Д. М. Васильков

ГЕОМЕТРИЧЕСКОЕ МОДЕЛИРОВАНИЕ И КОМПЬЮТЕРНАЯ ГРАФИКА

ВЫЧИСЛИТЕЛЬНЫЕ И АЛГОРИТМИЧЕСКИЕ ОСНОВЫ

Курс лекций

Минск
БГУ
2011

УДК 004.92+519.1(075.8)
ББК 32.973.26-018.2я73+22.176я73
В19

Печатается по решению
Редакционно-издательского совета
Белорусского государственного университета

Рецензенты:
зав. кафедрой дискретной математики
и алгоритмики ФПМИ БГУ, доктор
физико-математических наук, профессор *В. М. Котов*;
руководитель отдела разработки программного
обеспечения СП «Кредо-диалог»-ООО,
кандидат технических наук *Л. В. Красильникова*

Васильков, Д. М.

В19 Геометрическое моделирование и компьютерная графика : вычислительные и алгоритмические основы [Электронный ресурс] : курс лекций / Д. М. Васильков. – Минск : БГУ, 2011. Режим доступа : http://www.elib.bsu.by, ограниченный.

ISBN 978-985-518-559-9.

Курс лекций содержит разделы теории алгоритмов, вычислительной геометрии, геометрического моделирования и компьютерной графики, составляющие тематику предмета, посвященного теоретическим основам создания систем обработки векторной геометрической информации. Изложенный материал представляет интерес для студентов, обучающихся по специальностям «Информатика» и «Прикладная математика».

УДК 004.92+519.1(075.8)
ББК 32.973.26-018.2я73+22.176я73

ISBN 978-985-518-559-9

Оглавление

Глава 1

Введение

В настоящее время рынок программных продуктов предлагает широкий выбор систем автоматизированного проектирования, предназначенных для решения задач в различных областях инженерной деятельности, таких как машиностроение, архитектура, гражданское строительство, геодезия, картография и геоинформатика. Поистине огромен ассортимент компьютерных игр, поражающих неискушенного пользователя реалистичностью изображений. Не отстают от игр компьютерная анимация и кинематография, где применение средств компьютерной графики стало нормой. Конкурентоспособность программных продуктов все более зависит от их интеллектуальной насыщенности, которая проявляется в скорости обработки информации и сложности решаемых задач. Многое зависит от удобства интерфейса и скорости обучения работе с данным программным продуктом.

Возникает вопрос: какими знаниями должен обладать специалист, занимающийся разработкой систем геометрического проектирования и компьютерной графики? Каждый опытный разработчик понимает, что владения одним или несколькими языками программирования недостаточно. Мало также освоить продвинутые технологии, такие как COM или DirectX. Что вообще отличает вчерашнего выпускника программистского факультета от этакого гуру, который на своем рабочем месте не столько занимается написанием кода, сколько играет роль оракула, отвечая на вопросы коллег? Можно ответить так: разумеется, опыт и знания. Но что это за знания?

Что касается разработки сложных программных систем *в целом*, то ответ вкратце такой: изучайте алгоритмы. Или, более точно, фундаментальную теорию алгоритмов. Читайте Кнута, Кормена и К°, Ахо и К° и т. д. Нередко разработчик тратит уйму времени на оптимизацию кода на уровне средств языка, используя при этом

ассемблерные вставки и другие программистские хитрости, тогда как решением проблемы могла бы стать простая замена линейного контейнера на древовидный.

Чтобы проверить свой уровень знания алгоритмов, проведите эксперимент. Пусть требуется прочитать из файла и вывести на экран векторную карту, содержащую заданное число графических примитивов – точек, отрезков, многоугольников и строк текста. Предположим, что карта достаточно большая и не помещается полностью в области экрана, поэтому для ее просмотра необходимо реализовать операции панорамирования (с помощью полос прокрутки или «лапки») и масштабирования. Обе операции должны выполняться в реальном времени. Это означает, что изображение обновляется при каждом перемещении мыши. Начните с просмотра небольших карт, включающих до 1000 примитивов. Как будто все не так плохо, верно? Постепенно увеличиваем число примитивов до 1 миллиона и дальше. Если изображение обновляется по-прежнему без заметных задержек, значит алгоритмы и структуры данных выбраны верно. Если же перерисовка выполняется с опозданием, читайте статьи про многомерный поиск или главу 2 этой книги.

Данный курс лекций охватывает темы, связанные с разработкой алгоритмов для систем компьютерной графики и геометрического моделирования. Взятые по отдельности, эти темы входят в различные области прикладной математики и информатики, такие как фундаментальная теория алгоритмов, вычислительная математика, геометрическое моделирование, вычислительная геометрия и компьютерная графика. Каждая из названных дисциплин исследует свой круг задач, возникающих на различных этапах создания систем обработки геометрической информации.

Как правило, программные компоненты и библиотеки, отвечающие за решение геометрических задач, моделирование пространственных объектов и их отображение, формируют ядро, или, как часто говорят, «движок» системы, определяя ее качество в целом. Можно обобщить и сформулировать требования к такому ядру с точки зрения полноты базовых функций и их вычислительной эффективности. Предположим, что поставлена задача создать программное приложение, выполняющее обработку плоских или трехмерных векторных объектов, включая:

- навигацию в пределах сцены: панорамирование, прокрутку, масштабирование и позиционирование;

- эффективное с точки зрения быстродействия и затрат памяти решение геометрических задач большой размерности;
- моделирование кривых и поверхностей в пространстве;
- построение реалистических изображений трехмерных объектов.

В материале лекций мы рассмотрим основные вычислительные алгоритмы, необходимые для создания такого абстрактного приложения, уделив особое внимание анализу их эффективности.

При подготовке данного курса лекций использовался методический материал ряда известных монографий и учебных пособий. В частности, базовые алгоритмы и структуры данных изложены в замечательном учебнике [25]. Математические основы моделирования кривых и поверхностей описаны в классической работе Г. Фарина [7]. Монографии [3, 28] отражают современное состояние вычислительной геометрии. Практическая реализация некоторых геометрических алгоритмов рассмотрена в книге М. Ласло [27]. Двухтомник Фоли и Ван Дэма [32], а также работы [12, 29, 33, 34] содержат много полезной информации для тех, кто изучает алгоритмы компьютерной графики.

1.1. Модель вычислений

При анализе эффективности алгоритма оценивается зависимость времени его выполнения и затрат памяти от размера входных данных. Чтобы оценить время выполнения, нам понадобится понятие *модели вычислений*. Под моделью вычислений понимается набор допустимых элементарных операций, стоимость которых постоянна. Выбор модели существенно зависит от класса решаемых задач. Будем рассматривать в качестве модели абстрактную машину с произвольным доступом к памяти, каждая ячейка которой способна хранить вещественное число, и допускающую выполнение следующих элементарных операций:

1) арифметические операции ($+$, $-$, $*$, $/$);
2) операции сравнения ($<$, $\leq$, $>$, $\geq$, $=$, $\neq$);
3) переход по адресу;
4) вычисление аналитических выражений ($\sqrt{x}$, $\sin x$, $\log x$ и т. п.);

5) вычисление элементарных геометрических функций (расстояние от точки до прямой, площадь треугольника, скалярное и векторное произведения векторов и т. п.).

Заметим, что под элементарными геометрическими функциями понимаются лишь те, время вычисления которых ограничено сверху некоторой константой.

1.2. O-символика

Для формальной записи оценок трудоемкости алгоритмов, а также нижних оценок сложности решения задач используется так называемая O-символика (или *асимптотическая нотация*). Пусть g и f – неотрицательные функции, определенные на множестве целых неотрицательных чисел.

- Запись $g(n) = O(f(n))$ означает, что существуют константы C и $N_0 > 0$, для которых соотношение

$$g(n) \leq Cf(n)$$

выполняется для всех $n \geq N_0$. Символ O используется для обозначения *верхних* оценок скорости роста функций. Например, $an + b = O(n)$ (можно положить $C = a + 1$ и $N_0 = b$).

- Запись $g(n) = \Omega(f(n))$ означает, что существуют константы C и $N_0 > 0$, для которых соотношение

$$g(n) \geq Cf(n)$$

выполняется для всех $n \geq N_0$. Символ Ω используется для обозначения *нижних* оценок скорости роста функций.

- Запись $g(n) = \theta(f(n))$ означает, что существуют константы C_1, C_2 и $N_0 > 0$, для которых соотношение

$$C_1 f(n) \leq |g(n)| \leq C_2 f(n)$$

выполняется при всех $n \geq N_0$. Легко доказать, что $g(n) = \theta(f(n))$ тогда и только тогда, когда $g(n) = O(f(n))$ и $g(n) = \Omega(f(n))$.

1.3. Асимптотический анализ

Будем оценивать *асимптотическую эффективность* алгоритмов, то есть рост времени выполнения алгоритма при стремлении размера входных данных к бесконечности. Рассмотрим пример.

Задача 1. *На плоскости задано множество S из n отрезков. Определить, пересекаются ли какие-либо два из них.*

Простейший алгоритм состоит в проверке каждой пары отрезков на пересечение:

```
Идентификация-пересечения (S)
 1  for i ← 1 to n − 1 do
 2      for j ← i + 1 to n do
 3          if S[i] пересекает S[j]
 4              return TRUE
 5  return FALSE
```

Обозначим через $T(n)$ время выполнения алгоритма. Нетрудно заметить, что если S не содержит других пересекающихся отрезков, кроме S_{n-1} и S_n, то для идентификации пересечения придется перебрать все пары отрезков. Тогда время, затраченное на проверку пар отрезков на пересечение, будет равно

$$c_1(n-1) + c_1(n-2) + ... + c_1 = c_1\frac{n(n-1)}{2},$$

где c_1 – время, затраченное на проверку пересечения двух отрезков. Но кроме того, время работы алгоритма включает время реализации внешнего и внутреннего циклов (суммирование индексов и переходы по адресу), а также время, затраченное на инициализацию переменных и возврат значения. Поэтому суммарное время будет равно

$$T(n) = (c_1 + c_2)\frac{n(n-1)}{2} + c_3 n + c_4, \tag{1.1}$$

где c_2 и c_3 – время реализации, соответственно, одного внутреннего и одного внешнего цикла, а c_4 – время, затраченное на инициализацию переменных и возврат значений.

Покажем, что $T(n) = \Theta(n^2)$. Для этого запишем (1.1) в виде

$$T(n) = an^2 + bn + c,$$

где $a = (c_1 + c_2)/2$, $b = c_3 - (c_1 + c_2)/2$ и $c = c_4$, и выясним, при каких значениях C_1, C_2 и N_0 неравенство

$$C_1 n^2 \leq an^2 + bn + c \leq C_2 n^2 \tag{1.2}$$

выполняется для всех $n \geq N_0$. Разделим (1.2) на n^2:

$$C_1 \leq a + \frac{b}{n} + \frac{c}{n^2} \leq C_2.$$

Если положить

$$C_1 = \frac{a}{2}, \quad C_2 = a + |b| + c, \quad N_0 = \left\lceil \frac{2|b|}{a} \right\rceil$$

(заметим, что b может принимать отрицательные значения!), то получим, что

$$C_1 n^2 \leq T(n) \leq C_2 n^2$$

выполняется для всех $n \geq N_0$, то есть $T(n) = \theta(n^2)$.

Что это означает с точки зрения анализа эффективности алгоритма? Запись $T(n) = \theta(n^2)$ дает краткую и удобную характеристику того, как ведет себя функция $T(n)$ при достаточно больших значениях n. Кроме того, если рассматривать $\theta(f(n))$ как класс функций с одинаковой асимптотической оценкой, то алгоритмы также могут быть разделены на аналогичные классы в соответствии с их асимптотической сложностью. Ниже перечислены некоторые из таких классов и примеры представляющих их алгоритмов.

1. $T(n) = \theta(1)$ – алгоритмы, решающие задачу за константное время, независимо от ее размерности.

2. $T(n) = \theta(\log n)$ – алгоритмы типа «разделяй и властвуй», разбивающие задачу на подзадачи примерно одинакового размера с последующим рекурсивным решением одной из подзадач.

3. $T(n) = \theta(n)$ – алгоритмы, решающие задачу за один линейный проход либо уменьшающие размерность задачи на каждый проход со скоростью геометрической прогрессии.

4. $T(n) = \theta(n \log n)$ – алгоритмы типа «разделяй и властвуй», разбивающие задачу на две подзадачи примерно одинакового размера с последующим рекурсивным решением каждой из подзадач за линейное время.

5. $T(n) = \theta(n^2)$ – алгоритмы, осуществляющие обработку каждого элемента за линейное время.

6. $T(n) = \theta(2^n)$ – переборные алгоритмы.

1.4. Контекст применения алгоритмов

В представленной ниже таблице с точностью до порядка приводится количество элементарных операций, выполняемых алгоритмами из приведенных выше классов для решения задач размерности 10^3 и 10^6. Из таблицы следует, что при решении задач с числом примитивов порядка 10^6 (вполне реальная размерность для объектов машиностроения и архитектуры) алгоритмы сложности $O(n \log n)$ работают примерно в 100 000 раз быстрее, чем алгоритмы сложности $O(n^2)$! Такое же значительное преимущество имеют алгоритмы сложности $O(\log n)$ по сравнению с линейными.

$T(n)$	$n = 10^3$	$n = 10^6$
$\log n$	10	20
n	10^3	10^6
$n \log n$	10^4	10^7
n^2	10^6	10^{12}
2^n	10^{300}	10^{300000}

В программном приложении, связанном с обработкой больших объемов геометрических данных, различные функции требуют различной скорости обработки. Мало того, все функции можно разбить на классы по критерию максимально допустимого времени их выполнения.

Рассмотрим пример. Стандартное Windows-приложение представляет собой набор обработчиков сообщений, которые поступают от различных устройств, таких как клавиатура или мышь. Некоторые сообщения, например `WM_MOUSEMOVE`, могут приходить настолько часто, что для их обработки требуется предельно высокая скорость. Типичным примером обработчика сообщения `WM_MOUSEMOVE` является перерисовка содержимого векторной карты при панорамировании или масштабировании. За сотые доли секунды должен быть выполнен запрос на выборку геометрических объектов, которые пересекают прямоугольную область, задающую видимую часть карты. Нетрудно подсчитать, что алгоритм с линейной трудоемкостью не справится с такой задачей даже для карты небольшого размера.

Существуют «менее требовательные» сообщения, такие как сообщение `WM_LBUTTONDOWN`, посылаемое приложению при нажатии на

левую кнопку мыши. Предположим, что пользователь щелкает на карте, чтобы выделить объект для последующего редактирования или удаления. Позиция курсора и объекты карты заданы своими координатами, необходимо найти объект, ближайший к позиции курсора. Если в данной ситуации использовать простой линейный алгоритм, задержка вряд ли превысит 0.1–0.5 секунды и будет практически незаметна для глаза.

Наконец, некоторые сообщения вызывают обработчик, запускающий целый вычислительный процесс (например, построение модели рельефа земной поверхности). В этом случае пользователь готов подождать по ряду причин, одна из которых состоит в том, что, будучи проделанная вручную, подобная процедура займет не секунды, а дни или даже недели. Несколько секунд при размерности порядка 10^5–10^6 это время работы алгоритмов с трудоемкостью $O(n \log n)$.

Разумеется, в ряде случаев допустимо использовать алгоритмы с трудоемкостью $O(n^2)$ и выше – при условии, что более быстрые алгоритмы неизвестны.

Таким образом, выбор алгоритма определяется контекстом его применения. Здесь уместна еще одна рекомендация. Всегда начинайте оптимизацию программы с поиска «узкого места» – фрагмента кода (компоненты, процедуры или даже отдельного оператора), который вызывается *намного* чаще других фрагментов и суммарное время работы которого *резко* выделяет его относительно других участков программы. Уделите оптимизации данного фрагмента особое внимание, поскольку именно он определяет производительность всей системы в целом. Необходимо выполнить несколько итераций: устранение одного «узкого места» приводит к появлению нового «узкого места», которое, в свою очередь, также требует оптимизации. Указанные действия производятся до тех пор, пока они дают ощутимый выигрыш в общей производительности.

Описанный процесс называется *профилированием*. Развитые среды разработки, как правило, предоставляют для оптимизации кода необходимые инструменты.

1.5. Запись алгоритмов

В данной книге отсутствуют исходные тексты программ. Вместо них для записи алгоритмов используется псевдокод, правила оформления которого описаны в [25].

Существуют веские причины, чтобы не прибегать к помощи языков программирования для описания алгоритмов. Во-первых, нехорошо вводить читателя в заблуждение, предлагая фрагменты программ на C++ или Паскале, якобы готовые к использованию. Дело в том, что при записи алгоритма некоторые важные, но громоздкие детали реализации приходится опускать, чтобы более ясно продемонстрировать основную идею. В результате такой «фильтрации» может получиться программный код, который разительно отличается от промышленного варианта. Удивительно, но немногие разработчики знают, что функция, вычисляющая ориентацию точки относительно направленного отрезка, заданного другими двумя точками, и которая *правильно и быстро работает при любых значениях входных параметров*, занимает не одну и не две, а ... около 500(!) строк кода на языке C (см. [15]).

Во-вторых, изучать алгоритмы по программному коду просто неудобно. К счастью, Кормену с соавторами, похоже, удалось найти по-настоящему удачный формат записи алгоритма. Отличительной его чертой является использование структурных отступов вместо функциональных скобок типа **begin** - **end**. Оформленный в таком стиле алгоритм становится намного более компактным и понятным.

Перечислим другие важные соглашения. Символ $\triangleright$ означает начало комментария. Поля составных переменных (структур) и свойства объектов записываются с помощью квадратных скобок: например, $x[P]$ обозначает поле x структуры P. Для сравнения используется знак $=$. Присваивание переменной a значения b обозначается стрелкой: $a \leftarrow b$. Рассмотренный в разделе 1.3 алгоритм Идентификация-пересечения оформлен именно в этом стиле.

1.6. Структуры данных

Если не вдаваться в детали реализации[1], то каждую структуру данных можно рассматривать как *контейнер*, содержащий элементы некоторого множества. Контейнеры различаются друг от друга набором операций, производимых над множеством. Представление структур данных в виде списка функций, реализующих эти операции, с указанием их трудоемкости будем называть *внешним*. В отличие от *внутреннего* представления, которое включает организацию хранения

[1]Предполагается, что читатель знаком с базовыми структурами данных, такими как массивы, списки, деревья, очереди и стеки. Рекомендуемые учебники: [25, 26, 20, 22].

данных в памяти и алгоритмы, обеспечивающие нужное время доступа к элементам множества, внешнее представление рассматривает структуры данных как «черные ящики» с заданным набором интерфейсов[2]. Ниже мы дадим краткую классификацию структур данных относительно их внешнего представления.

Одна из общепринятых классификаций различает структуры данных по способу доступа к элементам множества. Таких способов всего три: *случайный*, *последовательный* и *доступ по ключу*. Контейнеры, которые их поддерживают, называются, соответственно, *вектор*, *список* и *отображение* (см. [31, 24]).

Любой элемент a вектора V может быть получен за константное время по его порядковому номеру (индексу) i вызовом функции $\textsc{At}(V,\ i)$. На псевдокоде эта операция записывается с использованием квадратных скобок: $a \leftarrow V[i]$. Относительно реализации вектора предполагается, что его элементы занимают непрерывную область памяти, что позволяет вычислить относительный адрес элемента по его номеру. Однако такой способ хранения делает крайне неэффективными операции вставки и удаления, поскольку требует перераспределения памяти.

Последовательный доступ выполняется с помощью функции $\textsc{Next}(L,\ a)$, которая за время $O(1)$ возвращает элемент списка L, следующий непосредственно за элементом a. Поиск нужного элемента начинается с первого элемента $head[L]$, от него осуществляется переход к следующему элементу и т. д. Таким образом, для того чтобы найти нужный элемент, возможно, придется просмотреть все элементы списка, то есть выполнить $O(n)$ действий.

Доступ по ключу предполагает, что элементы множества упорядочены по некоторому значению, которое называется *ключом*. Отображение M хранит пары вида (x, y), где x – ключ, а y – значение элемента. Ключ x однозначно определяет значение y, поэтому M реализует не произвольное отображение, а функцию. Поиск элемента осуществляется с помощью функции $\textsc{Find}(M,\ x)$, возвращающей значение y. Отображение может быть реализовано в виде АВЛ-дерева, которое выполняет поиск за время $O(\log n)$.

Структуры данных могут иметь специальные режимы вставки и удаления элементов. Различают *очереди*, которые разрешают вставлять новый элемент только в конец (то есть непосредственно за

[2] Для структур данных в таком понимании существует также другое название – *абстрактный тип данных* [21].

последним элементом), а удалять только первый элемент, и *стеки*, разрешающие вставку и удаление только с одного конца. Обе операции для очередей и стеков выполняются за константное время с помощью функций Push(A, a) и Pop(A), имеющих для очередей и стеков общее обозначение, но разную интерпретацию.

Относительно упорядоченных множеств существуют другие важные операции: вставка, удаление и поиск элемента по значению, разделение и объединение множеств, а также поиск минимального элемента. Каждое из указанных действий выполнимо за время $O(\log n)$.

Определенные выше операции над множествами с указанием времени их выполнения приведены в следующей таблице:

Операция	Время	Описание
At(A, i)	$O(1)$	Возвращает i-й элемент множества A.
Next(A, a)	$O(1)$	Возвращает элемент множества A, следующий за элементом a.
Prev(A, a)	$O(1)$	Возвращает элемент множества A, стоящий перед элементом a.
Push(A, a)	$O(1)$	Вставляет a за последним элементом множества A, если A – очередь, или перед первым элементом, если A – стек.
Pop(A)	$O(1)$	Удаляет первый элемент множества A и возвращает его.
Find(A, x)	$O(\log n)$	Возвращает второй элемент пары (x, y) отображения A.
Member(A, a)	$O(\log n)$	Возвращает true, если $a \in A$.
Insert(A, a)	$O(\log n)$	Добавляет в A элемент a.
Delete(A, a)	$O(\log n)$	Удаляет элемент a из множества A.
Min(A)	$O(\log n)$	Возвращает минимальный элемент множества A.
Split(A, a)	$O(\log n)$	Разделяет множества A на два подмножества A_1 и A_2 такие, что $A_1 = \{x \in A \mid x \leq a\}$ и $A_2 = A \backslash A_1$.
Merge(A_1, A_2)	$O(\log n)$	Объединяет множества A_1 и A_2 в одно множество $A = A_1 \cup A_2$. Предполагается, что $a_1 \leq a_2$ для любых $a_1 \in A_1$ и $a_2 \in A_2$.

Ниже приведена классификация структур данных относительно их внешнего представления. Для каждого типа контейнера указан набор операций над множествами, которые он поддерживает.

Контейнер	Интерфейс
1. Вектор	At
2. Список	Next, Prev
3. Очередь и стек	Push, Pop
4. Отображение	Find, Insert, Delete
5. Словарь	Member, Insert, Delete
6. Приоритетная очередь	Min, Insert, Delete
7. Сцепляемая очередь	Insert, Delete, Split, Merge
8. Прошитый словарь	Member, Insert, Delete, Next, Prev

Структуры данных (1) – (3) относятся к произвольным множествам, остальные – к упорядоченным.

При необходимости можно разработать структуру данных, которая поддерживает сразу несколько интерфейсов. Например, приведенный в таблице прошитый словарь объединяет два интерфейса: один относится к списку, другой – к словарю. Оба интерфейса могут быть реализованы с помощью АВЛ-дерева, каждый элемент которого хранит ссылку на следующий элемент относительно данного порядка.

