А.В. Скворцов, Н.С. Мирза

Алгоритмы построения и анализа триангуляции

Томский государственный университет

Факультет информатики

А.В. Скворцов, Н.С. Мирза

АЛГОРИТМЫ ПОСТРОЕНИЯ И АНАЛИЗА ТРИАНГУЛЯЦИИ

ИЗДАТЕЛЬСТВО ТОМСКОГО УНИВЕРСИТЕТА

2006

УДК 681.3
ББК 22.19
С 42

Скворцов А.В., Мирза Н.С.

С 42 Алгоритмы построения и анализа триангуляции. – Томск: Изд-во Том. ун-та, 2006. – 168 с.

ISBN 5-7511-2028-5

В книге рассматриваются различные виды триангуляций: триангуляция Делоне, триангуляция Делоне с ограничениями, оптимальная триангуляция. Приводятся различные варианты структур данных для представления триангуляции, разные способы проверки условия Делоне, 29 алгоритмов построения триангуляции Делоне, алгоритмы построения триангуляции Делоне с ограничениями и приближенные алгоритмы построения оптимальной триангуляции.

Рассматривается применение триангуляции Делоне с ограничениями для решения задач пространственного анализа на плоскости (оверлеи, буферные зоны, зоны близости) и моделирования рельефа (построение изолиний, изоконтуров, зон видимости, расчёт объёмов земляных работ). Описывается структура триангуляции переменного разрешения, используемая для моделирования рельефа, рассматриваются алгоритмы её построения.

Рекомендуется специалистам, занимающимся разработками в области ГИС и САПР. Может быть использована студентами, изучающими машинную графику, вычислительную геометрию и геоинформатику.

УДК 681.3
ББК 22.19

ISBN 5-7511-2028-5

Оглавление

Предисловие

Задача построения триангуляции 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.

В гл. 1 даются определения триангуляции, рассматриваются основные её свойства, приводятся основные используемые структуры данных, а также 4 способа проверки условия Делоне. В гл. 2–5 рассматриваются 4 группы алгоритмов построения триангуляции Делоне; всего 29 алгоритмов. Даётся классификация алгоритмов, приводятся их трудоёмкости, а также общие оценки алгоритмов.

В гл. 6 рассматривается обобщение триангуляции Делоне – триангуляция Делоне с ограничениями, используемая для решения широкого круга задач. В гл. 7 приводятся точные и приближённые алгоритмы построения оптимальной триангуляции.

В гл. 8 рассматриваются вопросы практической реализации алгоритмов триангуляции, приводится модифицированный алгоритм вставки структурных отрезков.

В гл. 9 описывается применение триангуляции для решения задач пространственного анализа, часто возникающих в ГИС и САПР.

В гл. 10 описываются триангуляционные структуры для моделирования поверхностей, приводятся алгоритмы их построения. В гл. 11 рассматривается задача разбиения триангуляции на полосы треугольников, возникающая при визуализации моделей поверхностей; приводятся алгоритмы их построения. В гл. 12 рассматриваются сверхбольшие триангуляционные модели данных, приводятся алгоритмы их построения и анализа.

В гл. 13 описывается применение триангуляции для моделирования поверхностей, а также ряд алгоритмов анализа триангуляционных моделей (упрощение триангуляции, построение изолиний, вычисление объёмов земляных работ и зон видимости).

Рассматриваемые в книге применения триангуляции с ограничениями реализованы авторами в рамках геоинформационной системы IndorGIS 6.0 и системы автоматизированного проектирования объектов строительства IndorCAD 6.0, широко используемых в России, Казахстане и Украине.

Все отзывы и пожелания авторы примут с благодарностью и просят направлять их по адресу: 634050, пр. Ленина, 36, Томский государственный университет, факультет информатики; или по электронной почте: skv@csd.tsu.ru, skv@indorsoft.ru, mirza@indorsoft.ru.

Глава 1. Определения и структуры данных

1.1. Определения

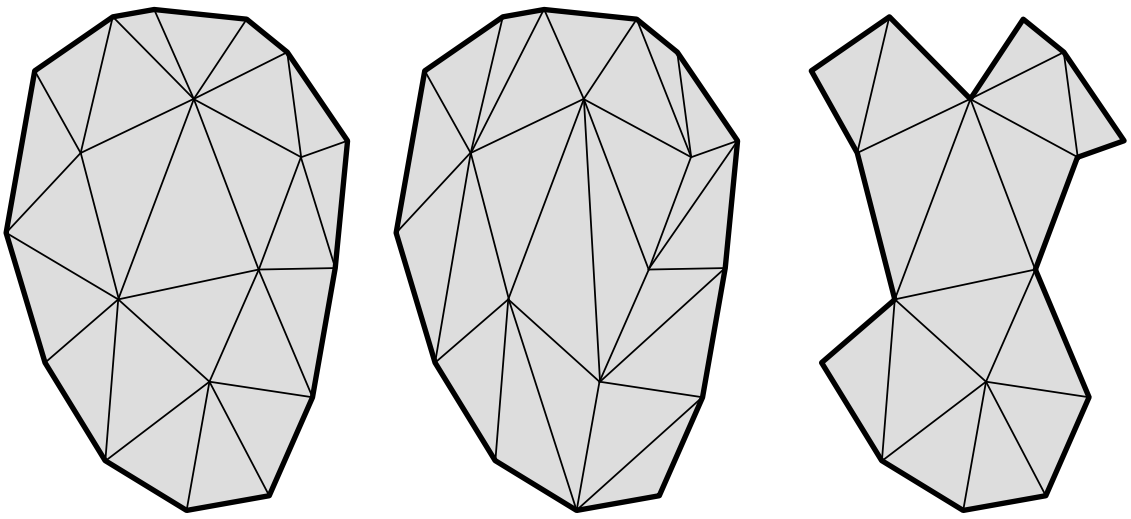
Впервые задача построения триангуляции Делоне была поставлена в 1934 г. в работе советского математика Б.Н. Делоне [1]. Трудоёмкость этой задачи составляет $O(N \log N)$. Существуют алгоритмы, достигающие этой оценки в среднем и худшем случаях. Кроме того, известны алгоритмы, позволяющие в ряде случаев достичь в среднем $O(N)$.

Для дальнейшего обсуждения введём несколько определений [1,12]:

Определение 1. *Триангуляцией* (англ. *triangulation*) называется планарный граф, все внутренние области которого являются треугольниками (рис. 1).

Соответствующие элементы триангуляции обычно называют *узлами* (англ. *node*), *ребрами* (англ. *rib*) и *треугольниками* (англ. *triangle*). Однако эта терминология не общепринята. Достаточно часто узлы называют *точками* (англ. *point*) или *вершинами* (англ. *vertex*), ребра – *дугами* (англ. *arc*), а треугольники – *гранями* (англ. *side, facet, edge*).

Определение 2. *Выпуклой триангуляцией* называется такая триангуляция, для которой минимальный многоугольник, охватывающий все треугольники, будет выпуклым (рис. 1,*а*,*б*). Триангуляция, не являющаяся выпуклой, называется *невыпуклой* (рис. 1,*в*).



(*а*) (*б*) (*в*)

Рис. 1. Примеры триангуляций: *а*, *б* – выпуклые, *в* – невыпуклая

Определение 3. Задачей построения триангуляции по заданному набору двумерных точек называется задача соединения заданных точек непересекающимися отрезками так, чтобы образовалась триангуляция.

Задача построения триангуляции по исходному набору точек является неоднозначной. В работе [64] дана верхняя оценка числа различных триангуляций, которые можно построить на множестве из N точек на плоскости: $59^N O(N^{-6})$ (более точно эту оценку можно записать как $59^v 7^b / C_{v+b+6}^b$, где v и b – число внутренних и граничных узлов триангуляции). Именно поэтому из-за неоднозначности задачи возникает вопрос, какая из двух различных триангуляций лучше?

Определение 4. Триангуляция называется *оптимальной*, если сумма длин всех рёбер минимальна среди всех возможных триангуляций, построенных на тех же исходных точках.

Давно предполагалось, что задача построения такой триангуляции, видимо, является NP-трудной [55,57]. И только недавно (в январе 2006 г.) в работе [60] было получено точное доказательство данного утверждения.

Поэтому для большинства реальных задач существующие алгоритмы построения оптимальной триангуляции неприемлемы ввиду слишком высокой трудоёмкости. При необходимости на практике применяют приближённые алгоритмы, рассматриваемые ниже в гл. 7.

Одним из первых был предложен следующий алгоритм построения триангуляции.

Жадный алгоритм построения триангуляции

Шаг 1. Генерируется список всех возможных отрезков, соединяющих пары исходных точек, и он сортируется по длинам отрезков.

Шаг 2. Начиная с самого короткого, последовательно выполняется вставка отрезков в триангуляцию. Если отрезок не пересекается с другими ранее вставленными отрезками, то он вставляется, иначе он отбрасывается. *Конец алгоритма.*

Заметим, что если все возможные отрезки имеют разную длину, то результат работы этого алгоритма однозначен, иначе он зависит от порядка вставки отрезков одинаковой длины.

Определение 5. Триангуляция называется *жадной*, если она построена жадным алгоритмом.

Трудоёмкость работы жадного алгоритма при некоторых его улучшениях составляет $O(N^2 \log N)$ [37]. В связи со столь большой трудоёмкостью на практике он почти не применяется.

Кроме оптимальной и жадной триангуляций, также широко известна триангуляция Делоне, обладающая рядом практически важных свойств [1,4,55,57].

Определение 6. Говорят, что триангуляция удовлетворяет *условию Делоне*, если внутрь окружности, описанной вокруг любого построенного треугольника, не попадает ни одна из заданных точек триангуляции.

Определение 7. Триангуляция называется *триангуляцией Делоне*, если она является выпуклой и удовлетворяет условию Делоне (рис. 2).

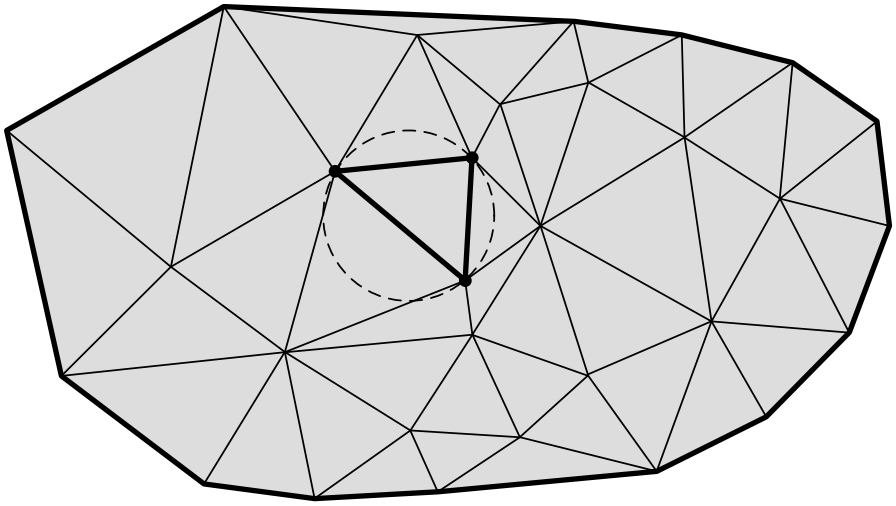


Рис. 2. Пример триангуляции Делоне с демонстрацией условия Делоне

Определение 8. Говорят, что пара соседних треугольников триангуляции удовлетворяет *условию Делоне*, если этому условию удовлетворяет триангуляция, составленная только из этих двух треугольников.

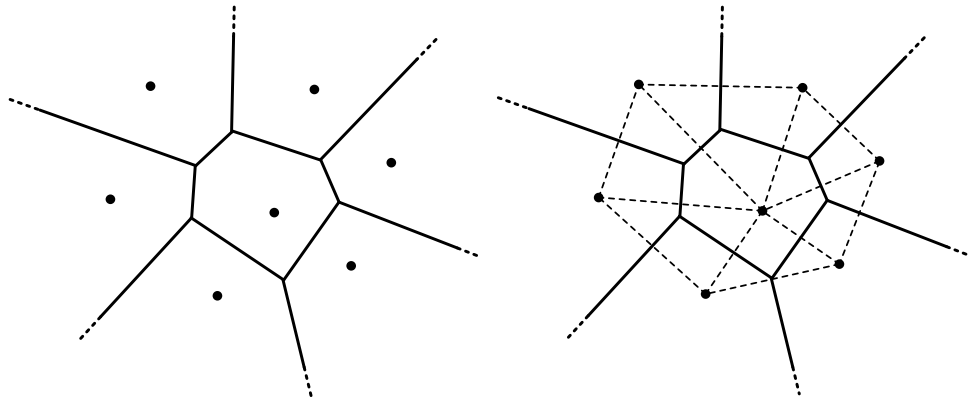
Определение 9. Говорят, что треугольник триангуляции удовлетворяет *условию Делоне*, если этому условию удовлетворяет триангуляция, составленная только из этого треугольника и трёх его соседей (если они существуют).

Триангуляция Делоне впервые появилась в научном мире как граф, двойственный диаграмме Вороного – одной из базовых структур вычислительной геометрии.

Определение 10. Для заданной точки $P_i \in \{P_1,...,P_N\}$ на плоскости *многоугольником (ячейкой) Вороного* называется геометрическое место точек на плоскости, которые находятся к P_i ближе, чем к любой другой заданной точке P_j, $j \neq i$ [71].

Совокупность многоугольников Вороного образует разбиение плоскости, представляющее векторную сеть.

Определение 11. *Диаграммой Вороного* заданного множества точек $\{P_1,...,P_N\}$ называется совокупность всех многоугольников Вороного этих точек (рис. 3,*а*).



(*а*) (*б*)

Рис. 3. Диаграммы Вороного: *а* – пример диаграммы;
б – двойственная диаграмме триангуляция Делоне

Диаграммы Вороного также иногда называют *разбиением Тиссена* и *ячейками Дирихле*.

Одним из главных свойств диаграммы Вороного является её двойственность триангуляции Делоне [1]. А именно, соединив отрезками те исходные точки, чьи многоугольники Вороного соприкасаются хотя бы углами, мы получим триангуляцию Делоне (рис. 3,*б*).

Многие алгоритмы построения триангуляции Делоне используют следующую теорему [4,46]:

Теорема 1. Триангуляцию Делоне можно получить из любой другой триангуляции по той же системе точек, последовательно перестраивая пары соседних треугольников ΔABC и ΔBCD, не удовлетворяющих условию Делоне, в пары треугольников ΔABD и ΔACD (рис. 4). Такая операция перестроения также часто называется *флипом* или *переброской ребра*.

Данная теорема позволяет строить триангуляцию Делоне последовательно, построив вначале некоторую триангуляцию, а потом последовательно улучшая её до выполнения условия Делоне.

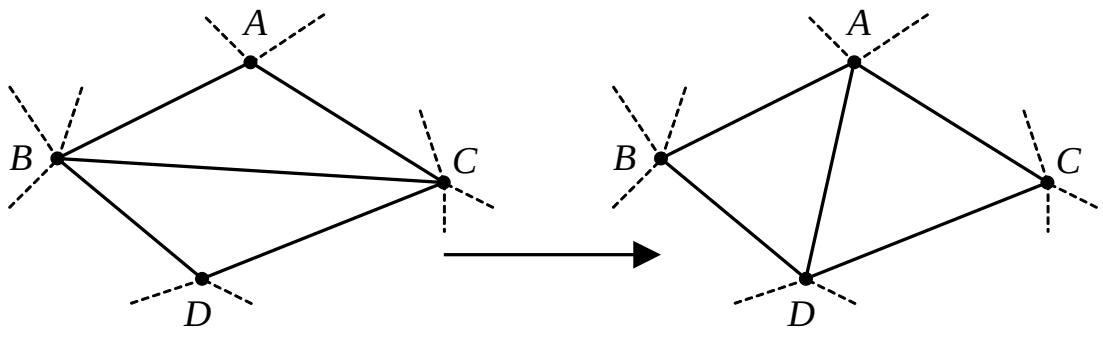


Рис. 4. Перестроение треугольников,
не удовлетворяющих условию Делоне

При проверке условия Делоне для пар соседних треугольников можно использовать непосредственно определение 6, но иногда применяют другие способы, основанные на следующих теоремах [4,45,47,66]:

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 в парах смежных треугольников, можно получить триангуляцию Делоне.

1.2. Количественные характеристики триангуляции

При разработке различных алгоритмов, а также при оценке их характеристик часто возникает необходимость оценки количества тех или иных элементов триангуляции. В соответствии с теоремой Эйлера о планарных графах мы имеем $N - R + F = 2$, где N – количество узлов в графе, R – рёбер, а F – граней (включая общее число внутренних областей в планарном графе и одну внешнюю бесконечную грань). Отсюда мы получаем для триангуляции:

$$N - R + T = 1,$$

где N – количество узлов в триангуляции, R – рёбер, а T – треугольников.

Для рёбер и треугольников также можно записать следующие зависимости от числа узлов:

$$T = 2 \cdot N - C - 2;$$
$$R = 3 \cdot N - C - 3,$$

где C – количество узлов (рёбер) на внешней границе триангуляции.

Так как это число C зависит от конкретной конфигурации узлов, то также возникает задача его оценки. В настоящее время известны оценки числа внешних рёбер в выпуклой триангуляции для некоторых распространённых законов распределения. Это число совпадает с количеством точек на выпуклой оболочке множества из N точек. Приведем некоторые из этих оценок.

Для равномерного распределения точек внутри любого заданного выпуклого многоугольника: $C = \theta(\log N)$. Для равномерного распределения точек внутри круга: $C = \theta(N^{1/3})$. Для нормального распределения точек на плоскости: $C = \theta(\log^{1/2} N)$.

Из этих оценок можно также получить среднее число рёбер, смежных с каждым узлом триангуляции: $\overline{R} = 2 \cdot R / N = 6 - (2 \cdot C - 6) / N$. Аналогичная оценка получается для среднего числа треугольников, смежных узлу: $\overline{T} = (2 \cdot R - C) / N = 6 - (3 \cdot C - 6) / N$. Учитывая, что в большинстве случаев $C = o(N)$, то $\overline{R} \approx 6$ и $\overline{T} \approx 6$.

Приведенные выше оценки верны для всех видов триангуляций. Для триангуляции Делоне также известны оценки некоторых характеристик треугольников, в частности следующие.

Соотношением сторон треугольника называется отношение самой длинной стороны треугольника и длины проекции противолежащей вершины на эту сторону. Для триангуляции Делоне это соотношение в среднем составляет $2\pi/3 + 7/(2\pi) \approx 3{,}20848$ [24].

Ожидаемая максимальная длина ребра в триангуляции Делоне равна $R_{\max} = \theta(\log^{1/2} N)$ [24].

Ожидаемое максимальное число смежных рёбер в узлах в триангуляции Делоно равно $B_{\max} = \theta(\log N / \log\log N)$ [24].

Ожидаемый минимальный угол в треугольниках триангуляции Делоне равен $\alpha_{\min} = \theta(N^{-1/2})$, а максимальный $\alpha_{\max} = \pi - \theta(N^{-1/5})$ [24].

1.3. Структуры для представления триангуляции

Как показывает практика, выбор структуры для представления триангуляции оказывает существенное влияние на теоретическую трудоёмкость алгоритмов, а также на скорость конкретной реализации. Кроме того, выбор структуры может зависеть от цели дальнейшего использования триангуляции.

В триангуляции можно выделить 3 основных вида объектов: *узлы* (точки, вершины), *рёбра* (отрезки) и *треугольники*.

В работе многих существующих алгоритмов построения триангуляции Делоне и алгоритмов её анализа часто возникают следующие операции с объектами триангуляции:

1. *Треугольник → узлы*: получение для данного треугольника координат образующих его узлов.
2. *Треугольник → рёбра*: получение для данного треугольника списка образующих его рёбер.
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 → рёбра*: получение для данного узла списка смежных рёбер.
7. *Узел → треугольники*: получение для данного узла списка смежных треугольников.

В каких-то алгоритмах некоторые из этих операций могут не использоваться. В других же алгоритмах операции с рёбрами могут возникать не часто, поэтому рёбра могут представляться косвенно как одна из сторон некоторого треугольника.

Рассмотрим наиболее часто встречающиеся структуры.

1.3.1. Структура данных «Узлы с соседями»

В *структуре «Узлы с соседями»* для каждого узла триангуляции хранятся его координаты на плоскости и список номеров (или указателей) смежных (соседних, т.е. с которыми есть общие рёбра) узлов (рис. 5) [58]:

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

Порядок следования смежных узлов в списке обычно не важен, но в некоторых задачах иногда требуется, чтобы этот список узлов был отсортирован по часовой стрелке или против (см. рис. 5).

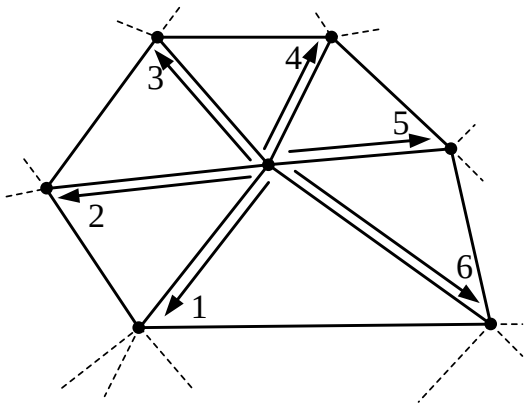


Рис. 5. Связи узлов структуры «Узлы с соседями»

По сути, список соседей определяет в неявном виде рёбра 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 6 (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 триангуляции, составляет $44 \cdot N$ байт.

1.3.2. Структура данных «Узлы и рёбра»

В *структуре «Узлы и рёбра»* в явном виде задаются узлы и рёбра. Треугольники в структуре отсутствуют. Для каждого ребра хранятся указатели на два концевых узла. Для треугольников хранятся указатели на три образующих треугольник ребра (рис. 6):

```
Узел = record
  X: число;      ← координата X
  Y: число;      ← координата Y
end;
Ребро = record
  Nodes: array [1..2] of Номер_узла;      ← список концевых узлов
end;
```

Данная структура часто применяется в случаях, когда требуется в явном виде представлять рёбра триангуляции, но нет необходимости работать с треугольникам. В частности, эта структура лучше всего подходит для построения жадной и оптимальной триангуляций.

Данная структура расходует достаточно мало памяти: при 8-байтовом представлении координат и 4-байтовых указателях получается примерно $40 \cdot N$ байт.

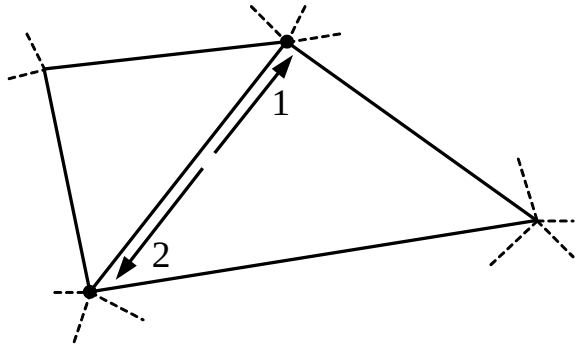


Рис. 6. Связи рёбер структуры «Узлы и рёбра»

1.3.3. Структура данных «Двойные рёбра»

В *структуре «Двойные рёбра»* основой триангуляции является список ориентированных рёбер. При этом каждое ребро входит в структуру триангуляции дважды, но направленными в противоположные стороны:

```
Узел = record
  X: число;      ← координата X
  Y: число;      ← координата Y
end;
Ребро = record
  Node: Номер_узла;       ← концевой узел ребра
  Next: Номер_ребра;      ← следующее по часовой стрелке в треугольнике справа
  Twin: Номер_ребра;      ← ребро-близнец, направленное в другую сторону
  Triangle: Номер_треугольника;      ← указатель на треугольник справа
end;
Треугольник = record      ← в записи нет обязательных полей
end;
```

Для каждого ребра хранятся следующие указатели (рис. 7) [41]:

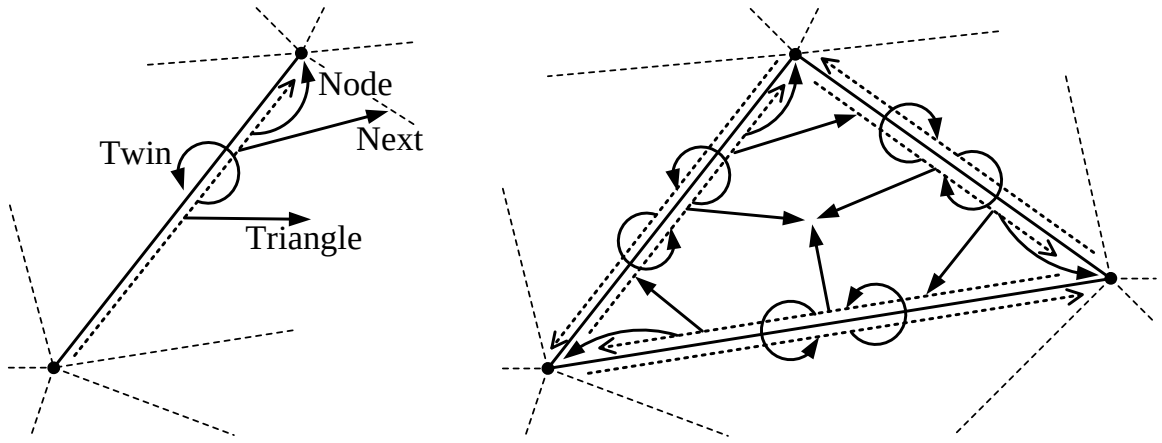
1) на концевой узел ребра;

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.



(*а*) (*б*)

Рис. 7. Связи рёбер (*а*) и неявное задание треугольников (*б*) в структуре «Двойные рёбра»

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).

1.3.4. Структура данных «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. 8) [18,48,65]:

```
Узел = record
  X: число;      ← координата X
  Y: число;      ← координата Y
end;
Треугольник = record
  Nodes: array [1..3] of Номер_узла;                ← образующие узлы
  Triangles: array [1..3] of Номер_треугольника;    ← соседние треугольники
end;
```

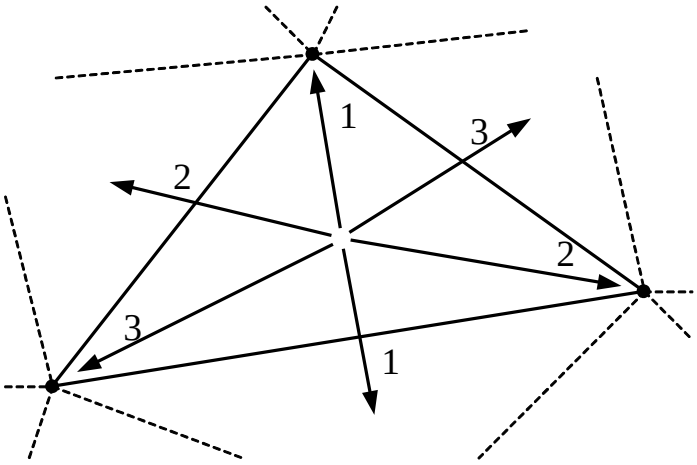


Рис. 8. Связи треугольников структуры «Узлы и треугольники»

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 (см. рис. 8). Рёбра в данной 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 данная структура триангуляции требует примерно $64 \cdot N$ байт.

Несмотря на то, что данная структура уступает «Узлам с соседями», она наиболее часто применяется на практике в силу своей относительной простоты и удобства программирования алгоритмов на её основе.

1.3.5. Структура данных «Узлы, рёбра и треугольники»

В *структуре «Узлы, рёбра и треугольники»* в явном виде задаются все объекты триангуляции: узлы, рёбра и треугольники. Для каждого ребра хранятся указатели на два концевых узла и два соседних треугольника. Для треугольников хранятся указатели на три образующих треугольник ребра (рис. 9) [25]:

```
Узел = record
  X: число;      ← координата X
  Y: число;      ← координата Y
end;
Ребро = record
  Nodes: array [1..2] of Номер_узла;                ← список концевых узлов
  Triangles: array [1..2] of Номер_треугольника;    ← соседние треугольники
end;
Треугольник = record
  Ribs: array [1..3] of Номер_ребра;     ← образующие рёбра
end;
```

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 (см. рис. 9).

Данная структура часто применяется на практике, особенно в задачах, где требуется в явном виде представлять рёбра триангуляции.

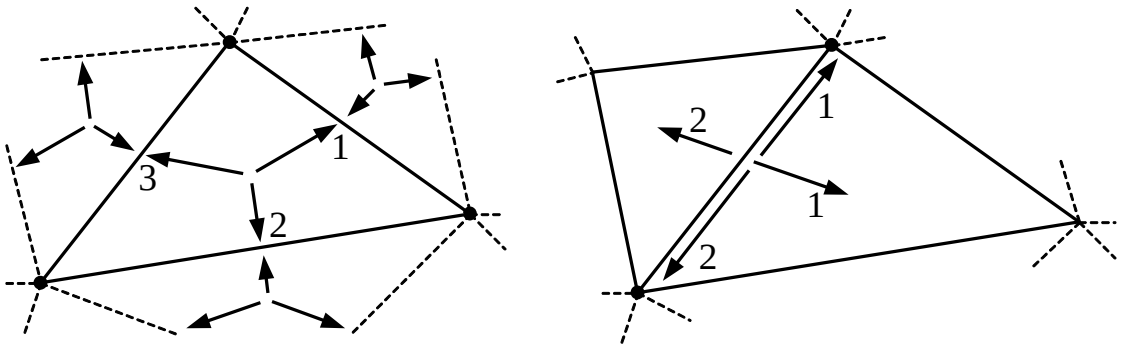


Рис. 9. Связи треугольников (слева) и рёбер (справа) структуры «Узлы, рёбра и треугольники»

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

1.3.6. Структура данных «Узлы, простые рёбра и треугольники»

В *структуре «Узлы, простые рёбра и треугольники»* в явном виде задаются все объекты триангуляции: узлы, рёбра и треугольники. Для каж-

дого ребра хранятся указатели на два концевых узла и два соседних треугольника. Для рёбер никакой специальной информации нет. Для треугольников хранятся указатели на образующих треугольник три узла и три ребра, а также указатели на три смежных треугольника (рис. 10):

```
Узел = record
  X: число;      ← координата X
  Y: число;      ← координата Y
end;
Ребро = record      ← в записи нет обязательных полей
end;
Треугольник = record
  Nodes: array [1..3] of Номер_узла;                  ← образующие узлы
  Triangles: array [1..3] of Номер_треугольника;      ← соседние треугольники
  Ribs: array [1..3] of Номер_ребра;                  ← образующие рёбра
end;
```

Данная структура часто применяется на практике, особенно в задачах, где требуется в явном виде представлять рёбра триангуляции.

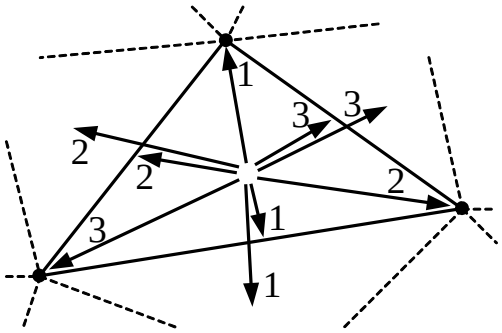


Рис. 10. Связи треугольников в структуре данных «Узлы, простые рёбра и треугольники»

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

В заключение этого раздела в табл. 1 представлены сводные характеристики приведенных структур данных, включая затраты по памяти и степень представления различных элементов триангуляции («–» – элемент отсутствует, «+» – присутствует, «±» – присутствует, но нет связей с другими элементами триангуляции).

В целом можно отметить, что структура «Узлы с соседями» менее удобна, чем остальные, так как не представляет в явном виде рёбра и треугольники. Среди остальных достаточно удобной в программировании является структура «Узлы и треугольники». Однако некоторые алгоритмы триангуляции требуют представления рёбер в явном виде, поэтому там можно порекомендовать структуру «Узлы, рёбра и треугольники».

Таблица 1. Основные характеристики структур данных

Название структуры данных	Память	Узлы	Рёбра	Треугольники
«Узлы с соседями»	$44 \cdot N$	+	–	–
«Узлы и рёбра»	$40 \cdot N$	±	+	–
«Двойные рёбра»	$64 \cdot N$	±	+	±
«Узлы и треугольники»	$64 \cdot N$	±	–	+
«Узлы, рёбра и треугольники»	$88 \cdot N$	±	+	+
«Узлы, простые рёбра и треугольники»	$88 \cdot N$	±	±	+

1.3.7. Преобразование структур данных

Задача смены структуры, в которой представлена триангуляция, может возникнуть, например, при построении жадной или оптимальной триангуляции. Алгоритмы их построения оперируют только с рёбрами и узлами, а поэтому они вынуждены использовать структуры данных типа «Узлы с соседями» (см. п. 1.3.1) или «Узлы и рёбра» (см. п. 1.3.2). С другой стороны, целью построения триангуляции может быть моделирование поверхности, для чего необходима структура данных, например «Узлы и треугольники» (см. п. 1.3.4). Именно поэтому и возникает задача перехода от одной структуры данных к другой.

Вначале рассмотрим достаточно простой алгоритм для перехода от структуры «Узлы и рёбра» к структуре «Узлы с соседями». Основной целью этого алгоритма является вычисление рёбер, смежных с узлами.

Алгоритм преобразования структуры данных «Узлы и рёбра» в структуру «Узлы с соседями»

Структуры данных. Исходные структуры представлены массивами `Nodes` и `Ribs`. В результате мы должны получить массив `NewNodes`. В работе алгоритма потребуется временный массив `R` длины *N* для подсчета числа рёбер, смежных с соответствующими узлами.

Шаг 1. Вычисляем количество смежных рёбер `R[i]`, входящих в каждый `i`-й узел триангуляции. Для этого вначале присваиваем ∀ `i`: `R[i]:=0`. Затем в цикле по `i` просматриваем все рёбра и для каждого ребра `Ribs[i]`, соединяющего узлы с номерами `a=Ribs[i].Nodes[1]` и `b=Ribs[i].Nodes[2]`, увеличиваем счетчики рёбер в узлах: `R[a]++`, `R[b]++`.

Шаг 2. Выделяем память для каждого узла структуры «Узлы с соседями», используя `R[i]` в качестве количества узлов в массиве `Nodes` узла. В результате получим новые записи `NewNodes[i]`. В `NewNodes[i]` поля с координатами `X`,`Y` копируем из соответствующих полей `Nodes[i]` исходной

структуры данных «Узлы и рёбра». Другие поля пока выставляем нулевыми: `NewNodes[i].Count:=0`, ∀ `j: NewNodes[i].Nodes[j]:=0`.

Шаг 3. В цикле по `i` просматриваем все рёбра и для каждого ребра `R[i]`, соединяющего узлы с номерами `a` и `b`, добавляем в узлы ссылки на смежный через это ребро узел и увеличиваем счётчики смежных узлов в новых узлах:

```
NewNodes[a].Nodes[NewNodes[a].Count]:=b; NewNodes[a].Count++;
NewNodes[b].Nodes[NewNodes[b].Count]:=b; NewNodes[b].Count++.
```

Конец алгоритма.

Трудоёмкость описанного алгоритма является линейной от числа узлов триангуляции.

Следующий алгоритм для перехода от структуры «Узлы и рёбра» к структуре «Узлы и треугольники» более сложен. В нём требуется создать взаимосвязанные структуры треугольников. Первые 3 шага этого алгоритма почти совпадают с предыдущим алгоритмом: в нём для каждого узла создаётся специальная временная структура данных.

Алгоритм преобразования структуры данных «Узлы и рёбра» в структуру «Узлы и треугольники».

Структуры данных. Исходные структуры представлены массивами `Nodes` и `Ribs`. В ходе работы алгоритма потребуется временный массив `R` длины *N* для подсчета числа смежных с узлами рёбер. Кроме того, понадобится временная модифицированная структура данных «Узлы с соседями» (расширенная списком смежных рёбер и треугольников, но без координат, так как мы их можем получить из исходного массива `Nodes`), которая будет представлена в массиве `TmpNodes`. Эта временная структура должна иметь следующий вид:

```
ВременныйУзел = record
  Count: целое;      ← количество смежных узлов
  Nodes: array [1..Count] of Номер_узла;          ← список смежных узлов
  Ribs: array [1..Count] of Номер_ребра;          ← список смежных рёбер
  Trns: array [1..Count] of Номер_треугольника;   ← смежные треугольники
end;
```

В этой структуре ребро `TmpNodes[i].Ribs[j]` должно соединяться с узлом `TmpNodes[i].Nodes[j]`, а треугольник `TmpNodes[i].Trns[j]` должен лежать между смежными к узлу рёбрами, определяемыми рёбрами `TmpNodes[i].Ribs[j]` и `TmpNodes[i].Ribs[j mod TmpNodes[i].Count+1]`.

Алгоритму понадобится ещё одна временная структура, которая вводит дополнительные поля для каждого ребра триангуляции и которая будет представлена для рёбер в массиве `TmpRibs`:

```
ВременноеРебро = record
  IndexInNode: array [1..2] of целое;      ← номера ребра в списках Ribs узлов ребра
  Trns: array [1..2] of Номер_треугольника;      ← смежные треугольники
end;
```

В этой структуре данных треугольник `TmpRibs[i].Trns[1]` должен лежать по правую сторону от вектора, образованного узлами `Ribs[i].Nodes[1]` и `Ribs[i].Nodes[2]` (рис. 11). Временный массив `IndexInNode` предназначен для быстрого определения следующего и предыдущего ребра в треугольнике, содержащего данное ребро.

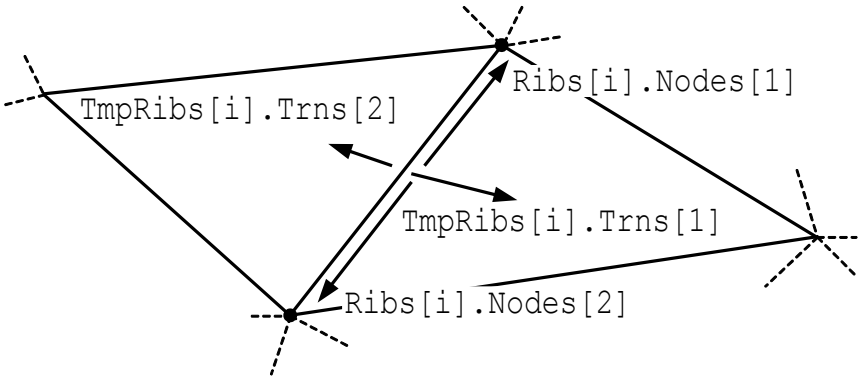


Рис. 11. Временные связи рёбер в алгоритме преобразования структуры данных «Узлы и рёбра» в структуру «Узлы и треугольники»

В результате работы алгоритма мы должны получить массив `NewNodes`.

Шаг 1. Вычисляем количество смежных рёбер `R[i]`, входящих в каждый `i`-й узел триангуляции. Для этого вначале присваиваем $\forall$ `i`: `R[i]:=0`. Затем в цикле по `i` просматриваем все рёбра и для каждого ребра `Ribs[i]`, соединяющего узлы с номерами `a=Ribs[i].Nodes[1]` и `b=Ribs[i].Nodes[2]`, увеличиваем счётчики рёбер в узлах: `R[a]++`, `R[b]++`.

Шаг 2. Создаем для каждого узла временные записи `TmpNodes[i]`, используя `R[i]` в качестве длин массивов `Nodes`, `Ribs` и `Trns`. Другие поля пока заполняем нулями и пустыми ссылками:

```
TmpNodes[i].Count:=0, ∀j: TmpNodes[i].Nodes[j]:=0,
∀j: TmpNodes[i].Ribs[j]:=0, ∀j: TmpNodes[i].Trns[j]:=0.
```

Шаг 3. Просматриваем все рёбра и для каждого ребра `R`, соединяющего узлы с номерами `a` и `b`, добавляем в узлы ссылки на `R` и узел, смежный через это ребро `R`, и увеличиваем счётчики смежных узлов в новых узлах:

```
TmpNodes[a].Nodes[TmpNodes[a].Count]:=b;
TmpNodes[a].Ribs[TmpNodes[a].Count]:=R; TmpNodes[a].Count++;
TmpNodes[b].Nodes[TmpNodes[b].Count]:=b;
TmpNodes[b].Ribs[TmpNodes[b].Count]:=R; TmpNodes[b].Count++.
```

Шаг 4. В каждом узле во временной структуре `TmpNodes` сортируем по часовой стрелке смежные узлы и рёбра (одновременно в массивах `TmpNodes[i].Nodes` и `TmpNodes[i].Ribs`). Для всех рёбер триангуляции массив `TmpRibs[i].Trns` заполняем пустыми ссылками (нулевые номера треугольников): $\forall$ `i,j: TmpRibs[i].Trns[j]:=0`.

Шаг 5. В цикле по `i` по каждому узлу делаем вложенный цикл по `j` по всем смежным рёбрам. Для каждого ребра определяем номер `k` узла в ребре и выставляем `TmpRibs[TmpNodes[i].Ribs[j]].IndexInNode[k]:=j`.

Шаг 6. В цикле по `i` по каждому ребру делаем вложенный цикл по `j` по двум смежным к ребру треугольникам и выполняем попытку создания нового треугольника из `TmpRibs[i].Trns[j]`, если этот треугольник ещё не создан (т.е. если `TmpRibs[i].Trns[j]=0`). Этот треугольник будет соединять концевые узлы текущего ребра (`Ribs[j].Nodes[j]`, `Ribs[j].Nodes[3-j]`) и ещё один узел, который можно определить с помощью списка смежных узлов в узле `Ribs[j].Nodes[j]`, используя `TmpRibs[i].IndexInNode[j]`. Кроме того, нужно определить 3 образующие треугольник ребра и выставить ссылки из этих рёбер в новый треугольник.

Шаг 7. Устанавливаем взаимные ссылки треугольников друг на друга. Для этого делаем цикл по `i` по каждому внутреннему ребру триангуляции (т.е. по всем `i`-м рёбрам, у которых $\forall$ `j: TmpRibs[i].Trns[j]`$\neq 0$) и для него устанавливаем друг на друга взаимные ссылки смежных треугольников `TmpRibs[i].Trns[0]` и `TmpRibs[i].Trns[1]`. *Конец алгоритма.*

При условии, что на шаге 4 используется сортировка с линейной трудоёмкостью (например, цифровая), то трудоёмкость данного алгоритма является линейной относительно общего числа узлов в триангуляции. Вообще, даже если использовать нелинейную сортировку, трудоёмкость алгоритма в среднем также будет линейной – $O(N)$. Это следует из того факта, что среднее число смежных с узлом рёбер в триангуляции является константой, не зависящей от N, а следовательно, и сортировка будет в среднем работать время $O(1)$.

В заключение отметим, что приведённый алгоритм можно легко модифицировать для получения и других структур данных, в которых в явном виде представлены треугольники.

1.4. Проверка условия Делоне

Одной 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.

1.4.1. 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 виде

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

или же как $(x^2+y^2)\cdot a - x\cdot b + y\cdot c - d = 0$, где

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

Тогда условие Делоне для любого заданного треугольника $\Delta\big((x_1, y_1), (x_2, y_2), (x_3, y_3)\big)$ будет выполняться только тогда, когда для любого узла (x_0, y_0) триангуляции будет $(a\cdot(x_0^2+y_0^2) - b\cdot x_0 + c\cdot y_0 - d)\cdot \operatorname{sgn} a \geq 0$, т.е. когда (x_0, y_0) не попадает внутрь окружности, описанной вокруг треугольника $\Delta\big((x_1, y_1), (x_2, y_2), (x_3, y_3)\big)$ [39]. Для упрощения вычислений можно заметить, что если тройка точек $(x_1, y_1), (x_2, y_2), (x_3, y_3)$ является правой (т.е. обход их в треугольнике выполняется по часовой стрелке), 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 операций сложения и вычитания.

Данную схему вычислений можно несколько усовершенствовать. Переместим систему координат на *x* и *y* по горизонтали и вертикали соответственно. Тогда уравнение окружности перепишется в виде

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

или, что эквивалентно, как

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

Проверка на основе этой модифицированной формулы требует уже только 15 операций умножения и возведения в квадрат, а также 14 операций сложения и вычитания.

1.4.2. Проверка с заранее вычисленной описанной окружностью

Предыдущий вариант проверки требует значительного количества арифметических операций. В большинстве алгоритмов триангуляции количество проверок условия многократно (в разных алгоритмах это число колеблется от 2 до 25 и больше) превышает общее число различных треугольников, присутствовавших в триангуляции на разных шагах её 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. Таким образом, центр (x_c, y_c) и радиус r окружности, описанной вокруг треугольника $\Delta\big((x_1,y_1),(x_2,y_2),(x_3,y_3)\big)$, 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 треугольника $\Delta\big((x_1,y_1),(x_2,y_2),(x_3,y_3)\big)$ будет выполняться только тогда, когда для любой другой точки (x_0,y_0) триангуляции $(x_0-x_c)^2+(y_0-y_c)^2 \ge r^2$.

Реализация такой процедуры проверки требует для каждого треугольника 36 операций умножения, возведения в квадрат и деления, а также 22 операций сложения и вычитания. На этапе непосредственного выполнения проверок требуется всего только 2 возведения в квадрат, 2 вычитания, 1 сложение и 1 сравнение.

Теперь заметим, что для каждого треугольника знать параметры описанной окружность и не обязательно. Проверка условия Делоне всегда выполняется для некоторой пары треугольников, а поэтому достаточно знать окружность только одного из этих треугольников. Тогда будем вычислять параметры описанной окружности лишь в том случае, если в паре анализируемых треугольников еще не вычислена ни одна окружность.

При таком подходе в среднем на 25–45% (в зависимости от используемого алгоритма триангуляции) уменьшается количество треугольников,

для которых необходимо вычислить описанные окружности. Таким образом, в среднем на один треугольник требуется 22–27 операций типа умножения и 13–17 операций типа сложения.

Если принять, что алгоритм триангуляции тратит в среднем по 5 проверок на каждый треугольник, то в среднем данный способ проверки требует около 6–7 операций типа умножения и 6 операций типа сложения. Если алгоритм тратит в среднем по 12 проверок на каждый треугольник, то соответственно по 4 той и другой операции. Точная же оценка среднего числа операций должна выполняться для конкретного алгоритма триангуляции и типичных видов исходных данных.

1.4.3. Проверка суммы противолежащих углов

В [39,45] показано, что условие Делоне для данного треугольника $\Delta\big((x_1,y_1),(x_2,y_2),(x_3,y_3)\big)$ будет выполняться только тогда, когда для любой другой точки (x_0,y_0) триангуляции $\alpha+\beta\le\pi$ (рис. 12). Это условие эквивалентно $\sin(\alpha+\beta)\ge 0$, т.е.

$$\sin\alpha\cdot\cos\beta+\cos\alpha\cdot\sin\beta\ge 0. \qquad (1)$$

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

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

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

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

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

Подставив эти значения в формулу (1) и сократив знаменатели дробей, получим следующую формулу проверки:

$$\big((x_0-x_1)(y_0-y_3)-(x_0-x_3)(y_0-y_1)\big)\cdot\big((x_2-x_3)(x_2-x_1)+(y_2-y_3)(y_2-y_1)\big)+$$

$$+\big((x_0-x_1)(x_0-x_3)+(y_0-y_1)(y_0-y_3)\big)\cdot\big((x_2-x_3)(y_2-y_1)-(x_2-x_1)(y_2-y_3)\big)\ge 0. \qquad (2)$$

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

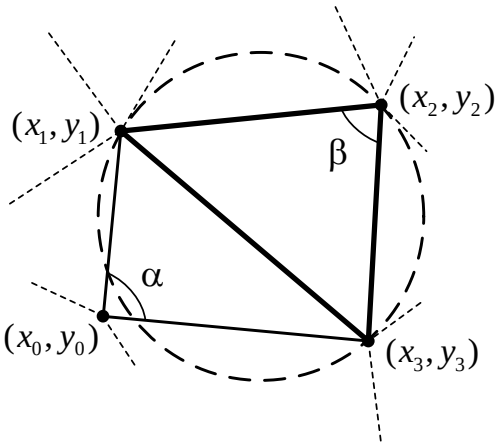


Рис. 12. Проверка условия Делоне по сумме противолежащих углов

1.4.4. Модифицированная проверка суммы противолежащих углов

В [67] предложено вычислять не сразу все скалярные и векторные произведения. Вначале нужно вычислить только часть из выражения (2), соответствующую $\cos\alpha$ и $\cos\beta$:

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

Тогда, если $s_\alpha < 0$ и $s_\beta < 0$, то $\alpha > 90^\circ$, $\beta > 90^\circ$, и поэтому условие Делоне не выполняется. Если $s_\alpha \geq 0$ и $s_\beta \geq 0$, то $\alpha \leq 90^\circ$, $\beta \leq 90^\circ$, и поэтому условие Делоне выполняется. Иначе требуются полные вычисления по формуле (2).

Такое усовершенствование позволяет в среднем на 20–40% (существенно зависит от алгоритма триангуляции) сократить количество выполняемых арифметических операций (примерно до 7 умножений, а также 9 сложений и вычитаний).

Дополнительным преимуществом этой проверки перед предыдущим способом является большая устойчивость к потере точности в промежуточных вычислениях с использованием вещественной арифметики с плавающей точкой [67].

В заключение этого раздела в табл. 2 даны сводные характеристики привёденных способов проверки условия Делоне.

В целом для применения можно порекомендовать модифицированную проверку суммы противолежащих углов, требующую минимального количества арифметических операций.

Таблица 2. Среднее количество выполняемых арифметических операций в различных способах проверки условия Делоне

Название способа проверки	Число ×	Число + и –
Через уравнение описанной окружности	15	14
С заранее вычисленной окружностью	~4…7	~4…6
Сумма противолежащих углов	10	13
Модифицированная сумма углов	~7	~9

Способ проверки с заранее вычисленной описанной окружностью следует применять только в алгоритмах, в которых треугольники перестраиваются редко. Это в первую очередь алгоритмы слияния, а также некоторые двухпроходные алгоритмы, рассматриваемые в следующих главах. Тем не менее следует заметить, что этот способ проверки требует дополнительных затрат памяти.

1.5. Алгоритмы построения триангуляции Делоне

В настоящее время известно значительное количество различных алгоритмов построения триангуляции Делоне. На рис. 13 приведена классификация только основных из них (жирным шрифтом выделены конкретные алгоритмы). Кроме этих алгоритмов, существуют и другие, менее известные, но они обладают заведомо худшими характеристиками.

В табл. 3 собраны основные характеристики всех этих алгоритмов. В колонке А представлена оценка трудоёмкости в худшем случае, в колонке Б – трудоёмкость в среднем, в колонке В – время работы на 10 000 точек в относительных единицах и в колонке Г – авторская экспертная оценка простоты реализации по 5-балльной системе (чем больше звездочек, тем алгоритм лучше). Все приведённые оценки времени получены нами после реализации алгоритмов в одном стиле (на структуре «Узлы и треугольники») и на одной программно-аппаратной платформе. Некоторые из алгоритмов не реализованы, поэтому в таблице там стоят прочерки. Заметим, что оценки времени достаточно условны и они, видимо, будут существенно отличаться в различных программных реализациях и на разных распределениях исходных точек. Более подробные результаты сравнений отдельных алгоритмов приведены в [17].

В целом из всего множества представленных алгоритмов по опыту лучше всего зарекомендовал себя *алгоритм динамического кэширования*. Примерно так же хорошо работает *алгоритм послойного сгущения*. Что немаловажно, оба эти алгоритма легко программируются на любых структурах данных.

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 треугольников R-деревом.**
  - 1.2.2. **Итеративный алгоритм с индексированием центров треугольников 2-D-деревом.**
  - 1.2.3. **Итеративный алгоритм с индексированием центров треугольников 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 изменённым 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 кривой, заполняющей плоскость.**
  - 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. Двухпроходные алгоритмы.
- 3.1. Двухпроходные алгоритмы слияния.
  - 3.1.1. **Алгоритм "Разделяй и властвуй".**
  - 3.1.2. **Рекурсивный алгоритм с разрезанием по диаметру.**
  - 3.1.3. **Алгоритм выпуклого полосового слияния.**
  - 3.1.4. **Алгоритм невыпуклого полосового слияния.**
- 3.2. **Модифицированный иерархический алгоритм.**
- 3.3. **Линейный алгоритм.**
- 3.4. **Веерный алгоритм.**
- 3.5. **Алгоритм рекурсивного расщепления.**
- 3.6. **Ленточный алгоритм.**

4. Пошаговые и прочие алгоритмы.
- 4.1. **Пошаговый алгоритм.**
- 4.2. Пошаговые алгоритмы с ускорением поиска соседей Делоне.
  - 4.2.1. **Пошаговый алгоритм с 2-D-деревом поиска.**
  - 4.2.2. **Клеточный пошаговый алгоритм.**
- 4.3. **Алгоритм построения через 3-мерные выпуклые оболочки.**

Рис. 13. Классификация алгоритмов построения триангуляции Делоне

Таблица 3. Общие характеристики алгоритмов триангуляции
(А, Б – трудоёмкости в худшем случае и в среднем, В – время работы, Г – простота)

Название алгоритма	А	Б	В	Г
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	★★★
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	★★★★

Продолжение табл. 3.

