FPGA Rasterization and Interpolation

Seth Baker

University of Pittsburgh – Johnstown

December 2025

Synopsis

The FPGA Rasterization and Interpolation Project was a Directed Study for the University of Pittsburgh at Johnstown during the fall term of 2025 undertaken by Seth Baker for research directed by Stephan Ohl. The project centered around implementing basic triangle rasterization and interpolation on the DE10-Nano Development and Education Board.

1 Introduction

To implement rasterization and interpolation in the DE10-Nano, there were several things that needed to be accomplished first. Understanding of the board's hardware and architecture, interfacing with the board, how an operating system is stored, what languages and tooling to use, available frameworks, and how to implement testing were all crucial considerations that had to be attended to long before any graphics were displayed.

1.1 Hardware and Architecture

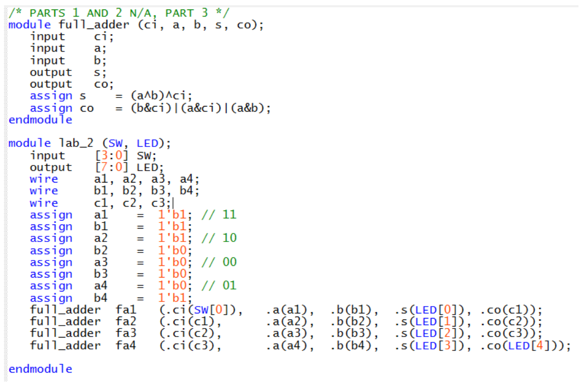
The DE10-Nano Development and Education Board's main hardware components are its Intel System-on-Chip (SoC) FPGA, which combines an FPGA fabric with a dual-core Cortex-A9 embedded core. The Altera SoC also integrates an ARM-based HPS made up of processor, peripherals, and memory interfaces. It also has DDR3 memory. Storage of a Linux-based operating system requires the user to upload an ISO image to the micro-SD card. Options for operating systems range from full-fledged GUIs to simple terminal-based distributions.

1.2 Tooling

The tooling utilized for this project included Intel Quartus Prime Lite 17.0 in compliance with the DE10-Nano open-source MiSTer framework, the MiSTer ISO (Linux-based distribution), Verilog HDL, the C programming language, Balena Etcher and Vim.

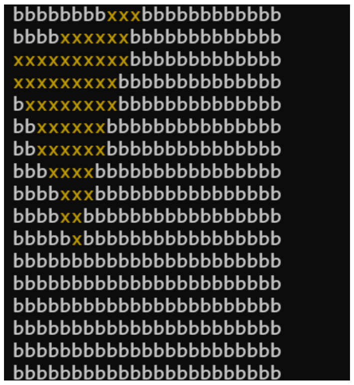
1.3 Testing

Various tests across numerous tooling were necessary to begin work on rasterization. These tests ranged from unrelated (but necessary) development of rbf files via Quartus Prime Lite to learn about digital logical circuits (see Figure 1.3.1) to C code development that tested the implementation of rasterization concepts via terminal graphics (see Figure 1.3.2).



```
/* PARTS 1 AND 2 N/A, PART 3 */
module full_adder (ci, a, b, s, co);
   input    ci;
   input    a;
   input    b;
   output   s;
   output   co;
   assign s    = (a^b)^ci;
   assign co   = (b&ci)|(a&ci)|(a&b);
endmodule

module lab_2 (SW, LED);
   input    [3:0] SW;
   output   [7:0] LED;
   wire     a1, a2, a3, a4;
   wire     b1, b2, b3, b4;
   wire     c1, c2, c3;
   assign   a1    =  1'b1; // 11
   assign   b1    =  1'b1;
   assign   a2    =  1'b1; // 10
   assign   b2    =  1'b0;
   assign   a3    =  1'b0; // 00
   assign   b3    =  1'b0;
   assign   a4    =  1'b0; // 01
   assign   b4    =  1'b1;
   full_adder  fa1   (.ci(SW[0]),   .a(a1),  .b(b1),  .s(LED[0]), .co(c1));
   full_adder  fa2   (.ci(c1),      .a(a2),  .b(b2),  .s(LED[1]), .co(c2));
   full_adder  fa3   (.ci(c2),      .a(a3),  .b(b3),  .s(LED[2]), .co(c3));
   full_adder  fa4   (.ci(c3),      .a(a4),  .b(b4),  .s(LED[3]), .co(LED[4]));

endmodule
```

