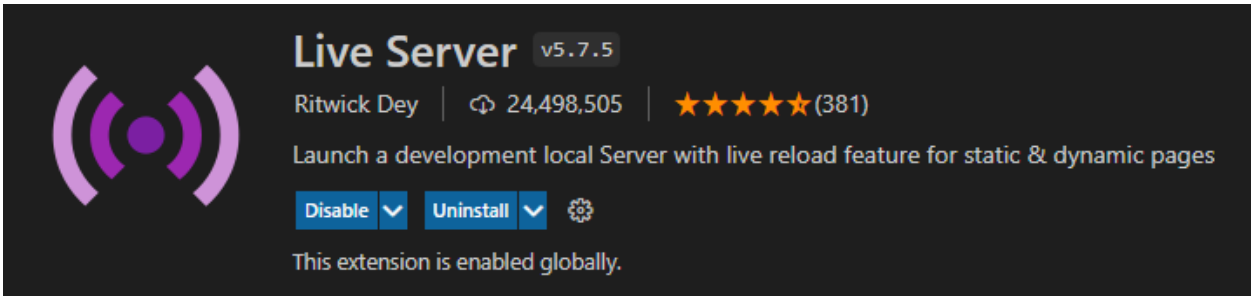
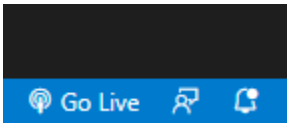
Installation

Step 1: Install VS Code

Step 2: Install extension “Live Server”



Step 3: Click “Go live” to start the server (see bottom of VS Code).



Note: Server will auto start after step 3, if not then manually go to http://localhost:5500/ to see Demos.

1) Timothy Chan's Convex Hull Algorithm

For this project I implemented Chan's O(n.log (h)) convex hull algorithm. The premise is fairly simple. Take a set of points, divide them up into maxsize n/m subsets, where m is the size of the full convex hull. Since we don't know what m is, we guestimate it with 2^{2^t} where t is each iteration (so it goes 4, 16, 256, etc). Once we divide up the points, we then run graham scan on each subset. Once each subset is graham scanned into subhulls, we then find the tangent point of each hull to the most recently added point to the convex hull, and run Jarvis march over the points, keeping the one that give the largest angle.

Time Complexity: O(n.log (h))

Space Complexity: O(2^{2^t})

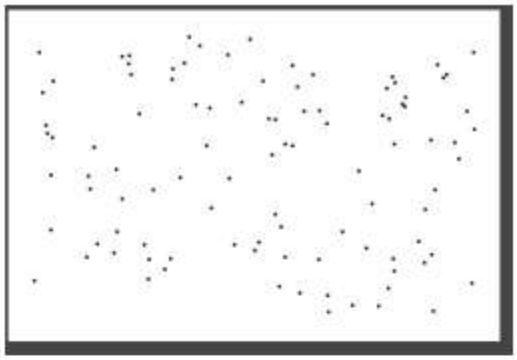
Code/ Snippet

```
function grahamScan(points) {
        let v_lowest = points[0];
        //find lowest point
        for (let i = 1; i < points.length; i++) {
            if (points[i].y > v_lowest.y) {
                v_lowest = points[i];
            }
            else if (points[i].y === v_lowest.y && points[i].x < v_lowest.x) {
                v_lowest = points[i];
            }
            //console.log(v_lowest);
        }
        if (v_lowest)

            //sort the array by angle
            points.sort(function (a, b) {
                //a is lowest point
                if (a.y === v_lowest.y && a.x === v_lowest.x) return -1;
                //b is lowest point
                if (b.y === v_lowest.y && b.x === v_lowest.x) return 1;
                //neither

                let ang_a = angleToPoint(v_lowest, a);
```

