Browser:
Firefox version: 48.0.2

Bresenham Lines:
For the Bresenham Line algorithm I started with the algorithm given in class by the teacher

Initialization:
(x0,y0) = first point on line
P0 = 2*dy-dx
Drawing:
Repeat dx times
Xk+1 = xk + 1
If (pk < 0)
    Yk+1 = yk // This line is unnecessary
    Pk+1 = pk + 2*dy
Else
    Yk+1 = yk + 1
    Pk+1 = pk + 2*dy – 2*dx

Then I realized to do the negative lines (when end_x > start_x and/or when end_y > start_y) we need to decrement instead of increment. So I added a x and y directional incrementor.
I drew this like an X shape on a piece of paper to figure out how to traverse the negative slope points.

```
var sx = (vs[0] < ve[0]) ? 1 : -1;//increment direction x
var sy = (vs[1] < ve[1]) ? 1 : -1;//increment direction y
```

By this point I figured out that the coordinate [0,0] was on the upper right hand corner on our “canvas”. I recognized this because when I was hard code testing with very little numbers less than 100 all the lines were in the upper right hand side.
Then I saw that for very vertical lines we run out of x faster than y. So I needed to calculate along Y instead of X. This would require me to swap all the x-coordinate variables into y-coordinate variables. I did this with this IF statement:

```
if (dx > dy){}else{//swap x and y }
```

The hardest part for me to figure out was the swap when the lines were very vertical. I did not know what condition to swap for. I did not even know that I needed to swap. The reason I was stuck was because when I wanted to draw triangle outlines (not filled) I kept getting unfinished triangles. There would only be two lines of the triangle and the third line would be missing or somewhere else.

So, I first ran the drawLineSegment() code for one minute to see what kind of lines I was missing. I saw that none of the lines were more than about 45-degrees vertical! I then started drawing on a paper vertical lines and some horizontal lines. That is when I saw that x finished faster than y in vertical lines. So I turned my drawing sideways so that y was now the horizontal axis. Thus, when there is more y-coordinates than x we need to pretend y is the new x!

That is how I figured out Bresenham’s line algorithm code.

Scanline Triangles:
Now, for the scanline triangles I knew from class, that if the triangle was not flat then we need to divide the triangle in two pieces with a fourth point connected to the middle vertex and parallel to the x-axis. So to find that x-coordinate of the new point/vertex I got the equation:

```
(vt0[0]) + ((vt1[1] - vt0[1]) / (vt2[1] - vt0[1])) * (vt2[0] - vt0[0])
```

But I realized that I was assuming the order of the vertices given to me. Thus I had to first sort the vertices. I wanted them in ascending Y order(v1[y]<=v2[y]<=v3[y]). So I made sortByY() helper function with following pseudo- code:

If (v1[y] > v2[y]){switch}
If (v1[y] > v3[y]){switch}
If (v2[y] > v3[y]){switch}

Now, after getting the fourth point and having the vertices in order I can now start filling the flat triangles. The scanlines just meant that I would pass the lines into the Bresenham function. But I saw that when we have a flat-triangle we need to start at the longest x-side (the edge of the triangle, with the most points along the x-axis, and parallel to the x-axis) then work our way up or down the y-axis while trimming away the x-points. So the longest x-side in a bottom-flat-triangle(/\ ), is the bottom and we need to go UP the y-axis so we increment. Then for a top-flat-triangle(\/), the longest x-side is the top, so we need to go DOWN the y-axis, thus we will decrement.

Therefore, I now had three cases:

1 – triangle is flat-top
2 – triangle is flat-bottom
3 – triangle needs to be split into two flat triangles using fourth point

So, I put in these three conditions and made the topFlatTriangle() and bottomFlatTriangle() helper functions to do the trimming and filling.

The hardest part for me here was the Y increment/decrement with trimming. I looked online for an idea of how to trim the sides while moving along Y. So I found that we can use the slopes of the two non-flat (non-horizontal) lines to compute the x-trim along the y-axis. Also, since my vertices were sorted, then I knew exactly which lines were horizontal. So, in bottom-flat triangles my v1 was the flat side and in the top-flat triangles my v3 was the flat side.

Finally, I just added another coloring for-loop to draw random-colored lines to prove I am using scanline method. That is how I computed scanlines.

Input/Output:
Clicking the toggle buttons in the browser will generate input and output

Browser:
Firefox version: 48.0.2