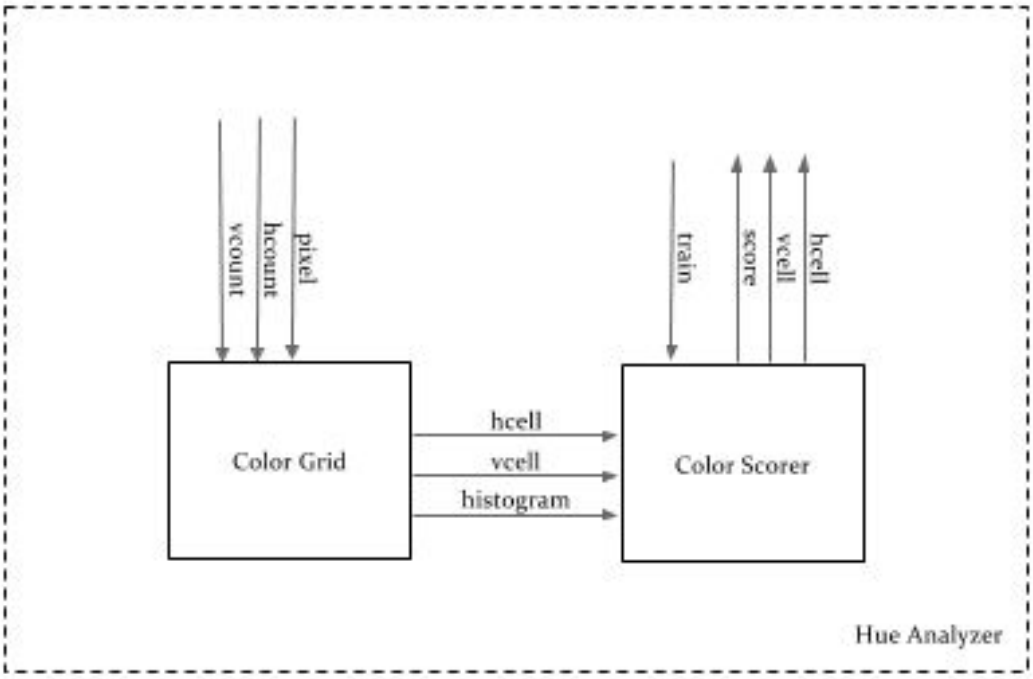


Figure 4: Block diagram showing the hue analyzer.

The color grid module accepts the quantized color and computes histograms. Since *hcount* and *vcount* are guaranteed to arrive in a fixed order - specifically left-to-right, top-to-bottom - the color grid module only needs to keep a running histogram for each of the 16 cells in the current row. When the histogram for a cell is complete, the valid signal is pulled high and the *hcell*, *vcell*, and *histogram* registers are filled with the correct value.

The color scorer receives the histogram and computes the similarity between a latched "gold" histogram and the incoming histogram with combinatorial logic. The *gold histogram* stores the correct histogram which perfectly matches the target object, so this similarity is directly related to the probability of the current cell containing the target object.

When the train signal is pulled high and *hcell* and *vcell* are in the center of the image (underneath the crosshairs which indicate the training object), the current histogram is stored into the "gold" histogram.

Edge Analyzer[3]

The edge analyzer is the primary detection module. It is a slightly altered FPGA implementation of the histogram of oriented gradients model proposed by Dalal et al. (2005) for human detection.

It receives the pixel data in the standard left-to-right, top-to-bottom order and uses line buffers to efficiently detect edges with both 2x2 and 3x3 kernels. The edges are classified and aggregated into local "histograms of oriented gradients" which are used as features for a KNN classifier.

The 2x2 kernel is a simple gradient operation, whereas the 3x3 kernel is the Sobel operator. Both kernels approximate the derivative in the vertical and horizontal directions, where the sign of the derivative is used to classify the type of edge.

