University of Southern California
Viterbi School of Engineering

EE 454 Milestone 4

Portable Ultrasound Final Report

Convex Hull Deployed on a System-on-Chip for
Small, Affordable Ultrasound Devices

Colin Cammarano, Jesus Garcia, Kevin Jiang, Tsung-Han Sher

2015 Dec 15

ABSTRACT

Portable ultrasound devices are still a rare technology in the rapidly growing field of biomedical engineering. Due to an ever increasing need for improved medical technology, there is an incentive to develop a simple, functional, portable, and low-cost alternative these larger devices. The quickhull algorithm, a divide and conquer algorithm for finding convex hulls on a two-dimensional cartesian plane, is designed to be deployed on a system-on-chip. For preliminary results the quickhull algorithm was implemented in scheme as a validation system; a small literature review was also conducted to understand the current ultrasound technologies. The quickhull algorithm was then implemented in Verilog and parallelized in Modelsim. The algorithm was then optimized for faster runtime.

INTRODUCTION

In the medical field, doctors rely on the use of accurate diagnostic tools in order to properly identify diseases and other medical issues within patients. Ultrasound imaging technology is one of the most ubiquitous diagnostic tools in the biomedical industry. These imaging devices operate by transmitting a high-frequency sound which bounces off an object and back to the imaging device which then processes the received sound waves. By calculating the time between transmission and reception, the object's relative distance to the imaging device can be calculated and an image of the object can be produced. A sound wave in the ultrasound wave is considered to have a frequency higher than 20KHz, but usually, medical ultrasound imaging typically uses sounds above the 10MHz range. Since sound waves travel through nearly any medium, ultrasound devices can be used to detect hidden or concealed objects. For example, ultrasound imaging is used by obstetricians to monitor the development of fetuses, which allows them to diagnose any potential medical conditions that the fetus or the mother might have.

One of the biggest challenges with ultrasound devices is that they are typically large and immobile. In large medical centers, ultrasound devices are often mounted on wheeled bases to allow the medical staff to move the device between hospital rooms. Such devices also require a significant amount of power and thus cannot typically be used outside of a hospital setting. Moreover, these ultrasound devices are prohibitively expensive, often costing upwards of $80,000 USD.

The purpose of this project is to design a portable and low cost alternative to the current medical ultrasound devices available on the market. This design would be a supplemental ultrasound device that can be easily carried and deployed in a variety of environments where it is difficult to provide access to ultrasound technology. Since the project relies on processing sound input into images, an algorithm that can construct geometry around a collection of data points, such as convex hull, must be implemented in the design. For this project, a parallelized implementation of quickhull will be implemented to process sound information into a visible image. Such a design can be implemented with a combination of Stateflow and Verilog and can be simulated on an FPGA board.

PREVIOUS WORK

Medical Ultrasound Imaging Device

The standard issue ultrasound imaging device in modern hospitals are high-performance and high-resolution imaging devices. These high-end machines allow for accurate streaming of internal body structures such as tendons, muscles, joints, arteries and vessels, and organs. The standard issue ultrasound machines cost between 20,000 and 90,000 USD. These powerful ultrasound devices used in hospital for medical uses have been used for more than fifty years, and therefore due to the maturity of this technology, medical professionals have relied on this system.

However for our research project, our goal is an affordable and portable ultrasound device. These two criterias: affordable and portable, are not met with these standard-issue ultrasound machines. The technology we are interested in is new and innovative, a completely different paradigm than the traditional ultrasound device.

Smartphone Ultrasound: MobiUS SP1 System by MobiSante

The Mobi-Sante MobiUS SP1 System is a smartphone-based architecture. It uses a transducer to sample for image data, and sends the raw data via wireless connection to a remote server in order to resolve the image.

The MobiUS SP1 sizes 5.1 inches by 2.76 inches by 0.5 inches and weighs 11.6 ounces. The image resolution is up to 480 by 480 pixels, and has a battery life of 60 minutes of continuous scan time. These specifications are ideal for a portable ultrasound device.

Priced at 100 USD, the MobiUS SP1 has been used and accoladed by many doctors in clinical references. However, our research is interested in a self-contained ultrasound device that does not require the dependence on a remote server for image resolution. This is because there are areas, such as disaster sites and remote communities, in need of ultrasound technology that does not have a connection to wireless services. Our project aims to have specifications similar to the MobiUS SP1 system, however prioritizes a self-contained system over optimal specifications.

M-Turbo Human by SonoSite

The M-Turbo Human ultrasound system is a versatile and durable ultrasound system resembling that of a laptop computer. Not only is it a self-contained high-resolution ultrasound device, it also supports a wide range of peripherals.

This device is much larger and heavier than the MobiUS SP1 system, meaning not optimal for our research. Also its support for a large number of peripherals is unnecessary for our goal in mind. Moreover, its price is around 15,000 USD, which is too expensive for a portable ultrasound device.

PROJECT APPROACH

Project Goal

This project was conceived with the goal of developing an affordable and portable ultrasound device that can be used used in the field environment. This project's goal was not intended to replace the current standard-issue ultrasound device, as that technology is already very mature for its field, but instead be a self-contained ultrasound device used outside of the modern medical facility.

Difficulties and Challenges

Since our goal of this project is to develop an affordable and portable ultrasound device, we will need to sacrifice performance in order to reduce cost and size. We expect to develop a system that has a lower resolution and sample rate. Also due to using a convex hull algorithm, our results may also have lower detail resolution.

Novelty

This project takes a novel approach to developing an effective, efficient, and cheap portable ultrasound device. The device itself is dependent on our parallelized implementation of quickhull. Using data parallelization (splitting the data and dispatching it to several independent processing elements) is something not done in other implementations.

Project Characterization

Our initial design is to have a transducer that generates ultrasound to sample the environment, of which the echos of these ultrasound waves would be received by a microphone with pass filters. This raw data would then be stored in memory and sorted by distance thresholds, and each data set of the thresholds would then be passed into the convex hull dedicated hardware to compute the convex hulls, and finally each convex hull would be outputted on a display as the image. A short task graph is given in figure 1.

Figure 1: Task graph of initial design

Algorithm Implementation

Quickhull in Scheme

Scheme is a functional, procedural programming language. Since quickhull is a tail-recursive algorithm, scheme was a very strong candidate for implementing quickhull for preliminary results. The Scheme implementation is given in Appendix I.

In addition scheme has a built-in drawing library, which allowed us to quickly visualize the quickhull points. Figure 2 displays a sample output of the Scheme implementation.

Figure 2: Convex hull example output in Scheme

Quickhull in Verilog Overview

The finite state machine for the quickhull implementation in verilog consists of six states: INITIAL, FIND_MAX_MIN, HULL_START, CROSS, HULL_RECURSE, and END. Below is an outline of each state's logic. Refer to figure 3 for the state machine design. The full verilog implementation is given in Appendix II.

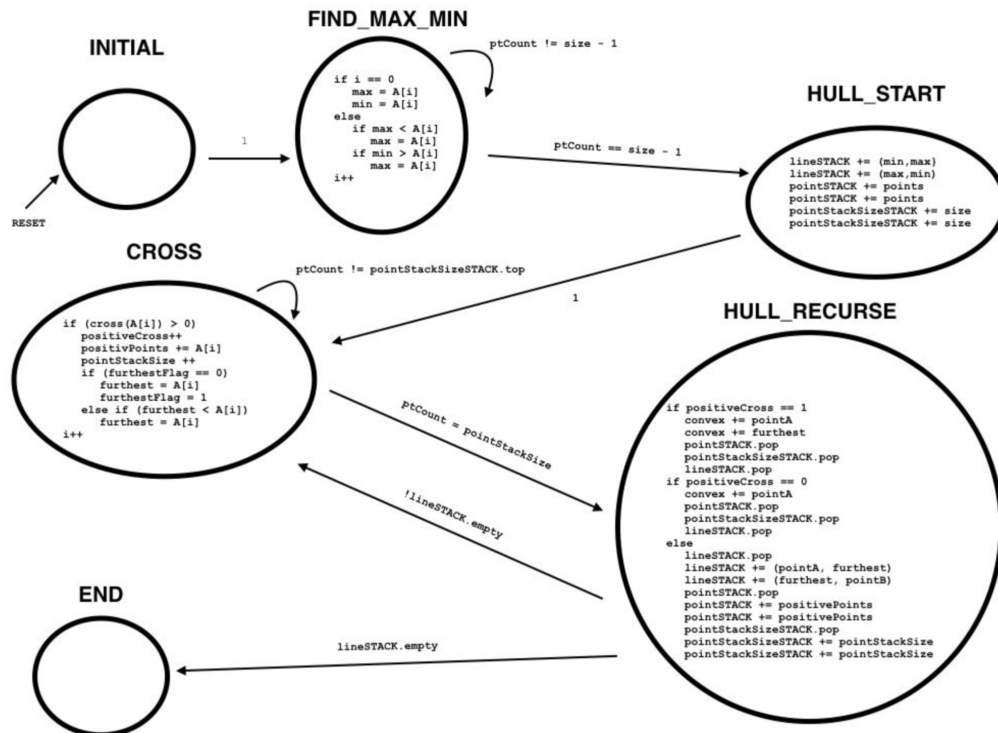


Figure 3: Finite State Machine of Quickhull in Verilog

INITIAL: Initializes all the variables, stacks, and counters to its initial values. It transitions to the next state unconditionally

FIND_MAX_MIN: Find the points with the largest and smallest x value in the set of points given for the convex hull. It transitions to the next state after all the points in the set has been evaluated once.

