Real-Time Statistical Saliency Using High Throughput Circuit Design and Its Applications in Psychophysical Study

by

David A. Blau

B.S., Computer Science (2006)
Massachusetts Institute of Technology

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2007

Author .....
Department of Electrical Engineering and Computer Science
May 25, 2007

Certified by .....
Edward Adelson
Professor of Brain and Cognitive Sciences
Thesis Supervisor

Accepted by .....
Arthur C. Smith
Chairman, Department Committee on Graduate Students

Real-Time Statistical Saliency Using High Throughput Circuit Design and Its Applications in Psychophysical Study

by

David A. Blau

Submitted to the Department of Electrical Engineering and Computer Science
on May 25, 2007, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Computer Science

Abstract

Using low level video data, features can be extracted from images to predict search time and statistical saliency in a way that models the human visual system. The statistical saliency model helps explain how visual search and attention systems direct eye movement when presented with an image. The statistical saliency of a target object is defined as distance in feature space of the target to its distractors. This thesis presents a real-time, full throughput, parallel processing implementation design for the statistical saliency model, utilizing the stability and parallelization of programmable circuits. Discussed are experiments in which real-time saliency analysis suggests the addition of temporal features. The goal of this research is to achieve accurate saliency predictions at real-time speed and provide a framework for temporal and motion saliency. Applications for real-time statistical saliency include live analysis in saliency research, guided visual processing tasks, and automated safety mechanisms for use in automobiles.

Thesis Supervisor: Edward Adelson
Title: Professor of Brain and Cognitive Sciences

Acknowledgments

I would like to thank my thesis advisor, Professor Edward Adelson, and supervisor, Ruth Rosenholtz, for presenting the challenging task of this project. I was looking for an opportunity to combine the many areas of research that interested me, and this project fit very well. The potential to do good work for excellent scientists was motivation to continue, even when things seemed impossible.

This work was developed in the digital electronics laboratory, run by Professor Anantha Chandrakasan and Gim Hom. Without a being granted access to equipment, permission to use development software, and help from circuit design experts, this project could not have been completed.

Last I would like to thank my family for the encouragement during the many sleepless nights dedicated to completing this research. Making my family proud kept me motivated for my entire academic career.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Understanding humans visual search can lead to understanding of how the visual system extracts information from the environment. Visual search is guided by both top-down mechanisms under conscious control and bottom-up mechanisms that cause certain features to pop out from their surroundings. Modeling visual search allows us to predict whether finding objects in clutter is difficult. With an accurate prediction model, we can infer statistical saliency for objects as they are encountered, where saliency is inversely proportional to search difficulty.

This thesis focuses on the problem of computing statistical saliency in real-time. The work describes an implementation using high throughput circuit design for parallel processing capable of performing the calculations in real time. Implementing the statistical saliency model in circuitry allows for live analysis and facilitates exploration of temporal features that are difficult to examine with still images.

1.1 Thesis Overview

Statistical saliency is the measure of how far a target lies from distractors in features space. Using appropriate features spaces, statistical saliency helps explain nontrivial characteristics of visual search. Chapter 2 describes the statistical saliency equations that are implemented

directly in circuits.

Real-time statistical saliency calculation is inefficiently performed using conventional implementations. Many arithmetic operations and image filters do not utilize parallelization or pipelining and result in wasted time and space in implementation. Chapter 3 describes the design of high throughput circuitry that is made to maximize parallel processing and pipelining. In high throughput design, minimal state is stored, few operators are idle, and independent calculations are parallelized.

Using high throughput circuit design, Chapter 4 describes the implementation of the statistical saliency equations in Chapter 2. The implementation is highlighted by its modularity. The features from the model are included, but can be easily exchanged for others.

Chapter 5 discusses the benefits of the static saliency real-time system. Two experiments demonstrate the utility of the time domain in saliency problems. Also discussed is an additional saliency feature that addresses the problems encountered in the experiments.

Chapter 2

The Statistical Saliency Model

Deterministically inferring meaning from visual stimuli requires quantifiable properties be calculated from the image. A direct approach aimed at mimicking biological vision involves performing image segmentation and object recognition using the image data. This method requires significant computational resources to be applied in real-time. Research shows there are features for which efficient mathematical calculation on local image regions produces a measurable property, namely saliency [5]. Saliency is inversely proportional to the difficulty of searching for a target among distractors [5], demonstrated by the asymmetries of otherwise complementary searches [7].

2.1 Search Asymmetries

A quantitative measure of a search task is the mean search time required to accurately identify a target amidst distractors [6]. Asymmetries in search time exist between identifying a target with feature, in a field of distractors with another feature and identifying a target exhibiting feature in distractors with feature. For example, it is more difficult to search for a stationary target among mobile distractors than it is to search for a mobile target among stationary distractors. Simple bottom-up explanations predict the asymmetry is the result of the unequal distributions of detectors for features target and distractor [8]. Explanations that

impose constraints on amounts of detectors are difficult to generalize to human behavior. Predicting difficulty as a measure of target saliency better fits with our understanding of human behavior.

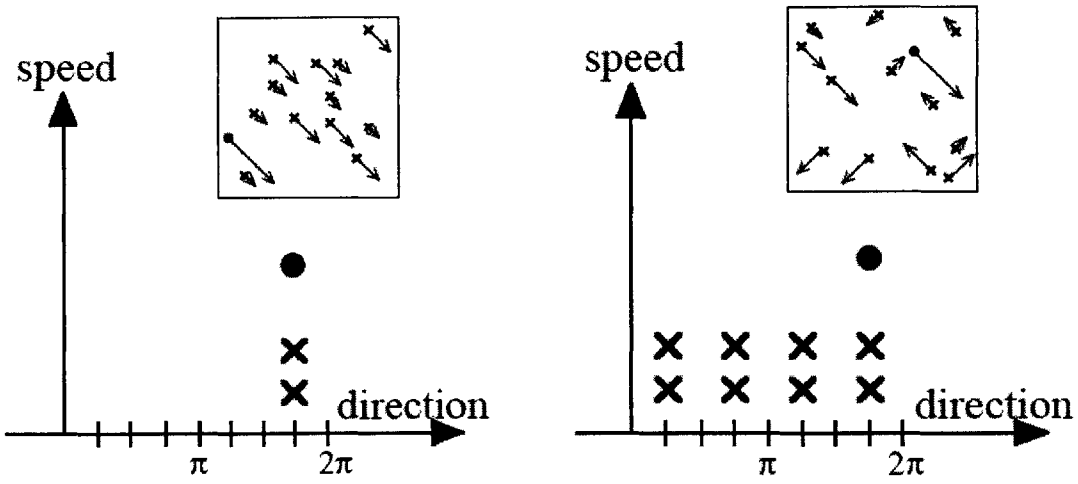
Experiments can be presented in both symmetric and asymmetric form. Switching the target and distractors in one representation of a feature space may predict equal search time, while in another representation predict different search difficulty. For example, a search task for a target in a field of distractors with differing velocities may be symmetric when velocity vectors are represented in cartesian components and asymmetric when expressed in polar form. Appropriate choice of feature space affects the prediction of search time.

The asymmetries in search tasks could be the result of asymmetric design or asymmetric mechanisms in the visual system. Because there are fewer search asymmetries than symmetries, relying on asymmetric mechanisms could be difficult to resolve with the corpus of verified results. Instead, the statistical saliency model treats the target as a point in a feature space and measures search time as a function of the distance to the mean and the covariance of the distractors. Using an statistical approach to determine search difficulty allows us to predict the salience of arbitrary targets among distractors.

2.2 Statistical Saliency

In a feature space of one dimension, a target's saliency is measured by its distance in the feature space from distractors [5]. It is easier to detect a target with a feature that lies outside the cloud of distracter features, in feature space. The features of the distractors is characterized by the mean μ and standard deviation σ; the distinction between the target feature and a small, dense distribution is more pronounced than between the target and a large, sparse distribution in which the target could be confused with a distracter. Analytically, the measure of the degree to which the target is an outlier of the distractors is summarized by the equation:

$$\delta = \frac{t - \mu}{\sigma} \tag{2.1}$$



(a) Uniform direction

(b) Varying directions and speeds

Figure 2-1: (a) Search is easy for targets with among distractors traveling in the same direction. (b) Search for a target among distractors of varying directions and varying speeds is more difficult because the covariance of the distractors is larger.

In motion analysis, detecting a moving target with velocity $\mathbf{v}$ in a field of distractors with mean velocity μ, the statistical saliency model is can be restated as finding the the Mahalanobis distance,

$$\Delta^2 = (\mathbf{v} - \mu)^T \Sigma^{-1} (\mathbf{v} - \mu) \tag{2.2}$$

where Σ is the covariance matrix. The target is more salient as the distance Δ between the target and distribution of distractor features increases. The statistical saliency model model correctly predicts the search time asymmetry between an oscillating target in stationary distractors and a stationary target in oscillating distractors. In the former task, the feature distribution is a point at the origin for all distractors while the target lies at a nonzero vector in the field. In the latter case, the target feature vector is zero and the distracter distribution surrounds the origin. The stationary target it is within a standard deviation of the distribution of mobile distractors, but the oscillating target is not. The saliency calculation is appropriate for a particular feature space representation and disparities between feature space representations must be accounted for better explain search time results.

Beyond simple oscillation, the statistical saliency model predicts the difficulty in searching for a target of unique speed among distractors traveling in the same direction with varying speed, versus searching for the same target among distractors traveling in varying directions and speeds, shown in Figure 2-1 [6]. In both situations, the saliency of the target is related to its distance in feature space to the distractors [7]. Restricted to a single direction, the distractor distribution results in relatively small and eccentric covariance ellipses. Without direction restriction, the covariance ellipses are larger to account for the possible directions and speeds. The target's distance from the distractor mean is changes only slightly for the case varying directions, but the covariance ellipse is much larger. The resulting saliency estimate is smaller, predicting longer search times.

2.3 Model Summary

The statistical saliency model quantifies the notion of how far a sample is from its surrounding. The model helps explain the different asymmetries observed in searching for various targets among distractors. Targets are difficult to find when surrounded by distractors of similar or highly variable features. Conversely, targets are easy to find when the are surrounded by features of different and uniform features. Using the statistical saliency model, search time estimation can be extrapolated for features in images and studied as a measure of bottom up pop-out. Though described for object motion, the statistical saliency model is generalizable to any feature set. In the system proposed, the statistical saliency model is applied to contrast, orientation, and color. The final saliency calculation is the length of the vector of contrast saliency, orientation saliency, color saliency. The full saliency value represents the estimation of feature pop-out in the image.

Chapter 3

High Throughput Circuit Design

While computer programs and simulations can always be used to demonstrate ideas, dedicated and embedded systems are well suited for visual perception research in applications from car safety to video surveillance. Data-rich problems, such as image analysis, cater to high throughput, parallel processing design for fast, reliable results. This chapter discusses full throughput mathematical circuitry for use as components in streaming computational systems. This chapter explains detailed design of addition, multiplication, and division circuitry for small integers and then outlines design for digital image filters.

3.1 Field Programmable Gate Arrays and Integrated Circuits

The medium on which the system is designed is a field programmable gate array (FPGA). A FPGA is a reprogrammable circuit with little preprogrammed function. Circuit designs, written in Hardware Descriptor Languages, can be synthesized and transferred to the FPGA via physical connection, allowing the FGPA to assume any programmable function. Many tools exist in developing synthesizable circuitry, both free and commercial; techniques in Hardware Descriptor Languages are outside the scope of this thesis.

FPGA implementation is often used as a prototype for an application specific integrated

circuit (ASIC). ASICs are significantly cheaper, faster, more stable, and can consume less power than FPGAs, but lack the ability to be modified once the circuits are printed. For well defined problems, ASICS may be sufficient and the techniques discussed in this chapter can be transferred directly. Often, the scope of the problem is changed by its proposed solutions and the ability to test new approaches becomes important. Scientific experimentation is better suited for the mutability of programmable gate arrays and for this reason, an FPGA is preferred.

FPGAs are organized as an array of slices, where each slice is an array of programmable logic blocks. A single block has one or more lookup tables and connective circuitry. Lookup tables are used rather than simple gates because they can achieve equivalent behaviors without physically changing any hardware attributes. With efficient utilization, one logic block may be used as part of multiple circuits, especially if each circuit requires a mutually exclusive part of the block. Along a slice, blocks are aligned to facilitate data efficient transfer and allow grouping. For example, A cascading logic operation such as addition fits into blocks along a single slice. Along the edges of an FPGA are specialized input/output blocks, dedicated to managing data traffic in and out of the FPGA through a single pin. These blocks will be used for video data and graphical output in the design. Last, FPGAs include increasing amounts of embedded circuits including hardware multipliers, dividers, memory blocks, buffers, and even microprocessors. These components require less physical space on the chip then their counterparts in programmable logic and are directly available to circuits synthesized in the FPGA. Extensive use is made of embedded memory blocks, as storage is the least efficient use of a logic block.