	2x2	3x3 (Sobel)

[3] Writing, implementation, and figures by Kevin Zhang, testbench implementation by Felipe Hofmann.

dx	-1 +1	+1 0 -1 +2 0 -2 +1 0 1
dy	+1 -1	+1 +2 +1 0 0 0 -1 -2 -1

After experimenting with the results, we find that although the Sobel edge detector offers greater noise invariance, the increased complexity introduces timing issues which causes the NTSC module provided us to become more glitchy, introducing more noise into the system and producing a net decrease in accuracy.

Therefore, we choose to use the simpler 2x2 operator which has the additional benefit of being delayed by only 1 clock cycle as opposed to being an entire line (1343 clock cycles) behind, making it easier to reason about the timing of our modules.

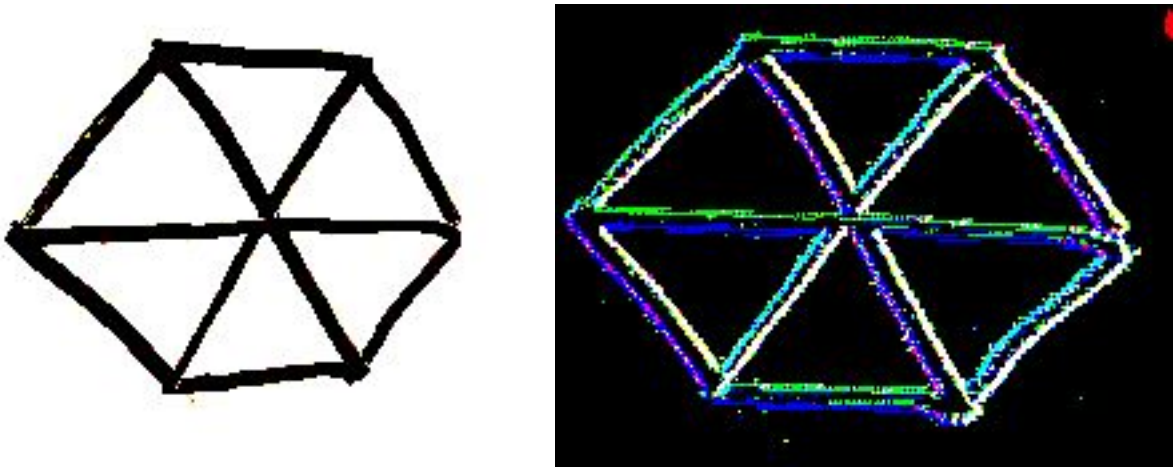


Figure 5: Edge detection and classification with different types of edges represented by different colors.

The edge detection algorithm and classified edge types[4] are shown in the above figure for a test image. Each color - light blue, white, green, blue, yellow, and purple - represents a different type of edge. This edge transform module feeds into the edge grid and edge scorer modules shown below.

[4] Note that left-facing and right-facing vertical edges are not present in this test image but are, in fact, classified as separate edge types.