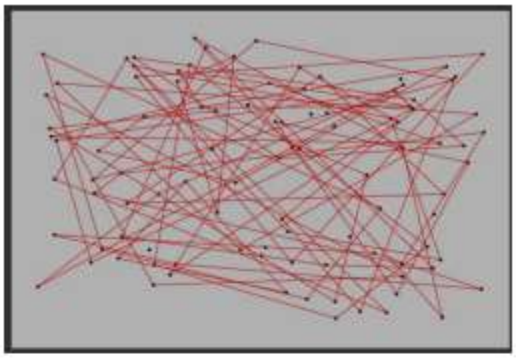
```
                let ang_b = angleToPoint(v_lowest, b);
                if (ang_a > ang_b) return 1;
                else return -1;
            });

        //remove doubles
        let i = 0;
        while (i < points.length - 1) {
            if (points[i].x === points[i + 1].x && points[i].y === points[i +
1].y) {
                points.splice(i + 1, 0);
            }
            i++;
        }

        let stack = [];
        if (points.length < 4) return points;
        //yeah im hardcoding this but we're not doing
        //graham scan with < 4 things anyway i think having
        //2 is ok

        stack[0] = points[0];
        stack[1] = points[1];
        let current = points[2];
        let index = 2;
        let stacklen = 2;
        while (index < points.length) {
            stacklen = stack.length;
            //console.log(stacklen);
            if (stacklen > 1) {//make sure there's at least 2 things before left
test
                let l = checkIfLeftTurn(stack[stacklen - 2], stack[stacklen - 1],
points[index])
                if (l) {
                    stack.push(points[index]);
                    index++;
                }
                else {
                    stack.pop();
                }
            }
            else {
                stack.push(points[index]);
                index++;
            }
```

```
        }

        return stack;

    }
```

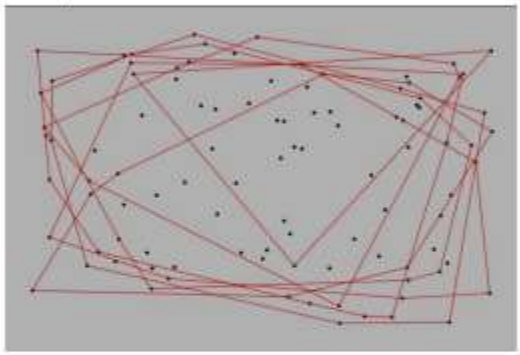
Demo Screenshots



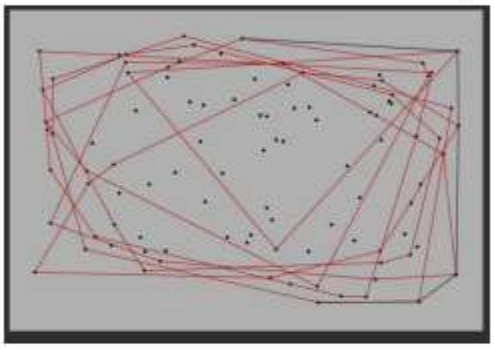
100 points.



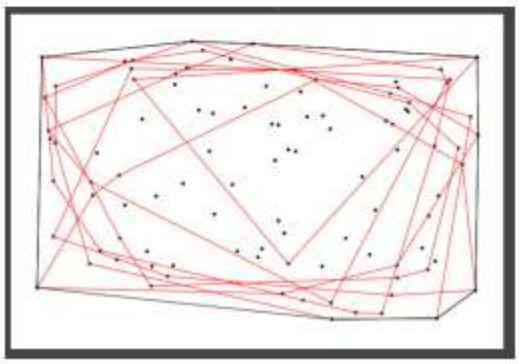
m=4. Looks like a mess. There *is* 100 points.



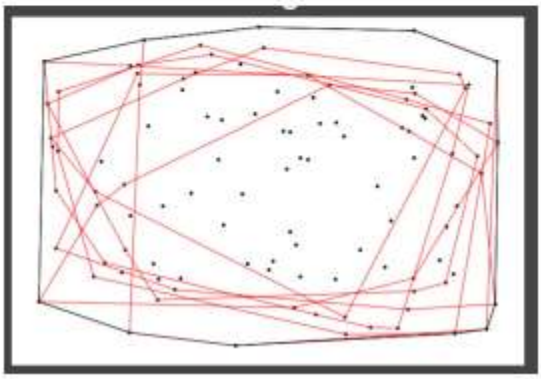
m = 16. Much cleaner.
This looks likesomething we can use.



