Федеральное государственное бюджетное образовательное учреждение высшего профессионального образования
Московский государственный технический университет имени Н.Э. Баумана

Домашняя работа №1
«Построение статической модели на основе статических данных»
по курсу:
«Моделирование»

Выполнил:
студент группы ИУ9-82
Иванов Георгий

Проверила:
Домрачева А.Б.

Москва, 2019

Содержание

1 Постановка задачи

Дана нерегулярная сетка, сформированная 10-ю точками, иллюстрирующая показатель прироста населения (в процентах) в десяти географически близких городских округах.

x	y	z
23.75	0.37	16
19.12	3.12	11
19.19	4.52	84
17.31	5.15	27
17.59	6.73	13
15.25	2.39	29
16.75	2.36	49
17.98	1.84	53
17.05	1.97	42
10.03	1.82	11

Построить триангуляционную сеть. Провести сгущение сети (не менее 1000 точек).

2 Теоретические сведения

Имитационная модель — модель, имитирующая поведение реального объекта или системы при заданных входных данных и с использованием вычислительной техники.

Имитационное моделирование — численный метод проведения на вычислительной технике экспериментов с математическими моделями, описывающими объект или систему в течении заданного или формируемого периода времени.

Имитационное моделирование предполагает идентификацию алгоритма преобразующего входные данные в выходные без учета физико-химических воздействий, протекающих в исследуемом объекте или системе. Различают статические (неизменные во времени) объекты и системы и динамические (изменяющиеся во времени) объекты и системы реального мира. Статическую модель можно рассматривать как частный случай динамической модели.

Нерегулярные триангуляционные сети являются способом цифрового отображения структуры поверхности, а также формой векторных цифровых географических данных, которые строятся методом триангуляции набора вершин (точек). Вершины соединяются серией ребер и формируют сеть треугольников.

Триангуляцией называется планарный граф, все внутренние области которого являются треугольниками. [1]

Выпуклой триангуляцией называется такая триангуляция, для которой минимальный многоугольник, охватывающий все треугольники, будет выпуклым. Триангуляция, не являющаяся выпуклой, называется **невыпуклой**. [1]

Задачей построения триангуляции по заданному набору двумерных точек называется задача соединения заданных точек непересекающимися отрезками так, чтобы образовалась триангуляция.

Говорят, что триангуляция удовлетворяет **условию Делоне**, если внутрь окружности, описанной вокруг любого построенного треугольника, не попадает ни одна из заданных точек триангуляции. [1]

Триангуляция называется **триангуляцией Делоне**, если она является выпуклой и удовлетворяет условию Делоне. (Рис. 1) [1]

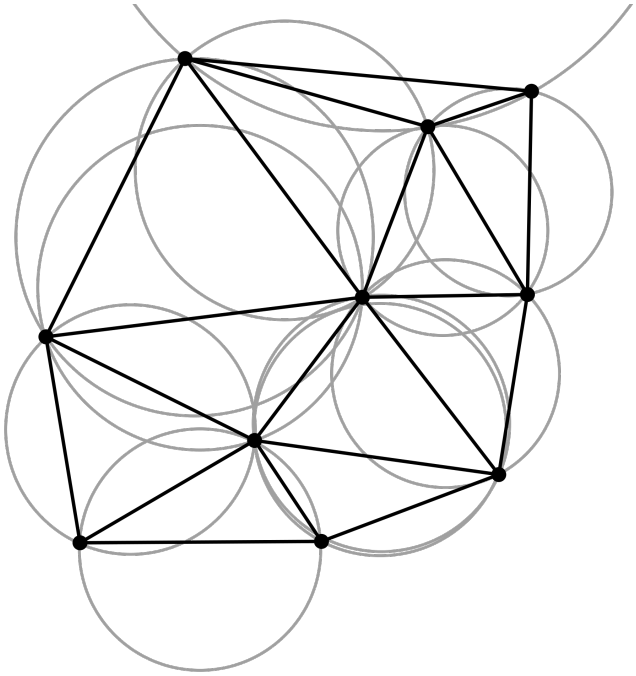

Рис. 1: Триангуляция Делоне

2.1 Итеративные алгоритмы построения триангуляции Делоне

Все итеративные алгоритмы триангуляции Делоне основываются на последовательном добавлении точек в частично построенную триангуляцию. Пусть имеется триангуляция Делоне из n-1 точки, тогда при добавлении очередной n-й точки надо выполнить следующие шаги:

1. Локализовать точку, то есть найти построенный ранее треугольник, в который попадает точка. Если точка попадает не внутрь триангуляции, то найти ближайший к ней треугольник.
2. Делаем один из следующих шагов в зависимости от положения точки.
   (a) Если точка попала на ранее вставленную (ϵ-окрестность), то она, как правило, отбрасывается.
   (b) Если точка попала на некоторое ребро, то оно разбивается на два новых. Оба смежных с ребром треугольника также делятся на два меньших.
   (c) Если точка попала строго внутрь какого-нибудь треугольника, то он делится на три новых.
   (d) Если точка попала вне триангуляции, то строится один или более новых.
3. После добавления новой точки условие Делоне может быть нарушено, поэтому надо проверить все вновь построенные треугольники и соседние с ними, а также выполнить после этого необходимые перестроения.

Сложность итеративного алгоритма складывается из сложности поиска треугольника, в который на очередном шаге добавляется точка, сложности построения новых треугольников, а также сложности соответствующих перестроений структуры триангуляции в результате неудо-

влетворительных проверок пар соседних треугольников полученной триангуляции на выполнение условия Делоне.

Чтобы несколько упростить алгоритм (точка попала вне триангуляции), можно предварительно внести в триангуляцию несколько таких дополнительных узлов, что построенная на них триангуляция заведомо накроет все исходные точки триангуляции. Такая структура обычно называется **суперструктурой**. В качестве можно выбрать квадрат, треугольник и n-угольник.

2.2 Итеративный алгоритм со статическим кэшированием поиска

Алгоритмы триангуляции с кэшированием поиска несколько похожи на алгоритмы триангуляции с индексированием центров треугольников. При этом строится кэш — специальная структура, позволяющая за время $O(1)$ находить некоторый треугольник, близкий к искомому. В отличие от алгоритмов триангуляции с индексированием, изменённые треугольники из кэша не удаляются, один и тот же треугольник может многократно находиться в кэше, а некоторые треугольники вообще там отсутствовать. Кэш в виде регулярной сети квадратов наиболее хорошо работает для равномерного распределения исходных точек и распределений, не имеющих высоких пиков в функции плотности.

В алгоритме триангуляции со статическим кэшированием поиска необходимо выбрать число m и завести кэш в виде 2-мерного массива r размером m×m ссылок на треугольники Первоначально этот массив надо заполнить ссылкой на суперструктуры. Затем после выполнения очередного поиска, в котором был найден некоторый треугольник T, необходимо обновить информацию в кэше: $r := ref\_T$. Размер статического кэша следует выбирать по формуле $m = s * N^{\frac{3}{8}}$, где s — коэффициент статического кэша. На практике значение s следует брать $\approx 0,6 - 0,9$

2.3 Построение статической модели

Методика построения статистической модели является частным случаем методики динамической модели в предположении, что состояние исследуемого объекта (или системы) остается неизменным.

1. Определяется вид представления данных: *табличный, графический, аналитический.*
2. Необходимо обозначить исследуемые и независимые показатели. Соответственно, это будут координаты (X, Y, Z).
3. Необходимо провести оптимизацию представления данных по выбраному алгоритму.
4. Далее формулизуется общее уравнение и формулируются одна и более вычислительных задач, конкретизируется формальное описание решений этих задач с целью выбора численного метода их решения. Повторно оценивается возможность упрощения модели, если это возможно.

Общие уравнения и сама вычислительная задача заложены в используемом алгоритме триангуляции. Алгоритм не может выявить, какой именно плоскости принадлежит данная точка, а следовательно триангуляция становится "рваной"и "нецелостной".

2.4 Алгоритм генерации высот. Шум Перлина.

Для реализации горного ландшафта необходимо проработать алгоритм генерации высот.

Самым простым способом для реализации алгоритма является использование встроенной функции для генерации пвсевдослучайной равномерно распределенной величины для задания высоты в каждой точке. Но это дает совершенно неприемлемый результат, так как точки находят хаотично в пространстве и выглядит это нереалистично.

Другой возможный способ — использование этой же самой функции с различными весами. Например, если точка находится ближе к центру сцены, то вес — меньше, преобладание Random меньше; чем дальше от центра — тем вес больше, случайные значения пребладают. Но данный способ также не оптимален.