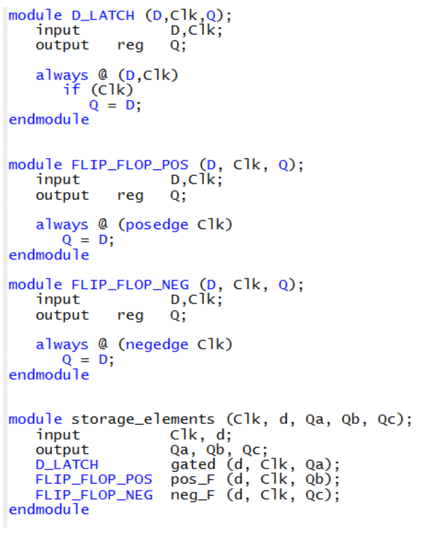
Figure 1.3.1 Full Adder



```
bbbbbbbbxxxbbbbbbbbbbbb
bbbbxxxxxxbbbbbbbbbbbbb
xxxxxxxxxxbbbbbbbbbbbbb
xxxxxxxxxbbbbbbbbbbbbbb
bxxxxxxxxbbbbbbbbbbbbbb
bbxxxxxxbbbbbbbbbbbbbbb
bbxxxxxxbbbbbbbbbbbbbbb
bbbxxxxbbbbbbbbbbbbbbbb
bbbbxxxbbbbbbbbbbbbbbbb
bbbbxxbbbbbbbbbbbbbbbbb
bbbbbxbbbbbbbbbbbbbbbbb
bbbbbbbbbbbbbbbbbbbbbbb
bbbbbbbbbbbbbbbbbbbbbbb
bbbbbbbbbbbbbbbbbbbbbbb
bbbbbbbbbbbbbbbbbbbbbbb
bbbbbbbbbbbbbbbbbbbbbbb
bbbbbbbbbbbbbbbbbbbbbbb
```

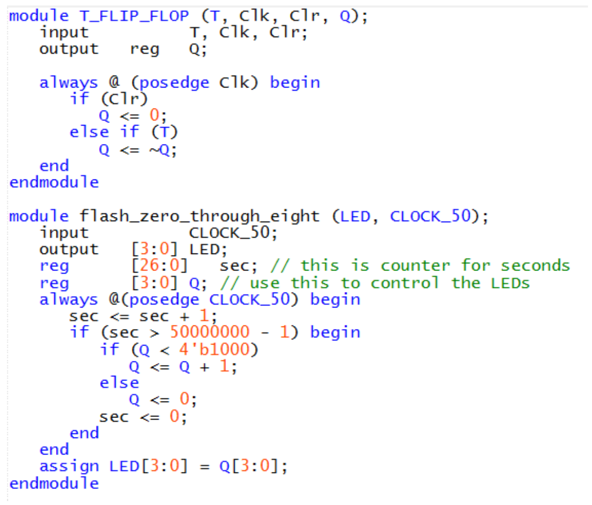
Figure 1.3.2 Terminal graphics

The first tests carried out were related to digital circuits. Due to the nature of FPGAs, it's paramount that the developer understands digital circuits and how to implement them in their respective development environment. For this project, multiple resources such as the Nandland YouTube channel and website, the Neso Academy YouTube channel, and the FPGAAcademy website aided heavily in my understanding of digital circuits and logic. FPGAAcademy had multiple labs associated with the DE10-Nano that provided a hands-on learning experience with digital circuits. Two of the most helpful labs were the Latches, Flip-Flops and Registers lab (see Figure 1.3.3) and the Counters lab (see Figure 1.3.4).



```
module D_LATCH (D,Clk,Q);
   input          D,Clk;
   output   reg   Q;

   always @ (D,Clk)
      if (Clk)
         Q = D;
endmodule


module FLIP_FLOP_POS (D, Clk, Q);
   input          D,Clk;
   output   reg   Q;

   always @ (posedge Clk)
      Q = D;
endmodule

module FLIP_FLOP_NEG (D, Clk, Q);
   input          D,Clk;
   output   reg   Q;

   always @ (negedge Clk)
      Q = D;
endmodule


module storage_elements (Clk, d, Qa, Qb, Qc);
   input          Clk, d;
   output         Qa, Qb, Qc;
   D_LATCH        gated (d, Clk, Qa);
   FLIP_FLOP_POS  pos_F (d, Clk, Qb);
   FLIP_FLOP_NEG  neg_F (d, Clk, Qc);
endmodule
```

Figure 1.3.3 Latches, Flip-Flops and Registers



```
module T_FLIP_FLOP (T, Clk, Clr, Q);
   input          T, Clk, Clr;
   output   reg   Q;

   always @ (posedge Clk) begin
      if (Clr)
         Q <= 0;
      else if (T)
         Q <= ~Q;
   end
endmodule

module flash_zero_through_eight (LED, CLOCK_50);
   input          CLOCK_50;
   output   [3:0] LED;
   reg      [26:0]   sec; // this is counter for seconds
   reg      [3:0] Q; // use this to control the LEDs
   always @(posedge CLOCK_50) begin
      sec <= sec + 1;
      if (sec > 50000000 - 1) begin
         if (Q < 4'b1000)
            Q <= Q + 1;
         else
            Q <= 0;
         sec <= 0;
      end
   end
   assign LED[3:0] = Q[3:0];
endmodule
```