3.2 Dual Port Random Access Memory Queues

Embedded in FPGAs are numerous small random access memory (RAM) blocks. Because circuits are easily parallelizable, FPGAs have large numbers of small RAM blocks instead of a single RAM chip so that each individual circuit can access a RAM block independently from other circuits. For image filtering problems, in which single pixels are streamed in

sequentially, many components must retain state until enough data has been calculated. In particular, image filters often require half of their taps be supplied information before the filters produce any output. At multiple stages in processing, short segments of the image must be stored for filtering.

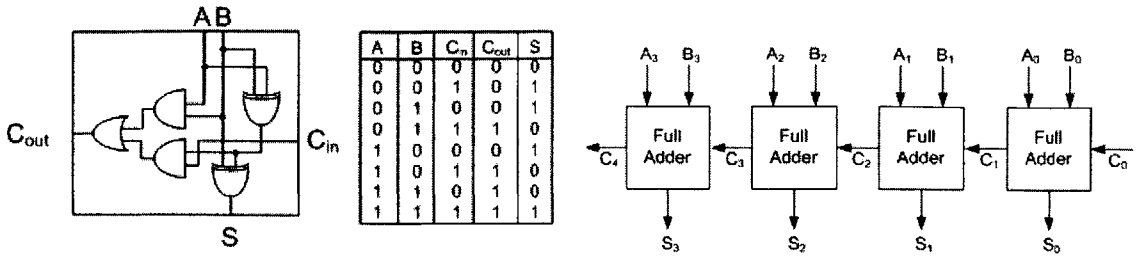
RAM is accessed and modified by ports. A memory block will have a number of pins for data, other pins for address, and others for enabling logic. Dual port RAM has two independent ports that can read from the same block of memory, with special rules for simultaneous operations on the same address in memory. Dual port RAM can be abstracted to a first-in-first-out queue by controlling the addresses at each port. By maintaining two counters, one for the start and another for the end, a dual port RAM can be treated as having a data input port, data output port, an enqueue pin and a dequeue pin. Queues are preferred over logic blocks for their efficiency and their simple circuitry.

3.3 Binary Arithmetic

Complex calculations performed by a processor can be reduced to binary and floating point arithmetic. The arithmetic operation performed most is addition. The addition operation is a design choice that affects the speed an accuracy of computation. As the designed system works in small integer, floating point calculation is not discussed.

3.3.1 Single Cycle Adder

In high throughput design, the ability to perform an addition in a single cycle is top priority. The simplest adder circuit is the ripple carry adder. The sum is calculated bit by bit and a carry signal is required to propagate down the circuit to the most significant bit. When the propagation becomes too long, ripple carry adders can be segmented into carry look-ahead adders that skip groups of bits at a time.



(a) Full Adder

(b) Ripple Carry Adder

Figure 3-1: (a) Full adder circuits add single bits with a carry out signal. (b) A sequence of full adders calculate a sum by adding each pair of bits from the inputs and a propagated carry bit. Because of the carry propagation, this circuit is a Ripple Carry Adder.

Ripple Carry Adders

Ripple carry adders are comprised of an array of full-adder units. A single full-adder adds three bits in unary, denoted A, B, and Carry-in, and produces a sum and carry-out value in binary, shown in Figure 3-1(a). The carry-out value of a full-adder is used as carry-in to the next full-adder of increasing significance, shown in Figure 3-1(b). The carry-out value of the most significant full adder can be used to detect integer overflow or underflow as there are more bits of information than bits in either input. The same value can also be used to store the sign of the number using twos complement arithmetic. Because the same adder circuitry works under twos complement, signed integers can be computed with equal efficiency.

The propagation delay of a ripple carry adder is proportional to the number of full-adder units in its chain. The number of full-adders is equal to the number of bits in addition. Each full adder performs at most three sequential binary operations on its inputs. For an eight bit addition, the sum will be valid twenty-four binary operations after the inputs are set.

Carry Look-Ahead Adders

Carry look-ahead adders focus on propagation signals instead of bit addition. In these adders, a group of four pairs of input bits is used to determine whether the group should generate a carry bit or propagate the carry bit passed in to the group, while determining four sum bits in parallel. By propagating the carry signals more quickly, more significant groups can begin

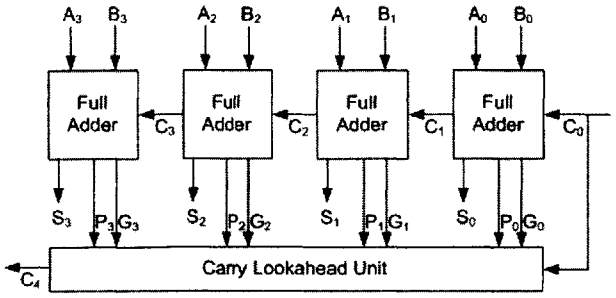


Figure 3-2: Carry Lookahead blocks use Generate and Propagate signals from full adders to quickly propagate a carry in past groups of full adders. Within a group, individual bit sums are calculated the same way as in four-bit Ripple carry adders.

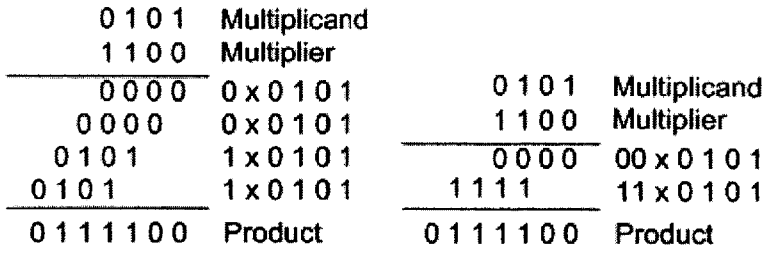
calculation sooner than by ripple carry propagation, but at a slight time penalty and with additional circuitry. A carry lookahead group of four full adders is shown in Figure 3.3.1. Because of the added complexity, carry look-ahead adders are only more efficient for sums of more than sixteen bits. In the presented system, very most additions are of sixteen bits or less and ripple carry adders are efficient.

3.3.2 Multiplier

Multiplier circuits are computationally and physically much more expensive than adder circuits. Computer processors have a very limited number of multipliers because they perform very few concurrent multiplications. FPGAs more embedded multipliers than processors, but even these numbers may be limiting for applications requiring many parallel multiplication. Embedded multiplier usage is can be mitigated in two ways: avoiding multipliers for constant multiplicands and pipelining adders for full throughput.

Twos Complement Binary Multiplication

A multiplication operation is a series of additions. In circuitry, additions can occur sequentially in time or sequentially in space. Iterative addition circuits require the same physical resources as a single adder, but require many cycles to complete. By cascading the adders as a large circuit, the multiplication can be performed without requiring a running product,



(a) Radix 2 Multiplication (b) Radix 4 Multiplication

Figure 3-3: Performing multiplication using a higher radix can speed up computation, but high radix multiplication requires precomputation of multiples of the multiplicand.

but the the circuitry required is quadratic in the number of bits.

Iterative multipliers store the multiplier, multiplicand, and a running product. During each cycle, the product is incremented by the multiplier and the multiplicand is decremented. For an eight bit multiplication, a sixteen bit product can be computed in eight cycles by shifting and adding.

Pipelining Multiplication

Cascading multipliers perform the addition and shifts are achieved by arranging adders in a cascaded and shifted pattern. Naive implementations may propagate a single n-bit multiplication across n^2 full-adders, while the other adder units remain unused. A more careful approach propagates the multiplier and multiplicand with each addition, allowing for each cascaded adder to perform a respective addition of the multiplication.

Multiplication can be achieved in fewer cycles by performing the additions with a higher radix. Using two bits instead of one multiplies uses the radix four instead of two, and completes the multiplication in half as many steps. Figure 3-3 demonstrates radix two multiplication in contrast to the speed of radix four multiplication.

Radix four multiplication requires the pre-computation of the three times the multiplier. This can be done by either adding the multiplier to twice the multiplier, where multiplication by two is shifting one bit towards the most significant bit. The extra adder can be avoided if 11_2 is Booth recoded as a subtraction by 01_2 for the next two more significant bits [3].

Subtraction and negative number support is possible because of twos-complement adders.

Multiplications are expensive operations, but when well pipelined, the cost of a full throughput 10-bit multiply can be that of five ripple carry adders.

3.3.3 Divider

Two types binary division algorithms exist: approximation and recurrence. Approximate methods are faster, converging in few cycles, but rely on analog division hardware. Division operations can be often avoided by application specific design choices. For the few necessary division operations, division can be done by approximate methods if hardware is available and by recurrence methods otherwise.

Approximate Methods

Newton-Rhapson division[4] involves estimating the reciprocal of the divisor and multiplying it to the dividend to find the quotient. The approximation successively calculates

$$r_{i+1} = r_i(2 - d \times r_i) \tag{3.1}$$

where r_i is the ith approximation to the reciprocal of the divisor d. This equation holds for divisions scaled such that $\frac{1}{2} < r < 1$. The quotient is estimated by multiplying r_i to the dividend. This method has higher accuracy if run for more cycles, but also relies on the denominator being greater than $\frac{1}{2}$, which cannot be guaranteed for arbitrary numbers between -1 and 1.

The Goldschmidt method[2] computes a quotient by successively multiplying dividend and divisor by a computed factor until the divisor converges to 1. Because numerator and denominator are multiplied by the same factor, the ratio remains constant. For the same reasons as the Newton-Rhapson method, additional cycles product higher accuracy for Goldschmidt division. The method also requires the divisor be scaled to be between 0.9 and 1.1.

Recurrence Methods

In an image analysis tool, time necessary to perform integer division is two orders of magnitude smaller than the time spent on image filtering. While the approximate division schemes converge to solutions quickly, their strict requirements of inputs are not worth the added complexity to scale the divisor. Though direct computation of integer division must be done bit by bit, there are no special cases necessary.

Integer division follows from the recurrence equation [3]:

$$P_{j+1} = R \times P_j - q_{n-(j+1)} \times D \tag{3.2}$$

where P_j is the partial remainder of division, R is the radix, q_i is the ith bit of the quotient, n is the number of bits in the quotient, and D is the denominator. The quotient is computed by comparing finding the largest bit value for which the divisor is less than the dividend at each bit, which is equivalent to long division radix 2.

Other implementations may include lookup tables for dividend-divisor pairs, but the speed gained requires exponential space. For eight and sixteen bit division, integer division is accurate and fast enough to be done without approximation.

3.3.4 Square Root

Square root algorithms also exist in both approximate and recurrence form with the same respective characteristics as division counterparts.

Approximate Methods

Newton-Rhapson root-finding methods[4] calculate square roots by solving for zeros of the equation:

$$x^2 - n = 0 \tag{3.3}$$

where x is the square root of n. Because the Newton-Rhapson method approximates roots by tangent estimation, convergence in the is dependent upon the input interval of search.

A good guess of the root will converge quickly while a poor guess may not. In general, the Newton-Rhapson method converges quadratically; the number of correct digits in the approximation doubles in each iteration.

An equivalent method is the Babylonian method, which can be derived from the Newton-Rhapson method. To find the square root of a number n, an initial value x is chosen, and each successive approximation is calculated as the mean of x and $\frac{n}{x}$. Like the Newton-Rahpson method, the Babylonian method converges quadratically.

For small integers, square root can be implemented as a lookup table. The size of the table is exponential in the number of bits in the radical. For inputs of with more than eight bits, a lookup table requires more space than logic of the arithmetic calculation

Restoring Method

Sixteen bit integers, in the absence of division hardware, can be computed using the principal equation[1],

$$(10_2 \times Q + D)^{10_2} = 100_2 \times Q^{10_2} + D \times (100_2 \times Q + D) \tag{3.4}$$

where each number is represented base 2, Q is the root calculated so far, and D are the digits remaining in the input number. As described by the equation, restoring methods construct the root one bit at a time. For each bit, to additional digits from the input are used to calculate the incremental quotient. Therefore, a n-bit square root method that performs $\frac{n}{2}$ additions and reconstructs $\frac{n}{2}$ bits requires $\frac{n^2}{2}$ full-adders. For a particular square root operation, only one adder unit is required and the rest can be used to pipeline subsequent square root calculations.

3.4 Digital Filters

Digital filters operate on images by point multiplying each pixel within the filters' support with a weight for each tap and then summing the weighted values. The operation is highly parallelizable because the multiplications are independent and the summing can be arranged

in logarithmic tree. Filters make heavy use of adder circuits described in section 3.3.

3.4.1 Kernel Pooling

How a particular filter computes a value is dependent upon the size, weighting, and shape of the filter kernel. Most saliency calculation is done on Gaussian filtered image data. Four different discrete Gaussian kernels are described and compared: Gaussian pyramid, small Gaussian, large Gaussian, and upsampling kernels. In performing saliency calculation of a particular image feature, less general kernels are used, including a difference-of-Gaussians kernel, and four difference-of-oriented-Gaussians kernels.

Kernel output computation is the sum of the pixel values at each tap, weighted by a constant. For the filters used in the Saliency implementation, the tap weights are no more than five bits, and can be achieved by fewer than two additions. Instead of using multiplication units, the tap weighting can be implemented with ripple carry adders of bit-width from five to fifteen. The weighted tap values can then be summed pairwise in a tree of adders, requiring computation time logarithmic in the number of taps.