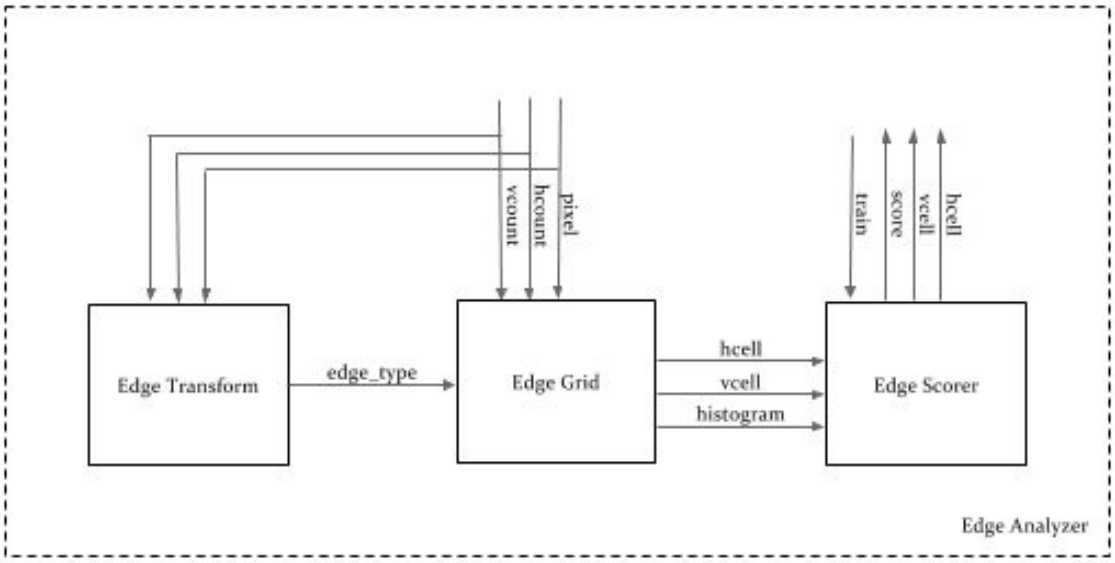
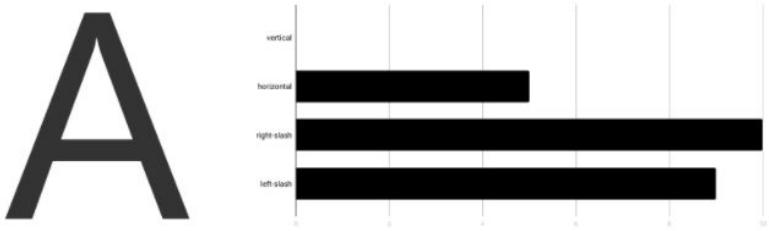


Figure 6: Block diagram showing the edge analyzer.

The *edge transform* module receives the pixel data in the standard left-to-right, top-to-bottom order and uses line buffers to efficiently detect edges with both 2x2 and 3x3 kernels.

The different types of edges are aggregated by the *edge grid* into a "histogram of oriented gradients" for each of the cells in the grid. As shown below, this histogram contains a lot of information about the object in the current cell and has been successfully used with SVMs for various image recognition tasks.



The histograms in the current cell (as well as the adjacent cells) are used as features for the *edge scorer* which implements a simple KNN classifier, returning a score proportional to the probability that the object is in the cell.

Linear Scorer[5]

The linear scorer module computes a linear combination of the two scores produced by the hue and edge detection modules. Since the timing for the 2 modules is different, the linear scorer is responsible for delaying the signals from the hue scorer by 1 clock cycle to keep it in sync with the edge detection module.

The coefficients for the linear combination are determined experimentally by trying various combinations of powers of 2 and evaluating the accuracy, resulting in coefficients with values ½ and ¼ for the edge and hue modules respectively.

[5] Design, writing and implementation by Kevin Zhang, testbench implementation by Felipe Hofmann.

The below figure shows a table surface with various objects. We train the detector to recognize the black dot located in the center of the screen underneath the crosshairs (crosshairs not shown here) by pressing button 1, and then release the button. As we move the black dot, we see that the red box indicates the most likely location for the black dot, while the other objects are identified but filtered out.

