УДК 519.688

ОБЗОР АЛГОРИТМОВ ПОСТРОЕНИЯ ТРИАНГУЛЯЦИИ ДЕЛОНЕ

А. В. Скворцов[1]

В работе рассматриваются многие известные алгоритмы построения триангуляции Делоне и предлагается их классификация. Для всех алгоритмов приводится оценка их трудоемкости в среднем и худшем случаях. Обсуждаются особенности реализации. Рассматриваются четыре структуры данных для представления триангуляции. Приводятся процедуры проверки условия Делоне и описываются процедуры слияния триангуляций.

Введение. Задача построения триангуляции Делоне является одной из базовых в вычислительной геометрии. К ней сводятся многие другие задачи, она широко используется в машинной графике и геоинформационных системах для моделирования поверхностей и решения пространственных задач.

Впервые задача построения триангуляции Делоне была поставлена в 1934 г. в работе советского математика Б. Н. Делоне [1]. Ее трудоемкость составляет $O(N \log N)$ арифметических операций. Существуют алгоритмы, достигающие этой оценки в среднем и худшем случаях. Кроме того, известны алгоритмы, позволяющие в ряде случаев достичь в среднем $O(N)$. Тем не менее, по настоящее время многие продолжают работать над усовершенствованием известных и созданием новых алгоритмов. Это обусловлено неустойчивостью ряда известных алгоритмов и неудовлетворительным временем их работы на реальных наборах данных.

В настоящей работе сделана попытка классификации и обобщения многих известных алгоритмов в соответствии с приведенной ниже схемой (жирным выделены конкретные алгоритмы).

1. Итеративные алгоритмы.
   - 1.1. **Простой итеративный алгоритм**.
     - 1.1.1. **Итеративный алгоритм “Удаляй и строй”**.
   - 1.2. Итеративные алгоритмы с индексированием поиска треугольников.
     - 1.2.1. **Итеративный алгоритм с индексированием R-деревом.**
     - 1.2.2. **Итеративный алгоритм с индексированием k-D-деревом.**
     - 1.2.3. **Итеративный алгоритм с индексированием квадродеревом.**
   - 1.3. Итеративные алгоритмы с кэшированием поиска треугольников.
     - 1.3.1. **Итеративный алгоритм со статическим кэшированием поиска**.
     - 1.3.2. **Итеративный алгоритм с динамическим кэшированием поиска**.
   - 1.4. Итеративные алгоритмы с измененным порядком добавления точек.
     - 1.4.1. **Итеративный полосовой алгоритм.**
     - 1.4.2. **Итеративный квадратный алгоритм.**
     - 1.4.3. **Итеративный алгоритм с послойным сгущением.**
     - 1.4.4. **Итеративный алгоритм с сортировкой вдоль фрактальной кривой**.
     - 1.4.5. **Итеративный алгоритм с сортировкой по Z-коду.**
2. Алгоритмы слияния.
   - 2.1. **Алгоритм “Разделяй и властвуй”.**
   - 2.2. **Рекурсивный алгоритм с разрезанием по диаметру.**
   - 2.3. Полосовые алгоритмы слияния.
     - 2.3.1. **Алгоритм выпуклого полосового слияния.**
     - 2.3.2. **Алгоритм невыпуклого полосового слияния.**
3. Алгоритмы прямого построения.
   - 3.1. **Пошаговый алгоритм.**

[1] Томский государственный университет, факультет информатики, пр. Ленина, д. 36, 634050, г. Томск; e-mail: skv@csd.tsu.ru

© Научно-исследовательский вычислительный центр МГУ им. М.В. Ломоносова

3.2. Пошаговые алгоритмы с ускорением поиска соседей Делоне.
3.2.1. **Пошаговый алгоритм с бинарным деревом поиска.**
3.2.2. **Клеточный пошаговый алгоритм.**

4. Двухпроходные алгоритмы.
4.1. **Двухпроходные алгоритмы слияния.**
4.2. **Модифицированный иерархический алгоритм.**
4.3. **Линейный алгоритм.**
4.4. **Веерный алгоритм.**
4.5. **Алгоритм рекурсивного расщепления.**
4.6. **Ленточный алгоритм.**

В работе рассматриваются четыре вида структур данных для представления триангуляции:

1. **“Узлы с соседями”.**
2. **“Двойные ребра”.**
3. **“Узлы, ребра и треугольники”.**
4. **“Узлы и треугольники”.**

Приводятся четыре процедуры проверки условия Делоне:

1. **Проверка через уравнение описанной окружности.**
2. **Проверка с заранее вычисленной описанной окружностью.**
3. **Проверка суммы противолежащих углов.**
4. **Модифицированная проверка суммы противолежащих углов.**

Также описываются пять процедур слияния двух триангуляций Делоне:

1. **“Удаляй-и-строй”**.
2. **“Строй-и-перестраивай”**.
3. **“Строй, перестраивая”**.
4. **“Слияние-и-перестроение”**.
5. **Слияние невыпуклых полосовых триангуляций**.

Статья организована следующим образом. В разделе 1 описываются общая постановка задачи, используемые базовые алгоритмы и структуры данных. В разделах 2 – 5 анализируются четыре группы алгоритмов построения триангуляции Делоне. Для каждого алгоритма приводятся оценки трудоемкости в худшем и среднем случаях. В заключении приводится сводная таблица всех рассматриваемых алгоритмов с указанием трудоемкости, скорости работы на реальных данных и общей экспертной оценки.

1. Определения и структуры данных. Для дальнейшего обсуждения введем несколько определений [1, 3, 7]:

Определение 1. *Триангуляцией* называется планарное разбиение плоскости на M фигур, из которых одна является внешней бесконечной, а остальные — треугольниками.

Определение 2. *Выпуклой триангуляцией* называется такая триангуляция, для которой минимальный полигон, охватывающий все треугольники, является выпуклым. Триангуляция, не являющаяся выпуклой, называется *невыпуклой.*

Определение 3. *Задачей построения триангуляции по заданному набору двумерных точек* называется задача соединения заданных точек непересекающимися отрезками таким образом, чтобы в полученной триангуляции между любыми двумя данными точками нельзя было построить новые отрезки без пересечения с уже существующими.

Задача построения триангуляции по исходному набору точек является неоднозначной, поэтому возникает вопрос, какая из двух различных триангуляций лучше. В литературе *оптимальной* называют такую триангуляцию, у которой сумма длин всех ребер минимальна. При этом показано [20 – 22], что задача построения такой триангуляции является NP-полной, т.е. имеет сложность $O\left(e^N\right)$. Поэтому для большинства практических задач существующие алгоритмы построения оптимальной триангуляции неприемлемы ввиду слишком высокой трудоемкости.

Широко известна триангуляция Делоне, которая не является оптимальной в упомянутом смысле [1, 3, 13, 20, 22], но обладает рядом других практически важных свойств.

Определение 4. Говорят, что триангуляция удовлетворяет *условию Делоне*, если внутрь окружности, описанной вокруг любого построенного треугольника, не попадает ни одна из заданных точек триангуляции (рис. 1). Такая триангуляция называется *триангуляцией Делоне.*

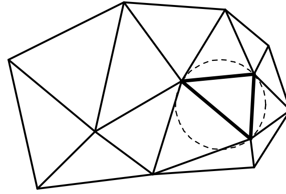

Рис. 1. Триангуляция Делоне

Определение 5. Говорят, что пара соседних треугольников триангуляции удовлетворяет *условию Делоне*, если этому условию удовлетворяет триангуляция, составленная только из этих двух треугольников.

Определение 6. Говорят, что треугольник триангуляции удовлетворяет *условию Делоне*, если этому условию удовлетворяет триангуляция, составленная только из этого треугольника и трех его соседей (если они существуют).

Многие алгоритмы построения триангуляции Делоне используют следующую теорему [3, 16].

Теорема 1. *Триангуляцию Делоне можно получить из любой другой триангуляции по той же системе точек, последовательно перестраивая пары соседних треугольников ABC и BCD, не удовлетворяющих условию Делоне, в пары треугольников ABD и ACD (рис. 2).*

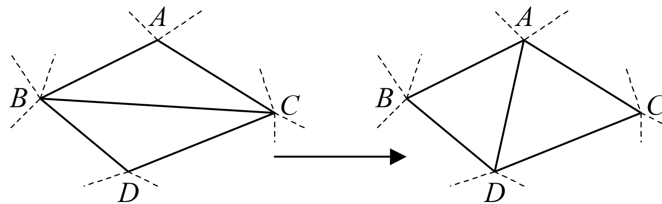

Рис. 2. Перестроение треугольников, не удовлетворяющих условию Делоне

Такую операцию перестроения часто называют *флипом*. Данная теорема позволяет строить триангуляцию Делоне последовательно, вначале строя некоторую триангуляцию, а потом последовательно улучшая ее в смысле условия Делоне. При проверке условия Делоне для пар соседних треугольников можно использовать непосредственно определение, но иногда используются другие способы, основанные на следующих теоремах [3, 15, 17, 26].

Теорема 2. *Триангуляция Делоне обладает максимальной суммой минимальных углов всех своих треугольников среди всех возможных триангуляций.*

Теорема 3. *Триангуляция Делоне обладает минимальной суммой радиусов окружностей, описанных около треугольников, среди всех возможных триангуляций.*

В данных теоремах фигурирует некая суммарная характеристика всей триангуляции (сумма минимальных углов или сумма радиусов), оптимизируя которую можно получить триангуляцию Делоне.

Триангуляция Делоне обладает рядом других важных свойств, выделяющих ее среди альтернативных классов триангуляций. В частности, можно выделить свойство двойственности триангуляции Делоне диаграмме (разбиению) Вороного [1, 6, 7].

1.1. Структуры для представления триангуляции. Как показывает практика, выбор структуры для представления триангуляции оказывает существенное влияние на трудоемкость алгоритмов, использующих данную структуру, а также на скорость конкретной реализации. Кроме того, структура триангуляции может зависеть от цели ее дальнейшего использования. В триангуляции можно выделить три основных вида объектов: *узлы* (точки, вершины), *ребра* (отрезки) и *треугольники*.

Во многих существующих алгоритмах построения триангуляции Делоне и алгоритмах анализа триангуляции часто используются следующие операции:

1. *Треугольник* → *узлы*: получение для данного треугольника трех образующих его узлов.
2. *Треугольник* → *ребра*: получение для данного треугольника списка образующих его ребер.
3. *Треугольник* → *треугольники*: получение для данного треугольника списка соседних с ним треугольников.
4. *Ребро* → *узлы*: получение для данного ребра координат образующих его узлов.
5. *Ребро* → *треугольники*: получение для данного ребра списка соседних с ним треугольников.
6. *Узел* → *ребра*: получение для данного узла списка смежных с ним ребер.

7. *Узел → треугольники*: получение для данного узла списка смежных с ним треугольников.
8. *Точка → объект триангуляции*: получение в заданной точке плоскости узла, ребра или треугольника.

В тех или иных алгоритмах некоторые из этих операций могут не использоваться. В некоторых алгоритмах операции с ребрами могут возникать не часто, поэтому ребра могут представляться косвенно как одна из сторон некоторого треугольника. Рассмотрим наиболее часто встречающиеся в литературе структуры.

1.1.1. Структура "Узлы с соседями". В *структуре "Узлы с соседями"* для каждого узла триангуляции хранятся его координаты на плоскости и список указателей на соседние узлы (список номеров узлов), с которыми есть общие ребра (рис. 3, 4) [23]. По сути, список соседей определяет в неявном виде ребра триангуляции. Треугольники же при этом не представляются вообще, что является обычно существенным препятствием для дальнейшего применения триангуляции. Кроме того, недостатком является переменный размер структуры узла, зачастую приводящий к неэкономному расходу оперативной памяти при построении триангуляции. Среднее число смежных узлов в триангуляции Делоне равно шесть (это доказывается по индукции или из теоремы Эйлера о планарных графах), поэтому при 8-байтовом представлении координат, 4-байтовых целых и 4-байтовых указателях суммарный объем памяти, занимаемый данной структурой, составляет $44 \cdot N$ байт.

```
Узел = record
  X: число; ← координата X
  Y: число; ← координата Y
  Count: целое; ← количество смежных узлов Делоне
  Nodes: array [1..Count] of Указатель_на_узел; ← список смежных узлов
end;
```

Рис. 3. Структура данных триангуляции "Узлы с соседями"

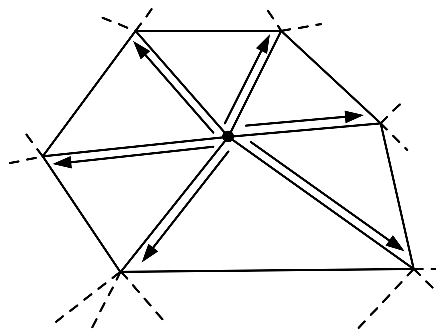

Рис. 4. Связи узлов структуры "Узлы с соседями"

1.1.2. Структура "Двойные ребра". В *структуре "Двойные ребра"* основой триангуляции является список ориентированных ребер. При этом каждое ребро входит в структуру триангуляции дважды, но направленными в противоположные стороны. Для каждого ребра хранятся следующие указатели (рис. 5, 6) [14]:

1) на концевой узел ребра;

2) на следующее по часовой стрелке ребро в треугольнике, находящемся справа от данного ребра;

3) на "ребро-близнец", соединяющее те же самые узлы триангуляции, что и данное, но направленное в противоположную сторону;

4) на треугольник, находящийся справа от ребра.

Последний указатель не нужен для построения триангуляции, и поэтому его наличие должно определяться в зависимости от цели дальнейшего применения триангуляции.

Недостатками данной структуры является представление треугольников в неявном виде, а также большой расход памяти, составляющий при 8-байтовом представлении координат и 4-байтовых указателях не менее $64 \cdot N$ байт (не учитывая расход памяти на представление дополнительных данных в треугольниках).