HULL_START: Places the initial two lines from the minimum to the maximum point and vice versa onto the line stack to initialize the quickhull evaluations. This also places all the points of the set onto the line stack as well as the number of points for each set. It transitions to the next state unconditionally.

CROSS: In this state it finds the point furthest from line at the top of the stack, as well as count the number of points that has a positive cross value with the line and keeps track of all the points that does. It will transition to the next state once every single point of the set at the top of the point stack has been evaluated.

HULL_RECURSE: If the number of points with a positive cross value is 0 or 1, it will update the convex hull points as well as pop all the stacks. If the number of cross values is larger than 1, then the

next set of lines will be placed on the line stack as well as all the points with a positive cross values on the point stack. It will return back to the CROSS state as long as the line stack is not empty, otherwise it will go to the ends state.

END: The end of the algorithm after the convex hull has been found.

A short example of quickhull is given in appendix III.

Quickhull Runtime

The original implementation had a runtime of $O(n^2)$ due to each CROSS state running the entire set of input points every single time. This is not a concern as long as our input set size does not become significantly large enough to impact the overall performance. However since we are already sacrificing performance for affordability and portability, a faster runtime was necessary. The $O(nlogn)$ algorithm, as described above, utilizes two more stacks, each the same size as the line stack (4096 bits). This allows for the implementation to evaluate the correct subset of points each CROSS state. This requires a constant increase of memory, which is a immensely beneficial tradeoff to improving the runtime.

Quickhull Results

In order to accurately gauge the performance of the portable ultrasound design in the above figure, our device was implemented in Verilog HDL and simulated in Modelsim. Initially, we implemented only a single processor in order test the our projected device's per-core performance, the results of which are shown in the following figure. Initial results were promising, showing a linear increase in runtime as the data set increased linearly in size. The Modelsim simulation of this core in the following figure also showed nominal operation when computing the convex hull of a 256 point set, which is the largest set that any single processing element in the device can compute. The results of our tests are given in table 1 and the waveform of the same results are given in figure 4.

Convex Set Size	Point Range	Runtime
16	0 - 31	4100ns
32	0 - 31	10200ns
64	0 - 31	22500ns
256	0 - 31	77900ns
16	0 - 63	4800ns
32	0 - 63	10300ns
64	0 - 63	20000ns
256	0 - 63	109000ns

Table 1: Single core computation times.

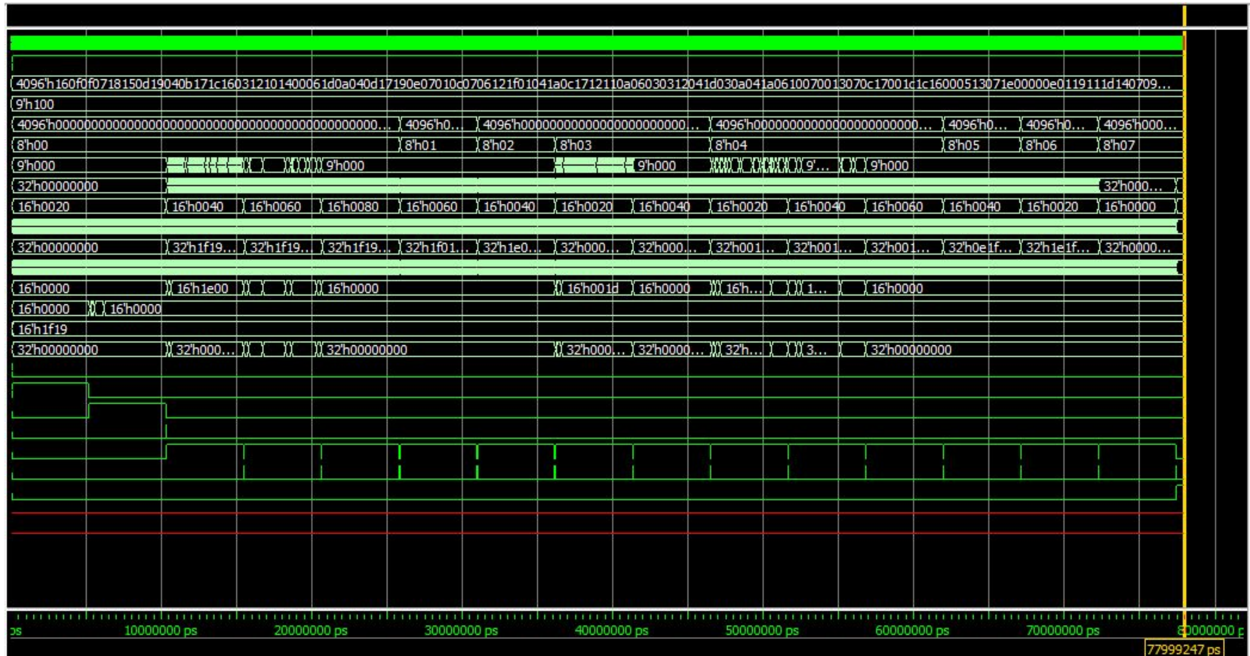


Figure 4: Modelsim waveform of single processing unit.

For milestone 4, a parallelized and multiprocessor solution was developed and implemented in Verilog HDL, and again, simulated in Modelsim. This time, the device was configured to have 8 processors computing the total convex hull in parallel. The results of this simulation is given in figures 5 and 6.

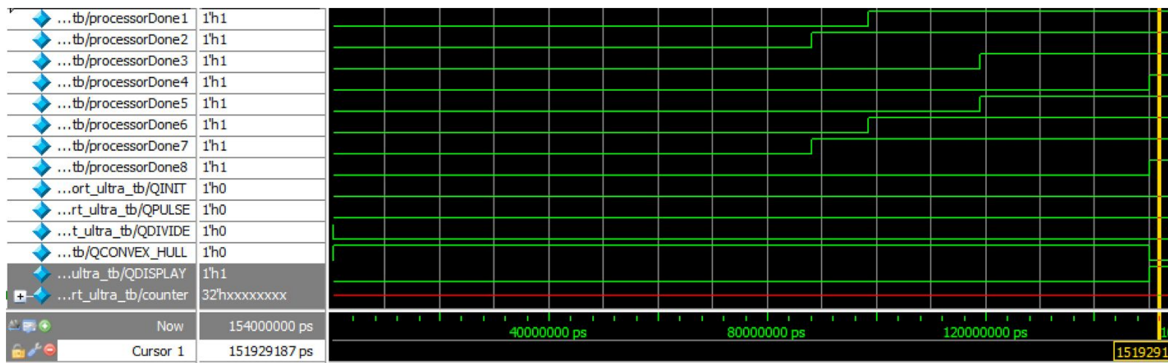


Figure 5: Processor execution times.

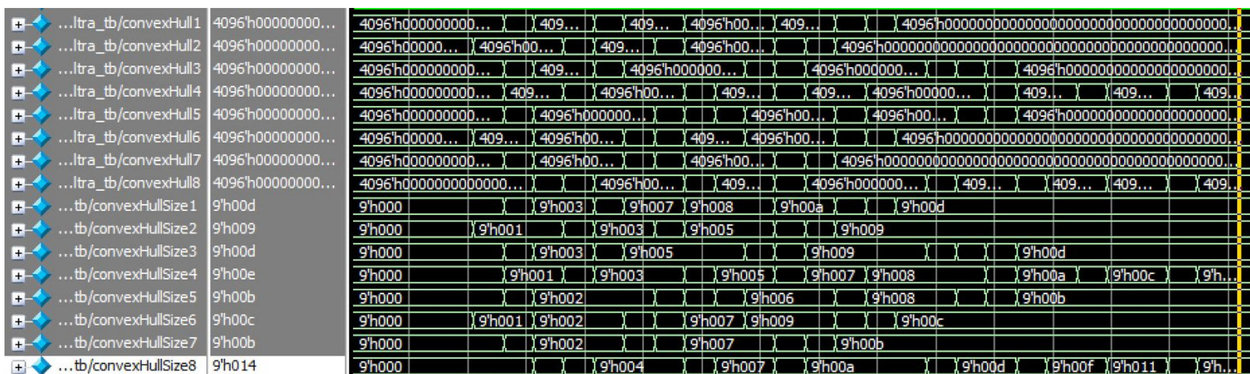


Figure 6: Convex hull computation results.

The multiprocessor implementation of our portable ultrasound device performs as expected. Given that each core has to compute a different set of 256 points, they will finish processing at different times. Overall, the per-core performance meets expectations.

Architecture

Due to the large number of points being processed upon, this project favored an architecture that split the points into subsets and delegated each subset to an individual core. Since processing a given subset of points does not depend upon any other subset of points, more processors can be used to scale the number of points in total. The application task graph of our architecture is given in figure 7.

In this project, the same core that divided the overall cloud of points also merged the sub-clouds after processing. An alternative considered was to have each slave core output its sub-cloud to the display output directly. If the scanning frequency over each slave core was fast enough, the image displayed would appear whole. However, this architecture was scrapped for several reasons:

(1) Delay between slave cores and display output could not be guaranteed to be equal. Some cores would inevitably be farther from the display output, causing unequal delays that could lead to suboptimal image quality.
(2) Following on (1), ensuring a fast enough scanning clock with unequally delayed slave cores would be challenging in terms of timing implementation.
(3) The image would be unpolished and still contain every edge from the sub-clouds, some of which would not be considered edges in the overall cloud. The overall image would be reminiscent of cracked glass.
(4) The master core would not be utilized fully since it would only be dividing and not merging clouds.