В заключение отметим, что всегда полезно иметь «джентельменский набор» алгоритмов и структур данных собственного производства (см. упражнения). Для начала запрограммируйте АВЛ-дерево, научитесь работать с хэш-таблицами, B-деревьями и т. д. В качестве альтернативы можно использовать готовые решения, например библиотеку STL, предлагающую эффективную реализацию большинства операций для работы с множествами.

Упражнения

1-1. Разработайте на языке C++ класс контейнера, методы которого позволяют работать с ним как со списком, очередью и стеком.

1-2. Разработайте класс, реализующий АВЛ-дерево. Создайте на его основе контейнеры типа словарь, прошитый словарь, приоритетная очередь и отображение.

1-3. Напишите класс для работы с 2-3-деревом (см. [20]) и реализуйте на его основе сцепляемую очередь.

Глава 2

Геометрический поиск

Пусть имеется набор геометрических данных, относительно которых делается запрос, например, требуется найти выпуклую оболочку заданного множества точек, подсчитать количество пересечений множества отрезков или число точек внутри некоторой области. Если запрос выполняется единожды, то он называется *уникальным*. Если для фиксированного набора исходных данных запрос может повторяться, то он называется *массовым*. При этом предметом оптимизации может стать структура данных, ускоряющая выполнение массового запроса, даже если на ее построение необходимо затратить большие ресурсы. Построение и оптимизация такой структуры до выполнения первого запроса называется *предобработкой*. В дальнейшем будем оценивать следующие характеристики алгоритмов:

- время выполнения запроса (уникального или массового);
- время предобработки;
- требуемая память.

Различают две модели задач геометрического поиска:

1. *Задачи регионального поиска*, в которых в качестве фиксированного набора данных выступает множество геометрических объектов на плоскости или в пространстве, а выполнение запроса состоит в подсчете или перечислении этих объектов внутри заданной области.
2. *Задачи локализации точки*, в которых набором исходных данных является разбиение пространства на области, а запрос заключается в определении, какой из областей принадлежит заданный геометрический объект.

2.1. Региональный поиск: подсчет

Начнем с простейшего случая, когда в качестве геометрических объектов рассматриваются точки, а в качестве области запроса – прямоугольник со сторонами, параллельными координатным осям (рис. 2.1). Такой прямоугольник называется *ортогональным*. Для зада-

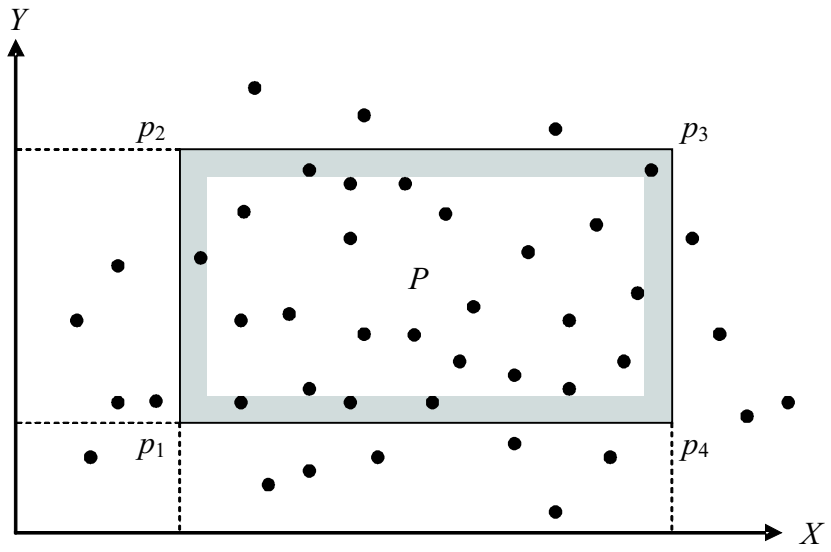


Рис. 2.1. Подсчет числа точек в прямоугольнике с помощью отношения доминирования

ния ортогонального прямоугольника достаточно указать координаты левого нижнего и правого верхнего углов, которые будем обозначать, соответственно, $P_{\min}$ и $P_{\max}$.

Задача 2 (Подсчет точек). *Дано множество S из n точек на плоскости. Сколько из них лежит внутри заданного ортогонального прямоугольника P?*

Очевидно, уникальный запрос выполняется за время $O(n)$, а расходы памяти составляют $O(2n)$. Существует ли способ выполнить запрос быстрее, чем за линейное время, при условии, что множество точек S фиксировано, а изменяется только прямоугольник запроса P? Рассмотрим общий метод, называемый методом *локусов*[1], суть которого состоит в том, что пространство запросов разбивается на подобласти, внутри которых ответ на запрос одинаковый. Говорят, что точка $w \in E^2$ доминирует над $v \in E^2$, если $v_x \leq w_x$ и $v_y \leq w_y$.

[1] Слово *locus* переводится как геометрическое место точек.

Для точки $p \in E^2$ обозначим через $Q(p)$ число точек множества S, над которыми доминирует p. Тогда число точек S в прямоугольнике $P = \{p_1, p_2, p_3, p_4\}$ равно

$$n(p_1, p_2, p_3, p_4) = Q(p_1) - Q(p_2) + Q(p_3) - Q(p_4).$$

Таким образом, задача сводится к быстрому вычислению величины $Q(q)$ для произвольной точки q. Поскольку допускается предобработка, упорядочим точки множества S так, чтобы гарантировать выполнение запроса за время $O(\log n)$. Для этого опустим на обе координатные оси $2n$ перпендикулярных прямых из каждой точки $p \in S$. Эти перпендикуляры разбивают плоскость на $(n+1)^2$ прямоугольные ячейки (некоторые ячейки неограничены с одной или двух сторон), внутри которых величины $Q(q)$ одинаковы и могут быть предварительно подсчитаны (рис. 2.2). При выполнении запроса с помощью двоичного поиска вычисляются прямоугольные области, содержащие вершины тестового прямоугольника P, и таким образом определяются соответствующие значения Q. Для примера на рис. 2.2 имеем:

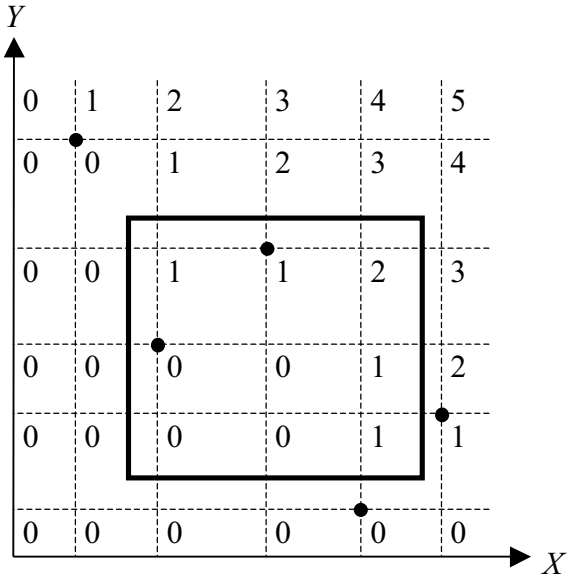


Рис. 2.2. Метод локусов: плоскость разбивается на прямоугольные ячейки таким образом, что точки внутри одной ячейки доминируют над одинаковым числом точек множества S (эти числа проставлены внутри ячеек)

$$\begin{aligned} Q(p_1) &= 0,\ Q(p_2) = 0, \\ Q(p_3) &= 3,\ Q(p_4) = 1, \end{aligned}$$

откуда

$$n(p_1, p_2, p_3, p_4) = 0 - 0 + 3 - 1 = 2.$$

Оценим затраченные ресурсы. Время выполнения запроса не превышает $O(\log n)$, на предобработку требуется $O(n^2)$ времени и $O(n^2)$ памяти. К сожалению, непомерно высокие затраты памяти существенно ограничивают допустимый размер входных данных, а значит и использование данного метода на практике.

2.2. Региональный поиск: перечисление

Задача 3 (Перечисление точек). *Дано множество S из n точек на плоскости. Перечислить те из них, которые лежат внутри заданного ортогонального прямоугольника P.*

Нетрудно заметить, что достижимая нижняя оценка сложности задачи 3 равна $\Omega(n)$: такое количество проверок требуется для перечисления точек из S, если все они лежат внутри прямоугольника P. Обозначим $m = |P \cap S|$ и рассмотрим другую нижнюю оценку – $\Omega(m)$, которая более ясно обозначает цель нашего исследования: при выполнении запроса мы должны стремиться исключить из рассмотрения как можно больше точек, не лежащих внутри P.

2.2.1. Метод сетки

Обозначим через R ортогональный прямоугольник, содержащий все точки множества S. Разобьем R вертикальными и горизонтальными прямыми на $k \times k$ одинаковые ячейки. Если точки S распределены равномерно внутри R, то ожидаемое число точек внутри одной ячейки равно $M = n/k^2$. Обратно: если мы хотим, чтобы каждая ячейка содержала в среднем M точек, то необходимо положить $k = \left\lceil \sqrt{n/M} \right\rceil$.

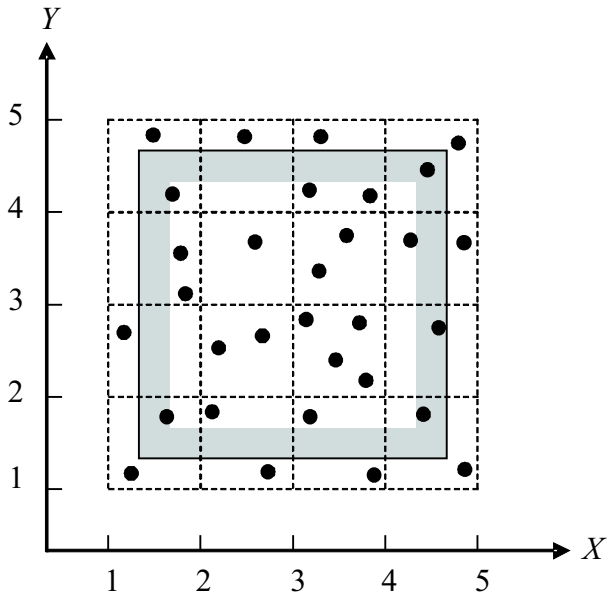


Рис. 2.3. Метод сетки: индексы ячеек можно вычислить за константное время

Такое разбиение хранится в памяти компьютера в виде матрицы Q размера $k \times k$, каждый элемент Q_{ij} которой содержит список точек множества S. Средняя длина такого списка равна M. Для перечисления точек множества S, находящихся внутри прямоугольника запроса P, необходимо вычислить интервалы $[i_{\min}, i_{\max}]$ и $[j_{\min}, j_{\max}]$ индексов ячеек сетки, имеющих непустое пересечение с P (рис. 2.3). Поскольку ячейки имеют одинаковый размер, то эти интервалы можно рассчитать за время $O(1)$.

В частности:

$$i_{\min} = \left\lfloor \frac{x[P_{\min}] - x[R_{\min}]}{r_x} \right\rfloor, \quad j_{\min} = \left\lfloor \frac{y[P_{\min}] - y[R_{\min}]}{r_y} \right\rfloor,$$

где через r_x и r_y обозначены ширина и высота ячейки:

$$\begin{aligned} r_x &= \frac{x[R_{\max}] - x[R_{\min}]}{k}, \\ r_y &= \frac{y[R_{\max}] - y[R_{\min}]}{k}. \end{aligned}$$

Из рис. 2.3 видно, что ячейки, пересекающие прямоугольник P, можно разделить на два типа: ячейки, полностью лежащие внутри P, и ячейки, частично пересекающие P. Индексы ячеек первого типа лежат в интервалах $[i_{\min}+1, i_{\max}-1]$ и $[j_{\min}+1, j_{\max}-1]$. Соответствующие им списки точек множества S сразу направляются в выходной поток. Каждая точка, принадлежащая ячейке второго типа, дополнительно проверяется на принадлежность прямоугольнику P. Ниже приведен алгоритм, реализующий запрос на перечисление точек:

```
GridSearch (P, Q)
 1  Вычислить i_min, i_max, j_min и j_max.
 2  for i ← i_min to i_max do
 3      for j ← j_min to j_max do
 4          if i > i_min and i < i_max and j > j_min and j < j_max then
 5              p ← head[Q[i, j]]
 6              while p ≠ NIL do
 7                  Push(A, p)
 8                  p ← next[p]
 9          else
10              p ← head[q[i, j]]
11              while p ≠ NIL do
12                  if p ∈ P then
13                      Push(A, p)
14                  p ← next[p]
15  return A
```

Оценим трудоемкость алгоритма. Нетрудно описать наихудший случай работы алгоритма, когда прямоугольник запроса P пересекает k^2 ячеек и при этом не содержит ни одной точки ($m = 0$). Такой запрос выполняется за время $O(k^2 + n) = O(n)$. Однако при условии, что внутри области R точки распределены равномерно, алгоритм работает быстро и имеет оценку трудоемкости в среднем $O(m)$.

2.2.2. Квадратичное дерево

Метод сетки работает неэффективно, если точки множества S расположены нерегулярно: некоторые ячейки не содержат ни одной точки, тогда как другие содержат их слишком много. Очевидно, данная проблема не может быть решена простым увеличением числа ячеек.

Рассмотрим структуру данных T, называемую *квадратичным деревом разбиения*. Сопоставим охватывающему прямоугольнику R вершину дерева T и введем константу M, $1 \leq M \leq n$. Если число точек внутри R превосходит M, то разобьем его на четыре равных прямоугольных домена двумя перпендикулярными прямыми, параллельными координатным осям. Образованным доменам соответствуют четыре вершины дерева T, выходящие из корневой вершины. Каждый из доменов рекурсивно разбивается до тех пор, пока число точек множества S внутри каждого неразбитого домена превосходит M. Пример разбиения и соответствующего ему квадратичного дерева приведен на рис. 2.4.

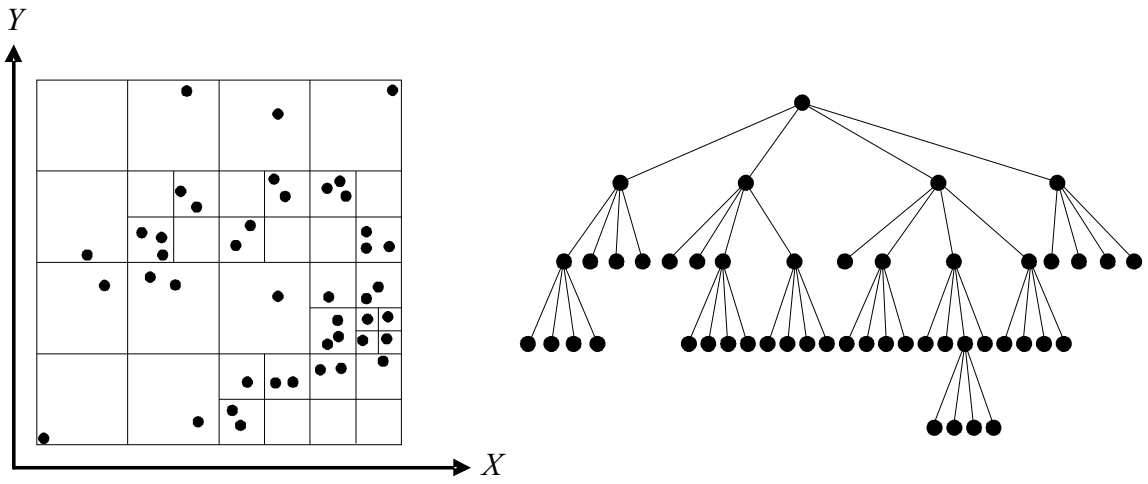


Рис. 2.4. Рекурсивное разбиение охватывающего прямоугольника и соответствующее ему квадратичное дерево T для $M = 3$

Поиск и перечисление точек производится последовательно от корня к листьям дерева: если тестовый прямоугольник P не пересекает корневой домен R, то поиск прекращается. Если пересечение имеет место, то проверяется, не охватывает ли P домен R целиком. Если охватывает, то все точки дерева перечисляются без проверки на принадлежность P. В противном случае, задача сводится к рекурсивной проверке каждого из четырех дочерних доменов корневой вершины, если таковые имеются.

Какой случай для работы алгоритма считать наихудшим? Введем понятие *продуктивности* запроса. Поскольку величина m входит в оценку любого алгоритма, продуктивность запроса определяется временем $T(n)$, затраченным на дополнительные действия, необходимые для перечисления m точек. Поэтому в качестве меры продуктивности будем использовать отношение $m/(T(n)+m)$. Таким образом, запрос, выполненный с наихудшей продуктивностью, это запрос, при котором *не перечислено ни одной точки.*

Оценим максимальное время $T(n)$ выполнения такого запроса. По определению оно равно максимально возможному числу посещенных вершин дерева T без перечисления точек. В зависимости от своего расположения относительно прямоугольника запроса P элементы разбиения могут быть отнесены к одному из трех видов:

1) *внешние* домены, не пересекающиеся с прямоугольником P;

2) *внутренние* домены, лежащие целиком внутри P;

3) *пересекающие* домены, частично перекрывающиеся с P.

При выполнении запроса внешние домены игнорируются и никак не влияют на его сложность. Мы можем не рассматривать также внутренние домены, поскольку, по нашему предположению, они не содержат внутри себя точек и поэтому далее не разбиваются и не анализируются. Остается рассмотреть пересекающие домены.

Существует только пять типов частичного пересечения домена с прямоугольником P (рис. 2.5). Пересекая домен данного типа, прямоугольник P может пересекать также его дочерние домены. Возможные способы таких пересечений и соответствующие типы дочерних доменов представлены на рис. 2.6. Можно выписать рекуррентные уравнения для числа $T_i(D)$ посещений дочерних доменов в зависимости от глубины H разбиения родительского домена i-го типа.

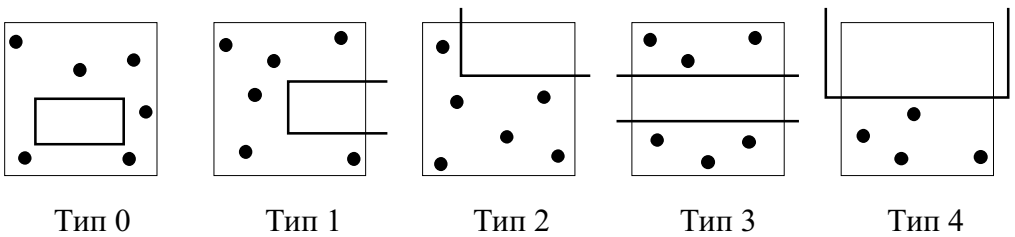


Рис. 2.5. Пять типов частичного пересечения домена с прямоугольником запроса

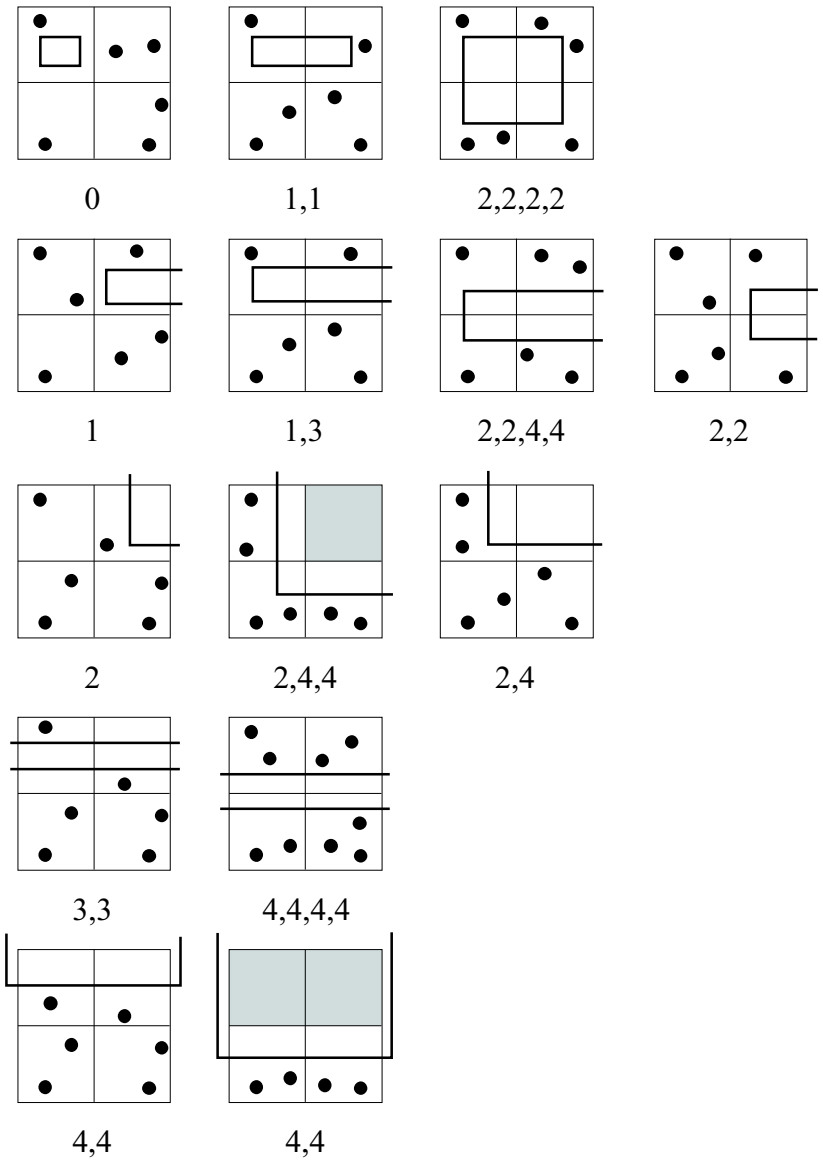


Рис. 2.6. Способы пересечения прямоугольника запроса P с дочерними доменами и типы этих доменов. Заштрихованы внутренние домены, число посещений внутри которых не учитывается

Действительно, пусть родительский домен 4-го типа имеет глубину H. Тогда получаем рекуррентное уравнение

$$T_4(H) = 2T_4(H-1),$$

решением которого является $T_4(H) = 2^{H+1}$. Для домена 3-го типа

имеем два уравнения

$$T_3(H) = \begin{cases} 2T_3(H-1) \text{ или} \\ 4T_4(H-1), \end{cases}$$

решая которые получаем неравенство $T_3(H) \leq 2^{H+2}$. Действуя аналогичным образом, можно выписать следующие верхние оценки:

$$\begin{aligned} T_2(D) &\leq 2^{H+2}, \\ T_1(D) &\leq 2^{H+3}, \\ T_0(D) &\leq 2^{H+3}. \end{aligned}$$

Все полученные оценки являются достижимыми, откуда следует, что существует такое расположение точек множества S, что при выполнении запроса будет проверено по крайней мере $2^{H+3} = O(2^H)$ узла и при этом не перечислено ни одной точки.

Если точки множества S распределены равномерно, то при больших n ожидаемое число точек в каждом листе дерева одинаковое и равно M, а число самих листьев не превосходит $\lceil n/M \rceil$. Тогда

$$H \leq \left\lceil \log_4 \frac{n}{M} \right\rceil,$$

откуда получаем верхнюю оценку для числа посещенных узлов

$$O(2^H) = O(2^{\log_4 n/M}) = O(\sqrt{n})$$

и нулевую продуктивность алгоритма в наихудшем случае.

