TAREFA 4 - GC 2021.1

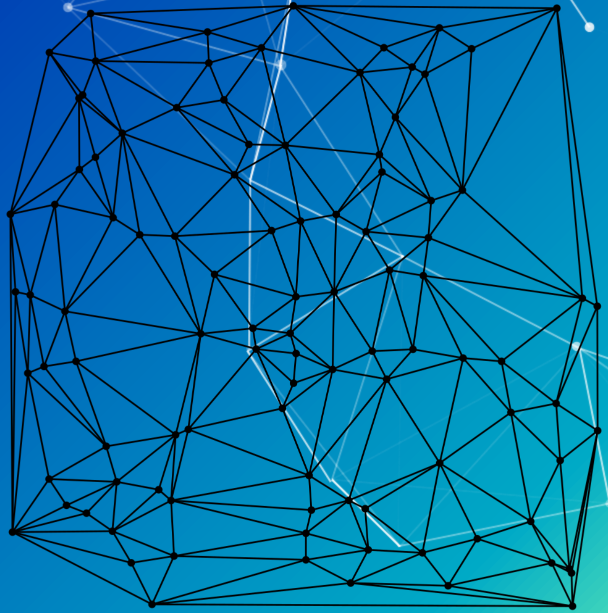
Carolina Herbster

IMPLEMENTAÇÃO

- Javascript
- Babylon.js
- Script pequeno, então contido em uma única página HTML
- Funções para realizar as triangulações e Voronoi
- Visualizações construídas a partir dos resultados
- Site disponível em: https://carolhmj.github.io/delaunay-voronoi/

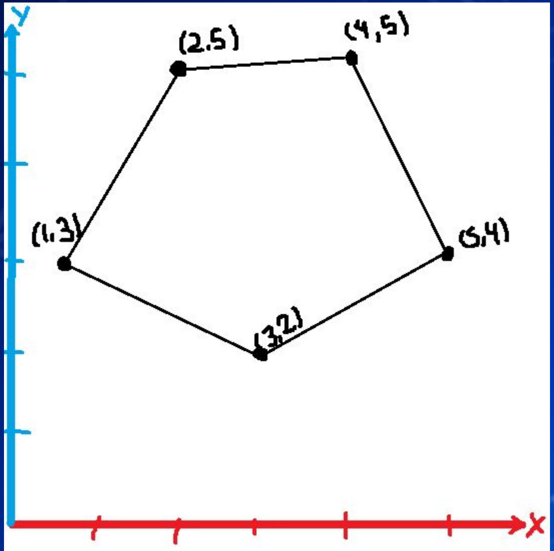
QUESTÃO 01

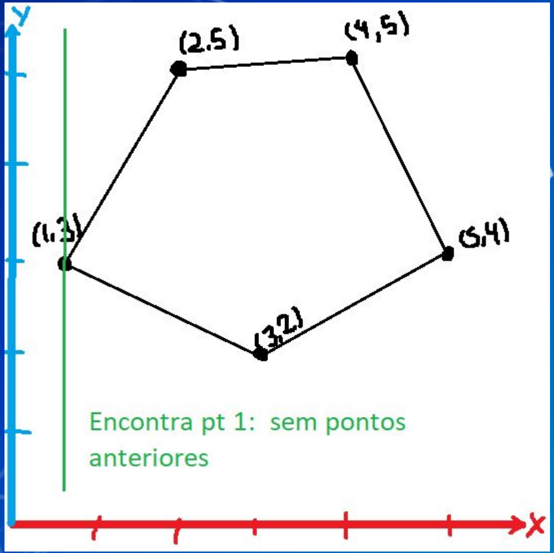
- Para resolver o problema de ponto em polígono em O(nlogn), podemos realizar uma triangulação do polígono em tempo O(nlogn) e depois percorrer os triângulos criados, calculando as coordenadas baricêntricas do ponto no triângulo, o que leva tempo O(n). Se o ponto estiver fora de todos os triângulos, então ele está fora do polígono.