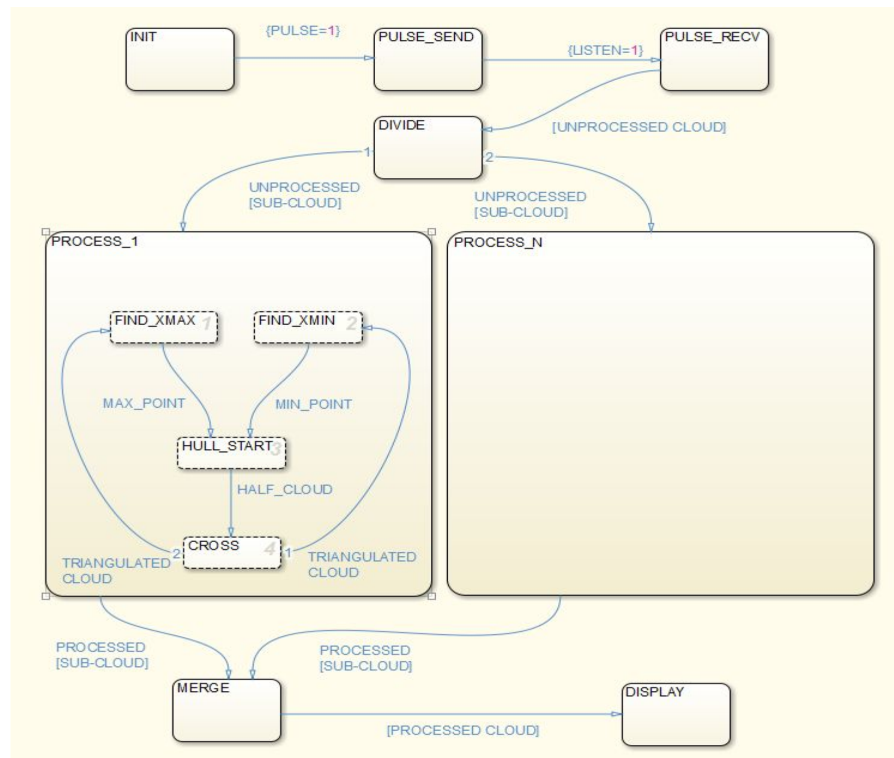


Figure 7: Application Task Graph

Under the chosen architecture, the delay to the display output is the worst delay of all the slave cores. This would lead to reduced frame rate with the benefit of greater image polygon quality. This was judged to be a good tradeoff to compensate slightly for other design decisions that prioritized performance over image quality. Displaying an image that was polished and complete was judged to be a minimum requirement for medical field work. Our network characterization graph is given in figure 8.

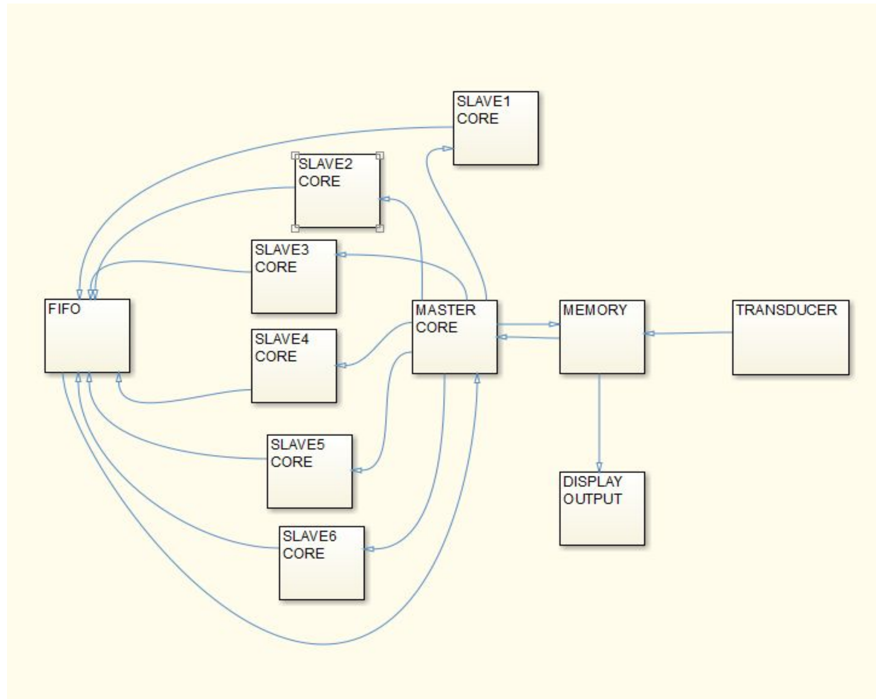


Figure 8: Network Characterization Graph

Parallelizing Multiple Quickhull Instances & Overall Finite State Machine

One of major aspects of this project was to develop a parallelized implementation of the quickhull algorithm in hardware. Once the single core Verilog implementation had been completed and tested, an overarching device finite state machine was implemented in Verilog. This device would accept a large input, representing the data collected by the transducer, and store it as a massive set of points. Since quickhull does not particularly lend itself to parallelization, the data itself needed to be computed concurrently. This data was then processed by a processing element known as the dispatcher, which separated out the large set into several smaller convex sets of points. These sets were then dispatched to their own processors; by doing this, we could parallelize our quickhull computations. Once the overall finite state machine had been designed, the single core Verilog code was modified so that it would support a parallelized implementation. To this end, a processor array module was implemented in Verilog. This module served as a controller that managed each processor during the device's operation. This processor array also contained the dispatcher, which, as previously mentioned, allowed for parallelized data. The overall device waveform is given in figure 9.

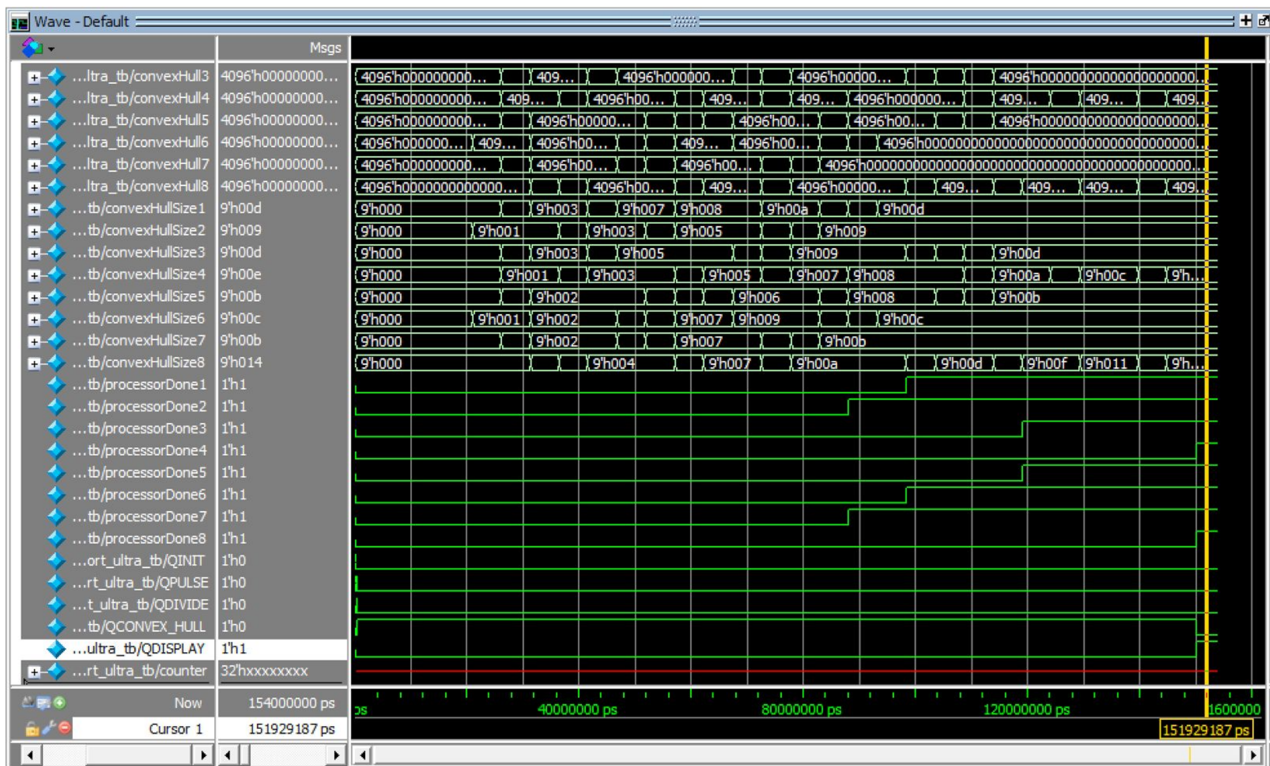


Figure 9: Overall device waveform.

CONCLUSION

Implementing a parallelized quickhull for software and hardware proved to be a success. We were able to create a software implementation of quickhull using Scheme. We were also able to implement a successful single processor for quickhull which showed nominal runtime when computing a convex hull for a dataset of 256 coordinate points as well as a parallelized 8-processor implementation of quickhull in Verilog which also performed as expected. Our results demonstrate the possibility of creating a more computationally efficient ultrasound machine that can be easily transported to and accessed by areas in the world that do not have access to this type of diagnostic technology.

Moving forward, we would determine functional correctness with an implementation of a higher number of processors. We would also attempt to work with actual input and output by using a transducer or similar technology to emit and transmit sound waves as well as VGA or another method to display an image.