Получить строгую оценку скорости алгоритма в среднем достаточно сложно. Однако задача упрощается, если предположить, что точки множества S распределены равномерно, а прямоугольник запроса ограничен по размеру относительно области R. Предположим, что P пересекает не более $k > 0$ листьев дерева T, где k не зависит от n. Тогда обход k листьев выполняется не более чем за время

$$O(k + H) = O\left(k + \log_4 \frac{n}{M}\right) = O(\log_4 n),$$

а число точек, содержащихся в листьях, не превосходит $kM = O(1)$. Получаем, что общее время работы алгоритма не превосходит $O(\log_4 n)$. Продуктивность алгоритма для данного случая равна $O(1/\log_4 n)$.

В случае, когда все точки множества S принадлежат прямоугольнику запроса P ($m = n$), требуется обход всех листьев дерева, и верхняя оценка времени работы алгоритма равна

$$O(n + H) = O(n) = O(m).$$

Эта оценка является оптимальной (продуктивность равна $O(1)$), хотя и совпадает с оценкой тривиального подхода, когда все точки проверяются на принадлежность прямоугольнику P.

2.2.3. 2-d-дерево

Полученные оценки в среднем для квадратичного дерева являются оптимистическими: они справедливы, если выполняется ряд условий, важнейшим из которых является сбалансированность дерева. Сбалансированность, в свою очередь, обеспечивается только при равномерном распределении точек, что далеко не всегда имеет место на практике. Причина этого недостатка кроется в том, что при построении дерева учитываются *количество* точек внутри домена, а не их фактическое местоположение.

Рассмотрим структуру данных T, называемую *2-d-деревом разбиения*, которая строится следующим образом:

1. Из точек множества S выбирается точка p, являющаяся медианой по x-координате. Через p проводится вертикальная прямая $l_x(p)$, разбивающая множество S на два подмножества S_L и S_R точек, лежащих, соответственно, слева и справа от этой прямой. Поскольку p является x-медианой, число точек в S_L и S_R отличается не более чем на 1. Прямая $l_x(p)$ ставится в соответствие корню дерева T.

2. Множество S_L разбивается горизонтальной прямой $l_y(p_L)$, проходящей через y-медиану p_L множества S_L. Аналогичным образом разбивается множество S_R. Прямые $l_y(p_L)$ и $l_y(p_R)$ ставятся в соответствие двум потомкам корневой вершины дерева T.

3. Полученные подмножества рекурсивно разбиваются вертикальными и горизонтальными прямыми до тех пор, пока не будут покрыты все точки множества S.

На рис. 2.7 представлено разбиение множества точек и соответствующее ему 2-d-дерево.

При выполнении запроса на перечисление точек проверяется положение тестового прямоугольника P относительно вертикальной прямой $l_x(p)$, связанной с корневой вершиной. Если P целиком лежит слева или справа от $l_x(p)$, то задача рекурсивно сводится к проверке одной из дочерних вершин. Если прямоугольник пересекает прямую, то точка p проверяется на принадлежность P и задача сводится к аналогичной проверке обеих дочерних вершин.

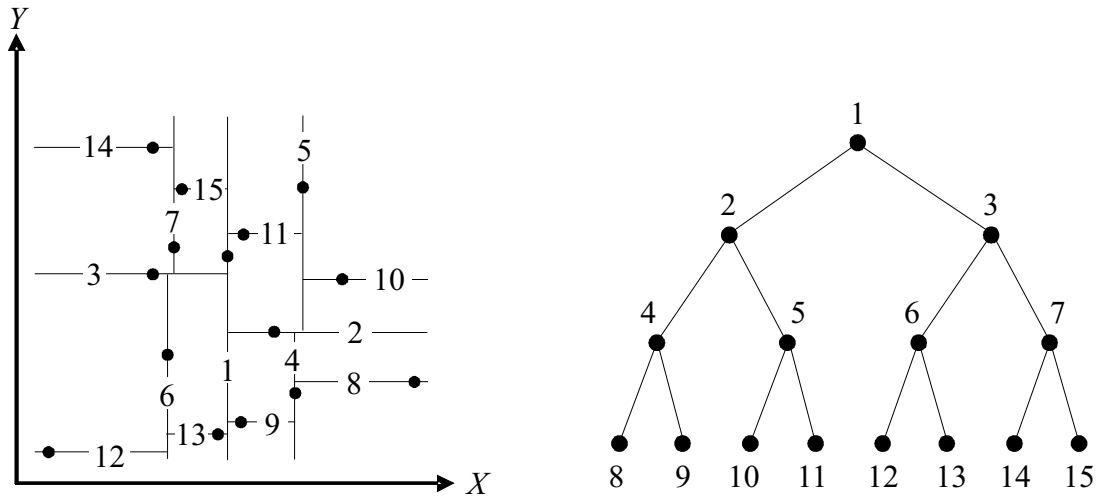


Рис. 2.7. Разбиение множества S и соответствующее ему 2-d-дерево

Важной характеристикой 2-d-дерева является его сбалансированность. Она гарантирует оценки выполнения запроса в среднем, совпадающие с аналогичными оценками для квадратичного дерева, с тем отличием, что точки множества S могут располагаться на плоскости произвольным образом. Однако 2-d-дерево имеет недостаток, не присущий квадратичному дереву: при вставке или удалении точек нарушается не только сбалансированность, но и сама древовидность структуры T, что требует дополнительных действий по перестройке 2-d-дерева.

2.3. Локализация точки: многоугольник

В задачах локализации в качестве запроса выступает произвольная точка на плоскости. Требуется определить ее местоположение относительно заданного планарного разбиения. *Планарное разбиение плоскости* представляет собой множество многоугольных областей

$\{P_1, \ldots, P_k\}$ таких, что

$$P_i \bigcap_{i \neq j} P_j = \emptyset, \quad \bigcup_{i=1}^{k} = E^2.$$

Начнем с простой задачи.

Задача 4 (Принадлежность простому многоугольнику). *Даны простой многоугольник P и точка z. Определить, находится ли z внутри P.*

Напомним, что многоугольник называется *простым*, если он не содержит самопересечений. Сделаем также следующие предположения:

- многоугольник P задан обходом вершин по часовой стрелке;
- никакие три вершины многоугольника P не лежат на одной прямой.

В случае если запрос уникальный, для решения задачи можно применить метод *выводящего луча* (рис. 2.8).

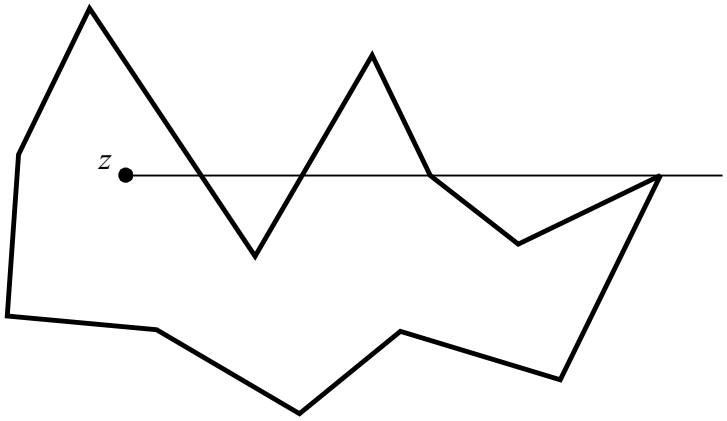


Рис. 2.8. Метод выводящего луча

1. Из точки z провести горизонтальный луч.
2. Подсчитать количество пересечений луча со сторонами многоугольника P. При прохождении луча через вершину пересечение учитывается, если две соседние к ней вершины лежат по разные стороны от прямой, задаваемой лучом, и не учитывается в противном случае.

3. Точка принадлежит многоугольнику тогда и только тогда, когда число пересечений нечетно.

Таким образом, время выполнения уникального запроса не превосходит $O(n)$. Рассмотрим теперь выполнение массового запроса отдельно для выпуклого и произвольного простого многоугольника.

Задача 5 (Принадлежность выпуклому многоугольнику). *Даны выпуклый многоугольник $P = p_1, \ldots, p_n$, $n > 2$, и точка z. Принадлежит ли точка z многоугольнику P?*

Можно ли использовать свойство выпуклости многоугольника для ускорения ответа на запрос? В дальнейшем мы не раз столкнемся с ситуацией, когда задачи над выпуклыми множествами решаются намного проще и эффективнее аналогичных проблем, в которых условие выпуклости отсутствует. Пока же заметим, что выпуклость означает упорядоченность входных данных и таким образом несет дополнительную информацию об объекте. Для определенности будем считать, что вершины многоугольника P упорядочены против часовой стрелки относительно произвольной внутренней точки. Следующий алгоритм устанавливает принадлежность точки z многоугольнику P за оптимальное время $\Theta(\log n)$, используя простой двоичный поиск:

```
ConvPolyPointLoc (P, z)
 1  i ← 2
 2  j ← n
 3  while i < j − 1 do
 4      k ← (i + j)/2
 5      if RightTurn(p_1, p_k, z) then
 6          j ← k
 7      else
 8          i ← k
 9  return InTriangle(z, p_1, p_i, p_j)
```

Многоугольник делится пополам хордой $\overline{p_1, p_k}$, затем выполняется проверка, с какой стороны от этой хорды расположена точка z (рис. 2.9, *а*). Далее рассматривается только часть многоугольника, содержащая z. Процесс деления продолжается до тех пор, пока число вершин не сократится до трех. В конце алгоритма с помощью функции InTriangle за константное время выполняется проверка принадлежности точки треугольнику.

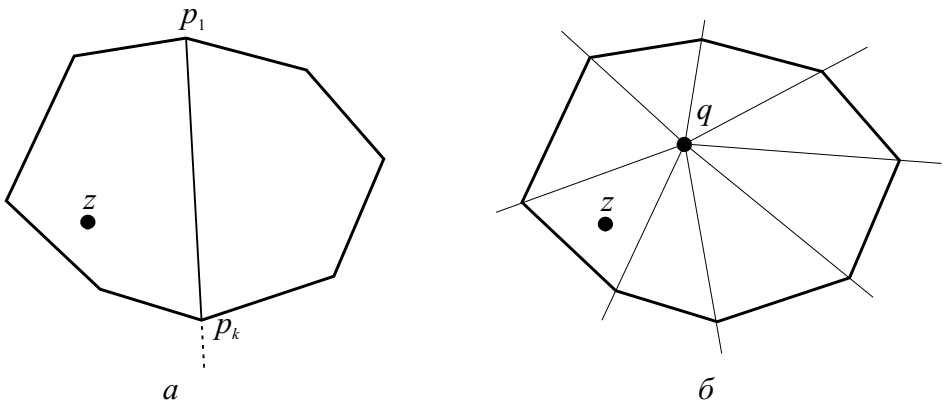


Рис. 2.9. Локализация точки относительно выпуклого многоугольника

Можно расширить круг решаемых задач, если вершины многоугольника рассматривать как упорядоченные естественным образом относительно некоторой внутренней точки. Следующий метод также основан на двоичном поиске:

1. Выбрать точку q внутри P, например, центр масс любых трех вершин (рис. 2.9, *б*).
2. Занести вершины P в массив в порядке возрастания полярного угла относительно точки q.
3. С помощью двоичного поиска найти пару соседних вершин $p_1, p_2 \in P$, задающих вместе с q сектор, содержащий точку z.
4. Точка z лежит внутри P тогда и только тогда, когда тройка (p_1, p_2, z) правая (тройка векторов (p_1, p_2, p_3) называется *правой*, если точка p_3 лежит справа от вектора $\overrightarrow{p_1 p_2}$).

Многоугольник P называется *звездным*, если существует точка $p \in P$ такая, что P содержит отрезок, соединяющий p с любой другой точкой P. Множество всех точек p, обладающих данным свойством, называется *ядром* многоугольника P.

Задача 6 (Принадлежность звездному многоугольнику).
Даны звездный многоугольник P и точка z. Принадлежит ли точка z многоугольнику P?

Если задана точка внутри ядра звездного многоугольника, то для звездного многоугольника может быть применен тот же подход, что и для выпуклого. Известно, что ядро может быть найдено за время $O(n)$, поэтому приведенные выше оценки справедливы и для задачи 6.

Нерешенным остается вопрос: можно ли улучшить время массового запроса для простого многоугольника?

2.4. Локализация точки: планарное разбиение

Планарный граф, уложенный на плоскости таким образом, что его ребра перейдут в прямолинейные отрезки, называется прямолинейным. Прямолинейный граф задает планарное разбиение плоскости на многоугольные области. Если в прямолинейном графе нет вершин со степенью меньше, чем 2, то все грани этого графа – простые многоугольники. Эффективные алгоритмы локализации на планарном разбиении основаны на предобработке, упорядочивающей планарное разбиение таким образом, что становится возможным бинарный поиск.

Задача 7 (Локализация точки на планарном разбиении). *Даны планарное разбиение* $G = \{P_1, \ldots, P_k\}$ *и точка* z. *Определить* $1 \leq i \leq k$ *такое, что* $z \in P_i$.

2.4.1. Метод полос

Пусть задано планарное разбиение G с n вершинами. Через каждую вершину проведем вертикальную прямую, получим $n + 1$ полосу (рис. 2.10). Внутри каждой из полос отрезки планарного разбиения

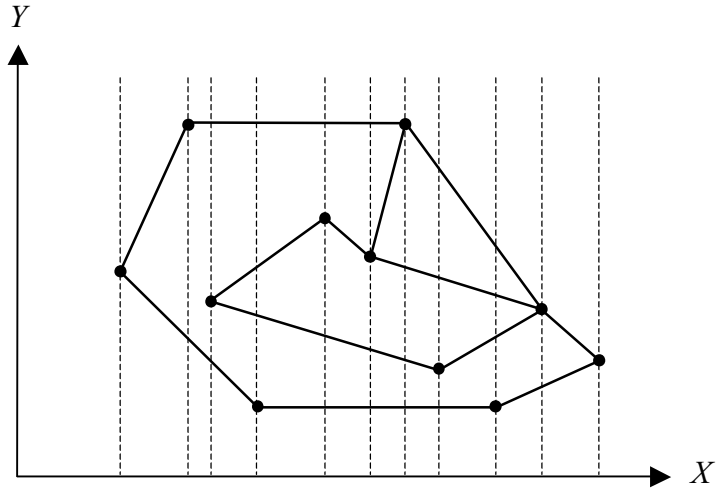


Рис. 2.10. Локализация точки: метод полос

не пересекаются, следовательно, возможно их полное упорядочивание. Применяя двоичный поиск в массиве полос, упорядоченном, скажем,

по возрастанию абсцисс, находим полосу, содержащую точку z за время $O(\log n)$. Затем, применяя тот же двоичный поиск внутри полосы, находим область, содержащую z.

Подсчитаем убытки. На хранение отрезков внутри каждой из n полос требуется $O(n^2)$ памяти. Это оценка наихудшего случая, и она достижима: в примере на рис. 2.11 общее число отрезков, хранящихся за каждой полосой, равно

$$2 + 4 + \ldots + n + (n-2) + \ldots + 2 = O(n^2).$$

При тривиальном подходе на сортировку отрезков внутри полос требуется $O(n^2 \log n)$ времени. Возникает естественное желание улучшить эту оценку. Приведенный ниже алгоритм позволяет сократить время предобработки до $O(n^2)$.

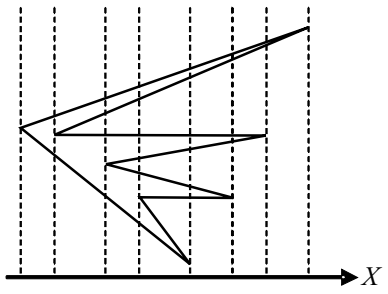


Рис. 2.11. $O(n)$ полос хранят $O(n^2)$ частей отрезков

2.4.2. Алгоритм заметающей прямой

Алгоритм заметающей прямой представляет собой общую методику, широко применяемую в вычислительной геометрии. Состоит она в следующем. Рассмотрим прямую l, параллельную одной из координатных осей, которая сканирует плоскость, двигаясь вдоль другой оси (рис. 2.12). Для определенности предположим, что l параллельна оси Y и движется в положительном направлении оси X. При этом поддерживается:

1. Список событий – упорядоченная последовательность точек на оси X, в которых происходит изменение статуса прямой l. Такие точки называются *критическими*.

2. Структура, содержащая информацию о статусе заметающей прямой. В общем случае эта информация содержит описание пересечения прямой с объектом.

В нашем случае в список событий заносятся абсциссы вершин графа G, упорядоченные по возрастанию. Статус прямой зависит от ее текущей абсциссы и представлен в виде структуры L, которая хранит ребра графа, пересекающие прямую и упорядоченные по возрастанию ординат точек пересечения. Структура L является динамической и

может быть реализована в виде сбалансированного поискового дерева, выполняющего операции вставки и удаления за логарифмическое время. На рис. 2.12 изображены положения заметающей прямой и

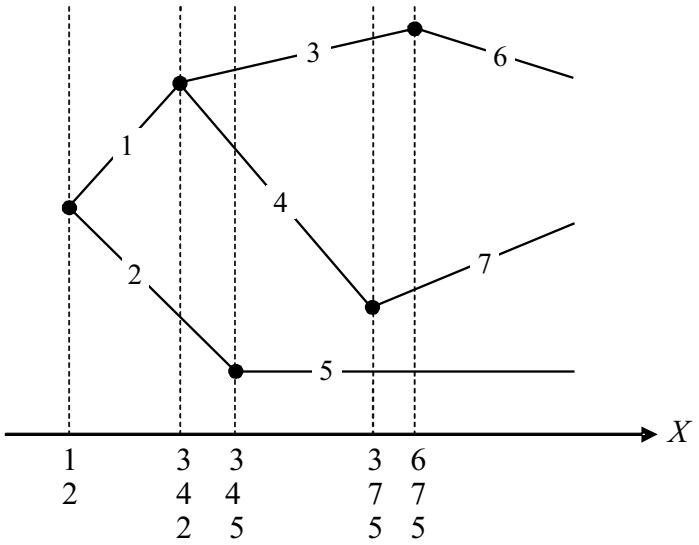


Рис. 2.12. Статус заметающей прямой изменяется только в критических точках

содержимое структуры L в каждой критической точке. При движении прямой слева направо в каждой новой критической точке происходит вставка в L новых ребер, лежащих правее l, и удаление из нее старых, лежащих левее l. Суммарное время, затраченное на вставку и удаление n ребер, не превосходит $O(n \log n)$, а на запоминание упорядоченного списка ребер внутри каждой полосы в сумме не превосходит $O(n^2)$. Получаем оценку $O(n^2)$. Ниже представлена схема алгоритма.

RetrieveStrips (G)
1 Упорядочить вершины G по возрастанию абсцисс
и поместить их в очередь E
2 **while** $E \neq \emptyset$ **do**
3 $p \leftarrow$ Pop(E)
4 Вставить в L ребра графа G, инцидентные p
и лежащие справа от p
5 Удалить из L ребра графа G, инцидентные p
и лежащие слева от p
6 Запомнить пересечение ребер в L с полосой справа от p.
7 **return** L

2.4.3. Метод детализации триангуляции

Существует ли метод со временем поиска $O(\log n)$ и использующий менее чем квадратичную память? Рассматриваемый ниже алгоритм принадлежит Киркпатрик (1983) и требует $O(n)$ памяти на предобработку и $O(\log n)$ на реализацию массового запроса.

Триангуляцией множества S из n точек называется максимальное по мощности множество непересекающихся отрезков с концами в S. Сделаем предположение, что планарное разбиение G является триангуляцией. Если планарное разбиение не является триангуляцией, то ее можно построить, добавляя недостающие ребра, за время $O(n \log n)$. Окружим триангуляцию G треугольной границей и протриангулируем область между границей и исходной триангуляцией (см. триангуляцию S_0 на рис. 2.13). (Докажите, что число ребер такой триангуляции равно $e = 3n - 6$, а число треугольников равно $f = 2n - 5$.)

Рассмотрим последовательность триангуляций $G = S_0, \ldots, S_{h(n)}$, где S_i получается из S_{i-1} по следующим правилам:

1. Из S_{i-1} удаляется некоторое подмножество несмежных вершин и инцидентные им ребра.
2. Триангулируются многоугольники, образованные в результате шага 1.

Таким образом, $S_{h(n)}$ – граничный треугольник. Рассмотрим ориентированный граф без циклов T, определяемый следующим образом:

1. Вершины графа соответствуют треугольникам триангуляций $S_0, \ldots, S_{h(n)}$. При этом считается, что $R \in S_i$, если он впервые получен в S_i при выполнении шага 2.
2. Дуга (R_k, R_j) существует тогда и только тогда, когда
   - R_j удаляется из S_{i-1} на шаге 1;
   - R_k создается в S_i на шаге 2;
   - $R_j \bigcap R_k \neq \emptyset$.

Для удобства узлы графа T изображают по строкам – каждой триангуляции S_i соответствует своя строка (рис. 2.13). Обозначим через $c(v)$ множество вершин-потомков вершины $v \in T$, а через $t(v)$ – треугольник, соответствующий вершине v. Тогда схема алгоритма локализации точки z выглядит следующим образом:

```
PointLocation(z, T)
1  v ← root[T]
2  if z ∉ t(v) then
3      return NIL
4  while c(v) ≠ ∅ do
5      for (по всем) u ∈ c(v) do
6          if z ∈ t(u) then
7              v ← u
8  return v
```

Проанализируем алгоритм. Его эффективность существенно зависит от стратегии выбора множества вершин для удаления при переходе от S_{i-1} к S_i. Эта стратегия состоит в выполнении двух условий. Обозначим через n_i число вершин в триангуляции S_i.

Условие 1. Существует число α такое, что $n_i = \alpha_i\, n_{i-1}$, $\alpha_i \le \alpha < 1$ для всех $i = 0, \ldots, h(n)$.

Условие 2. Существует число $H > 0$ такое, что любой треугольник $R_j \in S_i$ пересекает не более, чем H треугольников из S_{i-1} и наоборот.

Из условия 1 следует, что $h(n) \le \lfloor \log_{1/\alpha} n \rfloor = O(\log n)$. Действительно, числа n_i вершин триангуляций S_i ограничены сверху элементами следующей последовательности:

$$n = n_0, \alpha n_0, \alpha^2 n_0, \ldots, \alpha^{h(n)} n_0 = 1.$$

Кроме того, из обоих условий следует, что для хранения графа T требуется $O(n)$ памяти. Действительно, число узлов T равно числу всех треугольников в триангуляциях S_i, но $f_i = 2n_i - 5$, поэтому

$$\sum f_i < 2(n_0 + n_1 + \ldots + n_{h(n)}) \le 2n_0(1 + \alpha + \alpha^2 + \ldots + \alpha^{h(n)-1}) < \frac{2n}{1-\alpha}.$$

Оценим размер памяти, необходимой для хранения дуг графа. Из каждой вершины, согласно условию 2, выходит не более H дуг. Следовательно, всего их не более

$$\frac{2nH}{1-\alpha} = O(n).$$

Выполнение условий 1 и 2 гарантирует также, что время построения графа T не будет превосходить $O(n \log n)$, поскольку высота T равна $O(\log n)$, а время построения очередного уровня не превосходит $O(n)$.

