Lab 2

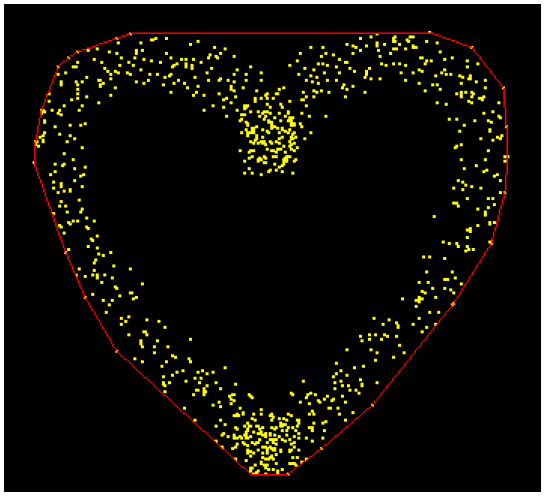
Alex Washburn, Zaynab Azzouzi

(1) brief description of what sort of inputs constitute degeneracies for the algorithm and how you handled these degeneracies;

1. Collinear points constitute degeneracies for the algorithm. In order to handle them, we modified our sorting comparator. What did the trick was that if points are collinear, we would sort them by increasing distance from p0. This works because we would go to the closer points first so they would be the ones popped off in favor of a more extreme (and eventually the most extreme) collinear point.
    a. A subcategory issue of collinearity that we ran into was actually within our sorting solution. In general, our algorithm deletes collinear points and only adds the extreme collinear point to the hull by utilizing left_strictly() when adding points. This works hand in hand with our sorting algorithm which is where our issue comes in. We were using manhattan distance to compute the distance from collinear points to p0. Because of this when we tried our algorithm on a horizontal or vertical line we would get incorrect results which stemmed from incorrect sorting. It turned out that points that were the farthest from p0 were getting classified as the closest because the distance was becoming negative (point.x- p0.x for horizontal lines, and point.y-p0.y for vertical lines). To fix this we changed our code to take the absolute value of the distance to ensure we only took into account the magnitude of the distance not the direction.

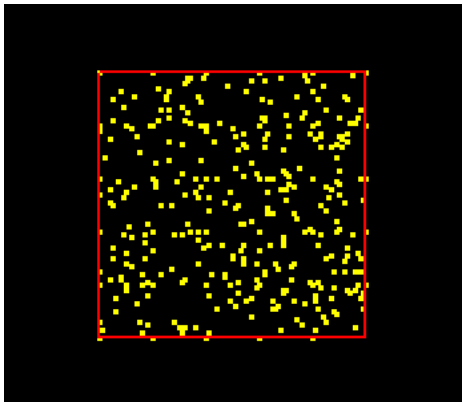
(2) pictures of the two initializers you created

1. Heart



```cpp
/* initializes n points in the shape of a heart
 * used equation from Wolfram MathWorld
 */
void initialize_points_heart(vector<point2d>& pts, int n) {

  printf("\ninitialize points heart\n");
  pts.clear(); // clear it out for safety

  double t = (2*M_PI) / n; // defining step size this way will give us n points
  point2d p;
  int SCALING_FACTOR = 100; // ensures that the heart is a good size

  for (double a = 0; a<2*M_PI; a+=t) {
    p.x = WINDOWSIZE/2 + SCALING_FACTOR*(sqrt(2) * sin(a)*sin(a)*sin(a))+ (random() % ((int)(.07*WINDOWSIZE)));
    p.y = WINDOWSIZE/2 + SCALING_FACTOR*(-cos(a)*cos(a)*cos(a) - cos(a)*cos(a) + 2*cos(a)) + (random() % ((int)(.07*WINDOWSIZE)));
    pts.push_back(p);
  }
  printf("heart inititalied with %lu points\n", pts.size());
}
```

