Robin Yancey, Prashant Gupta
Owens
EEC277 Graphics Architecture
March 14, 2017

GPU Architecture Analysis: Rules of Rasterization

Introduction: This project analyses the rasterizer of graphics processing unit focusing on rules or methods used for rasterizing geometry. Images are placed on the pixel grid by rendering multiple pieces of geometry such as triangles, lines, and points connected by vertices placed by the artist in a floating point value between 1 and -1 on the x and y axis which are converted to some fixed point integer. In order to maintain parallelization within the pipeline the GPU must have some sort of rule to determine the attributes of a pixel covered by more than one piece of geometry so that it does not color a pixel twice causing distortion. The following tests were used to determine these rules used as well as the number of bits and locations of sample points used to determine which triangle the pixel covers.

Triangle Rules of Rasterization: The first test carried out used to determine the triangle rules drew a red and blue triangle facing opposite directions and intersecting at a 45 degree angle and 135 degree angle through the pixel grid cutting half-way through all of the pixels along the intersection. (Note: To find the location of the pixels tested the number of pixels was confirmed to be 1000 in the x and y direction by outputting the color of the pixel in each direction until it turned black or was off of the screen). The color of the pixels along the intersection were output using the pixelRead() function and found to be red when the red triangle with its intersection on its left side was tested (as shown in figure 1, with code details in the appendix). When the triangles were drawn with the bottom flat at the center, the pixels at the intersection were blue since the side being tested is now the top of the blue and bottom of the red triangle. After multiple tests using different triangle rotations and opposite colors, it was confirmed that the rasterization rule is set to take the color of the triangle if it is the left upper side.

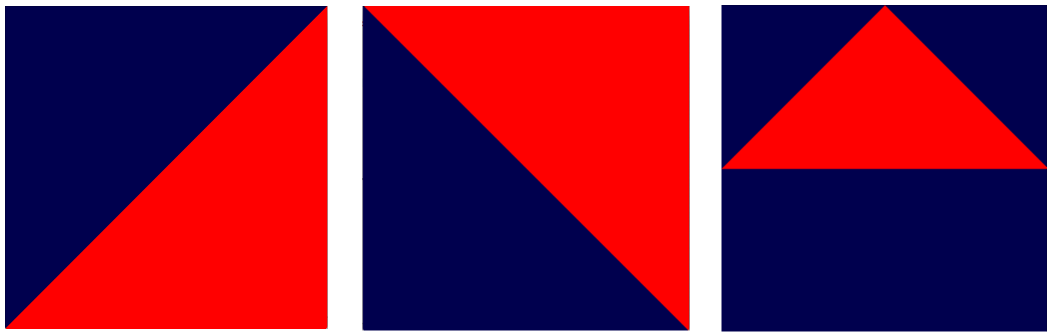

Figure 1. Triangle Intersecting at 45, 135 Degree Angles, and with Bottom

Point Rules of Rasterization: The rules for point rasterization were determined by placing the floating point input vertex point at the intersection of four pixels and the intersection of four pixels and outputting each color to see which one takes the color of the vertex. It was found that the pixel on the top right would be colored at the intersection of four pixels, the pixel on the top is colored at the horizontal intersection, and the right pixel at the vertical intersection. It was also found that when the point was placed at each corner of the screen the only time a colored pixel was rendered was in the bottom left corner since this is the only corner with a top right pixel on the grid. Example output can be found in the appendix.