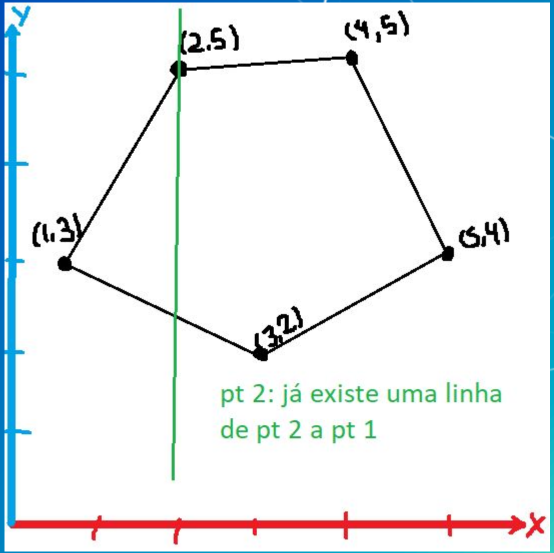


QUESTÃO 01

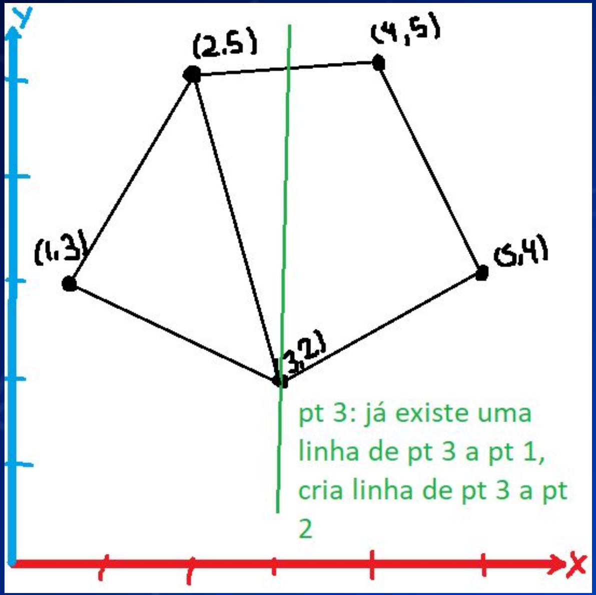
- Aplicando o algoritmo de varredura no polígono definido pelos pontos:



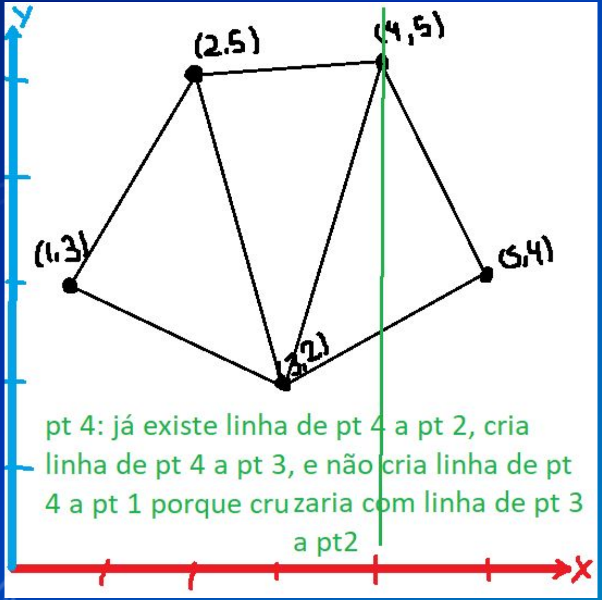




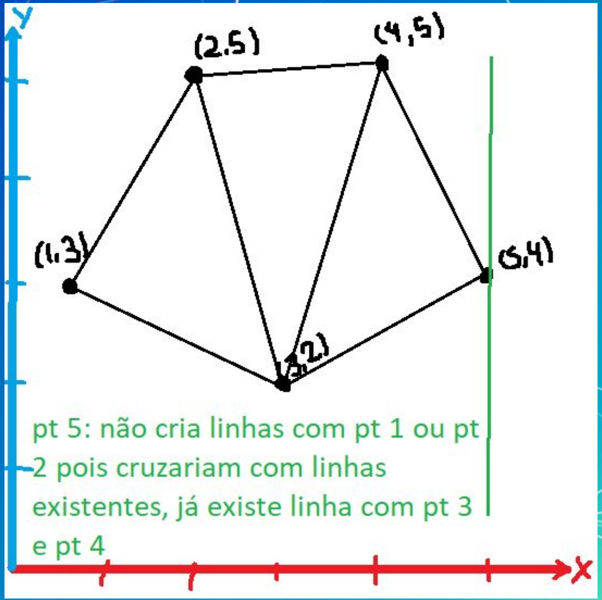
QUESTÃO 01



(2.5)
(4,5)
(1,3)
(5,4)
(3,2)
pt 3: já existe uma linha de pt 3 a pt 1, cria linha de pt 3 a pt 2



(2.5)
(4,5)
(1,3)
(5,4)
(3,2)
pt 4: já existe linha de pt 4 a pt 2, cria linha de pt 4 a pt 3, e não cria linha de pt 4 a pt 1 porque cruzaria com linha de pt 3 a pt2