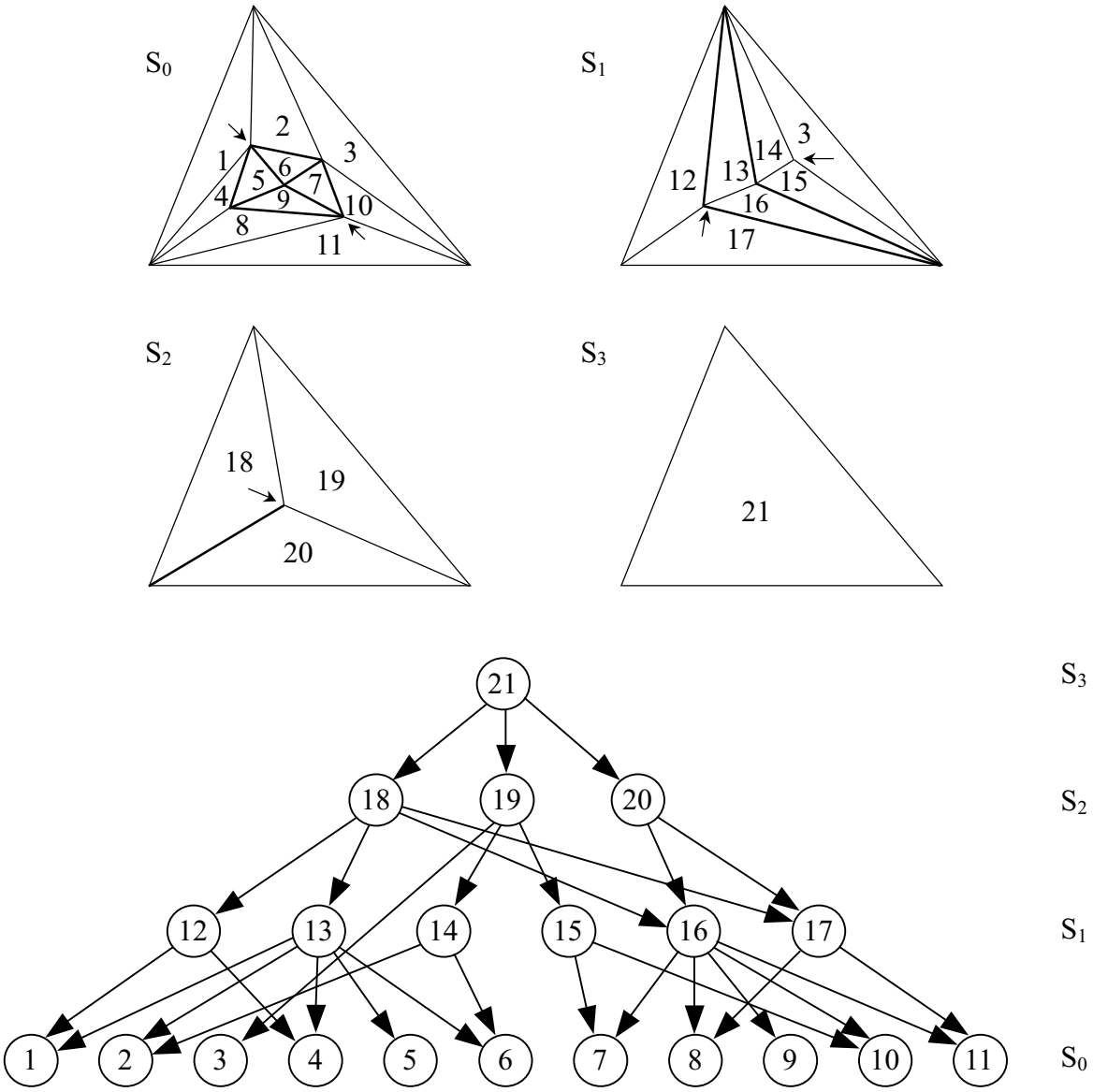


Рис. 2.13. Последовательность триангуляций G и соответствующий ей орграф T. Стрелками обозначены вершины, удаляемые из триангуляции S_{i-1} на шаге 1. Выделенные ребра добавлены в триангуляцию S_i на шаге 2

Нам остается ответить на вопрос: как выбирать вершины для удаления, чтобы обеспечить выполнение этих условий?

Критерий выбора следующий: удаляются несмежные вершины со степенью вершин меньше некоторой константы K. Легко доказать, что выполняется условие 2: если степень вершины v меньше K, то каждый из удаляемых треугольников пересекает не более $H = K - 2$ новых треугольников, поскольку триангуляция K-угольника содержит $K - 2$ треугольника.

Удаление вершин со степенью, меньшей K, гарантирует также выполнение условия 1. Для доказательства этого утверждения отметим два факта:

1. Всего ребер в триангуляции $3n - 6 < 3n$.
2. Каждое из ребер инцидентно ровно двум вершинам.

Из этих фактов следует, что сумма степеней вершин в триангуляции меньше $6n$. Действительно, пометим последовательно все ребра, при этом каждая пометка увеличивает сумму степеней вершин на 2, а всего ребер меньше $3n$. Значит, не менее $n/2$ вершин имеют степень меньше 12. Положим $K = 12$.

Пусть ν – число вершин, выбранных для удаления. Каждой из них инцидентно меньше 12 вершин, стало быть, при выборе из $n/2 - 3$ вершин мы сможем выбрать не менее $\frac{1}{12}\left(\frac{n}{2} - 3\right)$ вершин (3 вычитается за счет трех граничных вершин, которые не выбираются). Таким образом,

$$\nu \geq \left\lfloor \frac{1}{12}\left(\frac{n}{2} - 3\right) \right\rfloor,$$

откуда

$$n_i = n_{i-1} - \nu \leq n_{i-1} - \left\lfloor \frac{1}{12}\left(\frac{n_{i-1}}{2} - 3\right) \right\rfloor \approx n_{i-1}\left(1 - \frac{1}{24}\right),$$

то есть $\alpha \approx 1 - 1/24 < 0.959 < 1$, что гарантирует выполнение условия 1.

Таким образом, мы доказали, что локализацию точки на n-вершинном планарном разбиении можно осуществить за время $O(\log n)$ с использованием $O(n)$ памяти и $O(n \log n)$ времени предобработки.

Упражнения

2-1. Напишите однооконное графическое приложение с горизонтальной и вертикальной полосами прокрутки. По команде меню программа генерирует и отображает заданное количество точек, распределенных равномерно в прямоугольной области, превышающей по размеру клиентскую область окна примерно в 20 раз. С помощью прокрутки пользователь должен иметь

возможность просмотреть всю область, занимаемую точками. При этом перерисовка точек должна выполняться при каждом перемещении кнопки в полосе прокрутки. Для выбора точек, попавших в клиентскую область, используйте простое сравнение координат каждой точки с координатами клиентской области. Выполните тесты для 10^3, 10^4 и 10^5 и т. д. точек. Установите, при каком числе точек станет заметно замедление перерисовки при выполнении прокрутки.

2-2. Выполните предыдущее упражнение, используя для хранения точек

а) метод сетки;

б) квадратичное дерево;

в) 2-d-дерево.

Сравните производительность всех трех способов хранения.

2-3. Какие изменения необходимо внести в алгоритм поиска с помощью квадродерева, чтобы выполнять запросы на выборку множества отрезков и прямоугольников на плоскости.

2-4. Используя идею квадродерева, напишите программу, выполняющую геометрический поиск в пространстве с помощью *октодерева*, каждый узел которого соответствует параллелепипеду со сторонами, параллельными координатным осям. Найдите асимптотическую оценку наихудшего случая для октодерева.

2-5. С помощью тестов установите, как влияет значение M максимального числа точек в домене на скорость выборки.

2-6. Напишите программу, определяющую с помощью квадродерева принадлежность точки одному из множества непересекающихся ортогональных прямоугольников на плоскости.

2-7. Реализуйте метод детализации триангуляции. Сравните его производительность с методом из предыдущего упражнения.

Глава 3

Выпуклые оболочки на плоскости

В вычислительной геометрии понятие выпуклости занимает особое место. По сути, выпуклые фигуры являются аналогом упорядоченных множеств в общей теории алгоритмов. Если в условии задачи речь идет о выпуклых объектах, то для нее существует, как правило, простое и эффективное решение (см. разделы 2.3., 5.1. и главу 4). Между выпуклыми многоугольниками на плоскости и порядком на числовой прямой существует «линейная» связь: если задан последовательный обход вершин выпуклого многоугольника, то их абсциссы и ординаты могут быть упорядочены за время $O(n)$.

Построение выпуклой оболочки является своего рода модельной задачей, на примере которой могут быть продемонстрированы классические алгоритмы и приемы вычислительной геометрии. Нахождение выпуклой оболочки имеет также ряд важных практических применений, связанных с выделением границ объекта и распознаванием образов.

3.1. Определение и простой алгоритм построения

Выпуклой оболочкой множества точек S на плоскости, обозначается $\mathrm{conv}(S)$, называется наименьшая выпуклая область, содержащая все точки множества S (рис. 3.1). Нетрудно доказать, что $\mathrm{conv}(S)$ представляет собой выпуклый многоугольник, все вершины которого являются точками множества S.

Утверждение 1. *Точка выпуклого многоугольника P является его вершиной, если ее не содержит никакой отрезок с концами в P.*

Из данного утверждения, в частности, следует, что никакие три вершины $\mathrm{conv}(S)$ не лежат на одной прямой.

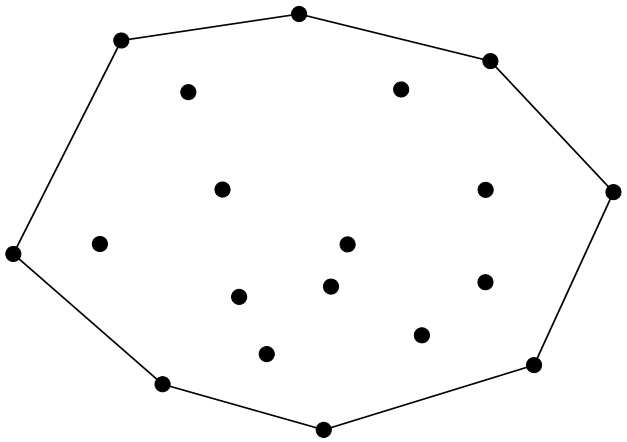


Рис. 3.1. Выпуклая оболочка множества точек

Задача 8 (Выпуклая оболочка). *Задано множество S из n точек на плоскости. Построить их выпуклую оболочку* $\mathrm{conv}(S)$.

Заметим, что требуется не просто перечислить вершины выпуклой оболочки, но также указать порядок их обхода. Для определенности будем искать обход вершин в направлении против часовой стрелки.

Теорема 1 (Нижняя оценка). *Для построения выпуклой оболочки n точек на плоскости требуется не менее $\Omega(n \log n)$ времени.*

Доказательство. Покажем, что к данной задаче за линейное время сводится задача сортировки с известной нижней оценкой $\Omega(n \log n)$. Пусть задана последовательность из n положительных действительных чисел $x_1, \ldots, x_n$. Чтобы их упорядочить, поставим в соответствие каждому числу x_i точку p_i с координатами (x_i, x_i^2) и найдем выпуклую оболочку полученного множества точек (рис. 3.2). Точки p_i лежат на параболе $y = x^2$, поэтому последовательный

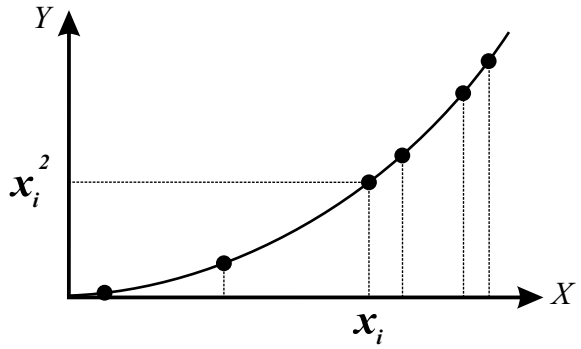


Рис. 3.2. К доказательству теоремы 1

обход вершин выпуклой оболочки этих точек даст упорядоченный список их абсцисс. В итоге на сортировку n чисел мы потратили $O(n)$ операций плюс число действий, необходимых для построения выпуклой оболочки. Поскольку сортировка не может быть выполнена быстрее, чем за $\Omega(n \log n)$, заключаем, что такая нижняя оценка справедлива также для задачи 8. ∎

Простой метод построения выпуклой оболочки основан на следующем утверждении.

Утверждение 2. *Точки $p_1, p_2 \in S$ являются вершинами* $\mathrm{conv}(S)$ *тогда и только тогда, когда все точки S, отличные от p_1 и p_2, либо лежат по одну сторону от прямой (p_1, p_2), либо являются внутренними точками отрезка $\overline{p_1, p_2}$.*

Таким образом, перебрав $O(n^2)$ всевозможных пар точек множества S и выполнив за линейное время проверку условия для каждой пары, можно найти все ребра $\mathrm{conv}(S)$. Чтобы построить выпуклую оболочку, необходимо затем упорядочить концы ребер по возрастанию полярного угла относительно точки S с минимальной абсциссой. Если таких точек несколько, то выбирается точка с минимальной ординатой. Получаем алгоритм с трудоемкостью $O(n^3)$. Разумеется, такая оценка нас не устраивает, поэтому займемся поиском более эффективного алгоритма.

3.2. Препроцессор

В предыдущих главах мы использовали препроцессорную обработку для ускорения выполнения массовых запросов. В данном случае процедура, также называемая препроцессором, зачастую позволяет существенно сократить размерность задачи и тем самым ускорить выполнение одного уникального запроса.

Точка $p \in S$ называется *крайней*, если существует прямая l, содержащая p, такая, что все точки множества S, кроме p, лежат в одной из двух полуплоскостей, задаваемых l. Прямая l называется *опорной* прямой множества S.

Утверждение 3. *Точка $p \in S$ является крайней тогда и только тогда, когда $p \in \mathrm{conv}(S)$.*

Утверждение 4. *Точка $p \in S$ не является крайней, если она принадлежит выпуклой оболочке произвольного подмножества крайних точек множества S.*

Последние два утверждения позволяют построить фильтр, с помощью которого можно исключить из множества S точки, заведомо не являющиеся крайними и, стало быть, вершинами выпуклой оболочки.

Алгоритм (препроцессор):

1. Найти точки множества S с минимальной и максимальной абсциссой и ординатой. Пусть это будут точки p_l, p_r, p_b и p_t (рис. 3.3).

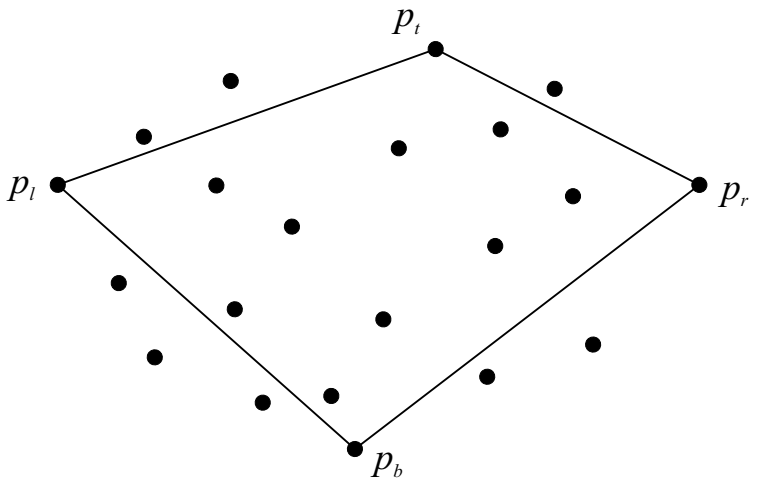


Рис. 3.3. Точки множества S, находящиеся внутри выпуклой оболочки четырех крайних точек S, не могут являться вершинами выпуклой оболочки S

2. Исключить из S точки, находящиеся внутри четырехугольника с вершинами в этих точках, поскольку они не являются крайними в силу утверждения 4.

Очевидно, препроцессор имеет линейную трудоемкость.

3.3. Алгоритм Джарвиса

В 1973 году Джарвис (Jarvis) заметил, что «простой» алгоритм из раздела 3.1. можно значительно ускорить, если более аккуратно выполнять перебор пар точек множества S. Действительно, пусть найдена вершина p_1 выпуклой оболочки (рис. 3.4). Выберем p_1 в качестве центра полярной системы координат. Джарвис предложил в качестве следующей вершины выпуклой оболочки выбирать точку S

с наименьшим полярным углом относительно p_1. Добавление новых вершин напоминает «заворачивание» множества S в ребра выпуклой оболочки, поэтому этот процесс называют также *заворачиванием подарка* (*gift wrapping*).

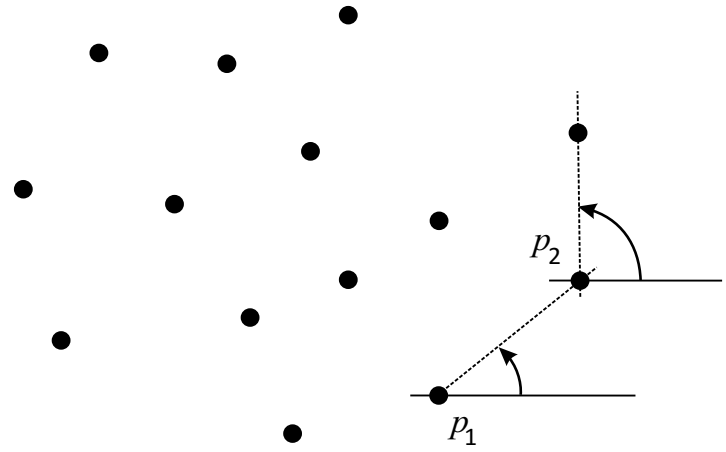


Рис. 3.4. Алгоритм Джарвиса

Рассмотрим вариант алгоритма Джарвиса, позволяющий избежать ресурсоемкого вычисления полярного угла. Поиск начнем с точки p_l, имеющей минимальную абсциссу. Если таких точек несколько, то выбираем из них точку с минимальной ординатой. Для каждой точки p_i вычислим тангенс полярного угла α_i относительно p_l:

$$\tan(p_i, p_l) = \tan \alpha_i = \frac{y_i - y_l}{x_i - x_l}. \tag{3.1}$$

В качестве очередной вершины выпуклой оболочки выберем точку p^* с минимальным значением тангенса. Здесь мы используем тот факт, что на интервале $\left(-\frac{\pi}{2}, \frac{\pi}{2}\right)$ тангенс строго возрастает и поэтому может использоваться для сравнения полярных углов точек, лежащих справа от начала координат. Если точек с минимальным значением тангенса окажется несколько, то из них выбирается наиболее удаленная от p_l.

Выбрав точку p^* в качестве нового начала полярной системы координат, ищем следующую вершину $\mathrm{conv}(S)$ среди точек S, лежащих справа от p^* и имеющих минимальный тангенс угла наклона, и т. д., пока не достигнем точки p_r с максимальной абсциссой. После этого ищем слева от p_r вершину, имеющую минимальный тангенс полярного угла относительно p_r, и т. д., пока не достигнем точки p_l.

```
JarvisHull(S)
 1  Найти точки p_l и p_r
 2  p_0 ← p_l
 3  repeat
 4      Найти точку p* такую, что
 5          tan(p*, p_0) = min {tan(p, p_0)| x_p > x_{p_0}, p ∈ S}
 6      Push(conv, p*)
 7      p_0 ← p*
 8  until p_0 = p_r
 9  repeat
10      Найти точку p* такую, что
11          tan(p*, p_0) = min {tan(p, p_0)| x_p < x_{p_0}, p ∈ S}
12      Push(conv, p*)
13      p_0 ← p*
14  until p_0 = p_l
```

Если обозначить через h число вершин выпуклой оболочки, то трудоемкость алгоритма Джарвиса можно оценить как $O(nh)$. Таким образом, несмотря на то что нам удалось сократить перебор, время работы алгоритма в наихудшем случае равно $O(n^2)$. При этом интересен анализ работы алгоритма в среднем, который основан на оценке величины $M(n)$ математического ожидания числа вершин выпуклой оболочки множества из n точек на плоскости. Существуют распределения точек, для которых $M(n) = o(\log n)$, поэтому для таких задач алгоритм Джарвиса работает асимптотически быстрее, чем алгоритмы с оптимальной оценкой $\Theta(n \log n)$. Следующая таблица содержит значения $M(n)$ для различных распределений и средние оценки времени работы алгоритма Джарвиса для этих распределений.

Распределение	$M(n)$	Оценка в среднем
Равномерное в круге	$O(\sqrt[3]{n})$	$O(n^{\frac{4}{3}})$
Нормальное на плоскости	$O(\sqrt{\log n})$	$O(n\sqrt{\log n})$
Равномерное в k-угольнике	$O(\log n)$	$O(n \log n)$

3.4. «Быстрая» выпуклая оболочка

Обозначим через p_l и p_r, соответственно, крайнюю левую и крайнюю правую точки множества S. Последовательность вершин выпуклой

оболочки, лежащих выше прямой (p_l, p_r), будем называть *верхней* выпуклой оболочкой S. Аналогичным образом определяется *нижняя* выпуклая оболочка.

В основе так называемого «быстрого» метода лежит идея, суть которой состоит в последовательном рекурсивном применении алгоритма препроцессора. Вершины вычисляются отдельно для верхней и нижней выпуклой оболочки, а затем объединяются в общую последовательность.

Верхняя выпуклая оболочка строится для подмножества точек S, лежащих выше направленной прямой (p_l, p_r). В качестве первых двух крайних точек этого подмножества выбираются p_l и p_r. Третью точку необходимо выбрать таким образом, чтобы гарантированно существовала опорная прямая, относительно которой все точки S располагались бы по одну сторону. Таким свойством обладает точка p_h, максимально удаленная от прямой (p_l, p_r) (для простоты изложения будем считать, что такая точка единственная). Прямая, параллельная (p_l, p_r) и проходящая через p_h, является опорной для множества S по определению (рис. 3.5).

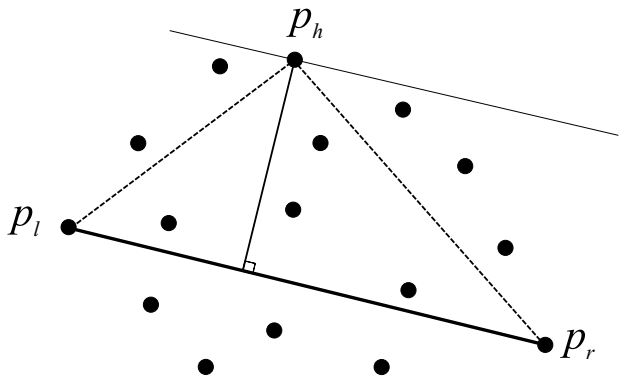


Рис. 3.5. «Быстрый» метод построения выпуклой оболочки

Заметим, что выбрать третью крайнюю точку можно другим способом, например, взять точку с максимальной ординатой. Однако описанный метод имеет одно существенное преимущество: «в среднем» он позволяет исключить из дальнейшего рассмотрения максимальное число точек множества S. Действительно, согласно утверждению 4, внутренние точки треугольника $\triangle(p_l, p_h, p_r)$ не являются вершинами выпуклой оболочки и могут далее не рассматриваться. Но поскольку из всех треугольников с основанием $\overline{p_l, p_r}$ треугольник $\triangle(p_l, p_h, p_r)$ имеет максимальную площадь, мы в праве ожидать, что в среднем внутренних точек также окажется максимальное количество.

После того как точка p_h найдена, рекурсивно строятся выпуклые оболочки слева от $\overrightarrow{p_l, p_h}$ и $\overrightarrow{p_h, p_r}$. Аналогично строится нижняя выпуклая оболочка S.

Обозначим символом «+» операцию сцепления двух списков точек.