Название алгоритма	А	Б	В	Г
Двухпроходный алгоритм невыпуклого полосового слияния	$O(N^2)$	$O(N)$	2,24	★★★★
Модифицированный иерархический алгоритм	$O(N^2)$	$O(N^2)$	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	★★★★★
Пошаговые и прочие алгоритмы				
Пошаговый алгоритм	$O(N^2)$	$O(N^2)$	–	★★
Пошаговый алгоритм с k-D-деревом поиска	$O(N^2)$	$O(N\log N)$	–	★★
Пошаговый клеточный алгоритм	$O(N^2)$	$O(N)$	–	★★
Алгоритм построения через 3-мерные выпуклые оболочки	$O(N\log N)$	$O(N\log N)$	–	★★

Из других хороших алгоритмов следует отметить *двухпроходный алгоритм невыпуклого полосового слияния* и *ленточный алгоритм*, но они несколько сложнее в реализации.

На практике триангуляция строится для решения каких-либо прикладных задач. При этом почти всегда возникает задача локализации некоторой точки плоскости на триангуляции – поиска треугольника, в который она попадает. Только в результате работы алгоритма динамического кэширования создается структура кэша, которая и позволяет эффективно выполнять указанную локализацию. Во всех остальных алгоритмах такой структуры не создаётся и её необходимо создавать дополнительно.

В следующих четырёх главах все эти алгоритмы построения триангуляции Делоне будут рассмотрены подробно.

Глава 2. Итеративные алгоритмы построения триангуляции Делоне

Все итеративные алгоритмы имеют в своей основе очень простую идею последовательного добавления точек в частично построенную триангуляцию Делоне. Формально это выглядит так.

Итеративный алгоритм построения триангуляции Делоне.

Дано множество из N точек.

Шаг 1. На первых трёх исходных точках строим один треугольник (предполагается, что точки не лежат на одной прямой, иначе надо выбрать другие точки).

Шаг 2. В цикле по n для всех остальных точек выполняем шаги 3–5.

Шаг 3. Очередная n-я точка добавляется в уже построенную структуру триангуляции следующим образом. Вначале производится локализация точки, т.е. находится треугольник (построенный ранее), в который попадает очередная точка. Либо, если точка не попадает внутрь триангуляции, находится треугольник на границе триангуляции, ближайший к очередной точке.

Шаг 4. 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.

Шаг 5. Проводятся локальные проверки вновь полученных треугольников на соответствие условию Делоне и выполняются необходимые перестроения. *Конец алгоритма.*

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

При построении новых треугольников возможны две ситуации, когда добавляемая точка попадает либо внутрь триангуляции, либо вне её. В первом случае строятся новые треугольники и число выполняемых алгоритмом действий фиксировано. Во втором необходимо построение дополнительных внешних к текущей триангуляции треугольников, причём их количество может в худшем случае равняться $n-1$, где n – число точек в

текущей триангуляции. 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. 14):

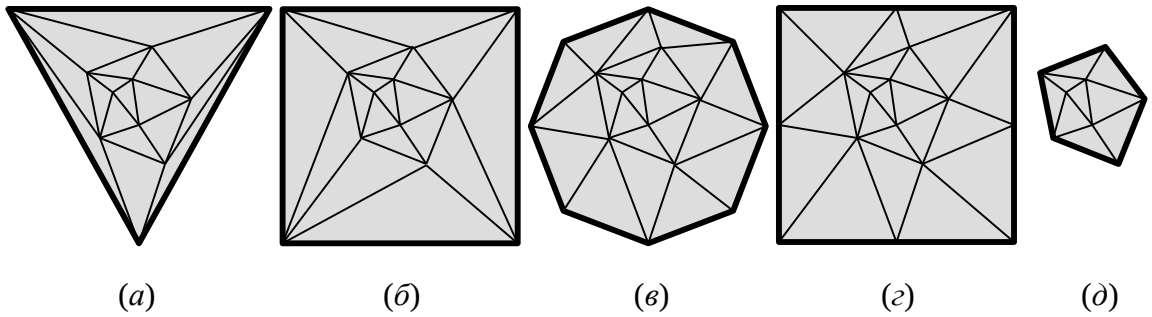


Рис. 14. Варианты суперструктур: *а* – треугольник; *б* – квадрат; *в* – точки на окружности; *г* – точки на квадрате; *д* – выпуклая оболочка

а) вершины равностороннего треугольника, покрывающего всё множество исходных точек (рис. 14,*а*);

б) вершины квадрата, покрывающего всё множество исходных точек (рис. 14,*б*);

в) $\theta(\sqrt{N})$ точек, равномерно распределенных по окружности, покрывающей всё множество исходных точек (рис. 14,*в*);

г) $\theta(\sqrt{N})$ точек, равномерно распределенных по квадрату, покрывающему всё множество исходных точек (рис. 14,*г*);

д) исходные точки, попадающие на выпуклую оболочку множества исходных точек (рис. 14,*д*).

В [17] 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 её снижение, но не более чем на 10 %.

Любое добавление новой точки в триангуляцию теоретически может нарушить условие Делоне, поэтому после добавления точки обычно сразу же производится локальная проверка триангуляции на условие Делоне. Эта проверка должна охватить все вновь построенные треугольники и сосед-

ние с ними. Количество таких перестроений в худшем случае может быть очень велико, что, по сути, может привести к полному перестроению всей триангуляции. Поэтому трудоёмкость перестроений составляет $O(N)$. Однако среднее число таких перестроений на реальных данных составляет чуть менее трёх [17].

Таким образом, наибольший вклад в трудоёмкость итеративного алгоритма даёт процедура поиска очередного треугольника. Именно поэтому все итеративные алгоритмы построения триангуляции Делоне отличаются почти только процедурой поиска очередного треугольника.

2.1. Простой итеративный алгоритм

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

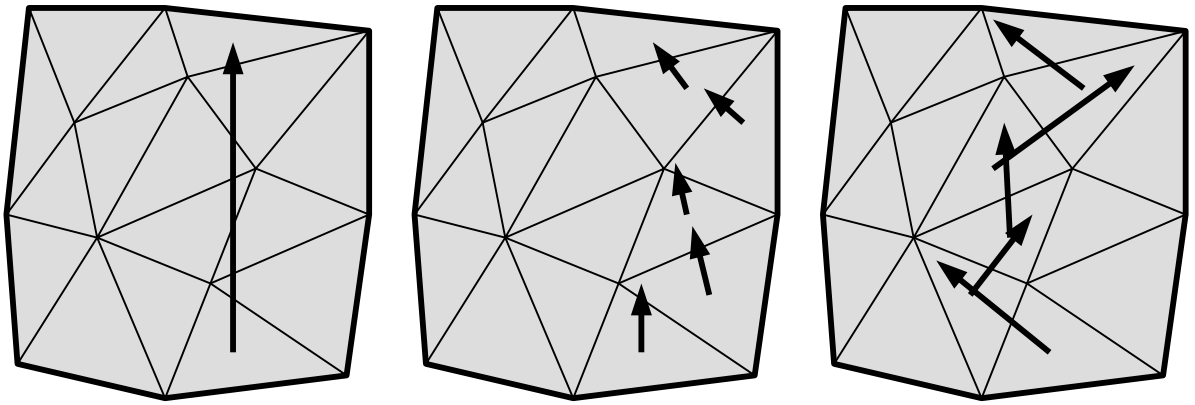
При этом в худшем случае приходится пересекать все треугольники триангуляции, поэтому трудоёмкость такого поиска составляет $O(N)$. Однако в среднем для равномерного распределения в квадрате нужно совершить только $O(\sqrt{N})$ операций перехода [65]. Таким образом, трудоёмкость простейшего итеративного алгоритма составляет в худшем $O(N^2)$, а в среднем – $O(N^{3/2})$ [45,48].

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 трудоёмкости построения триангуляции в среднем $O(N)$.

На практике обычно используются следующие способы поиска треугольника по заданной точке внутри него и по некоторому исходному треугольнику (рис. 15):

1. Проводится прямая через некоторую точку внутри исходного треугольника и целевую точку, а затем нужно идти вдоль этой прямой к цели [45] (рис. 15,*а*). При этом необходимо корректно обрабатывать ситуации, когда на пути могут встретиться узлы и коллинеарные рёбра.

2. Двигаться пошагово, на каждом шаге проводя прямую через центр текущего треугольника и целевую точку и затем переходя к соседнему треугольнику, соответствующему стороне, которую пересекает построенная прямая [48] (рис. 15,*б*).



(*а*) (*б*) (*в*)

Рис. 15. Варианты локализации треугольника в итеративных алгоритмах: *а* – переходы вдоль прямой; *б* – переход через ближайшее к цели ребро; *в* – переход через разделяющее ребро

3. Двигаться пошагово, на каждом шаге переходя 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 [17,67] (рис. 15,*в*). Этот способ обычно обеспечивает 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 узел, рёбра 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. 16,*а*) сразу же удаляются все треугольники, у которых внутрь описанных окружностей попадает новый узел (рис. 16,*б*). При этом все удаленные треугольники неявно образуют некоторый многоугольник. После этого на месте удаленных треугольников строится заполняющая триангуляция путем соединения нового узла с этим многоугольником (рис. 16,*в*) [72].

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 многоугольника, от эффективности работы которого зависит общая скорость алгоритма.

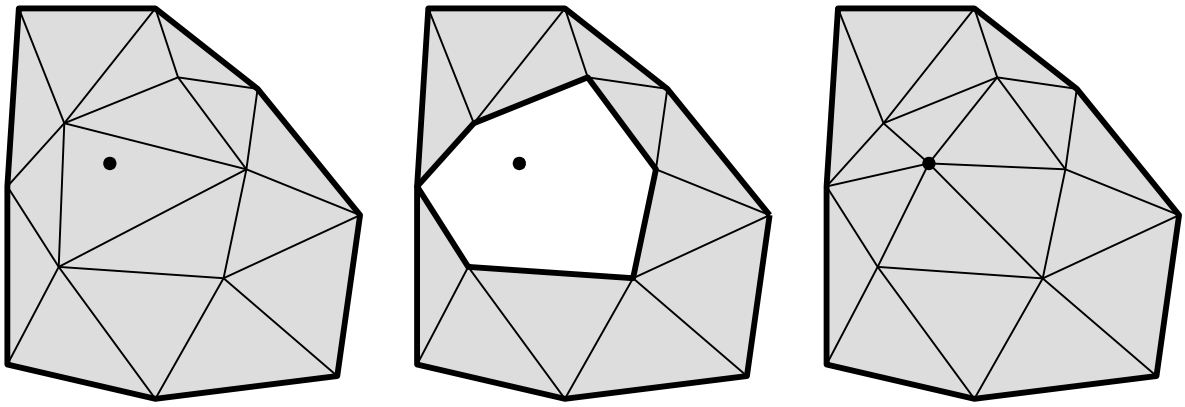


Рис. 16. Вставка точки в итеративном алгоритме «Удаляй и строй»:
а – локализация точки в треугольнике; *б* – удаление треугольников;
в – построение новых треугольников

В целом в зависимости от используемой структуры данных этот алгоритм может тратить времени меньше, чем алгоритм с перестроениями, и наоборот.

Оценка трудоёмкости 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, содержащие заданные точки 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 [40]. При удалении старых треугольников необходимо их удалять из R-дерева, а при построении новых – заносить.

Для 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.

Трудоёмкость поиска треугольника в R-дереве в худшем случае составляет $O(N)$, а в среднем – $O(\log N)$. При этом может быть найдено от 1 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 трудоёмкость 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 k-D-дерево (при $k = 2$) [12] 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 окажется достаточно близко 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.

Трудоёмкость поиска точки в k-D-дереве в худшем случае составляет $O(N)$, а среднем – $O(\log N)$ [12]. Далее может быть задействована процедура перехода по треугольникам, которая может иметь трудоёмкость в худшем случае $O(N)$. Также здесь имеются дополнительные затраты времени на поддержание структуры дерева – $O(\log N)$ при каждом построении и удалении треугольников. Отсюда получаем, что трудоёмкость алгоритма триангуляции с индексированием центров 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

В *алгоритме триангуляции с индексированием центров треугольников квадродеревом* в дерево [11,40] также помещаются только центры треугольников. В целом работа алгоритма и его трудоёмкость совпадают с таковыми для предыдущего алгоритма 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, изменённые 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 вообще там отсутствовать [17].

Основная идея кэширования заключается в построении некоторого более простого, чем триангуляция, планарного разбиения плоскости, 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 локализации искомого треугольника алгоритмом из простого итеративного алгоритма триангуляции Делоне. В качестве такого разбиения проще всего использовать регулярную сеть квадратов (рис. 17). 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 мгновенно можем найти квадрат $\lfloor x/m \rfloor, \lfloor y/m \rfloor$, где $\lfloor ... \rfloor$ – операция взятия целой части.

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.

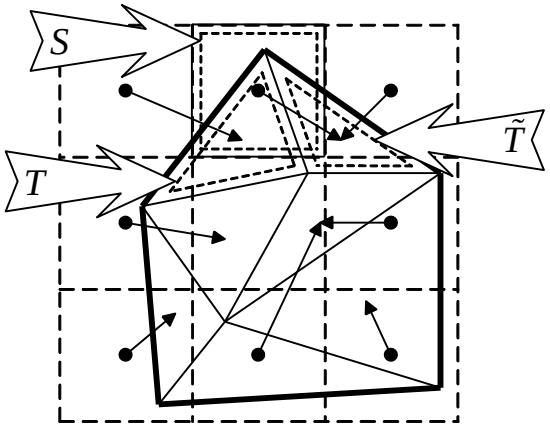


Рис. 17. Локализация точек в кэше (S – найденный квадрат, $\tilde{T}$– связанный с квадратом треугольник, T – конечный треугольник)

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 2-мерного массива r размером $m \times m$ ссылок на треугольники [17]. Первоначально этот массив надо заполнить ссылками на самый первый построенный треугольник. Затем после выполнения очередного поиска, который был начат с квадрата (i, j) и в котором был найден некоторый треугольник T, 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 брать $\approx 0,6 - 0,9$.

Первое время, пока кэш не обновится полностью, поиск может идти довольно долго, но потом скорость повышается. Этого недостатка лишён 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 4 раза (в 2 раза по обеим осям координат), переписывая при этом информацию из старого кэша в новый. При этом для увеличения кэша надо выполнить следующие пересылки (h – старый кэш, h' – новый): $\forall i, j = \overline{0,\ m-1}$: $h'_{2i,2j}, h'_{2i,2j+1}, h'_{2i+1,2j}, h'_{2i+1,2j+1} := h_{i,j}$. Данный алгоритм кэширования позволяет одинаково эффективно работать на маленьком и большом количестве точек, заранее не зная их числа.

Увеличение размера динамического кэша в 2 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 $\approx 3-8$.

Для большинства случайных распределений исходных точек данный алгоритм работает значительно быстрее всех остальных алгоритмов [17]. 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 учёта такой ситуации в алгоритм следует добавить дополнительную проверку. Если очередная добавляемая точка находится от предыдущей точки на расстоянии большем, чем некоторое Δ (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 непревзойдённым по скорости работы на большинстве реальных данных.

2.3.3. Трудоёмкости алгоритмов с кэшированием поиска

Для установления трудоёмкости алгоритмов с кэшированием нам понадобится следующая теорема, представленная в [45,48]:

Теорема 4. Пусть дана триангуляция Делоне на множестве точек, равномерно распределённых в квадрате. Пусть даны два треугольника из этой триангуляции. Тогда для перехода из одного треугольника в другой вдоль прямой, соединяющей некоторые точки этих треугольников, в среднем требуется $\theta(\sqrt{N}) = c\sqrt{N}$ операций, где $c = \text{const}$.

В следующей теореме устанавливается трудоёмкость алгоритма статического кэширования [17].

Теорема 5 (трудоёмкость алгоритма статического кэширования). Пусть даны N точек, на которых требуется построить триангуляцию Делоне и которые распределены в единичном квадрате равномерно и независимо. Пусть размер кэша равен $m \times m$, где $m \sim N^{3/8}$. Тогда трудоёмкость алгоритма со статическим кэшированием в среднем составляет $O(N^{9/8})$.

Доказательство. При использовании кэша, имеющего m^2 ячеек и первоначально пустом, в среднем первые m^2 операций приводят в сумме к $c \cdot \sum_{i=1}^{m^2} \sqrt{i}$ переходам (на основании теоремы 4). Будем считать, что последующие операции добавления точек в триангуляцию приведут к попаданию в ячейку кэша, в которую уже приходилось попадать, а потому будем считать, что локализацию точки необходимо проводить в пределах только

одной данной ячейки, а не всего единичного квадрата. Если принять, что в среднем в ячейке на *i*-м шаге находится i/m^2 точек, то это даёт ещё $c\cdot\sum_{i=m^2+1}^{N}\sqrt{i}/m$ переходов. Итого, общая трудоёмкость операций поиска равна

$$R(m)=c\cdot\left(\sum_{i=1}^{m^2}\sqrt{i}+\sum_{i=m^2+1}^{N}\sqrt{i}/m\right)\approx c\cdot\left(\int_{1}^{m^2+1}\sqrt{x}dx+\frac{1}{m}\int_{m^2+1}^{N+1}\sqrt{x}dx\right)\approx$$

$$\approx\overline{R}(m)=c\cdot\left(\int_{1}^{m^2}\sqrt{x}dx+\frac{1}{m}\int_{m^2}^{N}\sqrt{x}dx\right)=c\cdot\frac{2}{3}\left(x^{3/2}\Big|_{1}^{m^2}+\frac{1}{m}x^{3/2}\Big|_{m^2}^{N}\right)=$$

$$=c\cdot\frac{2}{3}\left(m^3-m^2-1+\frac{N^{3/2}}{m}\right).$$

Приравняв производную $\overline{R}(m)$ к нулю и приняв некоторые упрощения, получим оценку оптимального значения для размера кэша и оценку трудоёмкости:

$$\overline{R}'(m)=c\cdot\frac{2}{3}\left(3m^2-2m-\frac{N^{3/2}}{m^2}\right)=0\quad\Rightarrow\quad 3m^4-2m^3=N^{3/2}.$$

Так как при $m\to\infty$ член $2m^3$ становится пренебрежимо малым по сравнению с $3m^4$, то

$$3m^4\approx N^{3/2}\quad\Rightarrow\quad m^*\sim N^{3/8};\ \overline{R}(m^*)\sim N^{9/8},$$

что и *требовалось доказать*.

Теорема 6 (трудоёмкость алгоритма динамического кэширования). Пусть дано *N* точек, на которых надо построить триангуляцию Делоне и которые распределены в единичном квадрате равномерно и независимо. Пусть размер кэша увеличивается в 2 раза каждый раз при достижении числа точек в триангуляции $n=r\cdot m^2$, где *r* – коэффициент роста динамического кэша, а *m* – текущий размер кэша. Тогда трудоёмкость алгоритма статического кэширования в среднем составляет $O(N)$.

Доказательство. Рассмотрим цикл добавления точек при постоянном размере кэша $m=2^p$. Пусть при последнем увеличении кэша произошло копирование старых ячеек в 4 новых. Тем самым при использовании нового кэша вначале будет возможна локализация точек в среднем в пределах групп по 4 ячейки. Тогда первые m^2 операций вставки точек будут приводить в среднем к попаданию в ячейки, в которые мы ещё не попадали в данном цикле. А поэтому надо будет проводить локализацию в пределах 4 ячеек, т.е. придётся выполнять порядка $c\cdot\sqrt{i/m^2}=c\cdot\sqrt{i}/m$ переходов при

поиске, где i – номер текущей добавляемой точки, а c – некоторая константа.

Так как увеличение кэша в 2 раза производится при достижении числа точек в триангуляции, равного $r \cdot m^2$, то тогда $N \le r \cdot M^2$, где M – максимальный размер кэша. Пусть $M = 2^P$. Тогда можно записать суммарное количество операций переходов при поиске:

$$R(N,r) \le R_1(N,r) = \sum_{p=1}^{P-1} \left(\sum_{i=r\cdot 2^{2p}}^{r\cdot 2^{2p}+2^{p+1}-1} 2c \cdot \frac{\sqrt{i}}{2^p} + \sum_{i=r\cdot 2^{2p}+2^{p+1}}^{r\cdot 2^{2(p+1)}} c \cdot \frac{\sqrt{i}}{2^p} \right).$$

Поступив так же, как и в предыдущем случае – заменив суммы интегралами, найдём оценку числа переходов $\overline{R}(N,r)$:

$$R_1(N,r) \le R_2(N,r) = \int_p^P \left(\int_{r\cdot 2^{2p}}^{r\cdot 2^{2p}+2^{p+1}} 2c \cdot \frac{\sqrt{x}}{2^p} dx + \int_{r\cdot 2^{2p}+2^{p+1}}^{r\cdot 2^{2(p+1)}} c \cdot \frac{\sqrt{x}}{2^p} dx \right) dp =$$

$$= \left(\frac{4c}{3\cdot 2^p} x^{3/2} \Big|_{x=r\cdot 2^{2p}}^{x=r\cdot 2^{2p}+2^{p+1}} + \frac{2c}{3\cdot 2^p} x^{3/2} \Big|_{x=r\cdot 2^{2p}+2^{p+1}}^{x=r\cdot 2^{2(p+1)}} \right) \Bigg|_{p=1}^{p=P} =$$

$$= \left(\frac{2c}{3\cdot 2^p} \left((r\cdot 2^{2p} + 2^{p+1})^{3/2} - 2(r\cdot 2^{2p})^{3/2} + (r\cdot 2^{2(p+1)})^{3/2} \right) \right) \Bigg|_{p=1}^{p=P} \le$$

$$\le \left(\frac{2c}{3\cdot 2^p} (r\cdot 2^{2(p+1)})^{3/2} \right) \Bigg|_{p=1}^{p=P} = \left(\frac{cr^{3/2}\cdot 2^{2p+4}}{3} \right) \Bigg|_{p=1}^{p=P} =$$

$$= \frac{cr^{3/2}\cdot 16}{3} \left(\left(2^P\right)^2 - 4 \right) = \frac{cr^{3/2}\cdot 16}{3} \left(M^2 - 4 \right) \approx \overline{R}(N,r) = \frac{16\cdot c}{3} r^{1/2} N.$$

Таким образом, количество переходов в среднем случае линейно зависит от N: $O(R(N)) = O(N)$.

Кроме операций поиска в алгоритме динамического кэширования производятся локальные перестроения (их количество порядка $3 \cdot N$ [17]), а также операции увеличения кэша. Трудоёмкость увеличения кэша получается порядка $O\left(\sum_{p=1}^{P} 2^p \right) = O(2^{P+1} - 1) = O(N)$.

Итого, общая трудоёмкость алгоритма динамического кэширования в среднем на равномерном распределении в квадрате равна $O(N)$, что и *требовалось доказать*.

Таким образом, трудоёмкости 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 – $O(N)$.

2.4. Итеративные алгоритмы триангуляции с изменённым порядком добавления точек

В [45] 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, применяя алгоритм поиска из простого итеративного алгоритма. Удачно перестраивая порядок добавления точек, можно достичь очень неплохих результатов. Однако при этом на первый план может выйти трудоёмкость этой самой 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 [17]. В этом случае, подобрав соответствующее количество полос, можно существенно уменьшить расстояние между последовательно добавляемыми точками (рис. 18).

В [5] 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 сожалению, не учтено время, затрачиваемое на предобработку – разбиение на полосы. Поэтому на практике число полос лучше выбирать всё 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 $\approx 0{,}15 - 0{,}19$.

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

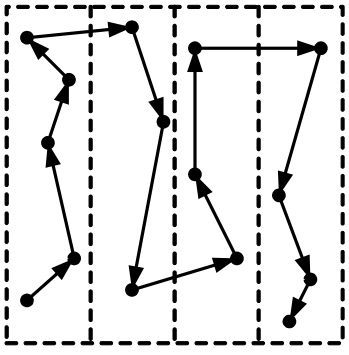


Рис. 18. Порядок выбора точек в итеративном полосовом алгоритме

2.4.2. Итеративный квадратный алгоритм

В *итеративном квадратном алгоритме триангуляции* [17,48] необходимо разбить плоскость с точками на $\theta(\sqrt{N})$ квадратов, а добавление точек производить последовательными группами, соответствующими смежным квадратам (рис. 19).

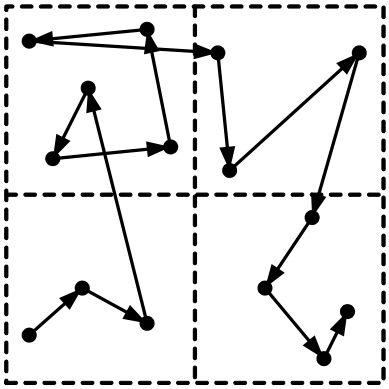


Рис. 19. Порядок выбора точек в итеративном квадратном алгоритме

В [48] разбиение производится на $\sqrt{N}$ квадратов, и трудоёмкость такого алгоритма составляет в худшем случае $O(N^2)$, а в среднем – $O(N^{3/2})$.

В [17] 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 трудоёмкость 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%) [17].

2.4.3. Итеративный алгоритм с послойным сгущением

В *итеративном алгоритме триангуляции с послойным сгущением* [19] 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 её 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 4 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 2-го типа, затем 3-го и в конце 4-го.

На рис. 20 приведен пример разбиения плоскости на квадраты при $u=3$, $v=2$ и $n=9\cdot5$ по этапам:

а) все точки слоя 1;

б) внутренние точки слоя 0;

в) *X*-граничные точки слоя 0;

г) *Y*-граничные точки слоя 0.

На рис. 20 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, имеет трудоёмкость $O(N^2)$. Если исходные точки распределены равномерно, то трудоёмкость алгоритма в среднем будет $O(N)$. Кроме того, в [19] показывается, что если исходные точки удовлетворяют некоторым фиксированным (не вероятностным) ограничениям, то трудоёмкость 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 пере-

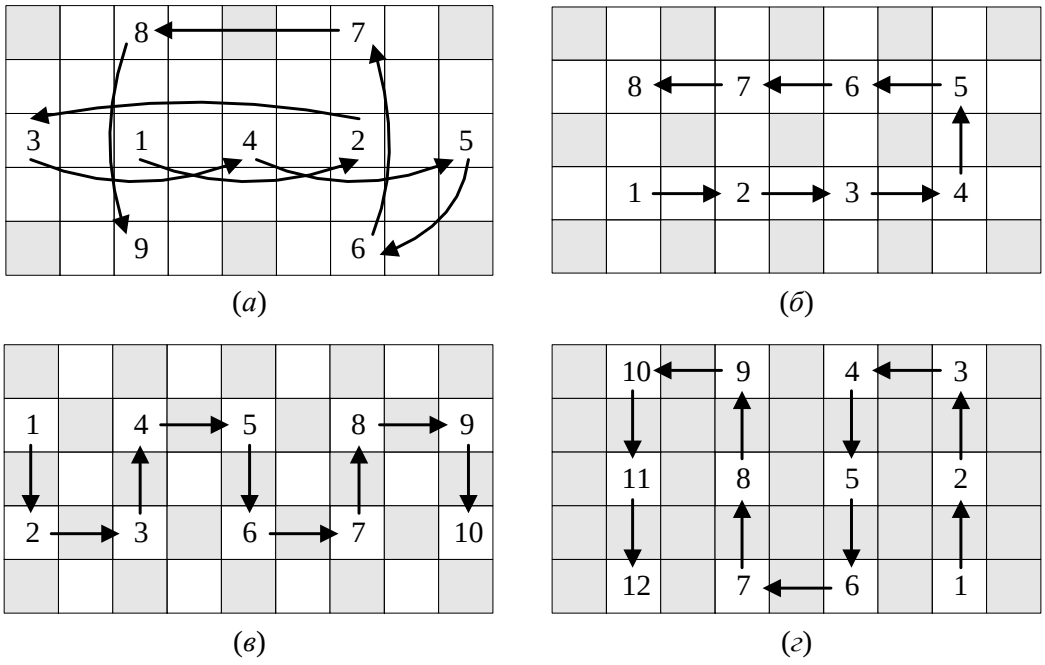


Рис. 20. Порядок выбора точек в итеративном алгоритме с послойным сгущением

страиваются. 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. Итеративный алгоритм с сортировкой вдоль кривой, заполняющей плоскость

Идея *итеративных алгоритмов с сортировкой вдоль кривой, заполняющей плоскость*, 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 [10].

При построении кривой Пеано используется представление точек в системе счисления по основанию 9. На первом шаге область определения исходных точек триангуляции делится на 9 равных квадратов. Эти квадраты нумеруются числами от 0 до 8 и образуется кривая первой итерации (рис. 21,*а*). Далее каждый из квадратов делится еще на 9 частей, и к номеру, полученному на первой итерации, в конце добавляется новая цифра (рис. 21,*б*).

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

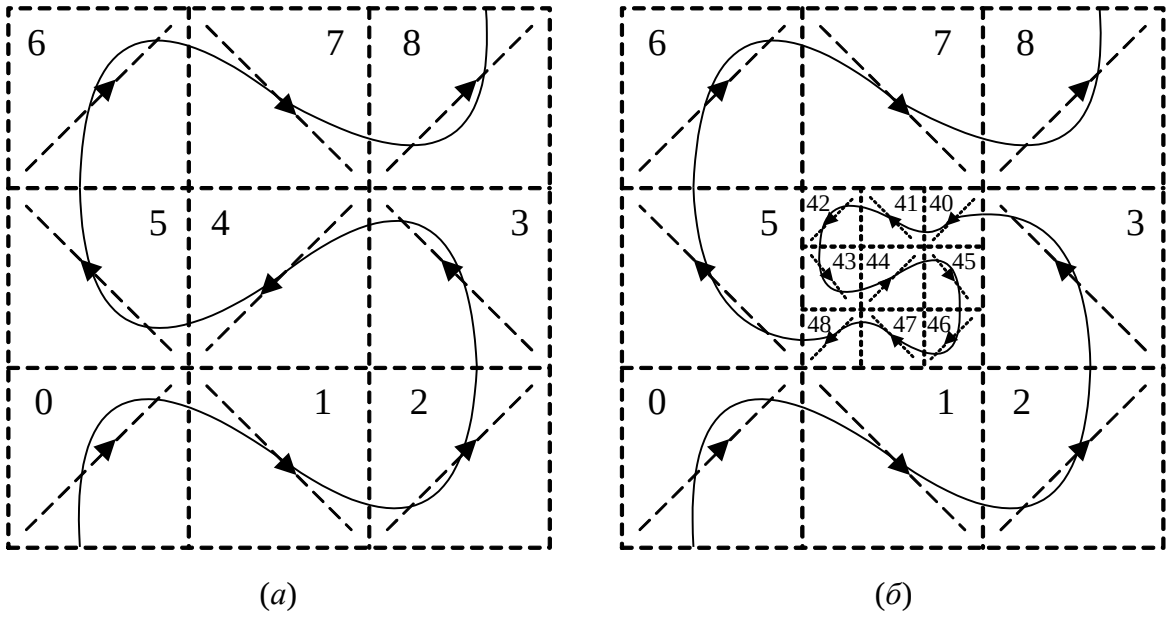
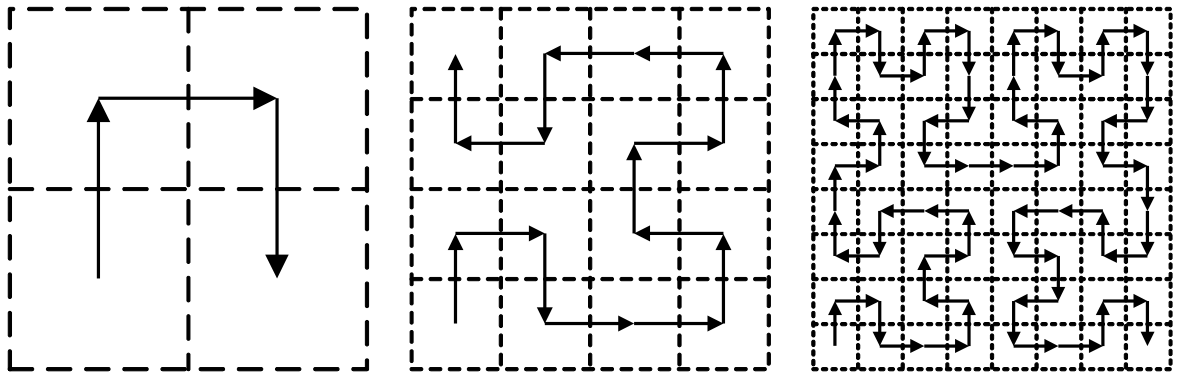


Рис. 21. Построение кривой Пеано: *а* – первая итерация;
б – вторая итерация в центральном квадрате

Аналогично кривой Пеано можно использовать кривую Гильберта. При этом деление области размещения исходных точек выполняется на 4 части в соответствии с рис. 22.



(*а*) (*б*) (*в*)

Рис. 22. Построение кривой Гильберта:
а – первая итерация; *б* – вторая; *в* – третья

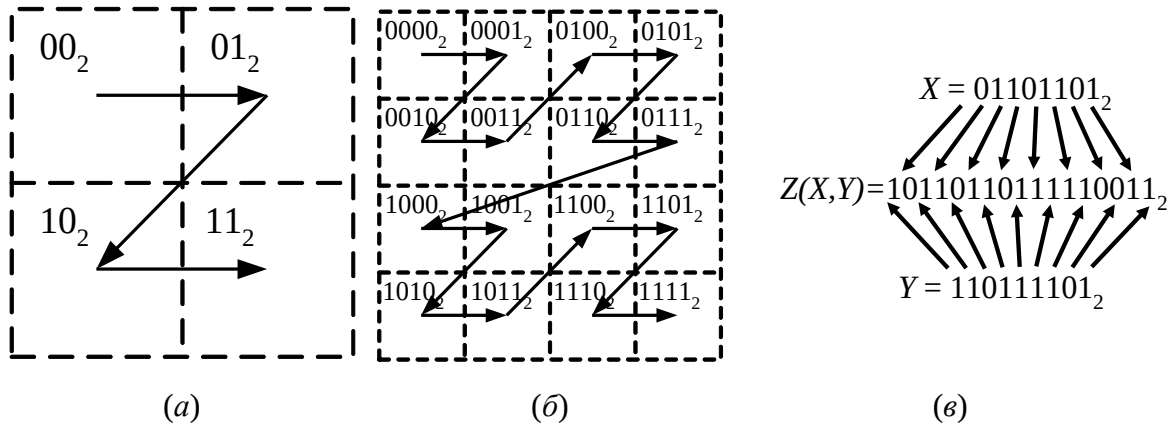
Трудоёмкость итеративных алгоритмов с сортировкой вдоль кривых Пеано и Гильберта составляет в худшем случае $O(N^2)$, а в среднем – $O(N)$. На практике данные алгоритмы работают примерно с той же скоростью, что и итеративный полосовой алгоритм.

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 4 равных прямоугольника и занумеровать их двоичными числами от 0 до 11_2 в соответствии с рис. 23,*а*. Далее каждый из полученных прямоугольников надо опять разбить на 4 части, опять занумеровать их двоичными числами от 0 до 11_2 и добавить к концу кода, полученному ранее (рис. 23,*б*). При необходимости так можно рекурсивно продолжить достаточно далеко.

Однако на практике столь сложные манипуляции проделывать не стоит. В действительности Z-код можно получить очень просто. Для этого надо 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 следует просто взять по очереди все биты координат X и Y с первого по последний и сформировать новую битовую последовательность (рис. 23,*в*).



(*а*) (*б*) (*в*)

Рис. 23. Итеративный алгоритм с сортировкой по Z-коду: *а*, *б* – порядок вставки точек в триангуляцию; *в* – побитовое получение Z-кода

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 для Z-кода, чем для кодов Пеано и Гильберта.

Трудоёмкость 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.

Глава 3. Алгоритмы построения триангуляции Делоне слиянием

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.

3.1. Алгоритм слияния «Разделяй и властвуй»

В *алгоритме триангуляции «Разделяй и властвуй»* [39,48] множество точек разбивается на две как можно более равные части с помощью горизонтальных и вертикальных линий (рис. 24). Алгоритм триангуляции рекурсивно применяется к подчастям, а затем производится *слияние* (объединение, склеивание) 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 всё множество на части по 3 и по 4 точки.

Если допустить, что число точек в триангуляции всегда будет >5, то всё множество точек можно разбить на элементарные части по 3 и по 4 точки. Действительно, $\forall N>5$:

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

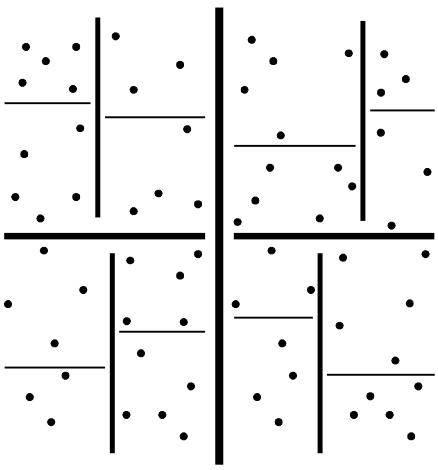


Рис. 24. Разбиение множества исходных точек в алгоритме триангуляции «Разделяй и властвуй»

Рекурсивный алгоритм триангуляции «Разделяй и властвуй».

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

Шаг 2. Иначе, если число точек $N = 4$, построить триангуляцию из 2 или 3 треугольников.

Шаг 3. Иначе, если число точек $N = 8$, разбить множество точек на две части по 4 точки, рекурсивно применить алгоритм, а затем склеить триангуляции.

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

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

Элементарные множества из трех или четырёх элементов (рис. 25) легко триангулируются (можно даже просто перебрать множество всех возможных вариантов).

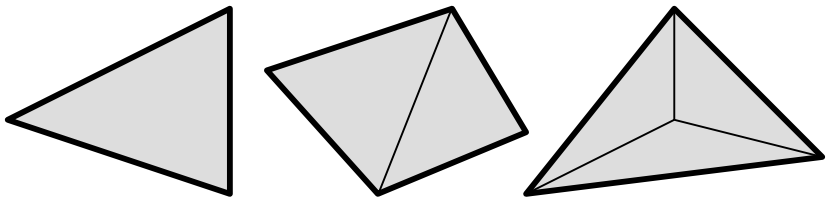


Рис. 25. Триангуляции множеств из 3 и 4 точек

Трудоёмкость алгоритма «Разделяй и властвуй» составляет $O(N \log N)$ в среднем и худшем случаях.

Основная и самая сложная часть алгоритма «Разделяй и властвуй» 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»

Процедура слияния триангуляций «Удаляй и строй» была предложена в [39,48] как часть соответствующего алгоритма триангуляции «Разделяй и властвуй».

Вначале для двух соединяемых триангуляций находятся две общие касательные P_0P_1 и P_2P_3, первая из которых становится текущей *базовой линией* (рис. 26,*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 как рост «пузыря» от базовой линии, пока не встретится какой-нибудь узел. «Пузырь» – это окружность, которая проходит через точки P_0 и P_1 и центр которой находится на срединном перпендикуляре к базовой линии. Например, на рис. 26,*б* первым таким найденным узлом становится точка A. На найденном узле и базовой линии строится новый треугольник (в нашем случае ΔP_0P_1A). Однако при этом иногда возникает необходимость предварительно удалить некоторые ранее построенные треугольники, которые перекрываются новым треугольником. Так, на рис. 26,*б* должен быть удален ΔP_0BA. После этого открытая сторона нового треугольника становится новой базовой линией (на рис. 26,*в* – AP_1), и цикл продолжается, пока не будет достигнута вторая касательная.

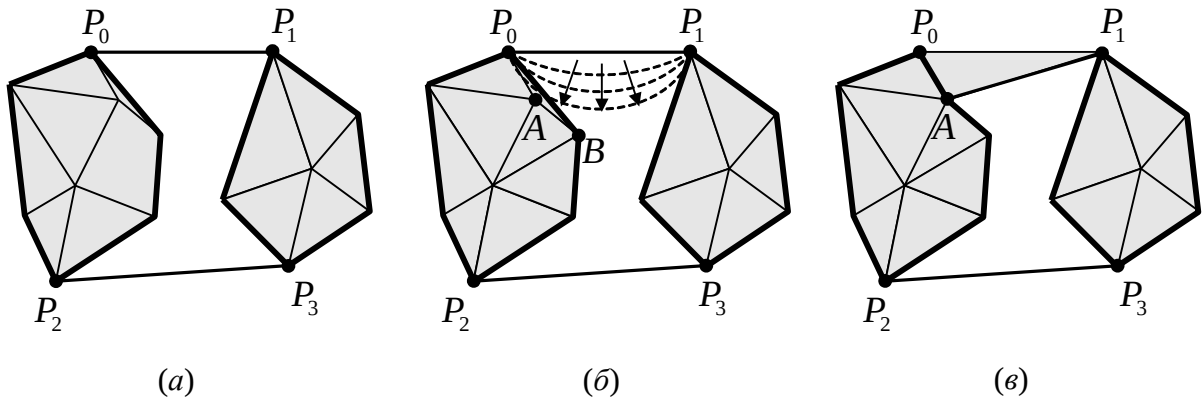


Рис. 26. Слияние триангуляций: *а* – поиск касательных; *б* – поиск ближайшего соседа Делоне для базовой линии; *в* – построение треугольника

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 трудоёмкость $O(N)$ относительно общего количества точек в двух объединяемых триангуляциях [48].

3.1.2. Слияние триангуляций «Строй и перестраивай»

В *процедуре слияния триангуляций «Строй и перестраивай»* имеется два этапа работы [17]. Вначале строятся заполняющие зону слияния треугольники между касательными (рис. 27,*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 $\Delta Q_1Q_2N_1$ и $\Delta Q_1Q_2N_2$ выбирается тот, который, во-первых, можно построить (т.е. $\angle Q_2Q_1N_1 < 180^\circ$ для первого и $\angle Q_1Q_2N_2 < 180^\circ$ 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. 27,*б*). Далее цикл построения треугольников продолжается, пока не будет достигнута вторая касательная.

На втором этапе все вновь построенные треугольники проверяются на выполнение условия Делоне с другими треугольниками и при необходимости треугольники перестраиваются (рис. 27,*в*).

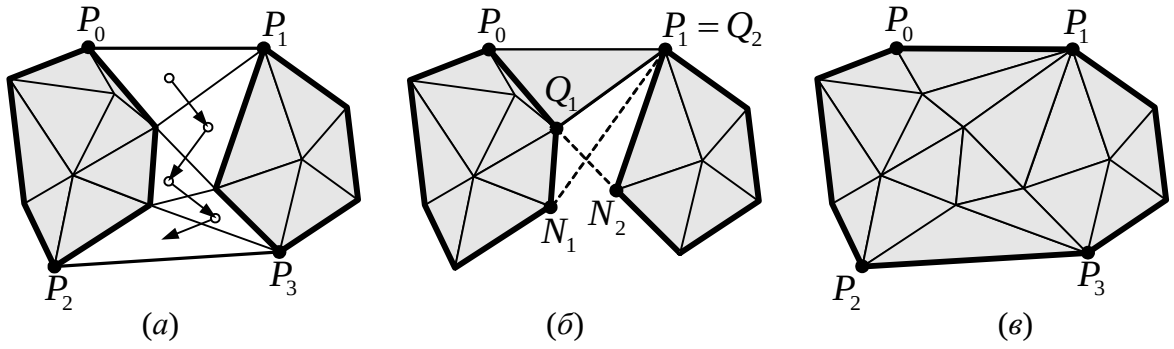


Рис. 27. Слияние триангуляций: *а* – предварительное слияние; *б* – выбор очередного треугольника; *в* – финальная триангуляция после перестроений треугольников

Процедура слияния «Строй и перестраивай» значительно проще в реализации предыдущей и обычно работает заметно быстрее на реальных данных. В целом она имеет трудоёмкость в худшем случае $O(N)$ 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 [17].

3.1.3. Слияние триангуляций «Строй, перестраивая»

Процедура слияния триангуляций «Строй, перестраивая» отличается от предыдущей только тем, что перестроения выполняются не на втором этапе работы, а непосредственно после построения очередного треугольника [17] (рис. 28). В таком случае отпадает необходимость помнить список всех построенных треугольников, несколько уменьшается общее коли-

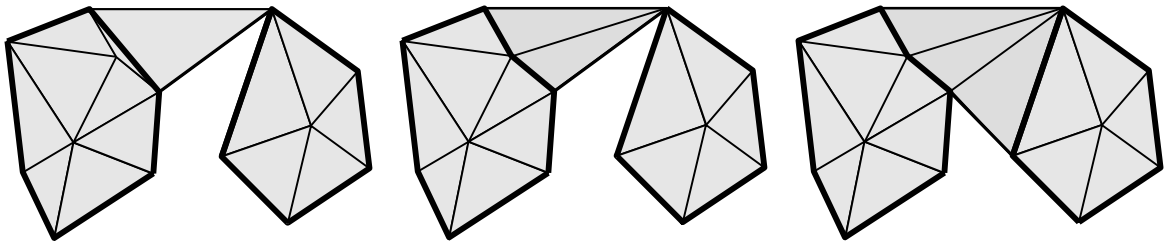


Рис. 28. Шаги слияния «Строй, перестраивая»: *а* – построение первого треугольника; *б* – немедленные перестроения; *в* – построение второго треугольника

чество 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, так как образующий её треугольник может быть перестроен.

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

3.2. Рекурсивный алгоритм с разрезанием по диаметру

Рекурсивный алгоритм триангуляции с разрезанием по диаметру 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 работе [58].

Для разделения множества точек на две части вначале необходимо построить выпуклую оболочку всех исходных точек (эта операция выполняется за время $O(N)$ [12]). Далее по выпуклой оболочке вычисляется диаметр D_1D_2 (рис. 29,*а*) [12]. Это выполняется в худшем случае за время $O(N\log N)$, а в среднем – за $O(N)$. Затем необходимо найти такую пару точек P_1 и P_2, что 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 участков выпуклой оболочки между D_1 и D_2 (рис. 29,*а*).

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. 29,*б*). Затем обе полученные триангуляции соединяются вдоль общего ребра

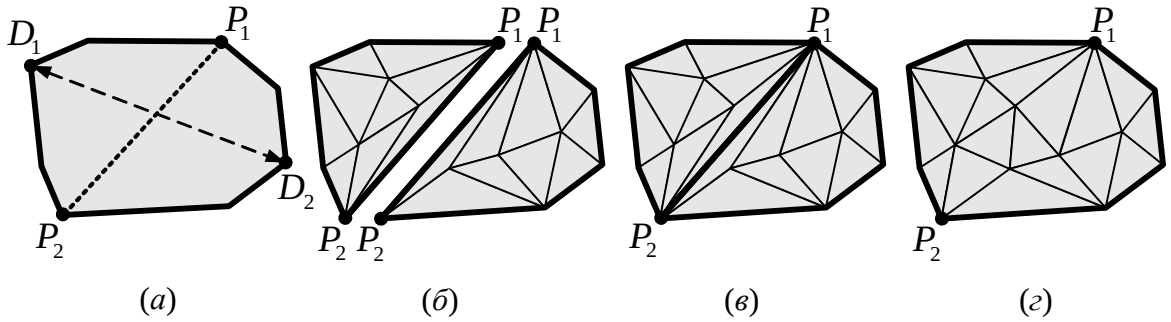


Рис. 29. Слияние триангуляций: *а* – поиск диаметра и точек разделения; *б* – раздельная триангуляция; *в* – соединение триангуляций по общему ребру; *г* – перестроения

P_1P_2 (рис. 29,*в*) и выполняются необходимые перестроения пар соседних треугольников для удовлетворения условия Делоне (рис. 29,*г*).

Данный алгоритм слияния по сравнению с «Разделяй и властвуй» сложнее в процедуре разделения точек на части, но проще в слиянии. Трудоёмкость 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 трудоёмкости двух предыдущих алгоритмов порождена их рекурсивным характером. Избавившись от рекурсии, можно попытаться улучшить и трудоёмкость триангуляции. В [17] предлагается разбить исходное множество точек на такие подмножества, чтобы построение по ним триангуляций занимало минимальное время, например за счёт 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. Полученные частичные полосовые триангуляции объединяются, при этом необходимо: 1) либо достраивать триангуляции до выпуклых, а затем использовать обычный алгоритм слияния из алгоритма «Разделяй и властвуй»; 2) либо применять более сложный алгоритм соединения невыпуклых триангуляций.

Алгоритм полосового слияния.

Шаг 1. Разбиение исходного множества точек на некоторые полосы.

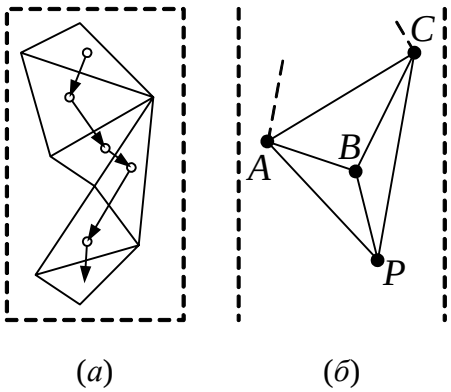
Шаг 2. Применение специального быстрого алгоритма получения невыпуклой триангуляции полосы точек.

Шаг 3. Слияние полученных триангуляций. *Конец алгоритма.*

Рассмотрим эти шаги подробнее.

Шаг 1. Множество всех точек разбивается на несколько 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 трёх (этого требует алгоритм, применяемый на следующем шаге алгоритма). Если это не выполняется для какого-либо столбца, то его нужно присоединить к соседнему. Трудоёмкость данного шага составляет $O(N)$ в соответствии с трудоёмкостью 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. 30,*а*). Вначале на трёх самых верхних точках в столбце строится первый треугольник, и он помечается как текущий. Затем последовательно перебираются все остальные точки в столбце, начиная с четвёртой, сверху вниз и добавляются к частичной триангуляции. Пусть ΔABC является текущим, точка B имеет наименьшую из точек треугольника координату Y, а P – очередная добавляемая точка из столбца. На очередном шаге необходимо выбрать, какой треугольник создавать – ΔABP или ΔBCP (рис. 30,*б*). Иногда один из этих треугольников построить невозможно из-за пересечений рёбер триангуляции. Если же построить можно оба треугольника, естественно выбрать тот, у которого минимальный из углов больше, так как тогда с большей вероятностью в будущем не придётся его перестраивать из-за невыполнения условия Делоне.



(*а*) (*б*)

Рис. 30. 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 построенная триангуляция полосы точек будет невыпуклой.

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

3.3.1. Выбор числа полос в алгоритме полосового слияния

Одним из важнейших параметров полосовых алгоритмов триангуляции является количество полос, которое следует выбрать так, чтобы минимизировать число последующих перестроений пар соседних треугольников. Для этого в [17] предлагается минимизировать среднюю суммарную длину рёбер всех полученных невыпуклых триангуляций. Сделаем следующие упрощающие предположения:

1. Координаты точек распределены в прямоугольной области шириной a и высотой b равномерно и независимо по X и Y.

2. Расстояние между точками будем определять по Манхеттену:

$$p(\{x_i,y_i\},\{x_j,y_j\}) = \left|x_i - x_j\right| + \left|y_i - y_j\right|.$$

Тогда средняя суммарная длина рёбер равна сумме средних длин по X плюс сумма средних длин по Y. Пусть ширина прямоугольной области есть a, высота – b, число полос – m. Тогда среднее число точек в полосе равно N / m, где N – общее число точек в триангуляции. По координате X расстояние между двумя точками в полосе в среднем равно 1/3 от ширины полосы (среднее разности двух равномерно распределённых на интервале величин). Заметим, что в соответствии с приведённым алгоритмом быстрой триангуляции полосы точек получается не менее $2k-3$ рёбер в каждой полосе (вначале строится один треугольник – 3 ребра; затем остаётся $k-3$ точки, и при каждом добавлении точки строятся 2 ребра; итого $3+(k-3)\cdot 2$ рёбер). Средняя ширина полосы равна a/m. Итого, по координате X во всех полосах сумма длин равна $(a/3m)(2k-3)m$.

Теперь найдём сумму длин рёбер по координате Y. Все построенные рёбра в невыпуклой триангуляции полосы должны принадлежать одной из трёх групп:

1. Множество рёбер, образующих левую границу триангуляции.
2. Множество рёбер, образующих правую границу.
3. Множество рёбер сшивания между границами.

Если пренебречь необходимыми перестроениями треугольников в процессе работы, то получается, что первые две группы оказываются ломаными, протянувшимися «почти» с верха полосы до низа («почти» потому, что с ростом N, а следовательно, и k среднее расстояние от границы интервала до ближайшей из k равномерно распределённых величин на интервале, равное $1/(k+1)$, стремится к нулю). Средняя длина по координате Y в каждой из этих двух групп составит $b-(2/k)b$. Третья группа рёбер является ломаной со средней длиной $b-(4/k)b$ по координате Y и ещё некоторыми дополнительными рёбрами. Пусть средняя общая длина по координате Y таких дополнительных рёбер равна qb. Тогда сумма по координате Y во всех полосах получается равной $\big(b-(2/k)b+b-(2/k)b+b-$ $-(4/k)b+qb\big)\cdot m=\big(3+q-(8/k)\big)\cdot bm$. Таким образом, приближённая сум-

марная длина рёбер в триангуляциях полос точек составляет $L(m) = (a/3)(2k-3) + \left(3 + q - (8/k)\right) \cdot bm$. Если пренебречь членом $8/k$, стремящимся к нулю при больших N, и учесть, что $k = N/m$, то, найдя производную L по m и приравняв её к нулю, получим приближённое оптимальное значение для m:

$$L(m) \approx L(m) = \frac{a}{3}(2k-3) + (3+q)am = \frac{2aN}{3m} - a + (3+q)bm;$$

$$L'(m) = -\frac{2aN}{3m^2} + (3+q)b = 0; \Rightarrow 2aN = 3(3+q)bm^2; \Rightarrow m^* = \sqrt{\frac{2aN}{3(3+q)b}}. \quad (3)$$

Эта оценка позволяет минимизировать сумму длин рёбер полосовых триангуляций, т.е. строить треугольники, которые не будут с большой вероятностью перестраиваться в дальнейшем. Таким образом, выбор числа полос в данном алгоритме влияет на количество последующих перестроений и, следовательно, на время работы всего алгоритма. Так как формула (3) включает неизвестную величину q, которую трудно оценить, то в [17] было проведено практическое исследование зависимости числа полос от количества исходных точек.