REFERENCES

Ali, Murtaza, Dave Magee, and Udayan Dasgupta. "Signal Processing Overview of Ultrasound Systems for Medical Imaging." *SPRAB12* (2008). Texas Instruments. Web.

Glader, Paul. "GE Is Latest to Make Handheld Ultrasound." *The Wall Street Journal*. Dow Jones & Company, 12 Feb. 2010. Web. 9 Sept. 2015.

"The MobiUS SP1 System." *Smartphone Ultrasound*. Mobisante, 2015. Web.

"Ultrasound Machine." *M-Turbo*. FujiFilm SoloSite, 2015. Web.

APPENDIX I - Scheme Implementation

```
(define-struct point (x y))
(define-struct line (point-A point-B))

(define cross-product (lambda (a-point a-line)
                        (-
                         (*
                          (- (point-x(line-point-A a-line)) (point-x a-point))
                           (- (point-y(line-point-B a-line)) (point-y a-point)))
                         (*
                          (- (point-y(line-point-A a-line)) (point-y a-point))
                           (- (point-x(line-point-B a-line)) (point-x a-point))))))

(define cross-map (lambda (list-of-points a-line)
                    (cond
                      [(empty? list-of-points) empty]
                      [#t (cons (cross-product (car list-of-points) a-line)
                                (cross-map (cdr list-of-points) a-line))])))

(define packed-filter-crossed (lambda (list-of-crossed)
                        (cond
                          [(empty? list-of-crossed) empty]
                          [(> (car list-of-crossed) 0)
                               (cons (car list-of-crossed)
                               (packed-filter-crossed (cdr list-of-crossed)))]
                          [#t (packed-filter-crossed (cdr list-of-crossed))])))

(define packed-filter-points (lambda (list-of-points list-of-crossed)
                               (cond
                                 [(empty? list-of-points) empty]
                                 [(> (car list-of-crossed) 0) (cons (car list-of-points)
                                      (packed-filter-points (cdr list-of-points)
                                      (cdr list-of-crossed)))]
                                 [#t (packed-filter-points (cdr list-of-points)
                                      (cdr list-of-crossed))])))

(define list-length (lambda (a-list)
                      (cond
                        [(empty? a-list) 0]
                        [#t (+ 1 (list-length (cdr a-list)))])))

(define flatten
  (lambda (list-of-points)
    (cond
      [(empty? list-of-points) empty]
      [(list? (car list-of-points)) (append (flatten (car list-of-points))
               (flatten (cdr list-of-points)))]
      [#t (cons (car list-of-points) (flatten (cdr list-of-points)))])))
```

```
(define hsplit (lambda (list-of-points a-line)
                 (cond
                   [#t
                    (local ((define crossed (cross-map list-of-points a-line))
                           (define packed-crossed (packed-filter-crossed crossed))
                            (define packed-points (packed-filter-points list-of-points
                                                    crossed)))
                      (cond
                        [(< (list-length packed-crossed) 2)
                 (cons (line-point-A a-line) packed-points)]
                        [#t
                         (local ((define point-max (foldr (lambda (a-point old-point)
                       (cond
                         [(> (cross-product a-point a-line)
                             (cross-product old-point a-line)) a-point]
                         [#t old-point])) (car list-of-points) (cdr list-of-points))))
                            (cond
                              [#t (flatten (list (hsplit packed-points
                                   (make-line (line-point-A a-line) point-max)
                                   (hsplit packed-points (make-line point-max
                                   (line-point-B a-line))))))]))]))]))))

(define find-min-x (lambda (list-of-points)
                     (foldr (lambda (a-point old-point)
                              (cond
                                [(< (point-x a-point) (point-x old-point)) a-point]
                                [#t old-point]))
                            (car points) (cdr points))))

(define find-max-x (lambda (list-of-points)
                     (foldr (lambda (a-point old-point)
                              (cond
                                [(> (point-x a-point) (point-x old-point)) a-point]
                                [#t old-point]))
                            (car points) (cdr points))))

(define quickhull (lambda (list-of-points)
                    (cond
                      [#t
                       (local ((define xmin (find-min-x list-of-points))
                               (define xmax (find-max-x list-of-points)))
                         (cond
                           [#t
                            (flatten (list (hsplit list-of-points (make-line xmin xmax))
                             (hsplit list-of-points (make-line xmax xmin))))]))]))))
```

APPENDIX II - Verilog Implementation

```
//`timescale 1 ns / 100 ps

module m_port_ultra_quickhull_processor (input CLK100MHZ,
    input reg [4095:0] points,               //4096 / (8 * 2) = 256 points in each set
    input reg [8:0] SS,
    output reg [4095:0] convexPoints,
    output [7:0] convexSetSizeOutput,
    output [8:0] positiveCrossCountOutput,
    output [31:0] crossValueOutput,
    output signed [31:0] furthestCrossValueOutput,
    output [15:0] lnIndexOutput,
    output [8:0] ptCountOutput,
    output [31:0] currentLineOutput,
    output [15:0] currentPointOutput,
    output [15:0] furthestOutput,
    output [15:0] xMinPointOutput,
    output [15:0] xMaxPointOutput,
    output QINITIAL, QFIND_MAX, QFIND_MIN, QHULL_START, QCROSS, QHULL_RECURSE, QEND,
    input CPU_RESETN);                //Same as points, 256 points

    // Variables
    localparam PTSIZE = 16;                  //Point Size: 16 bits long, two 8 bit dimensions
    localparam LNSIZE = 32;                  //Line Size = 2 coordinates:  32 bits long
    // localparam SS = 256;                  //Set Size, need to count up to 256 = 8 bits
    reg [LNSIZE * 256 - 1 : 0] lineFIFO;     //32 bits * number of points, just to be safe
    reg [15:0] lnIndex;                      //Only need 13 bits, but 16 just in case
    reg [15:0] cxIndex;                      //Only need 12 bits, but 16 just in case
    reg [15:0] ptIndex;
    reg [8:0] ptCount;
    reg [7:0] convexSetSize;

    reg [PTSIZE - 1 : 0] xMinPoint;
    reg [PTSIZE - 1 : 0] xMaxPoint;
    reg [LNSIZE:0] line;
    reg [8:0] positiveCrossCount;

    reg [PTSIZE - 1 : 0] furthest;
    reg [PTSIZE - 1 : 0] currPoint;
    reg [(PTSIZE / 2) - 1 : 0] currPoint_X;
    reg [(PTSIZE / 2) - 1 : 0] currPoint_Y;
    reg [LNSIZE - 1 : 0] currLine;
    reg [PTSIZE - 1 : 0] currLine_A;
    reg [(PTSIZE / 2) - 1 : 0] currLine_AX;
    reg [(PTSIZE / 2) - 1 : 0] currLine_AY;
    reg [PTSIZE - 1 : 0] currLine_B;
    reg [(PTSIZE / 2) - 1 : 0] currLine_BX;
    reg [(PTSIZE / 2) - 1 : 0] currLine_BY;
    reg signed [31:0] crossValue;
    reg signed [31:0] furthestCrossValue;
    reg [LNSIZE - 1: 0] nextLineAddr;
    reg [LNSIZE - 1: 0] nextLineAddr2;
    reg [PTSIZE - 1: 0] nextCXAddr;
    reg [PTSIZE - 1: 0] nextCXAddr2;

    reg furthestFlag;

    assign convexSetSizeOutput = convexSetSize;
    assign positiveCrossCountOutput = positiveCrossCount;
    assign crossValueOutput = crossValue;
    assign lnIndexOutput = lnIndex;
    assign ptCountOutput = ptCount;
    assign currentLineOutput = currLine;
    assign currentPointOutput = currPoint;
```

```
assign furthestOutput = furthest;
assign xMinPointOutput = xMinPoint;
assign xMaxPointOutput = xMaxPoint;
assign furthestCrossValueOutput = furthestCrossValue;

// State Machine Implementation
reg[6:0] state;

assign { QEND, QHULL_RECURSE, QCROSS, QHULL_START, QFIND_MIN, QFIND_MAX, QINITIAL } =
state;

localparam
    INITIAL        =      7'b0000001,
    FIND_XMAX      =      7'b0000010,
    FIND_XMIN      =      7'b0000100,
    HULL_START     =      7'b0001000,
    CROSS          =      7'b0010000,
    HULL_RECURSE   =      7'b0100000,
    END            =      7'b1000000;

// For loop integers
integer i = 0;
integer j = 0;

//NSL, register assignents, and State Machine
always @(posedge CLK100MHZ, negedge CPU_RESETN) begin

    ptIndex = PTSIZE * ptCount;

    j = 0;
    for (i = ptIndex; i < ptIndex + PTSIZE; i = i + 1) begin
        currPoint[j] = points[i];
        j = j + 1;
    end

    j = 0;
    for (i = ptIndex; i < ptIndex + (PTSIZE / 2); i = i + 1) begin
        currPoint_X[j] = points[i];
        j = j + 1;
    end

    j = 0;
    for (i = ptIndex + (PTSIZE / 2); i < ptIndex + PTSIZE; i = i + 1) begin
        currPoint_Y[j] = points[i];
        j = j + 1;
    end

    j = 0;
    for (i = lnIndex; i < lnIndex + LNSIZE; i = i + 1) begin
        currLine[j] = lineFIFO[i];
        j = j + 1;
    end

    j = 0;
    for (i = lnIndex; i < lnIndex + (LNSIZE/2); i = i + 1) begin
        currLine_A[j] = lineFIFO[i];
        j = j + 1;
    end

    j = 0;
    for (i = lnIndex; i < lnIndex + (PTSIZE/2); i = i + 1) begin
        currLine_AX[j] = lineFIFO[i];
        j = j + 1;
    end

    j = 0;
```