LeftConvexHull(S, p_l, p_r)
1 if $S = \{p_l, p_r\}$
2 **return** S
3 $p_h \leftarrow$ точка S, лежащая на максимальном расстоянии от (p_l, p_r)
4 $S_l \leftarrow$ точки S, расположенные слева от (p_l, p_h)
5 $S_r \leftarrow$ точки S, расположенные справа от (p_h, p_r)
6 **return** LeftConvexHull(S_l, p_l, p_h) + LeftConvexHull(S_r, p_h, p_r)

QuickHull(S, p_l, p_r)
1 Найти точки p_l и p_r с минимальной и максимальной абсциссой
2 $S_l \leftarrow$ точки S, расположенные слева от (p_l, p_r)
3 $S_r \leftarrow$ точки S, расположенные справа от (p_l, p_r)
4 **return** LeftConvexHull(S_l, p_l, p_r) + LeftConvexHull(S_r, p_r, p_l)

Оценим трудоемкость алгоритма. Обозначим через $T(|S|)$ время работы функции LeftConvexHull(S, p_l, p_r). Имеет место соотношение

$$T(|S|) = T(|S_l|) + T(|S_r|) + O(n).$$

Если выполняется неравенство

$$\max\{T(|S_l|), T(|S_r|)\} \leq \left\lceil \frac{T(|S|)}{2} \right\rceil,$$

то получаем оценку $T(|S|) = O(n \log n)$. Однако нетрудно найти пример, когда на каждой итерации удается сократить размерность задачи лишь на константу. Расположение точек на рис. 3.6 гарантирует, что при каждом рекурсивном вызове функции LeftConvexHull множество S_l будет пустым, а множество S_r будет содержать на одну точку меньше, чем S. Таким образом, имеем оценку наихудшего случая $O(n^2)$.

Тем не менее по сравнению с другими алгоритмами построения выпуклой оболочки описанный алгоритм демонстрирует на практике невероятную эффективность подобно тому, как алгоритм «быстрой» сортировки доминирует над алгоритмами сортировки, имеющими лучшие асимптотические оценки.

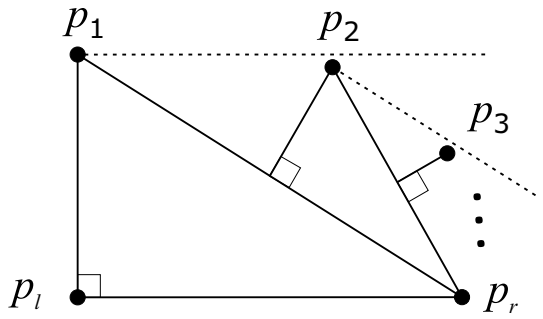


Рис. 3.6. Пример расположения точек множества S, для которого алгоритм QuickHull выполняет $O(n^2)$ шагов

3.5. Алгоритм Грехэма

Несмотря на существенные продвижения в поиске эффективных алгоритмов, мы не ответили на один важный вопрос: существует ли алгоритм с оптимальным быстродействием $\Theta(n \log n)$ при *любом* расположении точек?

Такой алгоритм существует, и не один. В 1972 году Грехэм (Graham) предложил алгоритм, который строит выпуклую оболочку точек за линейное время в предположении, что они упорядочены по полярному углу относительно произвольной внутренней точки. С тех пор, так называемый «обход Грехэма» используется при решении многих задач и входит в число классических методов вычислительной геометрии наряду с методом «локусов» и алгороитмом «заметающей прямой».

Независимо от способа упорядочения точек схема обхода Грехэма в целом не меняется (рис. 3.7):

1. Первые три точки упорядоченного списка помещаются в стек вершин выпуклой оболочки.

2. Следующая точка также помещается в стек вершин, но перед вставкой проверяется ее ориентация относительно двух предыдущих вершин. Если ориентация «неправильная» (на рисунке точки образуют *правый* поворот), то из стека выталкивается последняя помещенная в него вершина, и проверка повторяется нужное число раз.

3. Шаг 2 повторяется до тех пор, пока не будет достигнут конец упорядоченного списка точек.

Как и в алгоритме Джарвиса, мы будем сортировать точки S не по полярному углу, а по тангенсу полярного угла относительно крайней

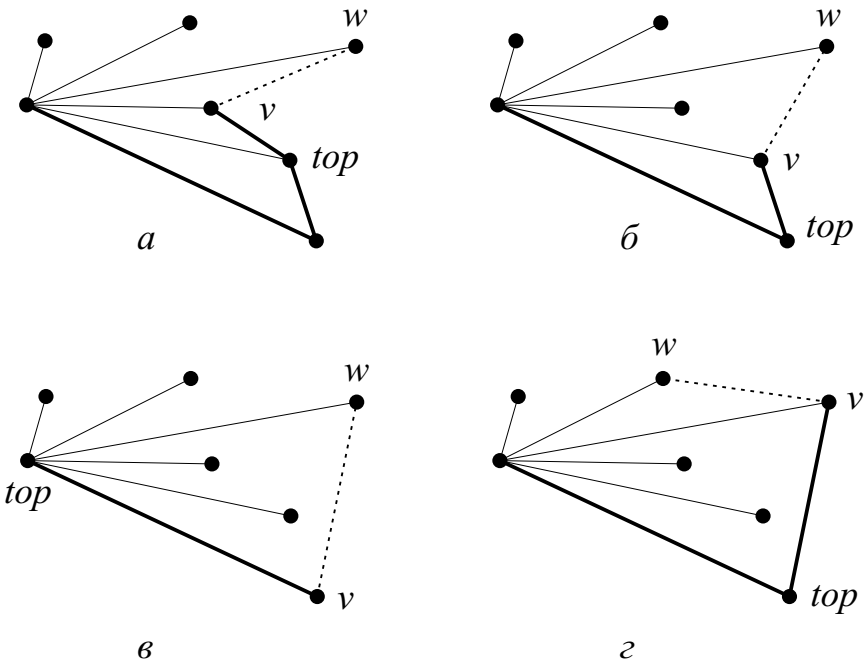


Рис. 3.7. Обход Грехэма: пройдены первые четыре точки (*а*), обнаружен правый поворот, из стека выталкивается вершина выпуклой оболочки уже пройденных точек (*б*, *в*), левый поворот – переход к следующей точке (*г*)

левой точки p_l (формула (3.1)). Найденные вершины выпуклой оболочки будем заносить в стек conv.

GrahamHull (S)
1 Найти точку p_l с минимальной абсциссой
2 Упорядочить точки S по возрастанию $\tan(p_i, p_l)$
и добавить их в очередь S_0
3 Push(conv, p_l)
4 $v \leftarrow$ Pop(S_0)
5 $w \leftarrow$ Pop(S_0)
6 **while** $w \neq p_l$ **do**
7 **if** (top[conv], v, w) образуют правый поворот **then**
8 $v \leftarrow$ Pop(conv)
9 **else**
10 Push(conv, v)
11 $v \leftarrow w$
12 $w \leftarrow$ Pop(S_0)
13 **return** conv

Следующее утверждение представляет собой важный теоретический результат, поэтому мы сформулируем его в виде теоремы.

Теорема 2. *Выпуклая оболочка n точек на плоскости может быть найдена за оптимальное время $\Theta(n \log n)$.*

Доказательство. Сортировка точек требует $O(n \log n)$ времени. Внутренняя часть цикла **while** включает константное число действий, и при этом на каждой итерации по крайней мере одна точка выбывает из рассмотрения. Таким образом, сложность обхода Грехэма $O(n)$. ∎

3.6. Алгоритм типа «разделяй и властвуй»

Несмотря на то что алгоритм Грехэма является оптимальным по быстродействию, существуют серьезные основания для поиска лучших решений, в частности:

1. Данный подход не может быть обобщен на пространства большей размерности.
2. Алгоритм нельзя распараллелить, то есть разбить задачу на подзадачи меньшего размера, чтобы воспользоваться возможностями компьютера, допускающего параллельные вычисления.
3. Алгоритм является *закрытым* (то есть все исходные данные задачи должны быть известны *до* начала обработки).

От первых двух недостатков можно избавиться, если для построения выпуклой оболочки применить стандартную схему «разделяй и властвуй»:

1. Если $|S| \leq 3$, то вернуть S.
2. Разбить S на два примерно равных подмножества S_1 и S_2.
3. Найти рекурсивно выпуклые оболочки $\mathrm{conv}(S_1)$ и $\mathrm{conv}(S_2)$.
4. Объединить $\mathrm{conv}(S_1)$ и $\mathrm{conv}(S_2)$ в выпуклую оболочку исходного множества $\mathrm{conv}(S)$.
5. Вернуть $\mathrm{conv}(S)$.

Объединение выпуклых оболочек на шаге 4 за линейное время даст общую трудоемкость $O(n \log n)$. Таким образом, построение выпуклой оболочки сводится к следующей задаче:

Задача 9. *Даны два выпуклых многоугольника. Найти их выпуклую оболочку.*

Следующий алгоритм решает эту задачу за линейное время.

ConvexPolygonsHull (P_1, P_2)

1 Найти внутреннюю вершину p многоугольника P_1
2 **if** $p \notin P_2$ **then**
3 Найти опорные прямые l и r к многоугольнику P_2, проходящие через p (рис. 3.8)
4 Удалить из рассмотрения более близкую к p цепочку вершин P_2, лежащих между l и r
5 За время $O(n)$ слить вершины многоугольников P_1 и P_2 в один список, отсортированный по полярному углу относительно точки p
6 Применить к упорядоченному списку обход Грехэма

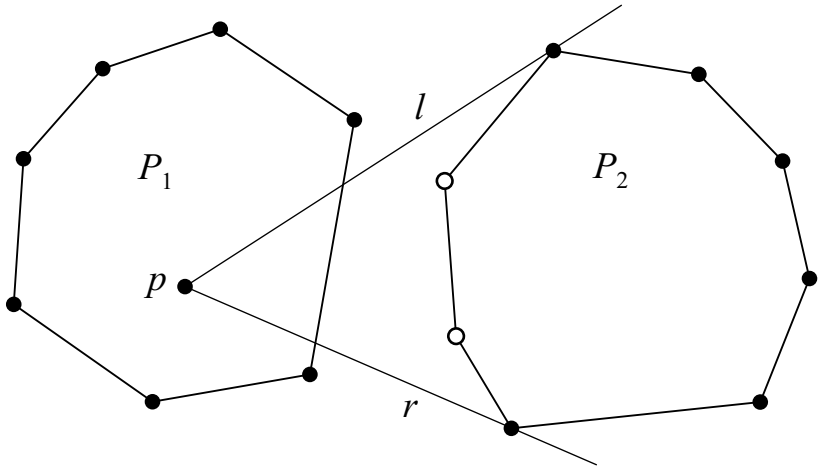


Рис. 3.8. Удаление цепочки вершин P_2 между l и r позволяет упорядочить вершины P_1 и оставшиеся вершины P_2 по полярному углу относительно точки p

Таким образом, если $|P_1| = n$, а $|P_2| = m$, то построение выпуклой оболочки двух выпуклых многоугольников занимает не более $O(n+m)$ времени.

Теорема 3. *Алгоритм типа «разделяй и властвуй» строит выпуклую оболочку n точек на плоскости за оптимальное время* $\Theta(n \log n)$.

3.7. Динамическое построение выпуклой оболочки

Усложним задачу. Предположим, что точки множества S не заданы все сразу, а поступают на вход алгоритма последовательно одна за другой. В этом случае нужно научиться достраивать (поддерживать) выпуклую оболочку по мере поступления новых точек. Напомним, что *закрытым* (*offline*) называется алгоритм, решающий задачу только при наличии всего набора исходных данных. Аналогично, алгоритм, обрабатывающий данные по мере их поступления, называется *открытым* (*online*).

Задача 10. *Задано множество* $S = \{p_1, \ldots, p_n\}$ *точек на плоскости. Построить выпуклые оболочки объединений множеств* $\mathrm{conv}(\{p_1, \ldots, p_k\}) \cup \{p_{k+1}\}$ *для* $k = 1, \ldots, n-1$.

Заметим, что выпуклую оболочку всех точек S можно построить с помощью online-алгоритма, решающего задачу 10. Как известно, нижняя оценка для задачи в такой постановке равна $\Omega(n \log n)$, поэтому для выпуклой оболочки объединения $\mathrm{conv}(\{p_1, \ldots, p_k\}) \cup \{p_{k+1}\}$ имеем нижнюю оценку $\Omega(\log k)$. Итак, задачу 10 можно переформулировать следующим образом.

Задача 11. *Даны выпуклый многоугольник* P *и точка* p. *За время* $O(\log n)$ *построить выпуклую оболочку множества* $P \cup \{p\}$.

Рассмотрим сначала неоптимальный алгоритм, а затем попытаемся его улучшить.

Алгоритм:

1. За время $O(n)$ упорядочить вершины многоугольника P по полярному углу относительно произвольной внутренней точки.
2. За время $O(\log n)$ найти вершины p_i и p_{i+1} многоугольника P, определяющие сектор, содержащий точку p.
3. Вставить p в упорядоченную последовательность вершин между p_i и p_{i+1}.
4. За время $O(n)$ выполнить обход Грехэма и построить выпуклую оболочку множества $P \cup \{p\}$.

Сложность данного метода $O(n)$, поэтому для исходной задачи 10 получаем верхнюю оценку $O(n^2)$.

Метод опорных прямых

Рассмотрим другой алгоритм, основанный на быстром поиске опорных прямых к многоугольнику P.

Алгоритм:

1. За время $O(\log n)$ локализовать точку p относительно многоугольника P (см. раздел 2.3).
2. Если $p \in P$, то алгоритм прекращает работу и возвращает в качестве искомого решения последовательность вершин P.
3. Найти две опорные прямые l и r, проходящие через точку p и вершины $v_l, v_r \in P$ (напомним, что прямая называется *опорной* относительно многоугольника P, если все точки P, кроме одной, лежат по одну сторону от этой прямой).
4. Удалить цепочку вершин P между v_l и v_r и вставить на ее место точку p (рис. 3.9).

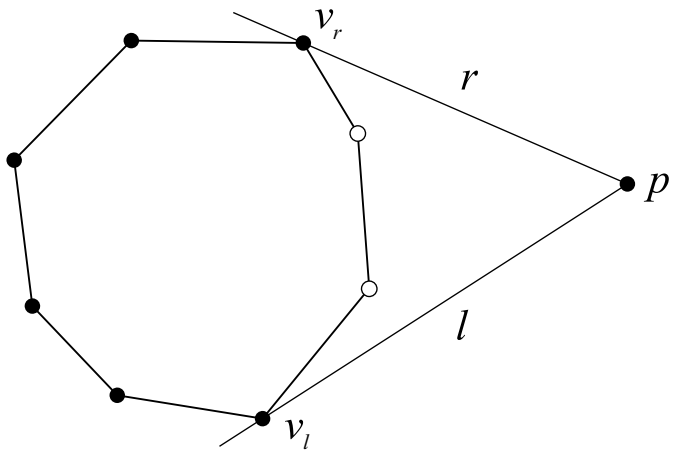


Рис. 3.9. Метод опорных прямых

Очевидно, что при таком подходе наибольшую трудность представляет поиск опорных прямых за логарифмическое время (поиск за линейное время тривиален). Будем считать, что вершины P упорядочены в направлении по часовой стрелке. Интуитивно понятно, что поскольку многоугольник P выпуклый и его вершины выстроены в упорядоченную последовательность, то должна существовать возможность организовать внутри этой последовательности быстрый поиск, подобный двоичному поиску в упорядоченном массиве.

Рассмотрим несколько определений. Опорную прямую будем называть *левой* или *правой* в зависимости от того, с какой стороны от нее лежит многоугольник P. Вершины v_l и v_r, через которые проходят опорные прямые, называются, соответственно, *левой* и *правой* опорными вершинами (рис. 3.10, *а*). Вершина v называется *вогнутой* относительно точки p, если открытый отрезок $\overline{p, v}$ имеет пересечение с P (рис. 3.10, *б*). Вершина, которая не является опорной или вогнутой, называется *выпуклой* (рис. 3.10, *в*).

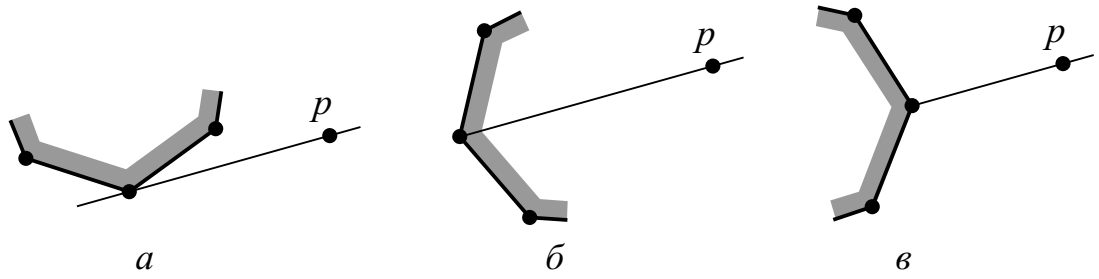


Рис. 3.10. Опорная (*а*), вогнутая (*б*) и выпуклая (*в*) вершины

Анализ схемы алгоритма показывает, что для хранения вершин P требуется динамическая структура данных, выполняющая за логарифмическое время поиск в упорядоченной последовательности, а также разбиение упорядоченной последовательности на две подпоследовательности и объединение двух последовательностей. Кроме того, нужна вставка нового элемента.

Структура данных с указанными свойствами хорошо известна: это так называемая *сцепляемая очередь* (см. раздел 1.6), которая может быть реализована в виде 2-3-*дерева* (*B-дерева* 1-го порядка) [20, 22]. 2-3-дерево моделирует упорядоченную цепочку элементов, значения которых хранятся в листьях дерева. Все листья расположены на одной высоте, а каждая внутренняя вершина t имеет две или три дочерние вершины, которые мы обозначим через t_L, t_M и t_R. Таким образом, высота H дерева находится в пределах $O(\log_3 n) \leq H \leq O(\log_2 n)$, где n – длина цепочки. Для обеспечения доступа к листьям за логарифмическое время внутренняя вершина t хранит также значения $l[t]$, $m[t]$ и $r[t]$ минимальных элементов своих дочерних вершин.

В нашей задаче листья 2-3-дерева хранят цепочку вершин многоугольника P в порядке их обхода по часовой стрелке (рис. 3.11). Внутренняя вершина t дерева содержит ссылки $l[t]$, $m[t]$ и $r[t]$ на начальные элементы цепочек для левого, среднего и правого поддерева

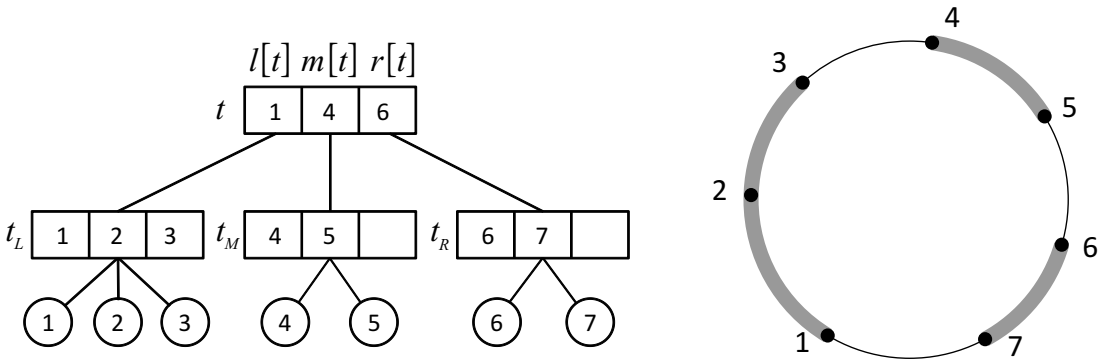


Рис. 3.11. 2-3-дерево и соответствующая цепочка вершин выпуклого многоугольника

(ссылка $r[t]$ может отсутствовать). Таким образом, в зависимости от того, сколько дочерних вершин содержит t, соответствующая цепочка вершин многоугольника условно разбивается на две или три примерно равные части, разделенные вершинами $l[t]$, $m[t]$ и $r[t]$. Зная типы этих граничных вершин относительно точки p, можно определить, в каком из поддеревьев находится искомая опорная вершина.

Рассмотрим этот шаг более подробно. Простым перебором вариантов нетрудно убедиться, что существует всего восемь различных комбинаций типов граничных вершин цепочки и их взаимного расположения относительно точки p (рис. 3.12). Каждая комбинация однозначно определяется типом вершин $l[t]$, $m[t]$ и углом $\alpha = \angle l[t], p, m[t]$.

Если установлено, что для цепочки имеет место комбинация 2 или 4, то ни левая, ни правая опорные вершины ей определенно не принадлежат. Комбинации 1 и 3 означают, что цепочка содержит обе опорные вершины, а (5, 8) и (6, 7) – что данная цепочка содержит, соответственно, левую или правую опорную вершину.

Теперь мы можем формально описать алгоритм. Он представляет собой рекурсивную функцию, которая принимает на вход внутреннюю вершину дерева и последовательно анализирует комбинации для дочерних вершин. Ниже приведена схема работы функции FindLeft для поиска левой опорной вершины. Для дочерней вершины, цепочка которой соответствует комбинации 1, 3, 5 или 8 (проверка осуществляется с помошью функции ContainsLeft), выполняется рекурсивный вызов. Особой обработки требует ситуация, когда отсутствует поддерево t_R: в этом случае в качестве правой границы цепочки t_M рассматривается самая правая ее вершина. Отдельно рассматривается также замкнутая цепочка, соответствующая корневой вершине.