Figure 1.3.4 Counters

After understanding digital circuits and logic, the next step of testing was delving into embedded Linux C programming. This topic was explored almost exclusively through FPGAAcademy's Embedded Systems lab section. The most useful of which were Introduction to Graphics and Animation and Using ASCII Graphics for Animation. Undoubtedly the most important testing conducted was done within the FPGA framework, specifically using the tutorials created by alanswx for creating a VGA controller using 8 bits of color (RGB332) and a VGA image viewer. The tutorials themselves were helpful but using them as a template and implementing different functionalities such as displaying random color generation per pixel (see Figure 1.3.5) and displaying a simple sine wave utilizing a LUT (see Figure 1.3.6) were the most impactful in my understanding of the framework and the process of implementing graphics in the DE10-Nano.

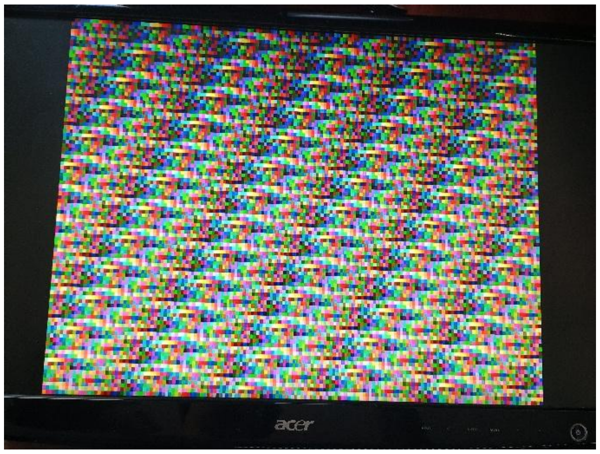


Figure 1.3.5 Random Color Generation per pixel

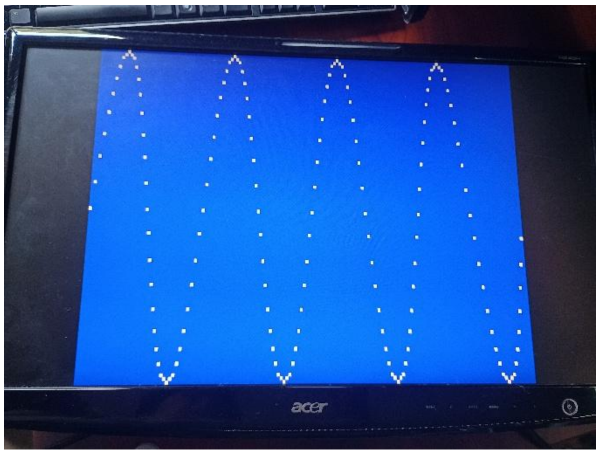
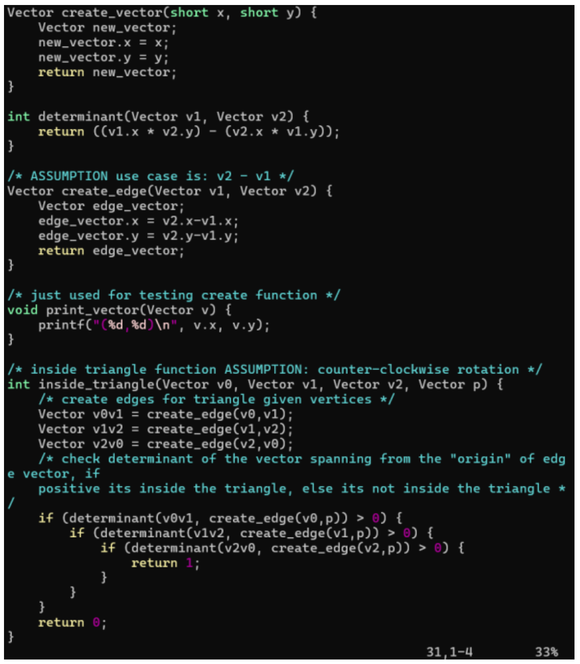