Число полос следует выбирать по следующей формуле:

$$\overline{m}^* = \sqrt{s \cdot (a/b) \cdot N},$$

где s – коэффициент разбиения на полосы алгоритма полосового слияния, который на практике следует взять $\approx 0{,}11 - 0{,}15$.

3.3.2. Алгоритм выпуклого полосового слияния

Пошаговая схема работы *алгоритма выпуклого полосового слияния* схематично изображена на рис. 31,*а–г*. Основная его идея заключается в построении выпуклых полосовых триангуляций и последующем применении любого алгоритма слияния, используемого в алгоритме «Разделяй и властвуй». Первые два шага данного алгоритма приведены в разд. 0, а здесь мы рассмотрим последний – третий шаг.

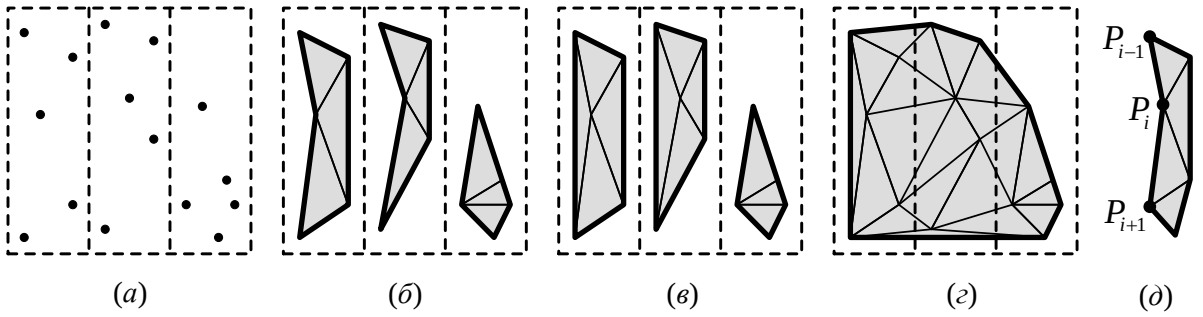


Рис. 31. Алгоритм выпуклого полосового слияния:
а–г – шаги работы алгоритма; *д* – достраивание выпуклости

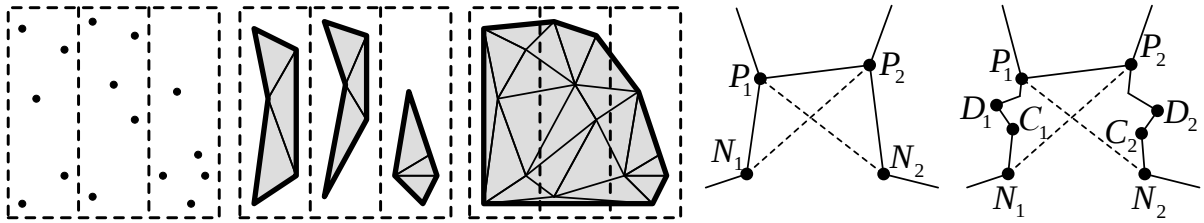
Шаг 3а. Выполняется обход всех граничных узлов P_i частичных триангуляций. Анализируются соседние к P_i вдоль границы узлы P_{i-1} и P_{i+1}. Если $\angle P_{i-1}P_iP_{i+1} < 180^\circ$, то в точке P_i нарушается условие выпуклости триангуляции и необходимо построить $\Delta P_{i-1}P_iP_{i+1}$, а дальнейший анализ граничных узлов надо продолжить с предыдущего узла P_{i-1} (рис. 31,*д*).

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

В целом трудоёмкость 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 следущем алгоритме невыпуклого слияния.

3.3.3. Алгоритм невыпуклого полосового слияния

Пошаговая схема работы *алгоритма невыпуклого полосового слияния* схематично изображена на рис. 32,*а–в*. Первые два шага этого алгоритма приведены в разд. 0, а здесь мы рассмотрим шаг 3.



(*а*) (*б*) (*в*) (*г*) (*д*)

Рис. 32. Алгоритм невыпуклого полосового слияния:
а–в – шаги работы алгоритма; *г* – слияние полос;
д – локальное достраивание выпуклости

Шаг 3. На вход алгоритма подаются невыпуклые полосовые триангуляции, а в ходе работы алгоритма перед очередным построением соединяющего ребра и, соответственно, треугольника производится достраивание выпуклости на некотором расстоянии от соединяющего ребра.

Рассмотрим этот шаг 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 алгоритме выпуклого слияния на очередном шаге достаточно было выбрать, какой треугольник строить – $\Delta P_1P_2N_1$ или $\Delta P_1P_2N_2$ (рис. 32,*г*). В алгоритме невыпуклого слияния необходимо проанализировать выпук-

лость 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. 32,*д*).

Невыпуклое слияние, по сути, является вариантом задачи триангуляции монотонного относительно вертикали многоугольника. Решение последней задачи приведено в [12].

При таком подходе удаётся заметно уменьшить число перестроений и, следовательно, время работы алгоритма. В [17] показано, что алгоритм невыпуклого слияния работает примерно на 10–15% быстрее, чем алгоритм выпуклого слияния.

Глава 4. Двухпроходные алгоритмы построения триангуляции Делоне

При построении триангуляции Делоне итеративными алгоритмами и алгоритмами слияния для каждого нового треугольника должно быть проверено условие Делоне. Если оно не выполняется, то должны последовать перестроения треугольников и новая серия проверок. На практике довольно много времени отнимают как раз проверки на условие Делоне и перестроения.

Для уменьшения числа проверок условия Делоне и упрощения логики работы алгоритмов можно использовать следующий подход. Вначале за первый проход нужно 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 условию Делоне. Допустимость такой двухпроходной стратегии устанавливается в теореме 1 (см. разд. 1.1).

4.1. 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), так как после того, как треугольник построен, он может тут же в результате неудачной проверки на условие Делоне исчезнуть, а на его месте появятся другие треугольники. Кроме того, в алгоритмах слияния сразу строится 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%, а 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. 33 приведен пример применения двухпроходной стратегии к алгоритму выпуклого полосового слияния.

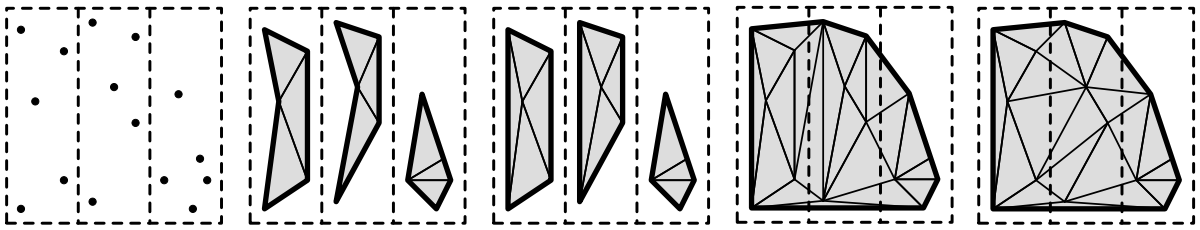


Рис. 33. Двухпроходный алгоритм выпуклого полосового слияния:
а–г – построение некоторой триангуляции; *д* – полное перестроение

В [17] показывается, что двухэтапные полосовые алгоритмы и «Разделяй и властвуй» работают в среднем на 10–15% быстрее оригинальных алгоритмов. Это, в частности, объясняется некоторым упрощением логики их работы.

4.2. Модифицированный иерархический алгоритм

Для итеративных алгоритмов двухпроходная стратегия, как правило, не годится, так как сразу же образуются узкие длинные треугольники, которые в дальнейшем делятся на другие ещё меньшие и ещё более узкие. Тем не менее в [59] описывается *модифицированный иерархический алгоритм*, являющийся, по сути, обычным итеративным алгоритмом, выполняемым за 2 прохода.

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

4.3. 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. 34,*а*). Затем, последовательно перебирая точки сверху вниз, они соединяются в одну невыпуклую триангуляцию (рис. 34,*б*). Далее триангуляция достраивается до выпуклой (рис. 34,*в*). И в заключение производится полное перестроение триангуляции для выполнения условия Делоне (рис. 34,*г*).

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

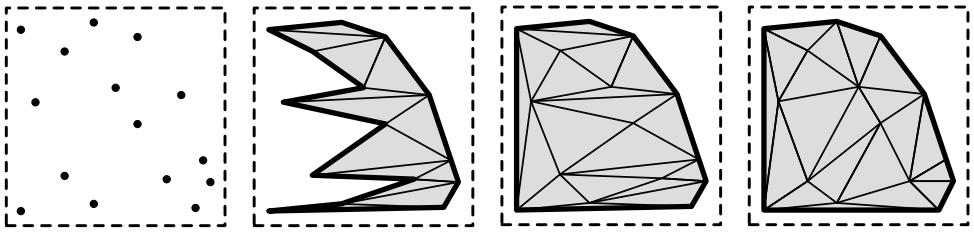


Рис. 34. Линейный алгоритм: *а* – исходные точки; *б* – невыпуклая триангуляция; *в* – достраивание до выпуклой; *г* – перестроение триангуляции

4.4. 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 (рис. 35,*а*). Далее для остальных точек вычисляется полярный угол относительно выбранной центральной точки и все точки сортируются по этому углу (рис. 35,*б*). Затем все точки соединяются рёбрами с центральной точкой и соседними в отсортированном списке (рис. 35,*в*). Потом триангуляция достраивается до выпуклой (рис. 35,*г*). И в заключение производится полное перестроение триангуляции для выполнения условия Делоне (рис. 35,*д*).

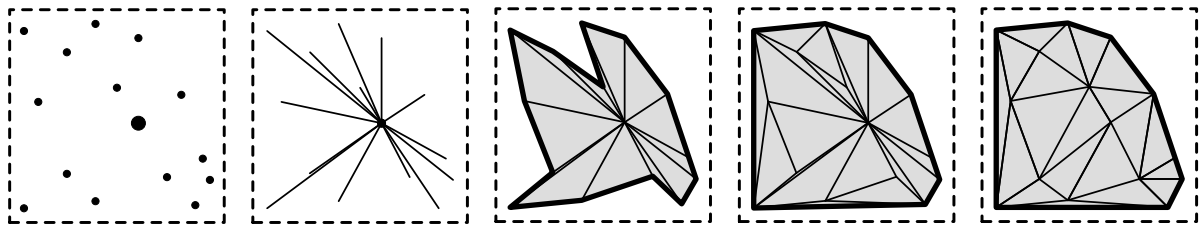


Рис. 35. Веерный алгоритм: *а* – исходные точки; *б* – круговая сортировка; *в* – невыпуклая триангуляция; *г* – достраивание до выпуклой; *д* – перестроение триангуляции

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

4.5. Алгоритм рекурсивного расщепления

Алгоритм рекурсивного расщепления работает в два прохода [54]. Второй проход аналогичен всем двухпроходным алгоритмам триангуляции, а первый похож на рекурсивный алгоритм с разрезанием по диаметру, но разрезание производится не 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 многоугольник оболочки примерно пополам. 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. 36,*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. 36,*б*). Кроме того, так как 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. 36,*в*). После построения триангуляций отдельных частей выполняется их соединение вдоль разделяющей ломаной (рис. 36,*г*,*д*).

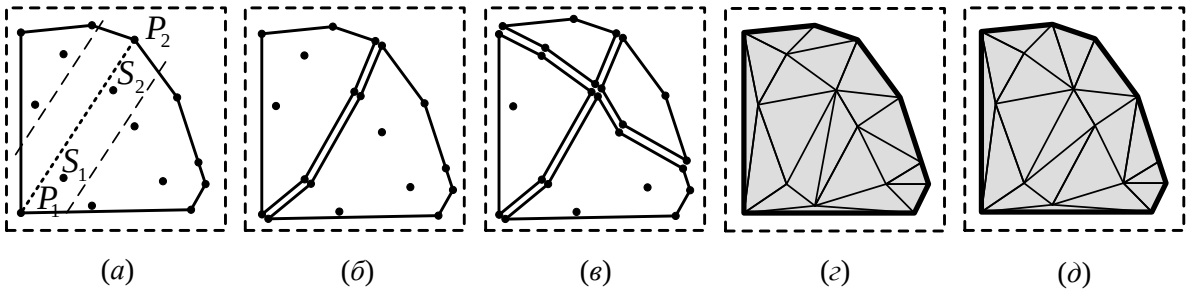


Рис. 36. Алгоритм рекурсивного расщепления: *а* – выбор направления и коридора; *б*–*г* – шаги расщепления; *д* – перестроение триангуляции

Теоретически алгоритм рекурсивного расщепления имеет трудоёмкость $O(N \log N)$ в среднем и худшем случаях [54]. Однако на практике процедура расщепления является сложной для реализации, медленной в работе и в целом алгоритм работает существенно медленнее любых двухпроходных алгоритмов слияния.

4.6. 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. 37,*а*). Затем точки сортируются внутри полос и последовательно соединяются в ломаные (рис. 37,*б*). В последующем все полосы склеиваются между собой при помощи процедуры слияния из алгоритма невыпуклого полосового слияния (рис. 37,*в*). После этого полученная триангуляция достраивается до

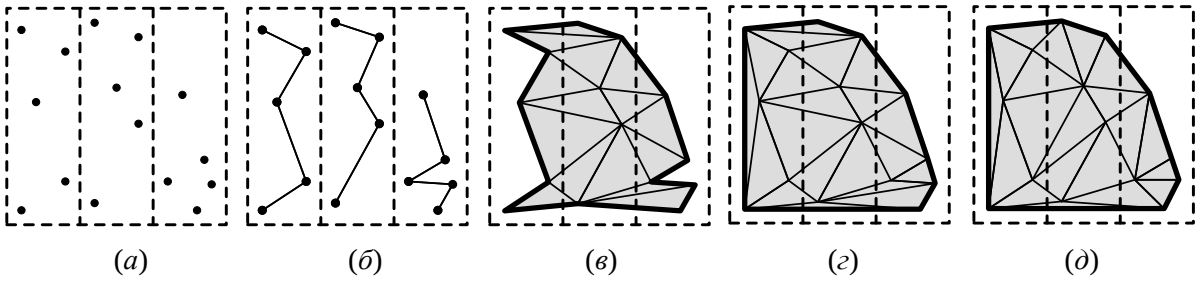


Рис. 37. Ленточный алгоритм: *а* – разбиение на полосы; *б* – построение в полосах ломаных; *в* – слияние полос; *г* – достраивание до выпуклости; *д* – перестроение триангуляции

выпуклой (рис. 37,*г*) и производится полное перестроение триангуляции для выполнения условия Делоне (рис. 37,*д*).

В данном алгоритме используется процедура слияния, соединяющая ломаные – рёбра будущей триангуляции. Поэтому наиболее удобно этот алгоритм реализуется на структуре данных «Узлы, рёбра и треугольники», представляющей рёбра в явном виде.

Главным параметром данного алгоритма является количество полос, которые необходимо выбрать по той же самой формуле, что и для алгоритмов полосового слияния:

$$m = \sqrt{s \cdot (a/b) \cdot N}\ ,$$

где s – коэффициент разбиения на полосы, значение которого на практике следует взять $\approx 0{,}1 - 0{,}15$.

Трудоёмкость данного алгоритма составляет в среднем случае $O(N)$.

Глава 5. Прочие алгоритмы построения триангуляции Делоне

Во всех рассмотренных выше алгоритмах на разных этапах построения триангуляции могут быть получены 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 [58].

5.1. Пошаговый алгоритм

Пошаговый алгоритм [58], известный также как *алгоритм прямого перебора* и *метод активных рёбер*, 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. 38). Базовая линия берется как один из отрезков многоугольника выпуклой оболочки всех исходных точек триангуляции.

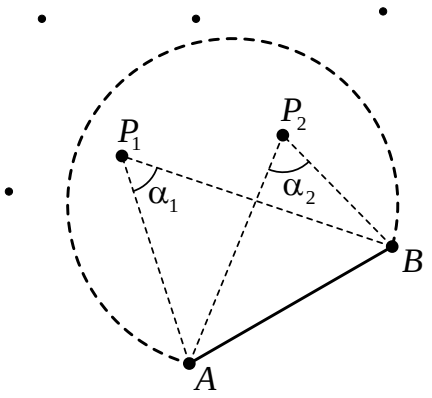


Рис. 38. Выбор очередной точки для включения
в триангуляцию в пошаговом алгоритме

Далее для базовой линии необходимо найти *соседа Делоне* – узел, который вместе с концами данной базовой линии в триангуляции Делоне является вершинами одного треугольника. Процесс поиска можно представить как рост «пузыря» от базовой линии, пока не встретится какой-нибудь узел. В данном случае «пузырь» – это окружность, которая проходит через точки A и B и центр которой находится на срединном перпендикуляре к базовой линии (похожее определение «пузыря» дано в п. 3.1.1).

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. 38 будет выбрана точка P_2). Найденный сосед Делоне соединяется отрезками с концами базовой линии и образует треугольник ΔAP_iB. Новые рёбра AP_i и BP_i построенного треугольника помечаются как новые базовые линии, и процесс поиска треугольников продолжается.

Трудоёмкость пошагового алгоритма составляет $O(N^2)$ в среднем и в худшем случае. Из-за столь большой трудоёмкости на практике такой алгоритм почти не применяется.

5.2. Пошаговые алгоритмы с ускорением поиска соседей Делоне

Квадратичная сложность пошагового алгоритма обусловлена трудоёмкой процедурой поиска соседа Делоне. В следующих двух алгоритмах предлагаются два варианта ускорения поиска.

5.2.1. Пошаговый алгоритм с k-D-деревом поиска

В *пошаговом алгоритме с k-D-деревом поиска* вначале все исходные точки триангуляции помещаются в k-D-дерево (при $k = 2$) [12] или любое другое, позволяющее эффективно выполнять региональный поиск в заданном квадрате со сторонами, параллельными осям координат.

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 2 раза (на рис. 39 цифрами обозначены возможные этапы роста «пузыря»).

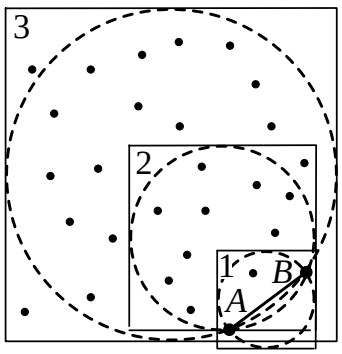


Рис. 39. Выбор очередной точки в пошаговом алгоритме с k-D-деревом поиска

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.

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

5.2.2. Клеточный пошаговый алгоритм

В *клеточном пошаговом алгоритме* предлагается построить клеточное разбиение плоскости и поиск очередного соседа Делоне вести, последовательно перебирая точки в близлежащих к базовой линии клетках [3]. 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 порядка $\theta(N)$.

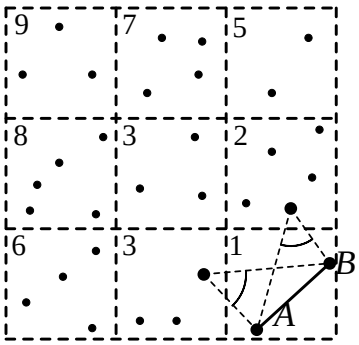


Рис. 40. Выбор очередной точки в клеточном алгоритме

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. 40 цифрами помечен порядок обхода клеток). Трудоёмкость данного алгоритма в среднем на ряде распространённых распределений составляет $O(N)$ [3], а в худшем случае – $O(N^2)$.

5.3. Алгоритм построения через трёхмерные выпуклые оболочки

Данный алгоритм основан на интересном свойстве триангуляции, связывающем её с выпуклыми оболочками в трёхмерном пространстве. Если каждую исходную точку (x, y) сдвинуть параллельным переносом на

параболоид, получив точку $(x, y, x^2 + y^2)$ в трёхмерном пространстве, то оказывается, что нижняя часть выпуклой оболочки таких трёхмерных точек в проекции на плоскость Oxy совпадает с триангуляцией Делоне [27].

Из этого свойства очевидным образом вытекает сам алгоритм. Вначале строится трёхмерная выпуклая оболочка на множестве точек, сдвинутых с плоскости на параболоид, а затем нижние грани выпуклой оболочки преобразуются в триангуляции. На первый взгляд может показаться, что построение выпуклой оболочки в пространстве является более сложной задачей, чем построение триангуляции Делоне. Однако в настоящее время известны очень эффективные алгоритмы построения выпуклых оболочек в трёхмерном пространстве, в среднем работающие за линейно-логарифическое время.

Одним из таких является алгоритм Quickhull [23], известный также как алгоритм быстрого построения выпуклой оболочки.

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

Глава 6. Триангуляция Делоне с ограничениями

6.1. Определения

Для дальнейшего рассмотрения введём понятия *полилиния* и *регион*.

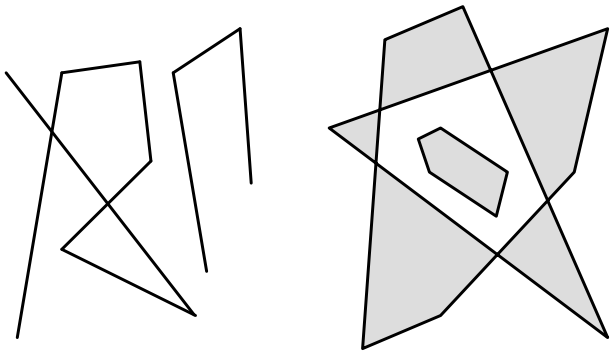
Определение 12. *Полилинией* называется фигура, состоящая из ненулевого числа ломаных (рис. 41,*а*).

Определение 13. *Регионом* называется фигура, состоящая из ненулевого числа многоугольников, причём допустимы самопересечения и пересечения различных многоугольников. При этом точки плоскости, принадлежащие k многоугольникам фигуры, считаются принадлежащими региону тогда и только тогда, когда $k \equiv 1 (\mathrm{mod}\ 2)$ (на рис. 41,*б* регион состоит из одного самопересекающегося пятиугольника и внутреннего треугольника).

Такие фигуры на практике часто используются для представления, например, линий с разрывами и многоугольников с дырками внутри.

Определение 14. *Задача построения триангуляции с ограничениями.* Пусть даны множества точек $\{P_1,...,P_k\}$, полилиний $\{Q_1,...,Q_l\}$ и регионов $\{R_1,...,R_m\}$. Необходимо на множестве точек $\{P_1,...,P_k\}$, вершин полилиний и вершин регионов построить триангуляцию таким образом, чтобы все отрезки полилиний и регионов проходили по рёбрам триангуляции. Кроме того, если множество регионов не пусто, то для всех построенных треугольников необходимо установление факта попадания в заданные регионы (при этом каждый треугольник может попасть одновременно в несколько регионов).

Определение 15. В задаче построения триангуляции с ограничениями составляющие ломаные исходных полилиний и границы исходных регионов называются *структурными линиями*.

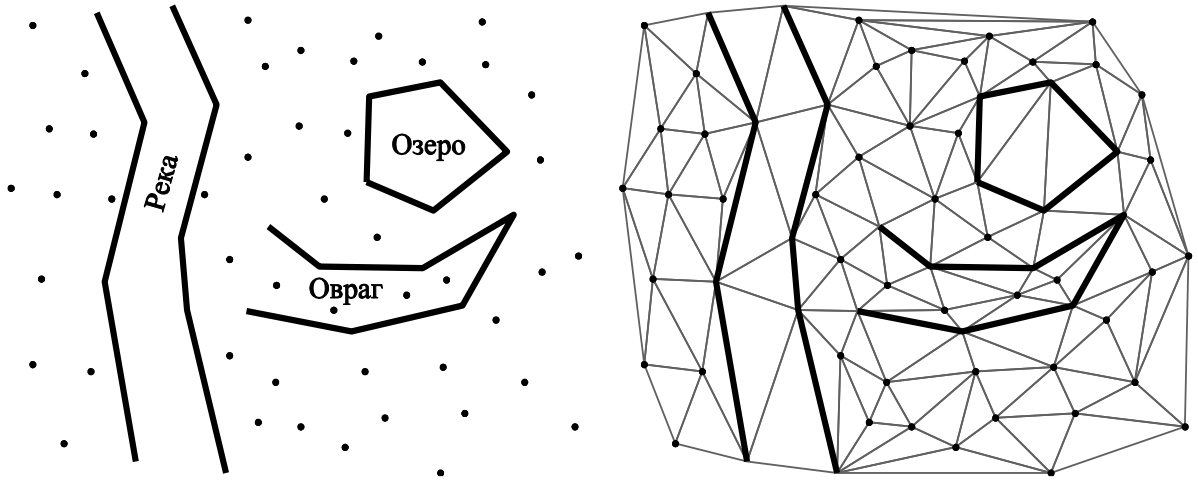


(*а*) (*б*)

Рис. 41. Примеры фигур: *а* – полилиния; *б* – регион

Определение 16. Рёбра триангуляции с ограничениями, по которым проходят исходные структурные линии, называются *структурными рёбрами* (фиксированными, неперестраиваемыми).

В такой постановке задача построения триангуляции наиболее часто используется при моделировании рельефа в геоинформационных системах. Задаваемые точки при этом определяют точки плоскости, в которых измерены высоты на поверхности, полилинии – проекции на плоскость структурных линий рельефа, а регионы – области интересов (рис. 42).



(*а*) (*б*)

Рис. 42. Пример триангуляции с ограничениями:
а – исходные данные; *б* – триангуляция

В случае, когда множества точек и полилиний пусты, а заданы только регионы, получается классическая задача построения триангуляции заданной замкнутой области.

Задача построения триангуляции с ограничениями, так же как и задача без ограничений, является неоднозначной. Среди различных видов триангуляций с ограничениями выделяют также три основных вида триангуляций: оптимальную, жадную и триангуляцию Делоне с ограничениями.

Определение 17. Триангуляция с ограничениями называется *оптимальной*, если сумма длин всех рёбер минимальна среди всех возможных триангуляций с ограничениями, построенных на тех же исходных данных.

Так как задача построения оптимальной триангуляции с ограничениями является NP-полной, то на практике она почти не применяется.

Обобщая рассмотренный в гл. 1 жадный алгоритм, получаем следующий алгоритм.

Жадный алгоритм построения триангуляции с ограничениями.

Шаг 1. Во множество исходных точек помещаются все вершины заданных полилиний и регионов, а также генерируется список всех возможных отрезков, соединяющих пары исходных точек, и он сортируется по длинам отрезков.

Шаг 2. Выполняется вставка отрезков в триангуляцию от более коротких до длинных. Вначале вставляются все отрезки, являющиеся частями исходных полилиний и регионов, а затем – остальные отрезки. Если отрезок не пересекается с другими, ранее вставленными отрезками, то он вставляется, иначе он отбрасывается. *Конец алгоритма.*

Заметим, что если все возможные отрезки имеют разную длину, то результат работы этого алгоритма однозначен, иначе он зависит от порядка вставки отрезков одинаковой длины.

Условием правильной работы жадного алгоритма является отсутствие среди исходных полилиний и регионов взаимных пересечений отрезков. Если таковые имеются, то от них надо избавиться до начала работы жадного алгоритма разбиением этих отрезков на части.

Определение 18. Триангуляция с ограничениями называется *жадной*, если она построена жадным алгоритмом.

Трудоёмкость работы жадного алгоритма при некоторых его улучшениях составляет $O(N^2 \log N)$ [37], не учитывая предварительного этапа удаления пересекающихся отрезков. При учёте предварительного этапа сложность получается $O(N^4 \log N)$, так как в результате разбиения отрезков на части может получиться $O(N^2)$ отрезков. В связи со столь большой трудоёмкостью на практике такой алгоритм применяется редко.

Определение 19. Триангуляция заданного набора точек будет называться *триангуляцией Делоне с ограничениями*, если условие Делоне выполняется для любой пары смежных треугольников, которые не разделяются структурными рёбрами.

Для построения триангуляции Делоне с ограничениями могут быть обобщены некоторые из приведенных выше алгоритмов. Наиболее хорошо для такого обобщения подходят итеративные алгоритмы триангуляции.

Алгоритмы триангуляции слиянием не подходят, так как не всегда возможно деление множества исходных объектов на непересекающиеся части (например, когда есть большой регион, охватывающий все остальные объекты).

Алгоритмы прямого построения триангуляции подходят больше, но в них необходимо добавить в процедуру поиска очередного соседа Делоне проверку пересечения со структурными линиями.

Большинство эффективных двухпроходных алгоритмов построения триангуляции в данном случае использовать нельзя, так как они либо яв-

ляются неприменимыми алгоритмами слияния, либо строят огромное количество узких вытянутых треугольников, которые почти всегда перестраиваются, а поэтому их применение неэффективно.

Для триангуляции с ограничениями наиболее удобно использовать структуры данных, представляющих в явном виде рёбра, так как для рёбер необходимо хранить дополнительную информацию о том, являются ли они структурными. Поэтому структуры «Узлы с соседями» и «Узлы и треугольники» неприменимы. Из оставшихся наиболее применяемой является структура «Узлы, простые рёбра и треугольники» как компромисс между расходом памяти и удобством применения. Дополнительным её достоинством является возможность простого эволюционного перехода к ней от итеративного алгоритма триангуляции Делоне без ограничений, использующего компактную структуру «Узлы и треугольники».

6.2. Цепной алгоритм построения триангуляции с ограничениями

Один из первых эффективных алгоритмов построения триангуляции с ограничениями основан на процедуре регуляризации планарного графа и триангуляции монотонных многоугольников [12]. Трудоёмкость этого алгоритма составляет $O(N \log N)$, где N – количество исходных отрезков.

Исходными данными для *цепного алгоритма* является множество непересекающихся отрезков на плоскости, по сути образующих планарный граф. Если в триангуляцию необходимо поместить также отдельные точки, то их следует добавить уже после работы данного алгоритма, например итеративным способом.

Цепной алгоритм построения триангуляции с ограничениями.

Шаг 1. Из множества исходных структурных отрезков формируем связанный планарный граф (рис. 43,*а*).

Шаг 2. Выполняется *регуляризация графа*, т.е. добавляются новые рёбра, не пересекающие другие, так что каждая вершина графа становится смежной хотя бы с одной вершиной выше неё и одной ниже. Регуляризация выполняется в два прохода с помощью вертикального плоского заметания [12]. В первом проходе снизу вверх последовательно находятся все вершины, из которых не выходят рёбра, ведущие вверх. Например, на рис. 43,*б* такой является вершина *B*. Проводя горизонтальную линию, обнаруживаем ближайшие пересекаемые ею слева и справа рёбра графа *AD* и *EF*. Затем в четырёхугольнике *DEHG* находим самую низкую вершину и проводим в неё ребро из *B*. Аналогично выполняется второй проход сверху вниз (рис. 43,*в*). В результате работы этого шага каждая область планарного графа становится монотонным многоугольником.

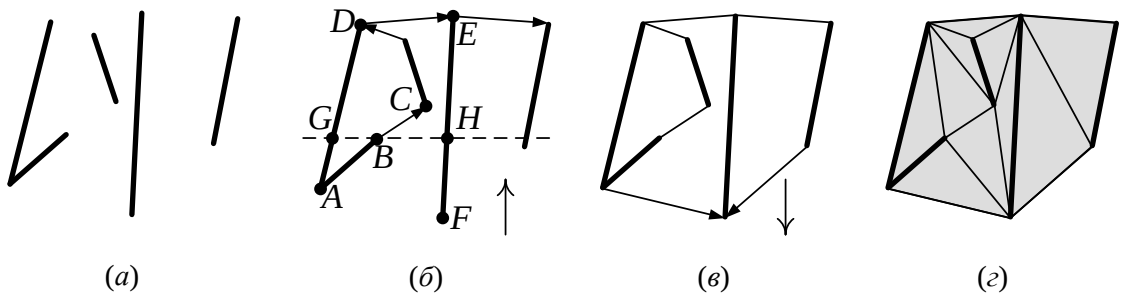


Рис. 43. Схема работы цепного алгоритма триангуляции:
а – исходные отрезки; *б* – проход снизу вверх регуляризации графа;
в – проход сверху вниз; *г* – триангуляция монотонных многоугольников

Шаг 3. Каждую область графа необходимо разбить на треугольники. Для этого можно воспользоваться алгоритмом невыпуклого слияния двух триангуляций (рис. 43,*г*). *Конец алгоритма.*

Для реализации цепного алгоритма триангуляции с ограничениями лучше всего использовать структуры данных, в которых рёбра представляются в явном виде, например «Двойные рёбра» (см. п. 1.3.3) или «Узлы, рёбра и треугольники» (см. п. 1.3.5).

Недостатком цепного алгоритма является то, что о форме получаемой триангуляции ничего заранее сказать нельзя. Это не оптимальная триангуляция, не жадная и не триангуляция Делоне с ограничениями. В цепном алгоритме могут получаться очень длинные вытянутые треугольники.

Для улучшения качества полученной триангуляции можно проверить условия Делоне для всех пар смежных треугольников, не разделенных структурным ребром, и при необходимости произвести перестроения. В результате будет получена триангуляция Делоне с ограничениями.

6.3. Итеративный алгоритм построения триангуляции Делоне с ограничениями

За основу *итеративного алгоритма построения триангуляции Делоне с ограничениям* может быть взят любой итеративный алгоритм построения обычной триангуляции Делоне, но наиболее удобно здесь использовать алгоритм динамического кэширования (см. п. 2.3.2), так как после окончания его работы будет дополнительно создана структура кэша, которая может быть использована для последующей быстрой локализации точек в триангуляции.

Итеративный алгоритм построения триангуляции Делоне с ограничениями.

Шаг 1. Вначале выполняется построение обычной триангуляции Делоне по всем исходным точкам и входящим в состав структурных линий.

Шаг 2. Выполняется вставка отрезков структурных линий в триангуляцию. При этом на первом этапе концы этих отрезков уже вставлены в триангуляцию как узлы.

Шаг 3. Выполняется классификация всех треугольников триангуляции по попаданию в заданные регионы. *Конец алгоритма.*

Второй этап этого алгоритма на практике может быть реализован поразному. Рассмотрим различные варианты процедуры вставки отрезков.

6.3.1. Вставка структурных отрезков «Строй, разбивая»

Алгоритм вставки структурных отрезков *«Строй, разбивая»* является наиболее простым в реализации и устойчивым в работе.

В нем необходимо, последовательно переходя по треугольникам вдоль вставляемого отрезка, находить точки его пересечения с рёбрами триангуляции (рис. 44,*а*). В этих точках пересечения нужно поставить новые узлы триангуляции, разбив существующие рёбра и треугольники на части (рис. 44,*б*). После этого все вновь построенные треугольники должны быть проверены на выполнение условия Делоне и при необходимости перестроены, не затрагивая фиксированных рёбер.

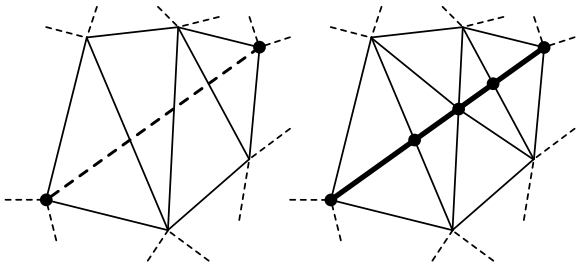


Рис. 44. Вставка структурных отрезков «Строй, разбивая»

В некоторых случаях недостатком данного алгоритма вставки может быть создание большого числа дополнительных узлов и рёбер триангуляции. В то же время в других случаях этот недостаток становится преимуществом, не позволяя образовываться длинным узким треугольникам, что особенно ценится при моделировании рельефа.

Другое преимущество этого алгоритма вставки по сравнению с последующими проявляется при попытке вставки структурного отрезка в триангуляцию, в которой среди пересекаемых им рёбер есть фиксированные. Такие рёбра, как и все остальные, просто разбиваются на две части.

6.3.2. Вставка структурных отрезков «Удаляй и строй»

В алгоритме вставки структурных отрезков «Удаляй и строй» необходимо, последовательно переходя по треугольникам вдоль вставляемого отрезка, найти все пересекаемые треугольники (рис. 45,*а*) и удалить их из триангуляции (рис. 45,*б*). При этом в триангуляции образуется дырка в виде некоторого многоугольника. После этого в триангуляцию вставляется структурный отрезок, делящий многоугольник-дырку на две части – левую и правую, которые затем заполняются треугольниками (рис. 45,*в*).

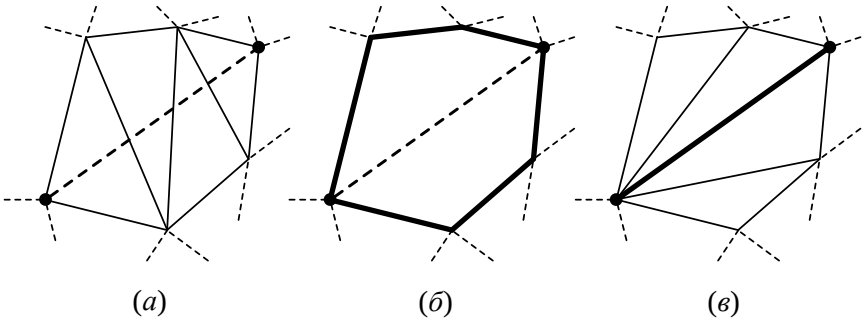


Рис. 45. Вставка структурных отрезков «Удаляй и строй»

Заполнение левой и правой частей дырки треугольниками можно произвести, проходя вдоль её границы и анализируя тройки последовательных точек границы. Если на месте этой тройки точек можно построить треугольник, то он строится и граница укорачивается. Такой цикл идет, пока количество точек в левой и правой границах больше двух.

После этого все вновь построенные треугольники должны быть проверены на выполнение условия Делоне и при необходимости перестроены, не затрагивая фиксированных рёбер.

Отдельным представляется случай, когда при вставке среди множества пересекаемых рёбер находятся фиксированные рёбра. Во избежание такой ситуации можно заранее еще до первого этапа работы алгоритма построения триангуляции Делоне с ограничениями найти все точки пересечения всех структурных отрезков и разбить этими точками отрезки на части. Такую операцию можно выполнить, например, с помощью алгоритма заметания плоскости [12].

Другим вариантом учета пересекаемых фиксированных рёбер является следующий алгоритм. Пусть при вставке очередного структурного отрезка AB обнаружено пересечение с некоторым фиксированным ребром CD в точке S (рис. 46,*а*). Тогда надо разбить пересекаемое ребро CD на две части CS и SD, также разбив смежные треугольники ΔCDE и ΔCDF на две части ΔCSE, ΔSDE и ΔCSF, ΔSDF соответственно (рис. 46,*б*). После этого задача вставки исходного отрезка AB сводится к двум вставкам рёбер AS и SB (рис. 46,*в*).

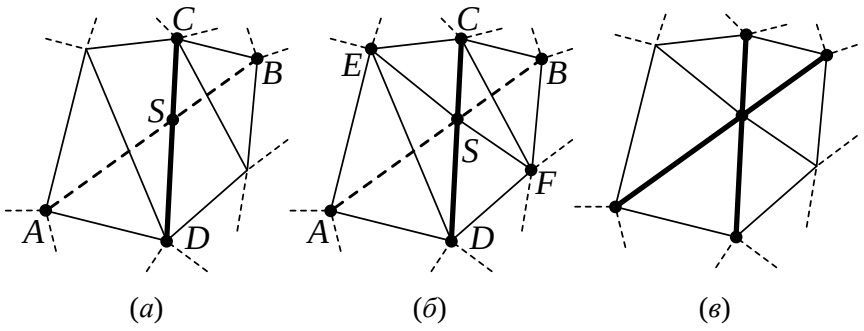


Рис. 46. Пересечение вставляемого структурного отрезка
с ранее вставленным фиксированным ребром

Теперь остановимся на вопросе влияния структуры данных на реализацию алгоритма вставки «Удаляй и строй». Наиболее сложной частью здесь является удаление треугольников, временное запоминание границы области удаленных треугольников и последующее её заполнение. Наиболее просто эта задача выполняется на структуре «Узлы, рёбра и треугольники», где просто запоминается список граничных рёбер.

На структуре «Узлы, простые рёбра и треугольники», часто используемой при построении триангуляции с ограничениями, ребро представляется в неявном виде как треугольник и номер образующего ребра, так как в описании ребра отсутствуют ссылки на смежные треугольники.

В данном алгоритме вставки при удалении треугольников возможны ситуации, когда, удаляя треугольники, необходимо сохранить некоторые образующие их фиксированные рёбра. Например, на рис. 47,*а* необходимо вставить структурный отрезок AB, при этом вблизи находится ранее вставленное фиксированное ребро KL. После удаления всех пересекаемых треугольников должно остаться ребро KL (рис. 47,*б*), затем необходимо заполнить треугольниками многоугольник $ABKLK$ слева от ребра AB (рис. 47,*в*).

Отметим также возможность возникновения ситуации, когда висячее фиксированное ребро вообще не будет связано с границей области удаленных треугольников (рис. 48,*а*,*б*). При этом задача заполнения области тре-

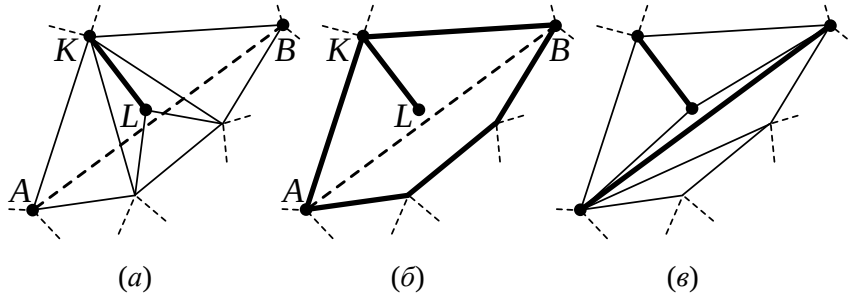


Рис. 47. Сохранение фиксированного ребра при удалении
треугольников в алгоритме вставки «Удаляй и строй»

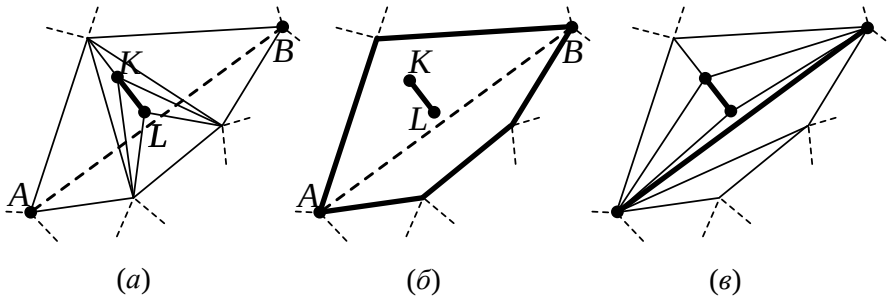


Рис. 48. Фиксированное ребро, не связанное с границей области удалённых треугольников

угольниками, безусловно, существенно усложняется. По сложности сама она в целом эквивалентна задаче построения триангуляции Делоне с ограничениями внутри заданного региона. Для ее решения нужно временно удалить мешающее фиксированное ребро *KL* из триангуляции, заполнить очищенную область треугольниками, а затем повторно вставить ранее удаленные фиксированные рёбра (рис. 48,*в*).

6.3.3. Вставка структурных отрезков «Перестраивай и строй»

Основная идея алгоритма вставки структурных отрезков «Перестраивай и строй» заключается в попытке уменьшения количества пересекаемых рёбер, перестраивая пары соседних треугольников до тех пор, пока не останется ни одного пересекаемого треугольника (рис. 49,*а–в*), т.е. вставляемый структурный отрезок не станет уже существующим ребром триангуляции (рис. 49,*г*).

Самым главным недостатком этого алгоритма является возможность образования тупиковых ситуаций, когда дальнейшие перестроения пар соседних треугольников не уменьшают числа пересекаемых треугольников. Пример такой ситуации приведен на рис. 50,*а*. Для разрешения ситуаций, когда нельзя выполнить перестроение треугольников с уменьшением числа

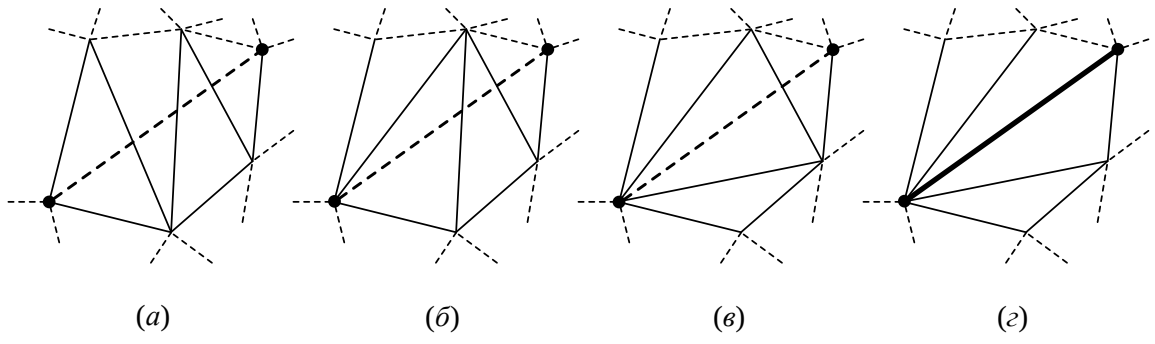


Рис. 49. Простое перестроение треугольников в алгоритме вставки структурных отрезков «Перестраивай и строй»

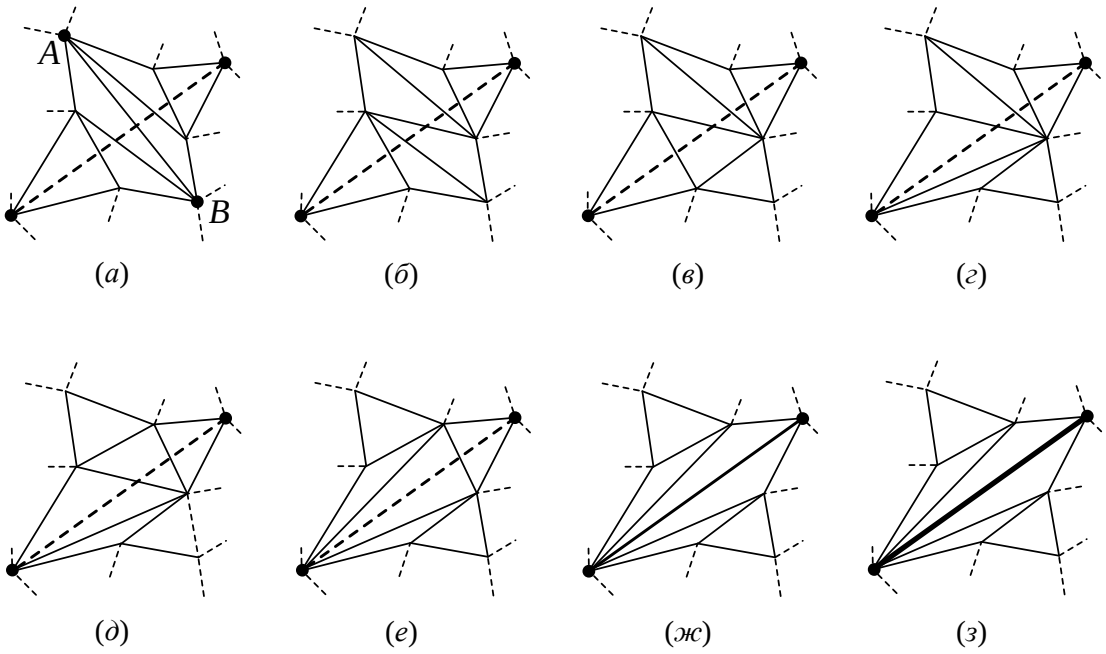


Рис. 50. Модифицированное перестроение треугольников в алгоритме вставки структурных отрезков «Перестраивай и строй»

пересечений, необходимо перестраивать те пересекаемые пары треугольников, чья общая сторона образует максимальный угол со вставляемым структурным ребром. И так до тех пор, пока не образуются пары смежных треугольников, чьи перестроения уменьшат число пересечений рёбер. На рис. 50,*а* таким ребром (среди тех, чьи смежные треугольники можно перестроить) является AB, поэтому оно и перестраивается (рис. 50,*б*). После этого выполняются обычные перестроения с уменьшением пересечений (рис. 50,*в–з*).

Случай, когда вставляемый структурный отрезок пересекает уже ранее вставленное фиксированное ребро, должен обрабатываться так же, как и в предыдущем алгоритме вставки «Удаляй и строй».

Трудоёмкости всех трёх рассмотренных алгоритмов вставки составляют в худшем случае $O(M^2)$, где M – количество узлов в триангуляции после завершения работы алгоритма триангуляции с ограничениями. Заметим, что при большом количестве взаимных пересечений структурных линий эта оценка составляет $O(N^4)$, где N – количество исходных точек и вершин исходных структурных линий.

Оценка трудоёмкости в среднем очень сильно зависит от распределения структурных линий. Если их количество невелико и они мало пересекаются между собой, то общая оценка трудоёмкости может составить $O(N)$.

6.4. Классификация треугольников

Теперь рассмотрим третий этап построения триангуляции с ограничениями – задачу классификации полученных треугольников триангуляции по признаку их попадания внутрь заданных регионов.

В простейшем случае можно для каждого отдельно взятого треугольника выбрать любую точку внутри него и проверить её на попадание во все заданные регионы. Трудоёмкость такой операции составляет $O(M)$, где M – число точек в границе региона. Тогда общая трудоёмкость алгоритма классификации составит $T(N,M)=O(NM)$.

Можно поступить по-другому [15]. Пусть для каждого треугольника необходимо выставить признак $C_i=1$, если он попадает внутрь какого-либо региона, и $C_i=0$, если нет. Предположим, что при вставке структурных линий, принадлежащих границам регионов, для каждого фиксированного ребра отмечалось, к какому региону он относится. При этом возможно, что одно и то же фиксированное ребро может относиться к нескольким регионам одновременно. Кроме того, если граница некоторого региона проходит через какое-то ребро многократно, то это количество прохождений также должно быть отмечено. В будущем при рассмотрении попадания треугольников в регион мы будем игнорировать рёбра с четным количеством прохождений в соответствии с определением региона.

Алгоритм определения попадания треугольников в заданный регион

Пусть дана триангуляция и для каждого фиксированного ребра отмечено, сколько раз граница данного региона проходит через ребро.

Шаг 1. Для каждого треугольника обнулить признак попадания внутрь региона $C_i:=0$.

Шаг 2. Отмечаем $R_i=1$ все фиксированные рёбра, по которым граница региона проходит нечетное число раз. Остальные рёбра отмечаем $R_i:=0$.

Шаг 3. Для каждого ребра с $R_i:=1$ проверяем два смежных треугольника T_{i_1} и T_{i_2}. Если $C_{i_1}=0$ и $C_{i_2}=0$, то определяем, попадает ли треугольник T_{i_1} внутрь региона (простая проверка попадания центра треугольника в регион). Если попадает, то выполняем шаг 4, начиная с треугольника T_{i_1}, иначе выполняем шаг 4, начиная с треугольника T_{i_2} (рис. 51,*а*).

Шаг 4. Выполняем поиск всех треугольников в заданной замкнутой области алгоритмом растровой заливки с затравкой [13]. При этом границей области заливки являются рёбра с $R_i:=1$. Каждый найденный треугольник отмечаем $C_j:=1$ (рис. 51,*б*). *Конец алгоритма.*

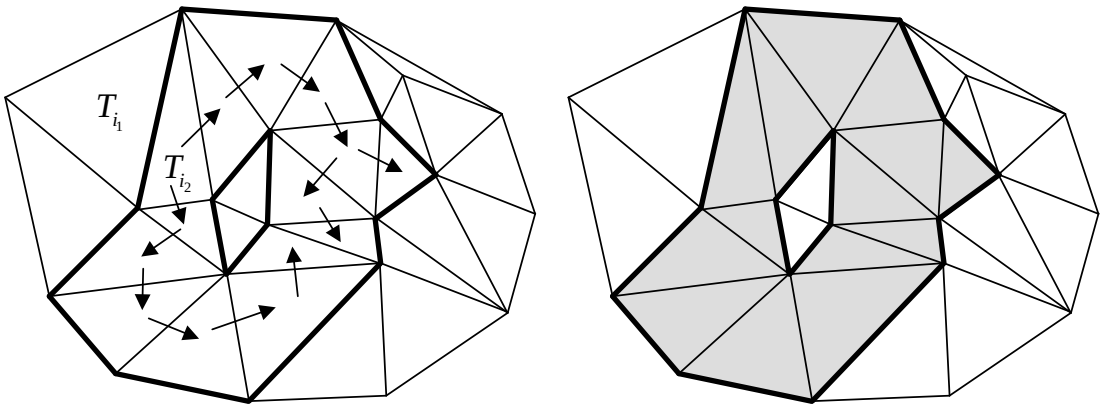


Рис. 51. Классификация треугольников

Трудоёмкость первого и второго шага данного алгоритма составляет в сумме $O(N)$. Сложность шага 3 равна $O(MR)$, где R – количество рёбер в границе, а M – количество несвязанных областей в регионе. Трудоёмкость шага 4 составляет $O(T)$, где T – общее количество треугольников внутри региона. Таким образом, так как $O(T) = O(N)$, то общая трудоёмкость данного алгоритма классификации треугольников составляет $O(N + MR + T) = O(N + MR)$.