1.1.3. Структура "Узлы, ребра и треугольники". В *структуре "Узлы, ребра и треугольники"* в явном виде задаются все виды объектов триангуляции: узлы, ребра и треугольники. Для каждого ребра

Узел = **record**
  X: число; ← координата X
  Y: число; ← координата Y
end;
Ребро = **record**
  Node: Указатель_на_узел; ← концевой узел ребра
  Next: Указатель_на_ребро; ← следующее по часовой стрелке ребро в треугольнике справа
  Twin: Указатель_на_ребро; ← такое же ребро (близнец), но направленное в другую сторону
  Triangle: Указатель_на_треугольник; ← указатель на треугольник справа
end;
Треугольник = **record**
end;

Рис. 5. Структура данных триангуляции “Двойные ребра”

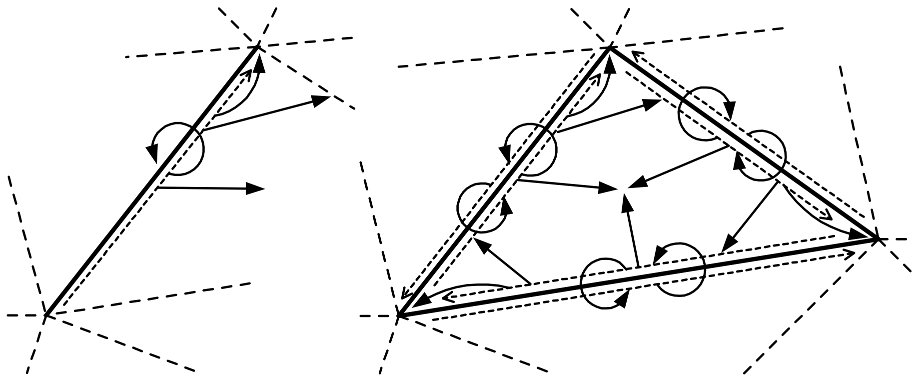

Рис. 6. Связи ребер (слева) и неявное задание треугольников (справа) в структуре “Двойные ребра”

хранятся указатели на два концевых узла и два соседних треугольника. Для треугольников хранятся указатели на три образующих треугольник ребра (рис. 7, 8) [11].

Недостатком данной структуры является большой расход памяти, составляющий при 8-байтовом представлении координат и 4-байтовых указателях примерно $88 \cdot N$ байт.

Узел = **record**
  X: число; ← координата X
  Y: число; ← координата Y
end;
Ребро = **record**
  Nodes: **array** [1..2] **of** Указатель_на_узел; ← список концевых узлов
  Triangles: **array** [1..2] **of** Указатель_на_треугольник; ← список соседних треугольников
end;
Треугольник = **record**
  Ribs: **array** [1..3] **of** Указатель_на_ребро; ← список образующих ребер
end;

Рис. 7. Структура данных триангуляции “Узлы, ребра и треугольники”

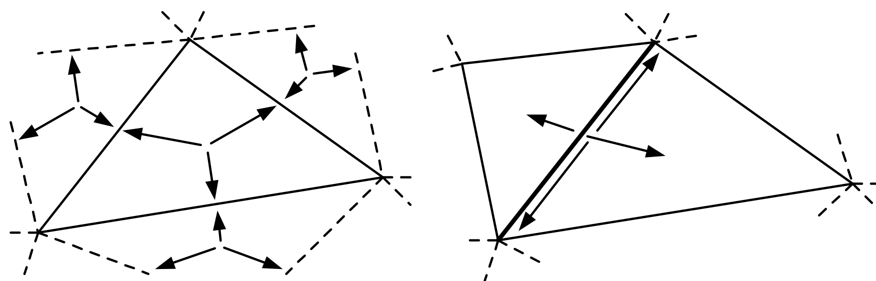

Рис. 8. Связи треугольников (слева) и ребер (справа) структуры “Узлы, ребра и треугольники”

1.1.4. Структура “Узлы и треугольники”. В *структуре “Узлы и треугольники”* для каждого треугольника хранятся три указателя на образующие его узлы и три указателя на смежные треугольники (рис. 9, 10) [9, 18, 25]. Нумерация точек и соседних треугольников производится в порядке обхода по часовой стрелке, при этом напротив точки с номером $i \in \{1, 2, 3\}$ располагается ребро, соответствующее соседнему треугольнику с таким же номером. Ребра в этой триангуляции в явном виде не хранятся. При необходимости же они обычно представляются как указатель на треугольник и номер ребра внутри него. При 8-байтовом представлении координат и 4-байтовых указателях требуется примерно $40 \cdot N$ байт. Данная структура триангуляции наиболее часто применяется на практике в силу своей компактности и относительной удобности в работе.

```
Узел = record
   X: число; ← координата X
   Y: число; ← координата Y
end;
Треугольник = record
   Nodes: array [1..3] of Указатель_на_узел; ← список образующих узлов
   Triangles: array [1..3] of Указатель_на_треугольник; ← список смежных треугольников
end;
```

Рис. 9. Структура данных триангуляции “Узлы и треугольники”

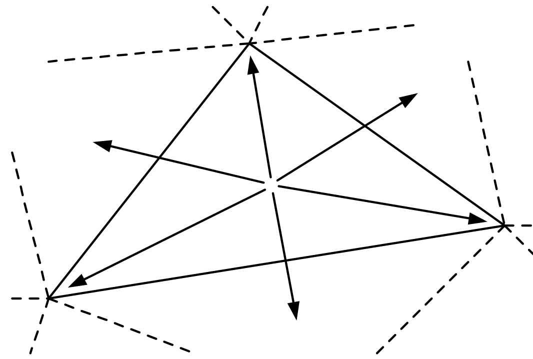

Рис. 10. Связи треугольников структуры “Узлы и треугольники”

1.2. Проверка условия Делоне. Одна из важнейших операций, выполняемых при построении триангуляции, является проверка условия Делоне для заданных пар треугольников. На основе определения триангуляции Делоне и теоремы 2 на практике обычно используют несколько способов проверки.

1. Проверка через уравнение описанной окружности.
2. Проверка с заранее вычисленной описанной окружностью.
3. Проверка суммы противолежащих углов.
4. Модифицированная проверка суммы противолежащих углов.

1.2.1. Проверка через уравнение описанной окружности. Уравнение окружности, проходящей через точки (x_1, y_1), (x_2, y_2), (x_3, y_3) можно записать в таком виде:

$$\begin{vmatrix} x^2+y^2 & x & y & 1 \\ x_1^2+y_1^2 & x_1 & y_1 & 1 \\ x_2^2+y_2^2 & x_2 & y_2 & 1 \\ x_3^2+y_3^2 & x_3 & y_3 & 1 \end{vmatrix} = 0.$$

Другая запись уравнения этой окружности имеет вид $(x^2 + y^2) \cdot a - x \cdot b + y \cdot c - d = 0$, где

$$a = \begin{vmatrix} x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \\ x_3 & y_3 & 1 \end{vmatrix}, \quad b = \begin{vmatrix} x_1^2+y_1^2 & y_1 & 1 \\ x_2^2+y_2^2 & y_2 & 1 \\ x_3^2+y_3^2 & y_3 & 1 \end{vmatrix}, \quad c = \begin{vmatrix} x_1^2+y_1^2 & x_1 & 1 \\ x_2^2+y_2^2 & x_2 & 1 \\ x_3^2+y_3^2 & x_3 & 1 \end{vmatrix}, \quad d = \begin{vmatrix} x_1^2+y_1^2 & x_1 & y_1 \\ x_2^2+y_2^2 & x_2 & y_2 \\ x_3^2+y_3^2 & x_3 & y_3 \end{vmatrix}. \tag{1}$$

Тогда условие Делоне для $\triangle\big((x_1,y_1),(x_2,y_2),(x_3,y_3)\big)$ выполняется только тогда, когда для любой другой точки (x_0,y_0) триангуляции справедливо неравенство $\big(a\cdot(x_0^2+y_0^2)-b\cdot x_0+c\cdot y_0-d\big)\cdot\operatorname{sgn}a\geqslant 0$, т.е. когда точка (x_0,y_0) не попадает внутрь окружности, описанной вокруг $\triangle\big((x_1,y_1),(x_2,y_2),(x_3,y_3)\big)$ [12]. Для упрощения вычислений можно заметить, что если тройка точек (x_1,y_1), (x_2,y_2), (x_3,y_3) является правой, то всегда $\operatorname{sgn}a=-1$, и наоборот, если тройка эта левая, то $\operatorname{sgn}a=1$.

Непосредственная реализация такой процедуры проверки требует 29 операций умножения и возведения в квадрат, а также 24 операции сложения и вычитания.

1.2.2. Проверка с заранее вычисленной описанной окружностью. Предыдущий вариант проверки требует значительного количества арифметических операций. Тем не менее, в большинстве алгоритмов триангуляции количество проверок условия многократно (в разных алгоритмах эта цифра колеблется от 2 до 25 и больше) превышает общее число различных треугольников, присутствовавших в триангуляции на разных шагах ее построения. Поэтому основная идея алгоритма проверки через заранее вычисленные окружности заключается в предварительном вычислении для каждого построенного треугольника центра и радиуса описанной вокруг него окружности, после чего проверка условия Делоне будет сводиться к вычислению расстояния до центра этой окружности и сравнению результата с радиусом. Центр (x_c,y_c) и радиус r окружности, описанной вокруг $\triangle\big((x_1,y_1),(x_2,y_2),(x_3,y_3)\big)$ можно найти как $x_c=b/2a$, $y_c=-c/2a$, $r^2=(b^2+c^2-4ad)/4a^2$, где значения a, b, c, d определены выше в (1).

Тогда условие Делоне для $\triangle\big((x_1,y_1),(x_2,y_2),(x_3,y_3)\big)$ будет выполняться только тогда, когда для любой другой точки (x_0,y_0) триангуляции будет $(x_0-x_c)^2+(y_0-y_c)^2\geqslant r^2$.

Реализация такой процедуры проверки требует для каждого треугольника 36 операций умножения, возведения в квадрат и деления, а также 22 операции сложения и вычитания. На этапе непосредственного выполнения проверок требуется всего только два возведения в квадрат, два вычитания, одно сложение и одно сравнение.

Если принять, что алгоритм триангуляции тратит в среднем по пять проверок на каждый треугольник, то в среднем данный способ проверки требует около девяти операций типа умножения и семь операций типа сложения. Если алгоритм тратит в среднем по 12 проверок на каждый треугольник, то количество операций уменьшается соответственно до шести и шести. Точная же оценка среднего числа операций должна выполняться для конкретного алгоритма триангуляции и типичных видов исходных данных.

1.2.3. Проверка суммы противолежащих углов. В [12, 24] показано, что условие Делоне для треугольника $\triangle\big((x_1,y_1),(x_2,y_2),(x_3,y_3)\big)$ будет выполняться только тогда, когда для любой другой точки (x_0,y_0) триангуляции будет $\alpha+\beta\leqslant\pi$ (рис. 11). Это условие эквивалентно $\sin(\alpha+\beta)\geqslant 0$, т.е.

$$\sin\alpha\cdot\cos\beta+\cos\alpha\cdot\sin\beta\geqslant 0. \tag{2}$$

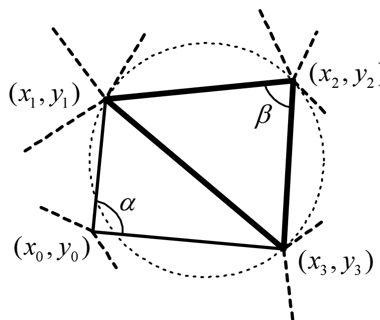

Рис. 11. Проверка суммы противолежащих углов

Значения синусов и косинусов углов можно вычислить через скалярные и векторные произведения векторов:

$$\cos\alpha=\frac{(x_0-x_1)(x_0-x_3)+(y_0-y_1)(y_0-y_3)}{\sqrt{(x_0-x_1)^2+(y_0-y_1)^2}\sqrt{(x_0-x_3)^2+(y_0-y_3)^2}},$$

$$\cos\beta=\frac{(x_2-x_1)(x_2-x_3)+(y_2-y_1)(y_2-y_3)}{\sqrt{(x_2-x_1)^2+(y_2-y_1)^2}\sqrt{(x_2-x_3)^2+(y_2-y_3)^2}},$$

$$\sin\alpha = \frac{(x_0-x_1)(y_0-y_3)-(x_0-x_3)(y_0-y_1)}{\sqrt{(x_0-x_1)^2+(y_0-y_1)^2}\sqrt{(x_0-x_3)^2+(y_0-y_3)^2}},$$

$$\sin\beta = \frac{(x_2-x_1)(y_2-y_3)-(x_2-x_3)(y_2-y_1)}{\sqrt{(x_2-x_1)^2+(y_2-y_1)^2}\sqrt{(x_2-x_3)^2+(y_2-y_3)^2}}.$$

Подставив эти значения в формулу (2) и сократив знаменатели дробей, получим следующую формулу проверки:

$$\begin{aligned}&\big((x_0-x_1)(y_0-y_3)-(x_0-x_3)(y_0-y_1)\big)\cdot\big((x_2-x_1)(x_2-x_3)+(y_2-y_1)(y_2-y_3)\big)+\\&+\big((x_0-x_1)(x_0-x_3)+(y_0-y_1)(y_0-y_3)\big)\cdot\big((x_2-x_1)(y_2-y_3)-(x_2-x_3)(y_2-y_1)\big)\geqslant 0.\end{aligned}\tag{3}$$

Непосредственная реализация такой процедуры проверки требует 10 операций умножения, а также 13 операций сложения и вычитания.