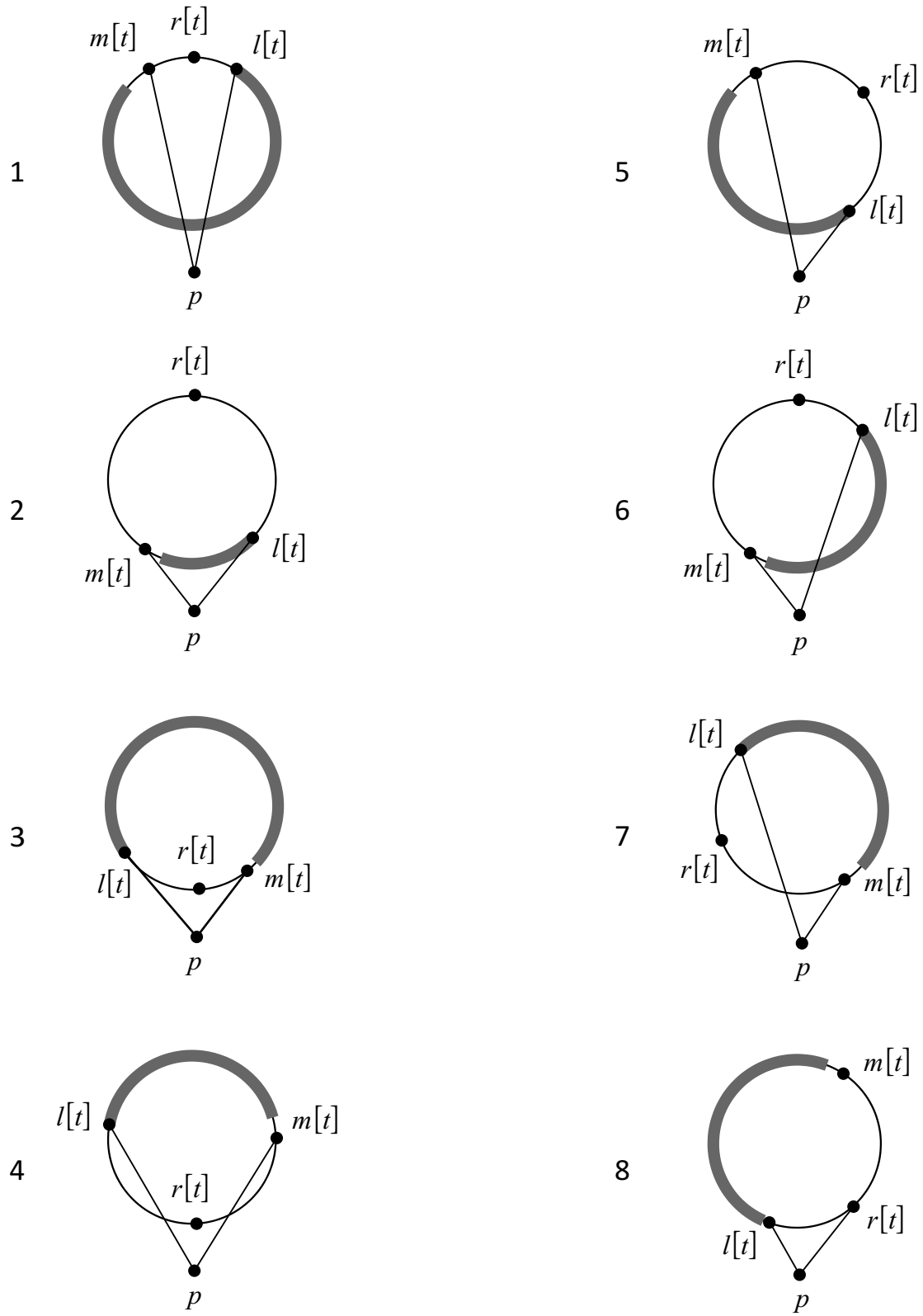


Рис. 3.12. Восемь комбинаций взаимного расположения граничных вершин цепочки и точки p

```
FindLeft(t, p)
 1  l ← l[t]
 2  m ← m[t]
 3  if t_R ≠ nil then
 4      r ← r[t]
 5  else
 6      r ← самая правая вершина в t_M
 7  if l, m или r – левая опорная вершина then
 8      return l, m или r
 9  if ContainsLeft(l, m, p) then
10      return FindLeft(t_L, p)
11  if ContainsLeft(m, r, p) then
12      return FindLeft(t_M, p)
13  if t_R ≠ nil then
14      if t определяет замкнутую цепочку
15          if ContainsLeft(r, l, p) then
16              return FindLeft(t_R, p)
17      else
18          c ← самая правая вершина в t_R
19          if ContainsLeft(r, c, p) then
20              return FindLeft(t_R, p)
```

Заметим, что глубина рекурсии не превышает $O(\log n)$, а функция ContainsLeft вычисляет номер комбинации за константное время. Таким образом, справедлива следующая теорема.

Теорема 4. *Выпуклая оболочка n точек на плоскости может быть построена с помощью открытого алгоритма за оптимальное время* $\theta(n \log n)$.

3.8. Аппроксимация выпуклой оболочки

Можно построить выпуклую оболочку быстрее, чем за $O(n \log n)$, если поставить цель получить не точное решение, а некоторую его аппроксимацию. Алгоритмы аппроксимации строят выпуклую оболочку некоторого подмножества S' исходного множества точек S, которая при возрастании размерности задачи приближается к реальной выпуклой оболочке. При этом за счет потери точности достигается выигрыш не только в скорости, но и в простоте алгоритма.

Алгоритм:

1. Найти минимальную и максимальную абсциссы $x_{\min}$ и $x_{\max}$ точек исходного множества S.
2. Разбить вертикальную полосу между ними на k полос *равной* ширины (рис. 3.13).
3. Внутри каждой полосы найти точки с минимальной и максимальной ординатой (*экстремальные* точки).
4. Для полученного подмножества экстремальных точек построить выпуклую оболочку с помощью обхода Грехэма.

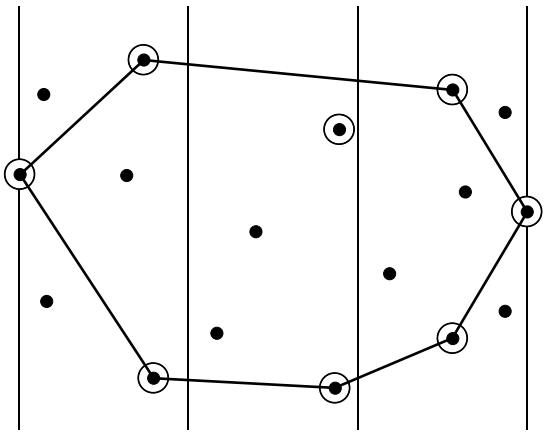


Рис. 3.13. Аппроксимация выпуклой оболочки с возможной потерей точек вблизи границы

Оценим трудоемкость алгоритма. На шаге 3 просматриваются все n точек, и для каждой из них определяется индекс содержащей ее полосы. Поскольку полосы имеют равную ширину, этот индекс может быть вычислен за константное время. Стало быть, если для каждой полосы хранить значение ординат экстремальных точек, то за один проход по всем точкам S эти значения могут быть вычислены за время $O(n)$. На шаге 4 выполняется обход Грехэма, сложность которого $O(k)$. Таким образом, общая трудоемкость алгоритма $O(n + k)$.

Осталось выяснить, действительно ли имеет место аппроксимация.

Теорема 5. *Любая точка $p \in S$, не принадлежащая приближенной выпуклой оболочке, находится от нее на расстоянии не более* $(x_{\max} - x_{\min})/k$.

Действительно, любая точка множества S, лежащая за пределами приближенной выпуклой оболочки, находится внутри прямоугольной области, ограниченной одной из k полос и ординатами экстремальных точек этой полосы. Если участок границы приближенной выпуклой оболочки, пересекающий данную полосу, также находится внутри этого прямоугольника (рис. 3.14), то расстояние от любой внешней точки до этого участка не может превышать ширину полосы.

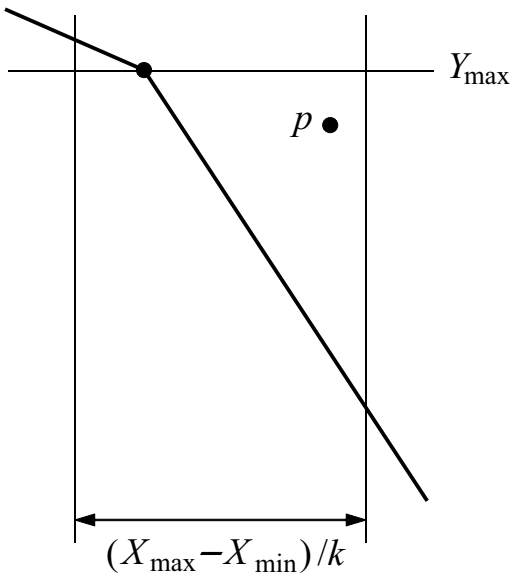


Рис. 3.14. Точка, не принадлежащая приближенной оболочке, отстоит от границы оболочки на расстоянии, не превышающем ширину полосы

Можно построить аппроксимацию другого рода – выпуклый многоугольник, заведомо содержащий все точки исходного множества. Для этого экстремальные точки в каждой полосе заменяются парами точек на границах полосы с той же ординатой, которую имеет экстремальная точка (рис. 3.15), и затем вычисляется выпуклая оболочка этих точек методом Грехэма. Для точности аппроксимации справедлива та же оценка.

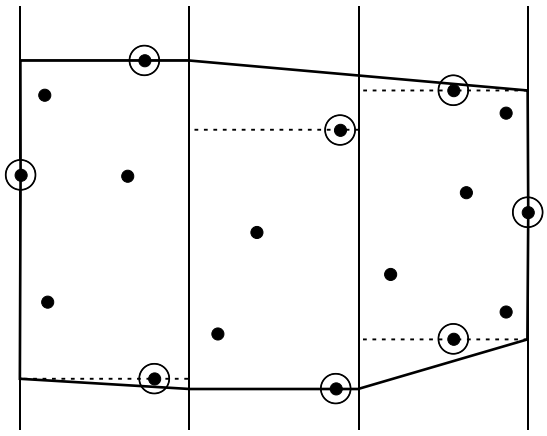


Рис. 3.15. Аппроксимация выпуклой оболочки, при которой все точки попадают в построенную область

3.9. Выпуклая оболочка простого многоугольника

Обратим внимание на следующий факт: при заданном упорядочении точек множества S по одной из координат или по полярному углу относительно произвольного центра выпуклая оболочка S может быть построена с помощью обхода Грехэма за линейное время. Возникает вопрос: является ли упорядочение по одной из координат необходимым условием существования линейного алгоритма? Оказывается, нет. Как будет показано ниже, если известен циклический обход точек без самопересечений, то построение выпуклой оболочки также возможно за время $O(n)$.

Напомним, что многоугольник называется *простым*, если никакая пара непоследовательных его ребер не имеет общих точек. Следующий алгоритм предложен Lee в 1983 году. Пусть внутренность многоугольника P расположена справа при обходе вершин в порядке возрастания их номеров (рис. 3.16). Обозначим через p_l и p_r, соответственно,

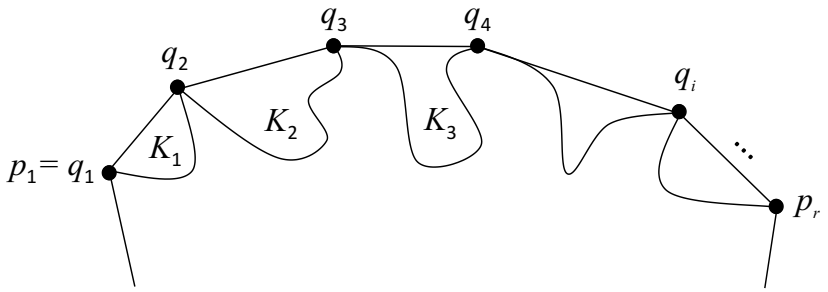


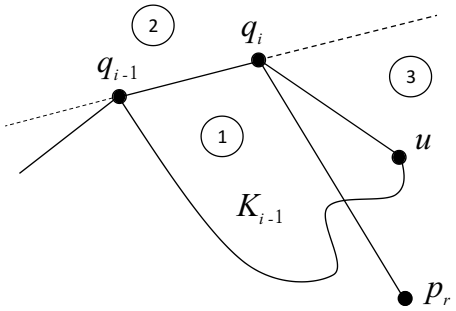
Рис. 3.16. Выпуклая оболочка простого многоугольника: экстремальные точки и карманы

крайнюю левую и крайнюю правую точку множества S. Они разбивают последовательность вершин P на две цепи: от p_l до p_r и от p_r до p_l. Будем строить раздельно верхнюю выпуклую оболочку (при движении от p_l до p_r) и нижнюю выпуклую оболочку (при движении от p_r до p_l). Для последовательности точек $p_1, \ldots, p_k$ обозначим через $q_1, \ldots, q_m$ вершины выпуклой оболочки этих точек. Точки $q_1, \ldots, q_m$ назовем *экстремальными*, подпоследовательность вершин $p_1, \ldots, p_k$ между точками q_i и q_{i+1} назовем *карманом* K_i, а ребро $\overline{q_i, q_{i+1}}$ – *крышкой* кармана. Идея алгоритма состоит в последовательном продвижении вдоль точек $p_1, \ldots, p_k$ и обнаружении экстремальных точек q_i. Точки q будем хранить в стеке.

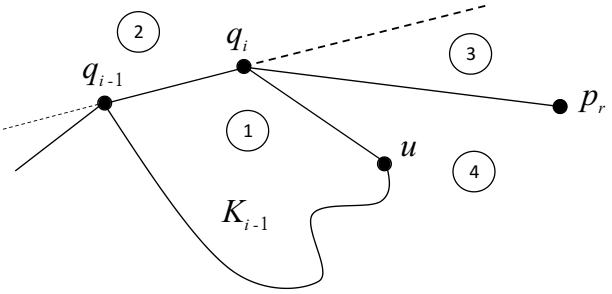
Предположим, что граница P просмотрена от вершины $p_1 = p_l$ до

p_s и вершина $p_s = q_i$ является критической. Пусть $u = p_{s-1}$. Возможны два варианта разбиения вертикальной полосы между точками p_l и p_r на области:

1. Точка u лежит справа от направленной прямой $\overrightarrow{p_r, q_i}$ или на ней:

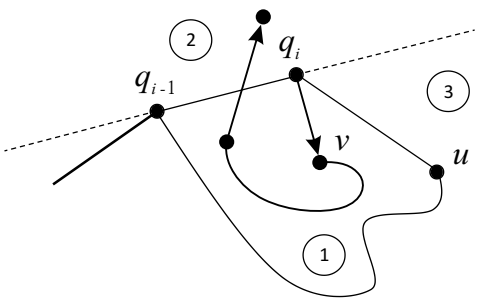


2. Точка u лежит слева от направленной прямой $\overrightarrow{p_r, q_i}$:



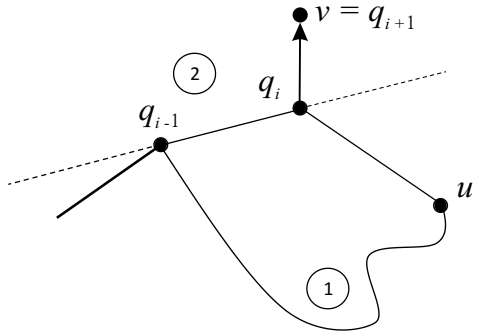
При каждом из вариантов плоскость условно разбивается на три или четыре области, как показано на рисунках. Обозначим через $v = p_{s+1}$ вершину, следующую за q_i. Возможны четыре варианта положения вершины v относительно указанных четырех областей:

1. Вершина v принадлежит области (1). В этом случае граница заходит в карман K_{i-1}:

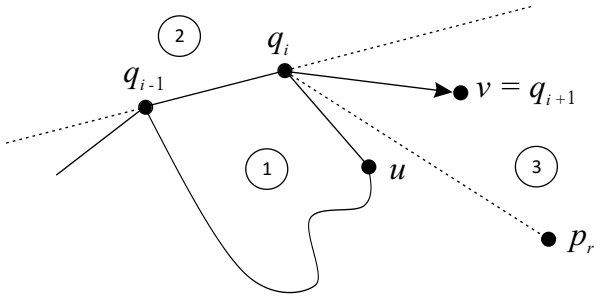


Будем продвигаться по границе до тех пор, пока не попадем в область (2) (это возможно, поскольку многоугольник простой). Переходим к случаю 2.

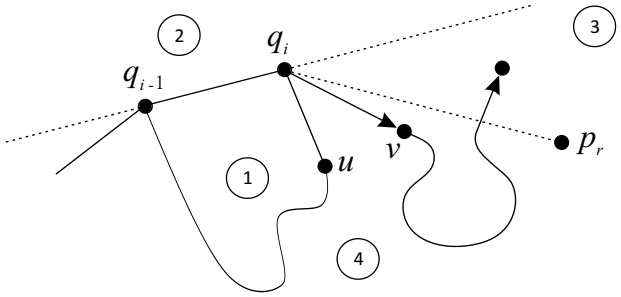
2. Вершина v принадлежит области (2). Это означает, что вершина $v = q_{i+1}$ становится критической. Далее ищем проходящую через вершину v опорную прямую к цепочке $q_1, \ldots, q_i$, удаляя точки q из стека, если тройка q_{top-1},q_{top},v ориентирована влево:



3. Вершина v принадлежит области (3). Значит вершина $v = q_{i+1}$ является критической, так как все пройденные точки $p_1, \ldots, p_s$ лежат слева от $\overrightarrow{q_i, v}$:



4. Вершина v принадлежит области (4), то есть граница заходит внутрь выпуклой оболочки. Будем продвигаться вдоль границы до тех пор, пока не выйдем из области (4) либо текущая точка не совпадет с крайней правой точкой p_r:



Если имеет место второй случай, то алгоритм заканчивает работу: верхняя выпуклая оболочка построена. В противном случае мы попадаем в область (3) или в область (2). Другие варианты невозможны, так как выйти за границу полосы между q_l и p_r мы не можем, так же как не можем пересечь границу карманов и их крышек.

Ниже представлена схема алгоритма. Обозначим через P очередь, содержащую точки многоугольника $p_l, \ldots, p_r$, а через Q – стек для хранения критических точек q. На выходе Q содержит вершины построенной верхней выпуклой оболочки.

```
SimplePolyHull(P)
 1  u ← Pop(P)
 2  v ← Pop(P)
 3  Push(Q, u)
 4  while P ≠ ∅ do
 5      if RightTurn(prev[top[Q]], top[Q], v) then          ▷ случаи 1–4
 6          if RightTurn(u, top[Q], v) then                 ▷ случаи 3,4
 7              if RightTurn(p_r, top[Q], v) then           ▷ случай 3
 8                  u ← v
 9                  Push(Q, v)
10              else                                        ▷ случай 4
11                  repeat
12                      v ← Pop(P)
13                  until RightTurn(p_r, top[Q], v)
14          else                                            ▷ случай 1
15              repeat
16                  v ← P
17              until LeftTurn(prev[top[Q]], top[Q], v)
18      else                                                ▷ случай 2
19          while RightTurn(prev[top[Q]], top[Q], v) do
20              Pop(Q)
21          Push(Q, v)
22      v ← Pop(P)
23  Push(Q, v)
24  return Q
```

Анализ алгоритма показывает, на каждой итерации из входной очереди P удаляется по крайней мере одна вершина (строка 22). Внутри внешнего цикла **while** (строка 4) объем вычислений не превышает константу, за исключением последней ветви **else** (строка 18), в

которой выполняется выталкивание вершин из стека. Всего таких выталкиваний в процессе работы алгоритма не более $O(n)$. Таким образом, справедлива следующая теорема.

Теорема 6. *Выпуклая оболочка простого многоугольника с n вершинами может быть построена за оптимальное время $O(n)$ с использованием $O(n)$ памяти.*

Упражнения

3-1. Выполните сравнительный анализ быстродействия алгоритмов Джарвиса и Грехэма для различных распределений на плоскости исходного множества точек.

3-2. Выполните сравнительный анализ эффективности алгоритма Грехэма и «быстрого» алгоритма.

3-3. Пусть точки множества S расположены в узлах регулярной сетки, стороны которой параллельны координатным осям. Какие модификации необходимо внести в алгоритм Грехэма для его корректной работы?

3-4. Измените входные данные для предыдущего упражнения: последовательно в цикле выполняйте поворот узлов сетки на угол 10^{-10} градуса вокруг самого левого нижнего узла. Вопрос прежний: какие дополнительные модификации требуются для корректной работы алгоритма Грехэма?

3-5. Реализуйте алгоритм Джарвиса с использованием и без использования препроцессорной обработки. Сравните время выполнения обеих программ для различных распределений точек.

3-6. Пусть точки исходного множества расположены на оси X и оси Y по равному количеству в отрицательной и положительной полуосях (то есть в виде «креста»). Постройте выпуклую оболочку такого множества с помощью алгоритма типа «разделяй и властвуй».

Глава 4

Выпуклые отсечения

В этой главе мы решаем задачу выпуклого отсечения, которая заключается в определении частей геометрического объекта, пересекаемых заданной выпуклой областью. В отличие от задач, рассмотренных в предыдущих главах, теперь нас не интересует размер потока входных данных, и основное внимание будет уделено технике отсечения одного объекта относительно другого.

Различают *регулярные* и *нерегулярные* отсечения. Регулярные отсечения выполняются относительно регулярного окна – ортогонального прямоугольника или параллелепипеда.

Любой алгоритм выпуклого отсечения основан на простом правиле: отрезок или многоугольник последовательно отсекается относительно прямой, проходящей через одну из сторон окна (рис. 4.1). Результатом

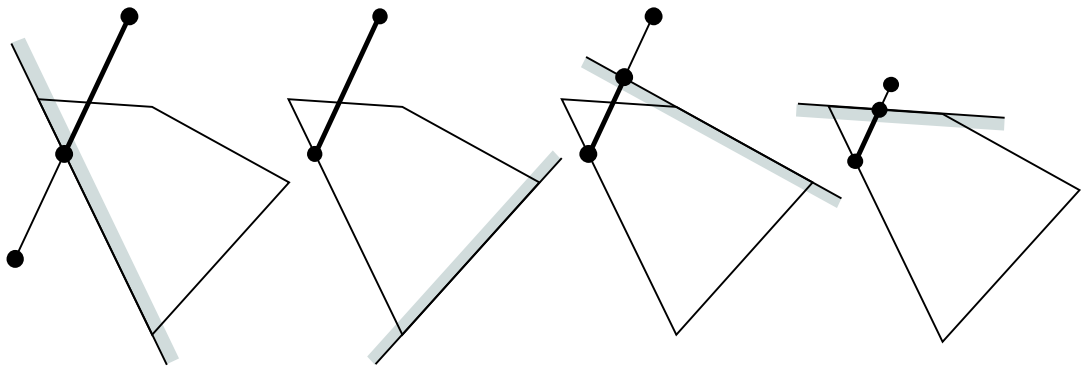


Рис. 4.1. Общая схема отсечения отрезка

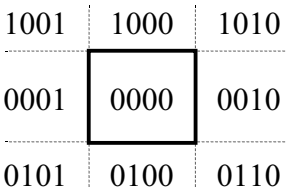
такого отсечения является новый отрезок или многоугольник, к которому применяется отсечение относительно прямой, проходящей через следующую сторону, и т. д. В трехмерном случае отсече-

ние проводится последовательно относительно плоскостей, содержащих грани отсекающего объема. Представленные в этом разделе алгоритмы различаются только по способу реализацией этой идеи. Ниже мы рассмотрим двумерные и трехмерные регулярные отсечения, а также отсечения относительно выпуклых объемов и отсечения многоугольников.

4.1. Двумерные отсечения

4.1.1. Определение видимости отрезка

Пусть регулярное окно W задано координатами левого нижнего и правого верхнего углов: $(X_{\min}, Y_{\min})$ и $(X_{\max}, Y_{\max})$. Существует ли способ установления факта пересечения (или непересечения) отрезка с окном без вычисления самого пересечения? Разумеется, такая проверка должна выполняться максимально быстро, чтобы эффективно производить отсев отрезков, лежащих целиком снаружи или внутри окна. Для выполнения такой проверки служит метод Сазерленда – Коэна (Sutherland – Cohen), суть которого заключается в следующем. Вся плоскость изображения естественным образом разбивается сторонами отсекающего окна на 9 частей:



Для идентификации этих областей вводится 4-разрядный битовый код $B = (b_0, b_1, b_2, b_3)$, с помощью которого однозначно определяется положение точки (x, y) относительно окна:

Бит	Положение	
$b_0 = 1$	$x < X_{\min}$	(левее)
$b_1 = 1$	$x > X_{\max}$	(правее)
$b_2 = 1$	$y < Y_{\min}$	(ниже)
$b_3 = 1$	$y > Y_{\max}$	(выше)

Согласно методу Сазерленда – Коэна оцениваются значения логического произведения $B_1 \wedge B_2$ и логической суммы $B_1 \vee B_2$ кодов концов проверяемого отрезка: отрезок является *тривиально видимым*, если

$B_1 \vee B_2 = 0$ и *тривиально невидимым*, если $B_1 \wedge B_2 \neq 0$. Обратные утверждения, вообще говоря, неверны, то есть, например, условие $B_1 \wedge B_2 = 0$ не позволяет сделать никаких выводов о положении отрезка относительно окна.