Line Rules of Rasterization: The line rules of rasterization were found by placing lines on top of the intersection between pixels and with endpoints landing half-way through a pixel. When the line input in floating point was drawn horizontally through the screen (from -1.0f, 0.0f to 1.0f, 0.0f) only the pixels on the top of the intersection were red, as shown in the example test in the appendix. When the line was drawn vertically through the center of the screen (from 0.0f, 1.0f to 0.0f, -1.0f), only the pixels on the right of the intersection between pixels were colored from top to bottom of the line. The line was then drawn to intersect the pixel from the top, bottom, left, and right of the mid pixel and from each of the four corners and ending quarter-way up from the bottom of the pixel (eg. 0.0005f, 0.0005f- see precision test) and at the intersection of two pixels to see if would be colored with the line color. The pixel that was intersected quarter-way through was found to red only starting at 0.0005f when it was drawn upward as in the third image below, while the line had to pass more than three quarters-way through the pixel to be red when drawn from the top as in the fourth image below (since it must be ¼ from the pixels bottom). On the other hand, the left/right side tests (see appendix) showed that the line had to go farther through the pixel than the mid left edge (at 0.002f, 0.001f) by only 0.0001f to it to turn red. When the lines were drawn from each of the four corners of the image it was found that, similar to the top/bottom test only the line entering the pixel from the top had to be drawn farther inward than quarter way through the pixel to be colored red (these are the last two images in the set below). Since the line coming from the bottom takes priority, it can be concluded that the rasterizer uses a rule that the line must enter quarter-way through the pixel on the bottom half to render the pixel the color of the line and lines coming from the top would have to go 0.0001f farther than three quarters of the way down. This shows that the rasterizer tests inside a diamond shaped region within the pixel square to see if enough of the line has entered to color the fragment. These rules are probably implemented so that pixels are not colored when most of the line is not inside the pixel square.

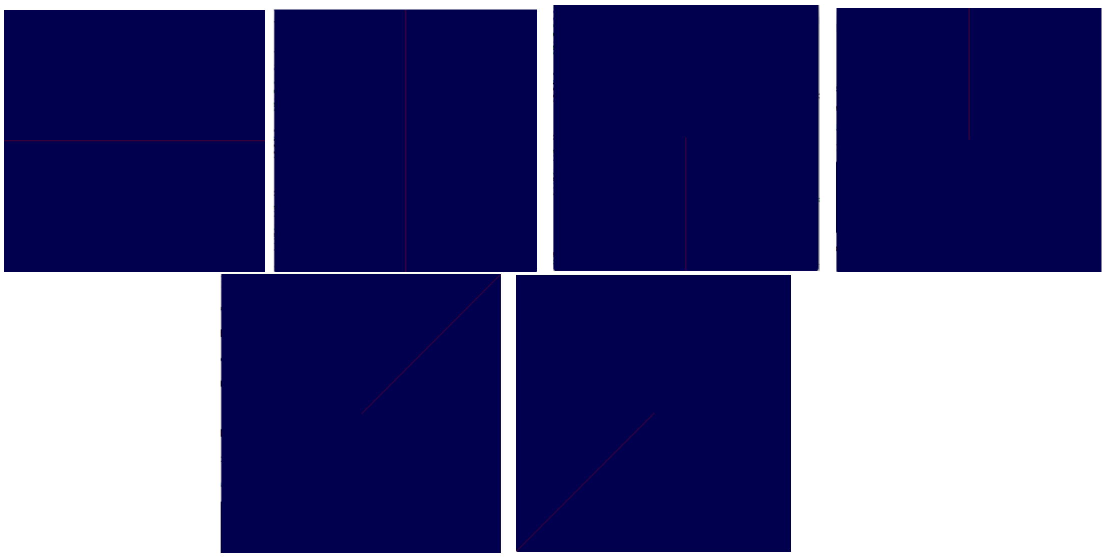
Figure 2. Line Rule Tests Output Images

Precision of Vertices and Sample Point Location: The precision of vertices was found by moving the vertices of the corners of the triangle from the first triangle test to move the line just far enough away from the sample point as to still have the color of the pixel show up as if it were still intersecting the sample point. For example, instead of placing the vertex at (-1.0f, -1.0f), the vertex is placed at -0.999f. It was found that this location through 0.99999 would still render the color of the pixel in which most of the triangle covered rather than the color using the top-left/shadow rule. When the vertex was moved to 0.999999f, the pixel rendered as if the vertex point was placed at 1.0f indicating the sub-pixel location placement of the vertex was closer to 1.0f. Further, it was found that by changing the sixth 9 to any number 6 or below it went back to the previous sub-pixel location (before 1.0f), and when a seventh digit (any 1 through 9) was added it snapped back to 1.0f. The floating point input sizes of individual pixels was confirmed to be the 0.002f during multiple point tests by drawing a point at 0.0f-0.001997f for x and y and seeing that the pixel color was red up until the same snap point precision (the vertex snaps to 0.002 at 0.001997), as shown in the example in the appendix. This makes sense because there are 1000 pixels and floating point input values are between -1 and 1 for x and y (so 2/1000=0.002). Therefore the first the bits of the floating point input hold the pixel/integer location and the second three hold the sub-pixel precision value. It is also assumed that the number of sub-pixel locations will be a power of two because dividing the pixel in half a number times will give a value that is the highest number held in a certain number of bits. Since the vertex snaps 0.000004f away from 1.0f at 0.0019961f 7 bits may be used to hold the sub pixel location (for accuracy) because it snaps at .996 and this number is just over half way through the last sub-pixel increment between the 127th and 128th (eg. 127.5*(1/128)=0.99609375).