(2.5)
(4,5)
(1,3)
(5,4)
(3,2)
pt 5: não cria linhas com pt 1 ou pt 2 pois cruzariam com linhas existentes, já existe linha com pt 3 e pt 4

QUESTÃO 01

- Aplicando a fórmula das coordenadas baricêntricas:

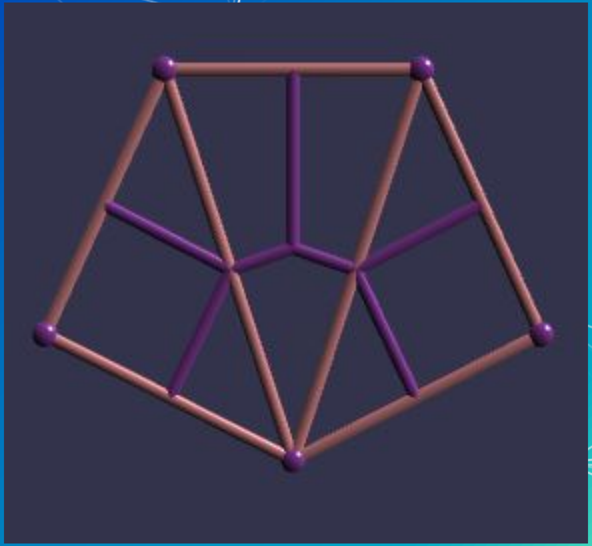
$$
\begin{aligned}
\lambda_1 &= \frac{(y_2 - y_3)(x - x_3) + (x_3 - x_2)(y - y_3)}{\det(T)} = \frac{(y_2 - y_3)(x - x_3) + (x_3 - x_2)(y - y_3)}{(y_2 - y_3)(x_1 - x_3) + (x_3 - x_2)(y_1 - y_3)}\,, \\
\lambda_2 &= \frac{(y_3 - y_1)(x - x_3) + (x_1 - x_3)(y - y_3)}{\det(T)} = \frac{(y_3 - y_1)(x - x_3) + (x_1 - x_3)(y - y_3)}{(y_2 - y_3)(x_1 - x_3) + (x_3 - x_2)(y_1 - y_3)}\,, \\
\lambda_3 &= 1 - \lambda_1 - \lambda_2\,.
\end{aligned}
$$

- T1: (0.4, 0.2, 0.4)
- T2: (0.6,-0.3,0.6)
- T3: (0.6,-0.8,1.2)
- Como as coordenadas do 1° triângulo estão entre 0 e 1, o ponto está dentro do 1° triângulo e dentro do polígono.

```
const baryCoords = function(point, triangle) {
    const x1 = triangle[0][0];
    const x2 = triangle[1][0];
    const x3 = triangle[2][0];

    const y1 = triangle[0][1];
    const y2 = triangle[1][1];
    const y3 = triangle[2][1];

    const x = point[0];
    const y = point[1];

    const detT = (y2-y3)*(x1-x3) + (x3-x2)*(y1-y3);

    const b1 = ((y2-y3)*(x-x3) + (x3-x2)*(y-y3)) / detT;
    const b2 = ((y3-y1)*(x-x3) + (x1-x3*(y-y3))) / detT;
    const b3 = 1 - b1 - b2;

    return [b1,b2,b3];
}
```

QUESTÃO 02

- Algoritmo de Voronoi implementado como o dual da triangulação de Delaunay:
  - Para cada triângulo de Delaunay:
    - Computar o circumcentro do triângulo
    - Ligar os circumcentros cujos triângulos tem arestas comuns
    - Ligar os circumcentros com a mediana das arestas do domínio