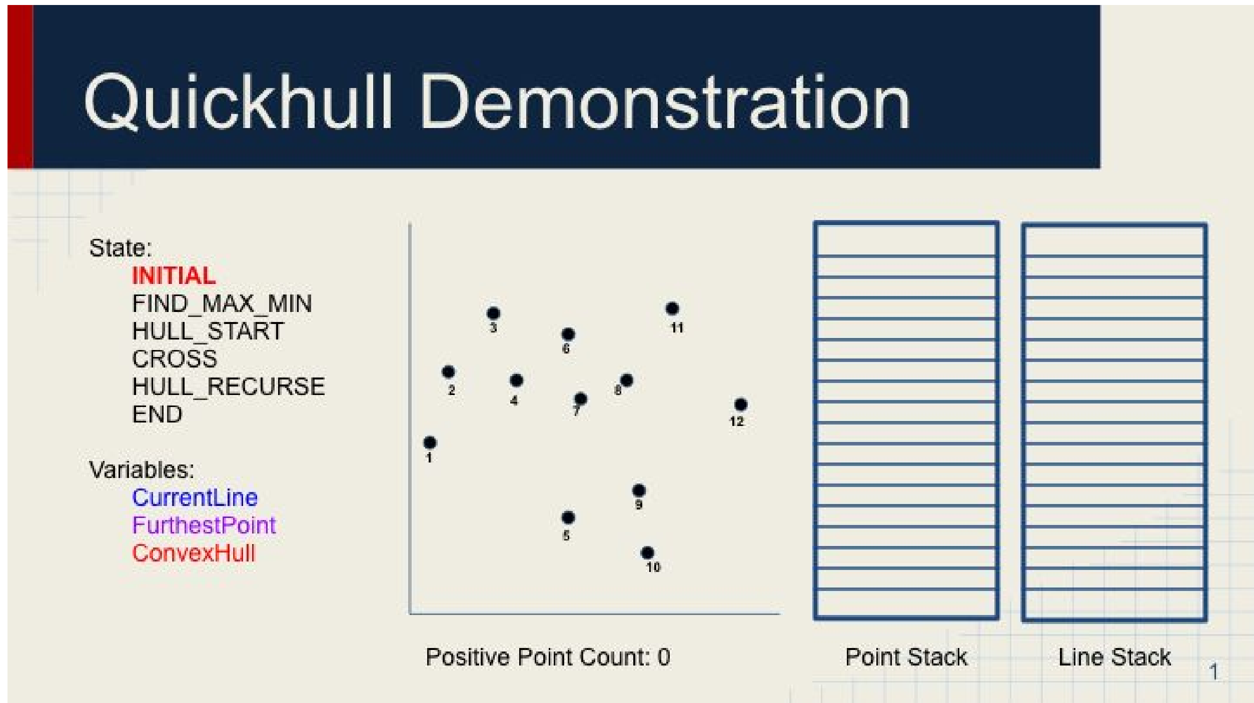
```
    for (i = lnIndex + (PTSIZE / 2); i < lnIndex + PTSIZE; i = i + 1) begin
        currLine_AY[j] = lineFIFO[i];
        j = j + 1;
    end

    j = 0;
    for (i = lnIndex + (LNSIZE/2); i < lnIndex + LNSIZE; i = i + 1) begin
        currLine_B [j] = lineFIFO[i];
        j = j + 1;
    end

    j = 0;
    for (i = lnIndex + PTSIZE; i < lnIndex + LNSIZE - (PTSIZE/2); i = i + 1) begin
        currLine_BX[j] = lineFIFO[i];
        j = j + 1;
    end

    j = 0;
    for (i = lnIndex + LNSIZE - (PTSIZE / 2); i < lnIndex + LNSIZE; i = i + 1) begin
        currLine_BY[j] = lineFIFO[i];
        j = j + 1;
    end

    j = 0;

    crossValue = (((currLine_AX - currPoint_X) * (currLine_BY - currPoint_Y)) -
((currLine_AY - currPoint_Y) * (currLine_BX - currPoint_X)));

    if (!CPU_RESETN) begin
        //Reset
        state <= INITIAL;
    end
    case (state)
        INITIAL: begin
            // State Logic
            lineFIFO <= 0;
            lnIndex <= 32;
            cxIndex <= 0;
            line <= 0;
            ptIndex <= 0;
            ptCount <= 0;
            positiveCrossCount <= 0;
            xMinPoint <= 0;
            xMaxPoint <= 0;
            crossValue <= 0;
            furthest <= 0;
            furthestCrossValue <= 0;
            furthestFlag <= 0;
            convexSetSize <= 0;
            convexPoints <= 0;
            // NSL
            state <= FIND_XMAX;

        end

        FIND_XMAX: begin
            //State Logic
            if (ptCount == 0) begin
                xMaxPoint <= currPoint;
            end
            else begin
                if (xMaxPoint < currPoint) begin
                    xMaxPoint <= currPoint;
                end
                else begin
                    //Do nothing
```

```
        end
    end

    //NSL
    if (ptCount != (SS - 1)) begin
        ptCount <= ptCount + 1;
        state <= FIND_XMAX;
    end
    else begin
        ptCount <= 0;
        state <= FIND_XMIN;
    end
end

FIND_XMIN: begin
    //State Logic
    if (ptCount == 0) begin
        xMinPoint <= currPoint;
    end
    else begin
        if (xMinPoint > currPoint) begin
            xMinPoint <= currPoint;
        end
        else begin
            //Do nothing
        end
    end

    //NSL
    if (ptCount != (SS - 1)) begin
        ptCount <= ptCount + 1;
        state <= FIND_XMIN;
    end
    else begin
        ptCount <= 0;
        state <= HULL_START;
    end
end

HULL_START: begin
    // State Logic
    nextLineAddr = {xMinPoint, xMaxPoint};
    j = 0;
    for (i = lnIndex; i < lnIndex + LNSIZE; i = i + 1) begin
        lineFIFO[i] = nextLineAddr[j];
        j = j + 1;
    end

    nextLineAddr2 = {xMaxPoint, xMinPoint};
    j = 0;
    for (i = lnIndex + LNSIZE; i < lnIndex + (LNSIZE * 2); i = i + 1) begin
        lineFIFO[i] = nextLineAddr2[j];
        j = j + 1;
    end
    lnIndex <= lnIndex + LNSIZE;

    // NSL
    ptCount <= 0;
    state <= CROSS;
end

CROSS: begin
    //State Logic
    //if (crossValue > 0) begin
    if (crossValue > 0 && ptCount != (SS)) begin
        positiveCrossCount <= positiveCrossCount + 1;
```