Тогда используем «шум», как способ реализации алгоритм генерации высот. В бытовом смысле, «шум» — это случайный мусор.

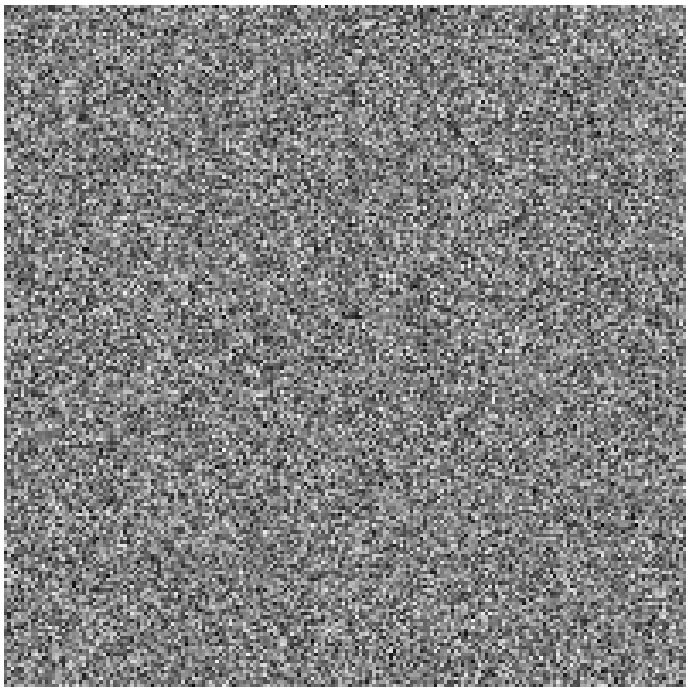

Рис. 2: Пиксельный шум

На рис. 2 представлен белый шум. Грубо говоря, это означает, что все пиксели случайны и независимы друг от друга. Усредненное значение цвета для этих пикселей должно быть #808080 (серый цвет). Для этого рисунка оно равно #848484, что довольно близко. Шум полезен для генерации случайных шаблонов, особенно для непредсказуемых природных явлений. Изображение выше может быть хорошей начальной точкой для создания, например, текстуры гравия.

Однако большинство вещей не чисто случайны. Дым, облака, ландшафт могут иметь некий элемент случайности, но они были созданы в результате очень сложных взаимодействий множества крохотных частиц. Белый шум содержит независимые частицы (или пиксели). Для генерации чего-то более интересного, чем гравий, нам нужен другой вид шума.

Шум Перлина — это градиентный шум, состоящий из набора псевдослучайных единичных векторов (направлений градиента), расположенных в определенных точках пространства и интер-

полированных функцией сглаживания между этими точками. Для генерации шума Перлина в одномерном пространстве необходимо для каждой точки этого пространства вычислить значение шумовой функции, используя направление градиента (или наклон) в указанной точке.

Функция «Perlin noise» проводит генерацию текстур методом генерации псевдослучайных чисел, однако все визуальные детали текстуры имеют одинаковый размер. Это свойство делает шум Перлина легко управляемым; множество масштабированных копий шума Перлина могут быть вставленными в математические выражения для создания самых разнообразных процедурных текстур.

Этот способ (шум Перлина) дает более реалистичные результаты.

3 Практическая реализация

Для решения поставленной задачи система точек триангулируется, затем для каждого треугольника добавляется новая точка, являющаяся его центром масс. Полученная таким образом система точек снова триангулируется и в набор добавляются центры масс новых треугольников. Процесс продолжается итеративно до тех пор, пока не будет получено необходимое количество точек (10000 точек).

Ниже приведена реализация некоторых методов программы на языке JavaScript, визуализация выполненяется с помощью библиотеки THREE.JS.