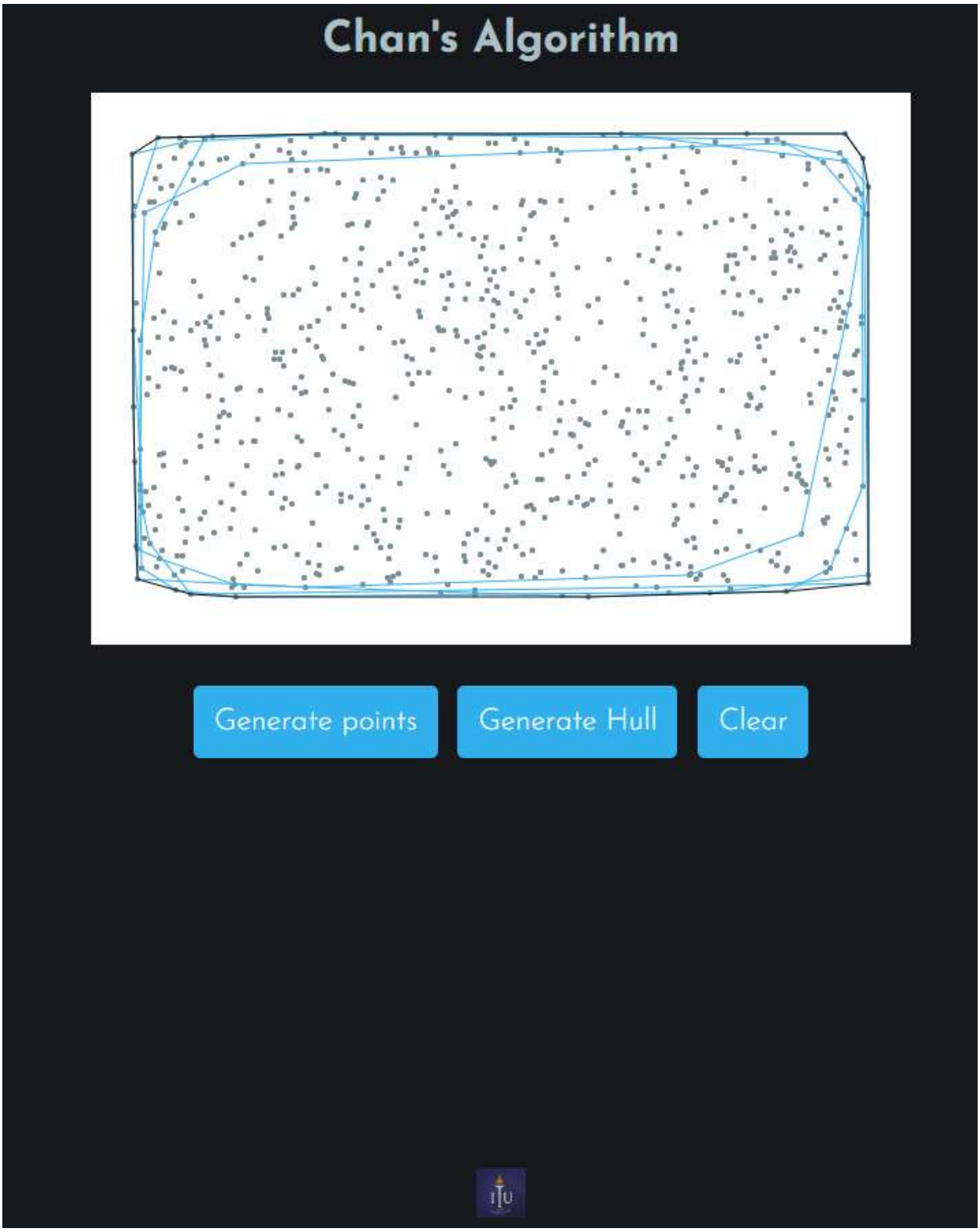
Black shows the outer hull in Jarvis march.



Complete Convex hull.



Manually add more points? Still works.



Reference Paper: https://www.cs.umd.edu/~patras/patra2012_convexhull_report.pdf

2) Steven Fortune's Voronoi Algorithm

A Javascript implementation of Steven Fortune's algorithm to efficiently compute Voronoi diagrams.