Conclusion: Understanding how the vertices are input into the program in floating point and converted to a fixed point integer will help future GPU artists properly render their image help future GPU designers increase the efficiency of rasterizers. The geometry rules of rasterization is important because overlapping triangle edges, drawing multiple pixels per point vertex or multiple lines of pixels per line would cause a great amount of image distortion. It is better to draw the proper number of pixels and use a "randomly" chosen than slow down the graphics pipeline by having more data sent per vertex to make a calculated decision on which pixels to shade. Using single (or few) sample points is efficient because during rasterization the GPU just needs to calculate a single point. These tests also show how important it is for vertices to be converted from floating point to fixed point precision in order to speed up rasterization and sub-pixel precision allows for vertices to be placed in a number of different locations. Instead the rasterizer is built so that it holds integers for pixels and only has to do full-precision calculation once per triangle and can use the sub-pixel increments for more complex shading calculations. This prevents the overlapping parts of the image and distortion that would be caused by snapping all vertices to pixel edges.

Appendix:

Triangle Test Example Output:

```
    static const GLfloat g_vertex_buffer_data[] = {

        // 135 degree angle with red on left
         1.0f,  1.0f, 0.0f,
        1.0f,  -1.0f, 0.0f,
        -1.0f, 1.0f, 0.0f,
    };
        glReadPixels(499,500, 1, 1, GL_RGB, GL_UNSIGNED_BYTE, &p[0]);
        printf("PIXEL= %d - %d - %d \n", p[0],p[1],p[2]);

PIXEL= 255 - 0 - 0

    static const GLfloat g_vertex_buffer_data[] = {

        // 45 degree angle with blue on left
         1.0f,  1.0f, 0.0f,
        -1.0f,  -1.0f, 0.0f,
        -1.0f, 1.0f, 0.0f,
    };
        glReadPixels(499,499, 1, 1, GL_RGB, GL_UNSIGNED_BYTE, &p[0]);
        printf("PIXEL= %d - %d - %d \n", p[0],p[1],p[2]);

PIXEL= 0 - 0 - 76
```

Point Test Example Output:

```
    static const GLfloat g_vertex_buffer_data[] = {

         0.0f,  0.0f, 0.0f,
    };

        glReadPixels(499,499, 1, 1, GL_RGB, GL_UNSIGNED_BYTE, &p[0]);
        printf("PIXEL= %d - %d - %d \n", p[0],p[1],p[2]);

PIXEL= 0 - 0 - 76

        glReadPixels(499,500, 1, 1, GL_RGB, GL_UNSIGNED_BYTE, &p[0]);
        printf("PIXEL= %d - %d - %d \n", p[0],p[1],p[2]);

PIXEL= 0 - 0 - 76

        glReadPixels(500,500, 1, 1, GL_RGB, GL_UNSIGNED_BYTE, &p[0]);
        printf("PIXEL= %d - %d - %d \n", p[0],p[1],p[2]);

PIXEL= 255 - 0 - 0

        glReadPixels(500,499, 1, 1, GL_RGB, GL_UNSIGNED_BYTE, &p[0]);
        printf("PIXEL= %d - %d - %d \n", p[0],p[1],p[2]);

PIXEL= 0 - 0 - 76
```

Line Test Example Output:

Line Tests on sides of mid pixel:

```
    static const GLfloat g_vertex_buffer_data[] = {

        //
```

```
         0.002f,  0.001f, 0.0f,
        1.0f,  0.001f, 0.0f,

    };
PIXEL= 0 - 0 - 76

    static const GLfloat g_vertex_buffer_data[] = {

        //
         0.0019f,  0.001f, 0.0f,
        1.0f,  0.001f, 0.0f,

    };
PIXEL= 255 - 0 - 0
```

Line Tests for a line between two lines of pixels:

```
    static const GLfloat g_vertex_buffer_data[] = {
         // horizontal line test
         1.0f,  0.0f, 0.0f,
  -1.0f,  0.0f, 0.0f,
    };
        glReadPixels(500,500, 1, 1, GL_RGB, GL_UNSIGNED_BYTE, &p[0]);
        printf("PIXEL= %d - %d - %d \n", p[0],p[1],p[2]);

PIXEL= 255 - 0 - 0

        glReadPixels(499,500, 1, 1, GL_RGB, GL_UNSIGNED_BYTE, &p[0]);
        printf("PIXEL= %d - %d - %d \n", p[0],p[1],p[2]);

PIXEL= 0 - 0 - 76

        glReadPixels(500,499, 1, 1, GL_RGB, GL_UNSIGNED_BYTE, &p[0]);
        printf("PIXEL= %d - %d - %d \n", p[0],p[1],p[2]);

PIXEL= 255 - 0 - 0

        glReadPixels(499,499, 1, 1, GL_RGB, GL_UNSIGNED_BYTE, &p[0]);
        printf("PIXEL= %d - %d - %d \n", p[0],p[1],p[2]);

PIXEL= 0 - 0 - 76
```

Line Tests for quarter way through pixel:

```
    static const GLfloat g_vertex_buffer_data[] = {
        // intersecting from bottom point of inter-pixel diamond
         0.0005f,  -1.0f, 0.0f,
        0.0005f,  -0.0005f, 0.0f,
    };
        glReadPixels(500,499, 1, 1, GL_RGB, GL_UNSIGNED_BYTE, &p[0]);
        printf("PIXEL= %d - %d - %d \n", p[0],p[1],p[2]);

PIXEL= 255 - 0 - 0

    static const GLfloat g_vertex_buffer_data[] = {
        // intersecting from top point of inter-pixel diamond
         0.0005f,  1.0f, 0.0f,
        0.0005f,  0.0005f, 0.0f,
    };
        glReadPixels(500,500, 1, 1, GL_RGB, GL_UNSIGNED_BYTE, &p[0]);
        printf("PIXEL= %d - %d - %d \n", p[0],p[1],p[2]);
```

```
PIXEL= 0 - 0 - 76

    static const GLfloat g_vertex_buffer_data[] = {
        // intersecting from top point of inter-pixel diamond
         0.0005f,  1.0f, 0.0f,
        0.0005f,  0.0004f, 0.0f,
    };
        glReadPixels(500,500, 1, 1, GL_RGB, GL_UNSIGNED_BYTE, &p[0]);
        printf("PIXEL= %d - %d - %d \n", p[0],p[1],p[2]);

PIXEL= 255 - 0 - 0
```

Precision Test Example Output:

```
    static const GLfloat g_vertex_buffer_data[] = {

        // pixel 500,500 snaps to blue
         0.001997f,  0.001997f, 0.0f,
    };

PIXEL= 0 - 0 - 76

    static const GLfloat g_vertex_buffer_data[] = {

        //pixel 500,500 is red from x=y=0.0f to 0.001997
         0.00199f,  0.00199f, 0.0f,

    };
PIXEL= 255 - 0 - 0

    static const GLfloat g_vertex_buffer_data[] = {

        //pixel 500,500 is red from x=y=0.0f to 0.001997
         0.001995f,  0.001995f, 0.0f,

    };
PIXEL= 255 - 0 - 0
```

The full Open GL code used for the project is located at the following link:
https://github.com/prag93/EEC_277_winter2017/tree/master/Final_Project

These tests were administered in OpenGL 4.5 and Intel Iris Pro Graphics.