Листинг 1 Создание суперструктуры (квадрата)

```
function createSuperSquare(left, right, yValue, cache) {

    var bottomLeft = new Point(left, yValue, left);
    var bottomRight = new Point(left, yValue, right);
    var topRight = new Point(right, yValue,right);
    var topLeft = new Point(right, yValue, left);

    var leftTr = new Triangle([topLeft,bottomLeft,bottomRight]);
    var rightTr = new Triangle([topLeft,topRight,bottomRight]);

    var diagonal = new Edge(topLeft, bottomRight, leftTr, rightTr);
    var leftEdge = new Edge(topLeft, bottomLeft, leftTr, null);
    var rightEdge = new Edge(topRight, bottomRight, rightTr, null);
    var topEdge = new Edge(topLeft, topRight, rightTr, null);
    var bottomEdge = new Edge(bottomLeft, bottomRight, leftTr, null);

    leftTr.edges = [leftEdge, bottomEdge, diagonal];
    rightTr.edges = [topEdge, rightEdge, diagonal];

    cache.updateSuper(leftTr,true);
    cache.updateSuper(rightTr,false);

    return [leftTr, rightTr];
}
```

Листинг 2 Вставка новой точки в триангуляцию

```
function calcTriangulationTr(point,cache,resultTriangles) {
    var initTriangle = cache.get(point);
    var targetTriangle = findTriangleBySeparatingEdges(point, initTriangle);

    for (i = 0; i < targetTriangle.points.length; i++) {
        var point1 = targetTriangle.points[i];
```

```
            if (point.isInEpsilonAreaPoint(point1)) {
                return null;
            }
        }

        var modifiedTriangles = [];
        var newTriangles = [];

        var targetTriangleEdges = targetTriangle.edges;
        for (var i = 0; i < targetTriangleEdges.length; i++) {
            if (isInEpsilonArea(distanceToLine(targetTriangleEdges[i].a,
            ↪ targetTriangleEdges[i].b, point), 0)) {
                var edge = targetTriangleEdges[i];
                var triangles;
                var newEdgeTriangles = [];

                if (edge.triangle1 === null || edge.triangle2 === null) {
                    triangles = putPointOnOutsideEdge(edge, point, newEdgeTriangles);
                    newTriangles = triangles.new;
                    modifiedTriangles = triangles.modified;
                } else {
                    triangles = putPointOnInnerEdge(edge, point, newEdgeTriangles);
                    newTriangles = triangles.new;
                    modifiedTriangles = triangles.modified;
                }
                for (var i=0;i<newTriangles.length;i++) {
                    resultTriangles.push(newTriangles[i]);
                }

                modifiedTriangles = modifiedTriangles.concat(newTriangles);
            }
        }

        if (newTriangles.length === 0) {
            var newInnerTriangles = [];
            var triangles = putPointInTriangle(targetTriangle, point, newInnerTriangles);
            modifiedTriangles = triangles.modified;
            newTriangles = triangles.new;

            for (var i=0;i<newTriangles.length;i++) {
                resultTriangles.push(newTriangles[i]);
            }
        }

        for (i = 0; i < newTriangles.length; i++) {
            cache.update(newTriangles[i]);
```

```
    }

    return modifiedTriangles;
}
```

Листинг 3 Попадание точки на внутреннее ребро (внешнее ребро - ребро суперструктуры)

```
function putPointOnInnerEdge(edge, point, newTriangles) {
    var a = edge.a;
    var b = edge.b;

    var abc = edge.triangle1;
    var abd = edge.triangle2;

    var c = abc.getOppositeNode(edge);
    var d = abd.getOppositeNode(edge);

    var bc = abc.getOppositeEdge(a);
    var bd = abd.getOppositeEdge(a);

    var obd = new Triangle([point,b,d]);
    var obc = new Triangle([point,b,c]);

    var oc = new Edge(point, c, abc, obc);
    var od = new Edge(point, d, abd, obd);
    var ob = new Edge(point, b, obd, obc);

    if (bc.triangle1.isEqual(abc))
        bc.triangle1 = obc;
    else
        bc.triangle2 = obc;

    if (bd.triangle1.isEqual(abd))
        bd.triangle1 = obd;
    else
        bd.triangle2 = obd;

    obd.edges = [ob,bd,od];
    obc.edges = [ob,bc,oc];

    abc.edges[1] = oc;
    abc.points[1] = point;
    abd.edges[1] = od;
    abd.points[1] = point;

    edge.b = point;
```

```
40
41      obc.updateValue();
42      obd.updateValue();
43      abc.updateValue();
44      abd.updateValue();
45
46      newTriangles = [obd,obc];
47      var modifiedTriangles = [abc,abd];
48
49      return {
50          new: newTriangles,
51          modified: modifiedTriangles
52      }
53  }
```