Gaussian Symmetric Filters

Gaussian filters can be separated to a horizontal and vertical decomposition. Isolating the horizontal filtering allows the kernel to be applied directly to input data, buffered by a shift register. For each new image pixel, data in the buffer is shifted so that the first pixel that was inserted is removed and the new pixel takes the place of the previous pixel in the buffer. The output of the horizontal kernel is stored in a queue (described in section 3.2).

When the queue has accumulated enough lines of image data, depending on the kernel size, pixels from the queued lines can be used as input to the vertical kernel. When a pixel from the first queued line is used, it can be discarded because the filter will no longer use the pixel in any more calculation. The output of the vertical kernel is data that has been filtered by both kernels and is the output of the entire digital filter.

Asymmetric Filters

Many feature selective filters are constructed as Gabor functions or oriented Gaussians and cannot be expressed as the convolution of orthogonal kernels. For asymmetric filters, enough lines must be queued to match the full support of the filter. As a new image pixel is read, the buffer of pixels used as filter taps is shifted by one horizontally. The column of least recently added pixels is discarded, remaining pixels shifted by one, and the the next column of queue data and the next pixel are read to the buffer. Though simpler in design, asymmetric filters require buffers of size proportional to the kernel size, while radially symmetric filters require buffers of length equal to the sum of the component kernel sizes.

3.4.2 Line Buffering

Filtering is a local image operation and can be performed without a full image. As image pixels are streamed in, only enough lines to fill the kernel support are necessary. Until enough image lines have been read, each line is enqueued to memory such that a single column can be retrieved in a dequeue operation. This can be accomplished by dequeuing the column as a datum arrives, and then enqueuing the column with the new datum Following this procedure, the length of data in the queue never exceeds that of the width of the image

3.5 Design Summary

The chapter describes the binary arithmetic and digital filter implementations that all have full throughput; given streaming data input, the results of computation are streamed as output after a delay. For streaming video input, full throughput circuit design is necessary for real-time computation.

Chapter 4

Statistical Saliency Visual System

The statistical saliency visual system is a streaming video analysis tool that outputs real-time statistical saliency information. Using the high throughput circuit designs presented in Chapter 3, the system can stream output while maintaining minimal state. The system is designed to be modular; the number of dimensions in a feature determine the saliency calculation scheme, while features can be interchanged. Implementation details can be found in Appendix A, including Verilog implementations and filter kernel tables.

Video data enters the system through RCA connection using the NTSC video standard. Video data is stored in one of two external RAM chips. A RAM reader module reads the memory for the video data and outputs luminance and chrominance signals to the saliency module. Inside the module, data is first blurred and downsampled to construct a level of a Gaussian pyramid. The pyramid data is immediately written to the video RAM to be used as input for the next pyramid level. Luminance pyramid data is used as input by the contrast and orientation filters and transmitted directly to the one dimensional and two dimensional saliency modules, respectively. All pyramid data channels are used as input to the three dimensional color saliency module. As the each saliency module finishes, the calculations are coalesced into a single saliency value and stored in the second external RAM chip. When all three modules are finished, the process is repeated with the first pyramid level as input, which writes the second pyramid level to video RAM. The calculations are repeated one last

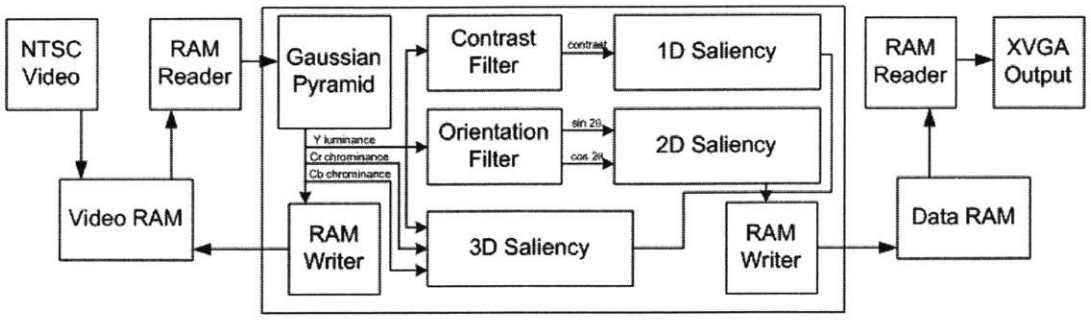


Figure 4-1: A block diagram of the statistical saliency visual system. Information enters from the NTSC video module, is processed by the saliency modules, and is displayed by the XVGA output module.

time with the second pyramid level as input. Then, the saliency stored on the second RAM is read and coalesced into a single saliency value for an image coordinate.

4.1 Video Input

NTSC video is decoded by a module that occupies input/output blocks connecting to a analog-to-digital converter that reads video data. Video signals are often compressed such that only one of the two chrominance values are sent with each luminance value. This lack of color information is justifiable because the human visual system and the statistical saliency visual system use luminance data more than chrominance. Pixel chrominance values are interpolated from neighboring pixels when absent. NTSC video is also interlaced horizontally. A frame of video data has either odd lines or even lines. NTSC video is read at 27MHz, while the rest of the system operates at 65MHz. Because of the difference in clock speeds, video is written directly to RAM so that calculation can continue uninterrupted.

Video data is stored in 18-bit segments in a single RAM chip that stores 36-bit words. The upper ten bits store luminance and the remaining eight bits store two four-bit chrominance values. A single address in memory stores two pixel values. Because video data arrives at half the speed of computation, four pixel values can be read between consecutive video memory writes, show in Figure 4.1.

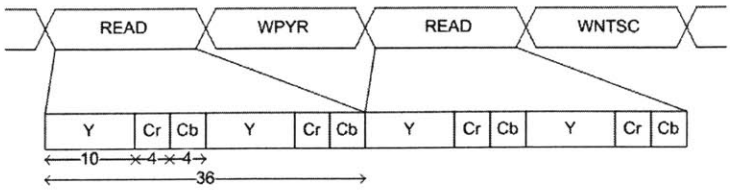


Figure 4-2: Video RAM Timing Diagram

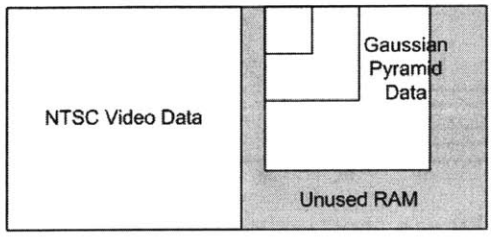


Figure 4-3: Video RAM Memory Layout

NTSC video is the standard used for the prototype system. Any video input can be used as input to the saliency modules, so long as the contrast and orientation filters are run on luminance data.

4.2 Gaussian Pyramid

Pixel data read from RAM is read to a Gaussian pyramid filter that blurs and downsamples the image. The radially symmetric blurring kernel has horizontal and vertical basis vectors with five pixel support. Horizontally filtered data is stored in the line buffers which are filtered by the vertical kernel. Only vertically filtered data for even horizontal and vertical image coordinates is output as valid pyramid data. The filter outputs downsampled horizontal and vertical image coordinates, filtered data, and an enable signal to control the subsequent filters and modules.

The pyramid data and coordinates is written back to video RAM for use as input to the next pyramid level. Pyramid data video RAM writes occur in between reads and opposite NTSC data writes, labeled WPYR in Figure 4.1. Pyramid data levels require decreasing

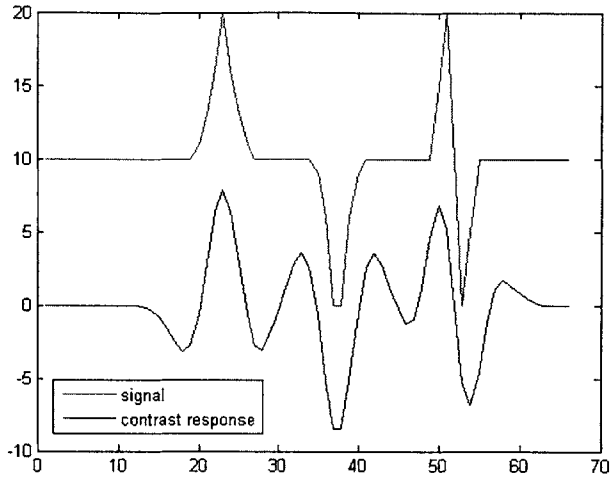


Figure 4-4: The contrast filter response is zero for flat regions and amplifies high frequency changes.

amounts of memory and the unused video RAM has space available, shown in Figure 4.2. Pyramid data, coordinates, and enable signal continue to the contrast filter, orientation, and color saliency modules.

When the pyramid data is read in as input, this module will blur and downsample the image again. Halving the image in size in each dimension white the feature detectors remain constant size is equivalent to doubling the feature size each level. The Gaussian Pyramid module mimics the levels and distributions of cells by increasing receptive field size.

4.3 Contrast Filter

Luminance pyramid data is passed through a difference-of-Gaussians filter to measure contrast. A difference-of-Gaussians filter is similar to an on-center, off-surround receptive field in that it will produce high values when luminance is high at pixels near its center taps and low at pixels at outer taps. When luminance is even, the positive center and the negative lobes cancel out resulting in low contrast. A contrast response to a one dimensional signal is

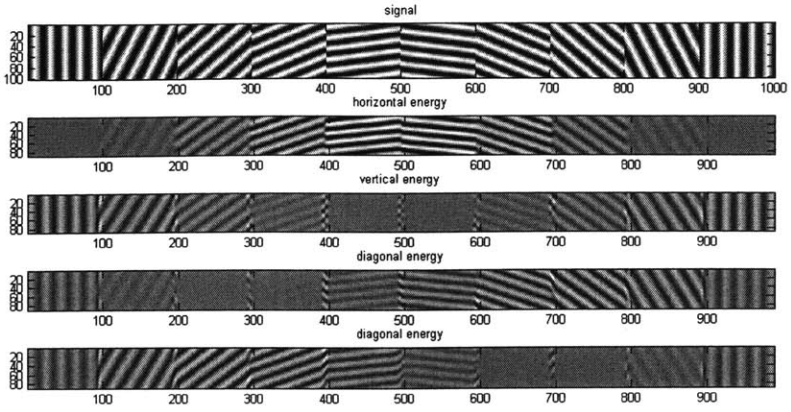


Figure 4-5: The original signal is a series of sinusoidal patterns, incrementally rotating by 20 degrees. The orientation energy responses to the signal favor orientations of a particular direction. Orientation filters are sensitive to 0 degrees, 45 degrees, 90 degrees, and 135 degrees.

shown in Figure 4.3. Contrast filtered data is used by the one-dimensional saliency module to calculate contrast saliency.

4.4 Orientation Filter

In addition to contrast, luminance pyramid data is also used in orientation selection. Pixel orientations are calculated as an energy optimization: an orientation at a pixel is the orientation that maximizes the the energy responses of four difference-of-oriented-Gaussians, separated by $\frac{\pi}{4}$ radians. The difference between horizontal and vertical squared responses approximates the sine of twice the orientation angle. Similarly, the difference between the diagonal squared responses approximates the cosine of twice the angle. A orientation energy responses to a sequence of images with varying orientation in Figure 4.4. The sine and cosine values are used by the two-dimensional saliency module to calculate orientation saliency.

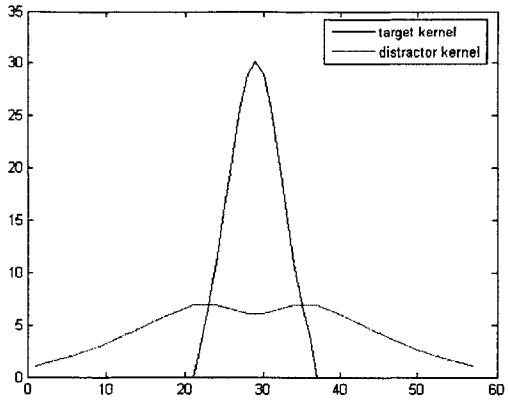


Figure 4-6: Target and Distractor Kernels

4.5 Saliency Modules

The saliency modules accept feature data as input and output streaming saliency values as output. Saliency is calculated as the target mean distance from the distractor mean normalized by the covariance. Each saliency module calculates target and distactor mean values, calculates the inverse covariance matrix, and computes a saliency value. For the one-dimensional case, the inverse covariance matrix is equivalent to division by the standard deviation. The output of the saliency calculations is coalesced for display.

4.5.1 Approximating Target Mean and Distractor Mean

The target mean at a pixel value is the response of the image region to a small Gaussian filter. The distractor mean is calculated as the difference between the target mean and the the response of the filtered image region to a large Gaussian filter, shown in Figure 4.5.1. This calculation is sensitive to objects that are the size of the small Gaussian filter, as they will increase the response of the target, but not the distractor. In subsequent pyramid levels, the smaller image size effectively increases the receptive field size of the target versus distractor.