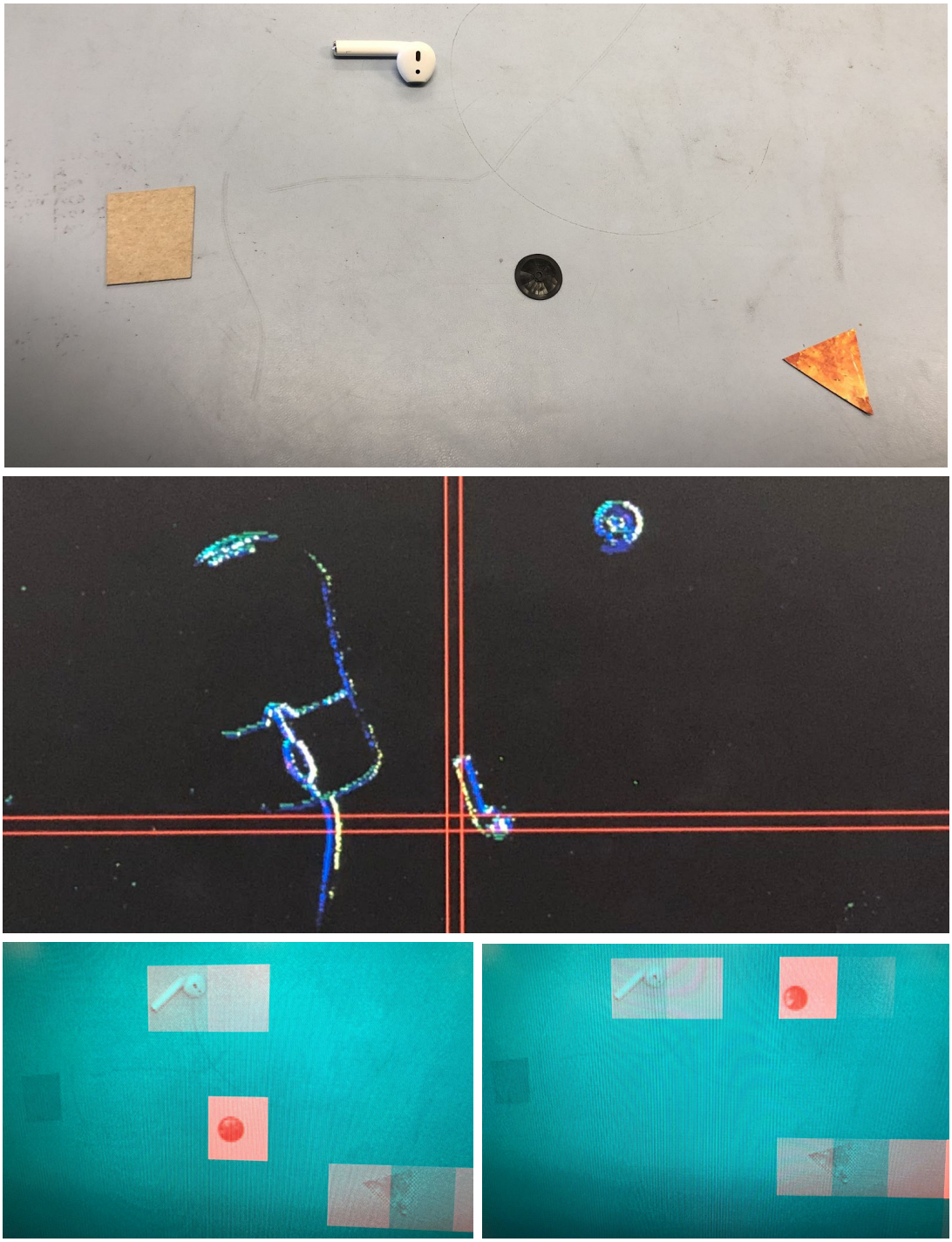


Figure 7: Tracking a black dot (target sample) and ignoring misc. objects.

Modules

NTSC Decoder[6]

Inputs: tv_in_ycrcb [9:0]
Outputs: ycrcb [29:0], field, vertical, horizontal, ready

This module reads the NTSC stream produced by the ADV7185 decoder and produces YCrCb values as well as the appropriate field/vertical/horizontal signals. These outputs are passed to the *NTSC to ZBT* module.

NTSC to ZBT[7]

Inputs: field, vertical ,horizontal, ready, ycrcb [29:0]
Outputs: ntsc_addr [18:0], ntsc_data [35:0], write_enable

This module uses the field/vertical/horizontal signals and counters to compute the current pixel position and outputs the memory address associated with that pixel. It also extracts the luminescence component of the ycrcb signal. These outputs are used by the primary controller to fill the ZBT memory.

The initial implementation was provided by the staff, and we made significant modifications. The raw pixel data is stored on ZBT memory which is 36 bits wide; the brightness of each pixel is represented by 8 bits, allowing us to pack 4 pixel brightnesses into each memory location. We use the 4 remaining bits to store the quantized color for pixels 1 and 3.

XVGA[8]

Outputs: hsync, vsync, hcount [10:0], vcount [9:0]

Generates XVGA display signals (1024 x 768 @ 60Hz). This is the standard module provided by the staff and is used without modifications.