Если в нашей триангуляции присутствует K регионов, то данный алгоритм нужно применить K раз и общая трудоёмкость классификации составит $O(KN + KMR)$. Рассмотрим модификацию этого алгоритма, позволяющую более эффективно выполнять классификацию нескольких регионов.

Алгоритм классификации треугольников по регионам

Пусть дана триангуляция и для каждого фиксированного ребра отмечено, сколько раз граница какого региона проходит через ребро. По результатам работы этого алгоритма для каждого треугольника будет получен список регионов, к которым он принадлежит.

Шаг 1. Для каждого треугольника обнуляем список регионов $S_i := \varnothing$. Все ребра триангуляции отмечаем $R_i := 0$.

Шаг 2. Для каждого региона P_k формируем список фиксированных рёбер, образующих его границу. Причем в список включаем только те рёбра, по которым граница региона проходит по ребру нечетное число раз.

Шаг 3. Выполняем шаг 4 в цикле для всех регионов P_k, $k = \overline{1, K}$.

Шаг 4. Отмечаем $R_i := k$ все фиксированные рёбра, принадлежащие текущему региону P_k. Далее для каждого отмеченного ребра проверяем два смежных треугольника T_{i_1} и T_{i_2}. Если $P_k \notin S_{i_1}$ и $P_k \notin S_{i_2}$, то определяем,

попадает ли треугольник T_{i_1} внутрь региона (простая проверка попадания центра треугольника в регион). Если попадает, то выполняем шаг 5, начиная с треугольника T_{i_1}, иначе выполняем шаг 4, начиная с треугольника T_{i_2} (см. рис. 51,*а*).

Шаг 5. Выполняем поиск всех треугольников в заданной замкнутой области алгоритмом растровой заливки с затравкой. При этом границей области заливки являются рёбра с $R_i := k$. Для каждого найденного треугольника T_i включаем в список S_i регион P_k. *Конец алгоритма.*

В этом алгоритме трудоемкость первого шага составляет $O(N)$, второго – $O(N + \sum_{k=1}^{K} R_k)$, где R_k – количество рёбер, составляющих границу региона P_k. Трудоемкость третьего и четвертого шагов составляет $O(R_k + M_k \cdot R_k + T_k) = = O(M_k \cdot R_k + T_k)$, где M_k – количество несвязанных областей в регионе, а T_k – общее число треугольников внутри региона P_k.

Таким образом, общая трудоёмкость алгоритма составляет $O(N + \sum_{k=1}^{K} R_k) + \sum_{k=1}^{K} O(M_k \cdot R_k + T_k) = O(N + \sum_{k=1}^{K} M_k \cdot R_k + \sum_{k=1}^{K} T_k)$. При условии, что регионы не пересекаются между собой, а граница каждого региона не проходит дважды через одно и то же ребро, получаем общую трудоёмкость, равную $O(N + \sum_{k=1}^{K} M_k \cdot R_k) = O(N)$.

В заключение этого раздела обратим внимание, что для сокращения времени классификации необходимо уменьшать количество узлов в триангуляции с ограничениями. В связи с этим отметим, что применение алгоритма вставки структурных отрезков «Строй, разбивая» не желательно, так как он порождает значительное количество дополнительных узлов и рёбер триангуляции, а поэтому существенно увеличивает время последующей классификации.

6.5. Выделение регионов из триангуляции

Задача выделения регионов из триангуляции является обратной по отношению к предыдущей – задаче классификации. Она используется при решении различных задач пространственного анализа на плоскости и моделировании поверхностей, описываемых в гл. 8 и 9.

Определение 20. Пусть дана некоторая триангуляцию и каждому треугольнику в ней сопоставлен некоторый код C_i. В *задаче выделения регионов из триангуляции* необходимо объединить все треугольники с одинаковыми кодами в регионы (рис. 52).

Для решения этой задачи можно использовать следующий алгоритм, предложенный в [15].

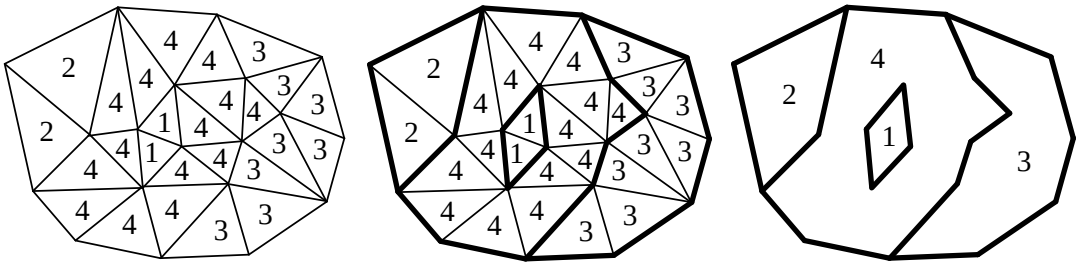


Рис. 52. Выделение регионов из триангуляции: *а* – исходные классифицированные треугольники; *б* – выделение рёбер; *в* – объединение рёбер в регионы

Алгоритм выделения регионов из триангуляции.

Шаг 1. Для каждого треугольника T_i установить признак D_i в 0. Обнулить список контуров готовых регионов $R_i = \varnothing, i = \overline{1,K}$, где количество разных кодов C_i, т.е. искомых регионов.

Шаг 2. Найти все рёбра, которые войдут в результирующие регионы, т.е. рёбра, имеющие смежные треугольники с разными кодами (рис. 52,*б*).

Шаг 3. Для каждого треугольника T_i с $D_i = 0$ выполнить шаг 4 с текущим треугольником T_i.

Шаг 4. Начиная с текущего треугольника T_i, методом затравки (как в алгоритме заливки растровой области методом затравки) построить список смежных треугольников S с кодом C_i. Для всех треугольников в S установить $D_i = 1$. Составить список B рёбер треугольников, не разделяющих два треугольника из этого списка. Взять произвольное ребро из списка B и, используя структуру триангуляции, обойти по рёбрам из списка B контур. Пройденные рёбра удалить из списка B. Пока список B не пуст, выполнять обходы для поиска оставшихся контуров. Выделенные контуры добавляем в список R_{C_i} *Конец алгоритма.*

Трудоёмкость работы шага 4 алгоритма определяется из трудоёмкости алгоритмов затравки и обходов. Алгоритм затравки, выделив t_i треугольников, работает время $O(t_i)$. Эти треугольники имеют $O(t_i)$ рёбер, и их обход займёт время $O(t_i)$. Поэтому общая сложность шага 4 составляет $O(t_i)$.

Тогда общая трудоёмкость всего алгоритма состоит из $O(N)$ на шагах 1–2 и $\sum_{i=1}^{K} O(t_i)$ на шагах 3–4. А так как $\sum_{i=1}^{K} t_i = M$, где M – общее количество треугольников в триангуляции, то сложность алгоритма выделения регионов составляет $O(M) + \sum_{i=1}^{K} O(t_i) = O(M) = O(N)$.

Глава 7. Оптимальная триангуляция

7.1. Точный алгоритм

Как сказано в п. 1.1, под *оптимальной* триангуляцией обычно понимают триангуляцию с минимальной суммарной длиной ребер.

В работе [60] было доказано, что задача построения такой триангуляции является NP-трудной, хотя точного доказательства данного факта в настоящее время не получено. Поэтому для большинства реальных задач существующие алгоритмы построения оптимальной триангуляции неприемлемы ввиду слишком высокой трудоёмкости.

По сути, любой точный алгоритм построения оптимальной триангуляции вынужден реализовать полный перебор всех возможных триангуляций, что сразу же приводит к экспоненциальной трудоёмкости алгоритма. Тем не менее приведенный ниже алгоритм позволяет несколько сократить общее время работы по сравнению с полным перебором всех возможных триангуляций, при этом оставаясь алгоритмом с экспоненциальной трудоёмкостью в худшем случае.

Общая схема работы алгоритма напоминает жадный алгоритм. Вначале строится выпуклая оболочка всех исходных точек. Затем алгоритм формирует полный список всех возможных ребер триангуляции, не считая тех, которые уже вошли в выпуклую оболочку. Далее запускается рекурсивная процедура вставки различных рёбер из множества ещё не вставленных ребер. При вставке необходимо проверить, чтобы очередное вставленное ребро не пересекалось ни с одним из ранее вставленных.

Работа рекурсивной процедуры должна останавливаться в двух случаях. Во-первых, когда все ребра триангуляции будут построены, т.е. при достижении глубины рекурсии, равной числу внутренних рёбер в триангуляции $3 \cdot N - 2 \cdot C - 3$, где C – число узлов на внешней границе триангуляции. Во-вторых, когда текущий частично построенный граф триангуляции уже имеет длину больше, чем у лучшей предыдущей найденной триангуляции. Этот алгоритм можно несколько улучшить, если вначале построить триангуляцию Делоне, тем самым, получив грубую верхнюю оценку суммарной длины рёбер оптимальной триангуляции. Это позволит не проверять многочисленные варианты, дающие результаты заведомо худшие, чем триангуляция Делоне.

Прежде чем формально описать алгоритм, отметим, что он оперирует только рёбрами и узлами. При этом треугольники как объекты в данном алгоритме не фигурируют. Именно поэтому для точного алгоритма наиболее предпочтительным является использование структур данных без треугольников, в частности структуры «Узлы и рёбра» (см. п. 1.3.2).

Алгоритм точного построения оптимальной триангуляции.

Шаг 1. На заданном множестве точек строим триангуляцию Делоне и определяем L – суммарную длину её рёбер. Эта величина будет первой верхней оценкой длины рёбер оптимальной триангуляции, которая должна постепенно улучшаться по ходу работы алгоритма. Построенную триангуляцию Делоне запоминаем в качестве лучшего найденного решения T.

Шаг 2. Строим выпуклую оболочку C на множестве исходных точек. Рёбра, образующие выпуклую оболочку, войдут в оптимальную триангуляцию, поэтому помещаем рёбра выпуклой оболочки в результирующее множество рёбер R.

Шаг 3. Генерируем множество U всех возможных рёбер триангуляции, которые далее в рекурсивной процедуре будем по очереди добавлять в R: $U = \{(N_i, N_j)\}, i \neq j, (N_i, N_j) \notin C$. Множество U сортируем по возрастанию длин рёбер. Размер множества U будет равен $N(N-1) - c$, где c – количество рёбер, вошедших в выпуклую оболочку.

Шаг 4. Вызываем рекурсивную процедуру ***Поиск(R, L, U)***.

Процедура ***Поиск*** $(\overline{R}, \overline{L}, \overline{U})$. Аргументы имеют смысл:

$\overline{R}$ – граф частично формированной триангуляции;

$\overline{L}$ – суммарная длина рёбер графа $\overline{R}$;

$\overline{U}$ – рёбра, которые можно добавлять в $\overline{R}$.

Шаг 1. Если $\overline{L} >= L$, то выходим из процедуры.

Шаг 2. Если количество рёбер $\overline{R}$ равно числу рёбер в T, значит, найдено лучшее решение, которое следует запомнить: $T := \overline{R}$, $L := \overline{L}$, после чего выходим из процедуры.

Шаг 3. В цикле проверяем каждое ребро r в $\overline{U}$. Если ребро r пересекает некоторое в $\overline{R}$, то удаляем r из $\overline{U}$. Иначе выполняем следующее:

Шаг 3.1. Удаляем r из $\overline{U}$, добавляем r в $\overline{R}$. Увеличиваем достигнутую суммарную длину рёбер на длину ребра r: $\overline{L} := \overline{L} + l_r$.

Шаг 3.2. Рекурсивно вызываем ***Поиск*** $(\overline{R}, \overline{L}, \overline{U})$.

Шаг 3.3. Удаляем r из $\overline{R}$, добавляем r в $\overline{U}$. Уменьшаем достигнутую суммарную длину рёбер: $\overline{L} := \overline{L} - l_r$.

Конец процедуры ***Поиск***.

Конец алгоритма.

Как было сказано выше, приведенный точный алгоритм построения оптимальной триангуляции обладает экспоненциальной трудоёмкостью, а

потому на существующей в настоящее время вычислительной технике в разумные сроки способен просчитать задачу с не более чем 30–50 узлами. Для большего числа узлов приходится применять другие виды триангуляций, дающие приближённые результаты.

В заключение данного раздела рассмотрим ещё один распространённый вид триангуляции: *оптимальную триангуляцию с ограничениями*, которая аналогично триангуляции Делоне с ограничениями требует, чтобы некоторые рёбра обязательно вошли в триангуляцию (см. п. 6.1). При этом оптимальная триангуляция с ограничениями обладает минимальной суммой рёбер среди всех возможных триангуляций, которые содержат заранее заданные рёбра.

Точный алгоритм построения оптимальной триангуляции с ограничениями практически совпадает с приведенным выше алгоритмом, за исключением шага 2, на котором мы должны заранее поместить исходно заданные рёбра-ограничения в результирующее множество рёбер *R*. Трудоёмкость такого модифицированного алгоритма, очевидно, также будет экспоненциальной, поэтому для практики нужны приближённые алгоритмы построения оптимальной триангуляции с ограничениями. В этой роли наиболее часто используется *жадная триангуляция с ограничениями* и *триангуляция Делоне с ограничениями* (см. п. 6.1).

7.2. Квазижадная триангуляция

Из-за крайне высокой (экспоненциальной) трудоёмкости задачи построения оптимальной триангуляции на практике применяются различные приближённые алгоритмы. Так, в качестве приближений оптимальной наиболее часто используются жадная триангуляция и иногда триангуляция Делоне. Однако в работе [42] показано, что суммарная длина рёбер в триангуляции Делоне может быть по крайней мере в $\Omega(N)$ раз больше, чем в оптимальной триангуляции. Несколько лучше ситуация с жадной триангуляцией. В [50] доказано, что длина её рёбер может быть в $O(\sqrt{N})$ раз больше, чем в оптимальной триангуляции, но не больше. Кстати, из этих фактов не следует, что жадная триангуляция всегда лучше триангуляции Делоне в смысле оптимальности. В статье [52] показано, что длина рёбер в жадной триангуляции может быть в $\Omega(\sqrt{N})$ раз больше, чем в триангуляции Делоне.

Приведенные выше оценки верны для произвольных наборов точек. В случае же если узлы триангуляции подчиняются равномерному распределению, то суммарная длина рёбер жадной триангуляции и триангуляции Делоне отличается от оптимальной не более чем в константу раз [26,53].

Другой хороший приближённый алгоритм представлен в работе [62]. Он находит триангуляцию, чья длина рёбер отличается от оптимальной

триангуляции не более чем в $O(\log N)$ раз. При этом трудоёмкость алгоритма составляет $O(N^2 \log N)$.

Лучшим же известным приближением оптимальной триангуляции является *квазижадная триангуляция*, описанная в [51], она отличается по длине не более чем в константу раз от оптимальной. Важное достоинство этой триангуляции в том, что она может быть построена всего за время $O(N^2 \log N)$.

Алгоритм построения квазижадной триангуляции незначительно отличается от обычной жадной триангуляции. Рассмотрим его.

Алгоритм построения квазижадной триангуляции [51].

В алгоритме термином *диагональ* обозначается любое ребро полного графа, которое ещё не вошло в строящуюся триангуляцию и не пересекает ни одного ранее построенного ребра.

Шаг 1. Вначале в триангуляцию помещаем все рёбра, входящие в выпуклую оболочку исходных точек.

Шаг 2. В цикле добавляем рёбра в частично готовый граф G, пока не получим триангуляцию. Для этого находим в частичном графе G диагональ минимальной длины (v_1, u_1). Если все следующие 6 условий верны, то добавляем в граф G ребро (v_0, u_0), иначе ребро (v_1, u_1) (рис. 53):

1. Диагональ (v_1, u_1) образует пустой (т.е. не содержащий внутри себя никаких узлов и рёбер графа G) треугольник (v_1, u_0, u_1) с двумя рёбрами (v_1, u_0) и (u_0, u_1), уже входящими в G.
2. Существует некоторая диагональ – отрезок (v_0, u_0), пересекающий отрезок (v_1, u_1) и образующий пустой треугольник (v_0, v_1, u_0) с двумя рёбрами (v_0, v_1) и (v_1, u_0), уже входящими в G.
3. В треугольнике (v_1, u_0, u_1) угол $\angle v_1, u_0, u_1 > 135°$.
4. $|v_0, u_0| < 1{,}1|v_1, u_1|$, где $|...|$ – длина соответствующего ребра.

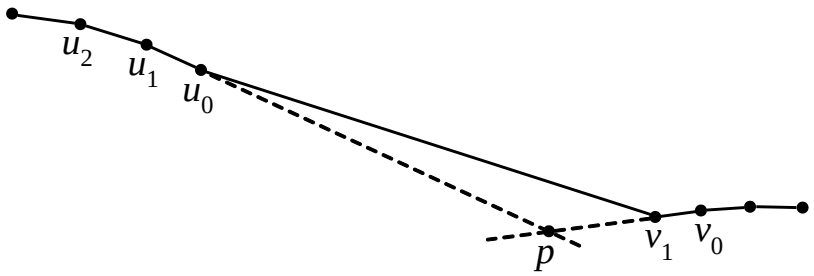


Рис. 53. Пример конфигурации точек, при которой выполняются все 6 условий на шаге 2 алгоритма построения квазижадной триангуляции

5. $|v_0, p| < 0{,}5|u_0, p|$, где p – это точка пересечения прямых (v_0, v_1) и (u_0, u_1).
6. Существует некоторое такое ребро (u_1, u_2) в G, что (v_1, u_0, u_1, u_2) образует пустой четырёхугольник с $\angle u_0, u_1, u_2 > 180°$.

Конец алгоритма.

Формально квазижадной триангуляцией называется такая триангуляция, которая построена приведенным алгоритмом.

7.3. Алгоритмы с локальным перестроением треугольников

Приведенный выше алгоритм построения квазижадной триангуляции имеет наилучшую из известных теоретических оценок отличия от оптимальной триангуляции – не более чем в константу раз. Однако эта константа достаточно велика и пока ещё точно не оценена.

Именно поэтому на практике используются различные эвристические алгоритмы, для которых теоретические оценки отличия от оптимальной триангуляции не получены, но которые в среднем показывают лучшие результаты, чем другие вышеупомянутые виды триангуляций.

Один из таких алгоритмов основан на последовательном улучшении триангуляции Делоне с помощью локальных перестроений [8].

Самый простой *алгоритм с локальными перестроениями* заключается в перестроении всех возможных пар соседних треугольников, если такое перестроение может привести к уменьшению суммарной длины рёбер триангуляции (рис. 54). Достоинством этого алгоритма является низкая трудоёмкость, составляющая в среднем только $O(N)$.

В более сложном *алгоритме с локально жадными перестроениями* для каждого узла триангуляции находится двойная окрестность (рис. 55), все рёбра в данной окрестности удаляются, а затем в этой окрестности заново генерируются рёбра жадным алгоритмом. Трудоёмкость данного алгоритма также составляет $O(N)$, однако константа пропорциональности существенно больше, чем в предыдущем алгоритме с перестроением пар соседних треугольников.

В статье [8] приведены результаты экспериментального сравнения работы различных приближенных алгоритмов построения оптимальной триангуляции. В частности, для равномерного распределения показано, что триангуляция Делоне даёт отклонение от оптимальной триангуляции порядка 2,5%, триангуляция Делоне с последующим локальным перестроением пар соседних треугольников – 0,3%, а триангуляция Делоне с последующими локально жадными перестроениями – порядка 0,03%.

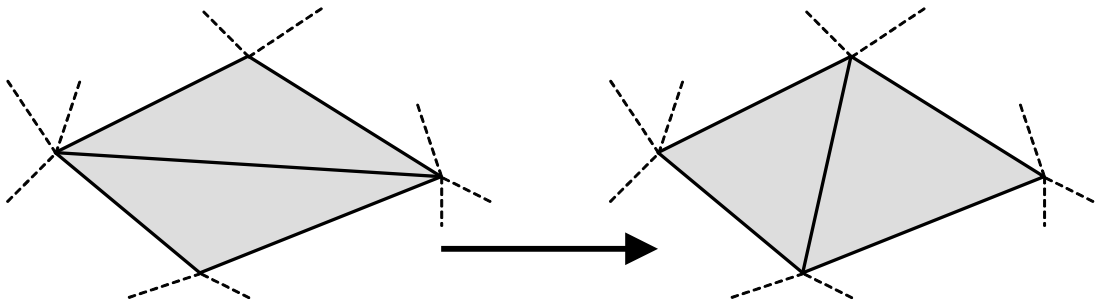


Рис. 54. Перестроение пар соседних треугольников

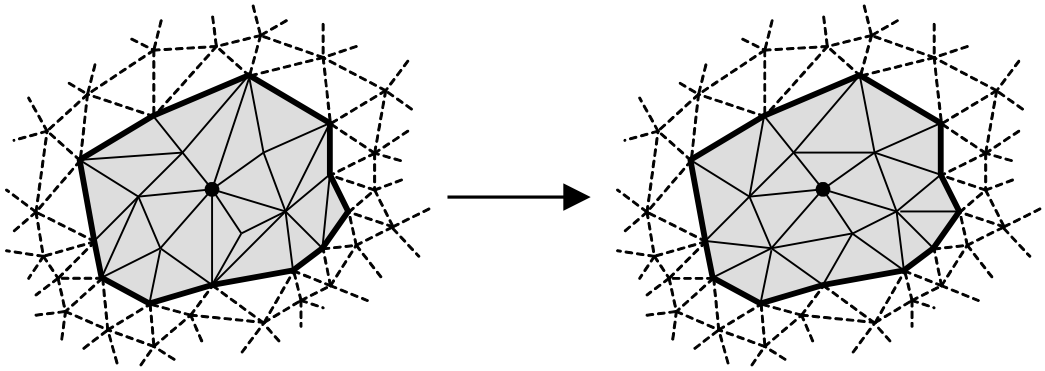


Рис. 55. Локально жадное перестроение двойных окрестностей узлов

Там же показано, что для неравномерных распределений отклонения могут быть заметно больше. Так, для равномерного на отрезках распределения (случайно генерируется $2\lfloor\sqrt{N}\rfloor$ отрезков, а затем на этих отрезках случайно равномерно выбираются N точек) триангуляция Делоне даёт отклонение порядка 35%, триангуляция Делоне с последующим локальным перестроением пар соседних треугольников – 6%, а триангуляция Делоне с последующими локально жадными перестроениями – порядка 0,07%.

Глава 8. Вычислительная устойчивость алгоритмов триангуляции

8.1. Причины возникновения ошибок при вычислениях

Проблема вычислительной устойчивости является одной из основных при решении большинства задач вычислительной геометрии. Многие внешне простые алгоритмы требуют учёта многочисленных крайних случаев, без чего алгоритм на практике просто не работает [12].

Так, за относительной внешней простотой описанных в предыдущих главах алгоритмов триангуляции и алгоритмов триангуляции с ограничениями в действительности скрываются многочисленные детали реализации, от которых существенно зависит устойчивость работы алгоритмов. Перечислим основные возникающие задачи.

Задача 1. *Проверка совпадения двух заданных точек*. Эта проблема является особенно актуальной в случае использования вещественной арифметики с плавающей точкой. Как известно, сравнение плавающих вещественных чисел на равенство производится всегда с заданной точностью ε. Здесь определяющим является выбор значения ε.

Несмотря на кажущуюся простоту, данная проблема имеет далеко идущие последствия. Как известно, в силу своей ограниченной точности обычные вещественные вычисления на компьютерах не обладают многими свойствами истинно вещественных чисел. Например, если мы используем числа, хранящиеся в памяти компьютера с помощью 3 значащих цифр, то результат вычисления следующих выражений может не совпадать, т.е. нарушается свойство ассоциативности:

$$(100+0{,}5)+0{,}5 \cong 100 \neq 101 \cong 100+(0{,}5+0{,}5).$$

Задача 2. *Проверка взаимного расположения двух точек относительно прямой*, проходящей через две заданные точки. Данная задача обычно очень просто решается методами аналитической геометрии. Записываем уравнение прямой, проходящей через две заданные точки – (x_1, y_1) и (x_2, y_2):

$$(x_1 - x)(y_2 - y) - (x_2 - x)(y_1 - y) = 0.$$

Затем подставляем в это уравнение вместо x и y координаты тестовых точек (x_3, y_3) и (x_4, y_4). Если значения выражений будут иметь одинаковый знак, то точки находятся по одну сторону от прямой, иначе – по разную. Результат выражения, равный нулю, будет означать попадание точки строго на прямую.

Здесь проблема заключается в потере точности промежуточных вычислений. Перемножая два n-значных числа, вообще говоря, получаем $2n$-значное число. На практике это обычно не учитывается и младшие n разрядов попросту отбрасываются. В итоге результат вычислений может показать, что тестовая точка лежит на прямой, хотя это не так. Для избавления от этого эффекта сравнение с нулем проводят с некоторой точностью ε. Несмотря на это, реальная точность вычислений все равно уменьшается в 2 раза, составляя не более $n/2$ исходных значащих цифр.

Задача 3. Проверка коллинеарности трёх заданных точек. Эта задача является частным случаем предыдущей, и ей свойственны те же проблемы с переполнением промежуточных вычислений.

Задача 4. Проверка взаимного расположения точки и треугольника. Здесь требуется определить: 1) не совпадает ли точка с одной из вершин треугольника; 2) не попадает ли точка на одно из его рёбер; 3) не попадает ли точка строго внутрь треугольника. Новым здесь является проверка попадания точки строго внутрь треугольника. Это решается путем трёхкратной проверки взаимного расположения заданной точки относительно различных рёбер треугольника, т.е. также сводится к предыдущим задачам.

Задача 5. Проверка порядка обхода трёх заданных точек. Здесь требуется определить, обходятся ли точки в заданном порядке по часовой стрелке или против. Эту задачу также решаем, записывая уравнение прямой, проходящей через две заданные точки, и подставляя в уравнение координаты третьей точки. После чего анализируем знак выражения. Таким образом, если

$$(x_1 - x_3)(y_2 - y_3) - (x_2 - x_3)(y_1 - y_3) < 0,$$

то точки обходятся по часовой стрелке, а если > 0, то против (это верно для левосторонней системы координат, для правосторонней системы все будет наоборот).

В данном алгоритме возникает та же самая проблема переполнения, что и при решении предыдущих задач.

Задача 6. Проверка выполнения условия Делоне для двух заданных смежных треугольников. Данная задача рассмотрена выше в разд. 1.4.

Задача 7. Локализация точки в триангуляции. Локализация точки в триангуляции состоит из выбора некоторого начального треугольника в триангуляции и последовательного перехода по треугольникам к цели. Эта задача рассмотрена выше в разд. 2.1.

Задача 8. Поиск точки пересечения двух прямых. Данная задача возникает при построении триангуляции Делоне с ограничениями, когда обнаруживается, что очередной вставляемый отрезок пересекается с ранее вставленным структурным ребром триангуляции.

В данном случае первой проблемой является то, что определяемая точка пересечения в силу ограниченности точности вычислений в большинстве случаев не лежит ни на одном из рёбер (ни на ранее существовавшем, ни на новых). Возможно, что эта точка лежит даже не в смежных с ребром треугольниках, поэтому в результате разбиения старого ребра на части образуются новые «вывернутые» треугольники, разрушая структуру триангуляции. На рис. 56 дан пример вставки ребра AB в триангуляцию, приводящий к пересечению с существующим ребром в точке S (пунктирными линиями размечена дискретная координатная сетка). В результате округления точка пересечения окажется немного выше реальной – в узле дискретной координатной сетки.

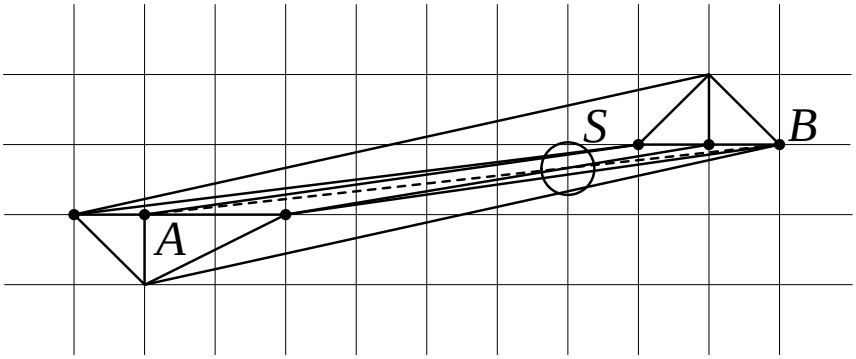


Рис. 56. Пример возможного «выворачивания» треугольников

Несмотря на кажущуюся экзотичность приведенного сценария возникновения ошибки, вероятность его велика уже при попытке вставить в триангуляцию порядка нескольких десятков взаимно пересекающихся структурных рёбер. Вдоль структурных рёбер образуются многочисленные узкие вытянутые треугольники, которые и создают указанную критическую ситуацию.

Вторая проблема в задаче поиска точки пересечения связана непосредственно с самим используемым способом вычислений. Обычно точка пересечения (x, y) находится следующим образом:

$$a = (x_1 - x_3)(y_4 - y_3) - (y_1 - y_3)(x_4 - x_3);$$
$$b = (x_4 - x_3)(y_2 - y_1) - (y_4 - y_3)(x_2 - x_1);$$
$$x = x_1 + a(x_2 - x_1)/b, \quad y = y_1 + a(y_2 - y_1)/b.$$

Здесь сложности возникают при нахождении точки пересечения двух «почти» коллинеарных отрезков. В результате потери точности возможно значительное смещение найденных координат от реального значения. Кроме того, из-за потерь точности мы можем предполагать, что отрезки пересекаются, хотя это не так. Тогда попытка вычисления их пересечения может привести к значительному удалению найденной точки от самих отрезков (для «почти» коллинеарных отрезков).

Таким образом, большинство поставленных проблем связано с потерей точности внутренних вычислений.

8.2. Применение целочисленной арифметики

Контролировать точность, используя стандартные вещественные типы данных, предлагаемые большинством распространенных языков программирования, весьма сложно. Почти все современные компьютеры поддерживают стандарты ANSI представления вещественных чисел, однако даже 10-байтовый тип extended позволяет хранить не более 20 значащих цифр. В то же время при описании 6-й задачи показано, что для корректных вычислений требуется $4n$-значная арифметика. Это означает, что реальная достижимая точность построения триангуляции Делоне составляет не более $20/4=5$ знаков в задании координат исходных данных. То есть значение точности ε для проверки совпадения двух точек, возникшее при описании 1-й задачи, следует установить не менее чем 10^{-5}, что не всегда приемлемо на практике.

Другой способ заключается в использовании в явном виде вычислений с фиксированной точкой. В этом случае можно точно контролировать все потери точности.

Еще более простым является переход к целочисленному представлению координат исходных точек. Например, используя обычные 32-битные целые числа, можно обеспечить точность представления в 9 значащих цифр, что является уже приемлемым в большинстве ситуаций.

Тогда для реализации алгоритма построения триангуляции понадобится реализовать несколько дополнительных функций, оперирующих с 32-, 64- и 128-битными числами. На платформе IA-32 для этого потребуется реализовать следующие функции:

1. **`Mul64(A,B)`**. Умножение 32-разрядных чисел. Результат возвращается 64-разрядным. Функция реализуется как одна команда ассемблера.

2. **`Sqr64(A)`**. Возведение 32-разрядного числа в квадрат. Результат возвращается 64-разрядным. Функция реализуется также как одна команда ассемблера.

3. **`MulSum64(A,B,C,D)`**. Сумма двух произведений 32-разрядных чисел A·B и C·D. Результат возвращается 64-разрядным. Функция реализуется с помощью 9 команд ассемблера.

4. **`MulDif64(A,B,C,D)`**. Разность произведений 32-разрядных чисел A·B и C·D. Результат возвращается 64-разрядным. Функция реализуется с помощью 11 команд ассемблера.

5. **`Mul128(A,B)`**. Умножение 64-разрядных чисел. Результат возвращается 128-разрядным. Функция реализуется 40 командами ассемблера.

6. **`Compare128(A,B,C,D)`**. Вначале вычисляется сумма двух произведений 64-разрядных чисел A·B и C·D. Затем получаемое 128-разрядное число сравнивается с нулем. Если оно меньше нуля, то возвращается false, иначе

– true. Функция реализуется с помощью двух вызовов функции `Mul128` и дополнительных 13 команд ассемблера.

Использование 64-битных процессоров может еще существеннее сократить реализацию этих функций до 1–4 команд ассемблера на функцию.

Таким образом, используя целочисленный способ представления исходных данных совместно с дополнительными операциями над 32-, 64- и 128-битными целыми числами, можно чётко решить поставленные задачи.

При использовании целочисленных вычислений отпадает необходимость использования величин ε, возникающих в задачах 1 и 3, и можно выполнять проверки на равенство непосредственно.

8.3. Вставка структурных отрезков

Теперь рассмотрим описанную в 8-й задаче проблему поиска точки пересечения и разбиения вставляемых структурных рёбер на части.

Несмотря на то, что мы можем выполнять вычисления с помощью дополнительных функций практически без потери точности, все равно точка пересечения двух прямых в общем случае будет иметь нецелые координаты, которые мы будем вынуждены округлить, т.е. в общем случае точка пересечения двух прямых не будет лежать на этих прямых.

Таким образом, появляется необходимость модификации всех алгоритмов вставки структурных линий так, чтобы учесть возможные нарушения структуры триангуляции и избавиться от них. Рассмотрим такой обобщенный алгоритм вставки.

Обобщенный алгоритм вставки структурных линий в триангуляцию с ограничениями.

Пусть L – сортированный по длине список еще не вставленных структурных отрезков.

Шаг 1. В L заносим все отрезки исходных структурных линий.

Шаг 2. Последовательно в цикле извлекаем (с удалением) из L самый длинный отрезок AB и пытаемся вставить этот отрезок в триангуляцию любым алгоритмом вставки, описанным выше. Если обнаруживается, что вставляемый отрезок пересекает некоторые ранее вставленные структурные рёбра (рис. 57,*а*), то их необходимо пометить как обычные нефиксированные рёбра (рис. 57,*б*). При этом надо найти все точки пересечения C_i, где $i=\overline{1,n}$, нового отрезка со вставленными рёбрами E_iF_i. Далее нужно вставить новые точки C_i в триангуляцию (рис. 57,*в*). Затем надо в список L поместить все отрезки C_iC_{i+1}, где $i=\overline{0,n}$, $C_0=A$, $C_{n+1}=B$. Также необходимо туда поместить все отрезки E_iC_i и C_iF_i, где $i=\overline{1,n}$. Если вставляемый в список отрезок имеет нулевую длину, то его вставлять не надо. Цикл вставки продолжается, пока список L не пуст (рис. 57,*г*). *Конец алгоритма.*

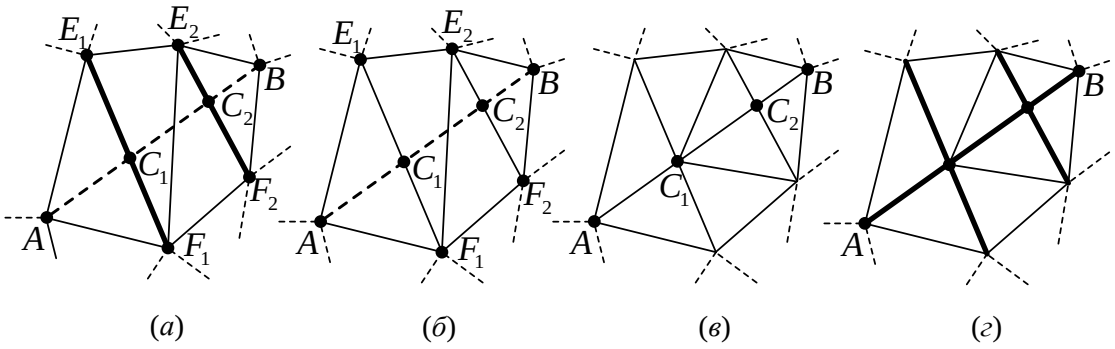


Рис. 57. Обобщенный алгоритм вставки структурных линий
в триангуляцию с ограничениями

В такой реализации данного алгоритма вставки на практике достаточно часто возникает ситуация, когда небольшое количество исходных структурных линий приводит к значительному разрастанию списка *L* в процессе работы. Например, задав 5 «почти» коллинеарных отрезков в качестве исходных структурных линий, можно в конце концов получить тысячи структурных рёбер. Происходит это вследствие того, что каждая пара отрезков после нахождения их пересечений образует 4 новых отрезка, также являющихся «почти» коллинеарными остальным отрезкам. Такое дробление идет до тех пор, пока размеры отрезков не станут сравнимыми с размерами единицы координатной сетки, когда маленькие отрезки становятся «совсем не» коллинеарными или размер отрезков становится настолько малым, что его дальнейшее деление невозможно.

Но и на этом микроуровне возможны проблемы. Например, пусть построена некоторая «плотная» триангуляция, когда в каждом узле координатной сетки имеется по узлу триангуляции и требуется вставить отрезок *AB*, который пересекается с ранее вставленным структурным ребром *CD* (рис. 58). В результате найденная точка пересечения *AB* и *CD* будет округлена, например, до точки *A*. В итоге в список *L* попадут отрезки *AC*, *AD* и опять *AB*. А так как мы извлекаем на каждом шаге из списка самое большое ребро, то список *L* будет бесконечно разрастаться за счёт постоянной вставки рёбер *AC* и *AD*.

Таким образом, в этом варианте алгоритм вставки также всё еще не годится для практической работы.

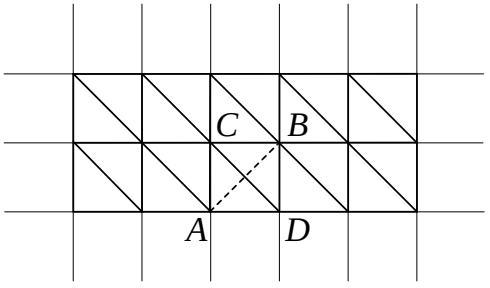


Рис. 58. Попытка вставки микроребра

Во избежание пересечения пар «почти» коллинеарных отрезков и проблем на микроуровне нужны ещё дополнительные модификации алгоритма.

Во-первых, при вставке очередного отрезка необходимо найти все узлы триангуляции, лежащие вблизи от отрезка на расстоянии не более некоторого ε_1. Тогда, если такие точки найдены, вставляемый отрезок разобьём этими точками на части, которые поместим в список L.

Во-вторых, найдя точку пересечения структурных рёбер, прежде чем вставлять новый узел, попробуем найти в окрестности радиуса ε_2 другой, ранее уже вставленный узел триангуляции.

На практике работа алгоритма значительно улучшается уже при значениях $\varepsilon \geq 3$. Увеличение ε приводит к сокращению размера списка L, но несколько увеличивает время выполнения дополнительного поиска точек в окрестностях. Наиболее приемлемыми с точки зрения быстродействия и качества являются значения $\varepsilon_1 = \varepsilon_2 = 10$.

В заключение отметим, что использование целочисленного представления исходных чисел позволяет, с одной стороны, явно контролировать точность вычислений, с другой – повысить скорость работы алгоритма построения триангуляции за счёт отказа от вещественных операций.

Тем не менее простой переход от вещественных вычислений к целочисленным приводит к другому неприятному эффекту – значительному росту количества структурных рёбер в триангуляции. Во избежание этого приходится несколько усложнять алгоритм, вводя дополнительные проверки на наличие совпадающих узлов триангуляции с точностью ε.

Глава 9. Пространственный анализ на плоскости

9.1. Построение минимального остова

В вычислительной геометрии известно множество задач, линейно сводимых к задаче построения триангуляции Делоне [12]. Рассмотрим наиболее часто встречающиеся на практике задачи.

Определение 21. В *задаче построения евклидова минимального остовного дерева* на заданных на плоскости N точках необходимо построить дерево, суммарная длина рёбер которого минимальна.

На практике эта задача в явном виде применяется для оптимизации длины линий электропередачи и телефонной сети. На основе остовного дерева может быть построено приближённое решение задачи коммивояжёра.

Теоретическая оценка трудоёмкости задачи построения минимального остова составляет $O(N \log N)$. В то же время известно, что на основе триангуляции Делоне минимальный остов может быть построен за линейное время (рис. 59). Тогда, используя любой алгоритм триангуляции Делоне, имеющий в среднем линейную трудоёмкость, можно построить и минимальный остов также в среднем за время $O(N)$.

Алгоритм построения минимального остовного дерева.

Шаг 1. Строим триангуляцию Делоне на множестве исходных точек.

Шаг 2. Создаём список всех рёбер полученной триангуляции и сортируем его по длине рёбер.

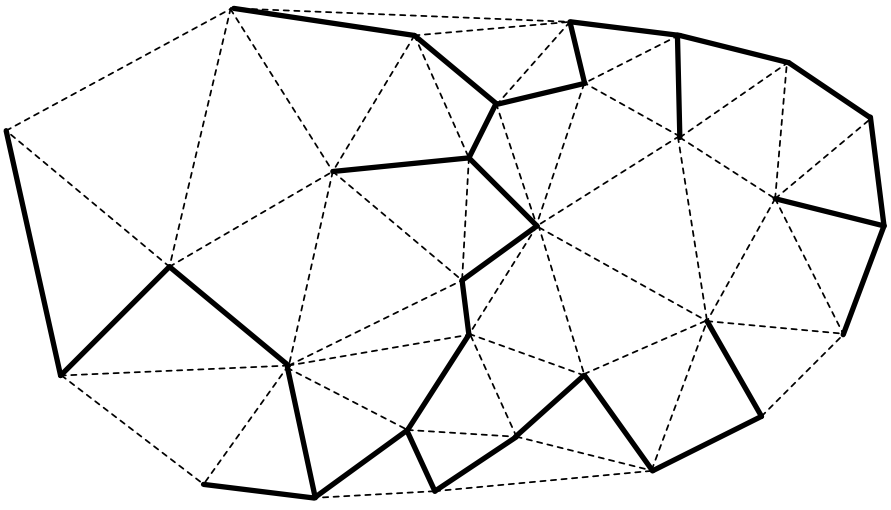


Рис. 59. Построение минимального остова по триангуляции Делоне

Шаг 3. Начинаем генерировать граф остова. Вначале в графе нет ни одного ребра, а вершинами выступают все исходные точки. Для каждой вершины устанавливаем два номера: $g_i := i$ – номер текущей компоненты связности (т.е. всего вначале имеется *N* компонент), $p_i := -1$ – номер смежной вершины выше по дереву связности (величина -1 означает отсутствие таковой).

Шаг 4. В цикле по всем рёбрам триангуляции Делоне жадным алгоритмом от самых коротких рёбер к длинным выполняем шаг 5, пытаясь вставить ребро A_iA_j в граф остова.

Шаг 5. Определяем номера компонент связности, в которые в настоящее время входят вершины A_i и A_j. Для этого выполняем следующий цикл. Пусть $k := i$. Пока $p_k \neq -1$, выполняем $k := p_k$. Полученное значение $k = k(A_i)$ является номером компоненты связности для вершины A_i (для ускорения последующего поиска выполняем еще один цикл: пусть $k := i$; пока $p_k \neq -1$, выполняем $m := k, k := p_k, p_m := k(A_i)$). Также находим номер компоненты и для вершины A_j. Если $k(A_i) \neq k(A_j)$, то вершины A_i и A_j соединяем ребром, при этом объединяем компоненты связности: $p_{k(A_i)} := k(A_j)$. *Конец алгоритма.*

9.2. Построение оверлеев

Основная идея решения с помощью триангуляции задач пространственного анализа на плоскости, таких как построение оверлеев (объединения, пересечения и разности) регионов, буферных зон, зон близости и др., заключается в применении следующего алгоритма [15].

Общий алгоритм пространственного анализа на плоскости.

Шаг 1. Построение триангуляции Делоне с ограничениями по множеству исходных данных.

Шаг 2. Классификация всех треугольников по некоторому принципу.

Шаг 3. Объединение классифицированных треугольников в регионы. *Конец алгоритма.*

Первые два шага этого алгоритма зависят от решаемой задачи пространственного анализа. Вначале рассмотрим задачу построения оверлеев.

Определение 22. Задача построения оверлеев определяется на регионах *A* и *B* как задача нахождения их: 1) объединения; 2) пересечения; 3) разности; 4) симметрической разности [9]. Результат должен быть представлен в виде одного региона.

Большинство существующих алгоритмов обладает различными ограничениями, мешающими их реальному применению. Наиболее суще-

ственным недостатком является невозможность оперирования регионами, имеющими самопересечения или состоящими из нескольких контуров. Другим алгоритмам, обрабатывающим произвольные данные, свойственна сложная реализация или неудовлетворительное время работы на больших наборах исходных данных.

Рассмотрим простой алгоритм решения поставленной задачи, лишённый упомянутых недостатков и имеющий приемлемую трудоёмкость.

Алгоритм построения оверлеев.

Шаг 1. Два исходных региона *A* и *B* (рис. 60,*а*) передаём в алгоритм построения триангуляции с ограничениями. При этом на множестве всех вершин и границ регионов (как структурных линий) будет построена триангуляция Делоне с ограничениями, а все треугольники будут проклассифицированы по признаку принадлежности регионам *A* и *B* (рис. 60,*б*).

Шаг 2. Каждый треугольник T_i полученной триангуляции классифицируем в зависимости от выполняемой операции:

Вариант 1 (объединение):

если $T_i \in A$ или $T_i \in B$, то $C_i = 1$, иначе $C_i = 0$.

Вариант 2 (пересечение):

если $T_i \in A$ и $T_i \in B$, то $C_i = 1$, иначе $C_i = 0$.

Вариант 3 (разность):

если $T_i \in A$ и $T_i \notin B$, то $C_i = 1$, иначе $C_i = 0$.

Вариант 4 (симметрическая разность):

если $T_i \in A$ исключающее или $T_i \in B$, то $C_i = 1$, иначе $C_i = 0$.

Шаг 3. Выполнить алгоритм выделения регионов из триангуляции (рис. 60,*в–е*). *Конец алгоритма.*

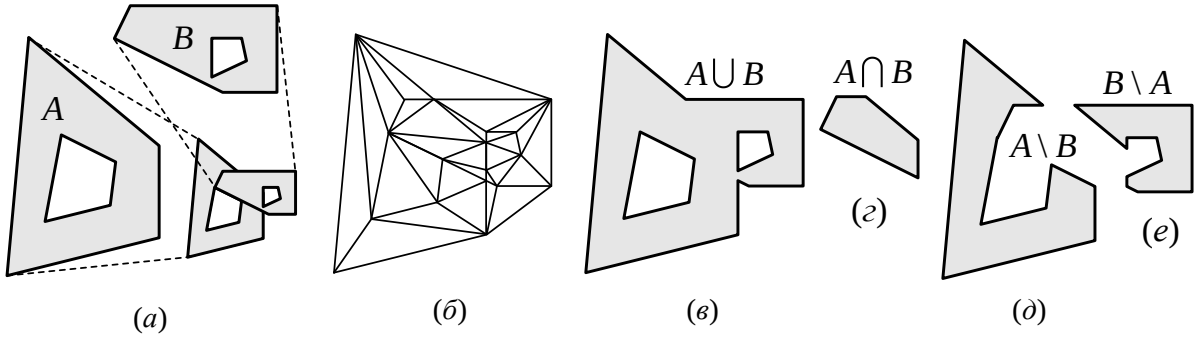


Рис. 60. Построение оверлеев: *а* – исходные регионы; *б* – триангуляция с ограничениями; *в* – объединение; *г* – пересечение; *д*, *е* – разности регионов

Сложности всех шагов алгоритма являются в среднем линейными, поэтому и общая трудоёмкость равна $O(N)$, где N – общее число вершин исходных регионов.

9.3. Построение буферных зон

Определение 23. Задача построения буферных зон требует определения геометрического места точек плоскости, удалённых от множества объектов $\{a_i\}$ не более чем на расстояние $S_i = S(a_i)$. На практике обычно рассматривают три вида объектов a_i: точки, ломаные и регионы. Примеры буферных зон для этих видов объектов приведёны на рис. 61. Границы таких буферных зон могут состоять из множества сегментов двух видов: отрезков и дуг. Так как обработка дуг обычно очень неудобна, то их аппроксимируют ломаными с некоторой заданной точностью. Поэтому с некоторыми ограничениями можно считать, что результатом построения буферных зон будет регион.

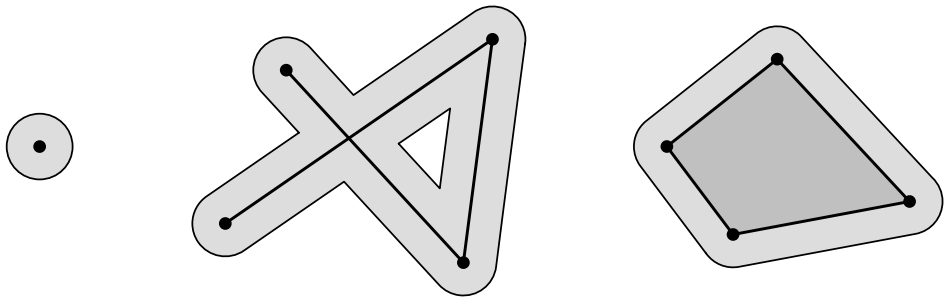


Рис. 61. Примеры буферных зон: *а* – буферная зона точки; *б* – ломаной; *в* – многоугольника

В большинстве существующих алгоритмов построения буферных зон вначале выполняется генерация зон для отдельных фигур, а затем с помощью операции объединения регионов получается искомый результат. Так как трудоёмкость операции объединения регионов с N вершинами в худшем случае составляет не менее $O(N^2)$, то и трудоёмкость таких алгоритмов при построении буферных зон для N линий, имеющих по N вершин, в худшем случае может составлять $O(N^N)$. Но, используя триангуляцию, данная оценка может быть в ряде случаев улучшена. Рассмотрим следующий алгоритм [15].

Алгоритм построения буферных зон.

Пусть дано множество точек, ломаных и регионов (рис. 62,*а*).

Шаг 1. Для всех исходных точечных объектов и вершин ломаных и регионов строятся многоугольники, аппроксимирующие вокруг них круговые буферные зоны (рис. 62,*б*). Выбор размера буферного многоугольника

может осуществляться в виде правильного многоугольника, вписанного, описанного или равного по площади истинному буферному кругу в зависимости от цели применения буферных зон.

Шаг 2. Для отрезков ломаных и границ регионов вычисляются прямоугольники (рис. 62,*в*), которые в объединении с ранее вычисленными круговыми зонами полностью определяют буферные зоны отрезков и ломаных.

Шаг 3. Полученные круговые многоугольники, прямоугольники и исходные регионы передаются на вход алгоритма построения триангуляции с ограничениями в качестве регионов (рис. 62,*г*).

Шаг 4. Все треугольники, попавшие хотя бы в один регион, выделяются в один общий регион (рис. 62,*д*). *Конец алгоритма.*

Самым трудоёмким шагом работы алгоритма является построение триангуляции с ограничениями. Если мы аппроксимируем круговые буферные зоны с использованием S сегментов, то будет построена триангуляция с $O(SN)$ узлами, где N – количество исходных точек и вершин ломаных и регионов. Таким образом, общая трудоёмкость алгоритма составляет в худшем $O(S^2 \cdot N^2)$, а в среднем – $O(SN)$.

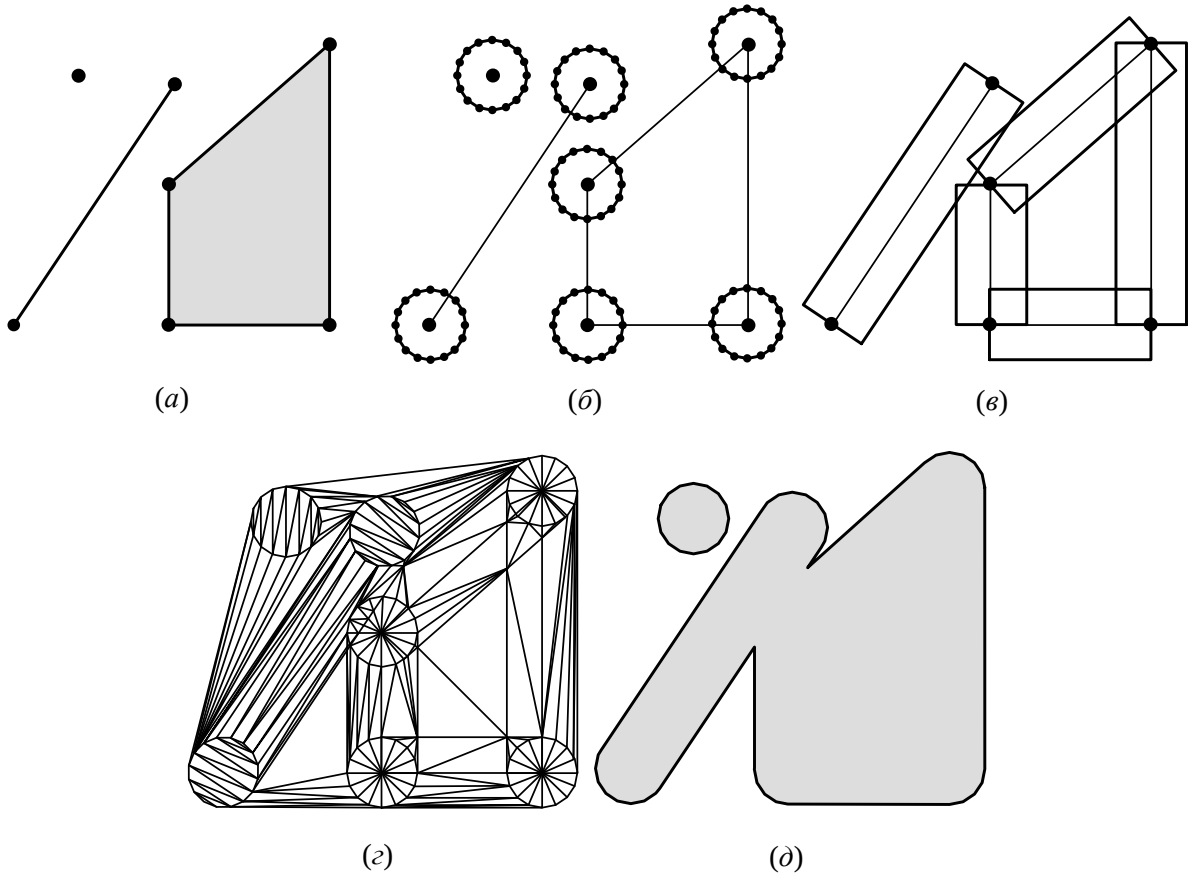


Рис. 62. Построение буферных зон: *а* – исходные объекты; *б* – буферные зоны точек и вершин; *в* – буферные зоны отрезков ломаных и границы регионов; *г* – триангуляция с ограничениями; *д* – построенная буферная зона

Одним из вариантов построения буферных зон является буферизация со «взвешиванием», когда размер буфера является индивидуальным для каждого объекта [9]. При этом аппроксимация круговых буферных зон может производиться многоугольниками с фиксированным числом вершин S либо с переменным, выбираемым на основании заданной точности. Чтобы во втором случае трудоёмкость алгоритма неограниченно не возрастала при увеличении размеров входных объектов, все индивидуальные S_i ограничивают сверху некоторым общим значением S.