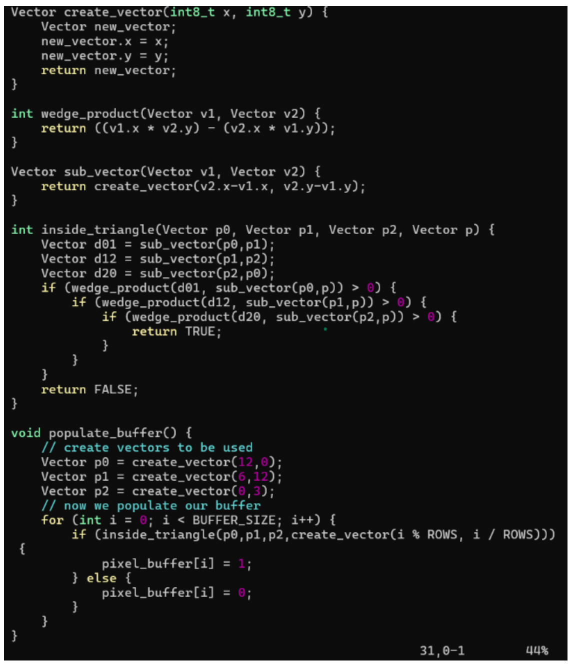
Figure 1.3.6 Simple Sine Wave

Another extremely important phase of testing was tests related to the rasterization algorithm, developed in C on Ubuntu WSL. I chose to implement the rasterization algorithm through vectors, utilizing the wedge product to determine if a pixel is inside a triangle defined by three vectors oriented counterclockwise. Other important functions defined were sub_vector for subtraction of two vectors and create_vector for simple vector creation of the Vector struct. There were two versions of this program – version 0 (see Figure 1.3.7), which simply plotted points to the terminal with regular int primitives, and version 1, which implemented the concept of a pixel buffer which was filled based on results from the algorithm that checks if pixels are inside the triangle, and uses typing for 8 bit integers (see Figure 1.3.8). Both version 0 and version 1 share the same output (see Figure 1.3.2). Both programs utilized the same pixel plotting functions in the terminal used in the Embedded Systems section of the FPGAAcademy labs.



```
Vector create_vector(short x, short y) {
    Vector new_vector;
    new_vector.x = x;
    new_vector.y = y;
    return new_vector;
}

int determinant(Vector v1, Vector v2) {
    return ((v1.x * v2.y) - (v2.x * v1.y));
}

/* ASSUMPTION use case is: v2 - v1 */
Vector create_edge(Vector v1, Vector v2) {
    Vector edge_vector;
    edge_vector.x = v2.x-v1.x;
    edge_vector.y = v2.y-v1.y;
    return edge_vector;
}

/* just used for testing create function */
void print_vector(Vector v) {
    printf("(%d,%d)\n", v.x, v.y);
}

/* inside triangle function ASSUMPTION: counter-clockwise rotation */
int inside_triangle(Vector v0, Vector v1, Vector v2, Vector p) {
    /* create edges for triangle given vertices */
    Vector v0v1 = create_edge(v0,v1);
    Vector v1v2 = create_edge(v1,v2);
    Vector v2v0 = create_edge(v2,v0);
    /* check determinant of the vector spanning from the "origin" of edg
e vector, if
    positive its inside the triangle, else its not inside the triangle *
/
    if (determinant(v0v1, create_edge(v0,p)) > 0) {
        if (determinant(v1v2, create_edge(v1,p)) > 0) {
            if (determinant(v2v0, create_edge(v2,p)) > 0) {
                return 1;
            }
        }
    }
    return 0;
}
                                                  31,1-4         33%
```

Figure 1.3.7 Version 0

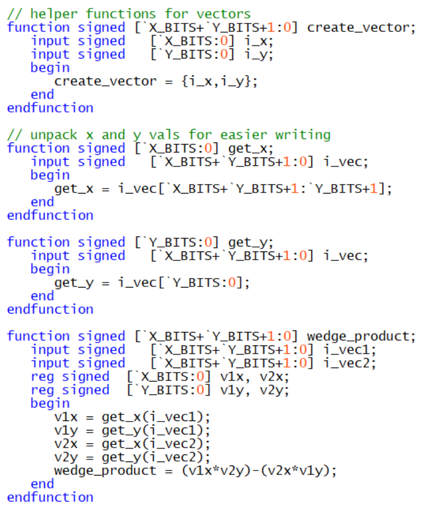


```
Vector create_vector(int8_t x, int8_t y) {
    Vector new_vector;
    new_vector.x = x;
    new_vector.y = y;
    return new_vector;
}

int wedge_product(Vector v1, Vector v2) {
    return ((v1.x * v2.y) - (v2.x * v1.y));
}

Vector sub_vector(Vector v1, Vector v2) {
    return create_vector(v2.x-v1.x, v2.y-v1.y);
}

int inside_triangle(Vector p0, Vector p1, Vector p2, Vector p) {
    Vector d01 = sub_vector(p0,p1);
    Vector d12 = sub_vector(p1,p2);
    Vector d20 = sub_vector(p2,p0);
    if (wedge_product(d01, sub_vector(p0,p)) > 0) {
        if (wedge_product(d12, sub_vector(p1,p)) > 0) {
            if (wedge_product(d20, sub_vector(p2,p)) > 0) {
                return TRUE;
            }
        }
    }
    return FALSE;
}

void populate_buffer() {
    // create vectors to be used
    Vector p0 = create_vector(12,0);
    Vector p1 = create_vector(6,12);
    Vector p2 = create_vector(0,3);
    // now we populate our buffer
    for (int i = 0; i < BUFFER_SIZE; i++) {
        if (inside_triangle(p0,p1,p2,create_vector(i % ROWS, i / ROWS)))
 {
            pixel_buffer[i] = 1;
        } else {
            pixel_buffer[i] = 0;
        }
    }
}
                                                  31,0-1         44%
```