1.2.4. Модифицированная проверка суммы противолежащих углов. В [24] предложено вычислять не сразу все скалярные и векторные произведения. Вначале нужно вычислить только те выражения в (3), которые соответствуют $\cos\alpha$ и $\cos\beta$:

$$s_\alpha = (x_0-x_1)(x_0-x_3)+(y_0-y_1)(y_0-y_3), \quad s_\beta = (x_2-x_1)(x_2-x_3)+(y_2-y_1)(y_2-y_3).$$

Если $s_\alpha < 0$ и $s_\beta < 0$, то $\alpha > 90°$ и $\beta > 90°$ (Делоне не выполняется), а если $s_\alpha \geqslant 0$ и $s_\beta \geqslant 0$, то $\alpha \leqslant 90°$ и $\beta \leqslant 90°$ (условие Делоне выполняется); иначе требуются полные вычисления по формуле (3). Такое усовершенствование позволяет в среднем на 20–40 % (существенно зависит от алгоритма триангуляции) сократить количество выполняемых арифметических операций (примерно до семи умножений и до девяти сложений и вычитаний).

2. Итеративные алгоритмы. Все *итеративные алгоритмы* имеют в своей основе идею последовательного добавления точек в частично построенную триангуляцию Делоне. Формально этот процесс может быть описан так.

Пусть имеется триангуляция Делоне на множестве из $(n-1)$ точек. Очередная n-я точка добавляется в уже построенную структуру триангуляции следующим образом.

Шаг 1. Вначале производится локализация точки, т.е. находится треугольник (построенный ранее), в который попадает очередная точка. Если точка не попадает внутрь триангуляции, то находится треугольник на границе триангуляции, ближайший к очередной точке.

Шаг 2. Если точка попала на ранее вставленный узел триангуляции, то такая точка обычно отбрасывается, иначе точка вставляется в триангуляцию в виде нового узла. При этом, если точка попала на некоторое ребро, то оно разбивается на два новых, а оба смежных с ребром треугольника также делятся на два меньших. Если точка попала строго внутрь какого-нибудь треугольника, он разбивается на три новых. Если точка попала вне триангуляции, то строится один или более треугольников. Затем проводятся локальные проверки вновь полученных треугольников на соответствие условию Делоне и выполняются необходимые перестроения.

Сложность данного алгоритма складывается из трудоемкости поиска треугольника, в который на очередном шаге добавляется точка, трудоемкости построения новых треугольников, а также трудоемкости соответствующих перестроений структуры триангуляции в результате неудовлетворительных проверок пар соседних треугольников полученной триангуляции на выполнение условия Делоне.

При построении новых треугольников возможны две ситуации: добавляемая точка попадает либо внутрь триангуляции, либо вне ее. В первом случае строятся новые треугольники и число выполняемых алгоритмом действий фиксировано. Во втором — необходимо построение дополнительных внешних к текущей триангуляции треугольников, причем их количество может в худшем случае равняться $n-3$. Однако за все шаги работы алгоритма будет добавлено не более $3\cdot N$ треугольников, где N — общее число исходных точек. Поэтому в обоих случаях общее затрачиваемое время на построение треугольников составляет $O(N)$.

Чтобы несколько упростить алгоритм, можно вообще избавиться от второго случая, предварительно внеся в триангуляцию несколько таких дополнительных узлов, что построенная на них триангуляция заведомо накроет все исходные точки триангуляции. Такая структура обычно называется *суперструктурой*. На практике для суперструктуры обычно выбирают следующие варианты (рис. 12):

а) вершины равностороннего треугольника, покрывающего все множество исходных точек;

б) вершины квадрата, покрывающего все множество исходных точек;

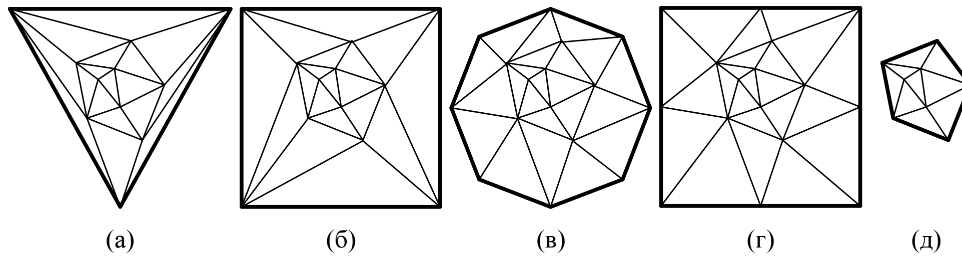

Рис. 12. Варианты суперструктур

в) $\theta(\sqrt{N})$ точек, равномерно распределенных по окружности, покрывающей все множество исходных точек;

г) $\theta(\sqrt{N})$ точек, равномерно распределенных по квадрату, покрывающему все множество исходных точек;

д) исходные точки, попадающие на выпуклую оболочку множества исходных точек.

В [8] приводятся результаты экспериментального сравнения различных вариантов суперструктур. При этом показывается, что при использовании суперструктуры на различных распределениях исходных данных возможно как увеличение скорости работы алгоритмов, так и ее снижение, но не более чем на 10 %.

Любое добавление новой точки в триангуляцию теоретически может нарушить целостность условия Делоне, поэтому после добавления точки обычно сразу же производится локальная проверка триангуляции на условие Делоне. Эта проверка должна охватить все вновь построенные треугольники и соседние с ними. Количество таких перестроений в худшем случае может привести к полному перестроению всей триангуляции, поэтому трудоемкость перестроений составляет $O(N)$. Однако среднее число таких перестроений на реальных данных составляет только около трех [8].

Таким образом, наибольший вклад в трудоемкость итеративного алгоритма дает процедура поиска очередного треугольника. Именно поэтому все итеративные алгоритмы построения триангуляции Делоне отличаются друг от друга главным образом только процедурой поиска очередного треугольника.

2.1. Простой итеративный алгоритм. В *простом итеративном алгоритме* поиск очередного треугольника реализуется следующим образом. Берется любой треугольник, уже принадлежащий триангуляции (например, выбирается случайно), и последовательными переходами по связанным треугольникам ищется искомый треугольник.

При этом в худшем случае приходится пересекать все треугольники триангуляции, поэтому трудоемкость такого поиска составляет $O(N)$. Однако в среднем для равномерного распределения в квадрате нужно совершить только $O(\sqrt{N})$ операций перехода [25]. Таким образом, трудоемкость простейшего итеративного алгоритма составляет в худшем случае $O(N^2)$, а в среднем — $O(N^{3/2})$ [15, 18].

Во многих практически важных случаях исходные точки не являются статистически независимыми, при этом i-я точка находится вблизи $(i+1)$-й. Поэтому в качестве начального треугольника для поиска можно брать треугольник, найденный ранее для предыдущей точки. Тем самым иногда удается достичь на некоторых видах исходных данных трудоемкости построения триангуляции в среднем $O(N)$.

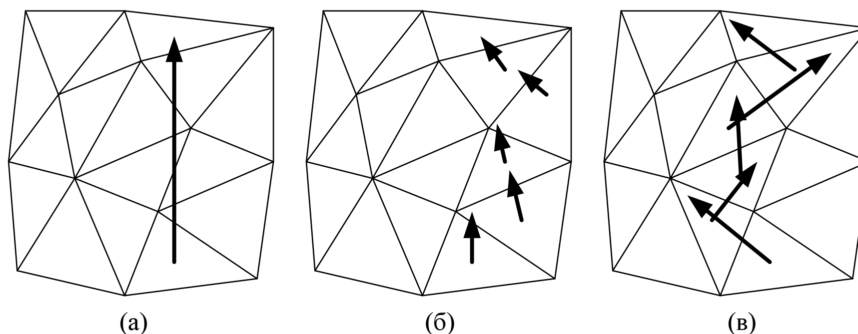

Рис. 13. Варианты локализации треугольника в итеративных алгоритмах

На практике обычно используются следующие способы поиска треугольника по заданной точке внутри него и по некоторому исходному треугольнику (рис. 13).

1. Проводится прямая через некоторую точку внутри исходного треугольника и целевую точку. Затем поиск осуществляется вдоль этой прямой к цели [15]. При этом необходимо корректно обрабатывать ситуации, когда на пути могут встретиться узлы и коллинеарные ребра.

2. Двигаться по одному шагу, на каждом шаге проводя прямую через центр текущего треугольника и целевую точку, и затем переходя к соседнему треугольнику, соответствующему стороне, которую пересекает построенная прямая [18].

3. Двигаться по одному шагу, на каждом из которых надо переходить через такое ребро текущего треугольника, что целевая точка и вершина текущего треугольника, противолежащая выбираемому пересекаемому ребру, лежат по разные стороны от прямой, определяемой данным ребром [8]. Этот способ обычно дает более длинный путь до цели, но он алгоритмически проще и поэтому быстрее. Для правильной работы данного алгоритма поиска существенным является то, что в триангуляции выполняется условие Делоне. Если условие Делоне нарушено, то иногда возможно зацикливание алгоритма.

После того как требуемый треугольник найден, в нем строятся новые узлы, ребра и треугольники, а затем производится локальное перестроение триангуляции.

2.1.1. Итеративный алгоритм “Удаляй и строй”. В *итеративном алгоритме “Удаляй и строй”* не выполняется никаких перестроений. Вместо этого при каждой вставке нового узла (рис. 14 а) сразу же удаляются все треугольники, у которых внутрь описанных окружностей попадает новый узел (рис. 14 б). При этом все удаленные треугольники неявно образуют некоторый полигон. После этого на месте удаленных треугольников строится заполняющая триангуляция путем соединения нового узла с этим полигоном (рис. 14 в) [27].

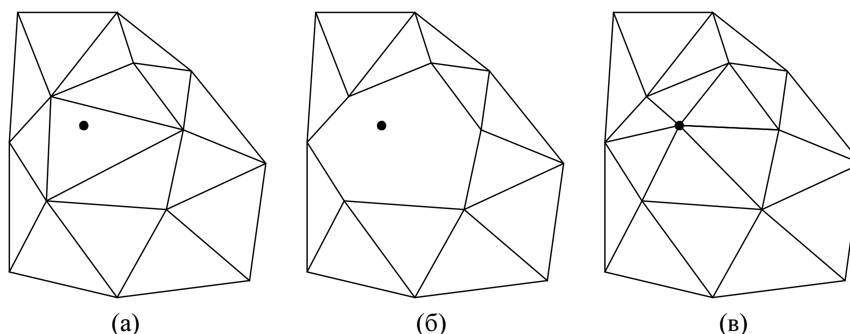

Рис. 14. Вставка точки в итеративном алгоритме “Удаляй и строй”

Данный алгоритм строит сразу все необходимые треугольники в отличие от обычного итеративного алгоритма, где при вставке одного узла возможны многократные перестроения одного и того же треугольника. Однако здесь на первый план выходит процедура выделения контура удаленного полигона, от эффективности работы которого зависит общая скорость алгоритма. В целом, в зависимости от используемой структуры данных этот алгоритм может тратить времени меньше, чем алгоритм с перестроениями, и наоборот.

Трудоемкость данного алгоритма полностью совпадает с оценками для простого итеративного алгоритма.

2.2. Алгоритмы с индексированием поиска треугольников. В *алгоритмах с индексированием поиска* все ранее построенные треугольники заносятся в некоторую структуру, с помощью которой можно достаточно быстро находить треугольники в заданной точке плоскости.

2.2.1. Итеративный алгоритм с индексированием треугольников. В *алгоритме триангуляции с индексированием треугольников* для всех построенных треугольников вычисляется минимальный объемлющий прямоугольник со сторонами, параллельными осям координат, и заносится в R-дерево [13]. При удалении старых треугольников необходимо их удалять из R-дерева, а при построении новых — заносить.

Для выполнения поиска треугольника, в который попадает текущая вставляемая в триангуляцию точка, необходимо выполнить стандартный точечный запрос к R-дереву и получить список треугольников, чьи объемлющие прямоугольники находятся в данной точке. Затем надо выбрать из них тот треугольник, внутрь которого попадает точка.

Отметим, что структура R-дерева не позволяет найти объект, ближайший к заданной точке. Именно поэтому данный алгоритм триангуляции с использованием R-дерева следует применять только с суперструктурой, чтобы исключить попадание очередной точки вне триангуляции.

Трудоемкость поиска треугольника в R-дереве в худшем случае составляет $O(N)$, а в среднем — $O(\log N)$. При этом может быть найдено от одного до N треугольников, которые надо затем все проверить. Кроме того, появляются дополнительные затраты времени на поддержание структуры дерева: $O(\log N)$ при каждом построении и удалении треугольников. Отсюда получаем, что трудоемкость алгоритма триангуляции с индексированием треугольников в худшем случае составляет $O(N^2)$, а в среднем — $O(N \log N)$.

2.2.2. Итеративный алгоритм с индексированием центров треугольников k-D-деревом. В *алгоритме триангуляции с индексированием центров треугольников k-D-деревом* в дерево [13] помещаются только центры треугольников. При удалении старых треугольников необходимо удалять их центры из k-D-дерева, а при построении новых — заносить.

Для выполнения поиска треугольника, в который попадает текущая вставляемая в триангуляцию точка, необходимо выполнить нестандартный точечный запрос к k-D-дереву. Поиск в дереве необходимо начинать с корня и спускаться вниз до листьев. В случае если потомки текущего узла k-D-дерева (охватывающий потомки прямоугольник) не покрывают текущую точку, то необходимо выбрать для дальнейшего спуска по дереву потомка, ближайшего к точке поиска.

В результате будет найден некоторый треугольник, центр которого будет близок к заданной точке. Если в найденный треугольник не попадает заданная точка, то далее необходимо использовать обычный алгоритм поиска треугольника из простого итеративного алгоритма построения триангуляции Делоне.

Трудоемкость поиска точки в k-D-дереве в худшем случае составляет $O(N)$, а в среднем — $O(\log N)$. Далее может быть задействована процедура перехода по треугольникам с трудоемкостью $O(N)$ в худшем случае. Дополнительные затраты времени на поддержание структуры дерева оцениваются как $O(\log N)$ при каждом построении и удалении треугольников. Отсюда получаем, что трудоемкость алгоритма триангуляции с индексированием треугольников в худшем случае составляет $O(N^2)$, а в среднем — $O(N \log N)$

2.2.3. Итеративный алгоритм с индексированием центров треугольников квадродеревом. При использовании этого алгоритма в дерево также помещаются только центры треугольников [6, 13]. В целом работа алгоритма и оценка трудоемкости совпадает с предыдущим алгоритмом триангуляции. Однако в отличие от алгоритма с k-D-деревом квадродерево более просто в реализации и позволяет более точно находить ближайший треугольник. В то же время, на неравномерных распределениях квадродерево уступает k-D-дереву.