Display[9]

Inputs: hsync, vsync, hcount [10:0], vcount [9:0], data [35:0]
Outputs: hsync, vsync, address [18:0], rgb [7:0]

This module accepts hcount/vcount and computes the appropriate memory address, taking into account the 2 cycle clock delay in reading from SRAM.

Furthermore, since the memory is 36 bits wide with 4 pixels packed into each memory location and the extra 4 bits used to store the chroma values, this only needs to read 1 value every 4 clock cycles.

The output of this module is a pixel value, which gets merged with the object recognition score produced by the linear scorer in the primary controller. Furthermore, buttons 0 and 3 will override the pixel data in memory and display the edge and color detection screens respectively.

---

[6] Provided by the 6.111 staff.
[7] Provided by the 6.111 staff, modified by Kevin Zhang.
[8] Provided by the 6.111 staff.
[9] Provided by the 6.111 staff, modified by Felipe Hofmann.

ZBT Memory[10]

Inputs: write_enable, address [18:0], write_data [35:0]
Outputs: read_data [35:0]

This module interfaces with the onboard SRAM. There is a two cycle delay on read/write and accepts/return 36-bit values.

Primary Controller[11]

Inputs: hcount [10:0], vcount [9:0], ntsc_addr [18:0], ntsc_data [35:0], write_enable, read_data [35:0],_____ address [18:0], score [3:0]
Outputs: hcount [10:0], vcount [9:0], data [35:0], address [18:0], write_data [35:0], write_enable

The primary controller connects all of our submodules and handles switching between read and write for our SRAM frame buffer. The decoder runs at 27 MHz while the XVGA runs at 65 MHz, but XVGA reads only happen once every 4 cycles due to the pixel packing described above, allowing us to store NTSC data on the free cycles.

BRAM[12]

Inputs: write_enable, address [18:0], write_data [3:0]
Outputs: read_data [3:0]

This module provides read and write ports which can be accessed from 2 different clock domains.

Absolute Diff[13]

Inputs: x1 [7:0], x2 [7,0]
Outputs: diff [7:0]

Assuming unsigned inputs, returns the absolute difference between the two values.

Histogram Diff[14]

Inputs: histogram1 [63:0], histogram2 [63:0]
Outputs: diff [7:0]

Computes the difference between 2 histograms, where each value is 8 bits. Purely combinatorial.

Edge Cache[15]

Inputs: clock, train, hcount [10:0], vcount [9:0], pixel [7:0]
Outputs: score [7:0]

---

[10] Provided by the 6.111 staff.
[11] Implemented by Kevin Zhang and Felipe Hofmann.
[12] Extracted from lab 5.
[13] Implemented by Felipe Hofmann.
[14] Implemented by Kevin Zhang.
[15] Implemented by Kevin Zhang.

This module connects to the edge and color scorers and waits for the valid signals. Due to the clock cycle delay in computing the score, this module is needed to cache valid scores in BRAM. Scores are stored into memory as an exponentially decaying average, acting as a sort of high-pass filter to prevent flickering.

Edge Scorer[16]

Inputs: clock, hcount [10:0], vcount [9:0], pixel [7:0]
Outputs: valid, hcell [3:0], vcell [3:0], score [7:0], gold_histogram [127:0]

When train is high, it caches the central histogram(s) and compares future histograms against this "gold" histogram with the histogram difference module.

Color Grid[17]

Inputs: clock, hcount [10:0], vcount [9:0], pixel [7:0]
*Outputs: valid, hcell*_latched *[3:0], vcell*_latched *[3:0], histogram*_latched *[127:0]*

Divides a screen into a 16x16 grid and computes histograms of the quantized color over each cell in the grid.

Edge Grid[18]

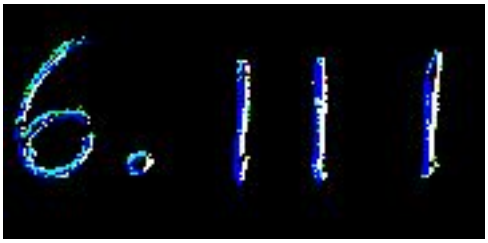
Inputs: clock, hcount [10:0], vcount [9:0], pixel [7:0]
*Outputs: valid, hcell*_latched *[3:0], vcell*_latched *[3:0], histogram*_latched *[127:0]*

Divides a screen into a 16x16 grid and computes histograms over each cell in the grid.