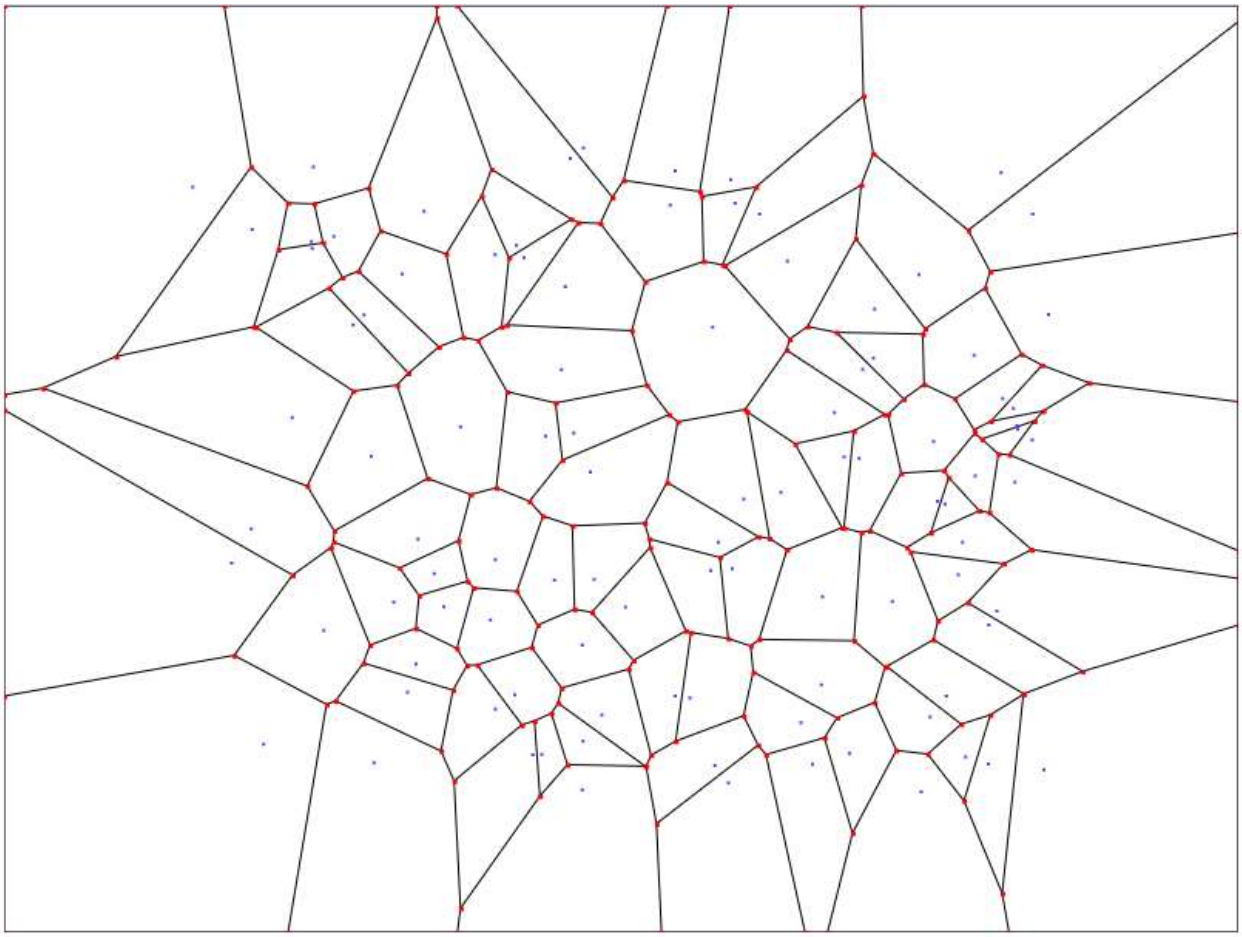
```
var voronoi = new Voronoi();
var bbox = {xl: 0, xr: 800, yt: 0, yb: 600}; // xl is x-left, xr is x-right, yt
is y-top, and yb is y-bottom
var sites = [ {x: 200, y: 200}, {x: 50, y: 250}, {x: 400, y: 100} /* , ... */ ];

// a 'vertex' is an object exhibiting 'x' and 'y' properties. The
// Voronoi object will add a unique 'voronoiId' property to all
// sites. The 'voronoiId' can be used as a key to lookup the associated cell
// in diagram.cells.

var diagram = voronoi.compute(sites, bbox);
```