Листинг 4 Реализация статического кэша

```
1   function StaticCache(size, min_x,max_x,min_z,max_z) {
2
3       this.min_x = min_x;
4       this.max_x = max_x;
5       this.x_size = (max_x - min_x) / size;
6
7       this.min_z = min_z;
8       this.max_z = max_z;
9       this.z_size = (max_z - min_z) / size;
10
11
12      this.cache = new Array(size);
13      for (var i = 0; i < size; i++) {
14          this.cache[i] = new Array(size);
15      }
16  }
17
18  StaticCache.prototype.getColumn = function(value) {
19      return Math.floor((value - this.min_x) / this.x_size);
20  };
21
22  StaticCache.prototype.getRow = function(value) {
23      return Math.floor((value - this.min_z) / this.z_size);
24  };
25
26  StaticCache.prototype.update = function (T) {
27      var center = T.center();
28      this.cache[this.getColumn(center.x)][this.getRow(center.z)] = T;
29  };
```

```
30
31  StaticCache.prototype.updateSuper = function (T, isLeft) {
32      if (isLeft) {
33          for (i = 0; i < this.cache.length; i++)
34              for (j = 0; j <= i; j++)
35                  if (i === j && Math.random() % 2 === 1)
36                      this.cache[i][j] = T;
37                  else if (i !== j)
38                      this.cache[i][j] = T;
39      }
40      else {
41          for (i = 0; i < this.cache.length; i++)
42              for (j = i; j < this.cache[i].length; j++)
43                  if (i === j && this.cache[i][j] === undefined)
44                      this.cache[i][j] = T;
45                  else
46                      this.cache[i][j] = T;
47      }
48  };
49
50  StaticCache.prototype.get = function(node) {
51      return this.cache[this.getColumn(node.x)][this.getRow(node.z)];
52  };
```

Листинг 5 Реализация шума Перлина

```
1   var gradP = new Array(512);
2
3   function fade(t) {
4       return t*t*t*(t*(t*6-15)+10);
5   }
6
7   function lerp(a, b, t) {
8       return (1-t)*a + t*b;
9   }
10
11  module.perlin2 = function(x, y) {
12      var X = Math.floor(x), Y = Math.floor(y);
13      x = x - X; y = y - Y;
14
15      X = X & 255; Y = Y & 255;
16
17      var n00 = gradP[X+perm[Y]].dot2(x, y);
18      var n01 = gradP[X+perm[Y+1]].dot2(x, y-1);
19      var n10 = gradP[X+1+perm[Y]].dot2(x-1, y);
20      var n11 = gradP[X+1+perm[Y+1]].dot2(x-1, y-1);
```

```
21
22      var u = fade(x);
23      return lerp(lerp(n00, n10, u),lerp(n01, n11, u),fade(y));
24  };
```

4 Результаты

Примеры результаты работы программы приведены на рисунках для разных количества точек (рис. 2-6).