Edge Transform[19]

Inputs: clock, hcount [10:0], vcount [9:0], pixel [7:0]
Outputs: edge_type [3:0]

Returns the edge type of a pixel a clock cycle after it is received and holds the value there for a single clock cycle. Note that that edge detection kernels - both 2x2 and 3x3 (Sobel) require signed operation and testbenches are incredibly helpful for catching overflow and underflow problems.



Note that we use the edge_transform implementation in our final design; the edge_transform_33 module provides an alternate implementation which is deprecated.

---

[16] Implemented by Felipe Hofmann.
[17] Implemented by Felipe Hofmann.
[18] Implemented by Kevin Zhang, testbench by Felipe Hofmann.
[19] Implemented by Kevin Zhang, testbench by Felipe Hofmann.

Deprecated Modules

These modules were implemented but did not make it into the final design; they can be found in the version control history. The reasons for removing these modules range from timing constraints (i.e. NTSC camera noise), synthesis issues (i.e. compilation times in excess of 30 minutes), or ambiguous effects on system accuracy.

Line Buffer

Inputs: hcount[10:0], vcount[9:0], data [35:0]
Outputs: ready, hcount[10:0], vcount[9:0], subimage[511:0]

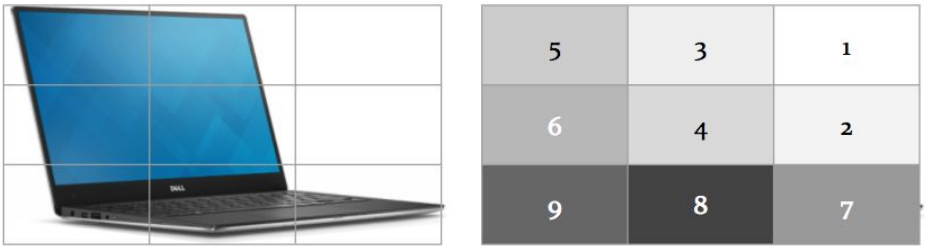
This module is responsible for caching 8-line subimages and feeding each 8x8 patch to the linear scorer by pulling ready high when the data is available.

Deprecated due to slow compile times.

Brightness Ordinal

Inputs: subimage[511:0], train
Outputs: score

This module accepts *subimage[64*8-1:0]* and train as inputs and returns a *score[3:0]*. It computes a brightness ordinal which splits the subimage up into an even grid and returns a sorted list of cells from brightest to darkest. This gives us a coarse measure of the morphology of the object.



As described above, this detector module also trains in 1 clock cycle and produces scores with purely combinatorial logic.

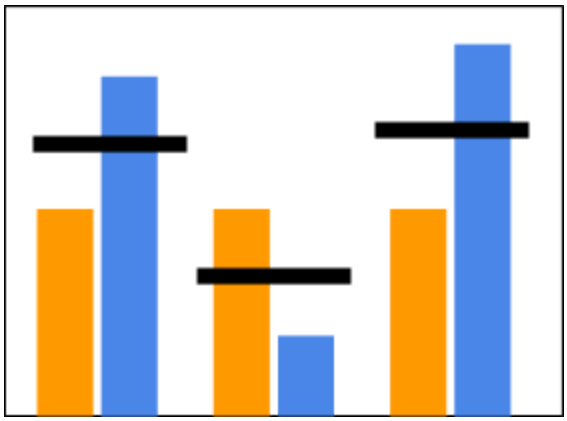
Deprecated due to slow compile times.

Edge Scorer w/ Autotune

Inputs: clock, hcount [10:0], vcount [9:0], pixel [7:0]
Outputs: valid, hcell [3:0], vcell [3:0], score [7:0], gold_histogram [127:0]

When train is switched high, it caches the central histogram(s) and compares future histograms against this "gold" histogram with the histogram difference module. While train is held high, it repeatedly measures the difference between the histogram for each cell and the gold histogram to get the range of expected scores.