QUESTÃO 02

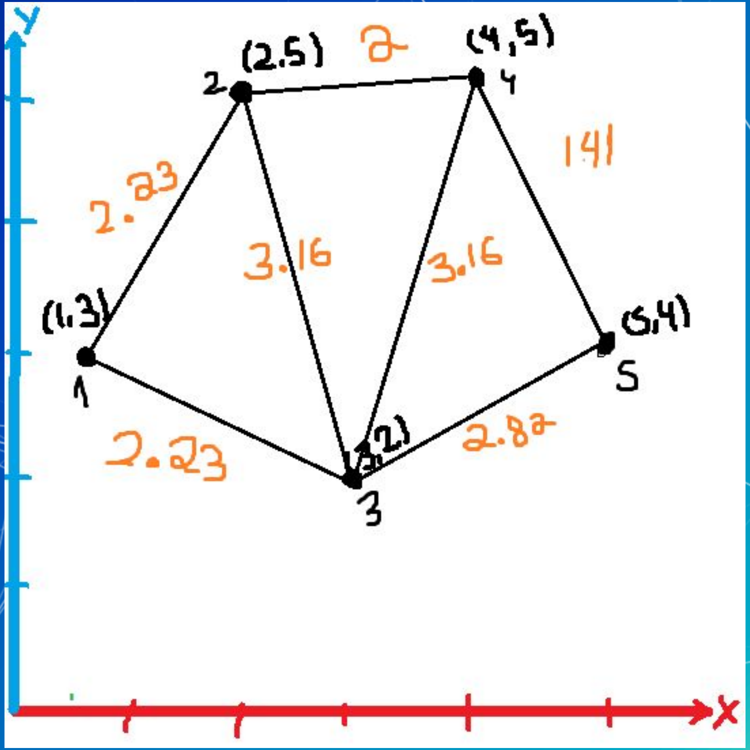
```javascript
const buildVoronoiFromDelaunay = function(triangles) {
  const circumcircles = [];
  const lines = [];

  //for each triangle compute the circumcircle
  for (let i = 0; i < triangles.length; i++) {
      circumcircles.push(getCircumcenter(triangles[i]));
  }

  // for each triangle, connect circumcircles of pairs of adjacent edges
  // and connect circumcircle with midpoint of domain line
  for (let i = 0; i < triangles.length; i++) {
      // for each edge
      for (let j = 0; j < 3; j++) {
          let edgej = [triangles[i][j%3], triangles[i][(j+1)%3]];
          let foundAdjEdge = false;
          for (let k = 0; k < triangles.length; k++) {
              if (i !== k) {
                  for (let l = 0; l < triangles.length; l++) {
                      let edgel = [triangles[k][l%3], triangles[k][(l+1)%3]];
                      if (equalLines(edgej, edgel)) {
                          lines.push([circumcircles[i], circumcircles[k]]);

                          foundAdjEdge = true;
                          break;
                      }
                  }
              }
          }
          if (!foundAdjEdge) {
              const midpoint = edgej[0].add(edgej[1]).scale(0.5);

              lines.push([circumcircles[i], midpoint]);
          }
      }
  }

  return lines;
```

QUESTÃO 03

- Para obter vizinhos-mais-próximos a partir de Delaunay, podemos observar que as arestas que ligam os pares de pontos na triangulação com certeza são as menores distâncias do ponto até seus vizinhos, então precisamos apenas olhar aquelas arestas para encontrar o vizinho mais próximo de todos.

QUESTÃO 03

- Seguindo a triangulação computada anteriormente, computamos os vizinhos mais próximos como:
  - p1: p2 e p3
  - p2: p4
  - p3: p1
  - p4: p5
  - p5: p4



QUESTÃO 04

- O algoritmo de Delaunay pode ser obtido através do método de avanço da fronteira utilizando o critério de Delaunay (maior ângulo). Dependendo da estrutura de dados usada para o teste de pontos, esse algoritmo pode se tornar O(n^2) ou O(nlogn)

QUESTÃO 05

- Algoritmo de varredura:
  - Para cada ponto, ordenados em coordenada x:
    - Testa todos os pontos anteriores, se a linha entre eles não cruza nenhuma linha existente, então criamos a linha.

```javascript
const buildSweepTriangulation = function(points) {
    const lines = [];

    const INTERSECTION_TOLERANCE = 1e-6;

    let testray = BABYLON.Ray.CreateNewFromTo(
        points[0], points[3]
    );
    const res = testray.intersectionSegment(points[1], points[2]);

    // go over the points, constructing the
    // triangles
    for (let i = 0; i < points.length; i ++) {
        const currP = points[i];
        // for each previous point
        for (let j = 0; j < i; j ++) {
            const prevP = points[j];
            // get ray going from previous to
            // current point
            const prevToCurr = BABYLON.Ray.CreateNewFromTo(prevP, currP);
            // check visibility
            let intersectsAny = false;
            for (let line of lines) {
                const intersectionDistance = prevToCurr.intersectionSegment(line[0], line[1], INTERSECTION_TOLERANCE);
                if (intersectionDistance !== -1 && intersectionDistance < prevToCurr.length - INTERSECTION_TOLERANCE) {
                    intersectsAny = true;
                    break;
                }
            }

            // if the line from prev to curr
            // point doesn't intersect any
            // previous line, that line can
            // be added
            if (!intersectsAny) {
                lines.push([
                    prevP,
                    currP
                ]);
            }
        }
    }

    const sweepGroup = buildTriangulationVisual(points, lines, "sweep", "#e1eb31", new BABYLON.Vector3(0,-5.5,0));

    return sweepGroup;
```