9.4. Построение зон близости

Определение 24. Задача построения зон близости требует определения всех точек плоскости, для которых расстояние s до объектов множества $\{a_i\}$ является минимальным.

В случае, когда все объекты являются точечными, данная задача определяется как задача построения диаграмм Вороного (разд. 1.1, рис. 3,*а*). Поэтому её построение на основе триангуляции Делоне не представляет сложности.

Отметим только, что для некоторых из заданных точек соответствующие многоугольники Вороного будут бесконечными фигурами. На практике этого не нужно, и поэтому можно реально ограничить всю плоскость некоторым регионом, обычно называемым *областью интересов*.

Алгоритм построения диаграмм Вороного [15].

Пусть дано множество точек и область интересов в форме прямоугольника (рис. 63,*а*). Требуется найти все многоугольники Вороного для этих точек (рис. 63,*д*).

Шаг 1. По исходному множеству точек строим триангуляцию Делоне (рис. 63,*б*).

Шаг 2. Для каждого треугольника триангуляции вычисляем центр описанной окружности.

Шаг 3. Для каждого узла вычисляем центр многоугольника Вороного. Для этого обходим вокруг текущего узла по смежным треугольникам и собираем центры их описанных окружностей. Если узел находится не на границе триангуляции, то таким образом мы соберем координаты соответствующего многоугольника Вороного этого узла (рис. 63,*в*). Если этот узел находится на границе, значит, многоугольник Вороного является бесконечной фигурой. Поэтому в этом случае необходимо выполнить отсечение двух его бесконечных сторон (рис. 63,*г*). *Конец алгоритма.*

Отметим также, что если среди исходных точек есть 4 или более точек, лежащих на одной окружности, то этот алгоритм выдаст некоторые многоугольники, у которых будут дублироваться последовательные точки контура. Поэтому такой случай необходимо дополнительно отследить.

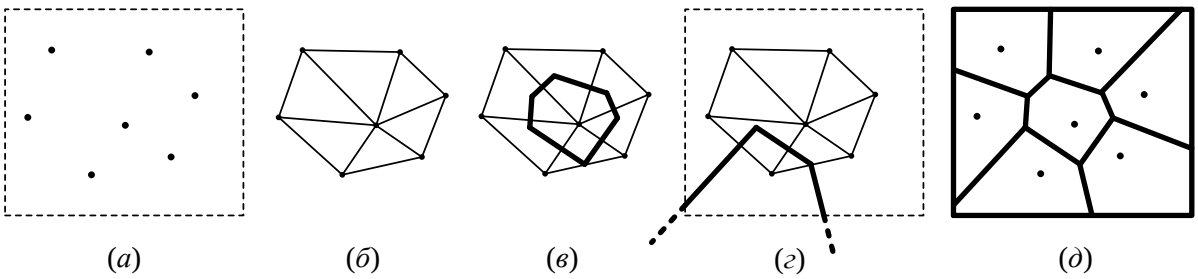


Рис. 63. Построение диаграмм Вороного: *а* – исходные данные и область интересов; *б* – построение триангуляции Делоне; *в,г* – построение многоугольников Вороного для внутренних и внешних узлов; *д* – сформированные диаграммы Вороного

9.5. Построение взвешенных зон близости

На практике диаграммы Вороного могут использоваться, например, для нахождения зон скорейшего обслуживания (зон близости) из заданных базовых пунктов [9]. Однако в действительности возможности базовых пунктов (скорость движения из них, удельные затраты на перемещение) могут быть разными.

Определение 25. Задача построения взвешенных зон близости требует определения всех точек плоскости, для которых расстояние S до объектов множества $\{a_i\}$, помноженное на веса $w_i > 0$, является минимальным.

В такой практически важной постановке задача построения зон близости рассматривается редко, что связано со сложностью получаемых геометрических структур (границы зон состоят из отрезков прямых и дуг окружностей). Однако с использованием триангуляции эта задача решается достаточно просто, при этом отрезки дуг будем аппроксимировать ломаными с заданной точностью.

Для решения данной задачи рассмотрим случай двух точечных объектов a_1 и a_2 с весами w_1 и w_2 .Если $w_1 = w_2$, то решением являются две полуплоскости, разделённые прямой – срединным перпендикуляром к отрезку $a_1 a_2$. Иначе пусть $w_1 > w_2$, $e = w_2 / w_1$. Тогда геометрическое место точек (x, y), ближайших ко второй точке, определяется следующим соотношением:

$$\sqrt{(x-x_1)^2 + (y-y_1)^2} \cdot w_1 < \sqrt{(x-x_2)^2 + (y-y_2)^2} \cdot w_2 .$$

Как видно, данное соотношение определяет круг, поэтому найдём уравнение определяющей его окружности в явном виде:

$$\left((x-x_1)^2 + (y-y_1)^2\right) \cdot w_1^2 = \left((x-x_2)^2 + (y-y_2)^2\right) \cdot w_2^2; \quad \Rightarrow$$

$$\Rightarrow \left(x - \frac{(x_1 - x_2 e^2)}{(1-e^2)} \right)^2 - \frac{e^2 (x_1 - x_2)^2}{(1-e^2)^2} + \left(y - \frac{(y_1 - y_2 e^2)}{(1-e^2)} \right)^2 - \frac{e^2 (y_1 - y_2)^2}{(1-e^2)^2} = 0 .$$

Отсюда получаем центр окружности (x_c, y_c) и её радиус R:

$$x_c = \frac{(x_1 - x_2 e^2)}{(1-e^2)}, y_c = \frac{(y_1 - y_2 e^2)}{(1-e^2)}, R = \frac{e\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}}{(1-e^2)}.$$

Для решения задачи для случая многих объектов необходимо рассмотреть все возможные пары объектов (a_i, a_j) и получить для них разбивающие плоскость линии $l_{i,j}$ (прямые или окружности). Далее нужно разбить плоскость сразу всеми полученными линиями $l_{i,j}$. При этом заметим, что каждый получившийся элемент разбиения r_k целиком принадлежит к какой-то одной зоне достижимости (иначе бы некоторые две точки из r_k принадлежали к разным зонам, достижимым из некоторых a_i, a_j, но тогда они должны быть разделены линией $l_{i,j}$, то есть принадлежать к разным элементам разбиения). После этого необходимо проклассифицировать все элементы разбиения на принадлежность соответствующим зонам и объединить их в регионы, соответствующие зонам.

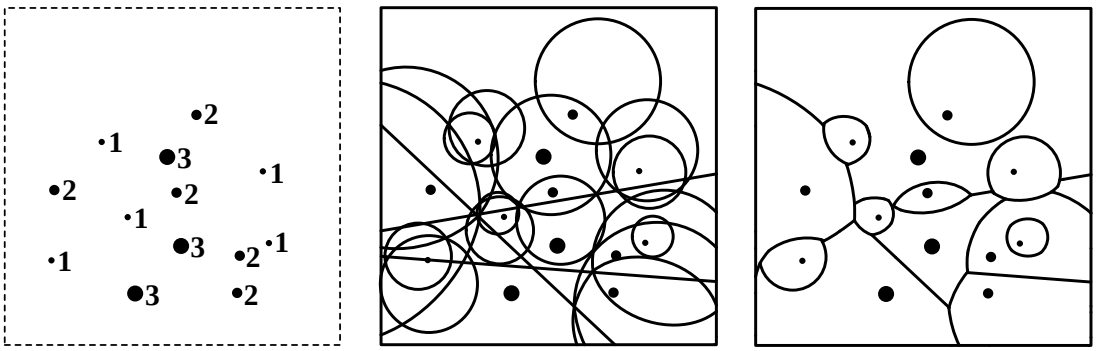
Также отметим, что, как и в диаграммах Вороного, некоторые регионы будут бесконечными, поэтому необходимо дополнительно задать прямоугольную область интересов.

Таким образом, получается следующий алгоритм.

Алгоритм построения взвешенных зон близости.

Пусть дано множество точек $\{a_i\}$ с весами $w_i > 0$ и область интересов в форме прямоугольника (рис. 64,*а*). Требуется найти все взвешенные зоны близости для этих точек (рис. 64,*в*).

Шаг 1. Если во множестве точек только один объект, то получается одна зона достижимости в форме области интересов. Если все объекты имеют одинаковые веса, то нужно выполнить алгоритм построения диаграмм Вороного и закончить работу.



(*а*) (*б*) (*в*)

Рис. 64. Построение взвешенных зон близости:
а – исходные данные с весами; *б* – разделяющие линии;
в – построенные зоны близости

Шаг 2. Для каждой пары объектов (a_i, a_j) находим линию $l_{i,j}$ (прямую или окружность). Если $l_{i,j}$ является окружностью, то аппроксимируем её ломаной. Выполняем её отсечение областью интересов (рис. 64,*б*). Количество точек аппроксимации *S* можно задать фиксированным либо вычислять для каждой окружности индивидуально, исходя из заданной точности построений.

Шаг 3. Все полученные на предыдущем этапе отрезки прямой и аппроксимирующие окружность ломаные необходимо подать в качестве структурных линий на вход алгоритма построения триангуляции Делоне с ограничениями.

Шаг 4. Классифицируем все треугольники, выбирая из множества $\{a_i\}$ ближайший достижимый объект.

Шаг 5. Выполняем алгоритм выделения регионов. Выдаём полученные регионы и завершаем работу (рис. 64,*в*). *Конец алгоритма.*

Трудоёмкость работы данного алгоритма складывается в первую очередь из квадратичного количества линий $l_{i,j}$ относительно исходного числа точек *N*. В целом она составляет в среднем $O(SN^2)$.

9.6. Нахождение максимальной пустой окружности

Определение 26. В *задаче нахождения наибольшей пустой окружности* требуется найти наибольшую окружность, не содержащую внутри ни одной точки заданного множества точек, центр которой лежит внутри выпуклой оболочки исходных точек (рис. 65,*а*).

Данная задача возникает при размещении какой-то службы или предприятия, при этом требуется максимально удалить объект от других заданных объектов. Размещаемый объект может быть источником загрязнений, поэтому необходимо минимизировать его воздействие на другие объекты, либо это магазин, и необходимо поместить его подальше от конкурентов.

В [12] показано, что центр искомой окружности лежит либо в узле диаграммы Вороного (т.е. является *окружностью Делоне* – окружностью, описанной вокруг некоторого треугольника триангуляции Делоне), либо в точке пересечения диаграммы Вороного и выпуклой оболочки исходных точек. В связи с этим возникает следующий алгоритм.

Алгоритм нахождения наибольшей пустой окружности.

Шаг 1. На множестве исходных точек строится диаграмма Вороного с помощью алгоритма, описанного в разд. 9.4 (рис. 65,*б*). При этом для каждой точки диаграммы оказываются вычисленными центр и радиус соответствующей окружности Делоне.

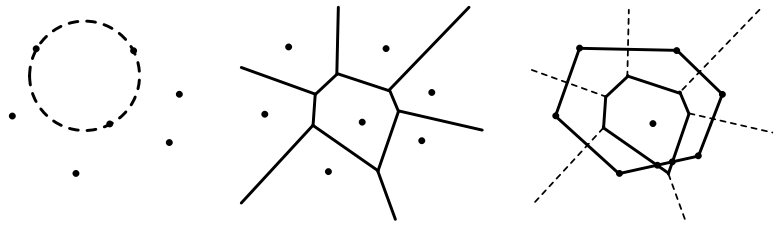


Рис. 65. Определение максимальной пустой окружности:
а – исходные данные; *б* – диаграмма Вороного; *в* – отсечение диаграммы Вороного выпуклой оболочкой точек

Шаг 2. Находится выпуклая оболочка исходных точек, и определяются точки её пересечения с конечными рёбрами диаграммы Вороного (рис. 65,*в*). Для этих точек пересечения вычисляется расстояние до ближайшей исходной точки, которая определяется одной из двух смежных ячеек диаграммы Вороного. Найденное расстояние определяет радиус наибольшей окружности, которую можно здесь разместить, не накрывая никаких точек.

Шаг 3. Среди всех окружностей, полученных на шагах 1–2, выбираем имеющую максимальный радиус. *Конец алгоритма.*

Трудоёмкость полученного алгоритма является в среднем линейной относительно количества исходных точек.

Глава 10. Триангуляционные модели поверхностей

10.1. Структуры данных

На практике для моделирования поверхностей, являющихся однозначными функциями высот от планового положения точки, используются два основных вида структур – регулярная (равномерная прямоугольная) и нерегулярная (триангуляционная) сети (рис. 66).

Основным недостатком регулярной сети является громоздкость представления данных. Реальные объекты для достаточно детального представления требуют огромного массива данных. Поэтому приходится выбирать между точностью представления (размером ячейки) и размером охватываемой территории.

В триангуляционной модели качество аппроксимации поверхности значительно выше, чем в регулярной. Но при этом существенно возрастает сложность обрабатывающих алгоритмов.

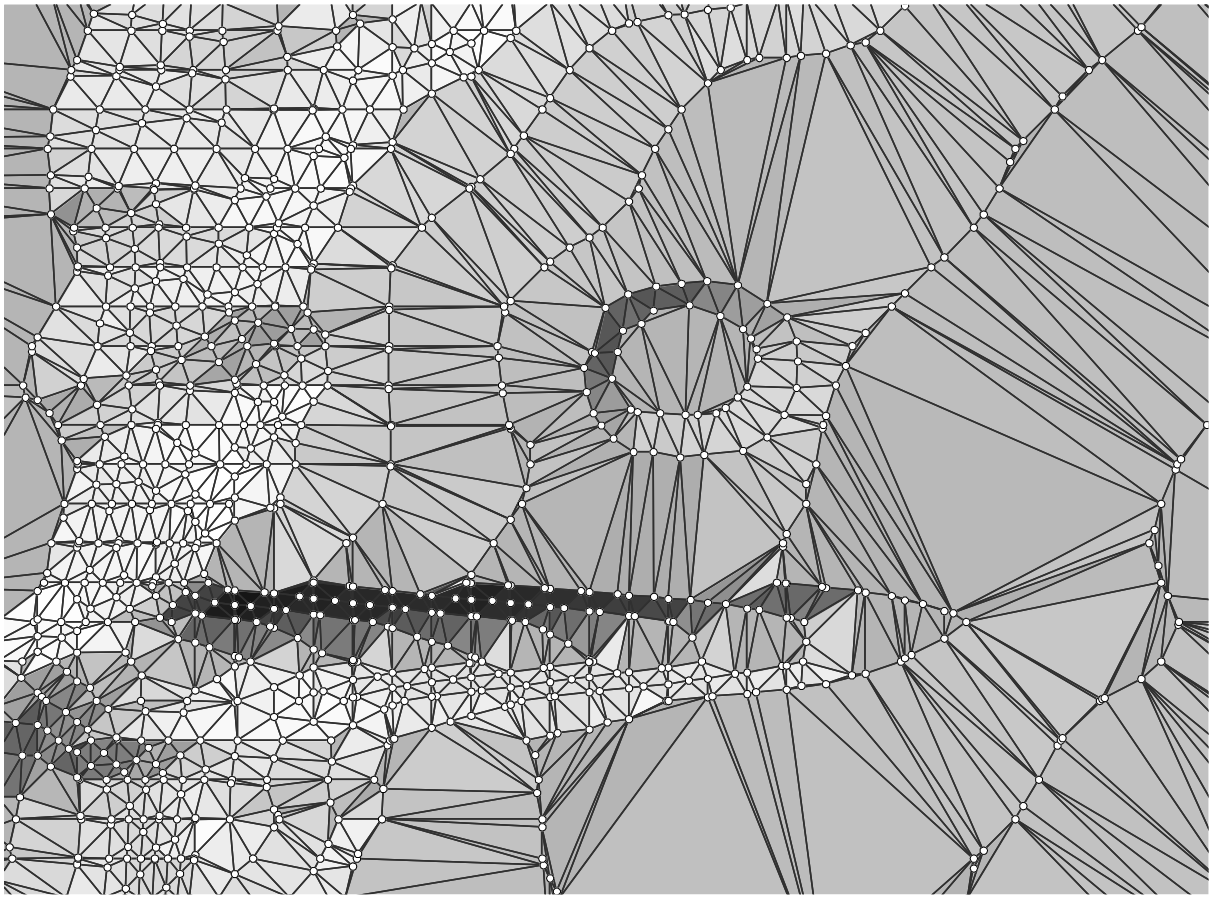


Рис. 66. Триангуляционная модель поверхности Земли

При построении модели рельефа на практике встречаются следующие виды исходных данных:

1. Трёхмерные *точки* на поверхности (высотные отметки на карте).

2. *Структурные линии рельефа* – линии, вдоль которых имеет место нарушение гладкости поверхности (линии обрывов, границы рек, ручьи, горные хребты, водоразделы, границы искусственных сооружений). Структурные линии задаются как трёхмерные ломаные.

3. *Изолинии* – линии одного уровня, вдоль которых поверхность является гладкой.

4. *Горизонтальные плато* – регионы, внутри которых высота поверхности повсюду одинакова (озера).

5. *Области интересов* – регионы, вне которых информация неизвестна или не интересует пользователя.

На практике модель рельефа применяется совместно с другими данными о местности, такими как расположение рек, лесов, дорог, домов и др. При этом в ряде случаев возникает потребность учёта непосредственно в модели рельефа данных о местности. Например, для упрощения различных расчётов можно потребовать, чтобы треугольники не пересекались с границами дорог, земельных участков, домов и др. Тогда при трёхмерной визуализации рельефа можно разным цветом раскрасить разные треугольники в зависимости от того, принадлежат ли они дороге, полю или лесу.

Таким образом, возникает еще один вид исходных данных для построения модели рельефа:

6. *Разделительные линии* – линии, изменяющие только структуру треугольников, не трогая формы поверхности.

Имея перечисленные виды исходных данных, мы можем построить модель рельефа, передав в алгоритм построения триангуляции Делоне с ограничениями все исходные данные. Но при этом вставка в триангуляцию структурных линий вида 2 должна выполняться алгоритмом вставки «Удаляй и строй» или «Перестраивай и строй», а данные видов 3–6 – алгоритмом «Строй, разбивая».

Использование вставки «Строй, разбивая» вызвано желанием минимизировать искажения формы поверхности при вставке линий, а также избежать появления длинных узких треугольников вдоль этих линий (особенно вдоль данных видов 3–5).

10.2. Переброски рёбер

Приведённые в предыдущем разделе виды исходных данных для построения триангуляционных моделей рельефа во многих случаях достаточны для построения моделей поверхностей. Тем не менее очень часто

автоматически построенные треугольники не дают правильной картины поверхности, например, если исходными данными для триангуляции являются изолинии, полученные в результате обработки картографических материалов (рис. 67).

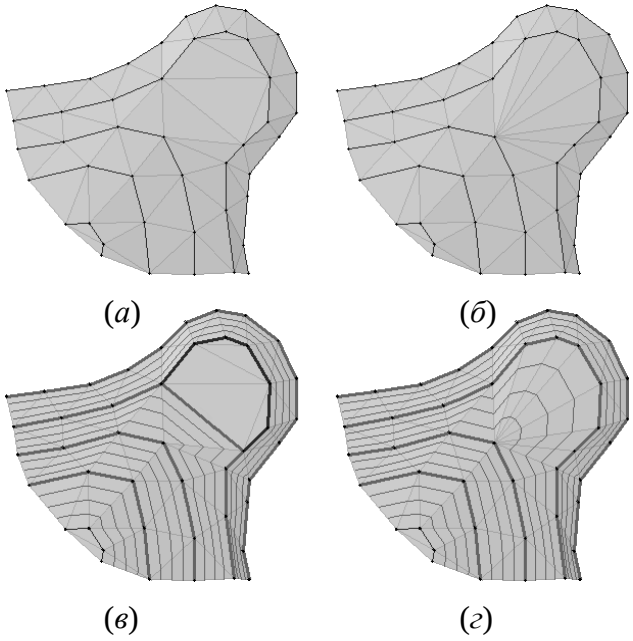


Рис. 67. Пример автоматического построения триангуляции:
а – автоматически построенная поверхность; *б* – исправленная поверхность; *в* – изолинии на автоматически построенной поверхности; *г* – изолинии на исправленной поверхности

Специально для ситуаций с некорректными треугольниками вблизи изолиний разработаны различные алгоритмы, в частности запрещающие построение треугольников, все 3 узла которых лежат на изолиниях одного уровня. Такие алгоритмы по мере возможности выполняют перестроение пар соседних треугольников (*перебрасывают рёбра*) до тех пор, пока не исчезнут запрещённые треугольники. Если такие треугольники остаются, а улучшающих переброс рёбер больше нет, то внутри всех недопустимых непрерывных плоских областей треугольников вставляются новые узлы триангуляции, высоты которых выбираются пропорционально расстояниям до двух смежных изолиний.

Тем не менее не во всех случаях такие алгоритмы приемлемы, а потому для достижения нужного качества модели приходится прибегать к ручному редактированию триангуляции – к ручной интерактивной переброске рёбер. В соответствующих графических системах (ГИС и САПР), работающих с триангуляционными моделями поверхностей, для этого предусмотрены специальные инструменты, позволяющие указать на карте ребро триангуляции и выполнить его переброску.

Все выполняемые переброски необходимо запоминать, чтобы в будущем их можно было повторить. Такая потребность может возникнуть, если была выполнена некоторая коррекция исходных данных и, как следствие, была перестроена вся триангуляция.

Таким образом, переброски рёбер должны выступать как ещё один тип исходных данных в дополнение к приведённым в предыдущем разделе. Удобнее всего переброски хранить как набор из плоских координат четырёх узлов двух смежных треугольников, что позволяет однозначно их идентифицировать и выполнять их перестроения.

Соответствующий алгоритм, выполняющий переброску рёбер, должен вначале найти по координатам узлов пару смежных треугольников, а затем выполнить перестроение. Для выполнения поиска необходимо последовательно переходить от некоторого начального треугольника до искомого, как в итеративном алгоритме построения триангуляции Делоне (см. п. 2.1). Трудоёмкость такого алгоритма составляет в среднем $O(\sqrt{N})$. При выполнении K перебросок такой алгоритм будет уже работать в среднем за время $O(K\sqrt{N})$, что может оказаться неудовлетворительным для больших значений K.

Для сокращения времени поиска треугольников можно использовать различные индексирующие и кэширующие структуры, например описанные в пп. 2.2–2.3, в частности, динамический кэш, образуемый после окончания работы алгоритма динамического кэширования (см. п. 2.3.2), позволяет снизить трудоёмкость алгоритма выполнения переброски до $O(1)$, а выполнения K перебросок – до $O(K)$.

В некоторых случаях при интенсивном ручном редактировании поверхностей образуется очень большой список перебросок рёбер. Это может получиться, например, когда оператор повторно выполняет переброски одних и тех же рёбер, визуально подбирая требуемую форму поверхности. Таким образом, возникает задача сокращения списка перебросок рёбер, позволяющая ограничить неконтролируемый рост объёмов исходных данных.

Описываемый здесь алгоритм сокращения списка перебросок рёбер базируется на том, что любую триангуляцию с помощью последовательных перебросок пар рёбер можно преобразовать в триангуляцию Делоне с ограничениями (см. п. 1.1). Кроме того, используется то свойство, что триангуляция Делоне обладает максимальной суммой минимальных углов среди всех возможных триангуляций.

Алгоритм сокращения списка перебросок рёбер.

На вход алгоритма подаётся ранее построенная триангуляция с уже выполненными перебросками рёбер.

Шаг 1. Очищаем список перебросок рёбер F.

Шаг 2. Создаём список непроанализированных пар соседних треугольников L, и помещаем туда все пары T_i, T_j, которые не удовлетворяют условию Делоне и для которых можно сделать переброску рёбер. Для каждой пары треугольников сопоставляем значение функции $m(T_i, T_j)$, определяющей, насколько увеличится сумма минимальных углов двух треугольников T_i и T_j, если выполнить переброску рёбер. Список L сортируем по убыванию значения $m(T_i, T_j)$.

Шаг 3. Пока список L не пуст, извлекаем из него первую пару T_i, T_j (т.е. с максимальным значением $m(T_i, T_j)$). Если T_i и T_j являются смежными и условие Делоне для этой пары не выполняется (несмотря на то, что в L изначально помещаются только смежные треугольники, эта смежность и условие Делоне могут быть нарушены при перестроении других пар треугольников), то выполняем и запоминаем в F переброску рёбер в этой паре треугольников, а в список L помещаем с учётом функции сортировки до четырёх новых пар смежных треугольников, для которых условие Делоне не выполняется. *Конец алгоритма.*

Стоит отметить, что, поскольку одна и та же триангуляция может быть получена с помощью различных последовательностей переброс, то в результате работы приведённого алгоритма список переброс рёбер может не только сократиться, но даже увеличиться. При этом триангуляции, построенные на одних и тех же исходных точках, но на разных списках флипов, будут геометрически эквивалентными. Возможность данного удлинения списка объясняется тем, что, поскольку порядок узлов и треугольников в двух эквивалентных триангуляциях может отличаться, восстановление построенной триангуляции до триангуляции Делоне может начаться с другого элемента. Соответственно, список переброс может получиться совершенно другой.

В общем случае, после работы данного алгоритма, размер списка переброс будет составлять в среднем не более $O(N)$, независимо от количества перестроений, сделанных пользователем интерактивно.

10.3. Упрощение триангуляции

Реальные триангуляционные модели рельефа земной поверхности обычно содержат огромное количество данных – миллионы и миллиарды точек и треугольников. В связи с этим возникают две основные проблемы: 1) как обрабатывать такие большие модели, если они не помещаются в память компьютера; 2) как их быстро отображать на экране компьютера. Вторая проблема стоит даже более жестко, чем первая, так как там часто предъявляется дополнительное требование работы в реальном режиме времени.

Основной подход к решению этих проблем заключается в построении упрощенных моделей поверхности, которые имеют значительно меньший размер.

Определение 27. Пусть имеется триангуляция T, содержащая $N = |T|$ узлов. В *задаче построения упрощающей триангуляции (задаче генерализации)* требуется найти такую триангуляцию t, что:

1) она содержит заданное количество узлов $n = |t| < N$ и имеет минимальное отклонение d от T: $d(T,t) = \min_{\tau} d(T,\tau), |\tau| = n$ (*задача, управляемая геометрией*);

2) она имеет отклонение d по вертикали от T не более чем на заданную величину ε и имеет минимальное количество узлов $n(t) = \min_{\tau} n(\tau)$, $d(T,\tau) < \varepsilon$ (*задача, управляемая ошибкой*).

Данная задача в обоих вариантах является NP-сложной [20], поэтому на практике используются приближенные алгоритмы, которые можно разделить в соответствии с применяемой стратегией на два основных класса, работающих «сверху вниз» и «снизу вверх» [49].

Стратегия «сверху вниз» начинает работу с простой аппроксимирующей модели, состоящей из одного или нескольких треугольников, покрывающих исходную триангуляцию. Далее в триангуляцию последовательно добавляются новые точки до тех пор, пока не будет достигнуто требуемое разрешение. Рассмотрим один из таких алгоритмов [36].

Алгоритм «Селектор Делоне».

Дана исходная триангуляция T и задана требуемая точность ε или нужное количество треугольников n.

Шаг 1. Строится триангуляция $\tilde{T}$ из одного или нескольких треугольников, покрывающих T. Для каждого треугольника $t_j \in \tilde{T}$ создается список L_j еще не использованных и попадающих внутрь t_j узлов p_i.

Шаг 2. Для каждого узла $p_i \in T$ вычисляется отклонение $d_i = d(p_i, \tilde{T})$ по вертикали от новой триангуляции $\tilde{T}$. Все списки точек L_j, связанные с $t_j \in \tilde{T}$, сортируются по величине d_i, а все треугольники $t_j \in \tilde{T}$ помещаются в сбалансированное дерево поиска по значению максимального отклонения $d'_j = \max_{p_i \in t_j} d_i$ внутри этого треугольника.

Шаг 3. Пока не достигнута требуемая точность или нужное количество треугольников, выполняется следующий цикл. В дереве поиска выбирается треугольник $t_j \in \tilde{T}$ с максимальным d'_j, а внутри t_j – узел p_i с максимальным d_i. Этот узел удаляется из списка L_j и вставляется в триангуляцию $\tilde{T}$ с помощью процедуры вставки из итеративного алгоритма триан-

гуляции. При этом некоторые треугольники в $\tilde{T}$ будут перестроены, поэтому необходимо перераспределить точки в списках L_j, соответствующих изменённым треугольникам, заново вычислить в них d_i, отсортировать списки и обновить дерево поиска. *Конец алгоритма.*

Трудоёмкость данного алгоритма зависит главным образом от сложности перераспределения треугольников, пересчёта отклонений, сортировки списков и обновления дерева поиска. В худшем случае трудоёмкость алгоритма составляет $O(Nn\log n)$, где n – количество треугольников в τ после завершения работы алгоритма. Однако в среднем на реальных данных этот алгоритм показывает трудоёмкость только $O(N\log n)$ [32].

На рис. 68 представлен пример использования селектора Делоне (в качестве исходной покрывающей триангуляции были использованы два треугольника, а требуемое количество узлов $n=100$).

В стратегии «снизу вверх» работа начинается с исходной триангуляции и число элементов триангуляции постепенно уменьшается до тех пор, пока не будет достигнуто требуемое количество узлов либо заданное допустимое отклонение ε упрощенной триангуляции от исходной.

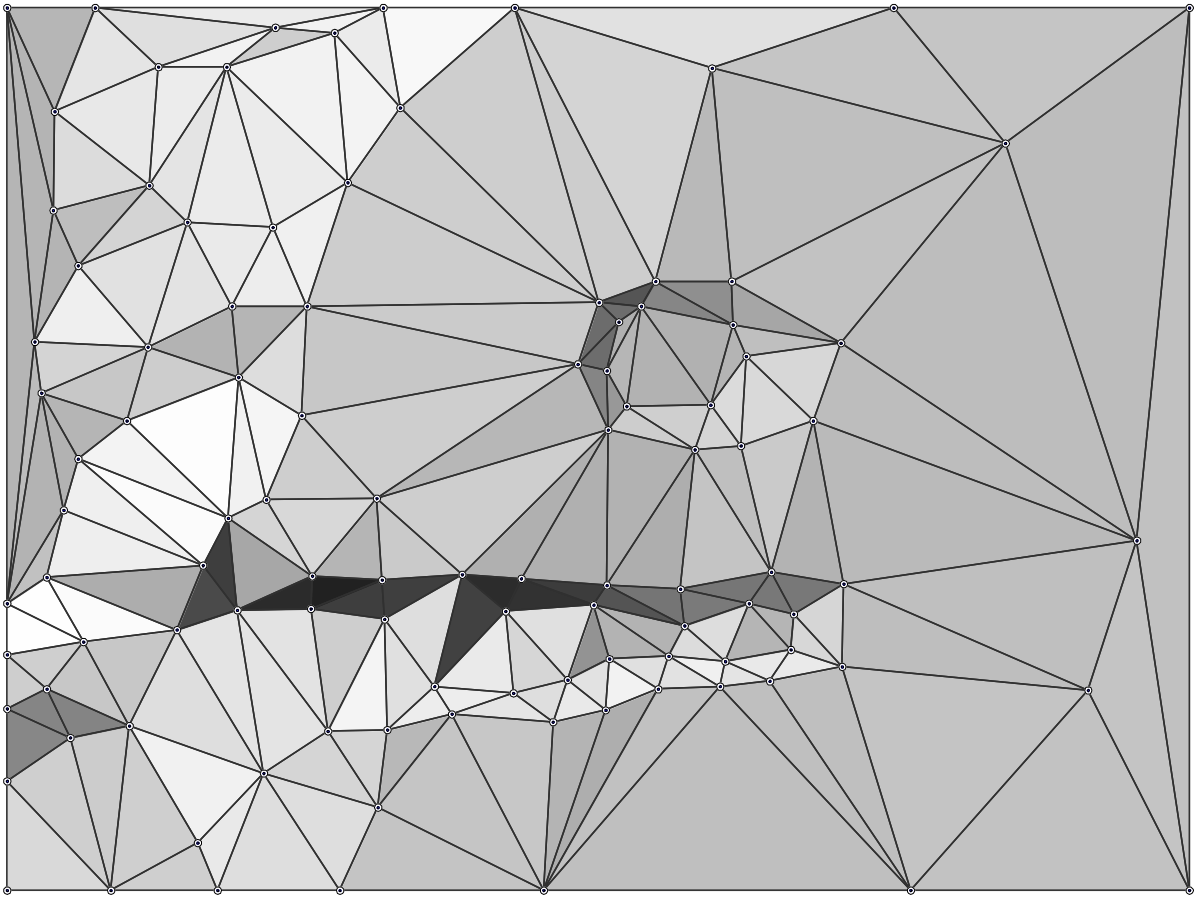


Рис. 68. Пример построения упрощённой модели с помощью селектора Делоне по модели рельефа, представленной на рис. 66

Уменьшение числа элементов обычно выполняется с помощью *локальной модификации* триангуляции – операции, заменяющей некоторую маленькую группу смежных треугольников на другую, покрывающую ту же область. На практике обычно применяют 3 вида *локальных модификаций* (рис. 69): а) удаление узла, б) коллапс ребра и в) коллапс треугольника.

Алгоритм локального упрощения триангуляции.

Дана исходная триангуляция T и задана требуемая точность ε или нужное количество треугольников n.

Шаг 1. Создаем новую триангуляцию как копию исходной $\tilde{T} := T$.

Шаг 2. Для каждого узла $p_j \in \tilde{T}$ устанавливаем отклонение от исходной модели $\pi_j := 0$ и вычисляем потенциальное новое отклонение $\tilde{\pi}_j$, которое возникнет на поверхности при его удалении (среднее высот его соседей Делоне, взвешенных по их удаленности от узла p_j).

Шаг 3. Для каждого ребра $r_j \in \tilde{T}$, соединяющего узлы p_{j_1} и p_{j_2}, вычисляем отклонение его центра от исходной модели $\rho_j := 0$ и потенциальное новое отклонение $\tilde{\rho}_j := (\pi_{j_1} + \tilde{\pi}_{j_1} + \pi_{j_2} + \tilde{\pi}_{j_2})/3$, которое возникнет при его коллапсе.

Шаг 4. Для каждого треугольника $t_j \in \tilde{T}$, соединяющего узлы p_{j_1}, p_{j_2} и p_{j_3}, вычисляем отклонение его центра от исходной модели $\tau_j := 0$ и потенциальное отклонение $\tilde{\tau}_j := (\pi_{j_1} + \tilde{\pi}_{j_1} + \pi_{j_2} + \tilde{\pi}_{j_2} + \pi_{j_3} + \tilde{\pi}_{j_3})/6$, которое возникнет при его коллапсе.

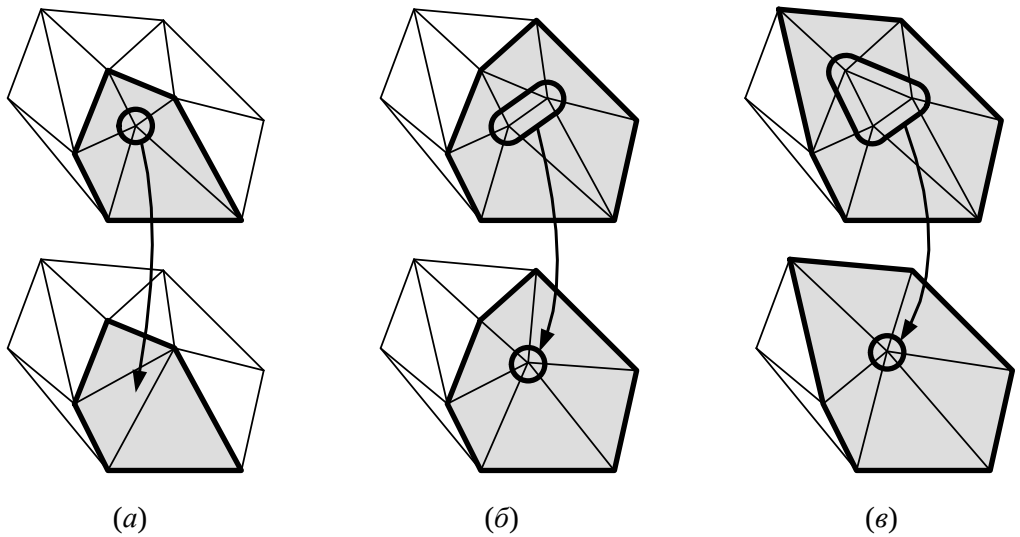


Рис. 69. Виды локальных модификаций триангуляции:
а – удаление узла; *б* – коллапс ребра; *в* – коллапс треугольника

Шаг 5. Все объекты триангуляции – узлы, рёбра и треугольники – помещаем в сбалансированное дерево поиска по значениям $\pi_j + \tilde{\pi}_j$, $\rho_j + \tilde{\rho}_j$ и $\tau_j + \tilde{\tau}_j$ соответственно.

Шаг 6. Пока не будет превышена допустимая точность или не будет достигнуто заданное количество треугольников, выполняется следующий цикл. В триангуляции выбирается объект (узел, ребро или треугольник) с минимальным значением величин $\pi_j + \tilde{\pi}_j$, $\rho_j + \tilde{\rho}_j$ или $\tau_j + \tilde{\tau}_j$. В соответствии с найденным минимумом выполняется удаление узла, коллапс ребра или коллапс треугольника. После этого необходимо выполнить расчёт текущих и потенциальных отклонений для всех вновь появившихся объектов триангуляции и обновить дерево поиска. *Конец алгоритма.*

Трудоёмкость данного алгоритма зависит главным образом от сложности расчёта отклонений для вновь появившихся объектов триангуляции, когда необходимо находить в исходной триангуляции треугольник, в который попадает исследуемая точка. Если для исходной триангуляции имеется кэш поиска (как в алгоритмах триангуляции с кэшированием), то поиск одной точки будет происходить в среднем за время $O(1)$. Так как в среднем при одной локальной модификации триангуляции затрагивается $O(1)$ объектов, а обновление дерева поиска занимает время $O(\log n_i)$, где n_i – текущий размер триангуляции, то в целом данный алгоритм имеет трудоёмкость в среднем около $O(N \log N)$.

По поводу двух стратегий «сверху вниз» и «снизу вверх» следует отметить, что первая из них, как правило, работает точнее при одинаковом наборе изменяющих операций (удаление/вставка узлов). Однако вторая стратегия в среднем работает быстрее и в ней можно использовать другие операции (например, коллапс рёбер и треугольников), что также может в ряде случаев повысить качество работы [32]. Кроме того, заметим, что для реализации первой стратегии достаточно процедур, предоставляемых любым итеративным алгоритмом триангуляции, в то же время для второй необходимы дополнительные алгоритмы для удаления точек, коллапса рёбер и треугольников, которые имеют свои сложности в реализации.

10.4. Мультитриангуляция

В предыдущем разделе была рассмотрена задача получения триангуляции требуемого разрешения. Однако на практике часто возникает задача получения триангуляции, разрешение которой может меняться на различных её участках. Наиболее часто такая задача возникает при трёхмерной визуализации моделей рельефа, когда в некотором секторе вблизи точки зрения необходимо иметь триангуляцию высокого разрешения, а для удаленных областей – низкого (на рис. 70 точка наблюдения помечена флажком, а жирными линиями выделен сектор видимости).

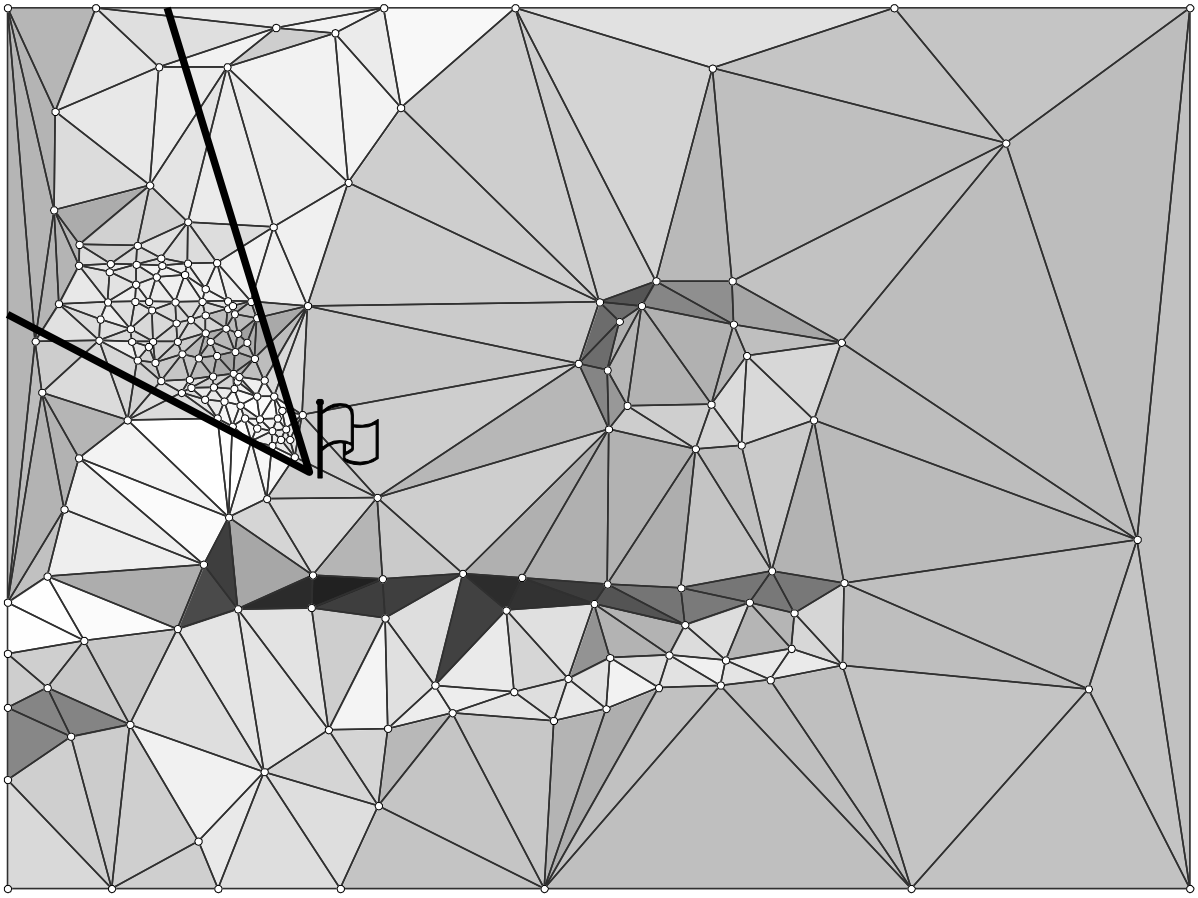


Рис. 70. Триангуляция переменного разрешения, построенная для визуализации из заданной точки наблюдения по модели рельефа, представленной на рис. 66

Другим примером является получение триангуляционной модели, имеющей высокое разрешение только вдоль некоторой ломаной. Это возникает, например, при построении профилей, анализе рельефа вдоль дорог, ЛЭП и др.

Основная проблема получения триангуляции переменного разрешения заключается в непрерывной сшивке областей разного разрешения. Многие ранние модели данных были основаны на вложенных разбиениях (квадродерево, иерархическая триангуляция) или на последовательности слоев данных различного разрешения. При этом либо не удавалось обеспечить непрерывность сшивания областей разного разрешения, либо в месте сшивки получались узкие длинные треугольники, которые существенно искажали форму поверхности.

В основе *мультитриангуляции* – модели данных, позволяющей получать триангуляционные модели требуемого разрешения, – лежат две основные идеи [63]. Во-первых, требуемая триангуляция может быть получена из некоторой другой модели с помощью последовательности локальных модификаций триангуляции (см. предыдущий раздел). Вторая идея восходит к методу детализации триангуляции Киркпатрика [43], когда для

фрагментов триангуляций различного разрешения строится ориентированный ациклический граф, в котором дугами кодируется пространственное наложение фрагментов различных разрешений (наложением считается такое пересечение, что размыкание области пересечения не пусто, в частности, касание треугольников узлами и рёбрами не считается наложением).

На рис. 71,*а* приведен пример последовательности триангуляций различного разрешения (жирными линиями обведены области, в которых триангуляция была подвергнута локальной модификации). Для этой последовательности на рис. 71,*б* отдельно вынесены T_0 – исходная триангуляция худшего разрешения и $T_1 - T_5$ – фрагменты, появившиеся в результате локальных модификаций. Заметим, что фрагменты, принадлежащие к одной триангуляции в последовательности, не налагаются друг на друга. Исходную триангуляцию T_0 также можно считать фрагментом.

На рис. 72 представлен ориентированный ациклический граф, корнем которого является T_0 – исходная триангуляция, а узлами – фрагменты локальных модификаций триангуляции. От узла T_i к узлу T_j проводится ориентированное ребро, если, применяя локальную модификацию T_j, удаляются треугольники, находящиеся во фрагменте T_i. Другими словами, для применения модификации T_j необходимо вначале применить T_i.

Такая структура графа собственно и называется *мультитриангуляцией* $T = \{T_i\}$. С её помощью можно получать различные триангуляции требуемого качества, комбинируя различные фрагменты T_i. На рис. 73 приведены примеры таких возможных триангуляций (в выражении типа $T_i \oplus T_j$ обозначено применение локальной модификации T_j к триангуляции T_i).

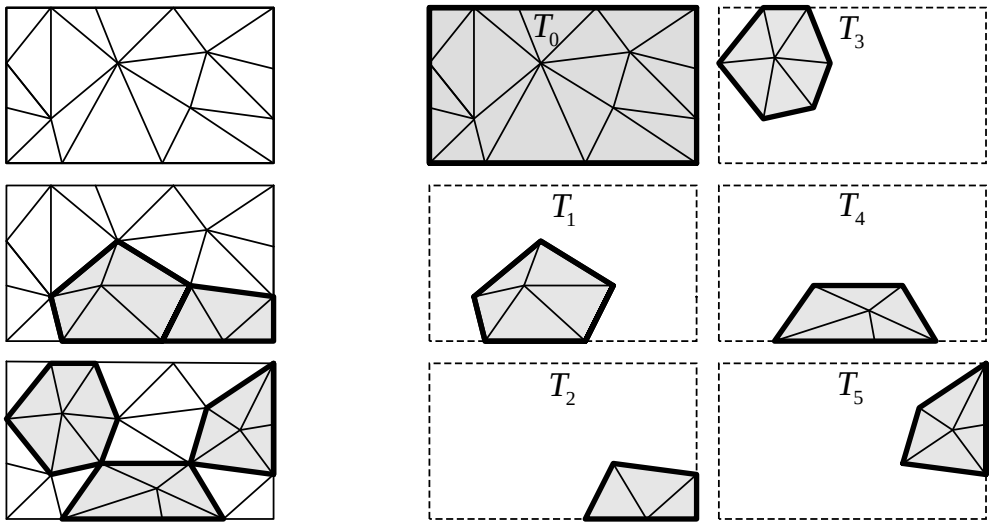


Рис. 71. Последовательность триангуляций различного разрешения (*а*) и её интерпретация как последовательность локальных изменений (*б*)

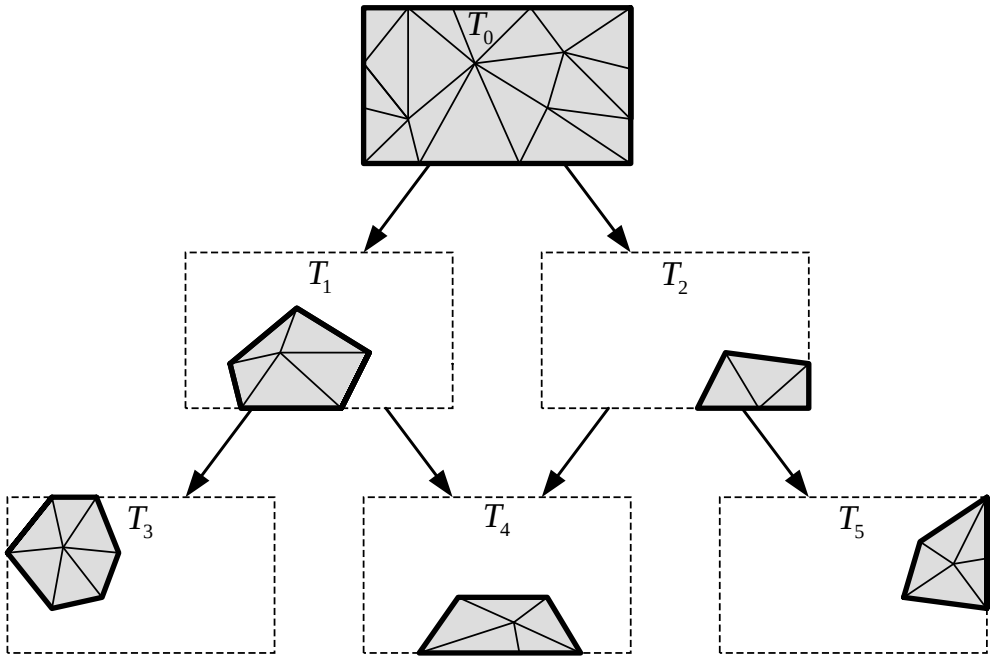


Рис. 72. Ориентированный ациклический граф, описывающий последовательность триангуляций, приведенных на рис. 71

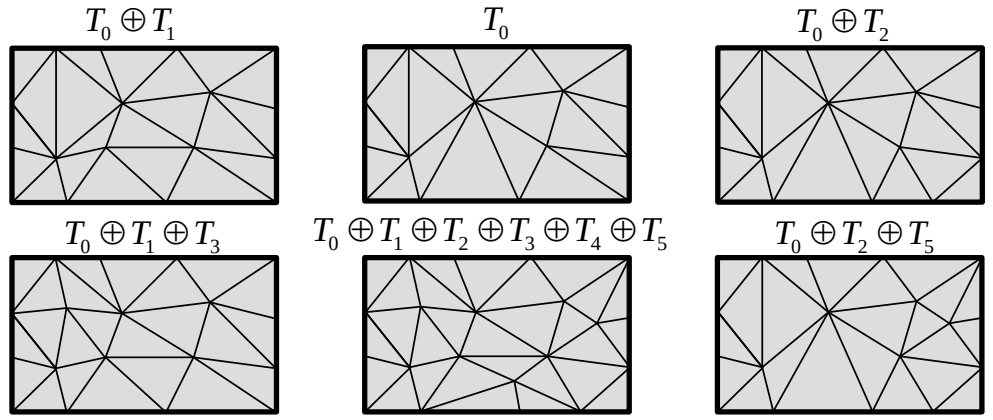


Рис. 73. Некоторые из 16 возможных триангуляций, которые можно получить с помощью мультитриангуляции, приведенной на рис. 72

Мультитриангуляция может быть *увеличивающей* (*уменьшающей*), если для любого ребра графа от T_i к T_j фрагмент T_i содержит меньше треугольников, чем T_j. В нашем примере на рис. 72 мультитриангуляция является увеличивающей.

В качестве фрагмента, являющегося узлом графа мультитриангуляции, могут выступать как группы новых треугольников, появившихся в результате локальных модификаций, так и отдельные треугольники.

10.5. Извлечение триангуляции из мультитриангуляции

Рассмотрим теперь алгоритм, позволяющий извлекать из увеличивающей мультитриангуляции триангуляцию требуемого разрешения.

Алгоритм извлечения.

Пусть дана мультитриангуляция $T = \{T_i\}$ и задана некоторая функция $\mathrm{c}()$, которая определяет, достаточное ли разрешение имеет треугольник t, передаваемый в эту функцию в качестве аргумента. Требуется найти минимальную триангуляцию, состоящую из треугольников из $T = \{T_i\}$, для каждого из которых $\mathrm{c}()$ возвращает истину.

Шаг 1. Для каждого узла T_i мультитриангуляции устанавливаем флаг $b_i := 0$. Каждый треугольник мультитриангуляции помечаем $d_i := 0$. Заносим T_0 в очередь активных узлов.

Шаг 2. Спускаемся по графу мультитриангуляции вниз методом поиска в ширину. Пока очередь активных узлов не пуста, извлекаем из неё фрагмент T_i, и если он имеет флаг $b_i := 0$, то анализируем все треугольники t_k в его составе, имеющие $d_k = 0$. Если условие $\mathrm{c}(t_k)$ выполняется, то помечаем его $d_k := 1$. Если $\mathrm{c}(t_k)$ неверно, то помещаем в очередь активных узлов фрагмент T_j ниже по графу от T_i, который заменяет треугольник t_k, а также устанавливаем $b_j := 1$.