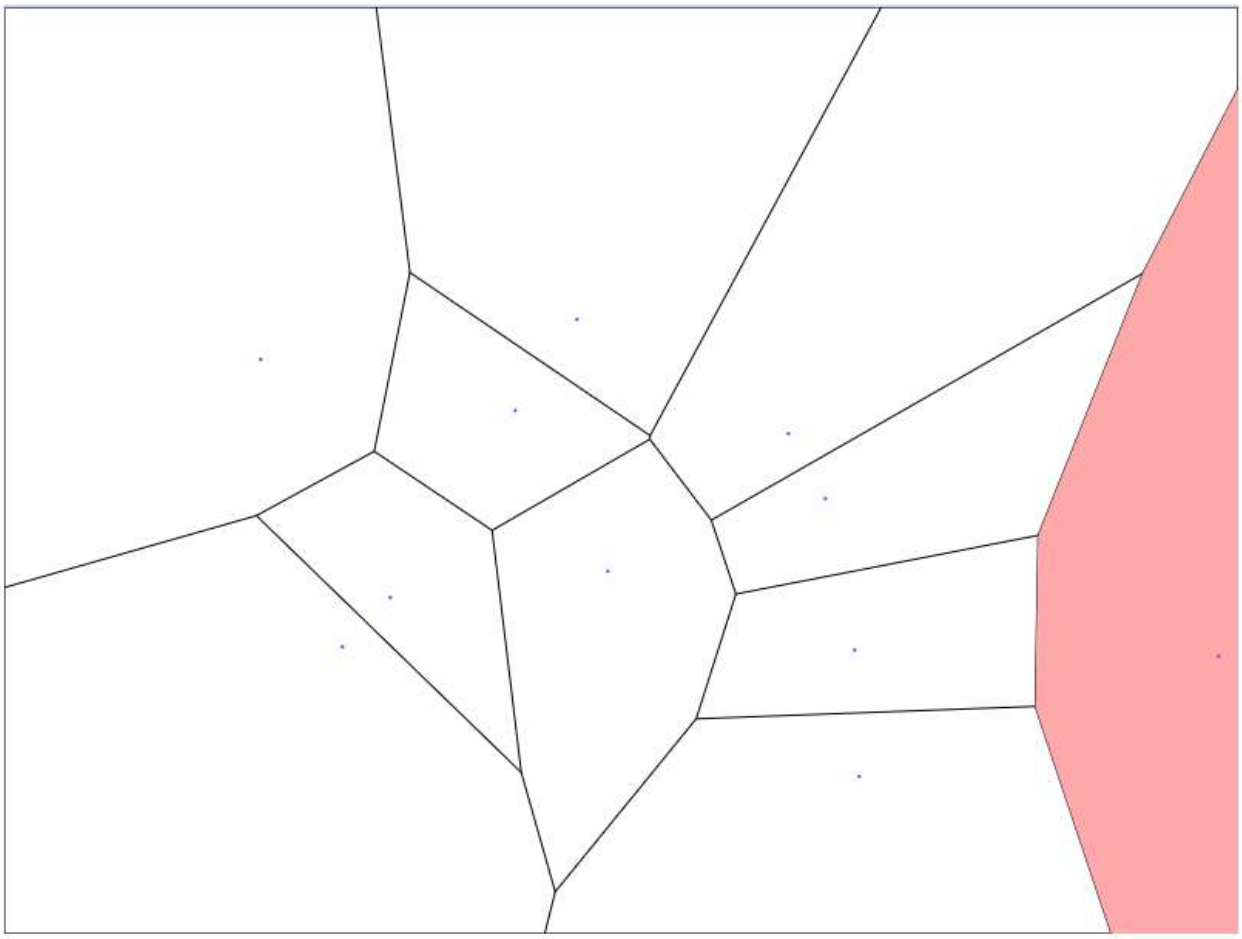
Reference Paper: https://www.cs.umd.edu/class/spring2020/cmsc754/Lects/lect11-vor.pdf

Time Complexity: O(n.log(n))

Space Complexity: O(log(n))