Figure 1.3.8 Version 1

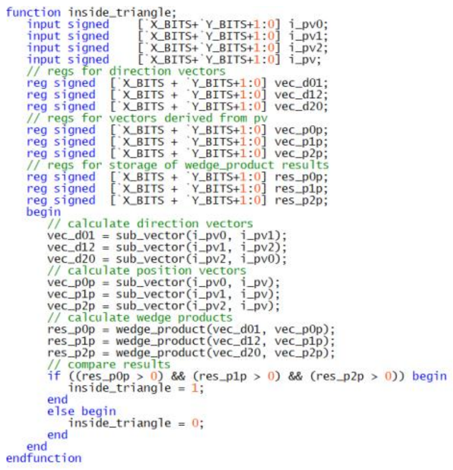
2 Rasterization

After completing all the testing and gaining the prior knowledge specified in the previous section, the implementation of triangle rasterization in the FPGA fabric was relatively easy. The codebase used was the VGA image viewer by alanswx utilized in the testing section. To implement rasterization, I started by creating macros X_BITS and Y_BITS which represent the number of bits reserved in a vector (data element in Verilog) for the x coordinate and y coordinate, respectively, and implemented a simple function for creating a vector. I also added helper functions to get the x and y coordinates from a given vector, which is helpful in readability as well as reducing the likelihood of bugs resulting from invalid indexing, and then implemented a function for the wedge product (see Figure 2.1). Next, I implemented a function for checking if the input point vector is inside the triangle, which had very similar logic as the C programming function (see Figure 2.2). Then, there's a very simple bit of code inside the always block at the positive edge of the clock cycle – it creates a vector whose x and y correspond with the current counter associated with the index of the vmem bit vector and checks if that vector is inside the triangle. If it does not pass the check, it is filled with some background color (in my case, blue) and if it does pass it is filled with a different color (white) (see Figure 2.3).



```
// helper functions for vectors
function signed [`X_BITS+`Y_BITS+1:0] create_vector;
   input signed   [`X_BITS:0] i_x;
   input signed   [`Y_BITS:0] i_y;
   begin
      create_vector = {i_x,i_y};
   end
endfunction

// unpack x and y vals for easier writing
function signed [`X_BITS:0] get_x;
   input signed   [`X_BITS+`Y_BITS+1:0] i_vec;
   begin
      get_x = i_vec[`X_BITS+`Y_BITS+1:`Y_BITS+1];
   end
endfunction

function signed [`Y_BITS:0] get_y;
   input signed   [`X_BITS+`Y_BITS+1:0] i_vec;
   begin
      get_y = i_vec[`Y_BITS:0];
   end
endfunction

function signed [`X_BITS+`Y_BITS+1:0] wedge_product;
   input signed   [`X_BITS+`Y_BITS+1:0] i_vec1;
   input signed   [`X_BITS+`Y_BITS+1:0] i_vec2;
   reg signed  [`X_BITS:0] v1x, v2x;
   reg signed  [`Y_BITS:0] v1y, v2y;
   begin
      v1x = get_x(i_vec1);
      v1y = get_y(i_vec1);
      v2x = get_x(i_vec2);
      v2y = get_y(i_vec2);
      wedge_product = (v1x*v2y)-(v2x*v1y);
   end
endfunction
```