4.5.2 Estimating Covariance

The covariances between each pair of dimensions i and j can be calculated using the covariance equation:

$$\mathrm{Cov}(X_i, X_j) = \mathrm{E}(X_i \cdot X_j) - \mathrm{E}(X_i)\mathrm{E}(X_j) \tag{4.1}$$

Using the target means as an estimation for the expected value, the equation becomes

$$\mathrm{Cov}(X_i, X_j) = T_{X_i \cdot X_j} - T_{X_i} T_{X_j} \tag{4.2}$$

which can be easily calculated by filtering the product of two dimensions and subtracting the product of the individual dimensions' target means. To account for the distractors, this formula is reapplied. The target mean covariance is smooth with the large Gaussian filter and the product of the dimensions' distractor means is subtracted from the result. The final covariance estimate measures how targets vary with eachother and with respect to how the distractors vary.

Using covariance calculations, the covariance matrix $\mathbf{\Sigma}$ can be constructed such that

$$\mathbf{\Sigma}_{i,j} = \mathrm{Cov}(X_i, X_j) \tag{4.3}$$

where i and j are both the row and column in the covariance matrix and dimensions of saliency features.

In the one dimensional saliency module, the variance is calculated using only the distractor means, and the standard deviation is used in saliency calculation rather than the variance.

4.5.3 Covariance Matrix Inverse

The ratio of eigenvalues of the covariance matrix define the ratio of major and minor axes of covariance ellipses of constant standard deviation. From the inverse covariance matrix, targets-distractor differences can be scaled to the dimensions of the ellipse. Matrix inversion

is straightforward: the inverse is the transpose of the matrix of cofactors divided by the determinant of the matrix. To prevent the covariance matrix from being singular, noise is added along the diagonal. In one dimension, inverting the variance is simply division.

4.5.4 Saliency Calculation

The saliency calculation involves multiplying the vector of target-distractor mean differences by the inverse covariance matrix. For each dimension, the distractor mean is subtracted from the target mean and multiplied by each element in the dimension's column of the inverse covariance matrix. The saliency is the length of the resulting vector. This formula can be expressed as a matrix equation:

$$\mathbf{s} = \left[T^T - D^T\right]^T \mathbf{\Sigma}^{-1} \left[T^T - D^T\right] \tag{4.4}$$

where $\mathbf{\Sigma}$ is the covariance matrix, T is a row vector of target mean values, and D is the row vector of distractor mean values. The calculated statistical saliency is the length of the vector $\mathbf{s}$. In one dimension, the saliency calculation is the target mean minus the distractor mean divided by the standard deviation.

The saliency calculation results in calculating the length of the difference of target and distractor means normalized by the covariance estimate. For data sets with very low variance, targets that lie far from the distractor mean are considered highly salient. Conversely, if the data set has high variance distractors then the same distance will lie within an ellipse of lower standard deviation and the saliency estimate will be lower.

4.6 Scale-Space Pooling

The saliency calculation for a pyramid level is the square root of the sum of squared saliency estimates for each feature. Lower dimensional saliency modules finish before higher dimensional models and their outputs must be delayed to synchronize output. Using queues, the contrast and orientation saliency calculations must be delayed to the color saliency estimate.

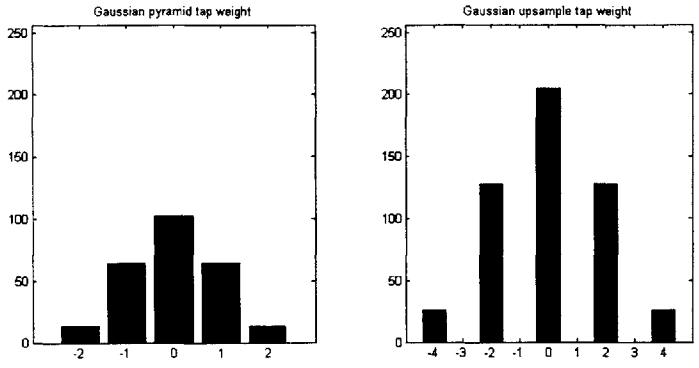


Figure 4-7: Gaussian Upsampling Kernel

Using the three values, the final eight-bit saliency value is calculated for the pixel, at the particular level. Four eight-bit saliency values can fit in the RAM chip for a single address. The second RAM chip is written every fourth cycle with four pixels of saliency calculations.

Before the second and third levels are written to the RAM chip, they are upsampled to the size of the original pyramid level. Upsampling is performed by a modified Gaussin pyramid filter. The Gaussian upsampling kernels are zero for every other tap, twice the magnitude for nonzero taps, and twice as wide. The upsample kernel is shown in Figure 4.6. While the third pyramid level is upsampled, the output is used in the final scale-space pixel saliency estimation. For each pixel, the saliency value for each level is squared and summed. The square root of the sum is the final saliency calculation for the pixel. This value is written to in place of the third pyramid, leaving space for the first two levels to be overwritten by the next frame's calculations. The layout of the saliency RAM is shown in Figure 4.6.

4.7 VGA Output

The saliency values are displayed as grey scale values under the XVGA standard. The display is 1024x768 pixels refreshed at 60MHz. Saliency values are presented in screen coordinates with the origin in the top left. This standard is chosen to be most compatible with display

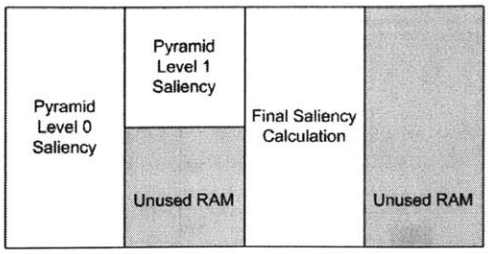


Figure 4-8: Saliency RAM Memory Layout

devices.

4.8 Implementation Summary

This chapter describes the implementation of the streaming statistical saliency visual system. The calculation follows from the statistical saliency equations in Chapter 2 and is possible by the high throughput circuit designs presented in Chapter 3.

Chapter 5

Real-Time Saliency Experiments

The statistical saliency visual system allows for experiments to incorporate time dependent variables. Without encroaching on motion saliency, real-time saliency experiments can suggest augmentations to the algorithm. This chapter discusses two experiments in which the control and probe tests have identical static saliency calculations, but the changes in probes suggest they may be more salient.

The two experiments are variants of the same idea. For a distractor distribution, a given saliency measurement may be caused by different targets. Should the target change to any other state along the same covariance ellipse, the saliency calculation will not change, but subjects may be more attentive to the change. The two experiments describe discrete and continuous changes to the target, respectively, without changing the saliency calculation.

The model is capable of handling these cases. For each experiment, individual solutions are presented that address the specific problem. In the final discussion, a more general partial derivative saliency calculation method is proposed that captures the idea of change saliency.

5.1 Flicker Experiments

Human vision is attentive to flashing and flickering patches of light. Even though no motion may be observed, sudden changes in image brightness, color, pattern, or orientation may have higher saliency than the steady state before and after the change. A flicker experiment can be constructed in which a target alternates between two states at equal distance from the distractors in feature space. It is predicted that, though saliency does not change, the flickering target is considered more salient.

5.1.1 Methods

The control is a target circle of 75% brightness among distractor circles of 50% brightness. The circles all have identical properties except brightness. The probe is an identical arrangement of circles, except the target circle alternates between 75% and 25% brightness. The control and probe are presented on opposite sides of the visual field. Surrounding each group of circles are identical instances of the letter 'T' at 50% brightness. In one of the groups of letter 'T' is a single letter 'L.' The group with the 'L' is randomly selected on for each presentation. Upon presentation, subjects are asked to find the 'L' in the fields of 'T' distractors.

The statistical saliency model predicts that the control and probe have equal saliency and should not influence the search time whether the 'L' is near the control or near the probe.

5.1.2 Results

Because the brightness of the probe target circle transitions discontinuously, the saliency calculation cannot measure the transition. As a result, the circles offer no calculable low level cues in the search. Observations suggest the flickering target will bias search towards the probe, until subjects learn there is no correlation. A bias towards the probe suggests the flickering target is more salient than its distractors.

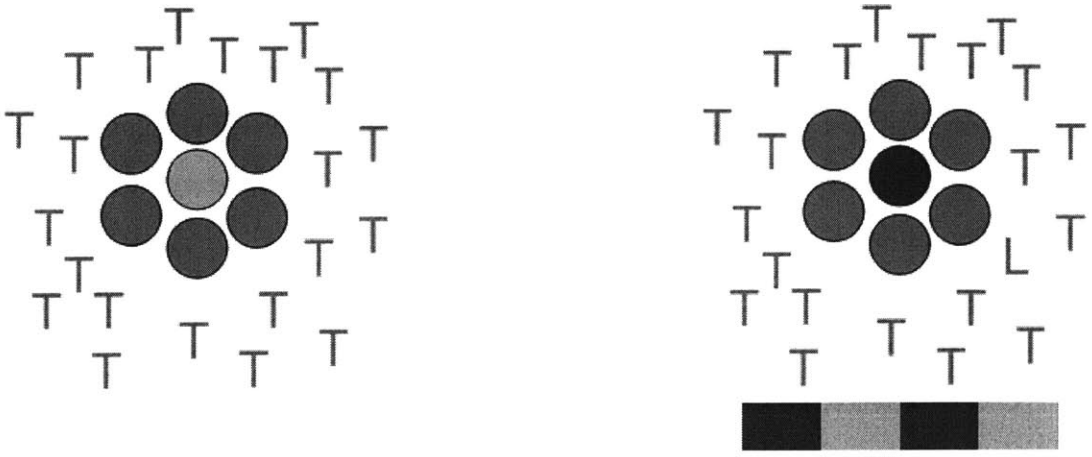


Figure 5-1: The flicker experiment compares search times for the letter 'L' among distractors of the letter 'T' while circles distractors give a irrelevant cue. The center circle on the right alternates between dark and light colors that are equally distant from the distractor colors. If the flickering circle is more salient, search times should be lower when the 'L' is on the right side.

5.1.3 Discussion

In one-dimensional feature spaces, a saliency value can result from two possible states. When targets make discontinuous transitions between the two states, the model can only calculate saliency before and after the change. The change itself may be more salient than either state, but it cannot be measured in a single frame of video.

One possible solution to measuring change is to update saliency values by exponentially decay rather than zero order hold. Two saliency calculations occur per pixel than video updates. Discrete transitions would have a few frames of intermediate saliency before converging to the transitioned state. The rate of decay would affect the persistence of the previous state. Choosing a parameter too high would make intermediate states too close to the transitioned state and a small resulting saliency calculation. Choosing a parameter too low would smooth transitions and filter high frequency transitions, creating inaccurate saliency results.

The example described a saliency ambiguity for brightness, such transitions can occur

in any feature space. The control and probe target circles could have equal size difference from the distractors. They could have oriented patterns equal angular distances from the oriented patterns of distractors. They could exhibit motion in directions equally divergent from distractors. For discrete transitions, exponential decay of the saliency calculation could reveal saliency differences.

5.2 Covariance Ellipse Tangential Change Experiments

The continuous analog to the flicker experiment is the experiment in which the target transitions continuously along the covariance ellipse for the given saliency value. Exponential decay may cause an intermediate salient state between discrete transitions, continuous transitions along the covariance ellipse contour never transition through states of different saliency. Though choice of color-space is controversial, the experiment is best illustrated in color saliency. It is predicted that a target of revolving or pulsating color will be more salient than an unchanging target, even if both targets are equally salient among distractors.

5.2.1 Methods

The control is a target circle of a particular color among distractor circles of varied color. The circles all have identical properties except color. The probe is an identical arrangement of circles, except the target circle color transitions continuously along the covariance ellipse incident with the control color. The control and probe are presented on opposite sides of the visual field. Surrounding each group of circles are identical instances of the letter 'T' with a fixed color. In one of the groups of letter 'T' is a single letter 'L.' The group with the 'L' is randomly selected on for each presentation. Upon presentation, subjects are asked to find the 'L' in the fields of 'T' distractors.

The statistical saliency model predicts that the control and probe have equal saliency and should not influence the search time whether the 'L' is near the control or near the probe.

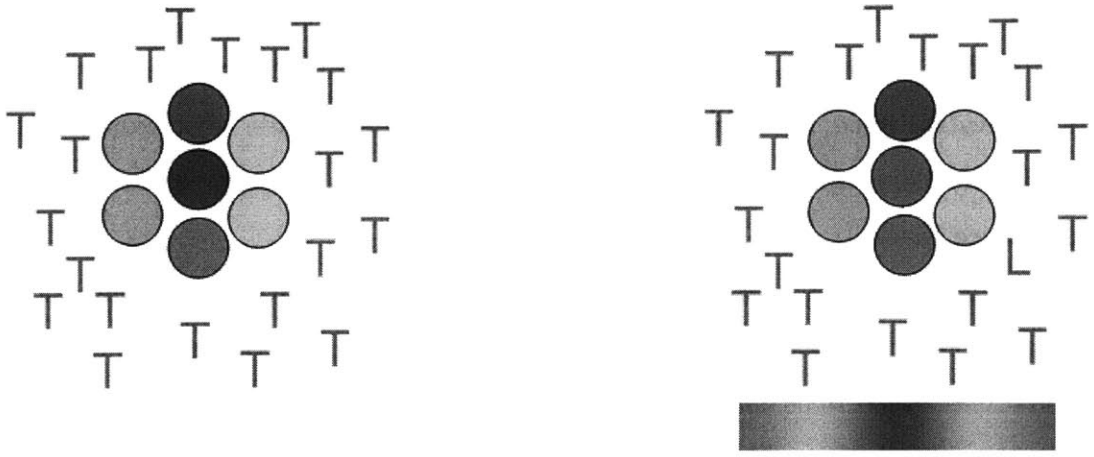


Figure 5-2: Similar to the flicker experiment in Figure 5.1.1, the tangential change experiment compares search times for the letter 'L' among distractors of the letter 'T' while circles distractors give a irrelevant cue. The center circle on the right smoothly transitions to colors that have the same feature distance from thh distractors circles. Under the model, the circle never changes saliency. If the changing circle is actually more salient, search times should be lower when the 'L' is on the right side.