4.1.2. Алгоритм Сазерленда – Коэна для регулярного окна

Изложенная выше идея отсечения отрезка наиболее явно представлена в алгоритме Сазерленда – Коэна, согласно которому отрезок с концами в точках $p_1 = (x_1, y_1)$ и $p_2 = (x_2, y_2)$ поочередно разбивается прямыми, проходящими через стороны окна, на две части. Затем та часть, конец которой оказался по внешнюю сторону от окна, отбрасывается, и этот конец заменяется на точку пересечения отрезка с прямой. Таким образом, отрезок укорачивается поочередно всеми сторонами окна. Ниже представлена схема алгоритма (алгоритм возвращает FALSE, если отрезок тривиально невидим):

```
ClipLine2dSC(W, p1, p2)
 1  for i ← 1 to 4 do
 2      b1 ← GetCode(W, i, p1)
 3      b2 ← GetCode(W, i, p2)
 4      if b1 ≠ b2 then
 5          p ← Intersection(W, i, p1, p2)
 6          if b1 = 0 then
 7              p1 ← p
 8          else
 9              p2 ← p
10      else if b1 = 0 then
11          return FALSE
12  return TRUE
```

4.1.3. Алгоритм Лайэнга – Барски для регулярного окна

Хотя с вычислительной точки зрения этот алгоритм менее эффективен, чем алгоритм Сазерленда – Коэна, он демонстрирует универсальный подход, используемый также в алгоритмах для выпуклого и трехмерного отсечения.

Алгоритм Лайэнга – Барски (Liang – Barsky) основан на параметрическом представлении отрезка

$$p(t) = (p_2 - p_1)t + p_1,$$

или в скалярном виде

$$\begin{aligned} x(t) &= (x_2 - x_1)t + x_1, \\ y(t) &= (y_2 - y_1)t + y_1, \end{aligned}$$

где параметр t изменяется в пределах $0 \le t \le 1$.

Точка $p(t)$ отрезка видима, очевидно, тогда, когда она лежит в пределах окна, то есть

$$\begin{aligned} X_{\min} &\le (x_2 - x_1)t + x_1 \le X_{\max}, \\ Y_{\min} &\le (y_2 - y_1)t + y_1 \le Y_{\max}. \end{aligned}$$

Перепишем эти неравенства отдельно:

$$\begin{aligned} -(x_2 - x_1)t &\le x_1 - X_{\min}, \\ (x_2 - x_1)t &\le X_{\max} - x_1, \\ -(y_2 - y_1)t &\le y_1 - Y_{\min}, \\ (y_2 - y_1)t &\le Y_{\max} - y_1. \end{aligned} \tag{4.1}$$

Система неравенств (4.1) относительно t вместе с неравенством $0 \le t \le 1$ задают диапазон изменения параметра t, соответствующий видимой части отрезка:

$$t_{\min} \le t \le t_{\max}.$$

Если система (4.1) не имеет решений, то отрезок невидим. Изначально положим $t_{\min} = 0$, а $t_{\max} = 1$. Стратегия алгоритма состоит в поэтапном уточнении этих значений путем проверки неравенств (4.1).

Заметим, что каждое из неравенств (4.1) имеет общий вид

$$mt \le c. \tag{4.2}$$

При этом в зависимости от знака m неравенство (4.2) ограничивает значение t сверху или снизу. Рассмотрим все варианты.

1. Пусть $m > 0$. Тогда неравенство (4.2) преобразуется к виду

$$t \le \frac{c}{m} = B,$$

   где величина B служит *верхней* границей для параметра t. Относительно нее возможны три случая:

   а) Если $B \le t_{\min}$, то система (4.1) оказывается несовместной, и отрезок невидим (в этом случае он целиком расположен по невидимую сторону относительно прямой, соответствующей данному неравенству).

б) Если $B \geq t_{\max}$, то значения $t_{\min}$ и $t_{\max}$ не изменяется, и никаких выводов относительно видимости отрезка не делается.

в) Если $t_{\min} < B < t_{\max}$, то значение $t_{\max}$ обновляется: $t_{\max} = B$.

2. Пусть теперь $m < 0$. Опять возможны три случая:

   а) Если $B \leq t_{\min}$, то это неравенство не влияет на значения $t_{\min}$ и $t_{\max}$.

   б) Если $B \geq t_{\max}$, то отрезок невидим.

   в) Если $t_{\min} < B < t_{\max}$, то обновляется нижняя граница для параметра t: $t_{\min} = B$.

3. Пусть, наконец, $m = 0$. Это возможно лишь в двух случаях: когда $x_1 = x_2$ или $y_1 = y_2$. Пусть, к примеру, $x_1 = x_2$. Это означает, что отрезок вертикальный и необходимо лишь установить, с какой стороны он расположен относительно вертикальной прямой, проходящей через данную сторону окна. Проверяем неравенство: $0 = t(x_2 - x_1) \leq c$. Если $c \leq 0$, то отрезок целиком расположен за пределами окна и поэтому невидим. В противном случае значения $t_{\min}$ и $t_{\max}$ остаются неизменными.

На рис. 4.2 представлена функция TRangeTest, вычисляющая новые значения $t_{\min}$ и $t_{\max}$ на основе параметров m и c. Она является универсальной и может быть использована также для нерегулярных и трехмерных отсечений. Функция ClipLine демонстрирует пример ее использования в алгоритме Лайэнга – Барски.

4.1.4. Алгоритм Сайруса – Бека для выпуклого окна

Алгоритм Сайруса – Бека представляет собой расширение алгоритма Лайэнга – Барски на произвольные выпуклые окна. Отрезок здесь также представляется в параметрической форме, и требуется найти значения параметра t, соответствующие части отрезка, лежащей внутри окна.

Как и в алгоритме Лайэнга – Барски, для каждой из сторон окна составляется неравенство относительно параметра t. Затем каждое неравенство по очереди анализируется, и при необходимости производится сжатие диапазона $[\, t_{\min}, t_{\max} \,]$.

```
RangeTest(m, c, t_min, t_max)
 1  if m < 0 then
 2      r ← c/m
 3      if r > t_max then
 4          return FALSE
 5      if t_min < r then
 6          t_min ← r
 7  else if m > 0 then
 8      r ← c/m
 9      if r < t_min then
10          return FALSE
11      if r < t_max then
12          t_max ← r
13  else if c < 0 then
14      return FALSE
15  return TRUE


ClipLine2dLB(W, p_1, p_2)
 1  t_min ← 0
 2  t_max ← 1
 3  dx ← x[p_2] − x[p_1]
 4  if RangeTest(−dx, x[p_1] − X_min[W], t_min, t_max)) then
 5      if RangeTest(dx, X_max[W] − x[p_1], t_min, t_max)) then
 6          dy ← y[p_2] − y[p_1]
 7          if RangeTest(−dy, y[p_1] − Y_min[W], t_min, t_max)) then
 8              if RangeTest(dy, Y_max[W] − y[p_1], t_min, t_max)) then
 9                  if t_max < 1 then
10                      x_2 ← round(x[p_1] + t_max dx)
11                      y_2 ← round(y[p_1] + t_max dy)
12                  if t_min > 0 then
13                      x_1 ← round(x[p_1] + t_min dx)
14                      y_1 ← round(y[p_1] + t_min dy)
15                  return TRUE
16  return FALSE
```

Рис. 4.2. Отсечение отрезка методом Лайэнга – Барски

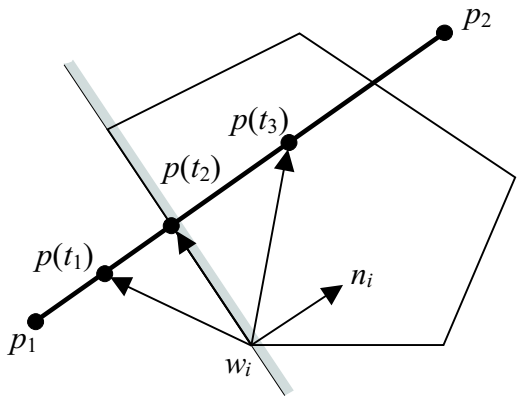


Рис. 4.3. Скалярное произведение $n_i(p(t) - w_i)$ отрицательно для $p(t_1)$, равно 0 для $p(t_2)$ и положительно для $p(t_3)$

Ответим сначала на вопрос: при каких t точка $p(t)$ и окно расположены в одной полуплоскости относительно прямой, проходящей через i-ю сторону окна? Рассмотрим вершину w_i окна и нормаль n_i к i-й стороне окна (рис. 4.3) и оценим знак скалярного произведения

$$n_i(p(t) - w_i).$$

Знак скалярного произведения двух векторов определяется знаком косинуса угла α между ними, причем косинус положительный, когда угол α острый. Из рис. 4.3 видно, что если угол α острый, то точка $p(t)$ и окно лежат по одну сторону от i-го ребра, если тупой, то – по разные, если прямой, то точка $p(t)$ принадлежит ребру. Таким образом, выполнение неравенства

$$n_i(p(t) - w_i) \geq 0, \quad i = 1 \ldots n$$

является условием принадлежности точки $p(t)$ окну. Перепишем

$$n_i(p_1 + t(p_2 - p_1) - w_i) \geq 0 \;\text{ или }\; n_i(p_1 - p_2)t \leq n_i(p_1 - w_i).$$

Как и в алгоритме Лайэнга – Барски, условия приняли вид $m_i t \leq c_i$. Отсечение отрезка производится применением процедуры TRangeTest к каждой стороне многоугольника.

4.2. Трехмерные отсечения

В трехмерном случае отсечения производятся относительно конечных областей пространства. Необходимость в этих операциях возникает при построении проекций трехмерных объектов. При этом в случае параллельных проекций отсечение проводится относительно параллелепипеда со сторонами, ортогональными координатным осям, а в случае центральной проекции – относительно правильной усеченной пирамиды. Рассмотрим оба случая.

4.2.1. Отсечения относительно параллелепипеда

Какие изменения необходимо внести в алгоритмы Лайэнга – Барски и Сазерленда – Коэна, чтобы они заработали для трехмерных отсечений?

В трехмерном случае отсекающий параллелепипед также задается двумя точками: $W_{\min} = (X_{\min}, Y_{\min}, Z_{\min})$ и $W_{\max} = (X_{\max}, Y_{\max}, Z_{\max})$. Векторное параметрическое представление отрезка не меняется:

$$p(t) = (p_2 - p_1)t + p_1,\ 0 \le t \le 1,$$

где $p_1 = (x_1, y_1, z_1)$ и $p_2 = (x_2, y_2, z_2)$. Как и на плоскости, точка $p(t)$ отрезка, находящаяся внутри области, удовлетворяет системе из шести неравенств $W_{\min} \le p(t) \le W_{\max}$ или в скалярном виде

$$\begin{array}{rcccl} X_{\min} & \le & (x_2 - x_1)t + x_1 & \le & X_{\max}, \\ Y_{\min} & \le & (y_2 - y_1)t + y_1 & \le & Y_{\max}, \\ Z_{\min} & \le & (z_2 - z_1)t + z_1 & \le & Z_{\max}, \end{array}$$

общий вид которых $mt \le c$. При отсечении методом Лайэнга – Барски для каждого из них вызывается процедура TRangeTest, уточняющая интервал $[t_{min},\ t_{max}]$ изменения параметра t, соответствующий видимой части отрезка.

Точки пересечения отрезка с плоскостями, ограничивающими параллелепипед, вычисляются также с использованием параметрического представления отрезка. Например, значение параметра t, соответствующее точке пересечения отрезка с плоскостью

$$x(t) = (x_2 - x_1)t + x_1 = x_{max} \quad \text{или} \quad t = \frac{x_{max} - x_1}{x_2 - x_1}.$$

Трехмерный алгоритм Сазерленда – Коэна предусматривает введение шестизначного кода $B = (b_0, b_1, b_2, b_3, b_4, b_5)$, с помощью которого однозначно определяется положение точки (x, y) относительно окна:

Бит	Положение	
$b_0 = 1$	$x < X_{\min}$	(левее)
$b_1 = 1$	$x > X_{\max}$	(правее)
$b_2 = 1$	$y < Y_{\min}$	(ниже)
$b_3 = 1$	$y > Y_{\max}$	(выше)
$b_4 = 1$	$z < Z_{\min}$	(спереди)
$b_5 = 1$	$z > Z_{\max}$	(сзади)

4.2.2. Отсечения относительно усеченной пирамиды

Эти отсечения применяются при построении центральных проекций трехмерных объектов (см. раздел 7.4.3). В результате применения гео-

метрических преобразований видимый объем преобразуется к каноническому виду, который представляет собой усеченную пирамиду, ограниченную шестью плоскостями:

$$\begin{aligned} -z \leq\ & x & \leq z, \\ -z \leq\ & y & \leq z, \\ z_{\min} \leq\ & z & \leq 1. \end{aligned}$$

Как и в случае параллелепипеда, необходимо эффективно произвести отсечение относительно этого объема. Согласно алгоритму Лайэнга – Барски любая точка $p(t)$ отрезка, находящаяся внутри канонической пирамиды, должна удовлетворять системе шести неравенств:

$$\begin{aligned} -z(t) &\leq x(t) \leq z(t), \\ -z(t) &\leq y(t) \leq z(t), \\ z_{min} &\leq z(t) \leq 1 \end{aligned}$$

или

$$\begin{aligned} -((z_2 - z_1)t + z_1) &\leq ((x_2 - x_1)t + x_1) \leq ((z_2 - z_1)t + z_1), \\ -((z_2 - z_1)t + z_1) &\leq ((y_2 - y_1)t + y_1) \leq ((z_2 - z_1)t + z_1), \\ z_{min} &\leq ((z_2 - z_1)t + z_1) \leq 1, \end{aligned}$$

общий вид которых $mt \leq c$. Например, неравенство

$$-((z_2 - z_1)t + z_1) \leq ((x_2 - x_1)t + x_1)$$

эквивалентно условию

$$-(z_2 - z_1 - x_2 + x_1)t \leq x_1 - z_1.$$

Алгоритм Сазерленда – Коэна для усеченной пирамиды предусматривает введение шестибитового кода, определяющего положение конца отрезка относительно усеченной пирамиды:

Бит	Положение	
$b_0 = 1$	$x > z$	(левее)
$b_1 = 1$	$x < -z$	(правее)
$b_2 = 1$	$y > z$	(выше)
$b_3 = 1$	$y > -z$	(ниже)
$b_4 = 1$	$z < z_{\min}$	(спереди)
$b_5 = 1$	$z > 1$	(сзади)

Пересечение отрезка с наклонными плоскостями вычисляется с помощью его параметрического представления. Для плоскости $x = z$ имеем

$$(x_2 - x_1)t + x_1 = (z_2 - z_1)t + z_1$$

или

$$t = \frac{z_1 - x_1}{x_2 - x_1 - z_2 + z_1}.$$

4.3. Отсечение многоугольников

Многоугольник может иметь два представления: как замкнутая ломаная на плоскости и как часть плоскости, ограниченная этой ломаной. В обоих случаях он задается списком вершин в произвольном порядке обхода. Однако в первом случае после применения операции отсечения многоугольник распадается на разрозненные отрезки, и результат отсечения уже не может быть представлен списком вершин (рис. 4.4). Поэтому для отсечения многоугольников должны

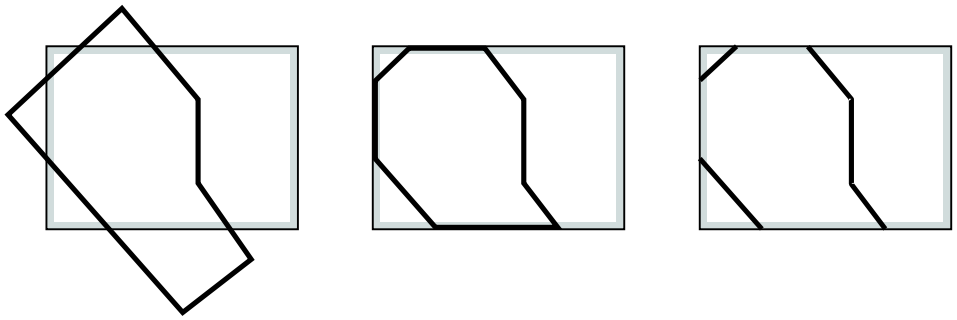


Рис. 4.4. Отсечение многоугольника как сплошной области и как набора отрезков

применяться алгоритмы, учитывающие специфику представления сплошных областей.

Рассмотрим алгоритм отсечения Сазерленда – Ходгмана, основанный на той же идее, что и алгоритм последовательного отсечения отрезка Сазерленда – Коэна: многоугольник последовательно отсекается прямыми, проходящими через стороны окна. Результатом работы каждого шага алгоритма является список вершин многоугольника, лежащих по *видимую* сторону от отсекающей прямой.

Обозначим через $P = (p_1, \ldots, p_n)$ и $Q = (q_1, \ldots, q_n)$ списки вершин соответственно исходного и отсеченного многоугольников. Отсечение

относительно прямой может удалить часть вершин, может породить новые. При этом пересечение одного ребра с прямой может породить только одну новую вершину. Возможны четыре случая (рис. 4.5):

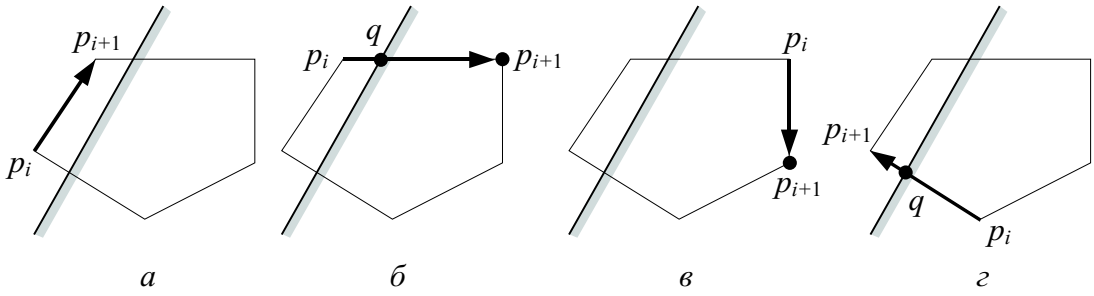


Рис. 4.5. Алгоритм Сазерленда – Ходгмана

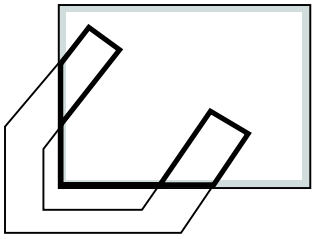
а) ребро (p_i, p_{i+1}) многоугольника целиком лежит по невидимую сторону от отсекающей прямой; в этом случае ни один из концов ребра не должен присутствовать в описании результирующего многоугольника;

б) вход в область видимости: p_i лежит по невидимую сторону, p_{i+1} – по видимую сторону от отсекающей прямой; в описание включаются точка q пересечения ребра с прямой и точка p_{i+1}: $\{q, p_{i+1}\} \to Q$;

в) ребро (p_i, p_{i+1}) целиком лежит по видимую сторону от отсекающей прямой; тогда в описание результирующего многоугольника включается только точка p_{i+1}: $p_{i+1} \to Q$;

г) выход из области видимости: p_i лежит по видимую сторону, p_{i+1} – по невидимую сторону от отсекающей прямой; в описание включается точка q пересечения со стороной окна: $q \to Q$.

Замечание 1. В приведенном выше алгоритме не упомянуты какие-либо ограничения на количество сторон отсекающего окна или на их расположение, поскольку алгоритм работает для любого выпуклого окна. Более того, нигде нет ограничения на размерность пространства, в котором производится отсечение, откуда следует, что алгоритм работает также для произвольного выпуклого отсекающего многогранника и плоского многоугольника в трехмерном пространстве. В этом случае поэтапное отсечение производится относительно плоскости, содержащей грань отсекающего многогранника.



Замечание 2. Как показано на рисунке, в результате отсечения часть сторон многоугольника могут оказаться вырожденными. В этом случае необходима несложная дополнительная процедура, разбивающая такой многоугольник на невырожденные составляющие.

Упражнения

2-1. Сравните быстродействие алгоритмов Сазерленда – Коэна и Лайэнга – Барски для отсечения отрезков относительно регулярного окна. В качестве теста напишите однооконное приложение, генерирующее заданное количество отрезков и отображающее их видимую часть относительно прямоугольной области фиксированного размера.

2-2. Напишите программу, выполняющую отсечение заданного многоугольника относительно фиксированного выпуклого многоугольника. Вершины отсекаемого многоугольника должны задаваться щелчками мыши. Доработайте алгоритм Сазерленда – Ходгмана таким образом, чтобы результатом отсечения всегда были невырожденные многоугольники.

Глава 5

Пересечения

Из предыдущей главы мы знаем, как производить отсечение одного геометрического примитива относительно другого. Рассмотрим теперь ситуацию, когда таких примитивов много и необходимо найти пересечения всех пар из заданного множества. В отличие от методов, рассмотренных в предыдущей главе, алгоритмы вычислительной геометрии используют операцию вычисления пересечения двух примитивов как одну операцию с некоторой фиксированной стоимостью. Необходимо построить алгоритм с оптимальной асимптотической оценкой времени выполнения, учитывающей размер потока входных данных.

5.1. Пересечение выпуклых многоугольников

Рассмотрим задачу нахождения пересечения двух выпуклых многоугольников с точки зрения вычислительной геометрии: даны два выпуклых многоугольника P_1 и P_2 с n и m вершинами соответственно. Построить оптимальный по быстродействию алгоритм, вычисляющий их пересечение.

Алгоритм Сазерленда – Ходгмана вычисляет это пересечение за $O(nm)$ в наихудшем случае. Действительно, для каждой из n сторон отсекающего многоугольника P_1 анализируются на пересечение все m сторон отсекаемого многоугольника P_2.

Существует ли более быстрый алгоритм? Нижняя оценка построения пересечения равна $\Omega(n + m)$, которая следует из простого факта, что пересечением P_1 и P_2 может оказаться многоугольник с $n + m$ вершинами (рис. 5.1, *а*). Следующий алгоритм, предложенный Шеймосом и Хоуи, тратит не более $\Theta(n + m)$ времени на построение пересечения многоугольников и поэтому является оптимальным по

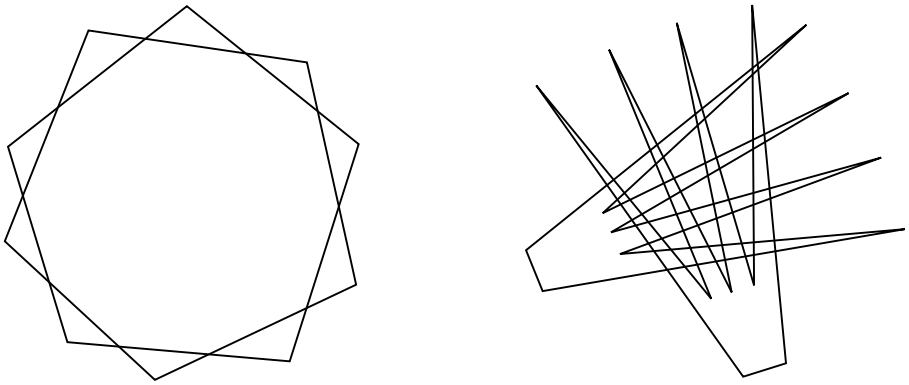


Рис. 5.1. Пересечение выпуклых многоугольников с n и m вершинами ($а$) может иметь $n+m$ вершин. Для звездных многоугольников ($б$) число точек пересечения ребер может достигать $O(n^2)$

быстродействию. Заметим, что в силу выпуклости многоугольников P_1 и P_2, их вершины могут быть отсортированы по возрастанию абсцисс и объединены в один общий список за время $O(n+m)$. Проведем через вершины обоих многоугольников вертикальные прямые, образующие $n+m-1$ полосу (рис. 5.2). Пересечением полосы с многоугольником является трапеция (в вырожденном случае – треугольник). Согласно

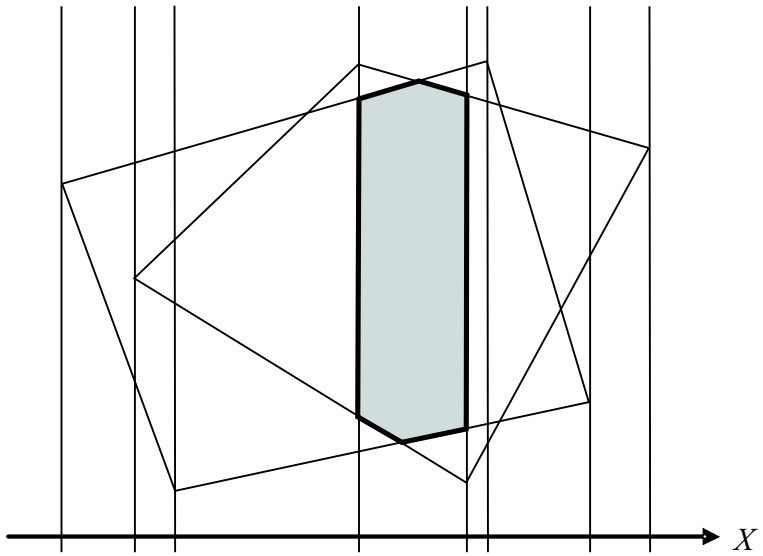


Рис. 5.2. Пересечение многоугольников строится как объединение пересечений трапеций

алгоритму полосы просматриваются последовательно слева направо, для каждой полосы за время $O(1)$ вычисляется пересечение трапеций, затем найденные пересечения объединяются в один многоугольник.