Demo Screenshots





3) Line-segment Intersection

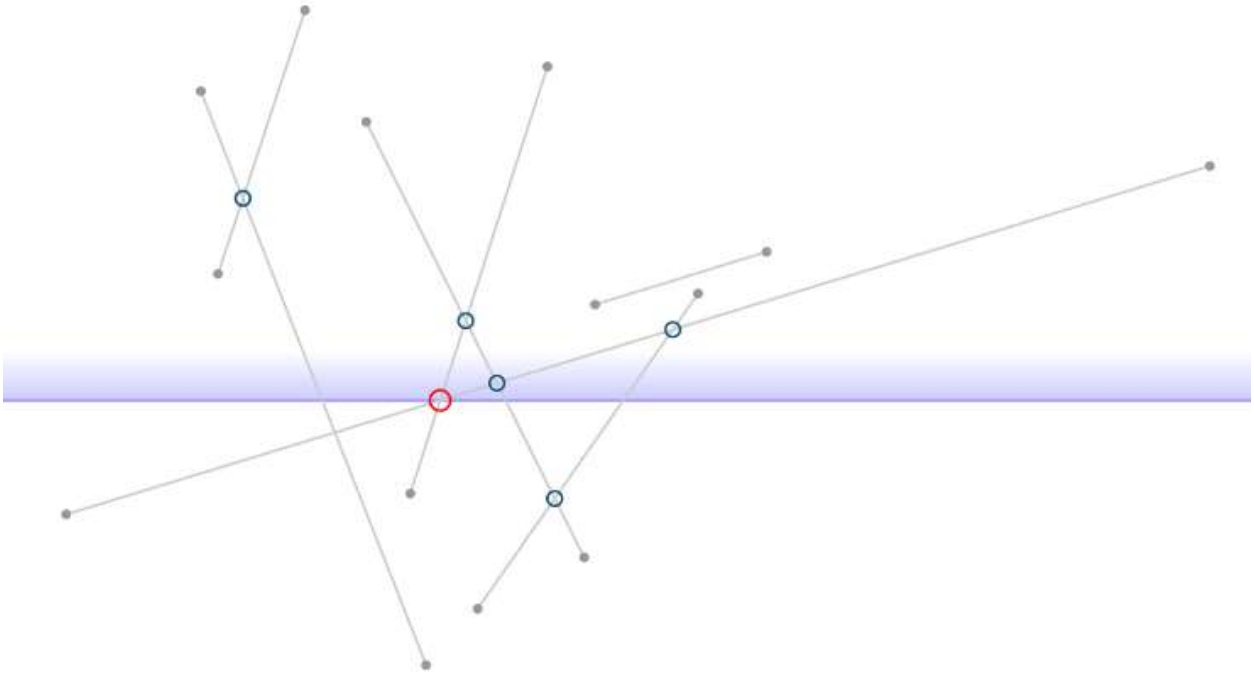
Events points and sweep line status is stored using 2 different data structures Queues and Balanced Binary tree.

Code/ Snippet

```
    init: function () {

        this.points = new js_cols.RedBlackSet( this.compare );
        this.events = new js_cols.RedBlackSet( this.compare );
        this.status = new js_cols.RedBlackSet( this.compare );

        SWEEP.SVG.init();
        SWEEP.Gui.init();
        SWEEP.Info.init();
        SWEEP.Sweepline.init();
        SWEEP.SVG.resize();
```

```
        SWEEP.Input();

        animate();
        function animate() {
            requestAnimationFrame( animate );
            TWEEN.update();
        }

    },
```