5.2.2 Results

For the same reasons as the flicker experiment, the statistical saliency model predicts the model and probe have equal saliency and search will be unbiased. Observations suggest search is biased towards the changing color. As the speed with which the probe circle changes color increases, the probe circle becomes much more salient. At one extreme, the probe circle appears to either be changing too slowly to be noticed and just as salient as the control. At the other extreme, the probe circle is changing wildly and will, with high probability, be the first point of focus upon presentation.

5.2.3 Discussion

In the continuous transition case, exponential decay will never create intermediate states of different saliency. The only measurable difference between the target and distractors is the rate of change within the feature space. Again, a single frame computation will also be

insufficient to measure changes in time. A solution to the problems of the flicker experiment and the tangential change experiment is a partial derivative saliency computation. The partial derivative must be taken relative to the changing feature. For example, partial derivatives in color may not correctly characterize changes in orientation, though both use the luminance channel.

Because the statistical saliency visual system is modular, additional saliency calculation can be performed in parallel after a module is created to calculate the new feature. Calculating first derivatives does not require any additional memory. Before writing to a memory address, the data at the address must be read and subtracted from the new value. The difference is the approximation to the derivative and the new data is overwrites the old data to repeat the computation in the next frame.

5.3 Overall Discussion

Real-time statistical saliency allows for the exploration of saliency in the time domain. The discussed experiments demonstrate two instances in which the model can easily be augmented to handle time varying situations. Including partial temporal derivatives involves a module to calculate the frame-to-frame differences and an additional instance of one of the saliency calculation modules. Though motion saliency was not discussed, local motion vectors can be computed and added to the saliency computation in a similar manner to derivatives. Easy addition of temporal features make the statistical saliency visual system a valuable tool in testing new feature spaces.

Chapter 6

Concluding Remarks

This thesis presents a real-time saliency analysis tool, implementing the statistical saliency model, and easily extensible to features in the time domain. The ability to stream output comes from the high throughput and parallel processing design of the modules. This tool is designed to work as a stand-alone saliency analysis system or as a tool for testing and evaluating saliency features.

6.1 Contributions

The most important contribution of this work is a system from which statistical saliency information can be calculated online for any live scene. Real-time saliency information can be used in industry to assist designers in reducing clutter, in vision tasks to direct expensive computation to important regions, and in automotive to detect salient objects in driving environments.

Experiments using streaming saliency information helps identify ways to include temporal saliency features in the computation. Simple feature changes can be represented as temporal derivatives and calculated efficiently. Other possible temporal extensions include higher order derivatives and motion analysis. Using the saliency calculation modules, testing the effectiveness of candidate features can be performed rapidly on live data.

The tool is a stand-alone system; with a camera input and monitor output, statistical saliency information can be made available. The design can be implemented on a printed circuit for low power and low cost production. Circuit design benefits from stability of immutability. A working embedded system will always work unless physically damaged.

6.2 Future Work

The design presented is a prototype for streaming visual applications. The first aspect to improve is data input. Other video standards exist that are more flexible and higher quality. A more appropriate method for input is FireWire or Ethernet connection. These methods allow for higher bandwidths and more flexible data formats. Because the saliency calculation modules require only channel input, the method by which the video data arrives can be independent.

A similar improvement can be made for the output. Aside from VGA output, the system could be used for rapid data collection. For this to be possible other forms of output are necessary, namely RS-232 or WiFi. The only changes would be to output circuitry and saliency calculation circuitry would remain unchanged.

Most of the future work is in developing motion saliency features in real time. Though motion estimation is a local calculation, many methods compute motion vectors by finding the best and nearest matching image region. Streaming design is not built to facilitate random access as data is quickly discarded. Real-time motion analysis may require additional memory and different calculation modules. The next steps in real-time statistical saliency calculation will involve motion saliency calculation.

Appendix A

DESIGN DETAILS

A.1 Arithmetic Units

A.1.1 Variable Width Adder.v

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:    12:52:25 01/18/2007
// Design Name:          Variable Width Adder
// Module Name:    adder
// Project Name:   Statistical Saliency
// Target Devices: Xilinx Virtex-II XC2V6000
// Tool versions:  Xilinx ISE Foundation 9.1
// Description:    Ripple Carry Adder with bit with parameter
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
```

```
//
////////////////////////////////////////////////////////////////////////////////

module adder(a, b, sum, add);

parameter BITS=8;
input[BITS:0] a, b;
output[BITS:0] sum;
input add;

wire[BITS:0] a, bb, c, ci,sum;
assign bb = add? b : ~b;
assign ci = {c[BITS-1:0],~add};

assign c[BITS:0]   = (a & bb) | (ci & (a | bb));
assign sum[BITS:0] = a ^ bb ^ ci;

endmodule
```

A.1.2 Ten Bit Booth Recoded Multiplier.v

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:    12:52:25 01/18/2007
// Design Name:          10 Bit Radix-4 Booth Recoded Multiplier
// Module Name:    mult10br4
// Project Name:   Statistical Saliency
// Target Devices: Xilinx Virtex-II XC2V6000
// Tool versions:  Xilinx ISE Foundation 9.1
// Description:    Integer binary radix-4 booth recoded multiplication module
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////

module mult10br4(clk, reset, enable, a, b, p);
//10 bits, booth recoding, radix 4 (pipelined with delay 5)

parameter BITS=9;
parameter BITS2=BITS*2+1;
parameter CYCLES=4;
input clk, reset, enable;
input[BITS:0] a, b;
output[BITS2:0] p;


reg[BITS2+1:0] product[CYCLES:0];
```

```
wire[BITS2:0] p = product[CYCLES][BITS2:0];

reg[BITS:0] m[CYCLES:0];//multiplicand;
reg c[CYCLES:0];                    //carry

wire[BITS+1:0] p1[CYCLES:0];//product +/- 1*multiplicand
wire[BITS+1:0] p2[CYCLES:0];//product +/- 2*multiplicand
wire[BITS+1:0] m1i,m2i;       //initial product

//initialial calculation
adder #(BITS+1) a1i( 11'd0, {a[BITS],a}, m1i, 1'b0);
adder #(BITS+1) a2i( 11'd0, {a,1'd0}, m2i, 1'b0);

adder #(BITS+1) a1a(product[0][BITS2+1:BITS+1],
        {m[0][BITS],m[0]}, p1[0], ~product[0][1]);
adder #(BITS+1) a1b(product[1][BITS2+1:BITS+1],
        {m[1][BITS],m[1]}, p1[1], ~product[1][1]);
adder #(BITS+1) a1c(product[2][BITS2+1:BITS+1],
        {m[2][BITS],m[2]}, p1[2], ~product[2][1]);
adder #(BITS+1) a1d(product[3][BITS2+1:BITS+1],
        {m[3][BITS],m[3]}, p1[3], ~product[3][1]);
adder #(BITS+1) a1e(product[4][BITS2+1:BITS+1],
        {m[4][BITS],m[4]}, p1[4], ~product[4][1]);

adder #(BITS+1) a2a(product[0][BITS2+1:BITS+1],{m[0],1'b0}, p2[0], c[0]);
adder #(BITS+1) a2b(product[1][BITS2+1:BITS+1],{m[1],1'b0}, p2[1], c[1]);
adder #(BITS+1) a2c(product[2][BITS2+1:BITS+1],{m[2],1'b0}, p2[2], c[2]);
adder #(BITS+1) a2d(product[3][BITS2+1:BITS+1],{m[3],1'b0}, p2[3], c[3]);
adder #(BITS+1) a2e(product[4][BITS2+1:BITS+1],{m[4],1'b0}, p2[4], c[4]);

wire[BITS+1:0] init;
assign init =
        b[1:0]==2'b00 ?  11'd0 :
        b[1:0]==2'b01 ?  {a[BITS],a}:
```

```
        b[1:0]==2'b10 ?    m2i:
                                m1i;

integer i;
always@(posedge clk)begin
        if (reset) begin
                for(i=0;i<=CYCLES;i=i+1)begin
                        product[i]<=0;
                        m[i]<=0;
                        c[i]<=0;
                end
        end else if (enable) begin
                product[0]      <= {init[BITS+1],init[BITS+1],init,b[BITS:2]};
                m[0]                    <= {a};
                c[0]                    <= b[1];

                for(i=1;i<=CYCLES;i=i+1) begin
                        case ( {product[i-1][1:0],c[i-1]} )

                         3'b001: product[i]<={p1[i-1][BITS+1],p1[i-1][BITS+1],
                                p1[i-1][BITS+1:0],product[i-1][BITS:2]};
                         3'b010: product[i]<={p1[i-1][BITS+1],p1[i-1][BITS+1],
                                p1[i-1][BITS+1:0],product[i-1][BITS:2]};
                         3'b011: product[i]<={p2[i-1][BITS+1],p2[i-1][BITS+1],
                                p2[i-1][BITS+1:0],product[i-1][BITS:2]};
                         3'b100: product[i]<={p2[i-1][BITS+1],p2[i-1][BITS+1],
                                p2[i-1][BITS+1:0],product[i-1][BITS:2]};
                         3'b101: product[i]<={p1[i-1][BITS+1],p1[i-1][BITS+1],
                                p1[i-1][BITS+1:0],product[i-1][BITS:2]};
                         3'b110: product[i]<={p1[i-1][BITS+1],p1[i-1][BITS+1],
                                p1[i-1][BITS+1:0],product[i-1][BITS:2]};
                         default:product[i]<={product[i-1][BITS2+1],
                                product[i-1][BITS2+1],product[i-1][BITS2+1:2]};
```

```
                endcase

                        c[i]<=product[i-1][1];
                        m[i]<=m[i-1];
                end //for()

        end //if(reset) else if (enable)
end //always@(posedge clk)

endmodule
```

A.1.3 Sixteen Bit Divider.v

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:    12:52:25 01/18/2007
// Design Name:          16 Bit Divider
// Module Name:    div16
// Project Name:   Statistical Saliency
// Target Devices: Xilinx Virtex-II XC2V6000
// Tool versions:  Xilinx ISE Foundation 9.1
// Description:    Integer binary restoring recurence division module
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////

module div16(clk,reset,enable, dividend, divisor, quotient, remainder);

parameter BITS=15;
parameter BITS2=31;
input clk, reset, enable;
input[BITS:0] dividend, divisor;
output[BITS:0] quotient, r;

reg[BITS:0] d[BITS:0];
reg[BITS2:0] qr[BITS+1:0];

wire[BITS:0] quotient, remainder;
```

```
assign {remainder, quotient} = qr[BITS+1];

wire[BITS+1:0] diff[BITS:0];
adder #(BITS+1)  a0( qr[0][BITS2:BITS],{1'd0, d[0]}, diff[0],1'b0);
adder #(BITS+1)  a1( qr[1][BITS2:BITS],{1'd0, d[1]}, diff[1],1'b0);
adder #(BITS+1)  a2( qr[2][BITS2:BITS],{1'd0, d[2]}, diff[2],1'b0);
adder #(BITS+1)  a3( qr[3][BITS2:BITS],{1'd0, d[3]}, diff[3],1'b0);
adder #(BITS+1)  a4( qr[4][BITS2:BITS],{1'd0, d[4]}, diff[4],1'b0);
adder #(BITS+1)  a5( qr[5][BITS2:BITS],{1'd0, d[5]}, diff[5],1'b0);
adder #(BITS+1)  a6( qr[6][BITS2:BITS],{1'd0, d[6]}, diff[6],1'b0);
adder #(BITS+1)  a7( qr[7][BITS2:BITS],{1'd0, d[7]}, diff[7],1'b0);
adder #(BITS+1)  a8( qr[8][BITS2:BITS],{1'd0, d[8]}, diff[8],1'b0);
adder #(BITS+1)  a9( qr[9][BITS2:BITS],{1'd0, d[9]}, diff[9],1'b0);
adder #(BITS+1) a10(qr[10][BITS2:BITS],{1'd0,d[10]},diff[10],1'b0);
adder #(BITS+1) a11(qr[11][BITS2:BITS],{1'd0,d[11]},diff[11],1'b0);
adder #(BITS+1) a12(qr[12][BITS2:BITS],{1'd0,d[12]},diff[12],1'b0);
adder #(BITS+1) a13(qr[13][BITS2:BITS],{1'd0,d[13]},diff[13],1'b0);
adder #(BITS+1) a14(qr[14][BITS2:BITS],{1'd0,d[14]},diff[14],1'b0);
adder #(BITS+1) a15(qr[15][BITS2:BITS],{1'd0,d[15]},diff[15],1'b0);

integer i;
always@(posedge clk)begin
        if(reset) begin
                for(i=0;i<=BITS;i=i+1) begin
                        d[i]<=1;
                        qr[i]<=0;
                end
                qr[BITS+1]<=0;
        end else if(enable) begin
                d[0]<=divisor;
                qr[0]<={8'd0,dividend};

                for(i=1;i<=BITS;i=i+1) begin
                        d[i]<=d[i-1];//pipe intermediate opperations
```