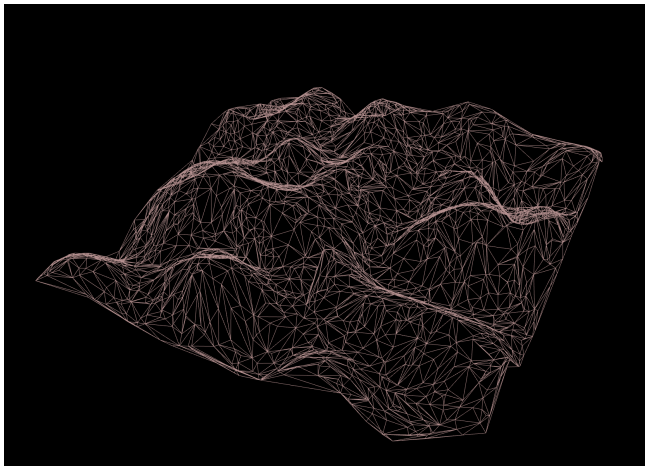

Рис. 3: 2500 точек

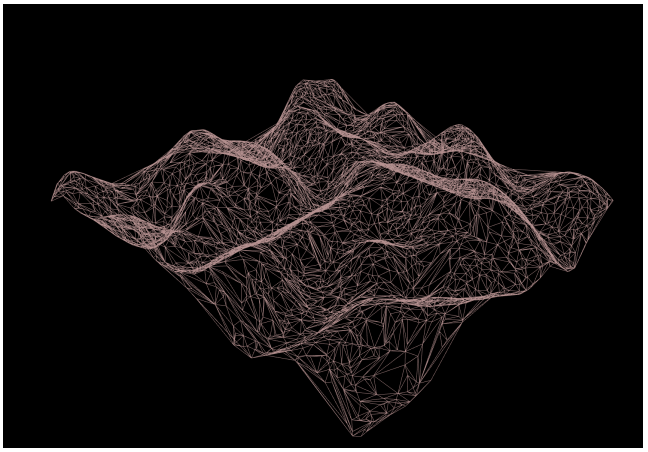

Рис. 4: 5000 точек

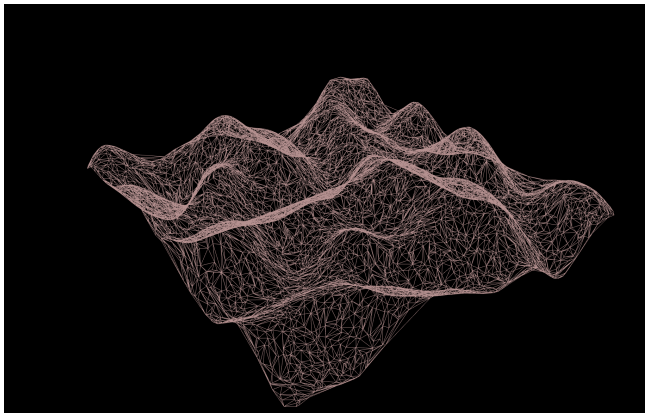

Рис. 5: 10000 точек

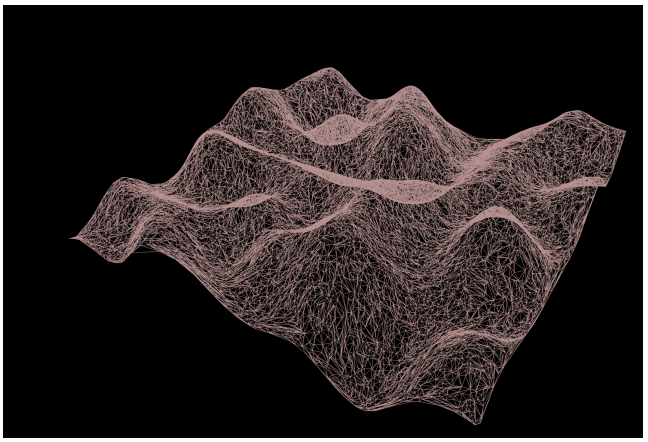

Рис. 6: 25000 точек

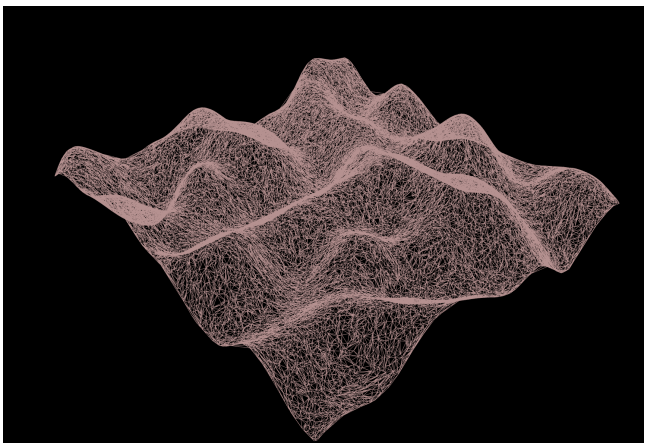

Рис. 7: 50000 точек

5 Вывод

В ходе выполнения домашнего задания были изучены алгоритмы построения триангуляции Делоне и реализован итеративный алгоритм с статическим кэшированием поиска, а также реализован шум Перлина. Итеративный алгоритм с статическим кэшированием поиска выигрывает по быстродействию почти у всех существующих (кроме динамическое кэширования поиска), так как имеют динамическую структуру — кэш, который служит для быстрой локализации точки. Данный факт был подтвержден для большого количества данных. Но проигрывает алгоритму с динамическим кэшированием поиска по причине того, что первое время, пока кэш не обновится полностью, поиск может идти довольно долго, но потом скорость повышается.

С помощью триангуляции можно строить модель статистических данных. Набор исследуемых значений может быть интерполирован с помощью триангуляционной сетки.

Список литературы

[1] Скворцов А. В. Триангуляция Делоне и ее применение. Изд-во Том. ун-та, 2002. 128 с.