2.3. Алгоритмы с кэшированием поиска треугольников. *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 измененные треугольники из кэша не удаляются (предполагается, что каждый удаленный треугольник как запись в памяти компьютера превращается в новый треугольник и поэтому допустимость ссылок на треугольники не нарушается при работе алгоритма), один и тот же треугольник может многократно находиться в кэше, а некоторые треугольники могут вообще там отсутствовать [8].

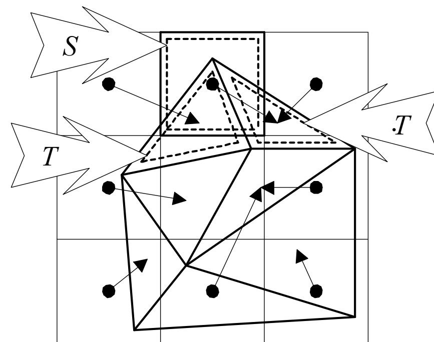

Рис. 15. Локализация точек в кэше (S — найденный квадрат, $\tilde{T}$ — связанный с квадратом треугольник, T — конечный треугольник)

Основная идея кэширования заключается в построении некоторого более простого планарного разбиения плоскости, чем триангуляция, в котором можно быстро выполнять локализацию точек. Для каждого элемента простого разбиения делается ссылка на треугольник триангуляции. Процедура поиска сводится к локализации элемента простого разбиения, перехода по ссылке к треугольнику и последующей локали-

зации искомого треугольника алгоритмом из простого итеративного алгоритма триангуляции Делоне. В качестве такого разбиения проще всего использовать регулярную сеть квадратов (рис. 15). Например, если данное планарное разбиение полностью покрывается квадратом $[0;1] \times [0;1]$, то его можно разбить на m^2 равных квадратов. Занумеруем их всех естественным образом двумя параметрами $i, j = \overline{0, m-1}$. Тогда по данной точке (x, y) мы можем найти квадрат $\lfloor x/m \rfloor, \lfloor y/m \rfloor$, где $\lfloor \ldots \rfloor$ — знак взятия целой части числа.

Кэш в виде регулярной сети квадратов наиболее хорошо работает для равномерного распределения исходных точек и распределений, не имеющих высоких пиков в функции плотности. В случае же если заранее известен характер распределения, можно выбрать какое-то иное разбиение плоскости (например, в виде неравномерно отстоящих вертикальных и горизонтальных прямых).

2.3.1. Итеративный алгоритм со статическим кэшированием поиска. В *алгоритме триангуляции со статическим кэшированием поиска* необходимо выбрать число m и завести кэш в виде двумерного массива r размером $m \times m$ для ссылок на треугольники [8]. Первоначально этот массив надо заполнить ссылками на самый первый построенный треугольник. Затем, после выполнения очередного поиска, в котором был найден некоторый треугольник T, начиная поиск с квадрата (i, j), необходимо обновить информацию в кэше: $r_{i,j} :=$ ссылка на T. Размер статического кэша следует выбирать по формуле $m = s \cdot N^{3/8}$, где s — коэффициент статического кэша. На практике значение s следует взять $\approx 0{,}6 - 0{,}9$.

Первое время, пока кэш не обновится полностью, поиск может идти довольно долго, но потом скорость повышается. Этого недостатка лишен следующий алгоритм.

2.3.2. Итеративный алгоритм с динамическим кэшированием поиска. В *алгоритме триангуляции с динамическим кэшированием поиска* необходимо завести кэш минимального размера, например 2×2. По мере роста числа добавленных в триангуляцию точек необходимо последовательно увеличивать его размер в два раза, переписывая при этом информацию из старого кэша в новый. При этом для увеличения кэша надо выполнить следующие пересылки (r — старый кэш, r' — новый): $\forall i, j = \overline{0, m-1}$: $r'_{2i,2j}, r'_{2i,2j+1}, r'_{2i+1,2j}, r'_{2i+1,2j+1} := r_{i,j}$. Данный алгоритм кэширования позволяет одинаково эффективно работать на малом и большом количестве точек, заранее не зная их числа.

Увеличение размера динамического кэша в два раза следует производить каждый раз при достижении числа точек в триангуляции $n = r \cdot m^2$, где r — коэффициент роста динамического кэша, а m — текущий размер кэша. На практике значение коэффициента роста динамического кэша следует выбрать $\approx 3 - 8$.

Трудоемкости алгоритмов триангуляции с кэшированием как и всех итеративных алгоритмов составляют в худшем случае $O(N^2)$, а в среднем на равномерном распределении для статического кэширования — $O(N^{9/8})$ и для динамического кэширования — $O(N)$ [8].

Для большинства случайных распределений исходных точек данный алгоритм работает значительно быстрее всех остальных алгоритмов [8]. Однако на некоторых реальных данных, в которых последовательные исходные точки находятся вблизи друг друга (например, точки изолиний карт рельефа), алгоритм динамического кэширования может тратить большее время, чем другие алгоритмы. Для учета такой ситуации в алгоритм следует добавить дополнительную проверку. Если очередная добавляемая точка находится от предыдущей точки на расстоянии большем, чем некоторое число $\triangle$ (порядка текущего размера клетки кэша), то поиск необходимо начать с треугольника из кэша, иначе нужно начать с последнего построенного треугольника. В такой модификации алгоритм динамического кэширования становится непревзойденным по скорости работы на большинстве реальных данных.

2.4. Итеративные алгоритмы с измененным порядком добавления точек. В [15] предлагается изменить порядок добавления точек так, чтобы каждая следующая точка была максимально близка к предыдущей добавленной точке. Тогда, запоминая треугольник, найденный на предыдущей итерации, можно использовать его в качестве отправной точки для текущего поиска, используя алгоритм поиска из простого итеративного алгоритма. Удачно перестраивая порядок добавления точек, можно достичь очень неплохих результатов. Однако при этом на первый план может выйти трудоемкость этой предобработки.

2.4.1. Итеративный полосовой алгоритм. В *итеративном полосовом алгоритме триангуляции* нужно разбить все точки на полосы по одной координате, а затем отсортировать все точки внутри полос по другой координате [8]. В этом случае, подобрав соответствующее количество полос, можно существенно уменьшить расстояние между последовательностью точек (рис. 16). В [4] теоретически определено оптимальное количество полос $m = \left\lfloor \sqrt{bN/3a} \right\rfloor$ для разбиения точек на полосы при равномерном независимом распределении точек в прямоугольнике размером $b \times a$, исходя из условия минимизации суммарного общего расстояния между последовательными точками разбиения. В данной оценке, к со-

жалению, не учтено время, затрачиваемое на предобработку — разбиение на полосы. Поэтому на практике число полос лучше выбирать все же в несколько раз меньше, чем приведенная оценка, по формуле $\overline{m} = \left\lfloor \sqrt{sbN/a} \right\rfloor$, где s — константа разбиения на полосы итеративного полосового алгоритма. При этом значение s следует выбирать $\approx 0{,}15 - 0{,}19$.

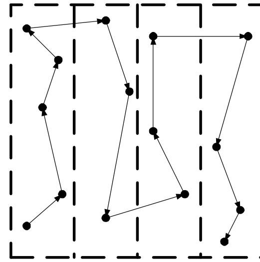

Рис. 16. Полосовой итеративный алгоритм

Трудоемкость данного алгоритма составляет в худшем случае $O(N^2)$, а в среднем при использовании алгоритма сортировки с линейной сложностью (например, цифровой сортировки) — $O(N)$.

2.4.2. Итеративный квадратный алгоритм. В *итеративном квадратном алгоритме триангуляции* [8, 18] необходимо разбить плоскость с точками на $\theta(\sqrt{N})$ квадратов, а добавление точек производить последовательными группами, соответствующими смежным квадратам (рис. 17).

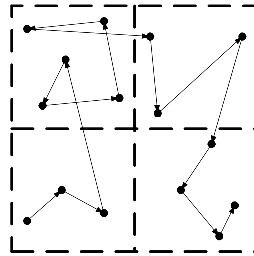

Рис. 17. Квадратный итеративный алгоритм

В [18] разбиение производится на $\sqrt{N}$ квадратов; трудоемкость такого алгоритма составляет в худшем случае $O(N^2)$, а в среднем — $O(N^{3/2})$.

В [8] разбиение производится на $m \times m$ квадратов, где $m = \left\lfloor \sqrt{sN} \right\rfloor$. Значение s следует выбирать $\approx 0{,}004 - 0{,}006$. При этом трудоемкость такого алгоритма составляет в худшем случае $O(N^2)$, а в среднем — $O(N)$. На практике данный алгоритм работает немного быстрее, чем предыдущий (примерно на 10–20 %) [8].

2.4.3. Итеративный алгоритм с послойным сгущением. В *итеративном алгоритме триангуляции с послойным сгущением* [10] необходимо разбить плоскость с точками на $n = (2^u + 1) \times (2^v + 1)$ элементарных ячеек-квадратов одинакового размера. Каждый квадрат нумеруется от 0 до 2^u по горизонтали и от 0 до 2^v по вертикали. Далее вводится понятие слоя. Считается, что точка принадлежит слою i, если оба номера ее квадрата кратны 2^i (тогда все исходные точки образуют слой 0, слой $i + 1$ будет подмножеством слоя i, а максимальный номер слоя $k = \min(u, v)$). По значениям пар номеров все точки слоя i делятся на четыре подмножества:

1) угловые точки (оба их номера кратны 2^{i+1}) — это слой $i + 1$;
2) внутренние точки (оба их номера не кратны 2^{i+1});
3) X-граничные точки (только номер по координате X кратен 2^{i+1});
4) Y-граничные точки (только номер по координате Y кратен 2^{i+1}).

Добавление точек в триангуляцию надо производить послойно от слоя с максимальным номером до нулевого. Внутри слоя нужно вначале внести все точки второго типа, затем третьего и в конце — четвертого.

На рис. 18 приведен пример разбиения плоскости на квадраты при $u = 3$, $v = 2$ и $n = 9 \cdot 5$ по этапам:
а) все точки слоя 1;
б) внутренние точки слоя 0;
в) X-граничные точки слоя 0;

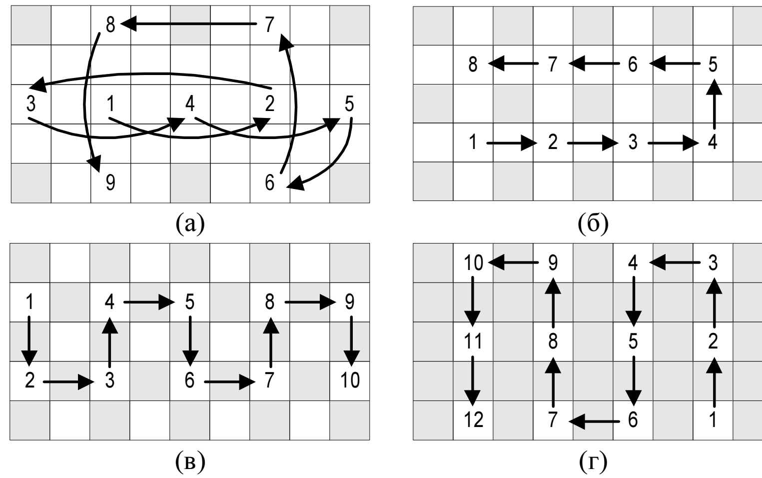

Рис. 18. Порядок выбора точек в итеративном алгоритме с послойным сгущением

г) Y-граничные точки слоя 0.

На рис. 18 числа (от 1 и больше) определяют порядок выбора квадратов (и соответственно точек внутри них) на каждом этапе; ранее обработанные квадраты затемнены.

Для произвольных наборов точек такой алгоритм (как и все другие итеративные алгоритмы) имеет трудоемкость $O(N^2)$. Если исходные точки имеют равномерное распределение, то трудоемкость алгоритма в среднем случае будет $O(N)$. Кроме того, в [10] показывается, что если исходные точки удовлетворяют некоторым фиксированным (не вероятностным) ограничениям, то трудоемкость алгоритма будет и в худшем случае $O(N)$.

Недостатком предыдущих двух алгоритмов (полосового и квадратного) является то, что при вставке каждой новой точки происходит частое построение длинных узких треугольников, которые в дальнейшем перестраиваются. В алгоритме со сгущением точек, как правило, удается избавиться от таких ситуаций, равномерно последовательно вставляя в триангуляцию узлы. За счет этого данный алгоритм строит меньше узких треугольников и поэтому быстрее выполняется на реальных данных, чем многие другие алгоритмы.

2.4.4. Итеративные алгоритмы с сортировкой вдоль кривой, заполняющей плоскость. Идея этих алгоритмов заключается в "разворачивании" плоскости в одну прямую, при этом близкие точки на этой прямой будут также близки и на исходной плоскости. В теории фракталов известно значительное количество кривых, заполняющих плоскость. Одними из наиболее известных и удобных для применения в задаче построения триангуляции являются кривые Пеано и Гильберта [5].