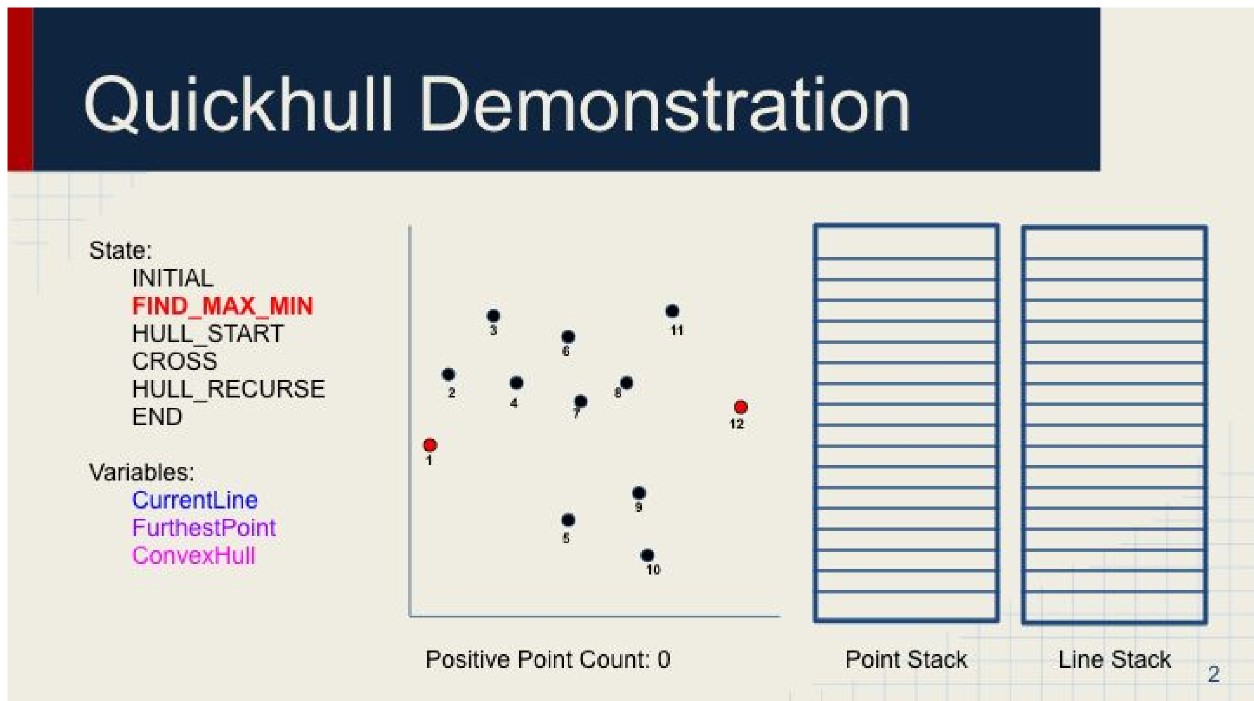
```
        if (furthestFlag == 0) begin
            furthestCrossValue <= crossValue;
            furthest <= currPoint;
            furthestFlag <= 1;
        end
        else begin
            if (furthestCrossValue < crossValue) begin
                    furthestCrossValue <= crossValue;
                    furthest <= currPoint;
            end
        end
    end

    //NSL
    if (ptCount != (SS)) begin
        ptCount <= ptCount + 1;
        state <= CROSS;
    end
    else begin
        ptCount <= 0;
        furthestFlag <= 0;
        state <= HULL_RECURSE;
    end

end

HULL_RECURSE: begin
    // State Logic

    if (positiveCrossCount == 1 && lnIndex != 0) begin
        nextCXAddr = currLine_A;
        j = 0;
        for (i = cxIndex; i < cxIndex + PTSIZE; i = i + 1) begin
            convexPoints[i] = nextCXAddr[j];
            j = j + 1;
        end
        nextCXAddr2 = furthest;
        j = 0;
        for (i = cxIndex + PTSIZE; i < cxIndex + (PTSIZE * 2); i = i + 1) begin
            convexPoints[i] = nextCXAddr2[j];
            j = j + 1;
        end
        cxIndex <= cxIndex + (2 * PTSIZE);
        convexSetSize <= convexSetSize + 2;

        lnIndex <= lnIndex - LNSIZE;
    end
    else if (positiveCrossCount == 0 && lnIndex != 0) begin
        nextCXAddr = currLine_A;
        j = 0;
        for (i = cxIndex; i < cxIndex + PTSIZE; i = i + 1) begin
            convexPoints[i] = nextCXAddr[j];
            j = j + 1;
        end
        cxIndex <= cxIndex + PTSIZE;
        convexSetSize <= convexSetSize + 1;

        lnIndex <= lnIndex - LNSIZE;
    end
    else begin
        nextLineAddr       = {furthest, currLine_A};
        nextLineAddr2      = {currLine_B, furthest};

        j = 0;
        for (i = lnIndex; i < lnIndex + LNSIZE; i = i + 1) begin
            lineFIFO[i] = nextLineAddr[j];
```

```
                    j = j + 1;
                end

                j = 0;
                for (i = lnIndex + LNSIZE; i < lnIndex + (LNSIZE * 2); i = i + 1) begin
                    lineFIFO[i] = nextLineAddr2[j];
                    j = j + 1;
                end
                lnIndex <= lnIndex + LNSIZE;
            end
            // NSL
            if ((lnIndex) != 0) begin
                positiveCrossCount <= 0;
                furthest <= 0;
                furthestCrossValue <= 0;
                ptCount <= 0;
                state <= CROSS;
            end
            else begin
                state <= END;
            end
        end

        END: begin
            //Wait
        end

    endcase
end

endmodule
```

Appendix III: Quickhull Example



Initial state: Variables initialized and resetted

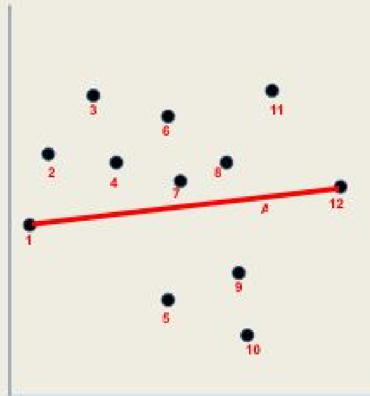


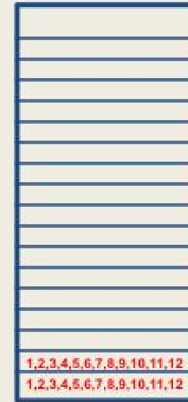
Find the minimum x point and maximum x point

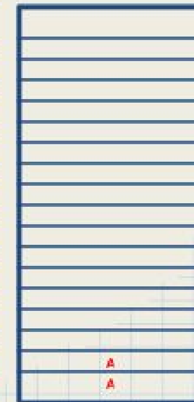
Quickhull Demonstration

State:
- INITIAL
- FIND_MAX_MIN
- **HULL_START**
- CROSS
- HULL_RECURSE
- END

Variables:
- CurrentLine
- FurthestPoint
- ConvexHull







Positive Point Count: 0

Point Stack

Line Stack

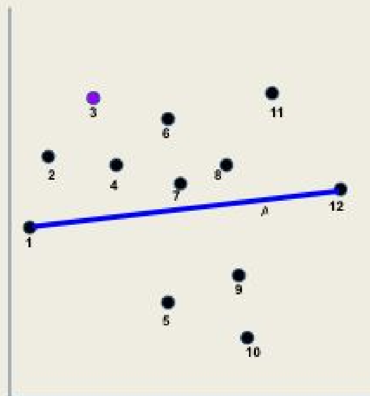
3

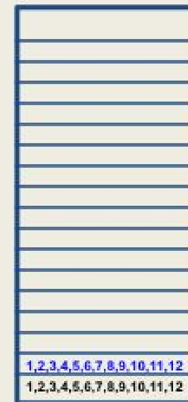
Adds two lines connecting x-min to x-max and x-max to x-min

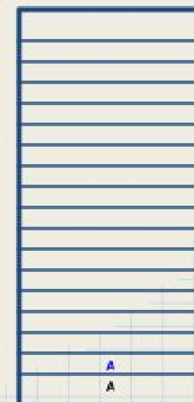
Quickhull Demonstration

State:
- INITIAL
- FIND_MAX_MIN
- HULL_START
- **CROSS**
- HULL_RECURSE
- END

Variables:
- CurrentLine
- FurthestPoint
- ConvexHull







Positive Point Count: 7

Point Stack

Line Stack

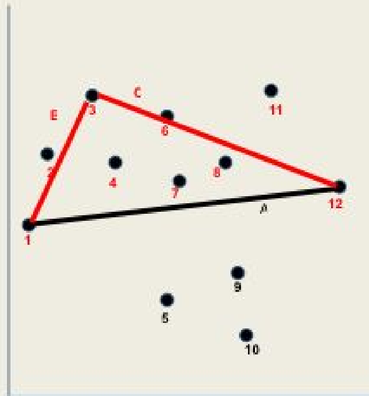
4

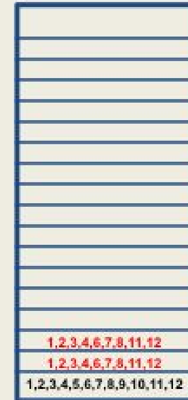
Finds furthest point from the line at the top of the stack

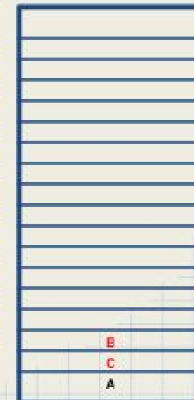
Quickhull Demonstration

State:
- INITIAL
- FIND_MAX_MIN
- HULL_START
- CROSS
- **HULL_RECURSE**
- END

Variables:
- CurrentLine
- FurthestPoint
- ConvexHull







Positive Point Count: 7

Point Stack

Line Stack

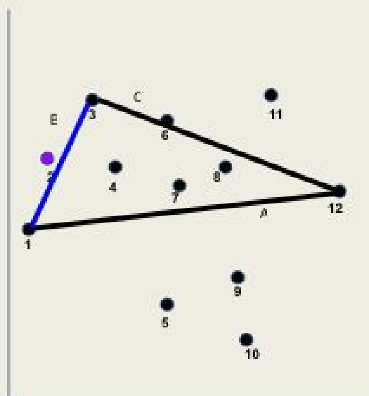
5

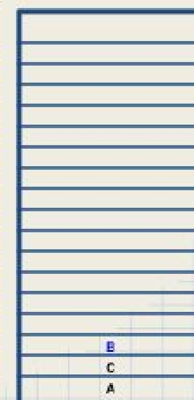
Since positive point count is greater than 1, adds two more lies to line stack. Stacks are popped first

Quickhull Demonstration

State:
- INITIAL
- FIND_MAX_MIN
- HULL_START
- **CROSS**
- HULL_RECURSE
- END

Variables:
- CurrentLine
- FurthestPoint
- ConvexHull





Positive Point Count: 1

Point Stack

Line Stack

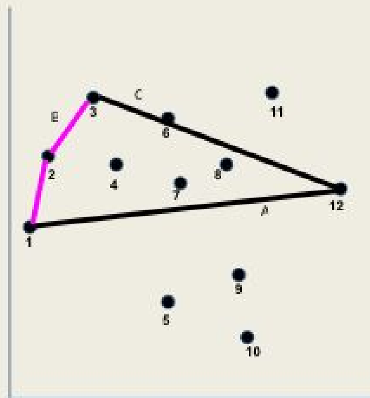
6

Finds furthest point from the line at the top of the stack.