```
                end
                for(i=1;i<=BITS+1;i=i+1) begin
                        if(diff[i-1][BITS+1])//if highest bit is 1 (negative)
                                qr[i]<={qr[i-1][BITS2-1:0],1'd0};
                                //shift remainder and append 0 to quotient
                        else
                                qr[i]<={diff[i-1][BITS:0],qr[i-1][BITS-1:0],1'd1};
                                //replace remainder, 1 to quotient
                end

        end //if(reset) else if (enable)
end //always@(posedge clk)

endmodule
```

A.1.4 Sixteen Bit Square Root.v

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:    17:33:20 03/03/2007
// Design Name:    16 Bit Square Root
// Module Name:    sqrt16
// Project Name:   Statistical Saliency
// Target Devices: Xilinx Virtex-II XC2V6000
// Tool versions:  Xilinx ISE Foundation 9.1
// Description:          Integer binary square root module
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
module sqrt16(clk, reset, enable, sq, r);

parameter BITS=15;
input clk, reset, enable;
input[BITS:0] sq;
output[BITS:0] r;

reg[30:0] rsq[7:0];
reg[7:0] root[7:0];
reg         add[7:0];

wire[16:0] d[7:0];
adder #(BITS+1) add0({rsq[0][30:14]},{7'd0,root[0],add[0],1'b1},d[0],add[0]);
```

```
adder #(BITS+1) add1({rsq[1][30:14]},{7'd0,root[1],add[1],1'b1},d[1],add[1]);
adder #(BITS+1) add2({rsq[2][30:14]},{7'd0,root[2],add[2],1'b1},d[2],add[2]);
adder #(BITS+1) add3({rsq[3][30:14]},{7'd0,root[3],add[3],1'b1},d[3],add[3]);
adder #(BITS+1) add4({rsq[4][30:14]},{7'd0,root[4],add[4],1'b1},d[4],add[4]);
adder #(BITS+1) add5({rsq[5][30:14]},{7'd0,root[5],add[5],1'b1},d[5],add[5]);
adder #(BITS+1) add6({rsq[6][30:14]},{7'd0,root[6],add[6],1'b1},d[6],add[6]);
adder #(BITS+1) add7({rsq[7][30:14]},{7'd0,root[7],add[7],1'b1},d[7],add[7]);

integer i;
always@(posedge clk) begin
        if(reset) begin
                for(i=0; i<=7; i=i+1) begin
                        rsq[i]<=0;
                        root[i]<=0;
                        add[i]<= 0;
                end
        end else if(enable) begin
                rsq[0]<={15'd0,sq};
                root[0]<=0;
                add[0]<=0;

                for(i=1; i<=7; i=i+1) begin
                        rsq[i]<={d[i-1][16],d[i-1][13:0],rsq[i-1][13:0],2'd0};
                        root[i]<={root[i-1][6:0],~d[i-1][16]};
                        add[i]<= d[i-1][16];
                end

        end
end

wire[BITS:0] r = {8'd0,root[7][6:0],~d[7][16]};

endmodule
```

Tap	1	2	3	4	5
Weight	13	63	103	63	13

Table A.1: Gaussian Pyramid kernel

Tap	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
Weight	0	0	0	2	6	15	31	47	54	47	31	15	6	2	0	0	0

Table A.2: Contrast Filter Inner Gaussian kernel

A.2 Kernel Tables

A.3 Saliency Modules

A.3.1 Target-Distractor Estimator

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:    15:30:51 02/16/2007
// Design Name:          Target and Distractor estimator
// Module Name:    targetdistractor
// Project Name:   Statistical Saliency
// Target Devices: Xilinx Virtex-II XC2V6000
// Tool versions:  Xilinx ISE Foundation 9.1
// Description:    Measures approximate difference of pixel from surroundings
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
```

Tap	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
Weight	1	2	5	9	14	21	27	32	34	32	27	21	14	9	5	2	1

Table A.3: Contrast Filter Outer Gaussian kernel

Tap	1	2	3	4	5	6	7	8	9	10	11	12	13
Weight	0	0	0	0	-1	-1	-1	-1	-1	0	0	0	0
	0	0	0	-1	-2	-2	-2	-2	-2	-1	0	0	0
	0	0	0	-1	-2	-3	-4	-3	-2	-1	0	0	0
	0	0	0	-1	-2	-3	-3	-3	-2	-1	0	0	0
	0	0	0	0	0	1	1	1	0	0	0	0	0
	0	0	1	2	3	5	6	5	3	2	1	0	0
	0	0	1	3	5	7	8	7	5	3	1	0	0
	0	0	1	2	3	5	6	5	3	2	1	0	0
	0	0	0	0	0	1	1	1	0	0	0	0	0
	0	0	0	-1	-2	-3	-3	-3	-2	-1	0	0	0
	0	0	0	-1	-2	-3	-4	-3	-2	-1	0	0	0
	0	0	0	-1	-2	-2	-2	-2	-2	-1	0	0	0
	0	0	0	0	-1	-1	-1	-1	-1	0	0	0	0

Table A.4: Orientation Filter Horizontal Difference of Oriented Gaussians kernel

Tap	1	2	3	4	5	6	7	8	9	10	11	12	13
Weight	0	0	0	0	0	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	0	0	0	0	0	0
	0	0	0	0	0	1	1	1	0	0	0	0	0
	0	-1	-1	-1	0	2	3	2	0	-1	-1	-1	0
	-1	-2	-2	-2	0	3	5	3	0	-2	-2	-2	-1
	-1	-2	-3	-3	1	5	7	5	1	-3	-3	-2	-1
	-1	-2	-4	-3	1	6	8	6	1	-3	-4	-2	-1
	-1	-2	-3	-3	1	5	7	5	1	-3	-3	-2	-1
	-1	-2	-2	-2	0	3	5	3	0	-2	-2	-2	-1
	0	-1	-1	-1	0	2	3	2	0	-1	-1	-1	0
	0	0	0	0	0	1	1	1	0	0	0	0	0
	0	0	0	0	0	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	0	0	0	0	0	0

Table A.5: Orientation Filter Vertical Difference of Oriented Gaussians kernel

Tap	1	2	3	4	5	6	7	8	9	10	11	12	13
Weight	0	0	0	0	0	0	0	-1	-1	0	0	0	0
	0	0	0	0	0	0	-1	-1	-1	-1	0	0	0
	0	0	0	0	0	0	-1	-2	-3	-2	-2	0	0
	0	0	0	1	1	1	0	-2	-3	-4	-2	-1	0
	0	0	0	1	3	4	3	0	-3	-3	-3	-1	-1
	0	0	0	1	4	6	6	4	0	-2	-2	-1	-1
	0	-1	-1	0	3	6	8	6	3	0	-1	-1	0
	-1	-1	-2	-2	0	4	6	6	4	1	0	0	0
	-1	-1	-3	-3	-3	0	3	4	3	1	0	0	0
	0	-1	-2	-4	-3	-2	0	1	1	1	0	0	0
	0	0	-2	-2	-3	-2	-1	0	0	0	0	0	0
	0	0	0	-1	-1	-1	-1	0	0	0	0	0	0
	0	0	0	0	-1	-1	0	0	0	0	0	0	0

Table A.6: Orientation Filter Left Diagonal Difference of Oriented Gaussians kernel

Tap	1	2	3	4	5	6	7	8	9	10	11	12	13
Weight	0	0	0	0	-1	-1	0	0	0	0	0	0	0
	0	0	0	-1	-1	-1	-1	0	0	0	0	0	0
	0	0	-2	-2	-3	-2	-1	0	0	0	0	0	0
	0	-1	-2	-4	-3	-2	0	1	1	1	0	0	0
	-1	-1	-3	-3	-3	0	3	4	3	1	0	0	0
	-1	-1	-2	-2	0	4	6	6	4	1	0	0	0
	0	-1	-1	0	3	6	8	6	3	0	-1	-1	0
	0	0	0	1	4	6	6	4	0	-2	-2	-1	-1
	0	0	0	1	3	4	3	0	-3	-3	-3	-1	-1
	0	0	0	1	1	1	0	-2	-3	-4	-2	-1	0
	0	0	0	0	0	0	-1	-2	-3	-2	-2	0	0
	0	0	0	0	0	0	-1	-1	-1	-1	0	0	0
	0	0	0	0	0	0	0	-1	-1	0	0	0	0

Table A.7: Orientation Filter Right Diagonal Difference of Oriented Gaussians kernel

Tap	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Weight	3	7	11	16	21	26	29	30	29	26	21	16	11	7	3

Table A.8: Small Gaussian Filter kernel

Tap	1	2	3	4	5	6	7	8	9	10
Weight	2	2	2	2	2	3	3	3	4	4
Tap	11	12	13	14	15	16	17	18	19	20
Weight	4	5	5	5	6	6	6	6	7	7
Tap	21	22	23	24	25	26	27	28	29	30
Weight	7	7	7	8	8	8	8	8	7	7
Tap	31	32	33	34	35	36	37	38	39	40
Weight	7	7	7	6	6	6	6	5	5	5
Tap	41	42	43	44	45	46	47	48	49	50
Weight	4	4	4	3	3	3	2	2	2	2
Tap	51									
Weight	2									

Table A.9: Large Gaussian Filter kernel

Tap	1	2	3	4	5	6	7	8	9
Weight	26	0	128	0	204	0	128	0	26

Table A.10: Gaussian Upsample Filter kernel

```
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////

module targetdistractor(clk, reset, enable, hin, vin, din,
        hout, vout, target_dout, distractor_dout

        );

parameter WIDTH = 360;
parameter HEIGHT = 243;
parameter DATABITS = 8;
parameter VGABITS = 10;
parameter DATABITS2 = 2*DATABITS+1;
parameter DATADIFF = DATABITS+1;
//buffer length is 25 lines * width + 25 samples + 4 cycles for mult
parameter DELAYBGF = 25*WIDTH;
parameter BUFFERLENGTH = DELAYBGF;
```

```
parameter DELAYLENGTH = 105;

input clk, reset, enable;
input [VGABITS:0] hin, vin;
input [DATABITS:0] din;
output[VGABITS:0] hout, vout;
output[DATABITS:0] target_dout, distractor_dout;

//apply small g filter to input
wire[VGABITS:0] sgf_h, sgf_v;
wire[DATABITS:0] sgf_d;
smallgfilter #(WIDTH,HEIGHT,DATABITS) sgf(clk, reset, enable,
        hin, vin, din,
        sgf_h, sgf_v, sgf_d);

//apply big g filter to small g filter output
wire[VGABITS:0] bgf_h, bgf_v;
wire[DATABITS:0] bgf_d;
biggfilter #(WIDTH,HEIGHT,DATABITS) bgf(clk, reset, enable,
        sgf_h, sgf_v, sgf_d,
        bgf_h, bgf_v, bgf_d);

reg[DATABITS:0] sgf_dd;
wire [10:0] fifosgf_count;
wire fifosgf_full, fifosgf_empty;
wire fifosgf_we = enable && sgf_h<WIDTH && sgf_v<HEIGHT;
wire fifosgf_re = enable &&
        (fifosgf_count>=BUFFERLENGTH-1 ||
        //(sgf_h>=77&&sgf_v>=25) ||
        (sgf_v>=HEIGHT && !fifosgf_empty));
wire [8:0] fifosgf_out;
wire [DATABITS:0] sgfdd = fifosgf_out[DATABITS:0];
wire [8:0] fifosgf_in = {sgf_dd};
fifo2048x9 fifosgf(clk, fifosgf_re, fifosgf_we,
```