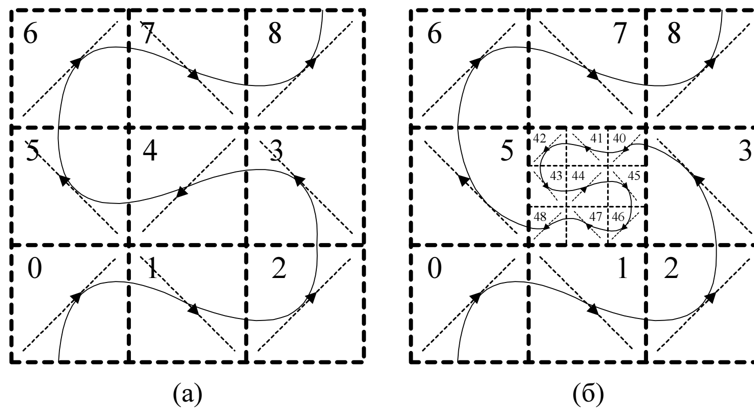

Рис. 19. Построение кривой Пеано (а – первая итерация, б – вторая итерация в центральном квадрате)

При построении кривой Пеано используется представление точек в системе счисления по основанию 9. На первом шаге область определения исходных точек триангуляции делится на девять равных квадратов. Эти квадраты нумеруются от 0 до 8; образуется кривая первой итерации (рис. 19а). Далее каждый из

квадратов делится еще на девять частей; к номеру, полученному на первой итерации, в конце добавляется новая цифра (рис. 19б).

При построении триангуляции необходимо для каждой исходной точки вычислить код Пеано, отсортировать точки по этому коду и в этом порядке вносить их в триангуляцию.

Аналогично кривой Пеано можно использовать кривую Гильберта. При этом деление области размещения исходных точек выполняется на четыре части в соответствии с рис. 20.

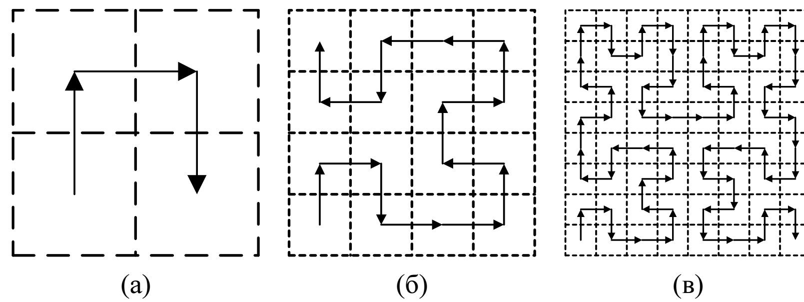

Рис. 20. Построение кривой Гильберта

Трудоемкость итеративных алгоритмов с сортировкой вдоль кривых Пеано и Гильберта составляет в худшем случае $O(N^2)$, а в среднем — $O(N)$. На практике данные алгоритмы работают примерно с той же скоростью, что и итеративный полосовой алгоритм.

2.4.5. Итеративный алгоритм с сортировкой по Z-коду. В *итеративном алгоритме с сортировкой по Z-коду* для каждой точки плоскости (x, y) строится специальный Z-код. Затем выполняется сортировка всех точек по этому коду, после чего точки вставляются в триангуляцию в этом порядке.

Для построения Z-кода нужно разбить прямоугольную область размещения исходных точек на четыре равных прямоугольника и занумеровать их двоичными числами от 0 до 11_2 в соответствии с рис. 21 а. Далее каждый из полученных прямоугольников надо опять разбить на четыре части, опять занумеровать их двоичными числами от 0 до 11_2 и добавить к концу кода, полученному ранее (рис. 21 б). При необходимости так можно рекурсивно продолжать достаточно далеко.

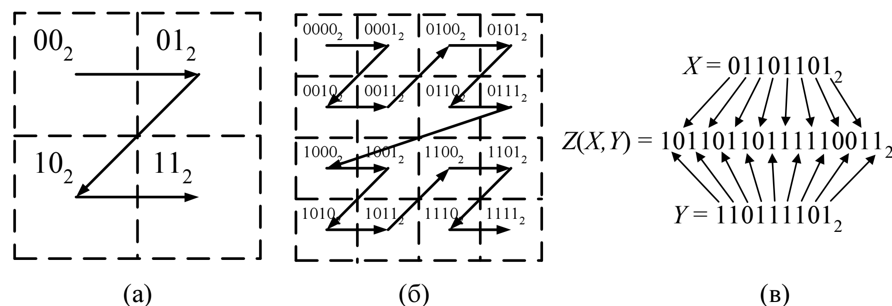

Рис. 21. Итеративный алгоритм с сортировкой по Z-коду (а, б – порядок вставки точек в триангуляцию, в – побитовое получение Z-кода)

Однако на практике столь сложные манипуляции проделывать не нужно. В действительности Z-код можно очень просто получить. Для этого нужно вначале перейти от исходных координат (x, y) к целочисленным координатам (X, Y), изменяющимся в диапазоне $(0; 2^k - 1)$, где k — некоторое целое число. После этого нужно взять по очереди все биты координат X и Y с первого по последний и сформировать новую битовую последовательность (рис. 21 в).

В отличие от кодов Пеано и Гильберта в Z-коде близкие значения кода не всегда соответствуют близкому расположению на плоскости, поэтому поиск треугольников иногда выполняется дольше, чем в предыдущем алгоритме. Но при этом сама процедура вычисления кода работает значительно быстрее.

Трудоемкость данного алгоритма составляет в худшем случае $O(N^2)$, а в среднем — $O(N)$. На практике данный алгоритм показывает практически ту же самую скорость работы, что и алгоритмы с сортировкой вдоль кривой, заполняющей плоскость.

3. Алгоритмы слияния. Концептуально все *алгоритмы слияния* предполагают разбиение исходного множества точек на несколько подмножеств, построение триангуляций на этих подмножествах, а

затем объединение (слияние) нескольких триангуляций в одно целое.

3.4. Алгоритм слияния “Разделяй и властвуй”. В этом алгоритме [12, 18] множество точек разбивается на две как можно более равные части, причем на разных уровнях рекурсии обычно применяется разбиение горизонтальными и вертикальными линиями по очереди (рис. 22). Алгоритм триангуляции рекурсивно применяется к подчастям, а затем производится *слияние* (объединение) полученных подтриангуляций. Рекурсия прекращается при разбиении всего множества на достаточно маленькие части, которые можно легко протриангулировать каким-нибудь другим простым способом. На практике удобно разбивать все множество на части по три и по четыре точки.

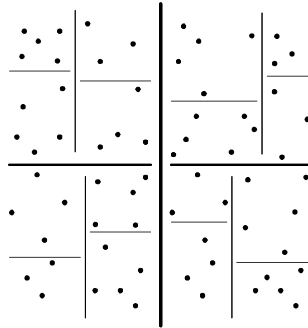

Рис. 22. Разбиение множества точек в алгоритме триангуляции “Разделяй и властвуй”

Если допустить, что число точек в триангуляции всегда больше пяти, то все множество точек можно разбить на элементарные части по три и по четыре точки. Действительно, $\forall N > 5$:

$$\left.\begin{array}{l} N = 3k; \\ N = 3k + 1 = 3(k-1) + 4; \\ N = 3k + 2 = 3(k-2) + 4 \cdot 2; \end{array}\right\} k \geqslant 2, N > 5.$$

Рекурсивный алгоритм триангуляции “Разделяй и властвуй” можно условно записать следующим образом.

Шаг 1. Если число точек $N = 3$, то построить триангуляцию из одного треугольника.

Шаг 2. Иначе, если число точек $N = 4$, то построить триангуляцию из двух или трех треугольников.

Шаг 3. Иначе, если число точек $N = 8$, то разбить множества точек на две части по четыре точки, рекурсивно применить алгоритм, а затем склеить триангуляции.

Шаг 4. Иначе, если число точек $N < 12$, то разбить множества точек на две части по три и $N - 3$ точки, рекурсивно применить алгоритм, а затем склеить триангуляции.

Шаг 5. Иначе (число точек $N \geqslant 12$) разбить множества точек на две части по $\lfloor N/2 \rfloor$ и $\lceil N/2 \rceil$ точки, рекурсивно применить алгоритм, а затем склеить триангуляции.

Элементарные множества из трех или из четырех элементов (рис. 23) легко триангулируются (можно даже просто перебрать множество всех возможных вариантов).

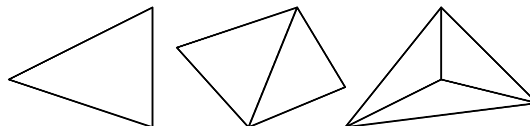

Рис. 23. Триангуляции множеств из трех и четырех точек

Трудоемкость алгоритма “Разделяй и властвуй” составляет $O(N \log N)$ в среднем и худшем случаях. Основная и самая сложная часть этого алгоритма заключается в слиянии двух частичных триангуляций. Для этого может использоваться несколько эквивалентных процедур слияния (обратим внимание, что все создаваемые частичные триангуляции являются выпуклыми и нижеследующие процедуры существенно используют этот факт):

1. “Удаляй-и-строй”.
2. “Строй-и-перестраивай”.
3. “Строй, перестраивая”.

3.1.1. Слияние триангуляций “Удаляй-и-строй”. Данная процедура слияния триангуляций была предложена в [12, 18] как часть соответствующего алгоритма триангуляции “Разделяй и властвуй”.

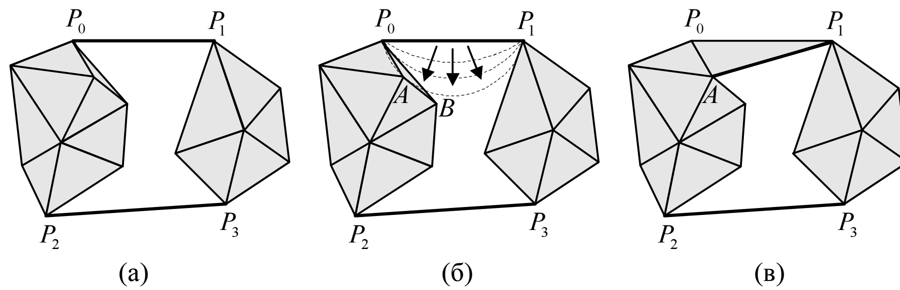

Рис. 24. Слияние триангуляций (а — поиск касательных, б — поиск ближайшего соседа Делоне для базовой линии, в — построение треугольника)

Вначале для двух триангуляций находятся две общие касательные P_0P_1 и P_2P_3, первая из которых становится текущей базовой линией (рис. 24 а). Затем от базовой линии начинается заполнение треугольниками промежутка между триангуляциями. Для этого необходимо найти ближайший к базовой линии узел любой из двух частичных триангуляций — так называемого соседа Делоне для базовой линии. Поиск этого соседа можно представить себе как рост "пузыря" от базовой линии, пока не встретится какой-нибудь узел. "Пузырь" — это окружность, проходящая через точки P_0 и P_1, с центром на срединном перпендикуляре к базовой линии. Например, на рис. 24 б первым таким найденным узлом становится точка A. На найденном узле и базовой линии строится новый треугольник (в нашем случае $\triangle P_0P_1A$). Однако при этом иногда возникает необходимость предварительно удалить некоторые ранее построенные треугольники, которые перекрываются новым треугольником. Так, на рис. 24 б должен быть удален $\triangle P_0BA$. После этого открытая сторона нового треугольника становится новой базовой линией (на рис. 24 в — AP_1), и цикл продолжается, пока не будет достигнута вторая касательная.

Данная процедура слияния очень похожа на алгоритмы прямого построения триангуляции, обсуждаемые ниже. Поэтому ей свойственны те же самые недостатки, а именно относительно большие затраты времени на поиск очередного соседа Делоне. В целом эта процедура имеет трудоемкость $O(N)$ относительно общего количества точек в двух объединяемых триангуляциях [18].

3.1.2. Слияние триангуляций "Строй-и-перестраивай". В *процедуре слияния триангуляций "Строй-и-перестраивай"* имеются два этапа [8]. Вначале строятся заполняющие зону слияния треугольники между касательными (рис. 25 а). Для этого первая касательная P_0P_1 делается текущей базовой линией Q_1Q_2. Относительно текущей базовой линии рассматриваются две следующие точки N_1 и N_2 вдоль границ сливаемых триангуляций. Из двух треугольников $\triangle Q_1Q_2N_1$ и $\triangle Q_1Q_2N_2$ выбирается тот, который, во-первых, можно построить (т.е. $\angle Q_2Q_1N_1 < 180°$ для первого и $\angle Q_1Q_2N_2 < 180°$ для второго треугольника), и, во-вторых, максимум минимального угла которого больше (так выбирается 'более равносторонний" треугольник, который с меньшей вероятностью будет перестроен в будущем). После этого открытая сторона вновь построенного треугольника становится новой базовой линией (рис. 25 б). Далее цикл построения треугольников продолжается, пока не будет достигнута вторая касательная.

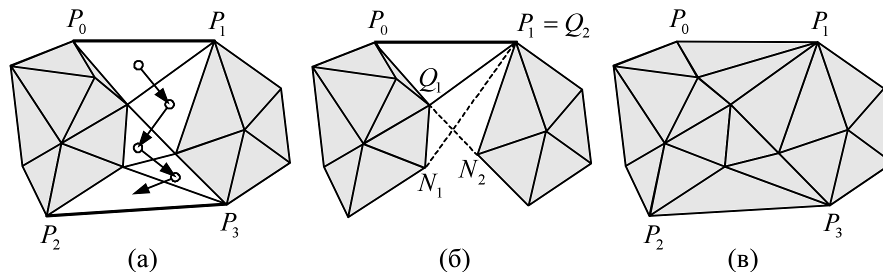

Рис. 25. Слияние триангуляций (а — предварительное слияние, б — выбор очередного треугольника, в — финальная триангуляция после перестроений треугольников)

На втором этапе все вновь построенные треугольники проверяются на выполнение условия Делоне с другими треугольниками и при необходимости выполняются перестроения треугольников (рис. 25 в).

Процедура слияния "Строй-и-перестраивай" значительно проще в реализации предыдущей и обычно работает заметно быстрее на реальных данных. В целом она имеет в худшем случае трудоемкость $O(N)$

относительно общего количества точек в двух сливаемых триангуляциях, а в среднем на большинстве распространенных распределений $O(\sqrt{N})$ или $O(M)$, где M — общее количество точек вдоль границ сливаемых триангуляций [8].