Figure 2.1



```
function inside_triangle;
   input signed    [`X_BITS+`Y_BITS+1:0] i_pv0;
   input signed    [`X_BITS+`Y_BITS+1:0] i_pv1;
   input signed    [`X_BITS+`Y_BITS+1:0] i_pv2;
   input signed    [`X_BITS+`Y_BITS+1:0] i_pv;
   // regs for direction vectors
   reg signed  [`X_BITS + `Y_BITS+1:0] vec_d01;
   reg signed  [`X_BITS + `Y_BITS+1:0] vec_d12;
   reg signed  [`X_BITS + `Y_BITS+1:0] vec_d20;
   // regs for vectors derived from pv
   reg signed  [`X_BITS + `Y_BITS+1:0] vec_p0p;
   reg signed  [`X_BITS + `Y_BITS+1:0] vec_p1p;
   reg signed  [`X_BITS + `Y_BITS+1:0] vec_p2p;
   // regs for storage of wedge_product results
   reg signed  [`X_BITS + `Y_BITS+1:0] res_p0p;
   reg signed  [`X_BITS + `Y_BITS+1:0] res_p1p;
   reg signed  [`X_BITS + `Y_BITS+1:0] res_p2p;
   begin
      // calculate direction vectors
      vec_d01 = sub_vector(i_pv0, i_pv1);
      vec_d12 = sub_vector(i_pv1, i_pv2);
      vec_d20 = sub_vector(i_pv2, i_pv0);
      // calculate position vectors
      vec_p0p = sub_vector(i_pv0, i_pv);
      vec_p1p = sub_vector(i_pv1, i_pv);
      vec_p2p = sub_vector(i_pv2, i_pv);
      // calculate wedge products
      res_p0p = wedge_product(vec_d01, vec_p0p);
      res_p1p = wedge_product(vec_d12, vec_p1p);
      res_p2p = wedge_product(vec_d20, vec_p2p);
      // compare results
      if ((res_p0p > 0) && (res_p1p > 0) && (res_p2p > 0)) begin
         inside_triangle = 1;
      end
      else begin
         inside_triangle = 0;
      end
   end
endfunction
```

Figure 2.2

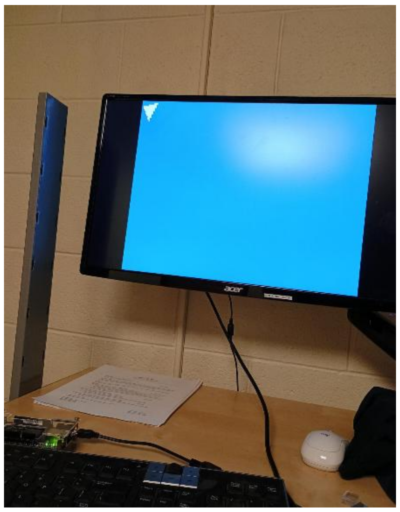


Figure 2.3

3 Interpolation

The implementation of interpolation into the rasterization project was a relatively easy addition but required knowledge of barycentric coordinates and fixed-point arithmetic for proper implementation. I started by attaching color bits to the front of every vector. Originally this value was 8 bits to stay consistent with the projects RGB332 format, but after proper implementation this was changed to 24 bits for 8 bits red, 8 bits green, and 8 bits blue colors. This allows for exponentially more different colors to make the final interpolation very smooth when transitioning colors. After attaching colors, I added a LUT for getting the area for the entire triangle. This is because this value is used in every pixel calculation, so it would be very expensive to keep performing the same operation. Next, I modified the always block to get the areas of triangles APC, ABP, and PBC (P representing the point vector that calculations are being done for), and determine alpha, beta, and gamma values for each of the triangles by using 16-bit fixed point arithmetic. Then, it gets interpolated RGB values by adding alpha, beta, and gamma by vectors A, B, and C’s individual RGB values (respectively). Then, it shifts each R, G, B 16 bits to account for fixed-point arithmetic. Finally, it writes the new interpolated values to vmem (see Figure 3.1). There was a critical bug that produced extremely odd-looking interpolation – I forgot to divide the calculated area by two for the subtriangles but remembered to do so to the area of the entire triangle, so proper interpolation appeared in the subtriangles with some garbage in the middle (see Figures 3.2 and 3.3). After this bug was fixed, the triangle showed proper interpolation (for RGB332 see Figure 3.4, for RGB888 see Figure 3.5).