QUESTÃO 05

- Algoritmo por diagonais:
  - Para cada polígono de entrada:
    - Cria a diagonal de p1 a pn e testa se essa diagonal está dentro do domínio
      - Se estiver, divide o domínio em dois polígonos nessa diagonal e continua recursivamente
      - Senão, testa com os pontos anteriores até conseguir uma diagonal

```
const buildDiagonalTriangulation = function(domain) {
    if (domain.length > 3) {
        const points = getDomainPoints(domain);
        const pointToCheck = points[0];

        // check if diagonal p_i+1,p_n is inside the domain
        for (let n = points.length - 1; n > 1; n--) {
            const diagonal =
                BABYLON.Ray.CreateNewFromTo(points[1], points[n]);
            if (rayInsideDomain(diagonal, domain)) {
                const [domainHalfA, domainHalfB] =
                    cutDomainWithLine([points[1], points[n]], domain);

                const lineDiagonal = [points[1], points[n]]
                const triA =
                    buildDiagonalTriangulation(
                      [...domainHalfA, [points[n], points[1]]]);
                const triB =
                    buildDiagonalTriangulation(
                      [[points[1], points[n]], ...domainHalfB]);

                return [
                    ...triA,
                    ...triB,
                ];
            }
        }
    } else {
        return domain;
    }
}
```

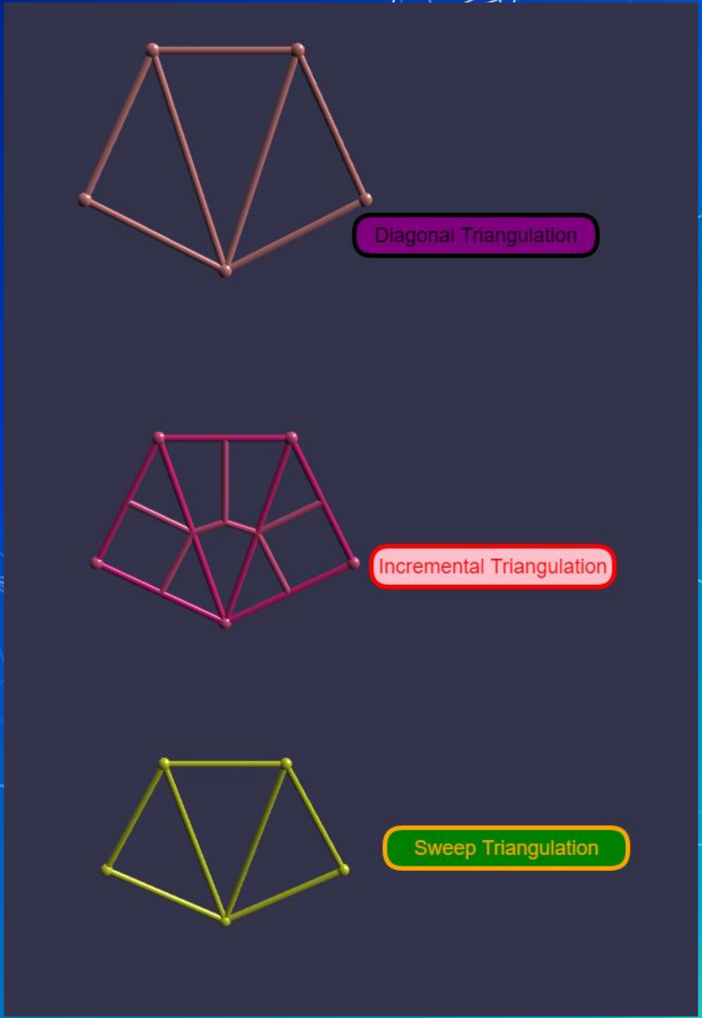
QUESTÃO 05:

- Algoritmo de avanço da fronteira:
    - Fronteira inicial é o fecho convexo
    - Para cada aresta da fronteira, classifica os outros pontos pelo maior ângulo (critério de Delaunay)
    - Para cada ponto classificado, testa se é possível criar arestas até o ponto
        - Se sim, criar arestas e atualizar a fronteira
        - Se não, continua buscando

```javascript
const buildAdvancingFrontierTriangulation = function(points, domain) {
    const frontier = [ ...domain];
    const lines = [ ...domain];

    while (frontier.length > 0) {
        const currFrontierLine = frontier.shift();

        // rank all other points that aren't endpoints of this edge with a criterion
        // (in this case, highest angle so that it is a delaunay triangulation)
        const angleWithLine = function(point, line) {
            const a = line[0].subtract(point).normalize();
            const b = line[1].subtract(point).normalize();

            const angle = Math.acos(BABYLON.Vector3.Dot(a, b));
            return angle;
        };

        const pointsToClassify = points.filter((p) => p !== currFrontierLine[0] && p !== currFrontierLine[1]);
        pointsToClassify.sort((p1, p2) => angleWithLine(p2, currFrontierLine) - angleWithLine(p1, currFrontierLine));

        for (let point of pointsToClassify) {
            // check if the two rays coming from the line endpoints to the current point don't intersect with any
existing lines
            const raya = BABYLON.Ray.CreateNewFromTo(currFrontierLine[0], point);
            const rayb = BABYLON.Ray.CreateNewFromTo(currFrontierLine[1], point);
            let intersectsAny = false;
            for (let line of lines) {
                if (rayLineIntersect(raya, line) || rayLineIntersect(rayb, line)) {
                    intersectsAny = true;
                    break;
                }
            }
            // if it doesn't, we can update the frontier with this new triangle
            if (!intersectsAny) {
                const linea = [currFrontierLine[0], point];
                const lineb = [currFrontierLine[1], point];

                // if one of the lines already exists on the frontier, we can remove it;
                // otherwise, we add it to the frontier
                let lineaFrontier = null;
                let linebFrontier = null;
                for (let i = 0; i < frontier.length; i++) {
                    const frontierLine = frontier[i];
                    if (equalLines(linea, frontierLine)) {
                        lineaFrontier = i;
                    } else if (equalLines(lineb, frontierLine)) {
                        linebFrontier = i;
                    }
                }

                if (lineaFrontier !== null) {
                    frontier.splice(lineaFrontier, 1);
                } else {
                    frontier.push(linea);
                    lines.push(linea);
                }

                if (linebFrontier !== null) {
                    frontier.splice(linebFrontier, 1);
                } else {
                    frontier.push(lineb);
                    lines.push(lineb);
                }

                break;
            }
        }

    }
    // when frontier is all depleted, we have our triangulation
    return buildTriangulationVisual(points, lines, "advGroup", "#ff2c92", new BABYLON.Vector3(0,0,0));
}
```

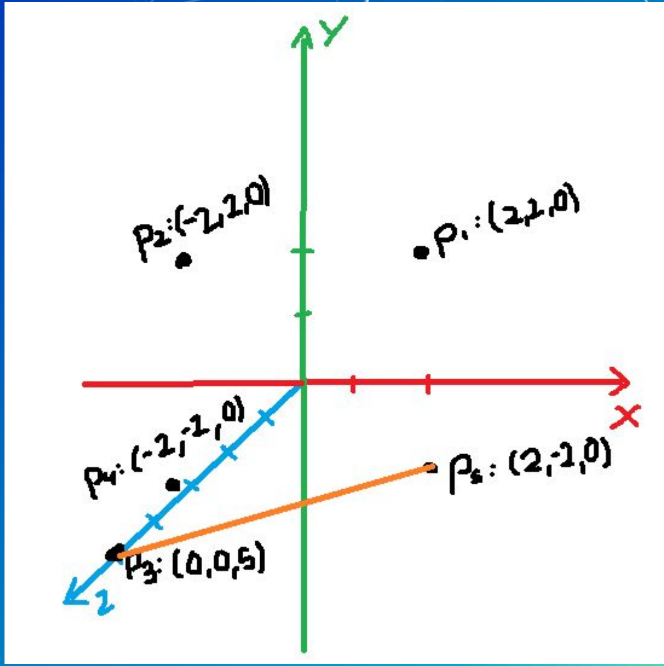
QUESTÃO 05

- Triangulações e Voronois visualizados:



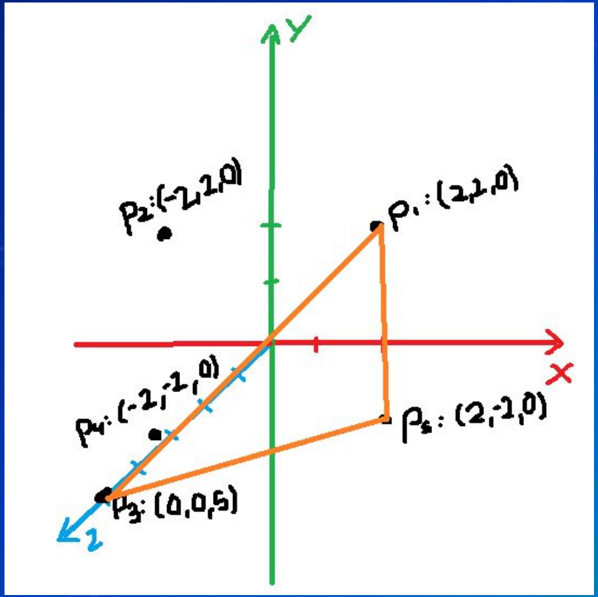
QUESTÃO 06

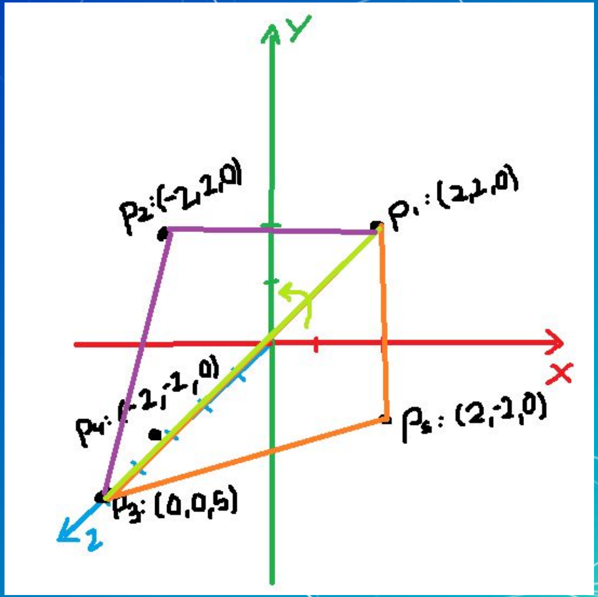
- A face convexa do fecho inicial pode ser determinada através da maior distância em um eixo, que no caso dos pontos dados, é a distância de 5 entre p3 e todos os outros pontos no eixo z. Por isso, escolhemos um ponto qualquer, por exemplo, p5 e formamos uma linha entre eles



QUESTÃO 6

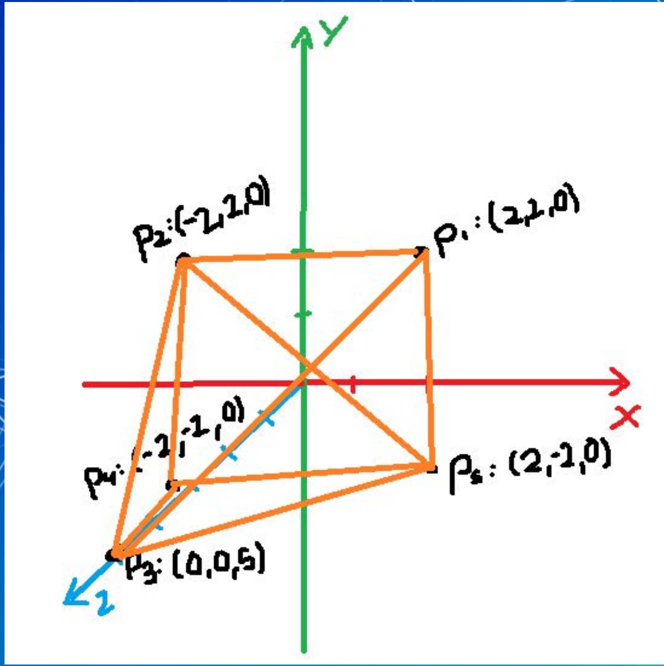
- O ponto seguinte é determinado como p1. Rotacionamos em torno da aresta p1p3 até obter o ponto p2:





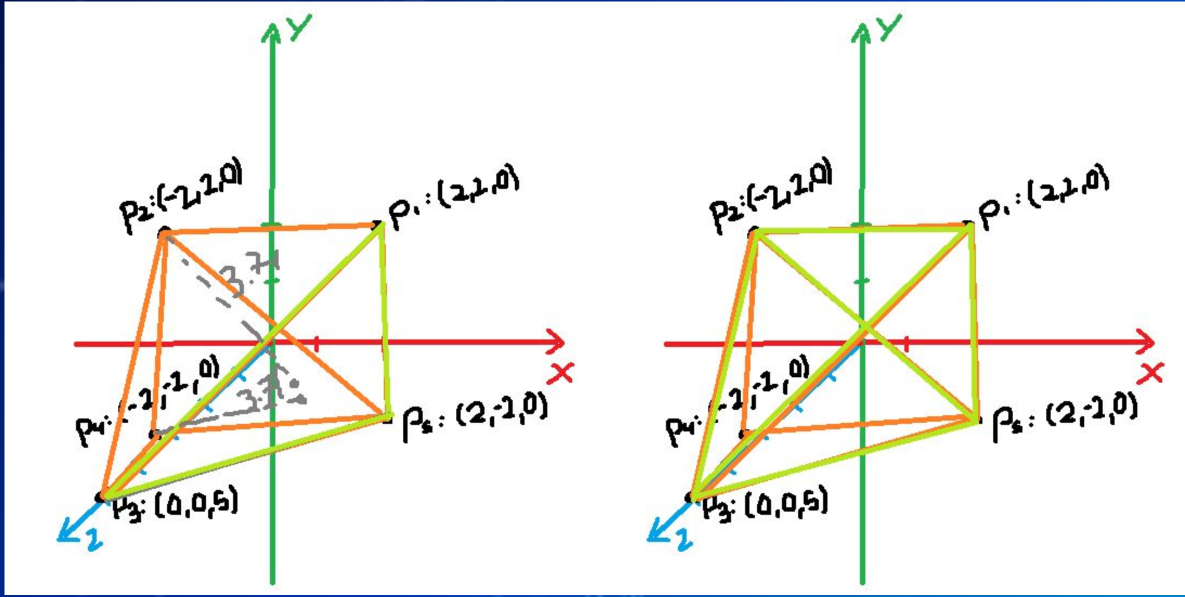
QUESTÃO 6

- Continuamos o processo até encontrar todos os triângulos:



QUESTÃO 06

- Escolhemos o plano p1,p3,p5 para começar. A distância dos pontos p2 e p4 a esse plano é a mesma, 3.71, então podemos escolher qualquer um dos dois. No caso, escolhemos p2:

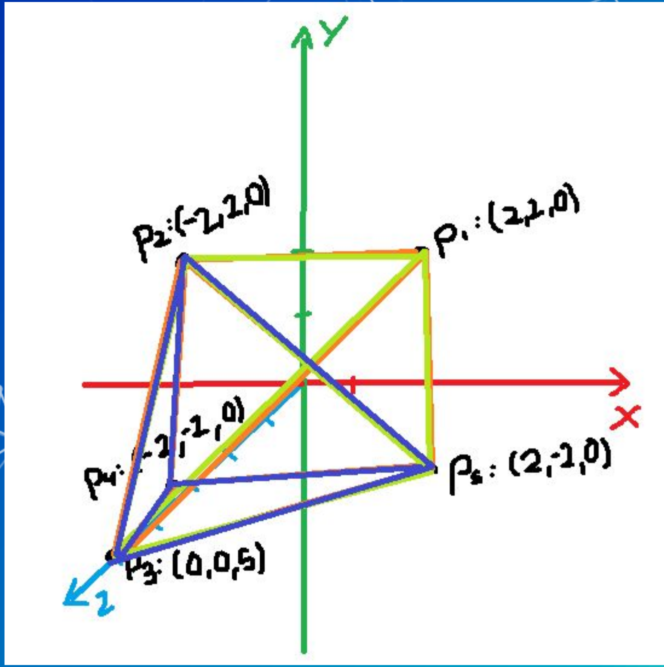


$$D_i = \hat{\mathbf{n}} \cdot (\mathbf{x}_0 - \mathbf{x}_i),$$

```
const pointPlaneDistance = function(p, [a, b, c]) {
  const normal = BABYLON.Vector3.Cross(
      b.subtract(a),
      c.subtract(a)).normalize();

  const dist = BABYLON.Vector3.Dot(
      normal,
      p.subtract(a));

  return dist;
}
```

QUESTÃO 06

- Continuamos o processo com a face p2,p3,p5, ambos p1 e p4 tem a mesma distância, 2.82. Tentamos utilizar o ponto p1, porém este levaria a uma colisão com o tetraedro já existente, logo usamos o ponto p4:



QUESTÃO 06

- A figura geométrica obtida pela tetraedralização é uma pirâmide quadrada formada por 2 tetraedros. Os dois tetraedros tem o mesmo volume, o qual é 13.33, logo o volume total é 26.66 unidades.

$$V = \frac{|(\mathbf{a}-\mathbf{d})\cdot((\mathbf{b}-\mathbf{d})\times(\mathbf{c}-\mathbf{d}))|}{6}.$$

```
const tetrVolume = function([a,b,c,d]) {
    const ad = a.subtract(d);
    const bd = b.subtract(d);
    const cd = c.subtract(d);
    const cross = BABYLON.Vector3.Cross(bd, cd);
    const dot = BABYLON.Vector3.Dot(ad, cross);
    return Math.abs(dot) / 6;
}
```

THANKS!

Any questions?

You can find me at

- Github: carolhmj
- carolhmj@gmail.com

CREDITS

Special thanks to all the people who made and released these awesome resources for free:

- Presentation template by SlidesCarnival