Demo Screenshot

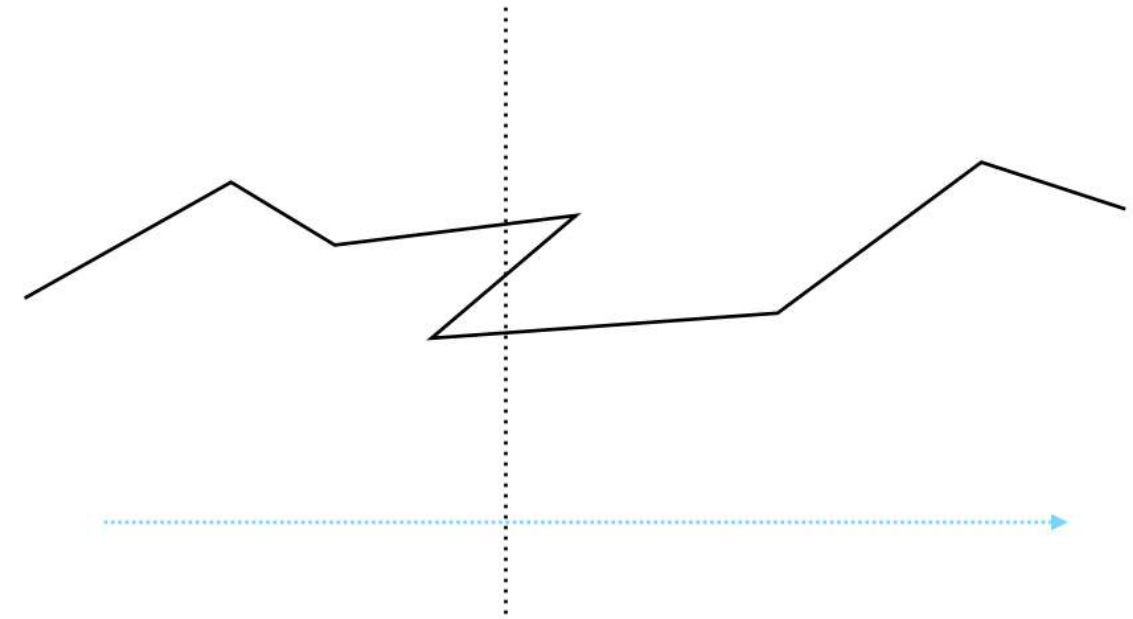


Time Complexity: O([n+i].log(n)) [i – insertion points]

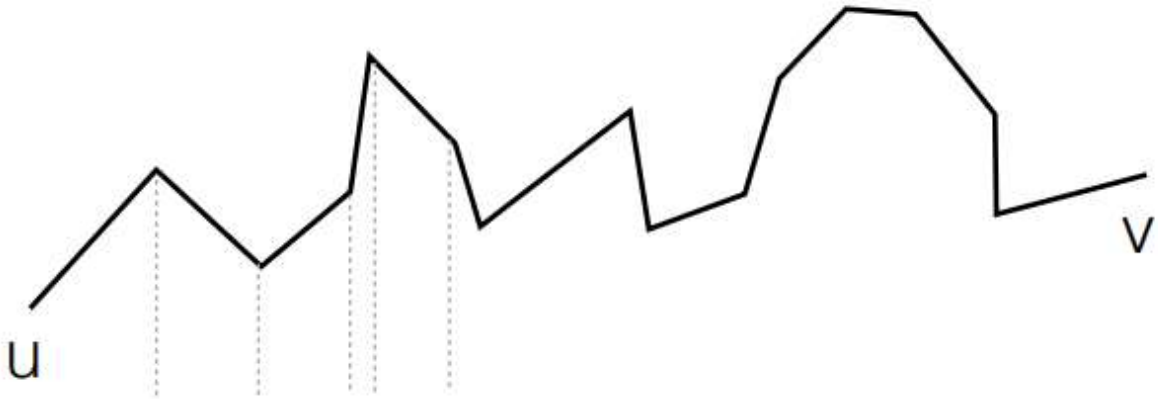
Space Complexity: O(n)

Reference: sweep.js (npm package library)