Шаг 3. Теперь необходимо для всех найденных локальных преобразований T_i, имеющих $b_i = 1$, найти все те, без которых их применять нельзя. Для этого нужно методом поиска в ширину подняться по графу мультитриангуляции снизу вверх и пометить значением $b_i := 1$ все узлы T_i, из которых идет ребро в T_j с $b_j = 1$.

Шаг 4. Теперь опять надо спуститься по графу методом поиска в ширину по всем узлам T_i, имеющим флаги $b_i = 1$, и применить соответствующие локальные преобразования. *Конец алгоритма.*

Если функция $\mathrm{c}()$ может быть вычислена за время $O(1)$, то общая трудоёмкость описанного алгоритма составляет $O(N)$ [63].

На практике для задачи интерактивной визуализации модели рельефа в качестве критерия $\mathrm{c}()$ может выступать видимый размер треугольника. Например, если этот размер меньше нескольких пикселей на экране компьютера, то, видимо, дальнейшее дробление треугольника является нецелесообразным.

Другим вариантом для критерия $\mathrm{c}()$ может быть сравнение требуемой от модели точности с заранее вычисленным для каждого треугольника отклонением ε_i от исходной поверхности. В задачах визуализации рельефа

этот критерий может быть дополнительно скомбинирован с предыдущим критерием.

В заключение рассмотрения данного алгоритма отметим, что в задаче интерактивной визуализации триангуляционных моделей поверхностей, когда имеются жёсткие ограничения по максимальному числу одновременно отрисовываемых треугольников на экране, алгоритм извлечения может быть немного модифицирован. На шаге 2 алгоритма в начале цикла можно дополнительно проверить, достигла ли текущая извлекаемая триангуляция максимально допустимого размера, и если так, то алгоритм должен закончить свою работу.

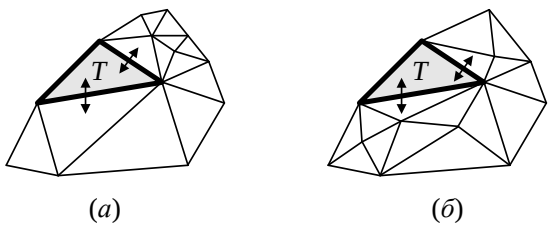
Обратите внимание, что приведённый алгоритм извлекает из мультитриангуляции топологически несвязанную триангуляцию, т.е. в которой нет связей между соседними треугольниками. Этот алгоритм вполне подходит для визуализации триангуляции на экране компьютера, однако неприменим для многих задач обработки и анализа триангуляции, где обязательным условием работы алгоритма является наличие сведений о соседних треугольниках, например при построении изолиний.

Таким образом, для извлечения топологически связанной триангуляции необходим другой алгоритм, а также некоторые изменения структур данных мультитриангуляции.

Основной проблемой при установлении соседства треугольников в извлечённой триангуляции является то, что один и тот же треугольник в различных извлечённых триангуляциях (извлечённых с разными критериями) может иметь разных соседей (рис. 74).

В работе [33] предложено дополнить структуру мультитриангуляции сведениями о всех возможных соседях треугольников, которые могут получиться при всех возможных критериях извлечения. Затем уже после извлечения триангуляции необходимо будет пробежаться по всем треугольникам в извлечённой триангуляции и определить их *актуальных* соседей, т.е. выбрать из списков возможных соседей тех, которые реально получились в результате работы алгоритма извлечения.

Такой подход, к сожалению, приводит к существенному увеличению затрат памяти на представление мультитриангуляции, а также тратит много времени на просмотр всех списков соседних треугольников.



(*а*) (*б*)

Рис. 74. Различные соседи одного и того же треугольника *T* в различных триангуляциях, извлечённых по разным критериям

Более экономичным по затратам памяти является подход, когда для каждого ребра каждого треугольника T хранится только один *наиболее вероятный* сосед, в качестве которого нужно взять треугольник G, который присутствовал в триангуляции в процессе построения мультитриангуляции в момент, когда треугольник T только появился в результате применения некоторого упрощающего преобразования. Если треугольник T присутствует в самой детальной (исходной) триангуляции, то в качестве наиболее вероятных соседей следует взять его соседей по исходной триангуляции.

При таком подходе с хранением только наиболее вероятных соседей возможны три типа ситуации при установлении соседства двух треугольников t_1 и t_2 в извлеченной триангуляции (рис. 75):

1. Обе взаимные ссылки на наиболее вероятные треугольники являются актуальными (рис. 75,*а*).

2. Одна из ссылок на наиболее вероятный треугольник является актуальной, а вторая – нет (рис. 75,*б*). В этом случае, сделав переход через ребро по актуальной ссылке из одного треугольника в другой, мы сможем установить и обратную актуальную ссылку.

3. Обе ссылки являются неактуальными (рис. 75,*в*). На рис. 75,*г* приведён пример последовательности упрощающих преобразований триангуляции T_i, на основе которых построена мультитриангуляция, из которой могла быть извлечена триангуляция $T = T_0 \oplus T_1 \oplus T_2 \oplus T_4$ с обеими неактуальными взаимными ссылками смежных треугольников t_1 и t_2 (рис. 75,*в*).

Для установления актуальных ссылок в таком случае можно использовать алгоритм, построенный на следующей идее.

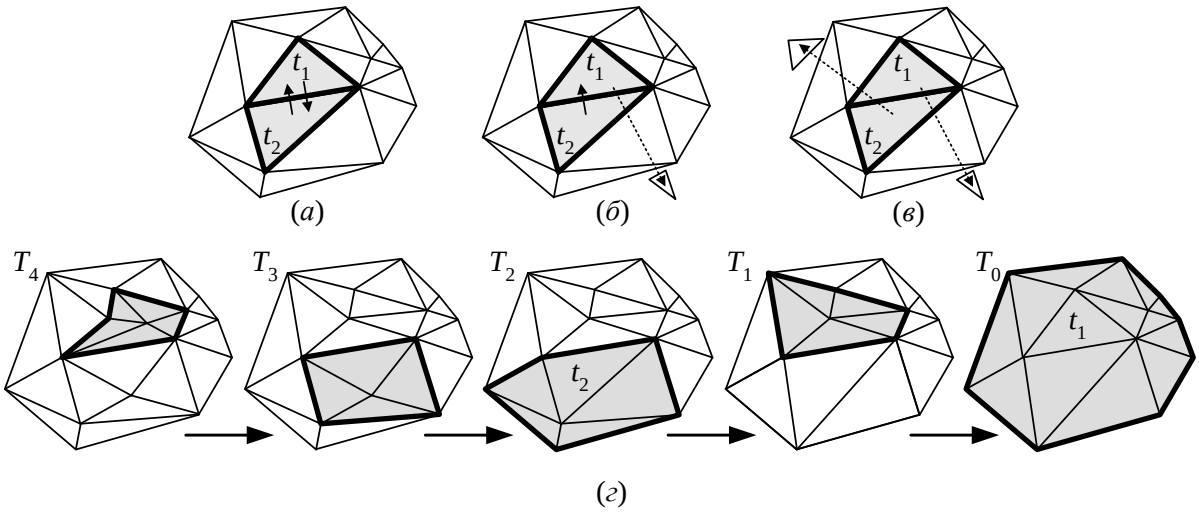


Рис. 75. Различные варианты актуальности наиболее вероятных соседних треугольников: *а* – взаимные ссылки являются актуальными; *б* – актуальна только одна ссылка; *в* – обе ссылки являются неактуальными; *г* – последовательность упрощающих преобразований, на основе которых построена мультитриангуляция

Вначале нужно перейти от треугольников с неактуальными ссылками к фрагментам мультитриангуляции, а затем идти по графу мультитриангуляции вверх до тех пор, пока в каком-то узле не сойдутся пути, ведущие от нескольких треугольников. Эти треугольники необходимо проверить между собой на смежность (геометрически, сравнивая координаты образующих узлов треугольников). Если смежность установлена, то треугольники удаляются из числа отслеживаемых и поиск повторяется, пока не будут отслежены все треугольники. Рассмотрим алгоритм детально.

Алгоритм установления неактуальных ссылок 3-го типа.

Шаг 1. Создаём из всех треугольников t_j с неактуальными ссылками (3-го типа, см. выше) список L из записей вида (T_k, T_i^0, t_j), где $T_k = T_i^0$ – фрагмент, который содержит треугольник t_j. Первый элемент этой записи будет меняться по ходу алгоритма, а второй – нет. Индексы i должны соответствовать порядку образования фрагментов при построении мультитриангуляции. Список L сортируем по возрастанию значения индекса i первого элемента записей.

Шаг 2. Пока список L не пуст, выполняем следующие действия. Берём первую запись в списке L и переходим от T_k вверх по дереву мультитриангуляции к следующему узлу: $T_k := Родитель(T_k)$. Далее восстанавливаем упорядоченность списка L, перемещая запись (T_k, T_i^0, t_j) в новое положение в соответствии с новым значением индекса k. Затем проверяем, есть ли другие записи с тем же значением T_k, но с другим T_i^0. Если есть, то проверяем из этих записей все пары треугольников t_{j_1} и t_{j_2} на предмет смежности. Если смежность установлена и для некоторого треугольника t_j определены все его смежные треугольники, то удаляем соответствующую запись из списка L. *Конец алгоритма.*

Данный алгоритм в целом работает медленнее, чем алгоритм с хранением для каждого ребра всех возможных актуальных соседей, однако он требует меньше дополнительной памяти.

В заключение рассмотрим ещё один алгоритм, который работает не медленнее, чем со списком всех возможных актуальных соседей, но при этом тратит меньшее количество дополнительной памяти.

Ситуация, когда треугольник имеет неактуальную ссылку через некоторое своё ребро, хорошо иллюстрируется приведённым выше рис. 75,*г*. Такая ситуация возникает, когда в мультитриангуляцию входит множество фрагментов с треугольниками, имеющих общие ребра не на границе триангуляции. Например, на рис. 75,*г* такие общие рёбра имеют треугольники во фрагментах T_1 и T_4, а также в T_2 и T_3.

Из этого вытекает следующая идея. Необходимо дополнить структуры данных мультитриангуляции так, чтобы для каждого ребра каждого треугольника в мультитриангуляции были ссылки на возможные упрощающий и детализирующие треугольники. При этом для каждого треугольника может быть до трёх ссылок на детализирующие треугольники (для каждого ребра), но только одна на упрощающий. На рис. 76,*а* приведён пример всех таких дополнительных ссылок между упрощающими и детализирующими треугольниками (ссылки обозначены пунктирными стрелками от упрощающих треугольников к детализирующим). Причем, можно заметить, эти ссылки очень просто вычислить в процессе построения мультитриангуляции (рис. 76,*б*).

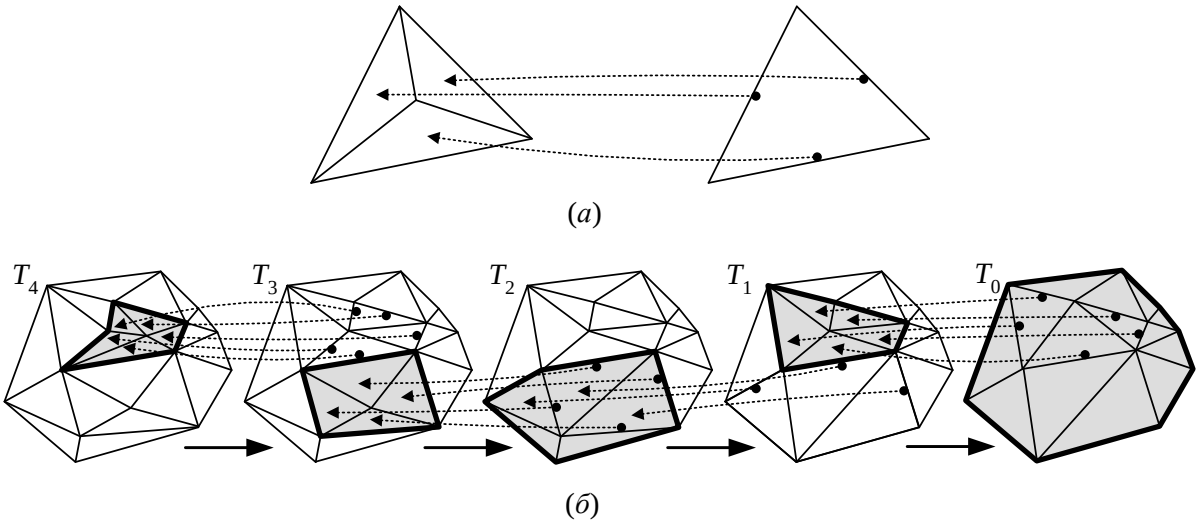


Рис. 76. Дополнительные ссылки между упрощающими и детализирующими треугольниками: а – ссылки влево на детализирующие треугольники и ответные ссылки вправо на упрощающие; *б* – ссылки для примера на рис. 75,*г*

Таким образом, алгоритм извлечения триангуляции из мультитриангуляции с дополнительными ссылками на детализирующие и упрощающие треугольники будет заключаться в следующем. Пусть в триангуляции обнаружена некоторая ссылка из треугольника через некоторое ребро на неактуальный треугольник *t*. Вначале будем переходить от *t* по ссылкам к упрощающим треугольникам и проверять, не являются ли они актуальными. Если среди упрощающих не будет найдено необходимого треугольника, то будет последовательно проверять все детализирующие.

10.6. Пирамида Делоне

Определение 28. *Пирамидой Делоне* называется мультитриангуляция, узлами которой являются отдельные треугольники и из которой можно извлекать только триангуляции Делоне [29] (рис. 77).

Такая структура в целом совпадает со структурой детализации триангуляции Киркпатрика [43], но строится иными способами.

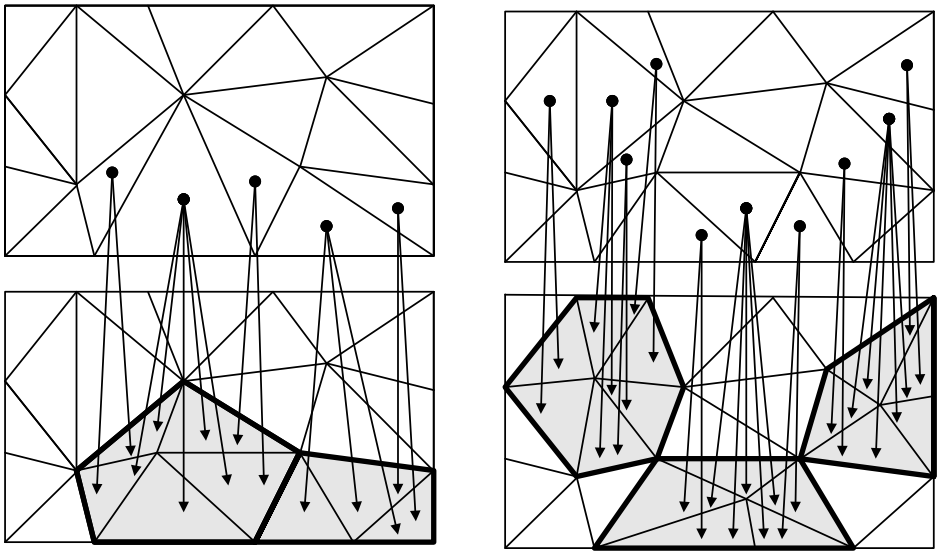


Рис. 77. Связи треугольников в пирамиде Делоне, построенной на последовательности триангуляций, приведенной на рис. 71

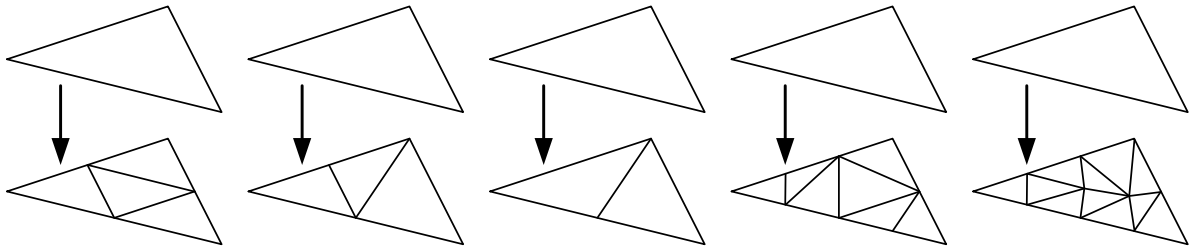
Для построения пирамиды Делоне нужно вначале сгенерировать последовательность триангуляций различного разрешения $\varepsilon_1 < \varepsilon_2 < ... < \varepsilon_h$ с помощью стратегий «сверху вниз» или «снизу вверх», изложенных выше, а затем найти все наложения треугольников между соседними уровнями.

10.7. Детализация триангуляции

Триангуляционные модели рельефа позволяют точно описать форму поверхности, однако во многих алгоритмах анализа требуется, чтобы треугольники были достаточно маленькими. Наиболее остро эта проблема встает при использовании различных методов конечных элементов и при визуализации рельефа. Такая задача, обратная упрощению триангуляции, называется задачей *детализации триангуляции*.

Основным назначением метода детализации триангуляции является повышение качества и точности вычислений. Он может быть применён ко многим обычным алгоритмам анализа поверхностей, таким как построение изолиний, расчёт объёмов земляных работ и зон видимости.

При детализации отдельные треугольники триангуляции разбиваются на меньшие треугольники. На рис. 78 приведены некоторые варианты разбиения треугольников при детализации триангуляции. Наиболее распространенным на практике является вариант с выборочным разбиением рёбер на две части. Вначале среди всех рёбер триангуляции выбираются те рёбра, длина которых превышает некоторый допустимый порог. Затем посередине этих рёбер выполняется вставка новых узлов (рис. 78,*а–в*). В других вариантах рёбра разбиваются на большее число частей, а также добавляются новые узлы внутри треугольников (рис. 78,*г*,*д*).



(*а*) (*б*) (*в*) (*г*) (*д*)

Рис. 78. Способы разбиения треугольников при детализации:
а – разбиение каждого ребра триангуляции на две части;
б, *в* – разбиение только длинных рёбер на две части;
г – разбиение рёбер на переменное число частей;
д – разбиение рёбер и вставка новых узлов

Рассмотрим вопрос определения высоты вновь вставляемых узлов. Самая простая линейная интерполяция по высотам смежных узлов практически не имеет смысла, поэтому на практике используются сплайновые поверхности, методы геостатистики или некоторые приближенные локальные методы. Последняя группа методов является наиболее простой в применении и, как правило, дает приемлемое качество аппроксимации [6].

Алгоритм интерполяции высот на рёбрах триангуляции. Необходимо найти высоты в серединах рёбер.

Шаг 1. Для каждого треугольника вычисляем векторы нормалей касательных (с помощью векторного произведения двух его сторон).

Шаг 2. Для всех узлов n_i триангуляции находим векторы нормалей касательных N_i. Для этого вычисляем среднее взвешенное нормалей всех смежных с узлом треугольников t_j^i. В качестве весов можно использовать следующие варианты (рис. 79) [6]:

1) $w_1 = (2S/a)\cdot\ln(p/(p-a))$, где S – площадь t_j^i, p – полупериметр t_j^i.

2) $w_1 = (\cos\beta - \cos(\alpha+\beta))/(c\sin\beta)$.

3) $w_3 = \sqrt{\sin\alpha}$.

4) $w_4 = \sin\alpha$.

5) $w_5 = \alpha$.

Шаг 3. Строим на каждом ребре AB кубический сплайн. Вначале вычисляем в $A=(x_A, y_A, z_A)$ и $B=(x_B, y_B, z_B)$ направляющие вектора касательных по направлению AB с помощью векторного произведения: $D_{AB}(A) = N_A \times N_{AB} = (dx_A, dy_A, dz_A)$, $D_{AB}(B) = N_B \times N_{AB} = (dx_B, dy_B, dz_B)$. Затем вычисляем $L = \sqrt{(x_B - x_A)^2 + (y_B - y_A)^2}$. В итоге отрезок кубического сплайна на ребре AB задается формулами (тогда, например, подставив в них значение $t = L/2$, можно получить высоту на середине ребра):

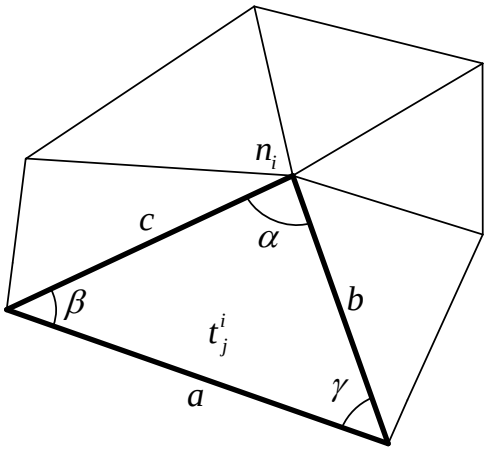


Рис. 79. Вычисление вклада треугольника n_j^i в нормаль касательной плоскости в узле n_i при интерполяции высот в триангуляции

$$x(t) = x_A + (x_B - x_A)\cdot t/L;\ \ y(t) = y_A + (y_B - y_A)\cdot t/L;$$

$$z(t) = c_3\cdot t^3 + c_2\cdot t^2 + c_1\cdot t + c_0,\ \text{где}\ c_0 = z_A,\ c_1 = z'_A;$$

$$c_2 = \left(3(z_B - z_A) + (2z'_A + z'_B)\cdot L\right)/L^2;\ c_3 = \left((z'_A + z'_B)\cdot L - 2(z_B - z_A) + \right)/L^3;$$

$$z'_A = dz_A/\sqrt{dx_A^2 + dy_A^2};\ \ z'_B = dz_B/\sqrt{dx_B^2 + dy_B^2}\ .\ \underline{\textit{Конец алгоритма.}}$$

10.8. Сжатие триангуляции

Как было сказано выше, реальные модели рельефа требуют огромных массивов памяти для хранения. Были затронуты проблемы обработки триангуляций, находящихся в памяти компьютера. В этом разделе будет рассмотрена задача компактного сохранения триангуляции в некоторый битовый поток долговременной памяти (например, на жёстком диске).

Задачу сжатия структуры триангуляции можно условно разбить на две составляющие: 1) *сжатие координат узлов* и 2) *сжатие топологии* (структуры графа триангуляции).

В п. 1.2 было показано, что при использовании распространённых структур данных затрачивается 8–16 байт на хранение координат узлов (при 4- или 8-байтовом представлении координат) и 28–72 байта на топологические связи объектов триангуляции. Видно, что наибольшую долю памяти (до 90%) занимает топология триангуляции.

Один из классических методов упаковки триангуляции заключается в разбиении триангуляции на некоторые *полосы* – последовательности смежных треугольников. Однако такой способ лучше всего подходит для визуализации, так как полная топология триангуляции при этом не сохраняется. Другой проблемой здесь является выбор минимального количества полос. В [28] показано, что эта задача является NP-полной.

Среди методов упаковки, сохраняющих топологию, одним из наиболее простых и удобных в применении является *метод шелушения* [31]. В работе алгоритмов сжатия/распаковки поддерживается некоторый *граничный многоугольник*, охватывающий область обработанных треугольников. Кроме того, имеется очередь *активных рёбер* триангуляции, т.е. рёбер, входящих в состав граничного многоугольника, но еще не обработанных. При сохранении триангуляции в выходной поток записывается с помощью 2 бит один из управляющих кодов: **VERTEX**, **SKIP**, **LEFT** или **RIGHT**. После кода **VERTEX** всегда идут 2 координаты некоторой вершины. Рассмотрим соответствующие алгоритмы этого метода.

Алгоритм упаковки триангуляции методом шелушения.

Шаг 1. Выбирается любой треугольник в триангуляции. В выходной поток посылаются координаты 3 образующих узлов этого треугольника. Три его ребра образуют начальный *граничный многоугольник* и входят в состав очереди активных рёбер (рис. 80,*а*).

Шаг 2. Пока очередь активных рёбер не пуста, извлекаем из её начала ребро *r* и пытаемся увеличить *граничный многоугольник* за счёт треугольника *t*, смежного с *r* с внешней стороны от текущей границы:

Шаг 2.1. Если узел *n* в *t* напротив ребра *r* не лежит на *граничном многоугольнике*, то посылаем в поток код **VERTEX** и координаты узла *n*, увеличиваем границу и очередь активных рёбер (рис. 80,*г*).

Шаг 2.2. Если узел *n* в *t* является следующим вдоль границы слева от ребра *r*, то посылаем код **LEFT** и увеличиваем границу и очередь активных рёбер (рис. 80,*б*).

Шаг 2.3. Если узел *n* в *t* является следующим вдоль границы справа от ребра *r*, то посылаем код **RIGHT** и увеличиваем границу и очередь активных рёбер (рис. 80,*в*).

Шаг 2.4. Если треугольника *t* не существует (рис. 80,*д*) или узел *n* в *t* не является смежным вдоль границы к ребру *r* (рис. 80,*е*), то посылаем в поток код **SKIP**. *Конец алгоритма.*

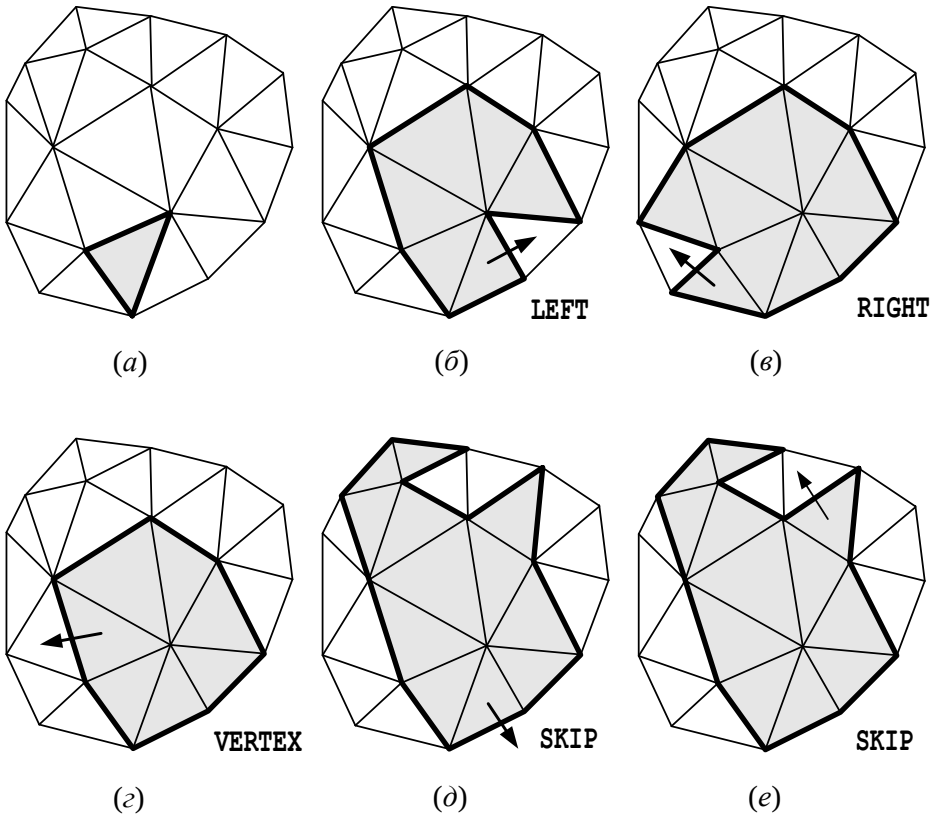
Аналогично построен и алгоритм распаковки.

Алгоритм распаковки триангуляции.

Шаг 1. Из входного потока считываются координаты трёх узлов, и на них строится треугольник, рёбра которого образуют начальный граничный многоугольник и входят в состав очереди активных рёбер.

Шаг 2. Пока очередь активных рёбер не пуста, извлекаем из её начала ребро *r*, пытаемся увеличить *граничный многоугольник* от ребра *r* в соответствии со считываемым из потока управляющим кодом:

Шаг 2.1. Для кода **VERTEX**: создаем новый узел *n* и считываем его координаты из потока. На ребре *r* и узле *n* создаем новый треугольник, увеличиваем границу и очередь активных рёбер.



(*а*) (*б*) (*в*)

(*г*) (*д*) (*е*)

Рис. 80. Упаковка триангуляции методом шелушения:
а – выбор начального треугольника; *б* – треугольник, замыкаемый налево; *в* – треугольник, замыкаемый направо; *г* – треугольник с новым узлом; *д* – треугольник не существует; *е* – треугольник замыкается некорректно

Шаг 2.2. Для кода **LEFT**: создаем новый треугольник на ребре *r* и узле, следующем вдоль границы слева от ребра *r*. Увеличиваем границу и очередь активных рёбер.

Шаг 2.3. Для кода **RIGHT**: создаем новый треугольник на ребре *r* и узле, следующем вдоль границы справа от ребра *r*. Увеличиваем границу и очередь активных рёбер.

Шаг 2.4. Для кода **SKIP**: ничего не делаем. *Конец алгоритма.*

Описанный метод шелушения сжимает топологические связи триангуляции, затрачивая в среднем на один узел примерно 4,2–4,4 бита, т.е. в 50–140 раз [31].

Данный алгоритм можно адаптировать для сжатия координат узлов. Для этого надо сохранять координаты узлов отдельно во второй поток, причем координаты очередного узла лучше представлять в относительных координатах от предыдущего сохранённого узла. После этого можно применить для второго потока какой-нибудь универсальный метод сжатия. В [70] описываются специальные методы сжатия такого потока, основанные на квантизации и методе кодирования энтропии.

Глава 11. Стрипификация триангуляции

11.1. Определения

В настоящее время для описания трёхмерных моделей графических объектов, а точнее их поверхностей, наиболее часто используются триангуляционные модели – *меши* (англ. *mesh* – ячейка сети). Это связано в первую очередь с тем, что в современном программно-аппаратном обеспечении (видеокартах и популярных библиотеках трёхмерной графики, в т.ч. OpenGL и DirectX) основным графическим примитивом является треугольник. Именно поэтому, даже если система трёхмерного моделирования построена не на триангуляционном методе, то она вынуждена формировать для моделируемых объектов соответствующие меши, которые затем передаются для отрисовки через OpenGL или DirectX в видеокарту.

Для вывода меша на экран можно передать в видеокарту для каждого треугольника меша трёхмерные координаты образующих его вершин. Однако для сокращения объёма передаваемых в видеокарту данных OpenGL и DirectX поддерживают ещё два способа – вывод полос треугольников: последовательных (*triangle strip*) и веерных (*triangle fan*). В этих способах для первого треугольника в последовательности задаются все 3 вершины, а для последующих смежных треугольников – только одна. То есть при достаточно большой длине последовательности общее число вершин, передаваемых для отрисовки в видеокарту, сокращается почти трёхкратно.

Типичные триангуляционные модели непредставимы в виде одной полосы. Поэтому возникает задача *стрипификации* триангуляции (*разделения на полосы*), которая заключается в разбиении триангуляции на некоторое небольшое число полос, минимизирующее объём данных, передаваемых в видеокарту для отрисовки (рис. 81). Если триангуляцию разрешается разбивать только на последовательные или веерные полосы, то эта задача эквивалентна минимизации числа полос.

В теории выделяют 3 вида полос: *последовательные* (англ. *strip*), *веерные* (англ. *fan*) и *произвольные* (англ. *general*). Полосы задаются в виде наборов узлов триангуляции $v_1, v_2, ..., v_k, k \ge 3$. В последовательных полосах треугольники определяются набором последовательно заданных узлов $v_i, v_{i+1}, v_{i+2}, i = \overline{1, k-2}$ (рис. 82,*а*), в веерных – первым узлом и парами последовательных узлов $v_1, v_i, v_{i+1}, i = \overline{2, k-1}$ (рис. 82,*б*), а в произвольных – узлами $v_j, v_i, v_{i+1}, i = \overline{2, k-1}$, где v_j, v_i – узлы предыдущего смежного треугольника (рис. 82,*в*). В вырожденном случае $k = 3$ (один треугольник в полосе) полосу можно трактовать как веерную, как последовательную и как произвольную.

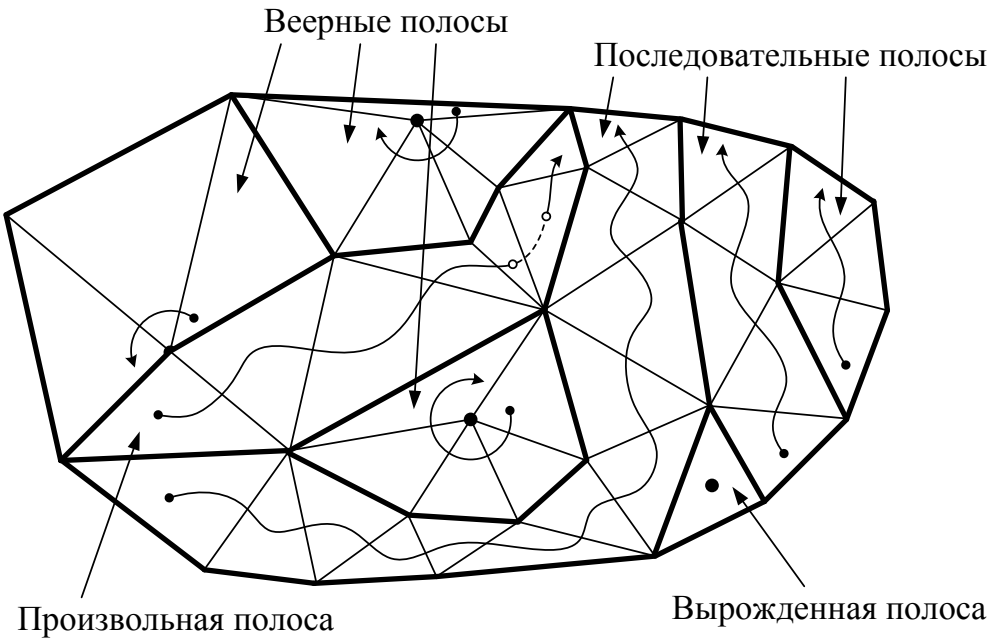


Рис. 81. Пример стрипификации триангуляции

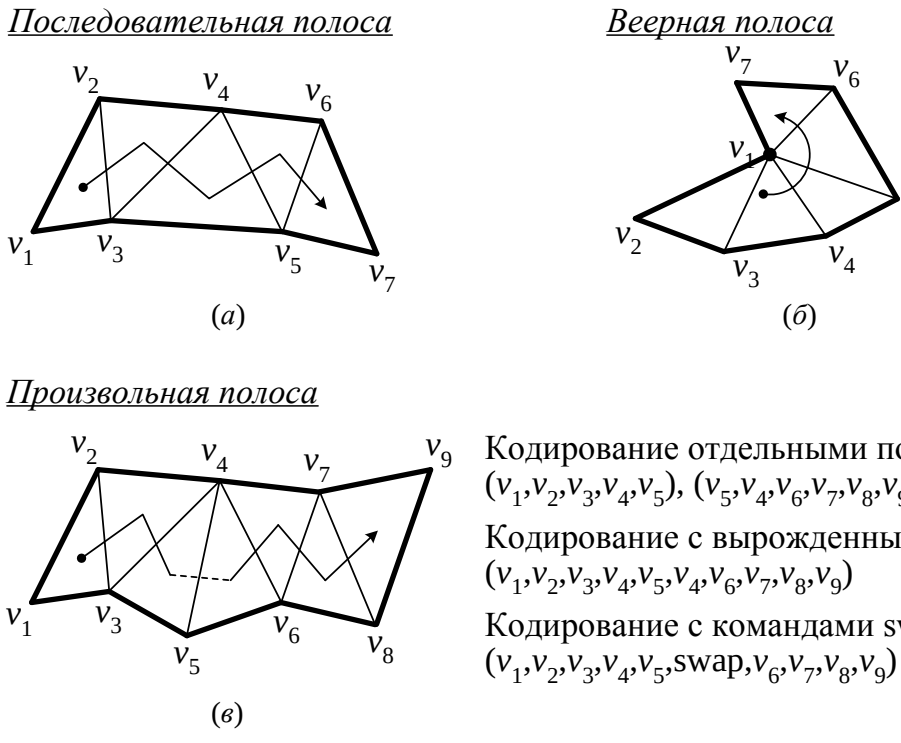


Рис. 82. Пример кодирования полос разного вида

На рис. 82,*в* приведена произвольная полоса, не являющаяся последовательной. Для её отрисовки можно передать в видеокарту две независимые последовательные полосы, суммарно затратив для их описания 11 вершин. Однако если представить ребро v_4, v_5 как вырожденный треугольник v_4, v_5, v_4, то полоса станет последовательной, и тогда для коди-

рования достаточно будет только 10 вершин. Такой подход с кодированием с вырожденными треугольниками используется в OpenGL и DirectX.

В некоторых других библиотеках трёхмерной графики (например, GL [38], которая стала прообразом OpenGL) применяется другой способ кодирования, основанный на использовании во входном потоке вершин специальных команд swap. При этом для выполнения отрисовки создаётся вспомогательная очередь из двух вершин. Если во входном потоке имеется вершина, то она помещается в очередь и треугольник, образованный тремя последними в очереди вершинами, отрисовывается. Если во входном потоке имеется команда swap, то две последние в очереди вершины меняются местами (рис. 83). Такой подход для кодирования примера на рис. 82,*в* потребует 9 вершин и 1 команду swap.

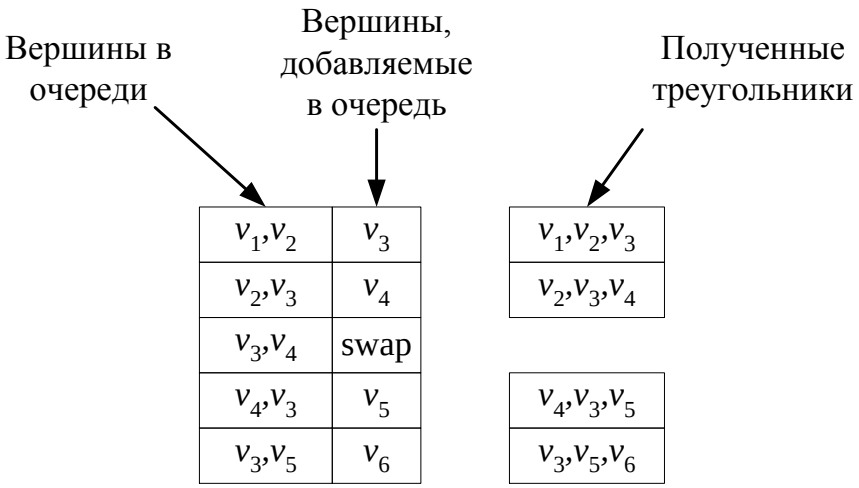


Рис. 83. Пример кодирования полос с командой swap

Описываемые в настоящем разделе алгоритмы применимы не только для плоской триангуляции, определенной в п. 1.1, но и для произвольных мешей – наборов треугольников в пространстве, описывающих поверхности произвольных объектов. Важным свойством меша является то, что каждое ребро любого треугольника должно принадлежать не более чем 2 треугольникам меша. Именно поэтому для описания мешей обычно применяется структура данных «Узлы и треугольники» (см. п. 1.3.4), используемая также для описания плоских триангуляций. В этой структуре для каждого ребра треугольника хранятся ссылки на соседние треугольники, имеющие общее ребро с данным треугольником.

Общие затраты памяти на описание меша с помощью полос (не считая команд swap) составляют $2S+T$ вершин, где S – общее число полос, а T – число треугольников в меше. Отсюда следует, что наибольшая экономия памяти в описании меша достигается при $S=1$. Однако реальные меши, как правило, невозможно описать с помощью одной-единственной полосы. Для того чтобы это понять, достаточно перейти от триангуляции к

двойственному ей графу смежных треугольников (заметьте, что этот граф можно получить для триангуляции Делоне как подмножество диаграмм Вороного). Задача поиска единственной произвольной полосы на триангуляции эквивалентна нахождению гамильтонова цикла в двойственном графе (рис. 84). А эта задача, как известно, является NP-полной [34].

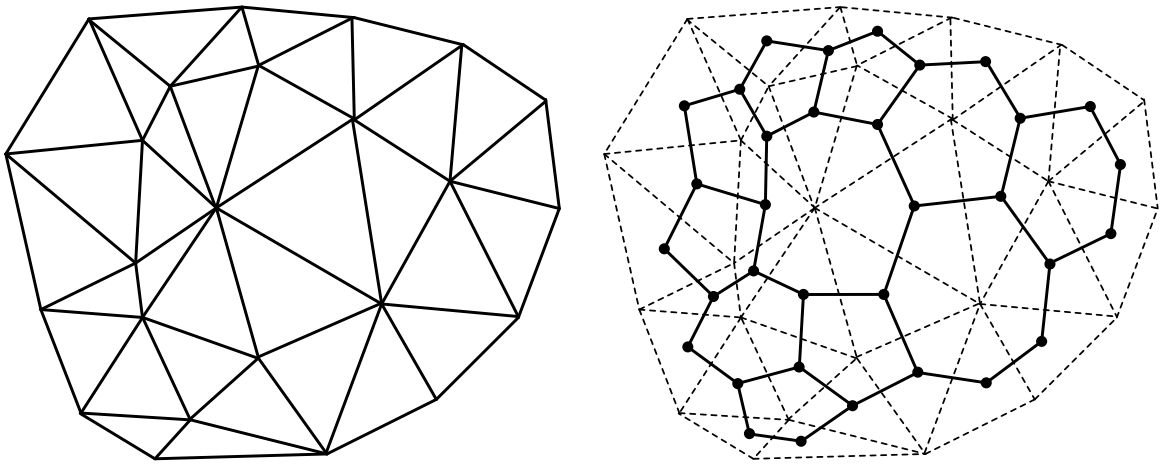


Рис. 84. Пример триангуляции (слева) и двойственного ей графа смежных треугольников (справа)

Триангуляция, двойственный граф которой содержит гамильтонов цикл, называется также *гамильтоновой*. Гамильтонова триангуляция всегда может быть представлена в виде одной полосы треугольников с использованием некоторого числа команд swap. Если для описания этой полосы не использовано ни одной команды swap, значит, эта полоса является последовательной, а сама триангуляция называется *последовательной*.

К сожалению, далеко не каждая триангуляция является Гамильтоновой. Однако известно, что в неё всегда можно добавить некоторое число дополнительных узлов – *точек Штейнера* (Steiner), и триангуляция станет гамильтоновой. Точки Штейнера в терминах триангуляции – это дополнительные узлы, добавляемые в триангуляцию, разбивающие некоторые ребра пополам и добавляющие два новых треугольника (рис. 85).

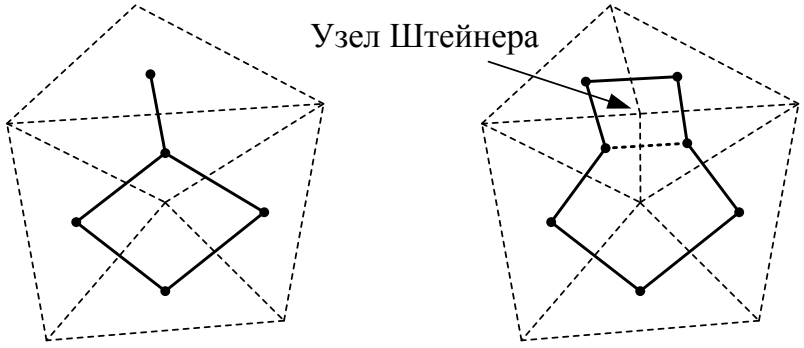


Рис. 85. Пример триангуляции без гамильтонова цикла (слева) и с гамильтоновым циклом (справа)

К понятию гамильтоновой триангуляции близким понятием является *обобщенная последовательная триангуляция*. В ней достаточно только наличия незамкнутого пути в графе, обходящего все вершины графа ровно 1 раз (в гамильтоновой триангуляции этот путь должен быть замкнут).

В [35] доказано, что задача разбиения триангуляции на минимальное число полос также является NP-полной. Именно поэтому на практике точные алгоритмы практически неприменимы, что вынуждает использовать различные приближенные алгоритмы.

11.2. SGI-алгоритм

В работе [21] описан один из наиболее простых приближённых алгоритмов стрипификации. Он основан на жадной стратегии.

В начале работы алгоритма для всех треугольников вычисляется число соседних треугольников (рис. 86,*а*). Эти числа будут использованы в качестве счётчиков количества соседей, ещё не вошедших в полосы. В дальнейшем по мере помещения треугольников в те или иные полосы, счётчики у их соседей должны будут уменьшаться.

Алгоритм работает, пока имеется хотя бы один треугольник, не вошедший ни в одну полосу. Для построения очередной полосы прежде всего выбирается начальный треугольник: среди ещё не использованных треугольников выбирается такой, который имеет наименьшее значение счётчика, т.е. имеет наименьшее количество соседей (рис. 86,*б*,*е*). После выбора треугольника сразу же уменьшаются на 1 счётчики у всех его соседей. Начальный треугольник объявляется текущим в полосе. Затем в цикле для

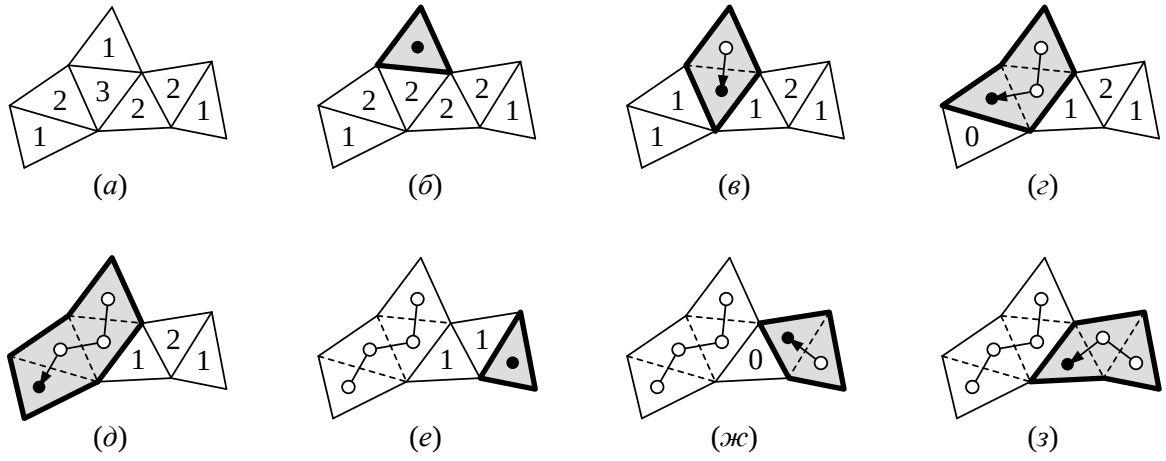


Рис. 86. Схема работы SGI-алгоритма: *а* – исходная триангуляция с вычисленными количествами смежных треугольников; *б* – выбор первого треугольника первой полосы; *в–д* – выбор следующих треугольников; *е* – выбор первого треугольника второй полосы; *ж–з* – выбор следующих треугольников второй полосы

текущего треугольника из числа смежных выбирается новый, ещё не использованный треугольник, который имеет минимальное значение счётчика (если минимальное значение счётчика имеет не один смежный треугольник, то выбор между ними делается произвольно). Выбранный треугольник добавляется в полосу и становится текущим (рис. 86,*б–д*,*ж–з*).

Основным недостатком алгоритма является большое число команд swap в генерируемых полосах. Однако это компенсируется очень высокой скоростью работы – трудоёмкость алгоритма линейная, а также простотой реализации.

11.3. Взвешенный SGI-алгоритм

В SGI-алгоритме, описанном выше, для выбора очередного треугольника используются некоторые числа, характеризующие треугольники. Во *взвешенном SGI-алгоритме* [44] к этим числам добавляются дополнительные эвристические слагаемые, позволяющие уменьшить количество команд swap и общее число полос. Таким образом, для каждого треугольника во взвешенном SGI-алгоритме вычисляется сумма трёх эвристических значений, определяемых следующим образом:

1. Количество соседей текущего треугольника, ещё не включенных в полосы (рис. 86). Эта эвристика является единственной в SGI-алгоритме; она предназначена для уменьшения количества коротких полос.

2. В этой эвристике для двух узлов текущего треугольника определяется количество соседних треугольников, ещё не включенных в полосы. Если эти значения окажутся одинаковыми, то эвристики двух возможных следующих треугольников делаются равными нулю. Если количество соседей у какого-то узла окажется больше, чем у другого, то соответствующему треугольнику присваивается значение +1, а меньшему −1 (рис. 87,*а*).

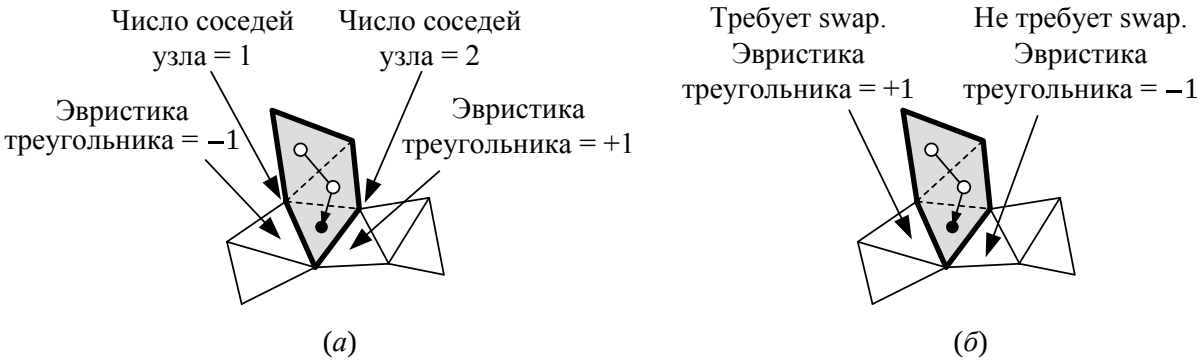


Рис. 87. Вычисление эвристик во взвешенном SGI-алгоритме:
а – количество соседей двух узлов текущего треугольника;
б – проверка необходимости использования команды swap

Эта эвристика сохраняет для последующего использования треугольники с большим числом соседей, тем самым оставляя бóльшую свободу выбора в последующем при построении полос.

3. Для двух возможных следующих треугольников проверяется, требуется ли использование команды swap. Если требуется, то эвристика считается равной +1, иначе −1 (рис. 87,*б*). Данная эвристика позволяет снизить общее число команд swap в генерируемых полосах треугольников.

В итоге, просуммировав значения этих трёх эвристик и получив *веса* треугольников, как и в предыдущем алгоритме, выбирается треугольник, имеющий меньший вес.

В настоящее время взвешенный SGI-алгоритм является одним из наиболее популярных, так как имеет линейную трудоёмкость и очень прост в реализации.

11.4. Алгоритм на основе дерева предшествования

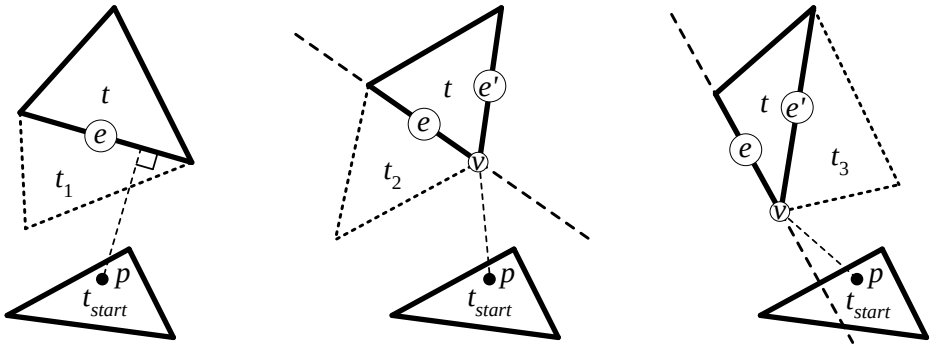
Алгоритм, представленный в работе [68], применим только для плоских выпуклых триангуляций. Он основан на *отношении предшествования* между треугольниками триангуляции.

Для того чтобы задать это *отношение предшествования*, нужно выбрать некоторый начальный треугольник t_{start} и любую точку p внутри это треугольника. Далее для всех остальных треугольников t триангуляции ($t \neq t_{start}$) предшествующий треугольник находится следующим образом. В треугольнике t определяется ближайшая к p точка. Если эта точка находится внутри некоторого ребра e (не на вершине треугольника), то *предшествующим* треугольнику t считается треугольник t_1, смежный с t через ребро e (рис. 88,*а*). В противном случае эта ближайшая точка совпадает с некоторым узлом v триангуляции. Тогда мы должны проанализировать два ребра триангуляции e и e', смежных с этим узлом v (при этом обозначение e и e' должно быть таким, что поворот от e к e' должен быть против часовой стрелки). Если точка p и треугольник t лежат по разную сторону от прямой, проходящей через ребро e, то предшествующим треугольнику t считается треугольник t_2, смежный с t через ребро e (рис. 88,*б*), иначе – треугольник t_3, смежный с t через ребро e' (рис. 88,*в*).