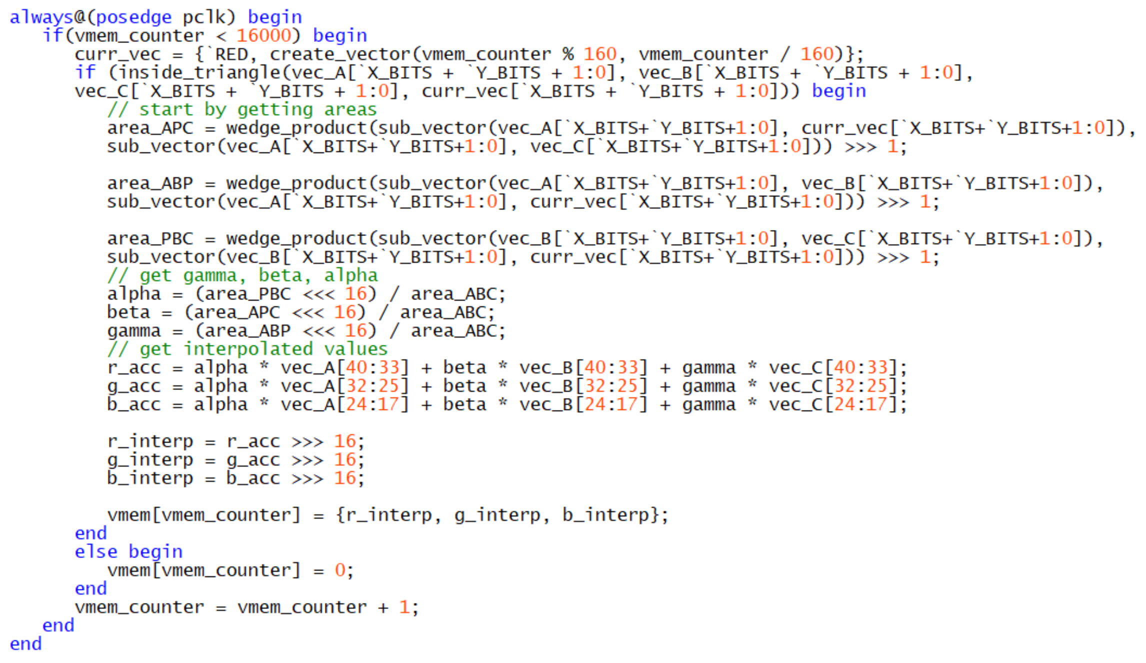

```
always@(posedge pclk) begin
   if(vmem_counter < 16000) begin
      curr_vec = {`RED, create_vector(vmem_counter % 160, vmem_counter / 160)};
      if (inside_triangle(vec_A[`X_BITS + `Y_BITS + 1:0], vec_B[`X_BITS + `Y_BITS + 1:0],
      vec_C[`X_BITS + `Y_BITS + 1:0], curr_vec[`X_BITS + `Y_BITS + 1:0])) begin
         // start by getting areas
         area_APC = wedge_product(sub_vector(vec_A[`X_BITS+`Y_BITS+1:0], curr_vec[`X_BITS+`Y_BITS+1:0]),
         sub_vector(vec_A[`X_BITS+`Y_BITS+1:0], vec_C[`X_BITS+`Y_BITS+1:0])) >>> 1;

         area_ABP = wedge_product(sub_vector(vec_A[`X_BITS+`Y_BITS+1:0], vec_B[`X_BITS+`Y_BITS+1:0]),
         sub_vector(vec_A[`X_BITS+`Y_BITS+1:0], curr_vec[`X_BITS+`Y_BITS+1:0])) >>> 1;

         area_PBC = wedge_product(sub_vector(vec_B[`X_BITS+`Y_BITS+1:0], vec_C[`X_BITS+`Y_BITS+1:0]),
         sub_vector(vec_B[`X_BITS+`Y_BITS+1:0], curr_vec[`X_BITS+`Y_BITS+1:0])) >>> 1;
         // get gamma, beta, alpha
         alpha = (area_PBC <<< 16) / area_ABC;
         beta = (area_APC <<< 16) / area_ABC;
         gamma = (area_ABP <<< 16) / area_ABC;
         // get interpolated values
         r_acc = alpha * vec_A[40:33] + beta * vec_B[40:33] + gamma * vec_C[40:33];
         g_acc = alpha * vec_A[32:25] + beta * vec_B[32:25] + gamma * vec_C[32:25];
         b_acc = alpha * vec_A[24:17] + beta * vec_B[24:17] + gamma * vec_C[24:17];

         r_interp = r_acc >>> 16;
         g_interp = g_acc >>> 16;
         b_interp = b_acc >>> 16;

         vmem[vmem_counter] = {r_interp, g_interp, b_interp};
      end
      else begin
         vmem[vmem_counter] = 0;
      end
      vmem_counter = vmem_counter + 1;
   end
end
```