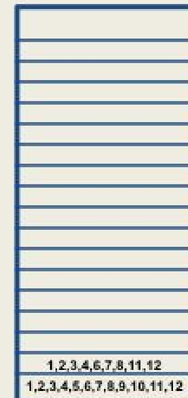
Quickhull Demonstration

State:
- INITIAL
- FIND_MAX_MIN
- HULL_START
- CROSS
- **HULL_RECURSE**
- END

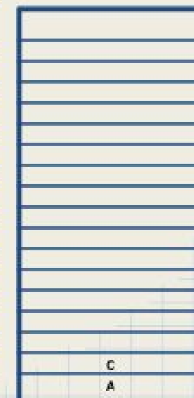
Variables:
- CurrentLine
- FurthestPoint
- ConvexHull



Positive Point Count: 1



Point Stack



Line Stack

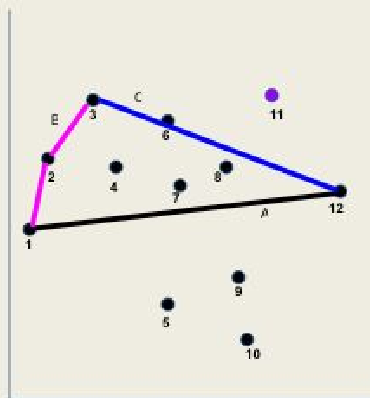
7

Positive point count is 1, therefore adds two lines to the convex hull. Stack is popped.

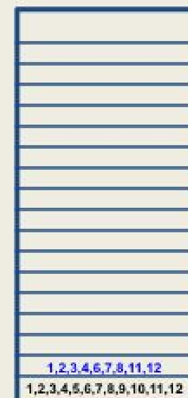
Quickhull Demonstration

State:
- INITIAL
- FIND_MAX_MIN
- HULL_START
- **CROSS**
- HULL_RECURSE
- END

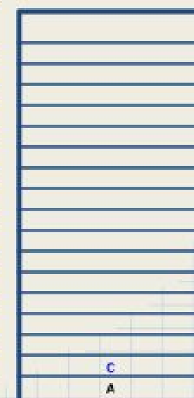
Variables:
- CurrentLine
- FurthestPoint
- ConvexHull



Positive Point Count: 2



Point Stack



Line Stack

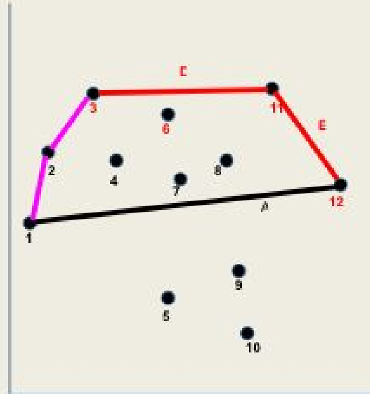
8

Finds furthest point from the line at the top of the stack.

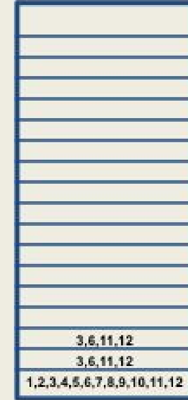
Quickhull Demonstration

State:
- INITIAL
- FIND_MAX_MIN
- HULL_START
- CROSS
- **HULL_RECURSE**
- END

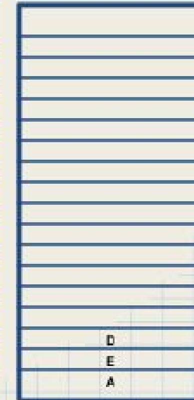
Variables:
- CurrentLine
- FurthestPoint
- ConvexHull



Positive Point Count: 2



Point Stack



Line Stack

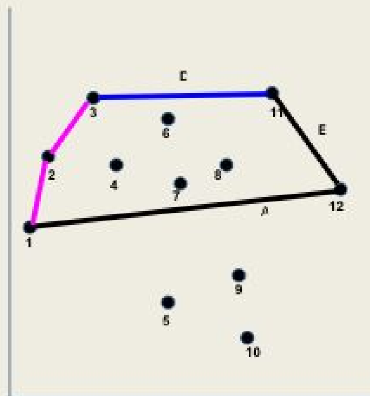
9

Since positive point count is greater than 1, adds two more lies to line stack. Stacks are popped first

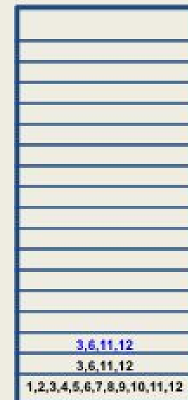
Quickhull Demonstration

State:
- INITIAL
- FIND_MAX_MIN
- HULL_START
- **CROSS**
- HULL_RECURSE
- END

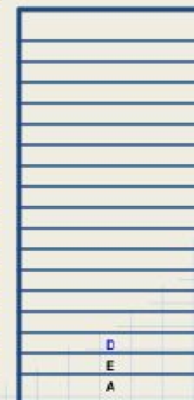
Variables:
- CurrentLine
- FurthestPoint
- ConvexHull



Positive Point Count: 0



Point Stack



Line Stack

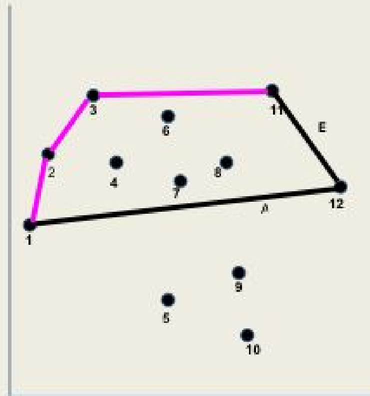
10

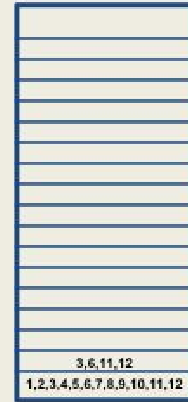
Finds furthest point, in this case there is none.

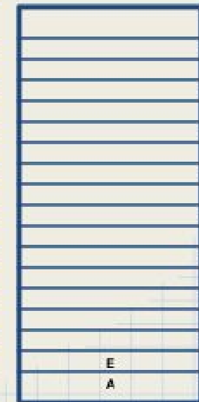
Quickhull Demonstration

State:
- INITIAL
- FIND_MAX_MIN
- HULL_START
- CROSS
- **HULL_RECURSE**
- END

Variables:
- CurrentLine
- FurthestPoint
- ConvexHull







Positive Point Count: 0

Point Stack

Line Stack

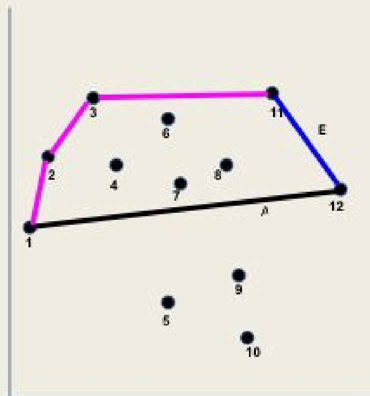
11

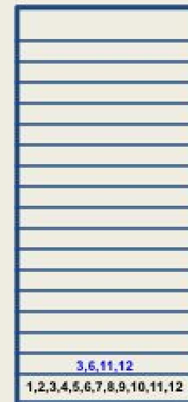
Positive point count is 0, therefore adds one lines to the convex hull. Stack is popped.

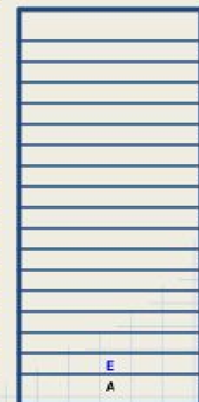
Quickhull Demonstration

State:
- INITIAL
- FIND_MAX_MIN
- HULL_START
- **CROSS**
- HULL_RECURSE
- END

Variables:
- CurrentLine
- FurthestPoint
- ConvexHull







Positive Point Count: 0

Point Stack

Line Stack

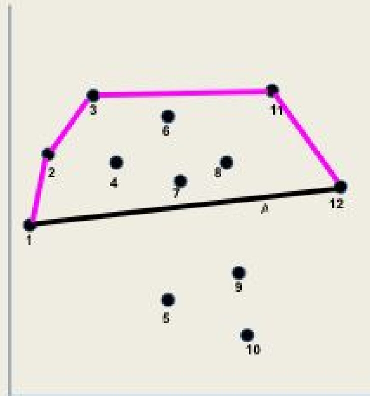
12

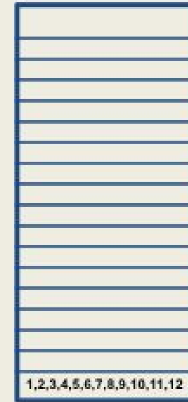
Finds furthest point, in this case there is none.

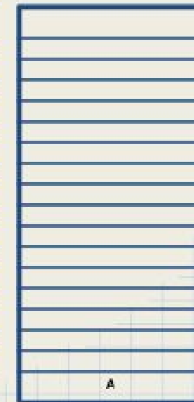
Quickhull Demonstration

State:
- INITIAL
- FIND_MAX_MIN
- HULL_START
- CROSS
- **HULL_RECURSE**
- END

Variables:
- CurrentLine
- FurthestPoint
- ConvexHull







Positive Point Count: 0

Point Stack

Line Stack

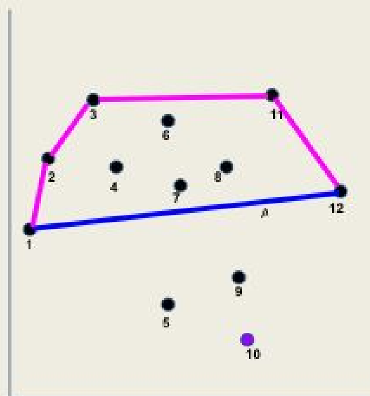
13

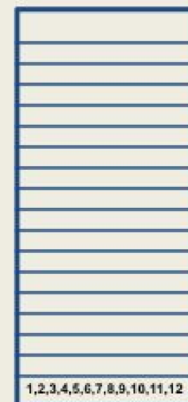
Positive point count is 0, therefore adds one lines to the convex hull. Stack is popped.