```
            fifosgf_in , reset , fifosgf_out ,
            fifosgf_full , fifosgf_empty , fifosgf_count );

reg[DATABITS:0] delay[DELAYLENGTH:0];
integer i;
always@(posedge clk) begin
        if(reset) begin
                sgf_dd<=0;
                delay[0]<=0;
        end else if(enable) begin
                sgf_dd<= sgf_v<HEIGHT? sgf_d : 0;
                delay[0]<=sgfdd;
        end

        for(i=1; i <=DELAYLENGTH; i=i+1) begin
                if(reset) begin
                        delay[i] <= 0;
                end /**/else if(enable) begin
                        delay[i] <= delay[i-1];
                end
        end

end

wire[DATABITS2+2:0] stw;
reg[DATABITS:0] sd, st;
reg[DATABITS:0] sdwd[3:0];
//mult8r4 msd(clk,reset,enable,{8'd1}/*bigk=1.06^1*/,bgf_d,sdw);
mult10br4 mst(clk,reset,enable,{10'd17}/*smlk=.068^17/256*/,
        {delay[DELAYLENGTH-5][DATABITS],delay[DELAYLENGTH-5]},stw);

wire[DATABITS:0] diff;
adder #(DATABITS) a(sd,st,diff,1'd0);
```

```
//assign outputs
reg[VGABITS:0] houtb[4:0], voutb[4:0];
reg[DATABITS:0] target_dout, distractor_dout;
reg[VGABITS:0] hout;
reg[VGABITS:0] vout;
always@(posedge clk) begin
        if(reset) begin
                sd<=0;
                st<=0;
                target_dout <= 0;
                distractor_dout <= 0;
                {houtb[4],houtb[3],houtb[2],houtb[1],houtb[0]} <= 0;
                {voutb[4],voutb[3],voutb[2],voutb[1],voutb[0]} <= 0;
                {sdwd[3],sdwd[2],sdwd[1],sdwd[0]}<=0;
        end else if(enable)begin
                //sd<=sdw[DATABITS:0];//*1/1
                {sd,sdwd[3],sdwd[2],sdwd[1],sdwd[0]}<={sdwd[3],sdwd[2],sdwd[1],sdwd[0],bgf
                st<=stw[16:8];//*17/256
                target_dout <= bgf_h>4&&bgf_h<=WIDTH+4 ? delay[DELAYLENGTH][DATABITS:0] :
                distractor_dout <= bgf_h>4&&bgf_h<=WIDTH+4 ? diff : 0;
                {hout,houtb[4],houtb[3],houtb[2],houtb[1],houtb[0]}
                        <= {houtb[4],houtb[3],houtb[2],houtb[1],houtb[0],bgf_h};
                {vout,voutb[4],voutb[3],voutb[2],voutb[1],voutb[0]}
                        <= {voutb[4],voutb[3],voutb[2],voutb[1],voutb[0],bgf_v};
        end
end

endmodule
```

A.3.2 Pairwise Filter

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:    17:37:33 02/25/2007
// Design Name:    Pairwise Filter
// Module Name:    pairfilter
// Project Name:   Statistical Saliency
// Target Devices: Xilinx Virtex-II XC2V6000
// Tool versions:  Xilinx ISE Foundation 9.1
// Description:    calculates the squared target
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////

module pairfilter(clk, reset, enable,
        hin, vin, a, b,
        hout, vout, dout
        );

parameter WIDTH = 360;
parameter HEIGHT = 243;
parameter DATABITS = 8;
parameter VGABITS = 10;
parameter DATABITS2 = 2*DATABITS+1;
parameter DATADIFF = DATABITS+1;
```

```
input clk, reset, enable;
input [VGABITS:0] hin,vin;
input [DATABITS:0] a, b;
output[VGABITS:0] hout,vout;
output[DATABITS:0] dout;

wire[DATABITS2+2:0] prod;
mult10br4 m(clk,reset,enable,{a[DATABITS],a},{b[DATABITS],b},prod);

reg[VGABITS:0] dhin[3:0];
reg[VGABITS:0] dvin[3:0];
reg[DATABITS:0] d;
always@(posedge clk) begin
        if(reset) begin
                d<=0;
                {dhin[3],dhin[2],dhin[1],dhin[0]}<=0;
                {dvin[3],dvin[2],dvin[1],dvin[0]}<=0;
        end else if(enable) begin
                d<=prod[16:8];
                {dhin[3],dhin[2],dhin[1],dhin[0]}<={dhin[2],dhin[1],dhin[0],hin};
                {dvin[3],dvin[2],dvin[1],dvin[0]}<={dvin[2],dvin[1],dvin[0],vin};
        end
end

wire[VGABITS:0] ho,vo;
wire[DATABITS:0] dd;

smallgfilter #(WIDTH,HEIGHT,DATABITS) sgf(clk,reset,enable,
        dhin[3],dvin[3],d,
        ho,vo,dd);

assign hout[VGABITS:0]=ho[VGABITS:0];
assign vout[VGABITS:0]=vo[VGABITS:0];
assign dout[DATABITS:0]=dd[DATABITS:0];
```

```
endmodule
```

A.3.3 Covariance Matrix Element

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:    15:30:51 02/16/2007
// Design Name:          Covariance Matrix Element
// Module Name:    covmatrix
// Project Name:   Statistical Saliency
// Target Devices: Xilinx Virtex-II XC2V6000
// Tool versions:  Xilinx ISE Foundation 9.1
// Description:    Calculates a single element of a covariance matrix
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
module covmatrix(clk, reset, enable,
        hin, vin, a, b, c, d, e, const,
        hout, vout, elt
        );

parameter WIDTH = 360;
parameter HEIGHT = 243;
parameter DATABITS = 8;
parameter VGABITS = 10;
parameter DATABITS2 = 2*DATABITS+1;
parameter DATADIFF = DATABITS+1;

//expected delays:
```

```
//a, b: start+0
//c    : end-7
//d, e: end-7

input clk, reset, enable;
input [VGABITS:0] hin,vin;
input [DATABITS:0] a, b, c, d, e, const;
output[VGABITS:0] hout,vout;
output[DATABITS:0] elt;

wire[DATABITS2+2:0] abprod, deprod;
mult10br4 mab(clk,reset,enable,{a[DATABITS],a},{b[DATABITS],b},abprod);
mult10br4 mde(clk,reset,enable,{d[DATABITS],d},{e[DATABITS],e},deprod);

reg[VGABITS:0] dhin[4:0];//delay input coords to synch multiplication output
reg[VGABITS:0] dvin[4:0];//to filter input
reg[DATABITS:0] ab, de;//, ded;
always@(posedge clk) begin
        if(reset) begin
                ab <= 0;
                de <= 0;
                {dhin[4],dhin[3],dhin[2],dhin[1],dhin[0]}<=0;
                {dvin[4],dvin[3],dvin[2],dvin[1],dvin[0]}<=0;
        end else if(enable) begin
                ab<=abprod[16:8];
                de<=deprod[16:8];
                {dhin[4],dhin[3],dhin[2],dhin[1],dhin[0]}<={dhin[3],dhin[2],dhin[1],dhin[0
                {dvin[4],dvin[3],dvin[2],dvin[1],dvin[0]}<={dvin[3],dvin[2],dvin[1],dvin[0
        end
end

wire[VGABITS:0] dhoutf, dvoutf;
wire[DATABITS:0] about;
biggfilter #(WIDTH,HEIGHT,DATABITS) bgf(clk,reset,enable,
```

```
        dhin[4],dvin[4],ab,
        dhoutf,dvoutf,about);

reg[DATABITS+1:0] sd, st;
reg[DATABITS+1:0] tmd, demconst, cov;

wire[DATABITS2+2:0] stw;//scale by bigk and smallk
reg[DATABITS+1:0] sdwd[3:0];
//mult10br4 msd(clk,reset,enable,{8'd1}/*bigk=1.06~1*/,{about[DATABITS],about},sdw);
mult10br4 mst(clk,reset,enable,{10'd17}/*smlk=.068~17/256*/,{c[DATABITS],c},stw);

wire[DATABITS+1:0] tmdw;//subtract (scaled) c from ab
adder #(DATABITS+1) atd(sd,st,tmdw,1'b0);
wire[DATABITS+1:0] demconstw;//subtract const from de
adder #(DATABITS+1) adec({de[DATABITS],de},{const[DATABITS],const},demconstw,1'b0);
wire[DATABITS+1:0] covw;//subtract everything so signs work
adder #(DATABITS+1) aall(tmd,demconst,covw,1'b0);

reg[VGABITS:0] dhout[6:0];
reg[VGABITS:0] dvout[6:0];
always@(posedge clk) begin
        if(reset) begin
                {dhout[6],dhout[5],dhout[4],dhout[3],dhout[2],dhout[1],dhout[0]}<=0;
                {dvout[6],dvout[5],dvout[4],dvout[3],dvout[2],dvout[1],dvout[0]}<=0;
                sd<=0;
                st<=0;
                tmd<=0;
                demconst<=0;
                cov<=0;
                {sdwd[3],sdwd[2],sdwd[1],sdwd[0]}<=0;
        end else if(enable) begin
                {dhout[6],dhout[5],dhout[4],dhout[3],dhout[2],dhout[1],dhout[0]}
                        <={dhout[5],dhout[4],dhout[3],dhout[2],dhout[1],dhout[0],dhoutf};
                {dvout[6],dvout[5],dvout[4],dvout[3],dvout[2],dvout[1],dvout[0]}
```

```
                    <={dvout[5],dvout[4],dvout[3],dvout[2],dvout[1],dvout[0],dvoutf};
                //sd<=sdw[DATABITS:0];//*1/1
                {sd,sdwd[3],sdwd[2],sdwd[1],sdwd[0]}<={sdwd[3],sdwd[2],sdwd[1],sdwd[0],{ab
                st<=stw[17:8];//*17/256
                tmd<=tmdw;
                demconst<=demconstw;
                cov<=covw;
        end
end

assign elt = cov[DATABITS:0];
assign hout = dhout[6];
assign vout = dvout[6];

endmodule
```

A.3.4 Determinant

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:    11:34:19 02/10/2007
// Design Name:          3x3 Matrix Determinant
// Module Name:    determinant3x3
// Project Name:   Statistical Saliency
// Target Devices: Xilinx Virtex-II XC2V6000
// Tool versions:  Xilinx ISE Foundation 9.1
// Description:    Cacluclates a symmetric matrix determinant
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
module determinant3x3(clk, reset, enable,
        hin, vin,
        c11, c12, c13, c22, c23, c33,
        hout, vout, det

        );

parameter WIDTH = 360;
parameter HEIGHT = 243;
parameter DATABITS = 8;
parameter VGABITS = 10;
parameter DATABITS2 = 2*DATABITS+1;
parameter DATADIFF = DATABITS+1;
```

```
parameter MULTCYCLES = 4;

input clk, reset, enable;
input [VGABITS:0] hin,vin;
input [DATABITS:0] c11, c12, c13, c22, c23, c33;
output[VGABITS:0] hout,vout;
output[DATABITS:0] det;

vgadelay #(WIDTH,HEIGHT,VGABITS,11,0) vgd(clk,reset,enable,hin,vin,hout,vout);
//5 for first round of mults, 4 for second round of mults, 3 adds

wire[DATABITS2+2:0] m1122,m1212,m1223;
mult10br4 m1a(clk,reset,enable,{c11[DATABITS],c11},{c22[DATABITS],c22},m1122);
mult10br4 m1b(clk,reset,enable,{c12[DATABITS],c12},{c12[DATABITS],c12},m1212);
mult10br4 m1c(clk,reset,enable,{c12[DATABITS],c12},{c23[DATABITS],c23},m1223);

wire[DATABITS2+2:0] m2312,m2313,m2213;
mult10br4 m2a(clk,reset,enable,{c23[DATABITS],c23},{c12[DATABITS],c12},m2312);
mult10br4 m2b(clk,reset,enable,{c23[DATABITS],c23},{c13[DATABITS],c13},m2313);
mult10br4 m2c(clk,reset,enable,{c22[DATABITS],c22},{c13[DATABITS],c13},m2213);

reg[DATABITS:0]    c13d[MULTCYCLES:0];
reg[DATABITS:0]    c33d[MULTCYCLES:0];
reg[DATABITS:0] m2312d[MULTCYCLES:0];
reg[DATABITS:0] m2313d[MULTCYCLES:0];
reg[DATABITS:0] m2213d[MULTCYCLES:0];
reg[DATABITS+1:0] sum1, sum2, sum3, sum13, sum2d;
reg[DATABITS:0] det;

wire[DATABITS2+2:0] m112233,m121233,m122313;
mult10br4 m3a(clk,reset,enable,m1122[17:8],{c33d[MULTCYCLES][DATABITS],c33d[MULTCYCLES]},m
mult10br4 m3b(clk,reset,enable,m1212[17:8],{c33d[MULTCYCLES][DATABITS],c33d[MULTCYCLES]},m
mult10br4 m3c(clk,reset,enable,m1223[17:8],{c13d[MULTCYCLES][DATABITS],c13d[MULTCYCLES]},m
```

```
wire[DATABITS+1:0] sum1w, sum2w, sum3w, sum13w, detw;
adder #(DATABITS+1) a1(m112233[17:8],{m2312d[MULTCYCLES][DATABITS],m2312d[MULTCYCLES]},sum
adder #(DATABITS+1) a2(m121233[17:8],{m2313d[MULTCYCLES][DATABITS],m2313d[MULTCYCLES]},sum
adder #(DATABITS+1) a3(m122313[17:8],{m2213d[MULTCYCLES][DATABITS],m2213d[MULTCYCLES]},sum

adder #(DATABITS+1)  a13(sum1, sum3, sum13w,1'b1);
adder #(DATABITS+1) a123(sum13,sum2d,detw,  1'b0);//maybe reverse

integer i;
always@(posedge clk) begin
        if(reset) begin
                for(i=0;i<=MULTCYCLES;i=i+1) begin
                        c13d[i]<= 0;
                        c33d[i]<= 0;
                      m2312d[i]<= 0;
                      m2313d[i]<= 0;
                      m2213d[i]<= 0;
                end

                sum1    <= 0;
                sum2    <= 0;
                sum3    <= 0;
                sum13   <= 0;
                sum2d <= 0;
                det     <= 0;

        end else if(enable) begin
                c13d[0]<= c13;
                c33d[0]<= c33;
                m2312d[0]<= m2312[17:8];
                m2313d[0]<= m2313[17:8];
                m2213d[0]<= m2213[17:8];
                for(i=1;i<=MULTCYCLES;i=i+1) begin
                        c13d[i]<=   c13d[i-1];
```

```
                    c33d[i]<=    c33d[i-1];
                  m2312d[i]<= m2312d[i-1];
                  m2313d[i]<= m2313d[i-1];
                  m2213d[i]<= m2213d[i-1];
            end

            sum1    <= sum1w;
            sum2    <= sum2w;
            sum3    <= sum3w;
            sum13   <= sum13w;
            sum2d <= sum2;
            det     <= detw[DATABITS:0];

        end
end

endmodule
```