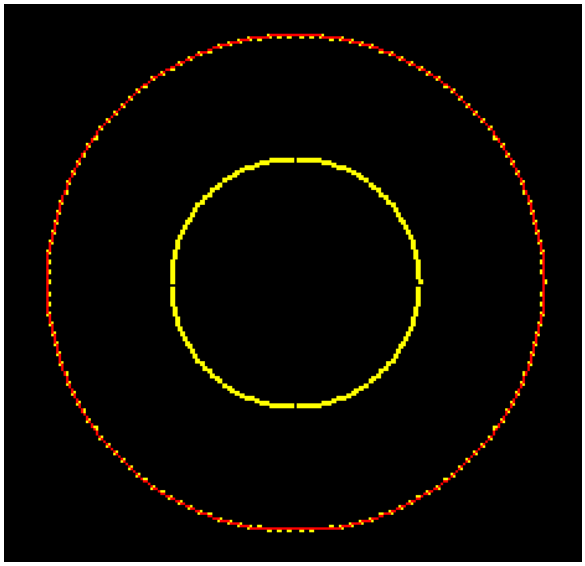
2. Square, filled with random points:



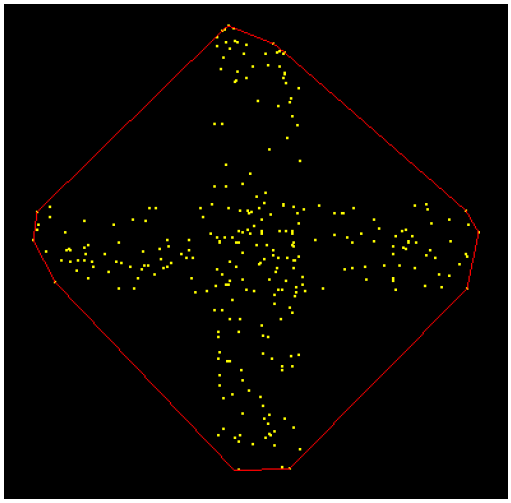
```
/*this initializer will always make 25 hardcoded points that are meant to test collinearity
 * user can input a number for parameter n and n points will be created in addition to the
 * 25 original points
 */
void initializer_square(vector<point2d>& pts, int n_additional_points){

  pts.clear(); //should be empty, but clear it to be safe
  for(int i = 200; i <= 300; i+=20){
    for(int j = 200; j <= 300; j+=20){
      point2d p;
      p.x = i;
      p.y = j;
      pts.push_back(p);
    }
  }
  //we are using user input to create n points in addition to our hard coded collinear points
  for (int i=0; i<(n_additional_points); i++) {
    point2d p_rand;
    p_rand.x = (int)200 + random() % ((int)(100));
    p_rand.y =  (int)200 + random() % ((int)(100));
    pts.push_back(p_rand);
  }
}
```

(3) pictures of the other (given) initializers you used;

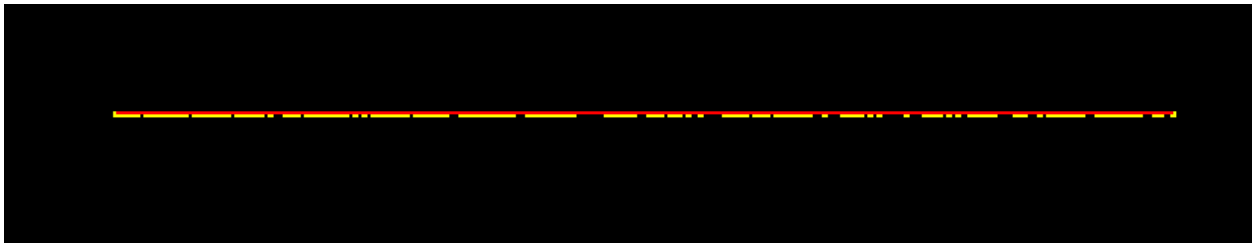
Circle:



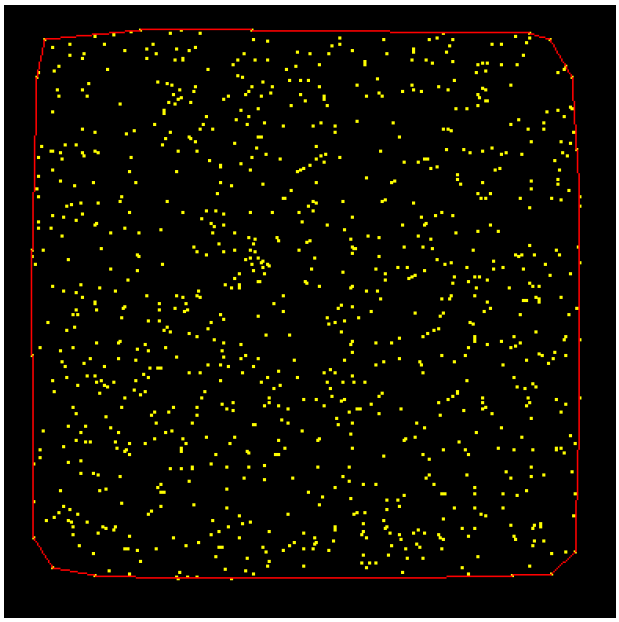
Cross:



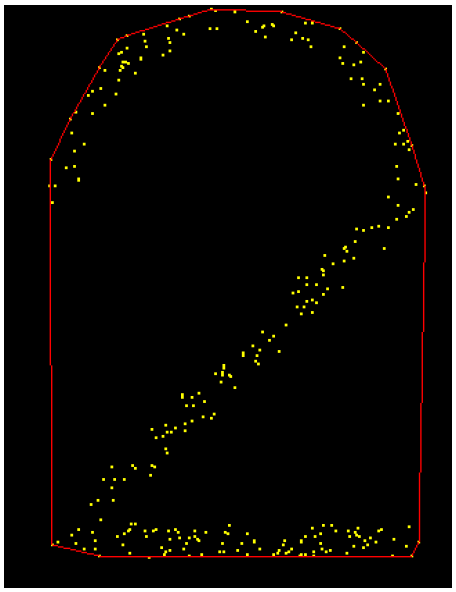
Horizontal line:



Random points:

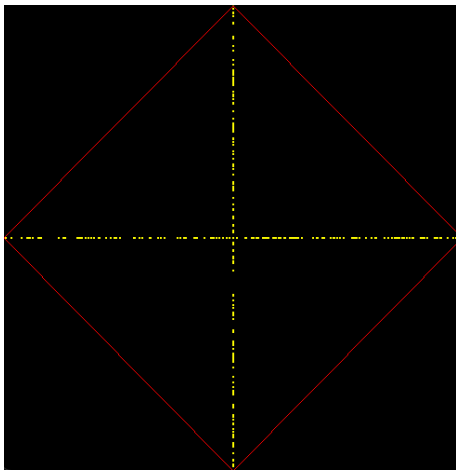


Tom's case:



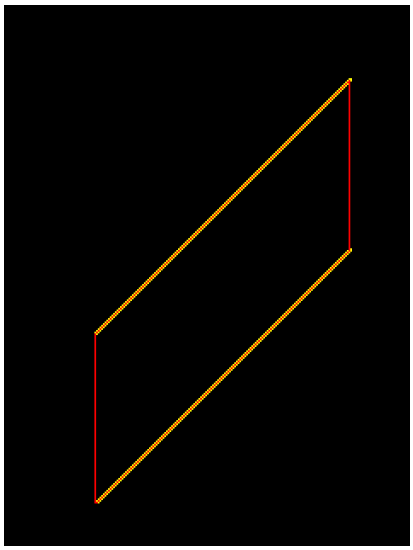
```
//1. tom's example
void initialize_points_2(vector<point2d>&pts, int n){
  printf("\ninitialize points 2\n");
  pts.clear();
  assert(pts.size() == 0);

  point2d p;
  int x_noise, y_noise;
  int pos, pos2;
  for (int i = 0; i < n; i++)
  {
    switch (i % 3) {
      case 0:
        p.x = (int)(0.25*WINDOWSIZE) + random() % ((int)(0.5*WINDOWSIZE));
        p.y = (int) (0.15*WINDOWSIZE);
        break;
      case 1:
        pos = random() % (int)(0.5*WINDOWSIZE);
        p.x = pos; p.y = pos;
        p.x += (int) (0.25*WINDOWSIZE);
        p.y += (int) (0.15*WINDOWSIZE);
        break;
      case 2:
        pos2 = random() % 180;
        p.x = (int)(0.5 * WINDOWSIZE) + (int) ((0.25*WINDOWSIZE) * cos((M_PI * pos2)/180));
        p.y = (int)(0.65 * WINDOWSIZE) + (int) ((0.25*WINDOWSIZE) * sin((M_PI * pos2)/180));
        break;
    }
    x_noise = random() % ((int) (0.05*WINDOWSIZE));
    y_noise = random() % ((int) (0.05*WINDOWSIZE));
    p.x += x_noise;
    p.y += y_noise;
    pts.push_back(p);
  }
}
```

Max and Abhi:



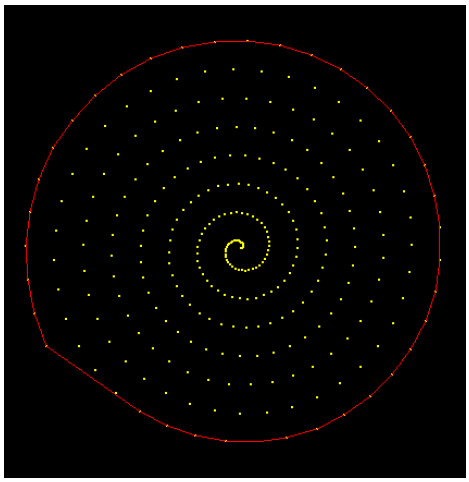
```
//2. Max and Abhi
void initialize_points_thin_cross(vector<point2d>&pts, int n) {
  printf("\ninitialize points thin cross\n");
  pts.clear();
  point2d p;
  for (int i = 0; i < n; i++) {
    // put points on horizontal line
    if (i % 2 == 0) {
      p.x = random() % (int)(WINDOWSIZE);
      p.y = WINDOWSIZE/2;
    }
    // put points on vertical line
    if (i % 2 == 1) {
      p.x = WINDOWSIZE/2;
      p.y = random() % (int)(WINDOWSIZE);
    }
    pts.push_back(p);
  }
}
```

Leah:



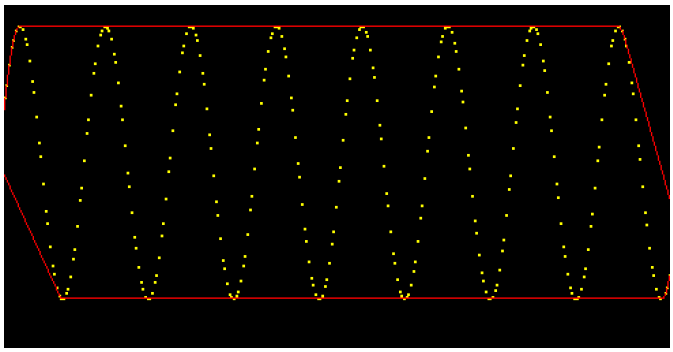
```
//3. Leah
void initialize_points_stripes(vector<point2d>& pts, int n) {
  printf("\ninitialize points stripes\n");
  pts.clear();
  double step = WINDOWSIZE/n;
  n = n/2;

  point2d p;
  point2d q;
  for (int i = 0; i < n; i++) {
    float t = step*i;
    p.x = t + WINDOWSIZE/4;
    p.y = t + WINDOWSIZE/5;
    q.x = t + WINDOWSIZE/4;
    q.y = t + 2*WINDOWSIZE/5;
    pts.push_back(p);
    pts.push_back(q);
  }
}
```

David:



```cpp
// 4. David
/* ***************************** */
/* Initializes pts with n points forming a spiral shape.
   The points are in the range [0, WINSIZE] x [0, WINSIZE].
*/
void initialize_points_spiral(vector<point2d>& pts, int n) {
    printf("\ninitialize points spiral\n");
    pts.clear();

    double centerX = WINDOWSIZE / 2; // center of circle
    double centerY = WINDOWSIZE / 2; // center of circle

    double theta_increment = 1 / M_PI; // angle increment for the spiral
    double max_radius = WINDOWSIZE / 2.5; // maximum radius to keep the spiral within bounds
    double radius_increment = max_radius / (n / 10.0); // increment based on number of points

    for (int i = 0; i < n; i++) {
        double theta = i * theta_increment * 0.5; // angle increases as i increases
        double radius = radius_increment * i * 0.1; // scale the radius to keep it spiral-like
        radius = fmin(radius, max_radius);  // ensures the radius doesn't exceed the boundary

        point2d p;
        p.x = centerX + radius * cos(theta);
        p.y = centerY + radius * sin(theta);

        pts.push_back(p);
    }
}
```

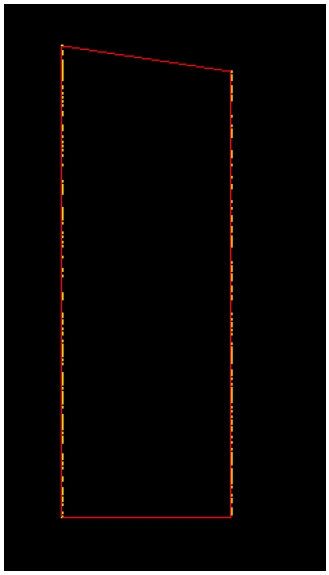
Jack and Manny:



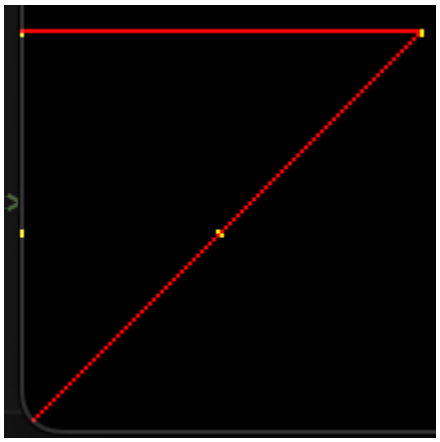
```cpp
// 5. Manny and Jack
void initialize_points_wave(vector<point2d>& pts, int n){
printf("\ninitialize points wave\n");
  //clear the vector just to be safe
  pts.clear();
  double step = (double)WINDOWSIZE / n;
  double amplitude = 100;  // Height of the wave
  double frequency = 0.1;  // Controls the number of waves
  point2d p;
  for (int i = 0; i < n; ++i) {
    p.x = i * step;
    p.y = WINDOWSIZE / 2 + amplitude * sin(frequency * p.x);
    pts.push_back(p);
  }
}
```

(EXTRA CASES)

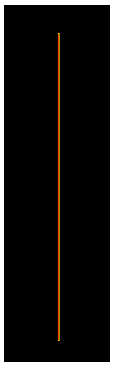
Two vertical lines:



Hardcoded triangle: *note there is a line that connects the left side, but it isn't rendered properly in the image window. However it exists in the convex hull we computed



Vertical line: ***the convex hull returned was just the two extreme points**



(4) pictures of the convex hulls computed for each initial configuration of points — for all these choose a reasonable value of n

All of the photos above have convex hulls computed as well as the initialized points - we hope this is ok!

(5) if your code does not work in all cases, explain.

1. Our code worked in all of our test cases, including the test cases with collinearity, a shape with less than 3 vertices, etc.

(6) any extra features.

1. We had no extra features

(7) Time you spent in: Thinking; Programming; Testing; Documenting; Total;

1. We spent most of our time during this lab *thinking* and *testing* (debugging). A first bug that took a while to overcome was the sorting: for some reason, using std:sort did not work correctly for us, even after checking the logic of the comparator multiple times. However, we used qsort(), which was slightly more rigorous to implement due to some data casting we needed to do to meet the requirements of the function: however, the sorting was done correctly *every time* after we started using qsort instead of std:sort. Another odd bug was that when we computed the Manhattan distance to break ties between two collinear points in sorting, we used the signed distance and not the magnitude, which caused a few problems for linear lines. We fixed this easily, but it was hard to track down the issue.

   Following these bugs, we kept thinking that we had everything worked out and then we'd come across another edge case that messed up our code, such as some collinearity. When this happened we spent a lot of time drawing out how our algorithm was building the hull and finding the holes in its logic. In comparison to the thinking and testing, we didn't program for very long because our code was fairly straight forward: all we had to do was implement some initializers and the Graham scan algorithm. We changed our comments from within the methods into official documentation at the top of each function the day before we submitted. We spent all of the time given for the lab, but this is because for a good portion of the lab we were over-complicating things.

(8) Brief reflection prompts (you don't need to address all): how challenging did you find this project? What did you learn by doing this project? What did

you wish you did differently? If you worked as a team, how did that go? What would you like to explore further?)

1. We didn't find this project super challenging, it was a fair assessment of what we learned. It helped us see the applications of finding the signed area of points and the power of categorizing points as left of, left strictly, and collinear. We learned about the code that goes into drawing points and shapes as well as got a refresher on comparators and sorting. Lastly we learned about the function graham scan and how to implement it. As mentioned earlier we over-complicated the lab when it came to sorting. We probably would have finished sooner had we caught our mistake earlier, so we wish we did that differently. We worked in a team and found it very helpful to bounce ideas off each other. Sometimes one person could see the issue with the code or a new edge case faster and could help explain it to the other.