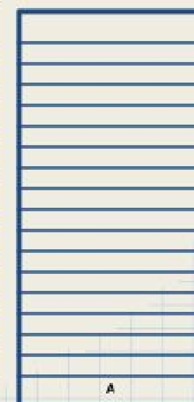
Quickhull Demonstration

State:
- INITIAL
- FIND_MAX_MIN
- HULL_START
- **CROSS**
- HULL_RECURSE
- END

Variables:
- CurrentLine
- FurthestPoint
- ConvexHull







Positive Point Count: 3

Point Stack

Line Stack

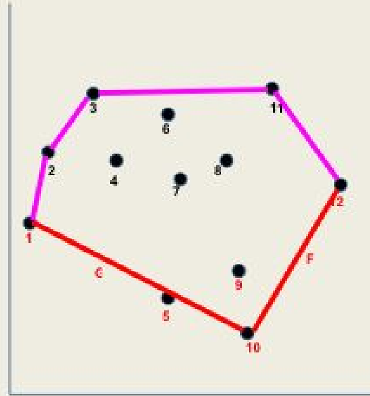
14

Now we are going to evaluate the other half of the set of points since we added the initial line twice.

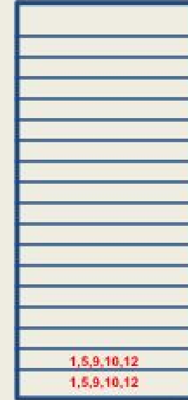
Quickhull Demonstration

State:
- INITIAL
- FIND_MAX_MIN
- HULL_START
- CROSS
- **HULL_RECURSE**
- END

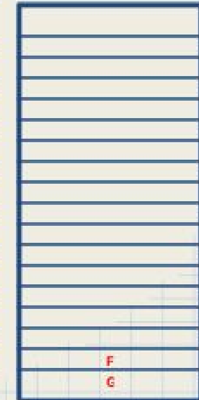
Variables:
- CurrentLine
- FurthestPoint
- ConvexHull



Positive Point Count: 3



Point Stack



Line Stack

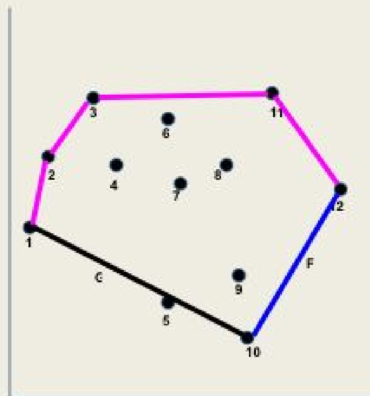
15

Since positive point count is greater than 1, adds two more lies to line stack. Stacks are popped first

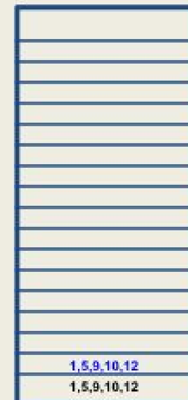
Quickhull Demonstration

State:
- INITIAL
- FIND_MAX_MIN
- HULL_START
- **CROSS**
- HULL_RECURSE
- END

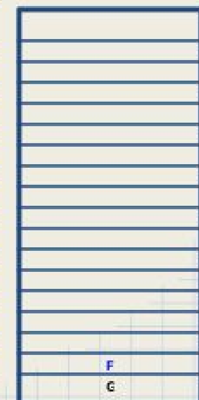
Variables:
- CurrentLine
- FurthestPoint
- ConvexHull



Positive Point Count: 0



Point Stack



Line Stack

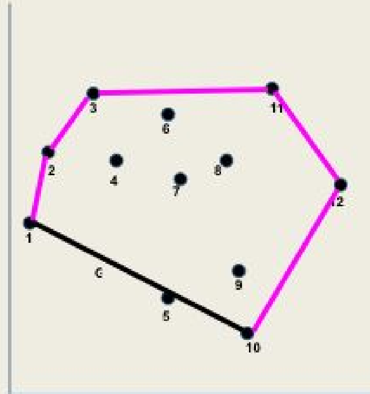
16

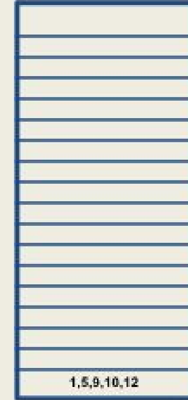
Finds furthest point, in this case there is none.

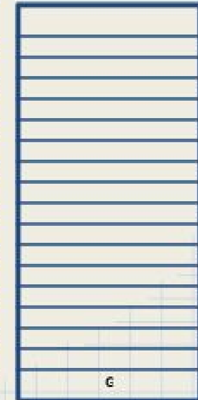
Quickhull Demonstration

State:
- INITIAL
- FIND_MAX_MIN
- HULL_START
- CROSS
- **HULL_RECURSE**
- END

Variables:
- CurrentLine
- FurthestPoint
- ConvexHull







Positive Point Count: 0

Point Stack

Line Stack

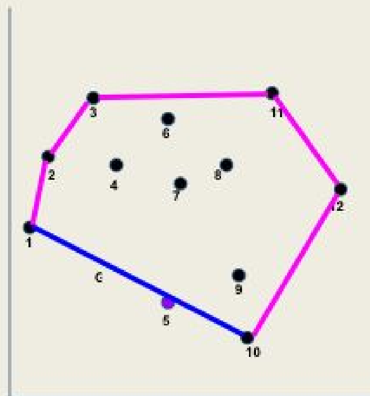
17

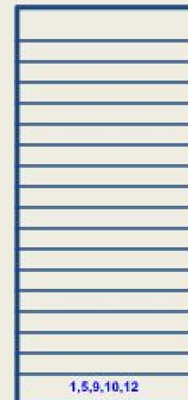
Positive point count is 0, therefore adds one lines to the convex hull. Stack is popped.

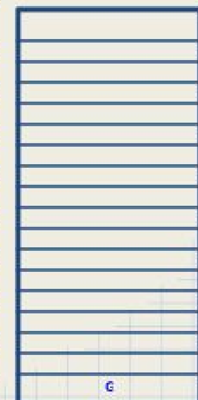
Quickhull Demonstration

State:
- INITIAL
- FIND_MAX_MIN
- HULL_START
- **CROSS**
- HULL_RECURSE
- END

Variables:
- CurrentLine
- FurthestPoint
- ConvexHull







Positive Point Count: 1

Point Stack

Line Stack

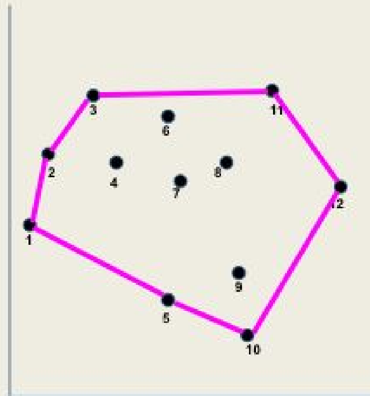
18

Finds furthest point from the line at the top of the stack.

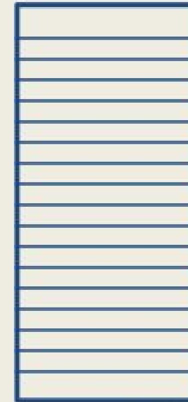
Quickhull Demonstration

State:
- INITIAL
- FIND_MAX_MIN
- HULL_START
- CROSS
- **HULL_RECURSE**
- END

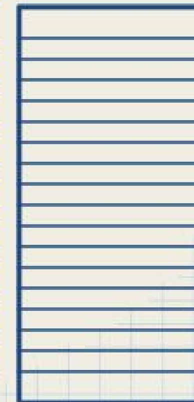
Variables:
- CurrentLine
- FurthestPoint
- ConvexHull



Positive Point Count: 1



Point Stack



Line Stack

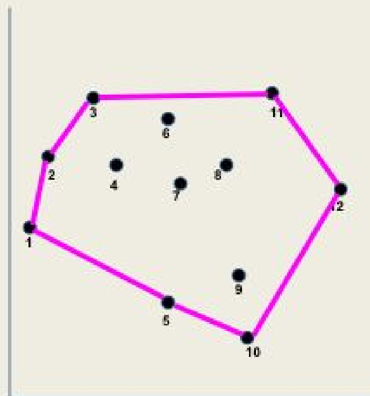
19

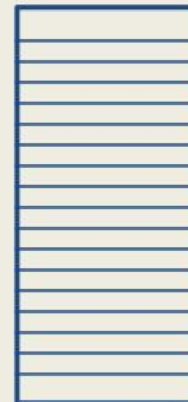
Positive point count is 1, therefore adds two lines to the convex hull. Stack is popped.

Quickhull Demonstration

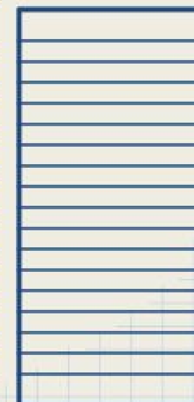
State:
- INITIAL
- FIND_MAX_MIN
- HULL_START
- CROSS
- HULL_RECURSE
- **END**

Variables:
- CurrentLine
- FurthestPoint
- ConvexHull





Point Stack



Line Stack

20

Since stacks are empty, go to end state.