3.1.3. Слияние триангуляций "Строй, перестраивая". Процедура, реализующая это слияние, отличается от предыдущей только тем, что перестроения выполняются не на втором этапе работы, а непосредственно после построения очередного треугольника (рис. 26) [8]. В таком случае отпадает необходимость помнить список всех построенных треугольников, несколько уменьшается общее количество проверок условия Делоне и количество выполненных перестроений. Однако при этом для некоторых структур данных (например, "Узлы и треугольник") необходимо прилагать дополнительные усилия для сохранения ссылки на текущую базовую линию, т.к. образующий ее треугольник может быть перестроен.

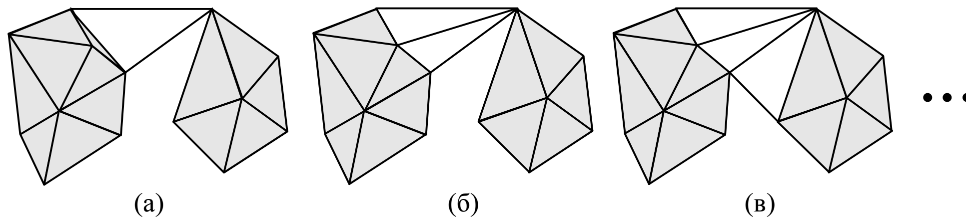

Рис. 26. Шаги слияния "Строй, перестраивая" (а — построение первого треугольника, б — немедленные перестроения, в — построение второго треугольника)

В целом эта процедура имеет такую же трудоемкость, что и предыдущая, и на практике работает немного медленнее ее.

3.2. Рекурсивный алгоритм с разрезанием по диаметру. Данный алгоритм триангуляции похож на "Разделяй и властвуй", но отличается от него способом разделения множества исходных точек на две части и процедурой слияния двух частичных триангуляций. Используемая здесь процедура слияния описана в [23].

Для разделения множества точек на две части вначале строится выпуклая оболочка всех исходных точек (эта операция выполняется за время $O(N)$ [7]). Далее по выпуклой оболочке вычисляется диаметр D_1D_2 (рис. 27 а) [7]. Трудоемкость этой операции составляет в худшем случае $O(N \log N)$, а в среднем — $O(N)$. Затем необходимо найти такую пару точек P_1 и P_2, чтобы отрезок P_1P_2 был "почти" перпендикулярен диаметру D_1D_2 и делил множество всех исходных точек примерно на две равные части. В качестве первого приближения для P_1 и P_2 можно взять точки посередине вдоль выпуклой оболочки между D_1 и D_2 (рис. 27 а).

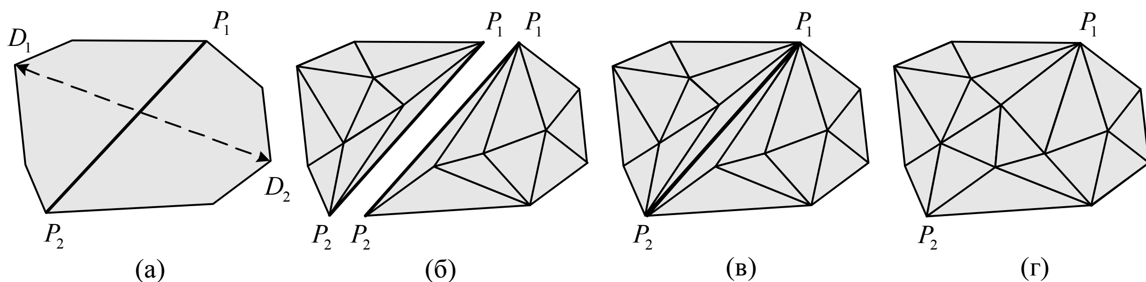

Рис. 27. Слияние триангуляций (а — поиск диаметра и точек разделения, б — раздельная триангуляция, в — соединение триангуляций по общему ребру, г — перестроения)

После выбора P_1 и P_2 все множество исходных точек делится на две части, причем P_1 и P_2 попадают в оба множества. После этого данный алгоритм триангуляции применяется рекурсивно к обеим частям (рис. 27 б). Затем обе полученные триангуляции соединяются вдоль общего ребра P_1P_2 (рис. 27 в) и выполняются необходимые перестроения пар соседних треугольников для выполнения условия Делоне (рис. 27 г).

Данный алгоритм слияния по сравнению с "Разделяй и властвуй" сложнее в процедуре разделения точек на части, но проще в слиянии. Трудоемкость алгоритма с разрезанием по диаметру составляет в худшем и в среднем случаях $O(N \log N)$. В целом алгоритм работает немного медленнее, чем "Разделяй и властвуй".

3.3. Полосовые алгоритмы слияния. Логарифмическая составляющая в трудоемкости двух предыдущих алгоритмов порождена их рекурсивным характером. Избавившись от рекурсии, можно попытаться улучшить и трудоемкость триангуляции. В [8] предлагается разбить исходное множество точек на такие подмножества, чтобы построение по ним триангуляций занимало минимальное время (например, за счет применения специальных алгоритмов, оптимизированных для конкретных конфигураций точек).

Основная идея *полосовых алгоритмов слияния* предполагает разбиение всего исходного множества точек на некоторые полосы и применение быстрого алгоритма получения невыпуклой триангуляции полосы точек. После получения множества частичных полосовых триангуляций они объединяются; при этом, так как полосы невыпуклые, приходится 1) либо достраивать полосы до выпуклых и затем использовать обычный алгоритм слияния из алгоритма "Разделяй и властвуй", 2) либо применять более сложный, но более эффективный алгоритм, позволяющий соединять невыпуклые триангуляции.

В целом алгоритмы полосового слияния логически состоят из трех последовательных шагов.

Шаг 1. Разбиение всего исходного множества точек на некоторые полосы.

Шаг 2. Применение специального быстрого алгоритма получения невыпуклой триангуляции полосы точек.

Шаг 3. Слияние полученных подтриангуляций.

Рассмотрим эти шаги подробнее.

Шаг 1. Множество всех точек разбивается по некоторому количеству столбцов по принципу одинаковой ширины столбцов или одинакового количества точек в столбцах (с помощью цифровой сортировки). Количество точек в каждом столбце должно получиться не менее трех (этого требует алгоритм, применяемый на следующем шаге алгоритма). Если это не выполняется для какого-либо столбца, то его нужно присоединить к соседнему. Трудоемкость данного шага составляет $O(N)$ в соответствии с трудоемкостью применяемых алгоритмов разбиения.

Шаг 2. Все точки внутри столбцов сортируются по вертикали (по координате Y), и затем каждый столбец триангулируется по отдельности. Для этого используется специальный алгоритм триангуляции (рис. 28 а). Вначале на трех самых верхних точках в столбце строится первый треугольник и помечается как текущий. Затем последовательно перебираются все остальные точки в столбце (начиная с четвертой) сверху вниз и добавляются к частичной триангуляции. Пусть $\triangle ABC$ является текущим, точка B имеет наименьшую из точек треугольника координату, а X — очередная добавляемая точка из столбца. На очередном шаге необходимо сделать выбор очередного создаваемого $\triangle ABX$ или $\triangle BCX$ (рис. 28 б). Иногда один из этих треугольников построить невозможно ввиду получаемых тогда пересечений различных отрезков триангуляции, и вопрос выбора тогда не стоит. Если же построить можно оба треугольника, естественно выбрать тот, у которого минимальный из углов больше, т.к. тогда с большей вероятностью в будущем не придется его перестраивать из-за невыполнения условия Делоне.

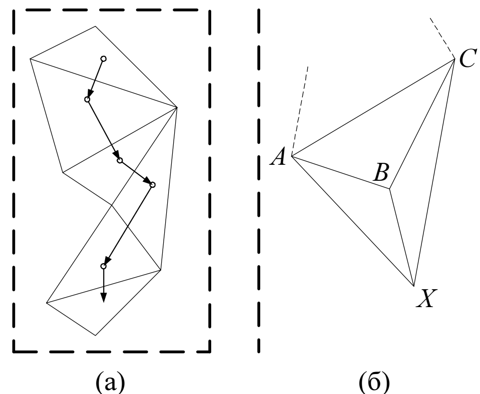

Рис. 28. Триангуляция полосы точек

После построения очередного треугольника надо проверить условие Делоне для вновь образовавшихся пар соседних треугольников и при необходимости перестроить их. Заметим, что при таком алгоритме будет построена невыпуклая триангуляция полосы точек.

Одним из важнейших параметров алгоритма триангуляции столбца точек является выбор количества полос, который осуществляется исходя из минимизации числа последующих перестроений пар соседних треугольников. Если предположить, что все N исходных точек распределены в прямоугольной области шириной a и высотой b, то число полос вычисляется как $m = \sqrt{sNa/b}$, где s — коэффициент разбиения [8].

На практике значение s следует взять $\approx 0{,}11 - 0{,}15$.

Трудоемкость данного шага будет в среднем $O(N)$ при условии выбора оптимального количества полос.

3.3.1. Алгоритм выпуклого полосового слияния. Пошаговая схема работы *алгоритма выпуклого полосового слияния* схематично изображена на рис. 29 а–г. Основная его идея заключается в построении выпуклых полосовых триангуляций и последующего применения любого алгоритма слияния, используемого в алгоритме “Разделяй и властвуй”.

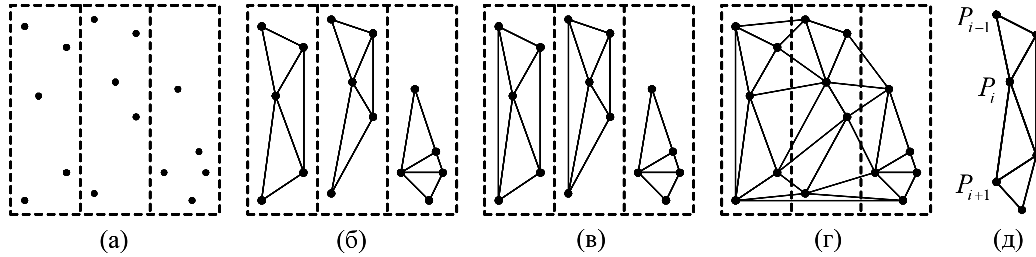

Рис. 29. Алгоритм выпуклого полосового слияния (а–г — шаги работы алгоритма, д — достраивание выпуклости)

Шаг 3а. Выполняется обход всех граничных узлов P_i частичных триангуляций. Анализируются соседние к P_i вдоль границы узлы P_{i-1} и P_{i+1}. Если $\angle P_{i-1}P_iP_{i+1} < 180°$, то в точке P_i нарушается условие выпуклости триангуляции и необходимо построить $\triangle P_{i-1}P_iP_{i+1}$, а дальнейший анализ граничных узлов надо продолжить с предыдущего узла P_{i-1} (рис. 29 д).

Шаг 3б. Далее необходимо последовательно склеить все столбцы друг с другом, используя алгоритм слияния из алгоритма триангуляции “Разделяй и властвуй”.

В целом трудоемкость алгоритма выпуклого полосового слияния составляет в среднем $O(N)$. Однако данный алгоритм делает много лишней работы, так как при построении выпуклой оболочки узкой полосы обычно получаются длинные узкие треугольники, которые почти всегда приходится перестраивать при слиянии. Этот недостаток исправляется в следующем алгоритме невыпуклого слияния.

3.3.2. Алгоритм невыпуклого полосового слияния. Пошаговая схема работы *алгоритма невыпуклого полосового слияния* схематично изображена на рис. 30 а–в.

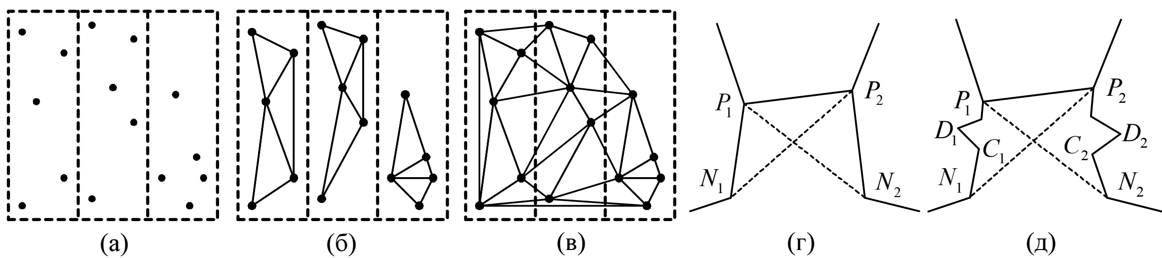

Рис. 30. Алгоритм невыпуклого полосового слияния (а–в — шаги работы алгоритма, г — слияние полос, д — локальное достраивание выпуклости)

Шаг 3. На вход алгоритма невыпуклого слияния подаются невыпуклые полосовые триангуляции. В ходе работы алгоритма перед очередным построением соединяющего ребра и, соответственно, треугольника производится достраивание выпуклости на некотором расстоянии от соединяющего ребра. Рассмотрим это подробнее. Пусть на очередном шаге слияния построен отрезок P_1P_2 (P_1 — на левой триангуляции, P_2 — на правой). Пусть N_1, N_2 — соседние точки по границам триангуляции (следующие ниже P_1, P_2). В предыдущем алгоритме на очередном шаге нужно было выбрать, какой треугольник строить — $\triangle P_1P_2N_1$ или $\triangle P_1P_2N_2$ (рис. 30 г). В алгоритме выпуклого слияния существенно используется то свойство, что граница первой триангуляции выше N_2 (начиная с P_1 и ниже) и граница второй триангуляции выше N_1 (начиная с P_2 и ниже) выпуклы. В алгоритме невыпуклого слияния необходимо проанализировать выпуклость границы и определить первую точку D_1 (D_2), начиная с P_1 (P_2), где нарушается выпуклость, и запомнить следующую за ней точку C_1 (C_2). Тогда перед анализом, какой треугольник слияния строить, проводится следующая проверка. Если C_1 (C_2) выше N_2 (N_1), то достраивается выпуклая оболочка на границе от C_1 до P_1 (от C_2 до P_2) и ищутся следующие точки D и C, если

они существуют (рис. 30 д).