Figure 3.1 Interpolation main algorithm

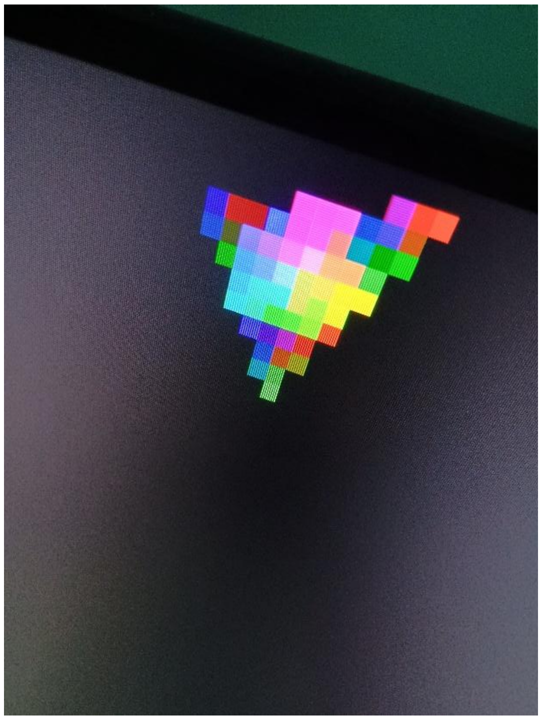


Figure 3.2 Bugged Interpolation Tiny

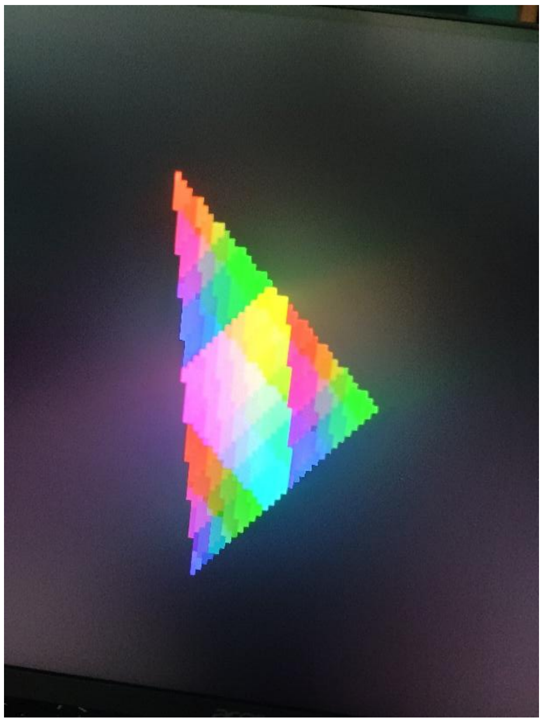


Figure 3.3 Bugged Interpolation Large

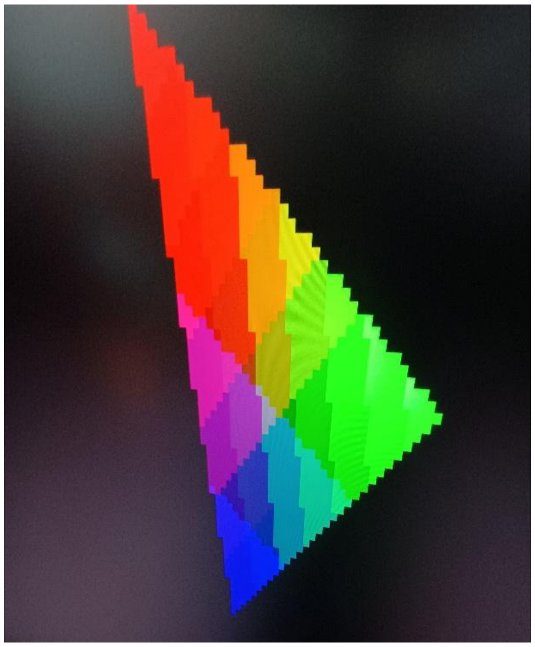


Figure 3.3 Fixed RGB332 Interpolation

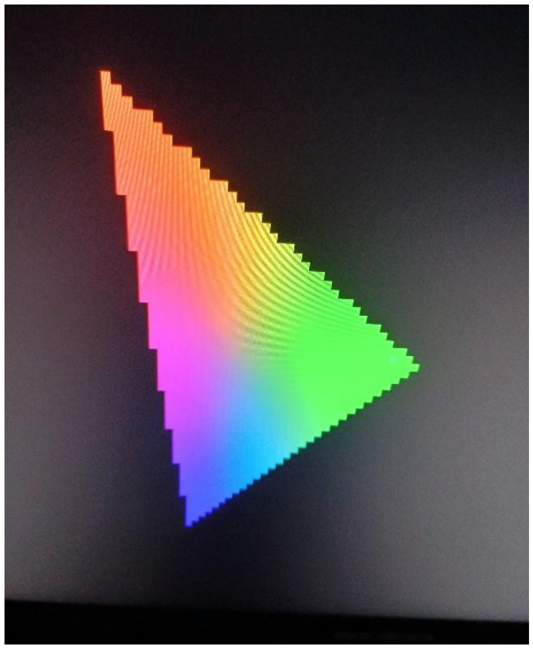


Figure 3.4 Fixed RGB888 Interpolation

4 References

MiSTer FPGA Documentation

https://mister-devel.github.io/MkDocs_MiSTer/

Nandland

https://nandland.com/

FPGAAcademy

https://fpgacademy.org/index.html

alanswx's MiSTer tutorials

https://github.com/alanswx/Tutorials_MiSTer

Rasterization

https://lisyarus.github.io/blog/posts/implementing-a-tiny-cpu-rasterizer-part-2.html

Barycentric Coordinates

https://graphicscompendium.com/intro/03-barycentric-coordinates

Fixed-Point Arithmetic

https://projectf.io/posts/fixed-point-numbers-in-verilog/