4) Triangulation of Simple Polygon using x-monotone Subdivisions



Not x-monotone



x-monotone

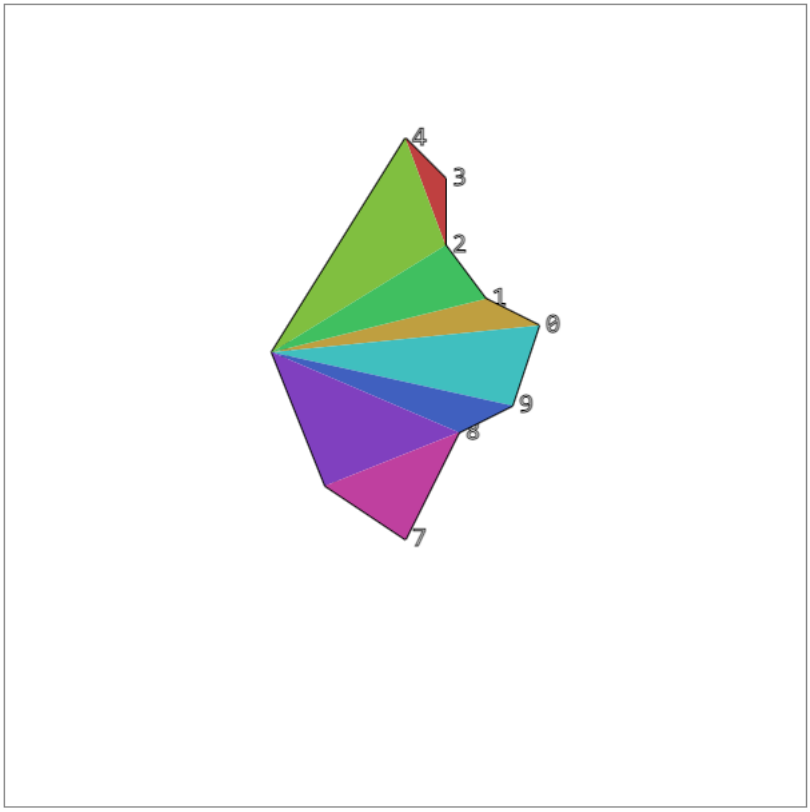
Code/ Snippet

```
function* triangulatePolygon(polygon) {
  const u = _.sortBy(polygon.vertices, pt => -pt.y);
  const n = polygon.vertices.length;
  const S = [0, 1];
```

```
const chainOf = vertexToChain(polygon);
console.log({ chainOf });

const vicinityOf = {};

function prev(idx) {
  if (chainOf[idx] == "l") return idx - 1 >= 0 ? idx - 1 : n - 1;
  if (chainOf[idx] == "r") return idx + 1 < n ? idx + 1 : 0;
}

function next(idx) {
  if (chainOf[idx] == "r") return idx - 1 >= 0 ? idx - 1 : n - 1;
  if (chainOf[idx] == "l") return idx + 1 < n ? idx + 1 : 0;
}

for (let i = 0; i < n; ++i) {
  vicinityOf[i] = [prev(i), next(i)];
}

function vertexWithId(id) {
  return polygon.vertices[id];
}

function* introduceDiagonal(a, b) {
  vicinityOf[a.id].push(b.id);
  vicinityOf[b.id].push(a.id);
  const thirdWheels = _.intersection(vicinityOf[a.id], vicinityOf[b.id]);
  for (let wheel of thirdWheels) {
    const vertices = polygon.vertices;
    yield [a.id, b.id, wheel].map(vertexWithId);
  }
}

for (let j = 2; j < n - 1; ++j) {
  console.log(j, "" + S);
  const head = S[S.length - 1];
  if (chainOf[u[j].id] !== chainOf[u[head].id]) {
    console.log("different chain");
    while (S.length) {
      const head = S.pop();
      if (S.length === 0) break;
      yield* introduceDiagonal(u[j], u[head]);
    }
    S.push(j - 1);
    S.push(j);
```

```
    } else {
      console.log("same chain");
      let lastPopped = S.pop();
      while (S.length) {
        const head = S[S.length - 1];
        console.log({ S, head });

        if (ccw(u[j], u[lastPopped], u[head]) <= 0)
          break;

        lastPopped = S.pop();
        yield* introduceDiagonal(u[j], u[head]);
      }

      S.push(lastPopped);
      S.push(j);
    }
  }

  S.pop();
  while (S.length > 1) {
    const head = S.pop();
    yield* introduceDiagonal(u[n - 1], u[head]);
  }
  console.log({ S });
}
```

Demo Screenshot



Time Complexity

First, the simple polygon is decomposed into a collection of simpler polygons, called monotone polygons. This step takes O(n log n) time.

Second, each of the monotone polygons is triangulated separately, and the result are combined. This step takes O(n) time.

Reference:

1) https://tildesites.bowdoin.edu/~ltoma/teaching/cs3250-CompGeom/spring16/Lectures/cg-polygontriangulation.pdf
2) https://www.cs.umd.edu/class/spring2020/cmsc754/Lects/lect05-triangulate.pdf