The below figure shows hypothetical thresholds for an example where the orange bars indicate the gold histogram and the blue bars indicate a negative samples. When the blue bars are on the wrong side of the threshold, the score is decreased by an amount proportional to the distance from the threshold, with a lower bound score of 0.



This approximates a linear support vector machine as it looks for a hard boundary which separates the positive and negatives samples and provides significantly better accuracy. It iteratively chooses a new threshold for each type of edge by moving it towards the halfway point between the positive and negative sample.

Deprecated due to increased camera noise. Although this increases system accuracy, it also increases the amount of noise produced by the NTSC modules.

Edge Transform 33[20]

Inputs: clock, hcount [10:0], vcount [9:0], pixel [7:0]
Outputs: edge_type [3:0]

Returns the edge type of a pixel 1343 clock cycles after it is received and holds the value there for a single clock cycle. This module's behavior is virtually identical to the 2x2 edge transform module except it is more effective at filtering out natural noise.

Deprecated due to increased camera noise. Although this is more effective at filtering out background noise (i.e. specs on the wall), it causes the NTSC modules to emit patches of pure black and white pixels which result in a net decrease in performance.

[20] Implemented by Kevin Zhang, testbench by Felipe Hofmann.

Takeaways

During the course of this project, we encountered a variety of implementation issues. We discuss some of these issues below and offer insights on how these issues could be avoided.

Integration

We started by implementing each of the modules (as well as the corresponding testbenches) which were approved in our initial proposal. This included a variety of modules which have since been deprecated[21] such as the brightness ordinal as well as more robust edge and color kernels.

Unfortunately, during the integration phase, we discovered that although we had accounted for all the timing and memory constraints, the ISE design software was unable to route and place our design in a reasonable amount of time due to the large number of highly connected wires needed to support our fully convolutional 16x16 design.

After discussing with the staff, we were forced to significantly scale down our design, resulting in the creation of the detection grid as well as a more efficient method for computing edge and color histograms which results in a significantly less accurate object detection framework.

This process took a large part of our time so in future projects, we would recommend taking a more incremental approach and integrate various modules together as soon as they are completed so that unexpected problems can be discovered as soon as possible.

Timing Constraints

Another unexpected issue was the sensitive timing requirements by the NTSC camera code. When using a large amount of combinatorial logic - even with appropriate pipelining to satisfy the clock specifications - the NTSC camera modules become incredibly sensitive to timing.

This completely destroys any modularity in the system as changing a single constant, even in a distant, completely unrelated module, can corrupt the camera signal. The corruption can manifest as individual white and black pixels which (1) flicker randomly across the screen or (2) flicker in a circular pattern around the brightness gradient produced by light sources (i.e. like the contours of a topological map). In more severe cases, patches of the screen (or even the entire screen) can turn black at irregular intervals.

A long-term solution would be to rewrite the NTSC modules to use a FIFO queue instead of relying on perfectly timed reads and writes to memory. A short-term solution is to latch as many values as possible, regardless of whether they are actually necessary, and add timing constraints in ISE to get clearer debug output.

Since timing constraints weren't covered in detail in 6.111, some additional background readings would have been extremely helpful for avoiding these issues, especially during the earlier stages of our project.

[21] See commit 3d3128 in the git repository for corresponding verilog.

Conclusion

Our project successfully demonstrated the ability to accurately track objects with a low-resolution camera at 60 FPS, while consuming less than 1 joule for each frame processed which is around a 50x improvement from the equivalent C++ code running on a standard CPU. We believe we gained a lot of valuable experience on converting "standard" algorithms designed for von Neumann architectures into a hardware implementation.

Our code is published online at https://github.com/k15z/6.111-porthos.

References

Navneet Dalal and Bill Triggs, "Histograms of Oriented Gradients for Human Detection"

Jamie Schiel and Andrew Bainbridge-Smith, "Efficient Edge Detection on Low-Cost FPGAs"