A.3.5 Matrix Inverse

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:    19:20:10 02/17/2007
// Design Name:          Matrix Inverse Element
// Module Name:    matinvert
// Project Name:   Statistical Saliency
// Target Devices: Xilinx Virtex-II XC2V6000
// Tool versions:  Xilinx ISE Foundation 9.1
// Description:    Computes an element of a symmetric matrix inverse
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
module matinvert(clk, reset, enable,
        hin, vin,
        a, b, c, d, det,
        hout, vout, elt

        );

parameter WIDTH = 360;
parameter HEIGHT = 243;
parameter DATABITS = 8;
parameter VGABITS = 10;
parameter DATABITS2 = 2*DATABITS+1;
parameter DATADIFF = DATABITS+1;
```

```
parameter DIVCYCLES = 7;

input clk, reset, enable;
input [VGABITS:0] hin,vin;
input [DATABITS:0] a, b, c, d, det;
output[VGABITS:0] hout,vout;
output[DATABITS:0] elt;

vgadelay #(WIDTH,HEIGHT,VGABITS,15,0) vgd(clk,reset,enable,hin,vin,hout,vout);
//5 mult, 1 add, 8 div

reg[DATABITS+1:0] ab,cd,s;
wire[DATABITS2+2:0] abw;
mult10br4 mab(clk,reset,enable,{a[DATABITS],a},{b[DATABITS],b},abw);
wire[DATABITS2+2:0] cdw;
mult10br4 mcd(clk,reset,enable,{c[DATABITS],c},{d[DATABITS],d},cdw);
wire[DATABITS+1:0] sw,nsw;
adder #(DATABITS+1) add(ab, cd, sw,1'b1);
adder #(DATABITS+1) answ(10'd0,sw,nsw,1'b0);

wire[DATABITS:0] ndetd4;
reg[DATABITS:0] detd[5:0];

adder #(DATABITS) andet(9'd0,detd[4],ndetd4,1'b0);

reg[DIVCYCLES:0] neg;
reg[DATABITS:0] elt;

wire[7:0] inv, rem;
div8 div(clk,reset,enable,s[7:0],detd[5]==0?8'd255:detd[5][7:0],inv,rem);
wire[DATABITS:0] ninv;
adder #(DATABITS) aninv(9'd0,{1'd0,inv},ninv,1'b0);

always@(posedge clk) begin
```

```
        if(reset) begin
                ab<=0;
                cd<=0;
                s <=0;
                detd[0]<=1;
                detd[1]<=1;
                detd[2]<=1;
                detd[3]<=1;
                detd[4]<=1;
                detd[5]<=1;
                neg<=0;
                elt<=0;
        end else if(enable) begin
                neg[DIVCYCLES:0]<={neg[DIVCYCLES-1:0],(sw[DATABITS+1]^detd[4][DATABITS])&&
                ab      <=abw[17:8];
                cd      <=cdw[17:8];
                s       <=sw[DATABITS+1]?nsw:sw;
                {detd[5],detd[4],detd[3],detd[2],detd[1],detd[0]}<=
                        {detd[4][DATABITS]?ndetd4:detd[4],detd[3],detd[2],detd[1],detd[0],
                elt<=neg[DIVCYCLES]?ninv:{1'd0,inv};
        end
end

endmodule
```

A.3.6 Saliency Calculation

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:    12:01:25 02/15/2007
// Design Name:          Saliency Calculation for 3 dimensional features
// Module Name:    saliencycalc3
// Project Name:   Statistical Saliency
// Target Devices: Xilinx Virtex-II XC2V6000
// Tool versions:  Xilinx ISE Foundation 9.1
// Description:    (t-d)^T Sigma^{-1} (t-d)
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////

module saliencycalc3(clk, reset, enable,
        hin, vin,
        ic11,ic12,ic13,ic22,ic23,ic33,
        tl,dl,ta,da,tb,db,
        hout, vout, s);

parameter WIDTH = 360;
parameter HEIGHT = 243;
parameter DATABITS = 8;
parameter VGABITS = 10;
parameter DATABITS2 = 2*DATABITS+1;
parameter DATADIFF = DATABITS+1;
```

```
input clk, reset, enable;
input [VGABITS:0] hin,vin;
input [DATABITS:0] ic11,ic12,ic13,ic22,ic23,ic33;
input [DATABITS:0] tl,dl,ta,da,tb,db;
output[VGABITS:0] hout,vout;
output[DATABITS:0] s;

vgadelay #(WIDTH,HEIGHT,VGABITS,19,0) vgd(clk,reset,enable,hin,vin,hout,vout);
//1 add, 4 mult, 1 add,1 add,1 add,1 add,

wire[DATABITS+1:0] lw,aw,bw;
adder #(DATABITS+1) al({tl[DATABITS],tl},{dl[DATABITS],dl},lw,1'b0);
adder #(DATABITS+1) aa({ta[DATABITS],ta},{da[DATABITS],da},aw,1'b0);
adder #(DATABITS+1) ab({tb[DATABITS],tb},{db[DATABITS],db},bw,1'b0);

reg[DATABITS:0] r11,r12,r13,r22,r23,r33;

reg[DATABITS+1:0] l,a,b;
reg[DATABITS+1:0] ld[5:0],ad[5:0],bd[5:0];
always@(posedge clk) begin
        if(reset) begin
                l<=0;
                a<=0;
                b<=0;

                r11<=0;
                r12<=0;
                r13<=0;
                r22<=0;
                r23<=0;
                r33<=0;

                ld[0]<=0;ad[0]<=0;bd[0]<=0;
```

```
            ld[1]<=0;ad[1]<=0;bd[1]<=0;
            ld[2]<=0;ad[2]<=0;bd[2]<=0;
            ld[3]<=0;ad[3]<=0;bd[3]<=0;
            ld[4]<=0;ad[4]<=0;bd[4]<=0;
            ld[5]<=0;ad[5]<=0;bd[5]<=0;
        end else if(enable) begin
            l<=lw;
            a<=aw;
            b<=bw;

            r11<=ic11;
            r12<=ic12;
            r13<=ic13;
            r22<=ic22;
            r23<=ic23;
            r33<=ic33;

            ld[0]<=l;          ad[0]<=a;          bd[0]<=b;
            ld[1]<=ld[0];ad[1]<=ad[0];bd[1]<=bd[0];
            ld[2]<=ld[1];ad[2]<=ad[1];bd[2]<=bd[1];
            ld[3]<=ld[2];ad[3]<=ad[2];bd[3]<=bd[2];
            ld[4]<=ld[3];ad[4]<=ad[3];bd[4]<=bd[3];
            ld[5]<=ld[4];ad[5]<=ad[4];bd[5]<=bd[4];
        end
end

wire[DATABITS2+2:0] l1,l2,l3;
mult10br4 m11(clk,reset,enable,l,{r11[DATABITS],r11},l1);
mult10br4 m12(clk,reset,enable,l,{r12[DATABITS],r12},l2);
mult10br4 m13(clk,reset,enable,l,{r13[DATABITS],r13},l3);

wire[DATABITS2+2:0] a1,a2,a3;
mult10br4 m21(clk,reset,enable,a,{r12[DATABITS],r12},a1);
mult10br4 m22(clk,reset,enable,a,{r22[DATABITS],r22},a2);
```

```
mult10br4 m23(clk,reset,enable,a,{r23[DATABITS],r23},a3);

wire[DATABITS2+2:0] b1,b2,b3;
mult10br4 m31(clk,reset,enable,b,{r13[DATABITS],r13},b1);
mult10br4 m32(clk,reset,enable,b,{r23[DATABITS],r23},b2);
mult10br4 m33(clk,reset,enable,b,{r33[DATABITS],r33},b3);

wire[DATABITS+1:0] la1w, la2w, la3w;
adder #(DATABITS+1) ala1(l1[17:8],a1[17:8],la1w,1'b1);
adder #(DATABITS+1) ala2(l2[17:8],a2[17:8],la2w,1'b1);
adder #(DATABITS+1) ala3(l3[17:8],a3[17:8],la3w,1'b1);

reg[DATABITS+1:0] la1, la2, la3, b1d, b2d, b3d;
always@(posedge clk) begin
        if(reset) begin
                la1 <=0;
                la2 <=0;
                la3 <=0;
                b1d <=0;
                b2d <=0;
                b3d <=0;
        end else if(enable) begin
                la1 <=la1w;
                la2 <=la2w;
                la3 <=la3w;
                b1d <=b1;
                b2d <=b2;
                b3d <=b3;
        end
end

wire[DATABITS+1:0] lab1w, lab2w, lab3w;
adder #(DATABITS+1) alab1(la1,b1d,lab1w,1'b1);
adder #(DATABITS+1) alab2(la2,b2d,lab2w,1'b1);
```

```
adder #(DATABITS+1) alab3(la3,b3d,lab3w,1'b1);

reg[DATABITS+1:0] lab1, lab2, lab3;
always@(posedge clk) begin
        if(reset) begin
                lab1 <=0;
                lab2 <=0;
                lab3 <=0;
        end else if(enable) begin
                lab1 <=lab1w;
                lab2 <=lab2w;
                lab3 <=lab3w;
        end
end

wire[DATABITS2+2:0] lab1lw,lab2aw,lab3bw;
mult10br4 ml(clk,reset,enable,lab1,ld[5],lab1lw);
mult10br4 ma(clk,reset,enable,lab2,ad[5],lab2aw);
mult10br4 mb(clk,reset,enable,lab3,bd[5],lab3bw);
reg[DATABITS2+2:0] lab1l,lab2a,lab3b;
always@(posedge clk) begin
        if(reset) begin
                lab1l <=0;
                lab2a <=0;
                lab3b <=0;
        end else if(enable) begin
                lab1l <=lab1lw;
                lab2a <=lab2aw;
                lab3b <=lab3bw;
        end
end
```

```
wire[DATABITS+1:0] lab1l2aw, sumw;
reg[DATABITS+1:0] lab1l2a, lab3bd, sum;
adder #(DATABITS+1)  alab12(lab1l[17:8],lab2a[17:8],lab1l2aw,1'b1);
adder #(DATABITS+1) alab123(lab1l2a,lab3bd,sumw,1'b1);
always@(posedge clk) begin
        if(reset) begin
                lab1l2a <=0;
                lab3bd <=0;
                sum <=0;
        end else if(enable) begin
                lab1l2a <= lab1l2aw;
                lab3bd <= lab3b[17:8];
                sum <= sumw;
        end
end

reg[DATABITS:0] s;
wire[7:0] sw;
sqrt8 scalc(clk,reset,enable,sum[7:0],sw);
always@(posedge clk) begin
        if(reset) begin
                s <=0;
        end else if(enable) begin
                s<={1'd0,sw};
        end
end

endmodule
```

Bibliography

[1] Ivan Flores. *The Logic of Computer Arithmetic.* Prentice Hall, Englewood Cliffs, NJ, 1963.

[2] Robert E. Goldschmidt. Applications of division by convergence. Master's thesis, M.I.T., 1964.

[3] David Patterson and John Hennessy. *Computer Organization and Design: The Hardware/Software Interface.* Morgan Kaufmann Publishers, San Francisco, Californi, second edition edition, 1998.

[4] W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling. *Numerical Recipes in C: The Art of Scientific Computing.* Cambridge University Press, 1992.

[5] Ruth Rosenholtz. A simple saliency model predicts a number of motion popout phenomena. *Vision Research*, (39):3157–3163, 1999.

[6] Ruth Rosenholtz. Search asymmetries? what search asymmetries? *Perception & Psychophysics*, 3(63):476–489, 2001.

[7] Ruth Rosenholtz. Visual search for orientation among heterogeneous distractors: Experimental results and implications for signal detection theory models of search. *J. Experimental Psychology*, 4(27):985–999, 2001.

[8] Treisman. Features and objects: The 14th bartlett memorial lecture. *Quarterly Journal of Experimental Psychology*, A(40):201–237, 1988.