Таким образом, для каждого треугольника триангуляции определяется некоторый предшествующий треугольник. Очевидно, что графом таких отношений является дерево с корнем в треугольнике t_{start} (рис. 89,*а*). Это дерево обладает одним важным свойством: ветви, исходящие из корня, как правило, идут по треугольникам «влево-вправо». Это означает, что эти ветви могли бы стать основой полос триангуляции, причем для их кодирования нужно небольшое число команд swap. Ещё одним достоин-

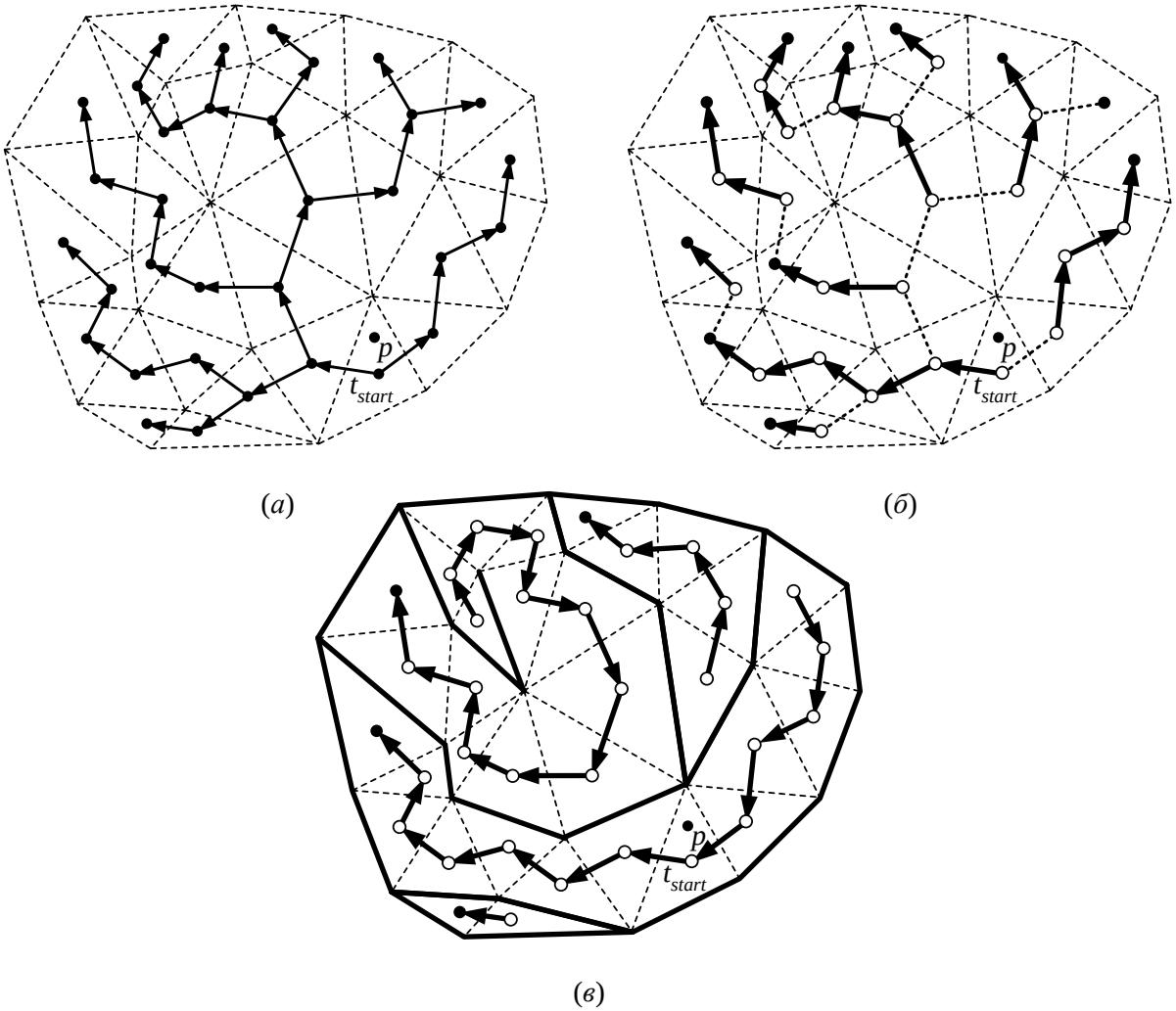
ством дерева предшествования является то, что для его представления не нужно никакой памяти – оно неявно представлено самой триангуляцией.

Далее по дереву предшествования методом поиска в глубину формируются полосы треугольников, причем так, чтобы не использовать коман-



(*а*) (*б*) (*в*)

Рис. 88. Отношение предшествования треугольников в триангуляции относительно точки p в начальном треугольнике t_{start}



(*а*) (*б*)

(*в*)

Рис. 89. Дерево отношения предшествования треугольников в триангуляции относительно точки p в начальном треугольнике t_{start}

ды swap (рис. 89,*б*). После этого следует просмотреть в триангуляции все пары соседних треугольников, и если в них находятся концы двух разных полос, то эти полосы надо объединить в одну полосу (рис. 89,*в*).

Трудоёмкость описанного алгоритма на основе дерева предшествования является линейной.

11.5. Быстрый генератор полос треугольников

В работе [5] предложен алгоритм быстрого генерирования полос треугольников (Fast Triangle Strip Generator – FTSG), который также основан на структуре остовного дерева. Однако, в отличие от предыдущего алгоритма, остовное дерево строится из графа смежности треугольников триангуляции другим способом. Рассмотрим этот алгоритм по шагам.

Алгоритм FTSG

Шаг 1. На графе смежности треугольников триангуляции строится остовное дерево алгоритма поиска в глубину, начиная с некоторого начального треугольника. При выборе возможного направления спуска по дереву следует выбирать следующий треугольник на основе эвристик взвешенного SGI-алгоритма (см. п. 11.3).

Шаг 2. (*Отщепления полос*). В цикле в остовном дереве выбирается узел максимальной глубины, имеющий два потомка. Если такого узла нет, то оставшаяся часть дерева становится полосой треугольников и цикл заканчивается. Если такой узел найден, то из этого узла и двух его деревьев-потомков формируется полоса треугольников.

Шаг 3. Все полосы, полученные на предыдущем шаге, необходимо разбить на последовательные и веерные.

Шаг 4. Все последовательные и веерные полосы, полученные на предыдущем шаге, необходимо попытаться объединить в более длинные. Объединение должно происходить в две фазы. В первой фазе выполняется объединение всех веерных полос и таких последовательных полос, которые не требуют дополнительных команд swap. Во второй фазе объединяются все остальные последовательные полосы. *Конец алгоритма.*

Таким образом, главное отличие данного алгоритма от предыдущего заключается в способе первичной генерации полос. В предыдущем полосы строятся методом поиска в глубину (подход «сверху-вниз» от корня вглубь дерева). В данном алгоритме полосы строятся методом поиска в ширину (подход «снизу-вверх», формируя полосы от самых глубоких узлов в дереве, имеющих два потомка).

В целом этот алгоритм даёт результат немного лучший, чем предыдущий алгоритм на основе остовного дерева.

Трудоёмкость описанного алгоритма FTSG является линейной.

11.6. Туннельный алгоритм

В основе туннельного алгоритма лежит понятие *туннеля* – пути в двойственном графе, состоящего из перемежающейся последовательности *полосовых* и *внеполосовых* рёбер, соответственно входящих и не входящих в полосы треугольников, причём эта последовательность должна начинаться и заканчиваться внеполосовыми ребрами.

Суть туннельного алгоритма заключается в поиске необходимых туннелей в двойственном графе (рис. 90,*а*) и инвертировании всех рёбер в туннеле (рис. 90,*б*): каждое внеполосовое ребро должно быть включено в некоторую полосу, а полосовые рёбра должны быть оттуда исключены. Смысл инвертирования заключается в том, что, если при этом не образуется замкнутых путей в двойственном графе (зацикленных полос треугольников), то общее число полос уменьшится на единицу (рис. 90,*в–г*). Поэтому главной частью туннельного алгоритма является поиск туннелей, не образующих после инвертирования замкнутых путей.

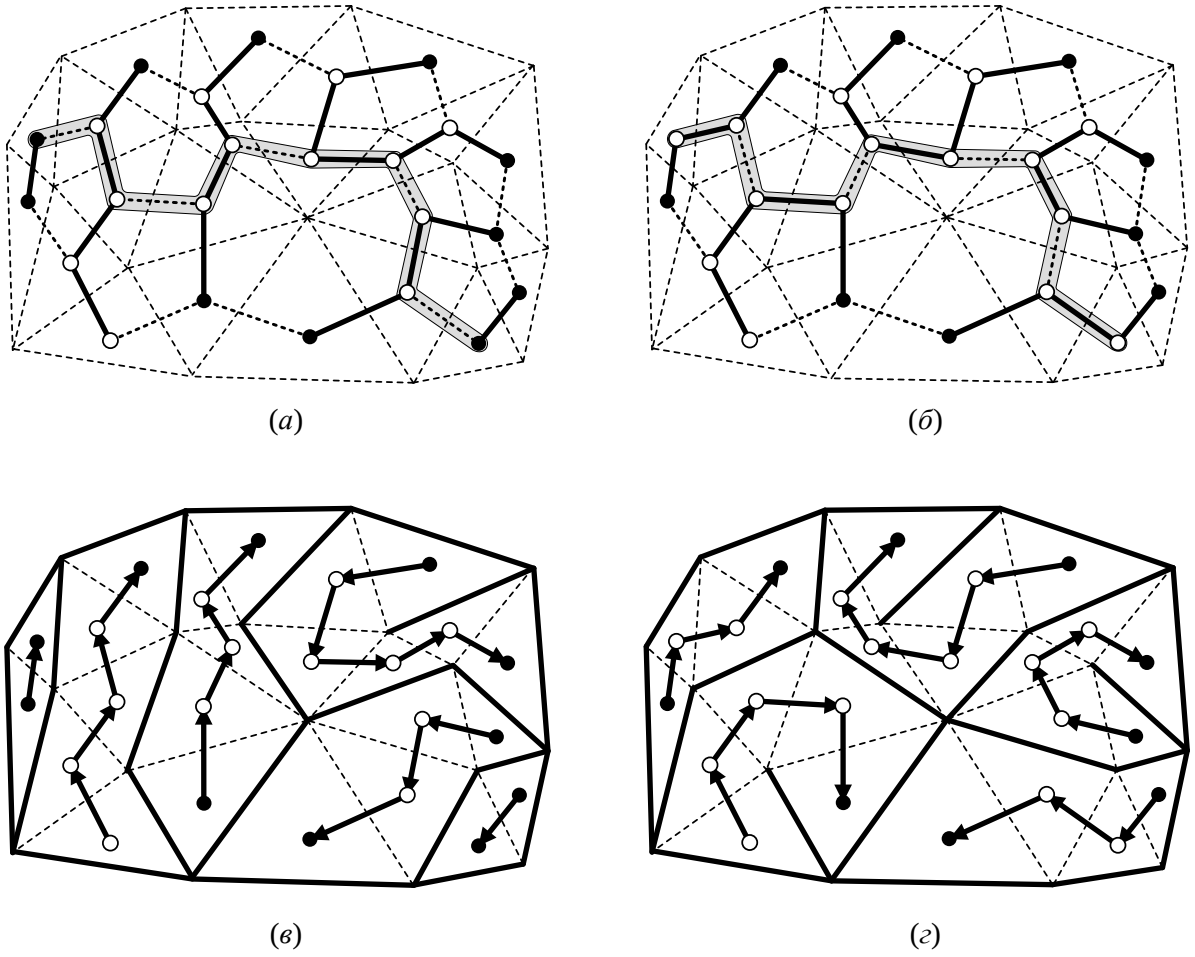


Рис. 90. Пример инвертирования туннеля: *а* – туннель в двойственном графе до инвертирования рёбер; *б* – туннель после инвертирования; *в* – стрипификация до инвертирования; *г* – стрипификация после инвертирования

В туннельном алгоритме вначале выбирается некоторое начало туннеля – некоторая вершина в двойственном графе, соответствующая концу некоторой полосы треугольников. Затем методом поиска в ширину ищется кратчайшая перемежающаяся последовательность рёбер, приводящая в другую вершину, соответствующая концу некоторой полосы треугольников. Затем выполняется инвертирование рёбер в туннеле. Поиск туннелей выполняется до тех пор, пока можно найти туннели. По окончании этого цикла достигается некоторый *локальный* минимум количества полос.

Следует отметить, что туннели всегда состоят из нечётного числа рёбер. В крайнем случае, туннель может состоять только из одного (неполосового) ребра. Инвертирование такого туннеля, по сути, будет означать соединение двух смежных полос треугольников.

Для устранения возможности образования зацикленных полос треугольников после инвертирования туннелей, при поиске туннелей необходимо учитывать следующие ограничения:

1. Последнее (неполосовое) ребро в туннеле не должно соединять два узла, принадлежащих одной полосе треугольников (рис. 91,*а*).

2. Если при поиске в туннель включено некоторое неполосовое ребро, соединяющее два узла одной полосы треугольников, то следующее полосовое ребро должно выбираться таким образом. Будем считать, что каждая полоса треугольников имеет ориентацию, получаемую в результате поиска в ширину направленной от начальной вершины. Тогда следующее включаемое полосовое ребро должно иметь ориентацию, противоположную направлению поиска в ширину (рис. 91,*в*).

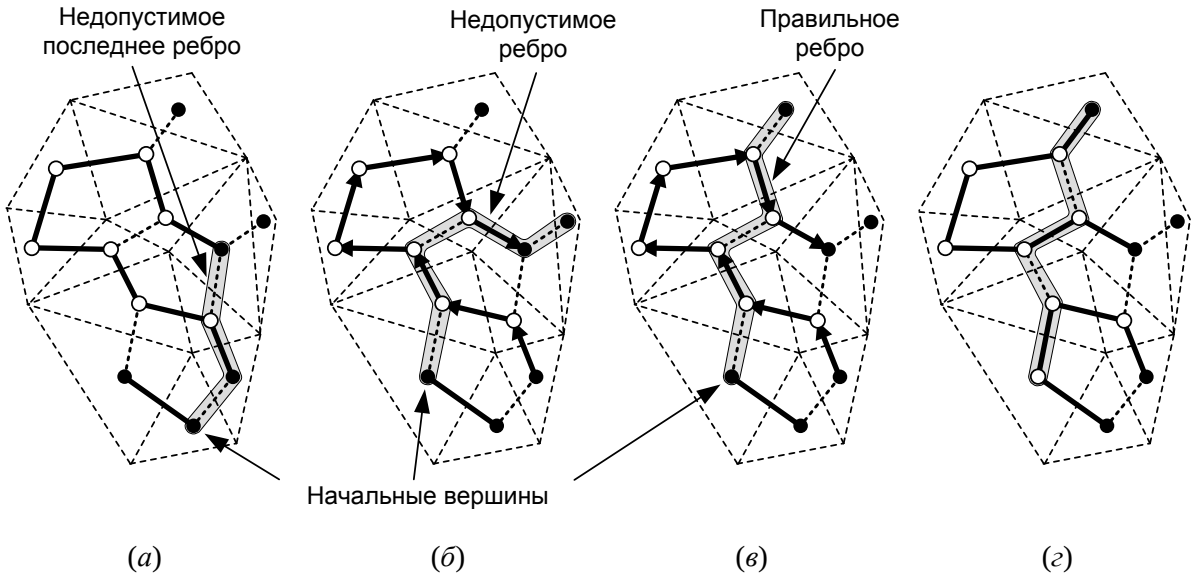


Рис. 91. Предотвращение зацикленности полос треугольников при поиске туннелей: *а* – недопустимый выбор последнего ребра в туннеле; *б* – недопустимый выбор полосного ребра; *в* – правильный выбор полосного ребра; *г* – результат инвертирования туннеля, полученного на рис. *в*

Среди всего множества известных алгоритмов туннельный позволяет получить самое маленькое число полос. В среднем он даёт результаты примерно в 5 раз лучшие, чем SGI-алгоритм. Однако при этом он и работает приблизительно в 5 раз дольше.

11.7. Стрипификация полигональных моделей

В данном разделе решается несколько более общая задача, чем в предыдущих разделах. Здесь рассматривается задача стрипификации не мешей (триангуляционных моделей), а произвольных *полигональных моделей* – моделей поверхностей, состоящих из наборов смежных многоугольников. Для таких полигональных моделей задача стрипификации заключается в триангулировании каждого нетреугольного многоугольника и генерации полос треугольников.

Так как большинство многоугольников можно триангулировать различными способами, то задача стрипификации полигональных моделей имеет несколько большую свободу в выборе полос треугольников, а потому и позволяет получить лучший результат, чем при стрипификации только триангуляционных моделей.

Для стрипификации полигональных моделей часто используются следующие основные подходы [28]:

1. *Статическая триангуляция*. Триангуляция всех нетреугольных многоугольников и последующая стрипификация общего меша любым алгоритмом, приведенным в предыдущих разделах.

2. *Полная динамическая триангуляция встречающихся полигонов*. При генерации полос SGI-алгоритмом (или его модификациями), если очередная полоса выходит в нетреугольный полигон, то этот полигон триангулируется, причём способом, позволяющим включить в полосу максимальное число или даже все треугольники полигона и по возможности не требующим команд swap (рис. 92,*а–в*). Если полигон является выпуклым, то в полосу можно включить все его треугольники, а команды swap не нужны (рис. 92,*г*).

Этот (динамический) подход даёт результат в среднем на 15% лучше, чем предыдущий (статический) [28].

3. *Частичная динамическая триангуляция встречающихся полигонов*. Отличие данного подхода от предыдущего заключается в том, что при входе полосы в очередной полигон может выполняться не полная его триангуляция, а в зависимости от ситуации только некоторая его часть. Когда полоса входит в полигон, на основе весов следует определить наиболее предпочтительную сторону полигона, через которую полоса должна выйти. И только после этого следует построить минимальную (по числу треугольников) полосу между входной и выходной сторонами полигона.

Данный подход обычно даёт результат хуже, чем предыдущий, и только в редких случаях лучше.

4. *Стрипификация четырёхугольных областей четырёхугольников* (алгоритм STRIPE). Данный подход применим для часто встречающихся на практике полигональных моделей поверхностей, состоящих преимущественно из четырёхугольников и частично треугольников.

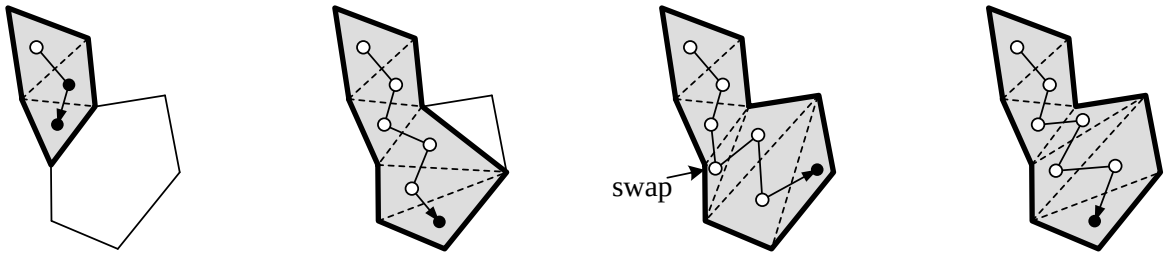


Рис. 92. Варианты триангуляции полигонов при выходе полосы в полигон:
а – полоса подошла к многоугольнику;
б – неудачная триангуляция с неполным включением в полосу;
в – триангуляция, требующая использования команды swap;
г – триангуляция, не требующая использования команды swap

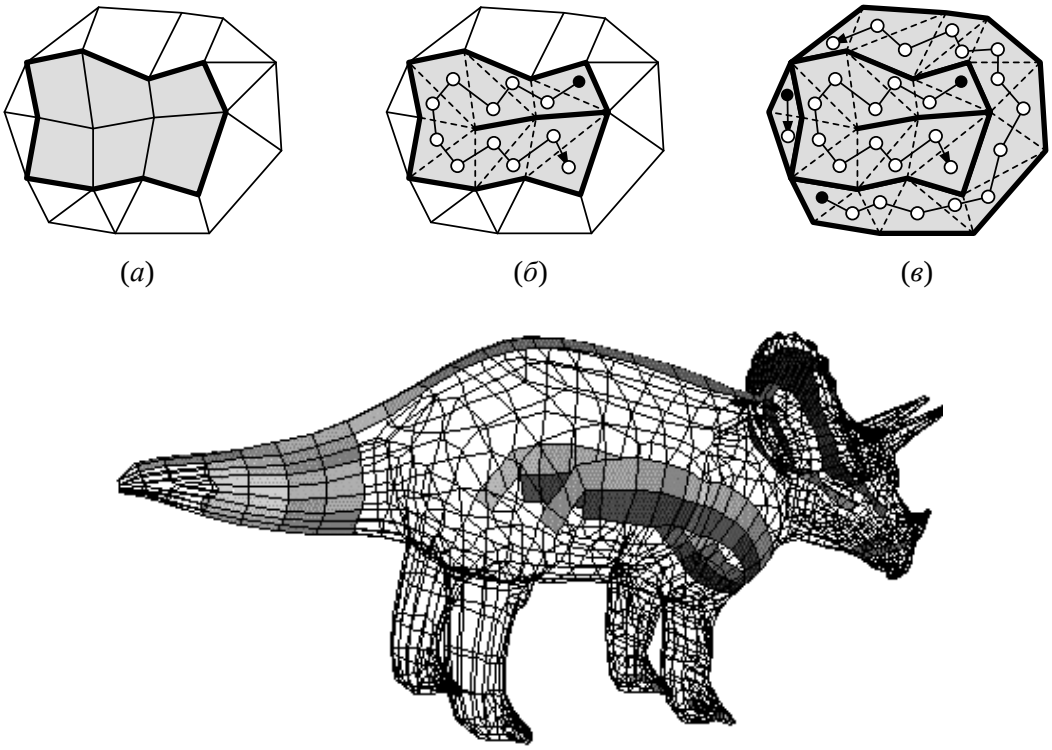


Рис. 93. Стрипификация четырёхугольных областей четырёхугольников:
а – выделение патчей – четырёхугольных областей четырёхугольников;
б – стрипификация патчей; *в* – стрипификация остальных полигонов;
г – пример стрипификации полигональной модели трицератопса [28]

Суть подхода заключается в том, что вначале необходимо выделить максимально большие *патчи* – четырёхугольные области смежных четырёхугольников (рис. 93,*а*), при этом область должна содержать не менее 5 четырёхугольников.

Затем каждую такую область необходимо разбить на одну полосу (рис. 93,*б*). Оставшиеся участки необходимо триангулировать любым предыдущим способом, основанным на SGI-алгоритме (рис. 93,*в*). На рис. 93,*г* приведен пример выделения четырёхугольных областей четырёхугольников для последующей стрипификации (пример взят из работы [28]).

Глава 12. Сверхбольшие триангуляции

12.1. Определение

Последние годы стремительно растут объёмы данных, по которым возникает необходимость построения триангуляционных моделей поверхностей. Так, типичной ситуацией является необходимость построения триангуляции по миллионам и даже миллиардам точек. К сожалению, практически все известные алгоритмы не способны обрабатывать такие объёмы, так как требуют, чтобы все структуры данных находились в оперативной памяти, имеющей ограниченный размер.

Сверхбольшими моделями данных в настоящее время называют такие модели, которые для своего полного представления требуют памяти больше, чем имеется в наличии оперативной памяти. То есть предполагается, что оперативная память, в отличие от внешней, ограничена.

Такого рода модели данных требуют особых алгоритмов, учитывающих ограничение по доступной оперативной памяти. Некоторые из таких алгоритмов будут рассмотрены в данной главе.

12.2. Построение сверхбольшой триангуляции Делоне

Одним из наиболее простых и логичных вариантов решения проблемы построения сверхбольших триангуляций является разбиение исходного множества точек на части и построение нескольких смежных триангуляций. Основной проблемой данного решения является отличие единой триангуляции (рис. 94,*а*) от совокупности триангуляций (рис. 94,*б*). Такая проблема называется проблемой *сшивки* триангуляций.

Для ликвидации дырок между смежными триангуляциями обычно в обе триангуляции добавляют некоторые дополнительные точки: либо точки пересечения рёбер триангуляции с разделяющей прямой (рис. 94,*в*), либо две точки по краям триангуляции (рис. 94,*г*).

Первый способ более предпочтителен, особенно при моделировании поверхностей, так как позволяет сохранить форму поверхности. Однако проблема здесь заключается в том, что для нахождения точек пересечения рёбер триангуляции с разделяющей прямой нужно иметь уже построенную триангуляцию!

Во втором способе с двумя дополнительными точками по краям триангуляции вдоль границы образуются длинные вытянутые треугольники (рис. 94,*г*), что совершенно неприемлемо при решении многих практических задач, например при моделировании поверхностей.

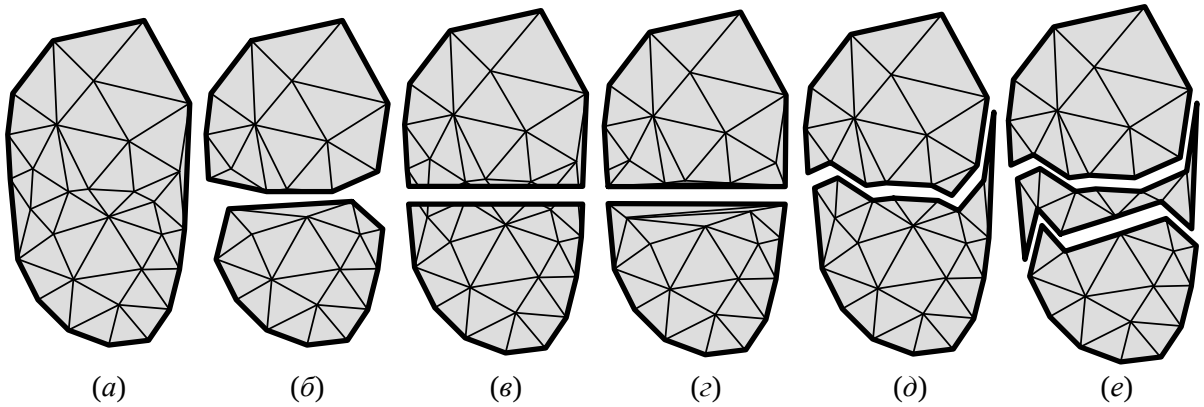


Рис. 94. Сшивка отдельно построенных смежных триангуляций: *а* – единая триангуляция; *б* – отдельные триангуляции без добавления дополнительных точек; *в* – отдельные триангуляции с дополнительными точками пересечения рёбер с разделяющей прямой; *г* – отдельные триангуляции с двумя дополнительными точками; *д* – отдельные невыпуклые триангуляции с дублирующимися основными точками; *е* – отдельные невыпуклые триангуляции со сшивающей триангуляцией

Более совершенный подход по сшивке триангуляций заключаются в построении двух невыпуклых триангуляций Делоне, разделённых некоторой цепью рёбер (рис. 94,*д*). Соответствующий алгоритм вначале делит исходные точки триангуляции на две части некоторой разделяющей прямой (рис. 95,*а*). На множестве точек первой части строится триангуляция Делоне, а потом из неё удаляются все треугольники, которые потенциально могли бы быть перестроены, если бы строилась единая триангуляция Делоне. Для удаления таких лишних треугольников используется свойство Делоне. Для этого надо по очереди для каждого крайнего снизу треугольника найти описанную окружность, и если она пересечёт разделительную линию (рис. 95,*б*), значит, внутрь этой окружности потенциально может

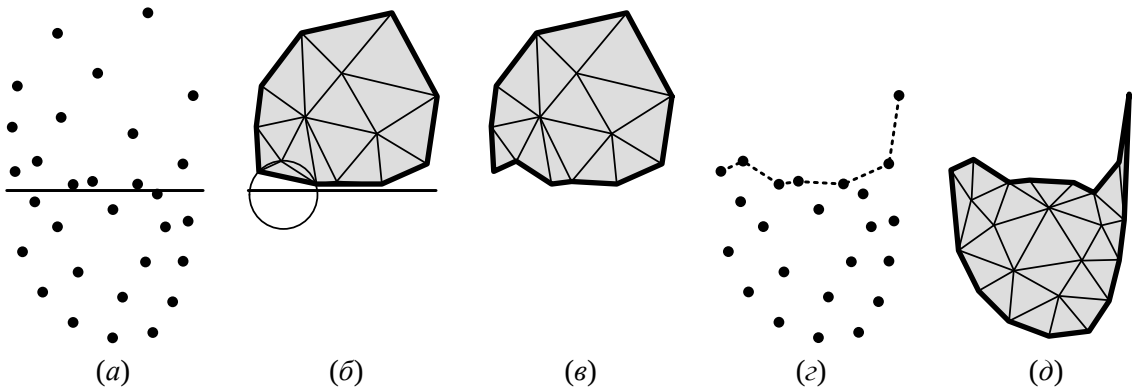


Рис. 95. Построение смежных невыпуклых триангуляций Делоне: *а* – разделение множества точек на две части; *б* – построение триангуляции Делоне по первой части точек и удаление лишних треугольников; *в* – построенная невыпуклая триангуляция Делоне; *г* – исходные данные для построения второй триангуляции Делоне (точки второй части и точки нижней границы первой триангуляции); *д* – построенная вторая триангуляция

попасть некоторая точка ниже разделительной линии, и потому данный треугольник будет невозможен в единой триангуляции Делоне.

В результате после удаления точек получится невыпуклая триангуляция (рис. 95,*в*). Тут следует отметить, что непосредственное удаление треугольников из первой триангуляции может привести к нарушению связности триангуляции, когда из-за удаления треугольников на краях могут остаться отдельные точки без треугольников (рис. 96,*а–б*). В крайнем случае могут быть удалены все треугольники (рис. 96,*в–г*). Поэтому на практике для предотвращения нарушения связности структуры триангуляции треугольники не удаляют, а только помечают как невидимые.

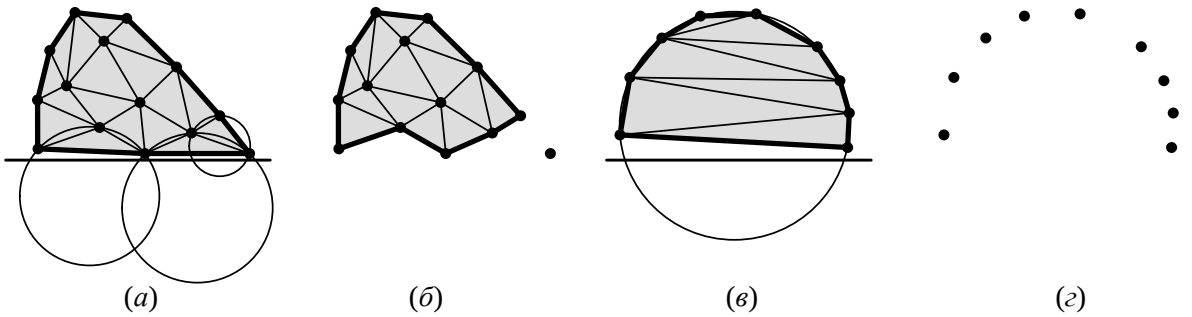


Рис. 96. Примеры нарушения связности триангуляции из-за удаления крайних треугольников: *а*,*в* – исходные триангуляции с разделяющими линиями; *б*,*г* – соответствующие триангуляции после удаления треугольников

После этого необходимо построить вторую триангуляцию на всех точках второй части и точках нижней границы из первой триангуляции (рис. 95,*г*). При этом вторая триангуляция должна получиться невыпуклой, для чего необходимо будет удалить из неё все треугольники, которые пересекаются с треугольниками из первой триангуляции (рис. 95,*д*).

Ещё один подход к сшивке триангуляций представлен на рис. 94,*е*. Вначале для каждой части точек строятся две независимые триангуляции Делоне, а затем из них удаляются треугольники, описанные окружности которых выходят за разделительную линию. После этого в области между этими триангуляциями строится ещё одна дополнительная, *сшивающая* триангуляция.

Итак, мы рассмотрели различные способы построения сверхбольшой триангуляции, разбивая задачу на две меньшие. В общем случае необходимо разбить множество точек на достаточно большое число частей. Рассмотрим один такой алгоритм, являющийся развитием последнего описанного подхода (рис. 94,*е*).

В клеточном алгоритме необходимо выполнить разбиение множества всех исходных точек на некоторые части (клетки), построить триангуляции в отдельных частях, а затем соединить отдельные триангуляции в единое целое. Основная идея алгоритма заключается в том, чтобы выбрать разме-

ры и количество частей так, чтобы количество точек в каждой части и количество треугольников в зоне слияния триангуляций было примерно равно друг другу и составляло $o(N)$. Это позволяет снизить размерность задачи и свести её к решению множества меньших задач. Рассмотрим алгоритм по шагам [14].

Клеточный алгоритм построения сверхбольшой триангуляции Делоне

Дано множество точек на плоскости в виде одного файла, причем занимаемый ими объём памяти превышает объёмы доступной оперативной памяти. Требуется построить триангуляцию Делоне.

Шаг 1. Разбиваем множество исходных точек на K частей с помощью клеточного разбиения. Для этого считываем из входного файла коор-

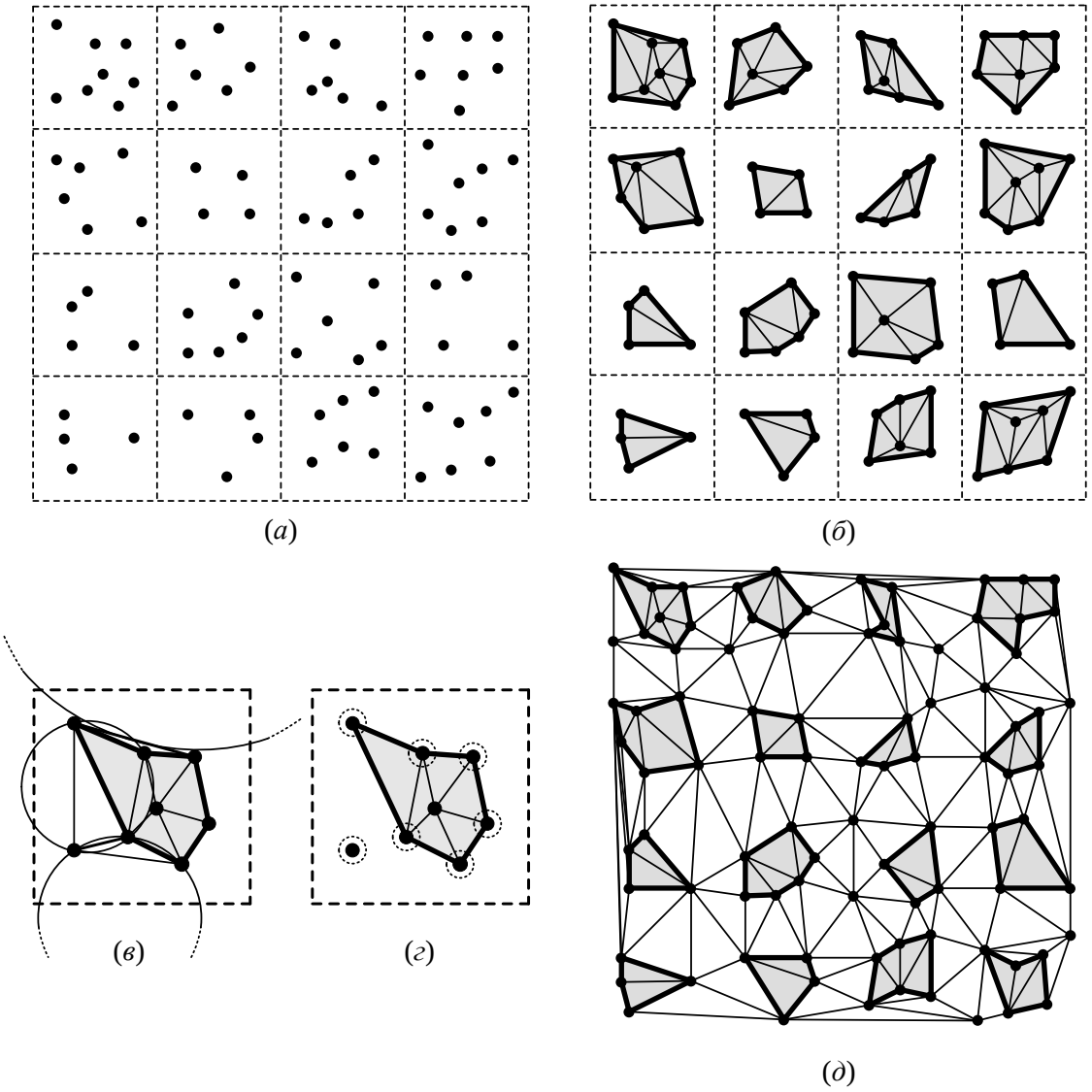


Рис. 97. Схема работы клеточного алгоритма построения сверхбольшой триангуляции Делоне: *а* – клеточное разбиение множества точек; *б* – построение триангуляций Делоне в клетках; *в* – определение неизменяемых треугольников Делоне; *г* – определение граничных точек; *д* – построение сшивающей триангуляции

динаты точек и записываем их в один из K файлов, соответствующих разным частям (рис. 97,*а*).

Шаг 2. Строим триангуляцию Делоне для множеств точек в каждом из K файлов (рис. 97,*б*). Если при этом количество точек в очередном файле слишком велико, то можно применить данный алгоритм рекурсивно, иначе – любой обычный алгоритм построения триангуляции Делоне.

Шаг 3. В каждой из K построенных триангуляций определяем, какие треугольники гарантированно не будут перестроены при объединении триангуляции, а какие могут быть потенциально перестроены. Для этого находим для каждого треугольника описанную окружность. Если она хотя бы частично выходит за границы текущей клетки, то условие Делоне в данном треугольнике может быть нарушено точками соседних клеток, а поэтому помечаем этот треугольник как перестраиваемый (рис. 97,*в*)

Шаг 4. В каждой триангуляции определяем точки, попадающие на границу триангуляции либо входящие в состав перестраиваемых треугольников, определённых на предыдущем шаге. Все перестраиваемые треугольники удаляем (рис. 97,*г*).

Шаг 5. Множества всех точек, найденных на предыдущем шаге, подаём на вход некоторого алгоритма построения сшивающей триангуляции Делоне (рис. 97,*д*). Если количество точек в этом множестве велико, то используем данный алгоритм рекурсивно, иначе применяем любой обычный алгоритм построения триангуляции Делоне.

Шаг 6. В полученной сшивающей триангуляции удаляем треугольники, которые пересекаются с треугольниками в отдельных триангуляциях. Для этого выполняем поочерёдное наложение на сшивающую триангуляцию всех отдельных триангуляций. *Конец алгоритма.*

Определим количество K клеток, исходя из минимизации используемой памяти. Наибольшие затраты памяти возникают тогда, когда мы строим триангуляции отдельных частей (шаг 2), сшивающую триангуляцию (шаг 5), а также когда выполняем оверлей двух триангуляций на шаге 6. Потребуем, чтобы размеры сшивающей триангуляции и триангуляций отдельных частей были равны.

Пусть N – общее количество исходных точек. Тогда, подобрав соответствующее разбиение, можно получить, чтобы в среднем в каждой клетке было $n = N / K$ точек.

Попробуем определить, какое количество точек M войдет в сшивающую триангуляцию. Для этого нужно найти число точек m, попадающих на границу триангуляции только одной из клеток (шаг 4 алгоритма). Тогда $M = mK$.

К сожалению, это число m и даже тип его функциональной зависимости очень сильно зависят от вида распределения исходных точек (см. п. 1.2). Для равномерного распределения в единичном квадрате $m = 5{,}8\sqrt{n}$.

Отсюда получаем $M = 5{,}8\sqrt{NK}$. Из условия равенства размера сшивающей триангуляции и триангуляций отдельных частей количество клеток разбиения $K \approx 0{,}31\sqrt[3]{N}$. При этом в среднем в каждой клетке и в сшивающей триангуляции окажется по $n = M = 3{,}23 \cdot N^{2/3}$ точек [14].

Трудоёмкость данного алгоритма в среднем является линейной.

В заключение отметим, что важным достоинством этого алгоритма является то, что он позволяет эффективно работать с большинством распространённых структур данных для представления триангуляций (см. п. 1.3). В результате работы алгоритма будут сгенерированы отдельные файлы, соответствующие триангуляциям в отдельных клетках, и один файл сшивающей триангуляции. При использовании двойной нумерации всех узлов и треугольников (номер клетки + номер узла или треугольника) можно достаточно легко работать со сгенерированной структурой данных. При этом по мере необходимости в оперативную память могут подгружаться те или иные части общей триангуляции.

12.3. Блочно-кластерная мультитриангуляция

В предыдущем разделе было описано, как можно построить сверхбольшую триангуляцию, разделив множество исходных точек на части и построив множество отдельных триангуляций. Однако использование такой сверхбольшой триангуляции на практике весьма затруднено из-за крайне большого числа треугольников.

В то же время в задачах анализа и визуализации триангуляции обычно задаётся область интересов (область на плоскости, в пределах которой требуется найти решение) и ограничения по точности требуемого решения. В п. 10.4 представлена *мультитриангуляция* – специальная структура данных, позволяющая очень эффективно представлять триангуляционные модели поверхностей. Важнейшим свойством мультитриангуляции является то, что она позволяет эффективно решать различные задачи на триангуляции в пределах заданной области интересов, а также учитывать ограничения по точности. Например, трудоёмкость построения изолиний по триангуляционной модели поверхности в заданном многоугольнике будет пропорциональна общему числу треугольников в этом многоугольнике, а не общему числу треугольников во всей триангуляции!

Если же необходимо построить изолинии определённой точности по всей модели поверхности, то с помощью мультитриангуляции можно быстро получить упрощённую (до некоторой степени) триангуляцию, а затем уже по ней построить изолинии. И в этом случае трудоёмкость будет также пропорциональна не общему числу треугольников в триангуляции, а размеру упрощенной триангуляции! В частности, в качестве критерия упрощения мы можем задать максимально допустимое число $N_{\max}$ тре-

угольников в упрощённой триангуляции, и тогда трудоёмкость станет пропорциональной $N_{\max}$.

Сама по себе мультитриангуляция разрабатывалась в первую очередь для целей быстрой визуализации моделей поверхностей на экране компьютера. Предполагалось, что имеющейся оперативной памяти достаточно много для представления всей структуры мультитриангуляции. В этом случае имеющиеся алгоритмы позволяют очень быстро извлекать из мультитриангуляции различные упрощённые триангуляции для последующей визуализации на экране компьютера.

Требование о полном представлении в памяти компьютера всей мультитриангуляции связано с высокой степенью *запутанности* графа мультитриангуляции (см. рис. 72), приводящей к тому, что при извлечении триангуляции часто приходится проходить по графу а) от самого верха до самого низа в случае извлечения триангуляции заданной точности либо б) достаточно широко (геометрически) в стороны при извлечении триангуляции в заданной области интересов. Пример запутанного графа мультитриангуляции приведён на рис. 98.

Для того чтобы мультитриангуляцию можно было использовать для представления сверхбольших моделей поверхностей, она должна иметь более простую структуру, которую можно было бы достаточно просто разделить на независимые части, соответствующие, во-первых, разным геометрически удалённым местам на поверхности и, во-вторых, фрагментам мультитриангуляции различной степени детализации. При таком разделении мультитриангуляции эти части можно было бы загружать в оперативную память по мере необходимости в зависимости от требуемого критерия детализации и наличия свободной оперативной памяти.

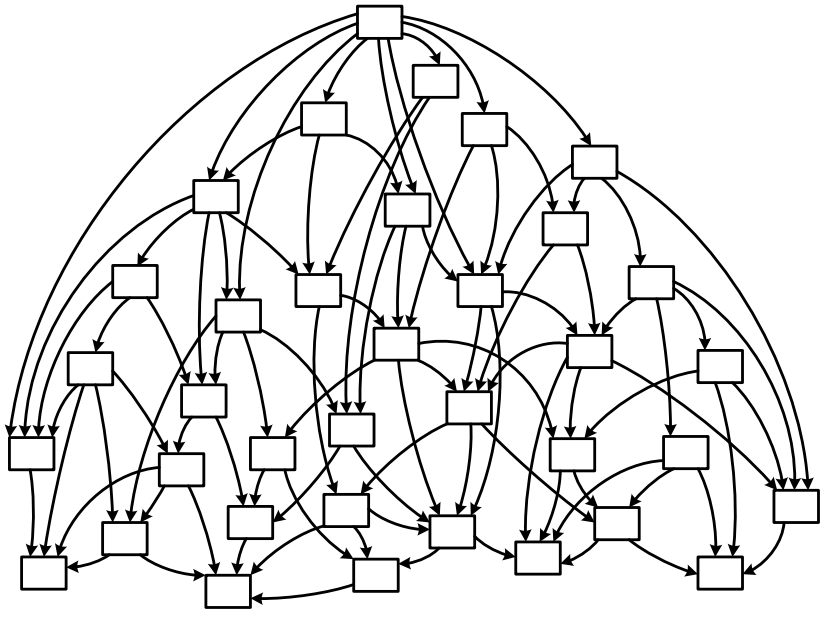


Рис. 98. Пример запутанного графа фрагментов мультитриангуляции

Для разделения графа мультитриангуляции по геометрически удаленным частям необходимо опираться на клеточный алгоритм, представленный выше в п. 12.2. По каждой триангуляции в клетках этого алгоритма следует построить отдельные мультитриангуляции, которые называют *кластерами*, а по сшивающей триангуляции строится одна объединяющая мультитриангуляция – *основная часть*. При этом кластеры не будут иметь связей (рёбер) между узлами разных кластеров. Кластеры будут иметь связи только внутри себя, а также с основной частью (рис. 99). Такая мультитриангуляция, разбитая на кластеры, называется *кластерной*.

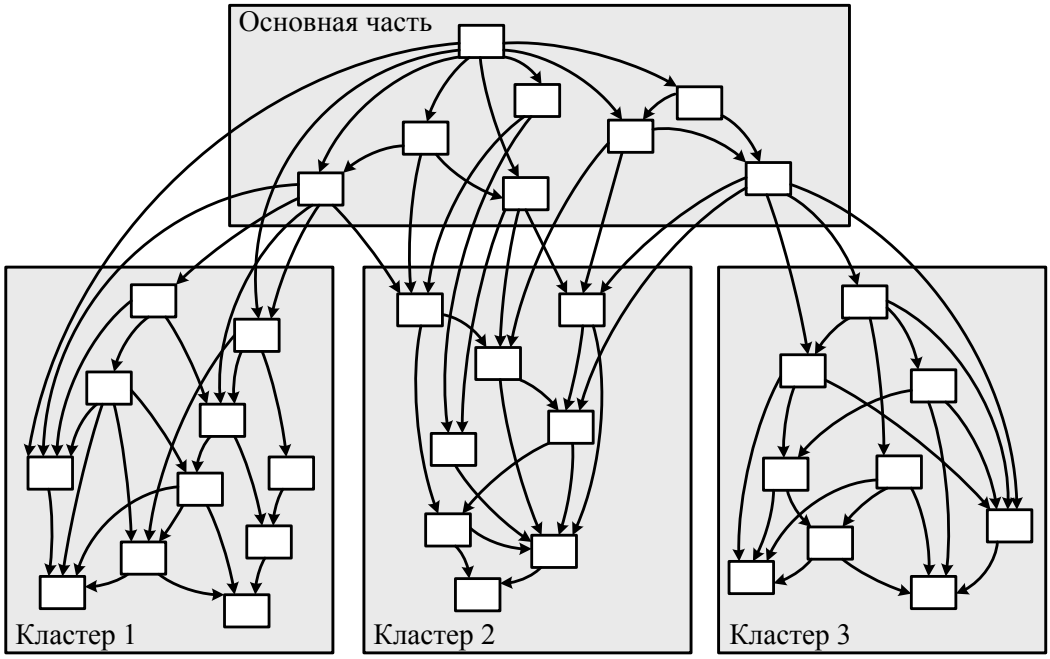


Рис. 99. Пример кластерной мультитриангуляции

Теперь рассмотрим вопрос организации графа мультитриангуляции внутри кластеров. Запутанная структура кластеров, когда имеются длинные нерегулярные связи сверху донизу графа, является следствием нерегулярного выбора детализирующих фрагментов мультитриангуляции (см. п. 10.4). Отсутствие связи между верхними и нижними уровнями графа мультитриангуляции может быть гарантировано только в случае, если фрагменты на промежуточных уровнях графа заменяют все треугольники, находящиеся выше по графу. При этом выбор фрагментов для мультитриангуляции нужно осуществлять так, чтобы минимальное число фрагментов заменяло сразу все треугольники. Иногда такие фрагменты выбрать сложно или даже невозможно (рис. 100,*а*), поэтому остаётся некоторое число незаменённых треугольников (рис. 100,*б*). Эти незаменённые треугольники необходимо добавить в ближайшие фрагменты (рис. 100,*в*), либо создать фрагменты, заменяющие треугольники на точно такие же (рис. 100,*г*).

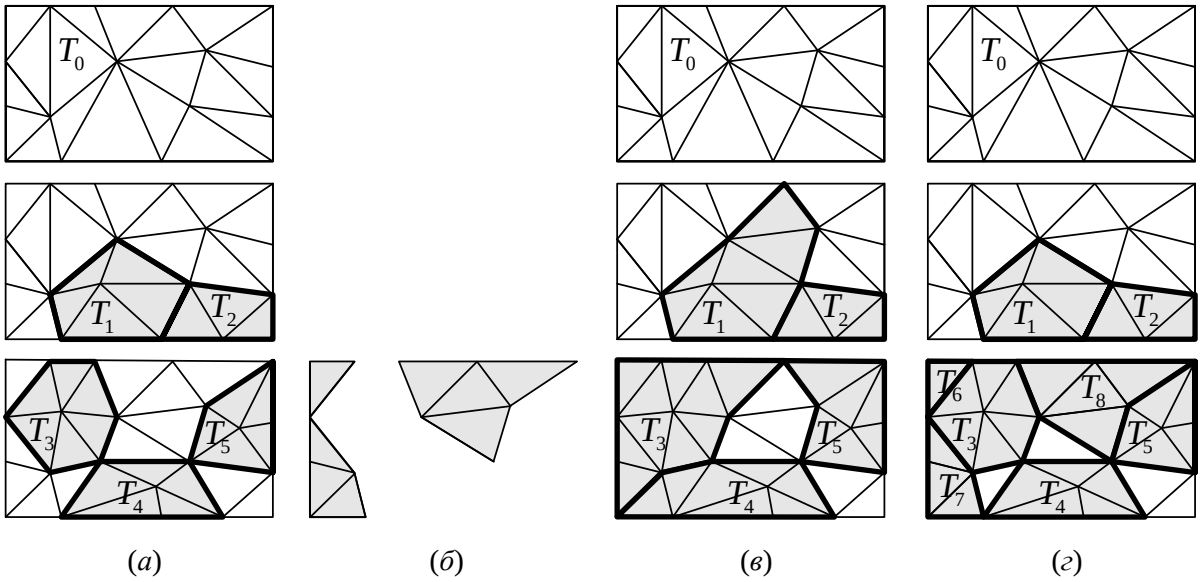


Рис. 100. Выбор фрагментов мультитриангуляции, заменяющих все треугольники: *а* – стандартный выбор фрагментов с незаменёнными треугольниками (как на рис. 71,*а*); *б* – незаменённые треугольники; *в* – фрагменты, дополненные ближайшими треугольниками; *г* – фрагменты с дополнительными фрагментами из незаменённых треугольников

Полный набор фрагментов, заменяющих все треугольники в триангуляции, называется *блоком*. Кластерная мультитриангуляция, в которой все кластеры дополнительно разбиты на блоки, называется *блочно-кластерной* (рис. 101).

Исключительно важным достоинством блоков мультитриангуляции является то, что граф мультитриангуляции может иметь связи только внутри блоков или между соседними блоками (а верхние блоки и с основной частью), так как каждый блок заменяет все треугольники. Именно это свойство позволяет эффективно расходовать оперативную память, загружая только необходимые блоки мультитриангуляции (основная часть должна всегда находиться в оперативной памяти целиком).

Алгоритм извлечения триангуляции из блочно-кластерной мультитриангуляции с учётом имеющихся ограничений на доступную оперативную память построен на стратегии поиска в ширину. Алгоритм поддерживает список неуточнённых фрагментов графа, упорядоченных по достигнутому критерию извлечения (например, по размеру треугольников на экране компьютера – в задаче визуализации мультитриангуляции). Вначале в этот список помещается только корень основной части. Этот же корень также объявляется в качестве текущей триангуляции. Пока список не пуст, из него извлекается и применяется к текущей триангуляции *применимый* (см. ниже) фрагмент T, уточняющий часть триангуляции с наихудшим значением критерия извлечения (например, часть, содержащую самые большие треугольники на экране компьютера). Затем определяются и добавляются в список новые фрагменты, уточняющие текущий фрагмент T.