При таком подходе удается заметно уменьшить число перестроений, а следовательно, и время работы алгоритма. В [8] показано, что алгоритм невыпуклого слияния работает примерно на 10 – 15 % быстрее, чем алгоритм выпуклого слияния.

4. Алгоритмы прямого построения. Во всех рассмотренных алгоритмах на разных этапах построения триангуляции могут быть построены треугольники, которые в дальнейшем будут перестроены в связи с невыполнением условия Делоне.

Основная идея *алгоритмов прямого построения* заключается в том, чтобы строить только такие треугольники, которые удовлетворяют условию Делоне в конечной триангуляции, а поэтому не должны перестраиваться [23].

4.1. Пошаговый алгоритм. *Пошаговый алгоритм* [23], известный также как *алгоритм прямого перебора* и *метод активных ребер*, концептуально похож на алгоритм слияния триангуляций “Удаляй-и-строй”, описанный выше.

В алгоритме вначале выбирается некоторая базовая линия AB, начиная от которой будут строиться все последующие треугольники (рис. 31). Базовая линия берется как один из отрезков многоугольника выпуклой оболочки всех исходных точек триангуляции.

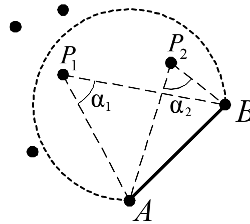

Рис. 31. Выбор очередной точки для включения в триангуляцию

Далее для базовой линии необходимо найти ближайшего соседа Делоне. Процесс поиска можно представить себе как рост “пузыря” от базовой линии, пока не встретится какой-нибудь узел. “Пузырь” — это окружность, проходящая через точки A и B, с центром на срединном перпендикуляре к базовой линии.

В пошаговом алгоритме для поиска соседа Делоне нужно выбрать среди всех точек P_i триангуляции такую, что $\angle AP_iB$ будет максимальным (например, на рис. 31 будет выбрана точка P_2). Найденный сосед Делоне соединяется отрезками с концами базовой линией и образует $\triangle AP_iB$. Новые ребра AP_i и BP_i построенного треугольника помечаются как новые базовые линии и процесс поиска треугольников продолжается.

Трудоемкость пошагового алгоритма составляет $O(N^2)$ в среднем и худшем случаях. Из-за столь большой трудоемкости на практике такой алгоритм практически не применяется.

4.2. Пошаговые алгоритмы с ускорением поиска соседей Делоне. Квадратичная сложность пошагового алгоритма обусловлена трудоемкой процедурой поиска соседа Делоне. В следующих двух алгоритмах предлагаются два варианта его ускорения.

4.2.1. Пошаговый алгоритм с бинарным деревом поиска. В *пошаговом алгоритме с бинарным деревом поиска* вначале все исходные точки триангуляции помещаются в любое бинарное дерево (например, k-D-дерево), позволяющее выполнять региональный поиск в заданном квадрате со сторонами, параллельными осям координат.

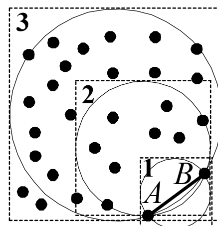

Рис. 32. Выбор очередной точки в клеточном алгоритме

Далее при выполнении поиска очередного соседа Делоне имитируется постепенный рост “пузыря”. Начальный “пузырь” определяется как окружность, диаметром которой является текущая базовая линия.

В дальнейшем, если внутри текущего "пузыря" не найдено никаких точек, то размер "пузыря" увеличивается, например, в два раза (на рис. 32 цифрами обозначены возможные этапы роста "пузыря"). Для определения точек, попавших в текущий пузырь, выполняется запрос к бинарному дереву на поиск всех точек внутри минимального квадрата, объемлющего текущий "пузырь". Среди найденных в квадрате точек отбрасываются все не попадающие в текущий "пузырь". Далее среди оставшихся точек прямым перебором находится искомый сосед Делоне.

Трудоемкость данного алгоритма с k-D-деревом в среднем на ряде распространенных распределений составляет $O(N \log N)$, а в худшем случае — $O(N^2)$.

4.2.2. Клеточный пошаговый алгоритм. В *клеточном пошаговом алгоритме* предлагается построить клеточное разбиение плоскости, а поиск очередного соседа Делоне вести, последовательно перебирая точки в близлежащих к базовой линии клетках [2]. Для этого на первом этапе все множество исходных точек разбивается вертикальными и горизонтальными равноотстоящими линиями на клетки и все точки помещаются в списки, соответствующие этим клеткам. Общее количество клеток должно быть равно $\theta(N)$.

Далее при выполнении поиска очередного соседа Делоне имитируется рост "пузыря" и проверяются все клетки с точками в порядке близости клеток к базовой линии (на рис. 33 цифрами помечен порядок обхода клеток). Трудоемкость данного алгоритма в среднем на ряде распространенных распределений составляет $O(N)$ [2] , а в худшем случае — $O(N^2)$.

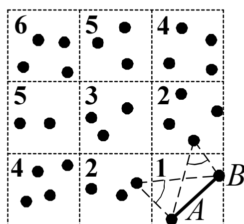

Рис. 33. Выбор очередной точки в клеточном алгоритме

5. Двухпроходные алгоритмы. При построении триангуляции Делоне итеративными алгоритмами и алгоритмами слияния для каждого строящегося треугольника после построения должно быть проверено условие Делоне. При этом приходится проводить проверки для трех пар, соответствующих трем соседним треугольникам к данному. Если проверка не выдержана, должны последовать перестроения треугольников и новая серия проверок. На практике довольно большую долю времени отнимают как раз проверки на условие Делоне и перестроения.

Для упрощения логики работы алгоритмов можно за первый проход построить некоторую триангуляцию, игнорируя выполнение условия Делоне, а после этого за второй проход проверить то, что получилось и провести нужные улучшающие перестроения для приведения триангуляции к триангуляции Делоне. Допустимость такой двухпроходной стратегии устанавливается в теореме 1.

5.1. Двухпроходные алгоритмы слияния. Наиболее удачно двухпроходная стратегия применима к алгоритмам слияния. В них приходится прикладывать довольно много алгоритмических усилий для того, чтобы обеспечить работу с "текущим треугольником" (например, при обходе триангуляции по границе, при слиянии, построении выпуклой оболочки), т.к. после того, как треугольник построен, он может тут же в результате неудачной проверки на условие Делоне исчезнуть, а на его месте могут появиться другие треугольники. Кроме того, в алгоритмах слияния сразу строятся достаточно много треугольников, которые в дальнейшем не перестраиваются. Общее количество выполняемых перестроений в алгоритме невыпуклого слияния составляет около 35 % от общего числа треугольников в конечной триангуляции, в алгоритме выпуклого слияния — 70 %, в алгоритме "Разделяй и властвуй" — 90 %, в то же время в простом итеративном алгоритме — 140 %. Именно поэтому наиболее хорошо для двухпроходной стратегии подходит алгоритм невыпуклого слияния. В алгоритмах "Разделяй и властвуй, выпуклого слияния и рекурсивном с разрезанием по диаметру на промежуточных этапах строится некоторое количество длинных узких треугольников, которые обычно затем перестраиваются.

На рис. 34 приведен пример применения двухпроходной стратегии к алгоритму выпуклого полосового слияния. В [8] показывается, что двухэтапные полосовые алгоритмы и "Разделяй и властвуй" работают в среднем на 10 – 20 % быстрее оригинальных алгоритмов. Это в частности объясняется некоторым упрощением логики работы алгоритмов.

5.2. Модифицированный иерархический алгоритм. Для итеративных алгоритмов двухпроход-

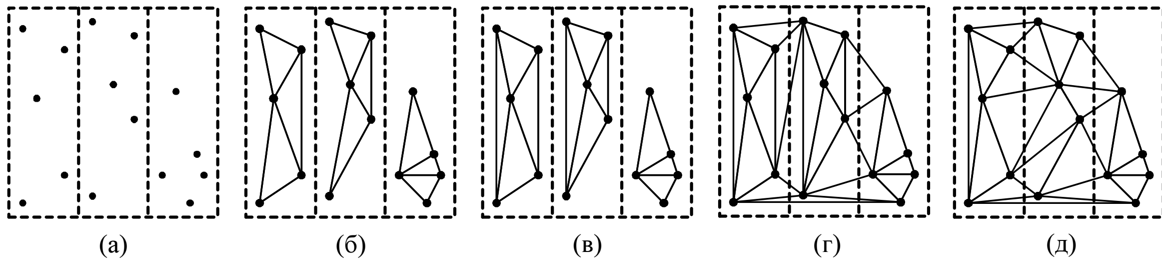

Рис. 34. Двухпроходный алгоритм выпуклого полосового слияния (а–г — построение некоторой триангуляции, д — полное перестроение)

ная стратегия, как правило, не годится, т.к. сразу же образуются узкие длинные треугольники, которые в дальнейшем делятся на другие еще меньшие и еще более узкие. Тем не менее, в [24] описывается *модифицированный иерархический алгоритм*, являющийся, по сути, обычным итеративным алгоритмом, выполняемым за два прохода.

Как и для оригинального простого итеративного алгоритма, трудоемкость данного составляет в худшем случае $O(N^2)$, а в среднем — $O(N^{3/2})$. На практике этот алгоритм работает значительно медленнее исходного. Тем не менее, он используется для построения специальных иерархических триангуляций, применяемых для работы с большими наборами исходных данных.

5.3. Линейный алгоритм. *Линейный алгоритм* (*алгоритм линейного заметания плоскости*) можно представить как частный случай двухпроходного алгоритма выпуклого слияния с одной полосой. В данном алгоритме вначале все исходные точки плоскости сортируются по вертикали (рис. 35 а). Затем, последовательно перебирая точки сверху вниз, соединяются в одну невыпуклую триангуляцию (рис. 35 б). Далее триангуляция достраивается до выпуклой (рис. 35 в). И в заключение выполняется полное перестроение триангуляции для выполнения условия Делоне (рис. 35 г).

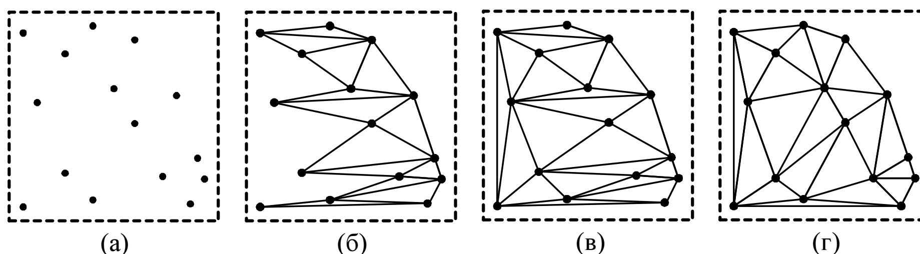

Рис. 35. Линейный алгоритм (а — исходные точки, б — невыпуклая триангуляция, в — достраивание до выпуклой, г — перестроение триангуляции)

Трудоемкость такого алгоритма составляет в среднем $O(N)$. Тем не менее, на практике этот алгоритм работает существенно медленнее полноценного алгоритма полосового слияния.

5.4. Веерный алгоритм. В *веерном алгоритме* триангуляции (*алгоритме радиального заметания плоскости*) вначале из исходных точек выбирается та, которая находится как можно ближе к центру масс всех точек. Далее для остальных точек вычисляется полярный угол относительно выбранной центральной точки и все точки сортируются по этому углу (рис. 36 б). Затем все точки соединяются ребрами с центральной точкой и соседними в отсортированном списке (рис. 36 в). Потом триангуляция достраивается до выпуклой (рис. 36 г). И в заключение выполняется полное перестроение триангуляции для выполнения условия Делоне (рис. 36 д).

Трудоемкость такого алгоритма составляет в среднем $O(N)$. Алгоритм работает примерно с той же скоростью, что и предыдущий линейный алгоритм.

5.5. Алгоритм рекурсивного расщепления. *Алгоритм рекурсивного расщепления* работает в два прохода [19]. Второй проход аналогичен всем двухпроходным алгоритмам триангуляции, а первый похож на рекурсивный алгоритм с разрезанием по диаметру, но разрезание производится не одним отрезком, а некоторой ломаной.

Перед началом работы алгоритма вычисляется выпуклая оболочка всех исходных точек. На каждом шаге рекурсии для заданного множества точек и их оболочки (не обязательно выпуклой) выполняется деление всех точек на две части. Для этого на оболочке находятся противоположные точки P_1 и P_2, делящие

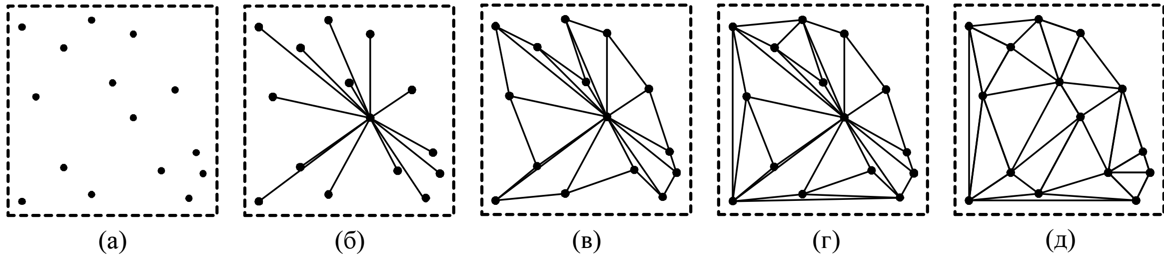

Рис. 36. Линейный алгоритм (а — исходные точки, б — круговая сортировка, в — невыпуклая триангуляция, г — достраивание до выпуклой, д — перестроение триангуляции)

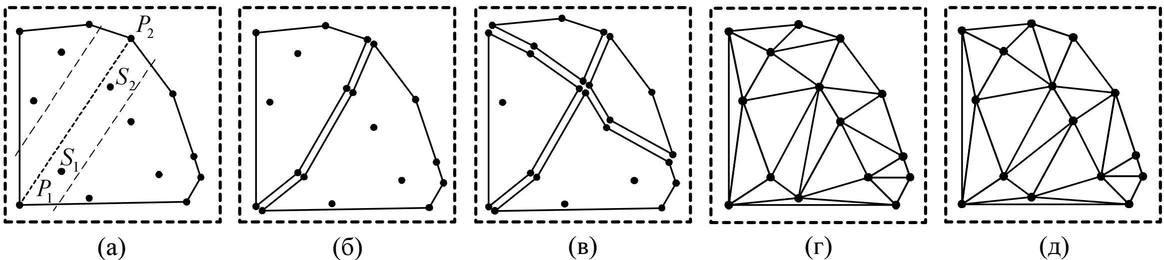

Рис. 37. Алгоритм рекурсивного расщепления (а — выбор направления и коридора, б–г — шаги расщепления, д — перестроение триангуляции)

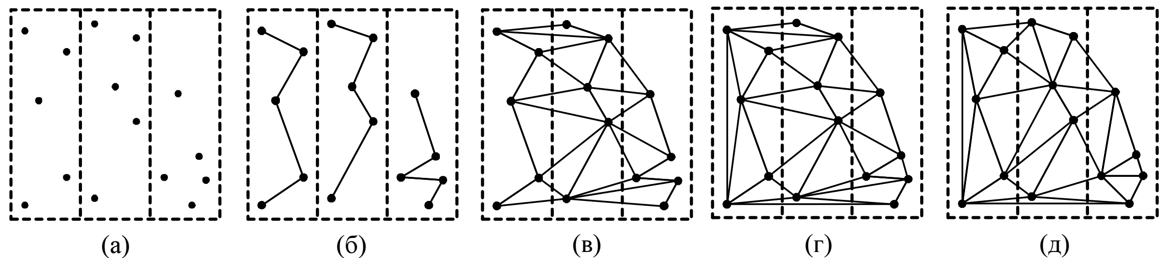

Рис. 38. Полосовой алгоритм (а — разбиение на полосы, б — построение в полосах ломаных, в — слияние полос, г — достраивание до выпуклости, д — перестроение триангуляции)

многоугольник оболочки примерно пополам (рис. 37 а). Затем находятся все точки S_i среди заданных, не попадающие на оболочку и находящиеся от прямой P_1P_2 не более чем на заданном расстоянии λ, т.е. попадающие в некоторый коридор расщепления (рис. 37 а). Затем точки P_1, S_i и P_2 последовательно соединяются в ломаную, которая разбивает исходное множество точек на две части. Разделяющая ломаная при этом попадает в оба множества (рис. 37 б). Кроме того, поскольку оболочка невыпуклая, то необходимо исключить случаи возможного пересечения построенной ломаной с оболочкой. Если полученные множества не являются треугольниками, то к ним опять рекурсивно применяется данный алгоритм (рис. 37 в). После построения триангуляций отдельных частей выполняется их соединение вдоль разделяющей ломаной.

Теоретически данный алгоритм рекурсивного расщепления имеет трудоемкость $O(N \log N)$ в среднем и худшем случаях [19]. Однако на практике процедура расщепления является сложной для реализации, медленной в работе и в целом алгоритм работает существенно медленнее любых двухпроходных алгоритмов слияния.

5.6. Ленточный алгоритм. Идея *ленточного алгоритма* предложена Ю. Л. Костюком. Некоторые элементы этого алгоритма похожи на алгоритм невыпуклого слияния.

На первом шаге все точки разбиваются на полосы (рис. 38 а). Затем точки сортируются внутри полос и последовательно соединяются в ломаные (рис. 38 б). Затем все полосы склеиваются между собой, используя процедуру слияния из алгоритма невыпуклого полосового слияния (рис. 38 в). После этого полученная триангуляция достраивается до выпуклой (рис. 38 г) и производится полное перестроение триангуляции для выполнения условия Делоне (рис. 38 д).

В данном алгоритме используется процедура слияния, соединяющая ломаные — ребра будущей три-

ангуляции. Поэтому наиболее удобно этот алгоритм реализуется на структуре данных "Узлы, ребра и треугольники", представляющей ребра в явном виде.

Трудоемкость данного алгоритма составляет в среднем $O(N)$.

Заключение. Подводя итоги данного обзора алгоритмов построения триангуляции Делоне, приведем сравнительную таблицу рассмотренных алгоритмов. В ней для каждого алгоритма приводятся трудоемкости в худшем и среднем случаях, время работы на 10 000 точек в относительных единицах и авторская экспертная оценка простоты реализации по пятибальной системе (чем больше звездочек, тем алгоритм лучше). Все приведенные оценки времени получены автором при реализации алгоритмов в одном стиле (на структуре данных "Узлы и треугольники"), в одной программной среде (Borland Pascal) и на одной аппаратной платформе (Intel). Некоторые из алгоритмов автором не реализованы, поэтому в таблице там стоят прочерки. Заметим, что оценки времени достаточно условны; видимо, они будут существенно отличаться в различных реализациях. Более подробные результаты сравнений отдельных алгоритмов приведены в [8].

Название алгоритма	Трудоемкость в худшем случае	Трудоемкость в среднем случае	Время работы на 10 000 точек	Простота программной реализации
Итеративные алгоритмы				
Простой итеративный алгоритм	$O(N^2)$	$O(N^{3/2})$	5,80	⋆⋆⋆⋆⋆
Итеративный алгоритм "Удаляй и строй"	$O(N^2)$	$O(N^{3/2})$	8,42	⋆⋆
Алгоритм с индексированием поиска R-деревом	$O(N^2)$	$O(N\log N)$	9,23	⋆⋆⋆
Алгоритм с индексированием поиска k-D-деревом	$O(N^2)$	$O(N\log N)$	7,61	⋆⋆⋆
Алгоритм с индексированием поиска квадродеревом	$O(N^2)$	$O(N\log N)$	7,14	⋆⋆⋆
Алгоритм статического кэширования	$O(N^2)$	$O(N^{9/8})$	1,68	⋆⋆⋆⋆⋆
Алгоритм динамического кэширования	$O(N^2)$	$O(N)$	1,49	⋆⋆⋆⋆⋆
Алгоритм с полосовым разбиением точек	$O(N^2)$	$O(N)$	3,60	⋆⋆⋆⋆⋆
Алгоритм с квадратным разбиением точек	$O(N^2)$	$O(N)$	2,61	⋆⋆⋆⋆⋆
Алгоритм послойного сгущения	$O(N^2)$	$O(N)$	1,93	⋆⋆⋆⋆
Алгоритм с сортировкой точек вдоль фрактальной кривой	$O(N^2)$	$O(N)$	5,01	⋆⋆⋆⋆
Алгоритм с сортировкой точек по Z-коду	$O(N^2)$	$O(N)$	5,31	⋆⋆⋆⋆⋆
Алгоритмы слияния				
Алгоритм "Разделяй и властвуй"	$O(N\log N)$	$O(N\log N)$	3,14	⋆⋆⋆
Рекурсивный алгоритм с разрезанием по диаметру	$O(N\log N)$	$O(N\log N)$	4,57	⋆⋆
Алгоритм выпуклого полосового слияния	$O(N^2)$	$O(N)$	2,79	⋆⋆⋆
Алгоритм невыпуклого полосового слияния	$O(N^2)$	$O(N)$	2,54	⋆⋆⋆
Алгоритмы прямого построения				
Пошаговый алгоритм	$O(N^2)$	$O(N^2)$	–	⋆⋆
Пошаговый алгоритм с бинарным деревом поиска	$O(N^2)$	$O(N\log N)$	–	⋆⋆
Пошаговый клеточный алгоритм	$O(N^2)$	$O(N)$	–	⋆⋆
Двухпроходные алгоритмы				
Двухпроходный алгоритм "Разделяй и властвуй"	$O(N\log N)$	$O(N\log N)$	2,79	⋆⋆⋆⋆
Двухпроходный алгоритм с разрезанием по диаметру	$O(N\log N)$	$O(N\log N)$	4,13	⋆⋆⋆
Двухпроходный алгоритм выпуклого полосового слияния	$O(N^2)$	$O(N)$	2,56	⋆⋆⋆⋆
Двухпроходный алгоритм невыпуклого полосового слияния	$O(N^2)$	$O(N)$	2,24	⋆⋆⋆⋆
Модифицированный иерархический алгоритм	$O(N^2)$	$O(N)$	15.42	⋆⋆⋆⋆⋆
Алгоритм линейного заметания	$O(N^2)$	$O(N)$	4,36	⋆⋆⋆⋆⋆
Веерный алгоритм	$O(N^2)$	$O(N)$	4,18	⋆⋆⋆⋆⋆
Алгоритм рекурсивного расщепления	$O(N\log N)$	$O(N\log N)$	–	⋆
Ленточный алгоритм	$O(N^2)$	$O(N)$	2,60	⋆⋆⋆⋆⋆

В целом из всего множества представленных алгоритмов по опыту автора лучше всего себя зарекомендовал *алгоритм динамического кэширования*. Примерно так же хорошо работает *алгоритм послойного сгущения*. Что немаловажно, оба эти алгоритма легко программируются на любых структурах данных. Из других хороших алгоритмов следует отметить *двухпроходный алгоритм невыпуклого полосового слияния* и *ленточный алгоритм*, но они несколько сложнее в реализации.

СПИСОК ЛИТЕРАТУРЫ

1. *Делоне Б.Н.* О пустоте сферы // Изв. АН СССР, ОМЕН. 1934, 4. 793–800.
2. *Ильман В.М.* Алгоритмы триангуляции плоских областей по нерегулярным сетям точек // Алгоритмы и программы. Вып. 10 (88). М., 1985. 3–35.
3. *Ильман В.М.* Экстремальные свойства триангуляции Делоне // Алгоритмы и программы. Вып. 10 (88). М., 1985. 57–66.
4. *Костюк Ю.Л., Грибель В.А.* Размещение и отображение на карте точечных объектов // Методы и средства обработки сложной графической информации (тезисы докладов всесоюзной конференции). Часть II. Горький, 1988. 60–61.
5. *Кроновер Р.М.* Фракталы и хаос в динамических системах. Основы теории / Пер. с англ. М.: Постмаркет, 2000.
6. *Ласло М.* Вычислительная геометрия и компьютерная графика на C++ / Пер. с англ. М.: БИНОМ, 1997.
7. *Препарата Ф., Шеймос М.* Вычислительная геометрия: Введение / Пер. с англ. М.: Мир, 1989.
8. *Скворцов А.В., Костюк Ю.Л.* Эффективные алгоритмы построения триангуляции Делоне // Геоинформатика. Теория и практика. Вып. 1. Томск: Изд-во Томского ун-та, 1998. 22–47.
9. *Фукс А.Л.* Изображение изолиний и разрезов поверхности, заданной нерегулярной системой отсчетов // Программирование. 1986. 4. 87–91.
10. *Фукс А.Л.* Предварительная обработка набора точек при построении триангуляции Делоне // Геоинформатика. Теория и практика. Вып. 1. Томск: Изд-во Томского ун-та, 1998. 48–60.
11. *Bjørke J.T.* Quadtrees and triangulation in digital elevation models // International Archives of Photogrammetry and Remote Sensing, International Society for Photogrammetry and Remote Sensing, Committee of the 16th International Congress of ISPRS, Commission IV. Part B4. 1988. **27**. 38–44.
12. *Guibas L., Stolfi J.* Primitives for the manipulation of general subdivisions and the computation of Voronoi diagrams // ACM Transactions on Graphics. 1985. **4**, N 2. 74–123.
13. *Guttmann A., Stonebraker M.* Using a relational database management system for computer aided design data // IEEE Database Engineering. 1982. **5**, N 2. 21–28.
14. *Heller M.* Triangulation algorithms for adaptive terrain modeling // Proc. of the 4th Intern. Symposium on Spatial Data Handling. 1990. 163–174.
15. *Lawson C.* Software for C^1 surface interpolation // Mathematical Software III. NY: Academic Press, 1977. 161–194.
16. *Lawson C.* Transforming triangulations // Discrete Mathematics. 1972. 3. 365–372.
17. *Lee D.* Proximity and reachability in the plane. Techn. Report R-831. Coordinated Sci. Lab., Univ. of Illinois at Urbana. Urbana, 1978.
18. *Lee D., Schachter B.* Two algorithms for constructing a Delaunay triangulation // Int. Jour. Comp. and Inf. Sc. 1980. **9**, N 3. 219–242.
19. *Lewis B., Robinson J.* Triangulation of planar regions with applications // The Computer Journal. 1978. **21**, N 4. 324–332.
20. *Lingas A.* The Greedy and Delaunay triangulations are not bad ... // Lect. Notes Comp. Sc. 1983. **158**. 270–284.
21. *Lloyd E.* On triangulation of a set of points in the plain. MIT Lab. Comp. Sc. Tech. Memo. N 88. Boston, 1977.
22. *Manacher G., Zobrist A.* Neither the Greedy nor the Delaunay triangulation of planar point set approximates the optimal triangulation // Inf. Proc. Let. 1977. **9**, N 1. 31–34.
23. *McCullagh M.J., Ross C.G.* Delaunay triangulation of a random data set for isarithmic mapping // The Cartographic Journal. 1980. **17**, N 2. 93–99.
24. *Midtbø T.* Spatial modeling by Delaunay networks of two and three dimensions. Dr. Ing. thesis. Department of Surveying and Mapping, Norwegian Institute of Technology, University of Trondheim. Trondheim, 1993.
25. *Shapiro M.* A note on Lee and Schachter's algorithm for Delaunay triangulation // Inter. Jour. of Comp. and Inf. Sciences. 1981. **10**, N 6. 413–418.
26. *Sibson R.* Locally equiangular triangulations // The Computer Journal. 1978. **21**, N 3. 243–245.
27. *Watson D.F.* Computing the n-dimensional Delaunay tessellation with application to Voronoi polytopes // The Computer Journal. 1981. **24**, N 2. 167–172.

Поступила в редакцию
19.01.2002