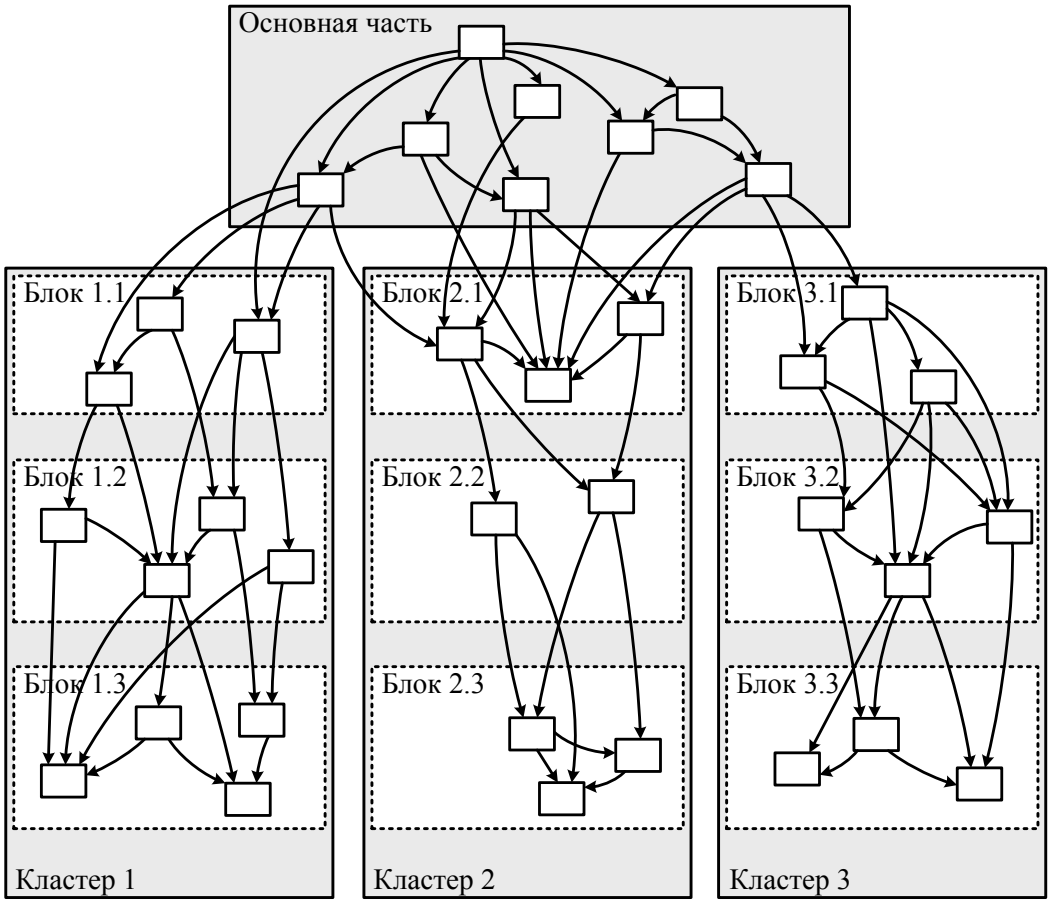


Рис. 101. Пример блочно-кластерной мультитриангуляции

Если новые фрагменты лежат в ещё не загруженных блоках, то делается попытка загрузить эти блоки из внешней памяти. Если доступная память уже вся израсходована, то до конца алгоритма извлечения больше не делается никаких попыток загрузки новых блоков. Соответственно, фрагменты, которые лежат в незагружаемых блоках, не помещаются в список неуточнённых фрагментов.

Затем для каждого нового неуточнённого фрагмента необходимо подняться вверх по графу по всем связям до уже применённых фрагментов, тем самым определяя дополнительные фрагменты, без применения которых нельзя использовать данные неуточнённые. То есть, обратите внимание, что некоторые фрагменты в списке неуточнённых нельзя применять, пока не будут применены некоторые другие фрагменты из этого списка. Это необходимо учитывать при извлечении из данного списка очередного фрагмента.

Алгоритм извлечения заканчивает работу в двух случаях: 1) либо закончился список неуточнённых фрагментов, 2) либо текущая полученная триангуляция достигла максимально допустимого размера, который может определяться количеством доступной памяти или производительностью видеокарты в случае вывода изображения на экран.

Глава 13. Анализ поверхностей

13.1. Построение разрезов поверхности

Одной из базовых задач анализа триангуляционных поверхностей является построение разрезов – вертикальных (профилей) и горизонтальных (изолиний).

В задаче построения профилей задается некоторая ломаная, вдоль которой необходимо построить разрез поверхности. Для этого нужно пройти вдоль этой ломаной с помощью варианта 1 алгоритма локализации треугольников в триангуляции (разд. 2.1, рис. 15,*а*), вычисляя последовательные трёхмерные точки пересечения ломаной с рёбрами триангуляции.

Задача построения изолиний, несмотря на внешнее сходство с профилями, значительное сложнее.

Определение 29. *Изолиниями* уровня h называют геометрическое место точек на поверхности, имеющих высоту h и имеющих в любой своей окрестности другие точки с меньшей высотой:

$$I_h = \{(x,y) \mid z(x,y) = h, \forall \varepsilon > 0 : \exists (x',y') : |(x',y'),(x,y)| < \varepsilon, z(x',y') < h\}.$$

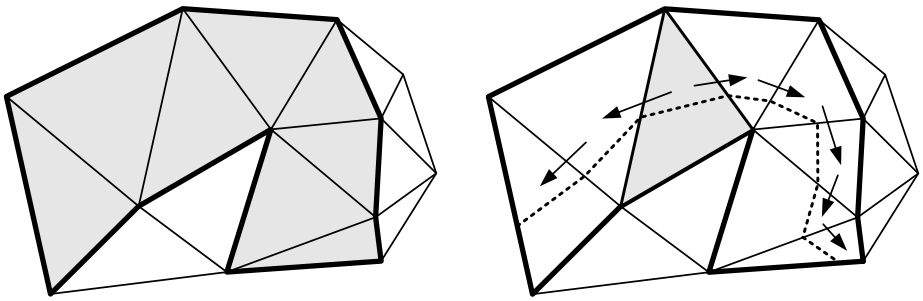
Условие наличия в любой окрестности точки с меньшей высотой позволяет избежать неопределённостей, когда в триангуляции имеются горизонтальные рёбра или даже треугольники (плато) с высотой h. В противном случае изолиния не будет представляться в виде линий.

Для построения изолиний высотой h можно применить следующий алгоритм.

Алгоритм построения изолиний.

Шаг 1. Помечаем каждый треугольник триангуляции, по которому проходят изолинии (т.е. выполняется условие $\min(z_1, z_2, z_3) < h < \max(z_1, z_2, z_3)$, где z_i – высоты трех его вершин), флагом $C_i := 1$, а все остальные треугольники – $C_i := 0$ (рис. 102,*а*). Если обнаружен хотя бы один треугольник, у которого хотя бы одно ребро лежит в плоскости изолинии, то h уменьшается на некоторое малое Δ и алгоритм повторяется заново.

Шаг 2. Для каждого треугольника с $C_i = 1$ выполняем отслеживание очередной изолинии в обе стороны от данного треугольника, пока один конец не выйдет на другой или на границу триангуляции (рис. 102,*б*). Каждый пройденный при отслеживании треугольник помечается $C_i := 0$. *Конец алгоритма.*



(*а*) (*б*)

Рис. 102. Построение изолиний: *а* – определение потенциальных треугольников; *б* – отслеживание изолиний

Трудоёмкость такого алгоритма, очевидно, является линейной относительно размера триангуляции.

Определение 30. Изоконтурами между уровнями h_1 и h_2 называют замыкание геометрического места точек на поверхности, имеющих высоту $h \in [h_1, h_2)$, т.е. множество точек $I_h = \overline{\{(x, y) \mid h_1 \le z(x, y) < h_2\}}$.

Определение 31. В *задаче построения изоконтуров* требуется построить множество непересекающихся регионов, каждый из которых представляет область, высоты точек внутри которой лежат в определённом диапазоне.

Обычно задаётся система диапазонов с помощью начального значения самого первого диапазона, конечного значения последнего диапазона и шага построения диапазонов.

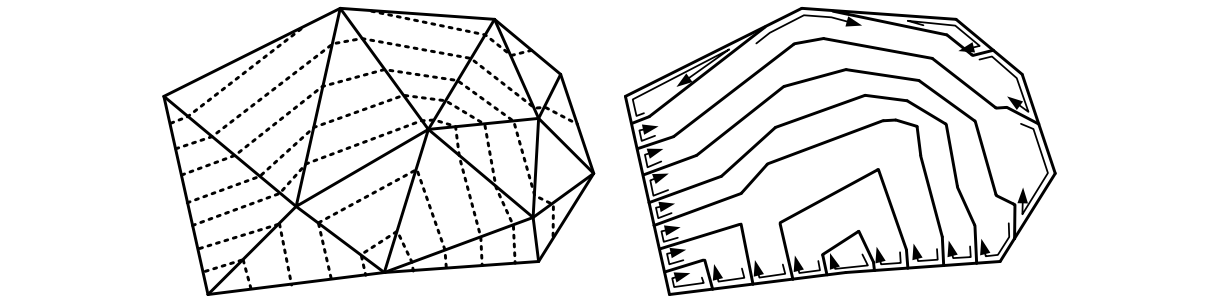
Алгоритм построения изоконтуров

Пусть заданы уровни $h_1, ..., h_M$. Необходимо построить изоконтуры.

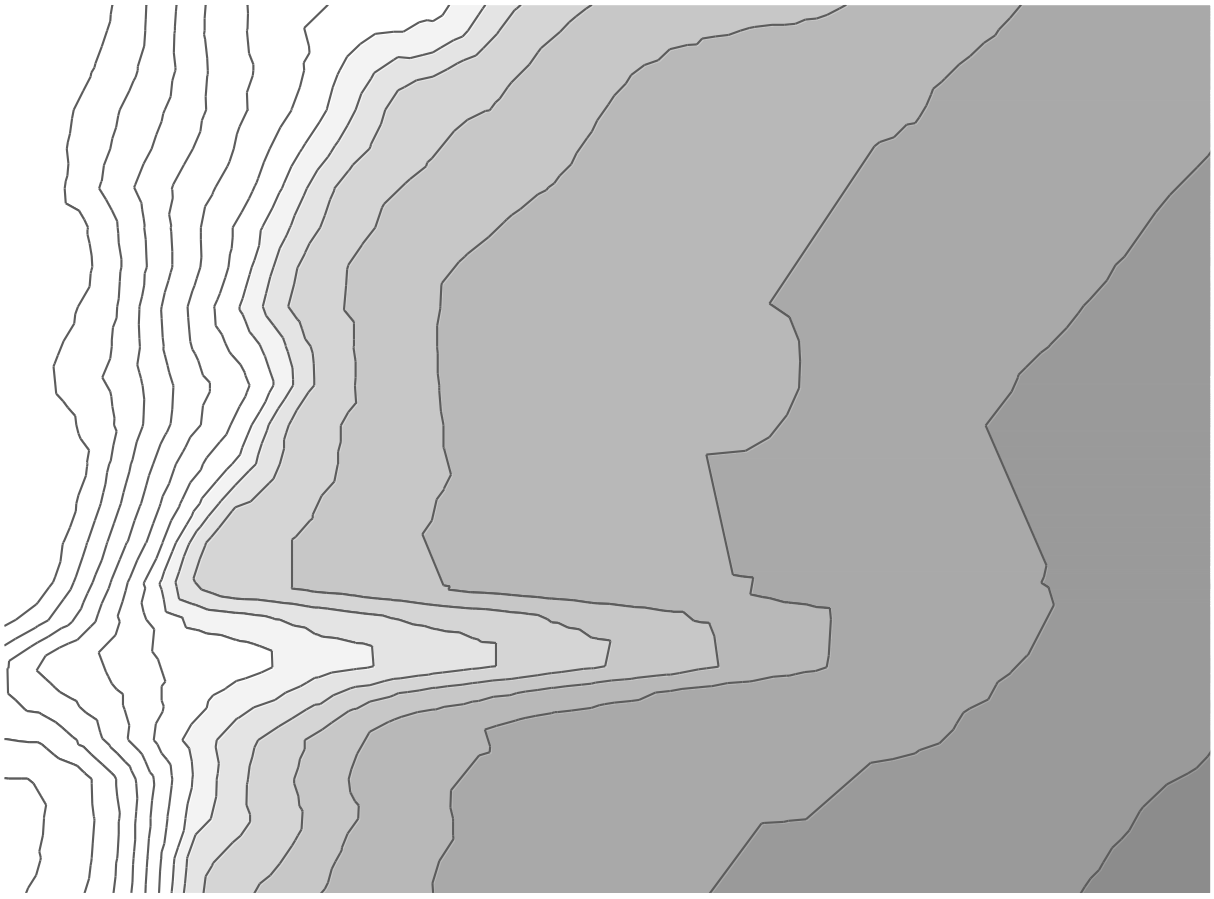
Шаг 1. Обнуляем множества ломаных, входящих в изоконтуры: $C_i = \varnothing$, $i = \overline{0, M}$.

Шаг 2. Для каждого уровня h_i строим изолинии (рис. 103,*а*). Каждую замкнутую изолинию добавляем во множество C_i.

Шаг 3. Определяем все кусочки границы триангуляции между точками выхода изолиний на границу. Формируем граф, в котором в качестве узлов выступают точки выхода на границу, а в качестве рёбер – кусочки границы между этими точками и рассчитанные изолинии. Каждая изолиния должна войти в граф дважды в виде одинаковых ориентированных рёбер, но направленных в разные стороны. Для рёбер – кусочков границы – устанавливаем такую ориентацию, чтобы внутренности (треугольники) триангуляции находились справа по ходу движения (рис. 103,*б*). В результате в каждом узле графа должны сходиться четыре ребра.



(*а*) (*б*)



(*в*)

Рис. 103. Построение изоконтуров: *а* – построение изолиний;
б – объединение изолиний и кусочков границы в изоконтуры;
в – пример расчета по модели рельефа, представленной на рис. 66

Шаг 4. По полученному графу строим контуры. Начинаем движение с любой вершины графа и двигаемся вперед в соответствии с ориентацией рёбер до тех пор, пока не вернёмся в начальную вершину. Повторный проход по одному и тому же ребру запрещён, для чего делаются специальные пометки на рёбрах. При попадании в узел графа из граничной цепочки далее надо двигаться по ребру, соответствующему изолинии, иначе – по граничному ребру. Обратим внимание, что каждая изолиния войдет в два контура, соответствующих разным диапазонам высот.

Шаг 5. Для каждого полученного на предыдущем шаге контура определяем, какому диапазону высот он соответствует. Для этого нужно взять и проверить любое ребро триангуляции, входящее в составе граничной цепочки, использованной в каждом контуре. На основании этого помещаем цепочку в соответствующее множество C_i (рис. 103,*в*). *Конец алгоритма.*

Трудоёмкость данного алгоритма линейно зависит от размера триангуляции и количества изолиний.

13.2. Сглаживание изолиний

Главными недостатками многих алгоритмов построения изолиний являются резкие изгибы и сильная осцилляция получаемых линий. Это связано с неравномерностью получаемых узловых точек изолиний и обычно используемым линейным методом интерполяции.

Попытки сгладить изолинии с помощью стандартных методов сглаживания кривых (полиномы, сплайны) не всегда приемлемы, т.е. при этом возможно пересечение изолиний разных уровней. Для устранения этого недостатка в [7] предложен специальный коридорный алгоритм. Суть его заключается в предварительном построении для всех изолиний неперекрывающихся коридоров и последующем построении в их пределах изолиний в виде ломаной минимальной длины или гладких кривых Безье.

Алгоритм построения гладких изолиний

Пусть необходимо построить изолинии уровней $h_1 < h_2 < ... < h_n$, сглаженных кривыми Безье.

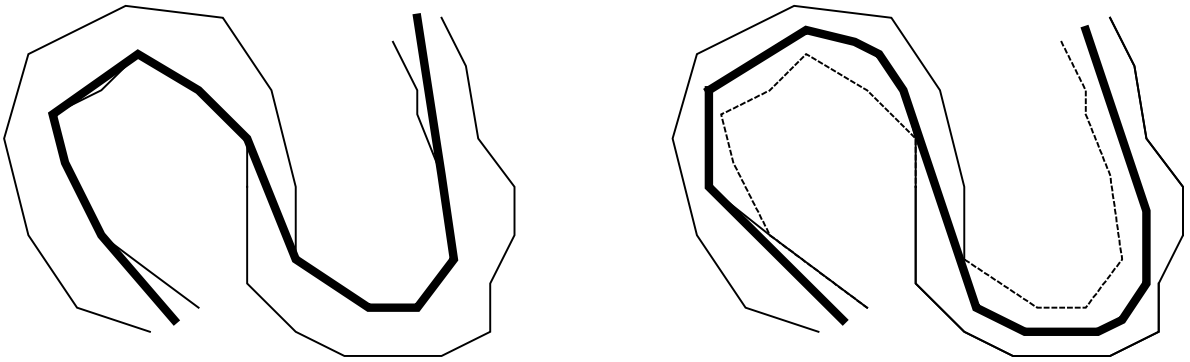
Шаг 1. Вычисляем максимально допустимое отклонение высот сглаженной изолинии от истинной $\Delta h = \max\limits_{i=1,\,n-1}(h_{i+1} - h_i)/2$.

Шаг 2. Вычисляем для каждой изолинии h_i коридор в виде обычных несглаженных изолиний $h_i - \Delta h$ и $h_i + \Delta h$.

Шаг 3. В полученных коридорах с помощью кривых Безье строится сглаживающая изолиния. *Конец алгоритма.*

Для сглаживания в [7] предлагаются три способа.

Самый простой заключается в построении ломаной минимальной длины, которая подобна резиновой нити с двумя закрепленными концами, растянутой внутри коридора (рис. 104,*а*). Однако при этом получается, что минимальная ломаная часто прижимается к одной из границ коридора, а количество осцилляций почти не уменьшается. Поэтому для этого нужно модифицировать коридор, сдвинув все вершины границ, совпадающие с узлами ломаной на участках выпуклостей, к центру коридора (рис. 104,*б*).



(*а*) (*б*)

Рис. 104. Построение ломаной минимальной длины:
а – в исходном коридоре; *б* – в модифицированном коридоре

Для получения действительно гладких изолиний можно использовать кубические кривые Безье, разместив их в вышепостроенных коридорах. При этом для каждого отрезка ломаной минимальной длины необходимо задать по две дополнительные точки, которые вместе с двумя вершинами отрезка определяют управляющие точки кривой Безье. Дополнительные точки должны задаваться так, чтобы наклон кривой в вершинах ломаной был непрерывным, а четырёхугольники, образованные управляющими точками (а поэтому и кривые Безье), лежали бы целиком внутри коридора.

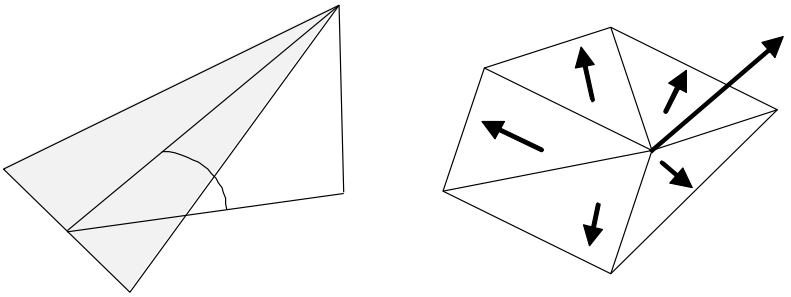
13.3. Построение изоклин

Решение задачи построения изолиний может быть положено в основу решения и других подобных задач. Практический интерес, например, вызывает *задача построения изоклин* – линий одинакового уклона поверхностей.

На практике широко распространены два способа выражения уклонов – с помощью градусной меры (угол) и процентов. Например, двухметровый подъём поверхности на 100 метров дистанции может быть выражен как 2% либо $1{,}15^\circ = \operatorname{arctg} 2/100$. При этом довольно просто перейти от процентного уклона к градусному и наоборот.

Рассмотрим теперь вопрос определения уклона поверхности. Существуют два способа расчёта уклонов. При первом из них (рис. 105,*а*) значения уклонов определяют локально для каждого треугольника как угол наклона пространственного треугольника к плоскости *XY*. Для этого вычисляется векторное произведение векторов двух сторон треугольника и получается вектор нормали к нему, который уже сравнивается с вертикалью.

На практике используется данный способ, однако чаще встречается подход, заключающийся в переходе от значений уклонов для треугольников к уклонам в их вершинах (рис. 105,*б*).



(*а*) (*б*)

(*в*)

Рис. 105. Расчёт уклонов: *а* – треугольников; *б* – узлов; *в* – пример расчёта по модели рельефа, представленной на рис. 66

Такой переход, с одной стороны, учитывает не только локальное положение каждого треугольника, но и их совместное расположение, а с другой – позволяет использовать алгоритм построения изолиний для расчёта изоклин. Действительно, путем замены координат *z* точек на значения рассчитанных уклонов получаем вторичное поле значений для данной поверхности, которое можно подать на вход алгоритма построения изолиний. Этот алгоритм построит линии, вдоль которых *z*-значения будут постоянными, а такие линии и будут являться изоклинами.

Аналогично расчёту изоклин можно построить и полосовые контуры между изоклинами, используя алгоритм построения изоконтуров из предыдущего раздела (рис. 105,*в*).

В заключение отметим, что расчёт контуров между изоклинами можно выполнить и другим способом. Для этого нужно просто определить уклон каждого треугольника и тем самым указать, в какой регион он должен быть включен. После чего нужно просто вызвать алгоритм выделения регионов. Отметим, что такой вариант проще в реализации, но выдаёт визуально хуже воспринимаемые изображения.

13.4. Построение экспозиций склонов

Еще одной часто возникающей в геоинформатике задачей является *построение экспозиций склонов*. Здесь требуется определить доминирующие направления склонов по странам света и разбить поверхность на регионы, в которых доминирует некоторое определённое направление. Так как для горизонтальных участков поверхности определение экспозиции не имеет смысла, то в отдельный регион выделяют области, являющиеся горизонтальными или имеющие незначительный уклон, например $\alpha < 5^\circ$. По странам света деление обычно выполняется на 4, 8 или 16 частей.

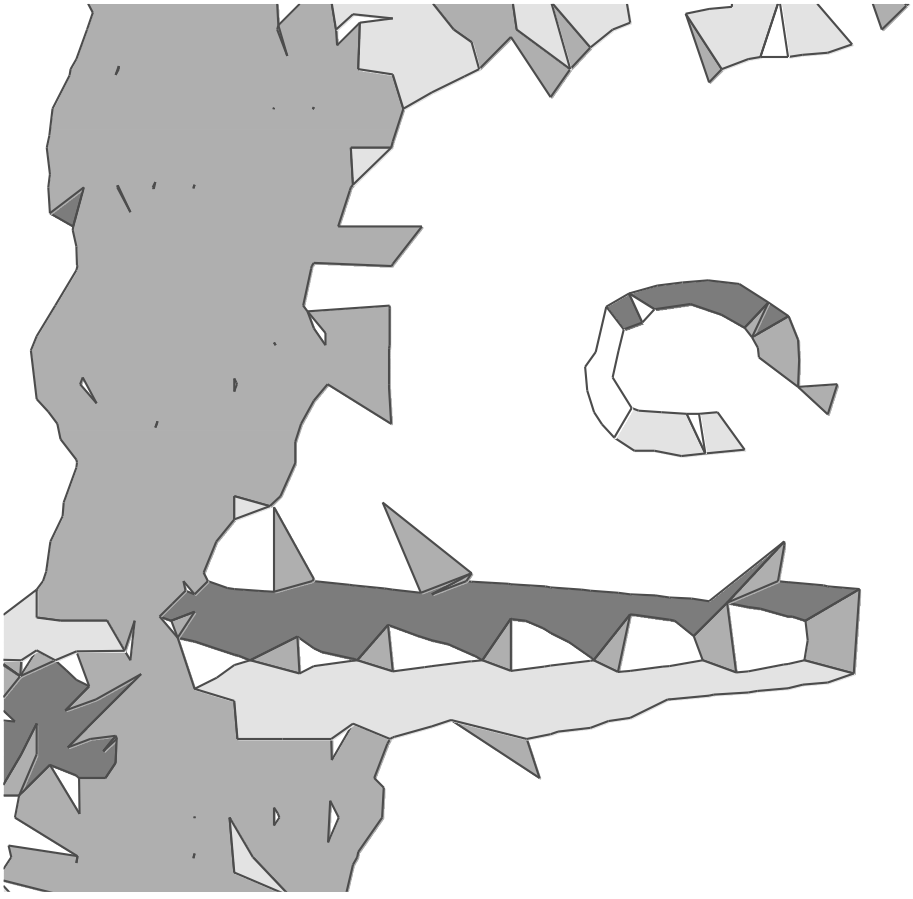


Рис. 106. Пример расчёта экспозиций склонов по модели рельефа, представленной на рис. 66

Задача расчёта экспозиций склонов обычно используется для анализа освещённости Земли. В связи с этим часто возникает потребность дополнительного учёта текущего положения Солнца, т.е. экспозиция вычисляется как направление между нормалью к треугольнику и направлением на Солнце.

Таким образом, каждый треугольник триангуляции может быть проклассифицирован по принципу принадлежности к тому или иному региону. После этого нужно просто вызвать алгоритм выделения регионов.

Пример выполнения расчёта экспозиций склонов по триангуляционной модели рельефа приведён на рис. 106. Белым представлены области с наклоном, а четырьмя темными – преимущественные направления уклона.

13.5. Вычисление объёмов земляных работ

В задаче расчёта объёмов земляных работ в дополнение к существующей модели рельефа задаётся желаемая модель. Требуется рассчитать, какую территорию нужно срезать, а какую засыпать, чтобы получить желаемую поверхность. При этом нужно определить объёмы перемещаемых масс грунта (сумма срезанного и засыпанного объёмов) и балансовый объём (разница срезанного и засыпанного объёмов, т.е. избыток или недостаток грунта).

В наиболее простой постановке желаемая форма рельефа задаётся как некоторый регион на карте, в пределах которого требуется выравнивание поверхности под заданный горизонтальный уровень. Это применяется для оценки объёмов работ при рытье котлованов с вертикальными стенками.

Решение задачи с котлованом выполняется следующим образом.

Алгоритм расчёта земляных работ при рытье котлована с вертикальными стенками и горизонтальным дном.

Шаг 1. Делается вырезка из общей триангуляции некоторой части по границе котлована. По сути, вначале делается копия исходной триангуляции и в неё вставляется граница котлована в качестве области интересов триангуляции, тем самым все треугольники вне котлована отбрасываются.

Шаг 2. Вызывается алгоритм построения изоконтуров для вырезанной триангуляции на требуемом уровне дна котлована. Алгоритм возвратит два региона, определяющих территории с излишком и с недостатком грунта соответственно.

Шаг 3. Для каждого треугольника вырезанной триангуляции выполняется сравнение с требуемым уровнем дна котлована. Если треугольник находится целиком выше дна, то его требуется засыпать, если целиком ниже – то срезать. Объём засыпки/срезки определяется как объём соответ-

ствующей треугольной призмы с двумя основаниями, одно из которых является текущим треугольником, а второе – проекцией этого треугольника на дно котлована. Если треугольник пересекается с плоскостью дна котлована, то делается сечение треугольника на две части, для которых отдельно вычисляются объёмы соответствующих призм. *Конец алгоритма.*

В более сложной постановке задачи расчёта земляных работ требуемая поверхность задается как другая независимая триангуляционная модель. В такой форме задача возникает при вертикальной планировке территорий самого разного назначения.

Обычно эта задача решается на регулярных моделях с предварительным преобразованием исходных триангуляционных моделей. В явном же виде на триангуляции обычно эту задачу не решают, так как существующие для этого алгоритмы весьма сложны и могут генерировать в худшем случае очень сложные регионы, имеющие число точек, пропорциональное квадрату общего числа узлов в исходных триангуляциях. Тем не менее в некоторых случаях возникает потребность в таких явных вычислениях, и поэтому можно использовать следующий достаточно простой алгоритм.

Определение 32. Пусть дана исходная модель рельефа в виде триангуляции T_1 и желаемая модель T_2. В *задаче расчёта земляных работ* требуется вычислить регион L, определяющий территорию, в пределах которого поверхность T_1 выше T_2, регион H, на котором T_1 ниже T_2, и регион E, на котором уровни T_1 и T_2 равны. Также требуется вычислить объём земли, который надо срезать, и объём, который надо насыпать.

Алгоритм расчёта земляных работ.

Шаг 1. Определяется минимальный многоугольник, охватывающий триангуляции T_1 и T_2 как пересечение охватываемых триангуляциями территорий.

Шаг 2. Создается новая триангуляция T, и в неё вносятся в качестве структурных рёбер все рёбра триангуляций T_1 и T_2. Для каждого узла n_i триангуляции T нужно вычислить высоты z_i^1 и z_i^2, определяющие высоты этого узла в триангуляциях T_1 и T_2 соответственно.

Шаг 3. Для каждого треугольника t_j новой триангуляции определяем, не пересекаются ли триангуляции T_1 и T_2 в пределах треугольника t_j:

1. Если $\forall k = \overline{1,3}: z_{jk}^1 = z_{jk}^2$, то оба треугольника лежат в одной плоскости, и поэтому треугольник t_j попадает в регион E.

2. Если $\forall k,l = \overline{1,3}: z_{jk}^1 \le z_{jl}^2$, то в пределах этого треугольника поверхность T_1 не выше T_2, и поэтому треугольник t_j попадает в регион H.

3. Если $\forall k,l=\overline{1,3}: z_{jk}^1 \geq z_{jl}^2$, то в пределах этого треугольника поверхность T_1 не ниже T_2, и поэтому треугольник t_j попадает в регион *L*.

4. Иначе, если $\exists\ k,l=\overline{1,3}: z_{jk}^1 > z_{jl}^2$ и $\exists\ m,n=\overline{1,3}: z_{jm}^1 < z_{jn}^2$, то поверхности пересекаются в пределах данного треугольника (рис. 107). Поэтому мы должны найти пересечение двух пространственных треугольников в виде некоторого отрезка, разделяющего треугольник на две части, которые в дальнейшем войдут в разные результирующие регионы *L* и *H*. После деления треугольника удобно рассчитать объёмы земляных работ в пределах данного треугольника.

Шаг 4. Собираем из всех найденных частей регионы *L*, *H* и *E*. *Конец алгоритма*.

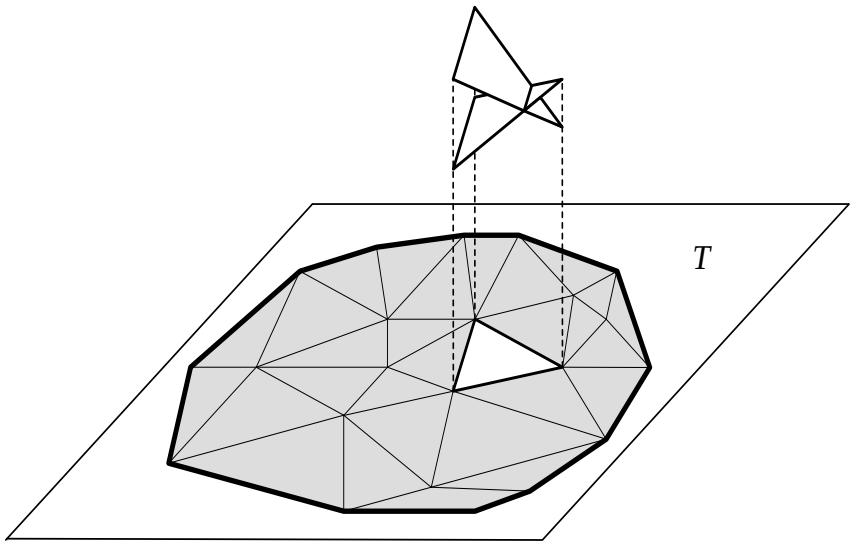


Рис. 107. Пересечение пространственных треугольников

Сложность данного алгоритма зависит в основном от трудоёмкости построения триангуляции *T*. В худшем случае это может быть $O((N_1+N_2)^2)$, где N_i – число узлов исходных триангуляций. Тем не менее в среднем эта величина может составлять $O(N_1+N_2)$ на равномерных распределениях исходных узлов триангуляций.

13.6. Построение зон и линий видимости

В задаче построения зон видимости по заданному положению наблюдателя в пространстве требуется определить, какие участки поверхности ему видны, а какие нет. Эта задача возникает, например, при размещении пожарных вышек, радарных станций, теле- и радиовышек, станций сотовой связи [61].

В ряде случаев эту задачу можно решать приближённо, например, переходя к растровому представлению, однако часто требуются более точные результаты расчётов.

Для решения данной задачи можно использовать общие методы удаления невидимых линий, применяемые в машинной графике, например алгоритмы z-буфера и плавающего горизонта.

Данную задачу иногда решают в упрощённом варианте, строя только *линии видимости*, которые представляют собой лучи, исходящие из точки видимости в разные стороны и разбитые на части по принципу видимости. Данная задача решается значительно проще, чем полный случай. Для этого вначале строится профиль поверхности вдоль этого луча, а потом методом плавающего горизонта формируются его видимые и невидимые части.

Для решения полной задачи можно использовать точный алгоритм, основанный на идее алгоритма плавающего горизонта [13]. В отличие от обычного алгоритма, используемого в машинной графике, в нашем случае горизонт будет представляться не в виде растра, а в виде ломаной кругового обзора. На рис. 108 по горизонтали откладывается азимут направления зрения, а по вертикали – максимальный текущий вертикальный горизонт зрения.

Алгоритм построения зон видимости.

Шаг 1. Устанавливаем текущий круговой плавающий горизонт в виде горизонтальной линии на уровне –90°.

Шаг 2. Последовательно анализируем все треугольники триангуляции от ближайших к точке зрения до самых удалённых. Каждый треугольник сравниваем с текущим горизонтом и выделяем те части треугольника, которые видны и не видны, и затем модифицируем текущий горизонт этим треугольником. *Конец алгоритма.*

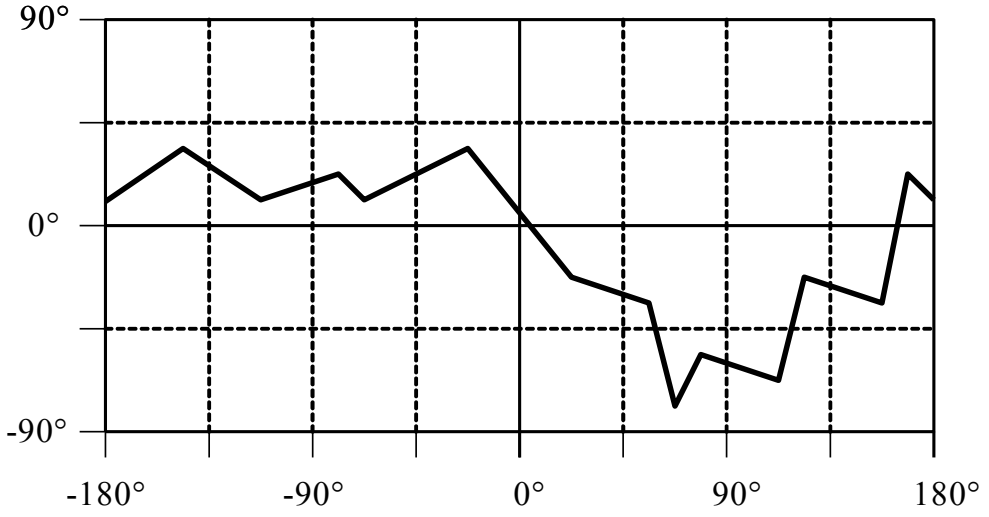


Рис. 108. Текущий круговой плавающий горизонт

В данном алгоритме самое сложное заключается в поиске такого правильного порядка обхода треугольников, чтобы все ранее анализируемые треугольники не заслонялись более поздними. К сожалению, такой порядок не всегда существует, хотя в практических задачах исключения возникают редко. В частности, в [30] показано, что для триангуляции Делоне такая ситуация не возникает никогда. На рис. 109 точкой обозначено положение наблюдателя, сплошными линиями – уже проанализированные треугольники. Анализ же пунктирных треугольников невозможен, так как перед каждым из них находится какой-то другой, закрывающий обзор. В таком сложном случае можно либо разрезать некоторый треугольник на части, либо просто продолжить анализ с наименее перекрываемого или самого близкого к наблюдателю треугольника. Второй вариант, возможно, даст ошибку в вычислении, но он значительно более прост.

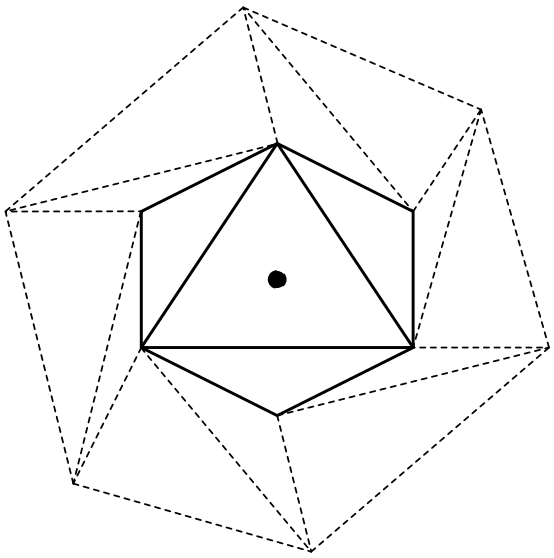


Рис. 109. Проблема правильного выбора порядка обхода треугольников

Для определения порядка обхода создаем список незаслоняемых треугольников L. Пока список не пуст, последовательно анализируем все его треугольники. После выполнения анализа очередного треугольника возможна ситуация, когда смежные с ним треугольники станут также незаслоняемыми, и тогда их надо также поместить в список L. Если список L станет пустым, то при наличии в триангуляции еще не проанализированных треугольников выбираем из них тот, центр которого находится ближе всех к точке зрения, и повторяем цикл анализа списка.

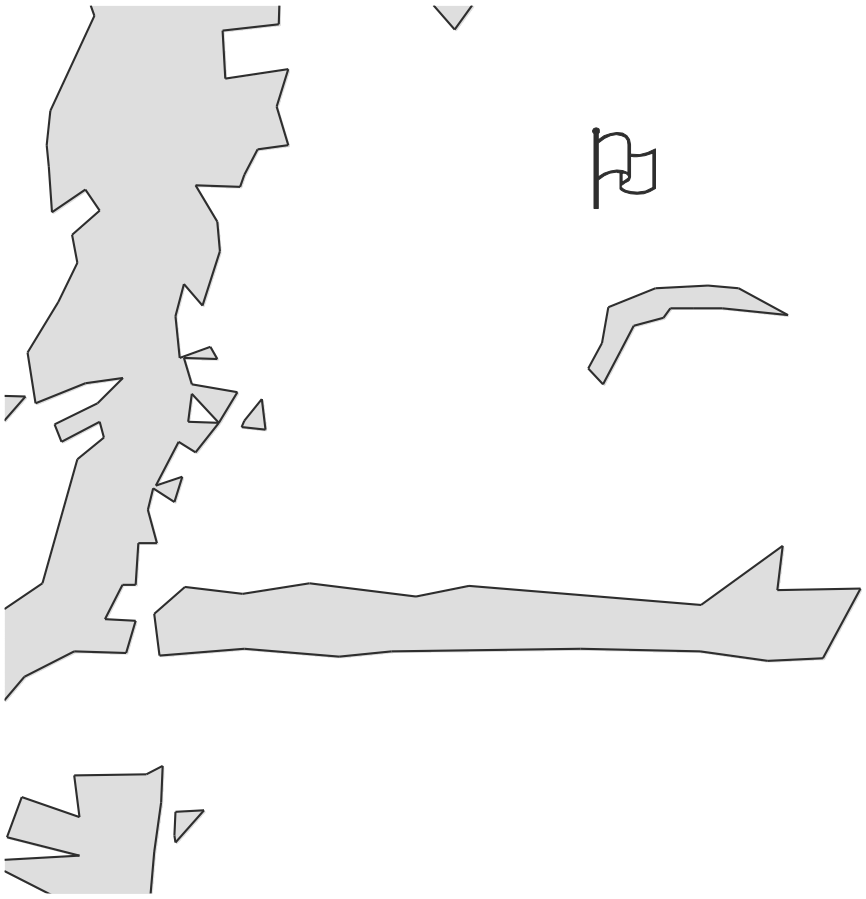


Рис. 110. Пример расчёта зон видимости по модели рельефа, представленной на рис. 66 (флажком отмечена точка наблюдения на высоте 5 метров от уровня земли, сплошной закраской изображены невидимые зоны)

Самым большим недостатком описанного алгоритма является чрезмерное количество узлов ломаных, делящих треугольники в зонах видимости, что не всегда нужно на практике. В худшем случае их количество может составлять $O(N)$. Поэтому для каждого треугольника разрезающую ломаную нужно упростить, оставить один или два отрезка.

На рис. 110 представлен пример выполнения расчёта зон видимости.

Литература

1. Делоне Б.Н. О пустой сфере // Изв. АН СССР. ОМЕН. 1934. № 4. С. 793–800.
2. Жихарев С.А., Скворцов А.В. Моделирование рельефа в системе ГрафИн // Геоинформатика: Теория и практика. Вып. 1. Томск: Изд-во Том. ун-та, 1998. С. 194–205.
3. Ильман В.М. Алгоритмы триангуляции плоских областей по нерегулярным сетям точек // Алгоритмы и программы. ВИЭМС. Вып. 10 (88). М., 1985. С. 3–35.
4. Ильман В.М. Экстремальные свойства триангуляции Делоне // Алгоритмы и программы. ВИЭМС. Вып. 10(88). М., 1985. С. 57–66.
5. Костюк Ю.Л., Грибель В.А. Размещение и отображение на карте точечных объектов // Методы и средства обработки сложной графической информации: Тезисы докладов Всесоюзной конференции. Ч. 2. Горький, 1988. С. 60–61.
6. Костюк Ю.Л., Фукс А.Л. Визуально гладкая аппроксимация однозначной поверхности, заданной нерегулярным набором точек // Геоинформатика-2000: Труды Международной научно-практической конференции. Томск: Изд-во Том. ун-та, 2000. С. 41–45.
7. Костюк Ю.Л., Фукс А.Л. Гладкая аппроксимация изолиний однозначной поверхности, заданной нерегулярным набором точек // Геоинформатика-2000: Труды Международной научно-практической конференции. Томск: Изд-во Том. ун-та, 2000. С. 37–41.
8. Костюк Ю.Л., Фукс А.Л. Приближенное вычисление оптимальной триангуляции // Геоинформатика. Теория и практика. Вып. 1. Томск: Изд-во Том. ун-та, 1998. С. 61–66.
9. Кошкарёв А.В., Тикунов В.С. Геоинформатика. М.: Картгеоиздат-Геодезиздат, 1993. 213 с.
10. Кроновер Р.М. Фракталы и хаос в динамических системах. Основы теории / Пер. с англ. М.: Постмаркет, 2000. 352 с.
11. Ласло М. Вычислительная геометрия и компьютерная графика на С++ / Пер. с англ. М.: БИНОМ, 1997. 304 с.
12. Препарата Ф., Шеймос М. Вычислительная геометрия: Введение / Пер. с англ. М.: Мир, 1989. 478 с.
13. Роджерс Д., Адамс Дж. Математические основы машинной графики / Пер. с англ. М.: Машиностроение, 1980. 204 с.
14. Скворцов А.В. Построение сверхбольшой триангуляции Делоне // Изв. вузов. Физика. 2002. №6. С. 22–25.
15. Скворцов А.В. Триангуляция Делоне и её применение. – Томск: Изд-во Том. ун-та, 2002. 128 с.
16. Скворцов А.В., Костюк Ю.Л. Применение триангуляции для решения задач вычислительной геометрии // Геоинформатика: Теория и практика. Вып. 1. Томск: Изд-во Том. ун-та, 1998. С. 127–138.

17. Скворцов А.В., Костюк Ю.Л. Эффективные алгоритмы построения триангуляции Делоне // Геоинформатика. Теория и практика. Вып. 1. Томск: Изд-во Том. ун-та, 1998. С. 22–47.
18. Фукс А.Л. Изображение изолиний и разрезов поверхности, заданной нерегулярной системой отсчётов // Программирование. 1986. № 4. С. 87–91.
19. Фукс А.Л. Предварительная обработка набора точек при построении триангуляции Делоне // Геоинформатика. Теория и практика. Вып. 1. Томск: Изд-во Том. ун-та, 1998. С. 48–60.
20. Agarwal P.K., Suri S. Surface approximation and geometric partitions // Proc. 5th ACM-SIAM Symp. on Discrete Algorithms. 1994. P. 24–33.
21. Akeley K., Haeberli P., Burns D. The tomesh.c program / Technical Report SGI Developer's Toolbox CD. Silicon Graphics. 1990.
22. Arkin E., Held M., Mitchell J., Skiena S. Hamiltonian triangulations for fast rendering // Second Annual European Symposium on Algorithms. Vol. 855. Springer-Verlag Lecture Notes in Computer Science, 1994. P. 36–47.
23. Barber C.B., Dobkin D.P., Huhdanpaa H. The Quickhull Algorithm for Convex Hulls // ACM Transactions on Mathematical Software. 1994. Vol. 22, № 4. P. 469–483.
24. Bern M., Eppstein D., Yao F. The expected extemes in a Delaunay triangulation // Inter. J. of Comp. Geom. and Appl. 1991. Vol. 1, № 1. P. 79–91.
25. Bjørke J.T. Quadtrees and triangulation in digital elevation models // International Archives of Photogrammetry and Remote Sensing, 16th Intern. Congress of ISPRS, Commission IV. Part B4. 1988. Vol. 27. P. 38–44.
26. Chang R.C., Lee R.C.T. On the average length of Delaunay triangulations // BIT. 1984. Vol. 24. № 3. P. 269–273.
27. Edelsbrunner H., Seidel R. Voronoi diagrams and arrangements // Discrete and Computational Geometry. 1986. Vol. 8, № 1. P. 25–44.
28. Evans F., Skiena S., Varshney A. Optimizing triangle strips for fast rendering // Proc. IEEE Visualization. 1996. P. 319–326.
29. De Floriani L. A pyramidal data structure for triangle-based surface description // IEEE Comp. Graphics and Applications. 1989. Vol. 9, № 2. P. 67–78.
30. De Floriani L., Falcidieno B., Nagy G., Pienovi C. On sorting triangles in a Delaynay tessellation // Algorithmica. 1991. № 6. P. 522–535.
31. De Floriani L., Magillo P., Puppo E. Compressing Triangulated Irregular Networks // Geoinformatica. 2000. Vol. 1, № 4. P. 67–88.
32. De Floriani L., Marzano P., Puppo E. Multiresolution Models for Topographic Surface Description // The Visual Computer. 1996. Vol. 12, № 7. P. 317–345.
33. De Floriani L., Magillo P., Puppo E., Bertolotto M. Variable resolution operators on a multiresolution terrain model // ACM 4th Workshop on Advances in Geographic Information Systems. 1996. P. 123–130.
34. Dillencourt M.B. Finding hamiltonian cycles in Delaunay triangulations is NP-complete // Canadian Conf. on Comput. Geometry. 1992. P. 223–228.

35. Evans F., Skiena S.S., Varshney A. Completing sequential triangulations is hard / Technical Report, Department of Computer Science, State University of New York at Stony Brook, March 1996.
36. Fowler R.J., Little J.J. Automatic extraction of irregular network digital terrain models // Computer Graphics. 1979. Vol. 13, № 3. P. 199–207.
37. Gilbert P.N. New results on planar triangulations. Tech. Rep. ACT-15, Coord. Sci. Lab., University of Illinois at Urbana, July 1979.
38. Graphics Library Programming Guide. Silicon Graphics, Inc. 1991.
39. Guibas L., Stolfi J. Primitives for the manipulation of general subdivisions and the computation of Voronoi diagrams // ACM Transactions on Graphics. Vol. 4, № 2. 1985. P. 74–123.
40. Guttmann A., Stonebraker M. Using a Relational Database Management System for Computer Aided Design Data // IEEE Database Engineering. 1982. Vol. 5, № 2.
41. Heller M. Triangulation algorithms for adaptive terrain modeling // Proc. of the 4th Intern. Symp. on Spatial Data Handling, July 1990. P. 163–174.
42. Kirkpatrik D.G. A note on Delaunay and optimal triangulations // Inform. Process. Lett. 1980. Vol. 10. P. 127–128.
43. Kirkpatrik D.G. Optimal search in planar subdivisions // SIAM J. Comput., 1983. Vol. 12, № 1. P. 28–35.
44. Kornmann D. Fast and simple triangle strip generation / Technical Report, Varian Medical Systems Finland, Espoo, 1999. 5 p.
45. Lawson C. Software for C^1 surface interpolation // Mathematical Software III. NY: Academic Press, 1977. P. 161–194.
46. Lawson C. Transforming triangulations // Discrete Mathematics. 1972. № 3. P. 365–372.
47. Lee D. Proximity and reachability in the plane // Tech. Rep. N. R-831, Coordinated Sci. Lab. Univ. of Illinois at Urbana. 1978. 157 p.
48. Lee D., Schachter B. Two algorithms for constructing a Delaunay triangulation // Int. Jour. Comp. and Inf. Sc. 1980. Vol. 9, № 3. P. 219–242.
49. Lee J. Comparison of existing methods for building triangular irregular network models of terrain from grid digital elevation models // Int. Journal of GIS. 1991. Vol. 5, № 3. P. 267–285.
50. Levcopoulos C. An $\Omega(\sqrt{n})$ lower bound for the nonoptimality of the greedy triangulation // Inform. Process. Lett. 1987. Vol. 25. P. 247–251.
51. Levcopoulos C., Krznaric D. Quasi-greedy triangulations approximating the minimum weight triangulation // Tech. Rep. N. LU-CS-TR. Dept. of Computer Science, Lund University, Sweden. 1995. P. 95–155.
52. Levcopoulos C., Krznaric D. Tight lower bounds for minimum weight triangulation heuristics // Inform. Process. Lett. 1996. Vol. 57. P. 129–135.
53. Levcopoulos C., Lingas A. The greedy triangulation approximates the minimum weight triangulation and can be computed in linear time in the average case // Technical Report LU-CS-TR. Dept. of Computer Science, Lund University, Lund, Sweden, 1992. P. 92–105.

54. Lewis B., Robinson J. Triangulation of planar regions with applications // The Computer Journal. 1978. Vol. 21, № 4. P. 324–332.
55. Lingas A. The Greedy and Delaunay triangulations are not bad… // Lect. Notes Comp. Sc. 1983. Vol. 158. P. 270–284.
56. Lloyd E. On triangulation of a set of points in the plain // MIT Lab. Comp. Sc. Tech. Memo. 1977. № 88. 56 p.
57. Manacher G., Zobrist A. Neither the Greedy nor the Delaunay triangulation of planar point set approximates the optimal triangulation // Inf. Proc. Let. 1977. Vol. 9, № 1. P. 31–34.
58. McCullagh M.J., Ross C.G. Delaunay triangulation of a random data set for isarithmic mapping // The Cartographic Journ. 1980. Vol. 17, № 2. P. 93–99.
59. 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, February 1993.
60. Mulzer W., Rote G. Minimum-weight triangulation is NP-hard // 22nd Annual Symposium on Computational Geometry, 2006.
61. Nagy G. Terrain visibility // Computers and Graphics. 1994. Vol. 18, № 6.
62. Plaisted D.A., Hong J. A heuristic triangulation algorithm // J. Algorithms. 1987. Vol. 8. P. 405–437.
63. Puppo E. Variable resolution triangulations // Computational Geometry. 1998. Vol. 11. P. 219–238.
64. Santos F., Seidel R. A better upper bound on the number of triangulations of a planar point set // J. Combin. Theory Ser. A. 2003. Vol. 102. P. 186–193.
65. Shapiro M. A note on Lee and Schachter's algorithm for Delaunay triangulation // Int. Jour. of Comp. and Inf. Sciences. 1981. Vol. 10, № 6. P. 413–418.
66. Sibson R. Locally equiangular triangulations // Computer Journal. 1978. Vol. 21, № 3. P. 243–245.
67. Sloan S.W. A fast algorithm for constructing Delaunay triangulations in the plane // Adv. Eng. Software. 1987. Vol. 9, № 1. P. 34–55.
68. Speckmann B., Snoeyink J. Easy triangle strips for TIN terrain models // Proc. of 9th Canadian Conference on Comput. Geometry, 1997. P. 239–244.
69. Stewart A.J. Tunneling for triangle strips in continuous Level-of-Detail meshes // Graphics Interface. 2001, June. P. 91–100.
70. Touma G., Rossignac J. Geometric compression through topological surgery // ACM Transactions on Graphics. 1998. Vol. 17, № 2. P. 84–115.
71. Voronoi G. Nouvelles applications des parameters continues à la therie des formes quadratiques. Deuxième Mémorie: Recherches sur les parralléloèddres primitifs // J. reine angew. Math. 1908. № 134. P. 198–287.
72. Watson D.F. Computing the n-dimensional Delaunay tessellation with application to Voronoi polytopes // The Computer Journal. 1981. Vol. 24, № 2. P. 167–172.

Научное издание

Скворцов Алексей Владимирович
Мирза Наталья Сергеевна

АЛГОРИТМЫ ПОСТРОЕНИЯ И АНАЛИЗА ТРИАНГУЛЯЦИИ

Редактор *Е.В. Лукина*

Лицензия ИД № 04617 от 24.04.2001 г.
Подписано в печать 05.06.2006 г. Формат 60x84 1/16.
Бумага офсетная № 1. Печать офсетная.
Печ. л. 10,50; усл.печ.л. 9,76; уч.-изд.л. 9,56. Тираж 500 экз. Заказ 404.

ОАО «Издательство ТГУ», 634029, г. Томск, ул. Никитина, 4.
Типография «Иван Фёдорович», 634003, Томск, Октябрьский взвоз, 1.

А.В. Скворцов

доктор технических наук, профессор
Томского государственного университета,
директор ООО "ИндорСофт".
Автор более 150 научных работ, в том числе
3 монографий и 4 учебных пособий.

Область научных интересов:
программирование, вычислительная геометрия, компьютерная графика, геоинформатика, разработка геоинформационных систем (ГИС) и систем автоматизированного проектирования (САПР).

Н.С. Мирза

кандидат технических наук,
руководитель проектов
ООО "ИндорСофт".
Автор более 40 научных работ.

Область научных интересов:
компьютерная графика, вычислительная геометрия, геоинформатика и автоматизированное проектирование.