Возникает вопрос: является ли выпуклость обоих многоугольников определяющим свойством, позволяющим построить алгоритм сложности $O(n+m)$? Можно ли добиться аналогичного результата, например, для многоугольников, вершины которых просто упорядочены по полярному углу? Ответ отрицательный. Действительно, как показано на рис. 5.1, *б*, для звездных многоугольников число точек пересечения ребер может достигать порядка $O(n^2)$.

5.2. Пересечение прямолинейных отрезков

5.2.1. Нижние оценки

Рассмотрим следующую задачу.

Задача 12 (Идентификация пересечения). *На плоскости даны n отрезков. Пересекаются ли какие-либо два из них?*

Один из способов получения нижней оценки сложности решения задачи состоит в сведении к ней задачи с известной нижней оценкой. В данном случае роль такой тестовой задачи играет задача об уникальности элементов:

Задача 13 (Уникальность элементов). *Даны n действительных чисел $x_1, \ldots, x_n$. Все ли из них различны?*

Для сведения задачи 13 к задаче 12 расположим на числовой прямой интервалы вида $[x_i, x_i]$ и рассмотрим их как отрезки нулевой длины. Они пересекаются тогда и только тогда, когда среди чисел x_i существуют хотя бы два одинаковых. Нижняя оценка для задачи 13 известна и равна $\Omega(n \log n)$, откуда заключаем, что эта же оценка справедлива для задачи 12.

Попытаемся теперь получить нижнюю оценку для более общей задачи:

Задача 14 (Пересечение отрезков). *На плоскости даны n отрезков. Найти все их пересечения.*

Можно рассуждать так: для нахождения всех пересечений необходимо найти хотя бы одно, то есть решить задачу 12. Вдобавок необходимо

перечислить все найденные пересечения. Поэтому задача 14 не может быть решена быстрее, чем за время $\Omega(n \log n + k)$, где k – число всех пересечений.

Следует помнить, что из всех нижних оценок интерес представляют только *достижимые* нижние оценки. Рассмотрим алгоритм для нахождения пересечения ортогональных отрезков, сложность которого совпадает с полученной нижней оценкой.

5.2.2. Пересечение ортогональных отрезков

Задача 15 (Пересечение ортогональных отрезков). *На плоскости даны n отрезков, лежащих на прямых, параллельных координатным осям. Найти все их пересечения.*

Тривиальный подход, требующий сравнения всех пар отрезков, дает трудоемкость $O(n^2)$. Улучшить эту оценку позволяет простая сортировка. Действительно, упорядочим концы отрезков по возрастанию их абсцисс и рассмотрим вертикальную заметающую прямую l, сканирующую плоскость вдоль оси X (рис. 5.3). Статус L заметающей

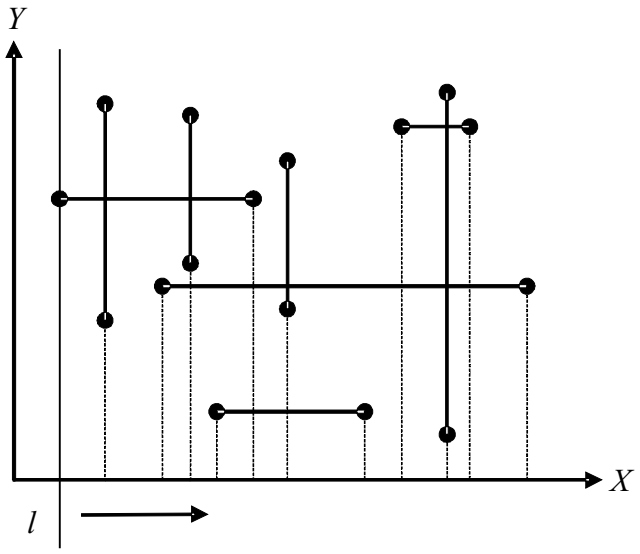


Рис. 5.3. Заметающая прямая при поиске пересечений ортогональных отрезков

прямой представляет собой динамическую структуру, хранящую упорядоченный по возрастанию ординат набор горизонтальных отрезков, пересекаемых прямой l.

Статус L изменяется только в критических точках – абсциссах концов горизонтальных отрезков. Проверки пересечений необходимо выполнять только при проходе вертикального отрезка и только с теми горизонтальными отрезками, которые содержатся в L.

Структура L должна поддерживать вставку и удаление горизонтальных отрезков за время, не превышающее $O(\log n)$, а определение k пересечений за время $O(k)$. Она может быть реализована, например, в виде сбалансированного по высоте поискового дерева. При проходе через вертикальный отрезок необходимо перечислить все горизонтальные отрезки дерева, ординаты которых лежат между ординатами концов вертикального отрезка. Для того чтобы на перечисление k отрезков затратить не более $O(k)$ времени, каждая вершина дерева L должна содержать также указатели на предыдущую и последующую вершины в соответствии с заданным порядком[1].

Ниже представлена схема алгоритма. Через E обозначена очередь, содержащая концы отрезков, отсортированные по возрастанию абсцисс. Дерево L содержит горизонтальные отрезки, пересекающие заметающую прямую и отсортированные в порядке возрастания ординат. Выходная очередь A содержит пары пересекающихся отрезков.

```
ОrtLineIntersection(E, A)
 1  while E ≠ ∅ do
 2      p ← Pop(E)
 3  if p – левый конец отрезка s then
 4          Insert(L, s)
 5      else
 6          if p – правый конец отрезка s then
 7              Delete(L, s)
 8          else          ▷ s – вертикальный отрезок
 9              t ← LowerBound(L, y_min[s])
10              while y[t] ≤ y_max[s] do
11                  Push(A, (t, s))
12                  t ← next[t]
```

Функция LowerBound$(L, y_{\min}[s])$ возвращает хранящийся в L горизонтальный отрезок с минимальной ординатой, которая не меньше ординаты нижнего конца отрезка s. Цикл в строке 10 выполняет линейный перебор горизонтальных отрезков из интервала $y_{\min}[s], \ldots, y_{\max}[s]$.

[1] Такая структура данных называется *прошитый словарь* (см. разд. 1.6).

Оценим трудоемкость алгоритма. Число итераций внешнего цикла составляет удвоенное число горизонтальных отрезков, то есть не более $O(n)$, на вставку и удаление элементов расходуется не более $O(\log n)$, это же время необходимо для поиска самого нижнего и самого верхнего горизонтальных отрезков, пересекающих встреченный вертикальный отрезок. Перечисление горизонтальных отрезков из найденного диапазона занимает в сумме не более $O(k)$.

Таким образом, найти пересечение n ортогональных отрезков можно за оптимальное время $\Theta(n \log n + k)$.

5.2.3. Пересечение произвольных отрезков

Для эффективного решения задачи 14 нам потребуется усовершенствовать структуру L, реализующую статус заметающей прямой l. Обозначим через x абсциссу прямой l и введем на множестве S отрезков отношение линейного порядка $<_x$: для $s_1,\ s_2 \in S$ соотношение $s_1 <_x s_2$ выполняется тогда и только тогда, когда точка пересечения s_1 с прямой l лежит ниже точки пересечения l с s_2 (рис. 5.4).

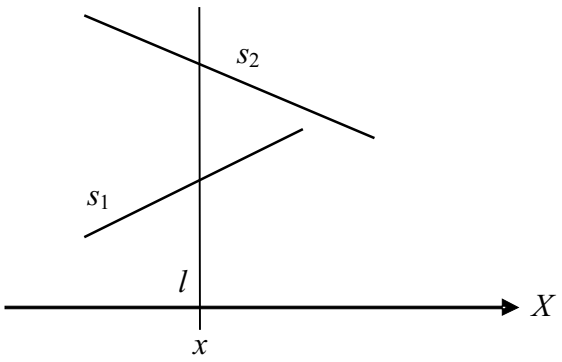


Рис. 5.4. Отношение $<_x$ линейного порядка между отрезками

При движении прямой l вдоль оси X ее статус может измениться только в трех случаях:

1. Встретился левый конец некоторого отрезка s. В этом случае отрезок s вставляется в L в соответствии с отношением порядка $<_x$.
2. Встретился правый конец отрезка s. В этом случае отрезок s удаляется из L.

```
LineIntersection(S, Q)
 1  Отсортировать 2n концов отрезков множества S
    и поместить их в очередь E
 2  while E ≠ ∅ do
 3      p ← Pop(E)
 4      if p – левый конец отрезка s then
 5          Insert(L, s)
 6          s1 ← Above(s, L)
 7          s2 ← Below(s, L)
 8          if s1 пересекает s справа от p then
 9              Push(A, (s1, s))
10          if s2 пересекает s справа от p
11              Push(A, (s2, s))
12      else if p – правый конец отрезка s then
13          s1 ← Above(s, L)
14          s2 ← Below(s, L)
15          if s1 пересекает s2 справа от p
16              Push(A, (s1, s2))
17          Delete(L, s)
18      else            ▷ p – точка пересечения
19          Пусть s1 и s2 – отрезки, пересекающиеся в p,
            причем s1 = Above(s2, L) слева от p
20          s3 ← Above(s1, L)
21          s4 ← Below(s2, L)
22          if s3 пересекает s2 справа от p
23              Push(A, (s3, s2))
24          if s4 пересекает s1 справа от p
25              Push(A, (s4, s1))
26          Поменять s1 и s2 местами в L
27      while A ≠ ∅ do
28          (s1, s2) ← Pop(A)
29          Пусть p – точка пересечения отрезков s1 и s2
30          if p ∉ E then
31              Push(Q, (s1, s2))
32              Insert(E, p)
```

Рис. 5.5. Алгоритм Бентли – Оттмана

3. Встретилась точка пересечения отрезков s_1 и s_2. В этом случае отрезки s_1 и s_2 меняются местами в L.

Для поиска пар пересекающихся отрезков определяющим является факт, что пересекающиеся отрезки рано или поздно становятся соседними в L при прохождении сканирующей прямой через одну из критических точек. Поэтому во всех трех перечисленных случаях необходимо проверять на пересечение отрезки, ставшие соседними в результате операции вставки, удаления или перемещения.

Найденная точка пересечения вставляется в очередь E в качестве еще одной критической точки. Время на вставку не должно превышать $O(\log n)$. Таким образом, в соответствии с классификацией, приведенной в разделе 1.6, структура данных E представляет собой очередь с приоритетом, которая может быть реализована в виде сбалансированного поискового дерева.

На рис. 5.5 представлена схема алгоритма Бентли – Оттмана, производящего поиск пересечений отрезков. Функции Above(s, L) и Below(s, L) возвращают отрезки, лежащие, соответственно, выше и ниже отрезка s в структуре L. Найденные пары пересекающихся отрезков помещаются в выходную очередь Q.

Нетрудно оценить трудоемкость алгоритма. Действительно, вставка и удаление в дерево L требует не более $O(n \log n)$ элементарных операций, в то время как вставка в дерево E k найденных точек пересечения (строка 31) требует не более $O(k \log n)$. В сумме получаем оценку $O((n + k) \log n)$.

Заметим, что в наихудшем случае $k = O(n^2)$ и тогда даже простое попарное сравнение отрезков работает быстрее алгоритма Бентли – Оттмана. Однако в случае, когда отрезки имеют относительно небольшую длину и распределены равномерно, число пересечений растет пропорционально числу отрезков и алгоритм Бентли – Оттмана работает намного быстрее попарного сравнения.

5.2.4. Идентификация пересечения

Вернемся к задаче 12. Для идентификации пересечения отрезков необходимо просто просмотреть в структуре L все пары отрезков, которые становятся соседними в критических точках.

Упрощенный вариант алгоритма Бентли – Оттмана для идентификации пересечения отрезков выглядит следующим образом:

```
DetectLineIntersection(S)
 1  Отсортировать 2n концов отрезков множества S
    и поместить их в вектор M
 2  for i ← 1 to 2n do
 3      p ← M[i]
 4      if p – левый конец отрезка s then
 5          Insert(L, s)
 6          s1 ← Above(s, L)
 7          s2 ← Below(s, L)
 8          if s1 пересекает s then
 9              return TRUE
10          if s2 пересекает s then
11              return TRUE
12      else if p – правый конец отрезка s then
13          s1 ← Above(s, L)
14          s2 ← Below(s, L)
15          if s1 пересекает s2 then
16              return TRUE
17          Delete(L, s)
18  return FALSE
```

Поскольку сложность каждой из $2n$ итераций цикла не превосходит $O(\log n)$, получаем, что факт пересечения какой-либо пары из n отрезков можно установить за оптимальное время $\Theta(n \log n)$.

Упражнения

2-1. Реализуйте алгоритм нахождения пересечения n прямоугольников на плоскости, используя технику заметающей прямой. Оцените сложность алгоритма. Является ли он оптимальным?

2-2. [28] Задано множество S, состоящее из n отрезков на плоскости. Разработать алгоритм, который решал бы вопрос о существовании прямой, пересекающей все отрезки S.

2-3. Можно ли ускорить алгоритм Бентли – Оттмана, если поместить отрезки (или их части) в домены квадродерева?

2-4. [28] Разработайте алгоритм выполнения массового запроса, который находит точки пересечения отрезка t с заданным множеством отрезков S.

Глава 6

Геометрические преобразования

Геометрические преобразования применяются для моделирования динамических процессов, включающих перемещение и модификацию объектов, а также для построения проекций трехмерных объектов. Ниже мы рассмотрим элементарные аффинные преобразования на плоскости и в пространстве, а также эффективные алгоритмы их вычисления.

6.1. Линейные преобразования на плоскости

Линейные преобразования ставят в соответствие произвольной точке $p = (x, y)$ на плоскости точку $p' = (x', y')$ по правилу

$$x' = ax + cy, \quad y' = bx + dy$$

или в матрично-векторной форме

$$p' = pM, \text{ где } M = \begin{pmatrix} a & b \\ c & d \end{pmatrix}.$$

Существуют два элементарных линейных преобразования, все остальные являются их композицией:

- поворот относительно начала координат и
- масштабирование.

При повороте против часовой стрелки на угол θ относительно начала координат образом точки $(x, y) = (r\cos\varphi, r\sin\varphi)$ является точка

$$\begin{aligned} (x', y') &= (r\cos(\varphi+\theta), r\sin(\varphi+\theta)) \\ &= (r\cos\varphi\cos\theta - r\sin\varphi\sin\theta, r\cos\varphi\sin\theta + r\sin\varphi\cos\theta) \\ &= (x\cos\theta - y\sin\theta, x\sin\theta + y\cos\theta), \end{aligned}$$

откуда получаем матрицу поворота

$$R(\theta) = \begin{pmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{pmatrix}.$$

Масштабирование (растяжение или сжатие вдоль осей) есть преобразование вида

$$x' = xS_x, \quad y' = yS_y,$$

и соответствующая матрица имеет вид

$$S(S_x, S_y) = \begin{pmatrix} S_x & 0 \\ 0 & S_y \end{pmatrix}.$$

Масштабирование называется *однородным*, если $S_x = S_y$, то есть не влияет на пропорции объекта.

Параллельный перенос является не линейным, а *аффинным* преобразованием. Аффинное преобразование ставит в соответствие точке $p = (x, y)$ на плоскости точку $p' = (x', y')$ по правилу

$$x' = ax + cy + t_x, \quad y' = bx + dy + t_y$$

или в матрично-векторной форме

$$p' = pM + t,$$

где вектор $t = (t_x, t_y)$ называется вектором *параллельного переноса*. Очевидно, при нулевом векторе t аффинное преобразование не может быть реализовано посредством умножения на некоторую матрицу.

6.2. Однородные координаты

Для того чтобы любое аффинное преобразование, включая параллельный перенос, могло быть реализовано посредством умножения на матрицу, вводятся так называемые *однородные координаты*: каждой точке $p = (x, y)$ на плоскости ставится в соответствие точка $p' = (wx, wy, w)$, заданная в однородных координатах для некоторого масштабного множителя $w \neq 0$. Декартовы координаты точки, представленной в однородных координатах, можно получить путем деления однородных координат на w. Представление точки с коэффициентом $w = 1$ называется *каноническим*.

С учетом однородных координат элементарные аффинные преобразования могут быть реализованы с помощью умножения на матрицу:

- поворота относительно начала координат:

$$R(\theta) = \begin{pmatrix} \cos\theta & \sin\theta & 0 \\ -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix},$$

- масштабирования:

$$S(S_x, S_y) = \begin{pmatrix} S_x & 0 & 0 \\ 0 & S_y & 0 \\ 0 & 0 & 1 \end{pmatrix},$$

- и параллельного переноса:

$$T(\delta x, \delta y) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ \delta x & \delta y & 1 \end{pmatrix}.$$

6.3. Композиция двумерных преобразований

Последовательное выполнение геометрических преобразований называется *композицией*. Рассмотрим пример. Для поворота точки $p = (x, y, 1)$ на угол θ относительно *произвольной* точки достаточно выполнить три преобразования:

1) параллельный перенос в начало координат (рис. 6.1, *а*);

2) поворот относительно начала координат (рис. 6.1, *б*);

3) параллельный перенос на вектор (x, y) (рис. 6.1, *в*).

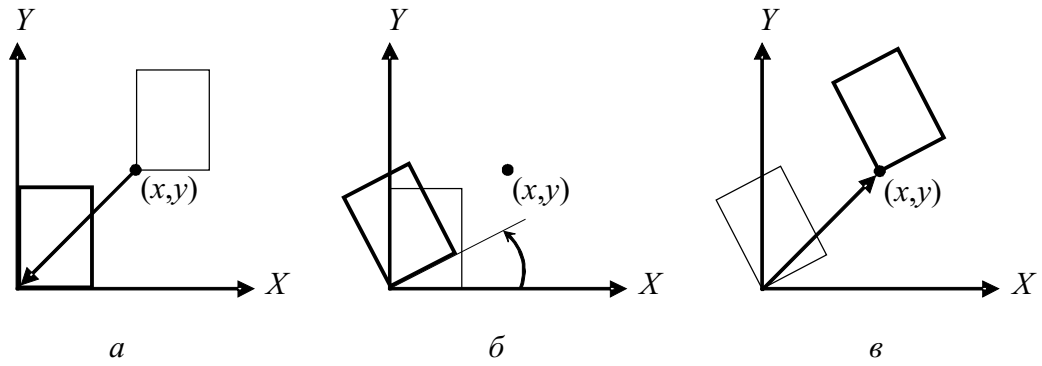


Рис. 6.1. Поворот относительно точки (x, y): параллельный перенос в начало координат (*а*), поворот относительно начала координат (*б*) и параллельный перенос на вектор (x, y) (*в*)

В результате получаем

$$\begin{aligned} p' &= (((p\,T(-x,-y))\,R(\theta))\,T(x,y) = \\ &= p\,(T(-x,-y)\,R(\theta)\,T(x,y)) = \\ &= p\,M. \end{aligned}$$

Таким образом, матрица, реализующая композицию геометрических преобразований, есть произведение матриц этих преобразований.

Как известно, в общем случае умножение матриц не является коммутативной операцией. В следующей таблице представлены пары элементарных преобразований, порядок выполнения которых не влияет на результат композиции:

M_1	M_2
Перенос	Перенос
Масштабирование	Масштабирование
Поворот	Поворот
Масштабирование (однородное)	Поворот

6.4. Эффективность вычислений

Матрица любой композиции элементарных аффинных преобразований имеет общий вид

$$M = \begin{pmatrix} r_{11} & r_{12} & 0 \\ r_{21} & r_{22} & 0 \\ t_x & t_y & 1 \end{pmatrix}.$$

При этом подматрица

$$\begin{pmatrix} r_{11} & r_{12} \\ r_{21} & r_{22} \end{pmatrix}$$

хранит информацию о всех поворотах и масштабированиях, а числа t_x и t_y описывают суммарный перенос. Если умножение pM производить напрямую, без учета структуры матрицы M, то это потребует 9 умножений и 6 сложений. С другой стороны, вычисление преобразования в виде

$$(x,y) \times M = \begin{pmatrix} r_{11}x + r_{21}y + t_x \\ r_{12}x + r_{22}y + t_y \end{pmatrix}$$

требует всего 4 умножения и 4 сложения. Такой прием дает особенно ощутимый эффект, когда преобразование применяется к объекту, состоящему из большого числа геометрических примитивов.

6.5. Трехмерные геометрические преобразования

По аналогии с геометрическими преобразованиями на плоскости трехмерные геометрические преобразования реализуются умножением на соответствующую матрицу размера 4×4. Точки в трехмерном пространстве также представимы в однородных координатах: $p = (wx, wy, wz, w)$ или в каноническом виде $p = (x, y, z, 1)$.

Говорят, что взаимно ортогональные единичные вектора e_1, e_2 и e_3 образуют *правую тройку*, если при повороте вокруг вектора e_3 против часовой стрелки на угол 90° вектор e_1 совместится с вектором e_2 (рис. 6.2). В определении *левой тройки* аналогичные повороты выполняются по часовой стрелке.

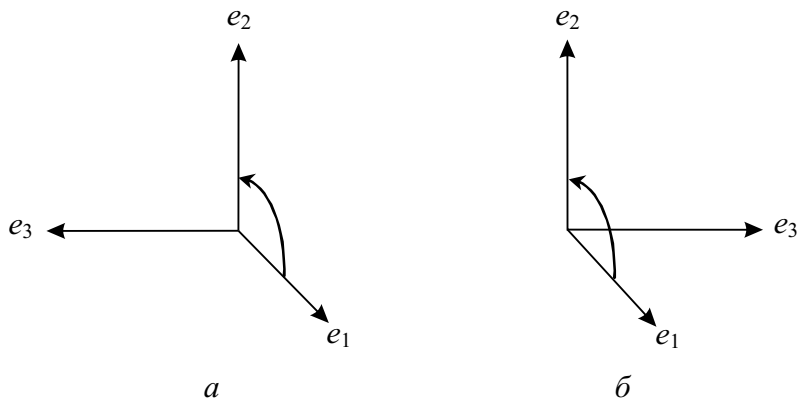


Рис. 6.2. Правая (*а*) и левая (*б*) тройки векторов

Определенная таким образом тройка (e_1, e_2, e_3) задает *правостороннюю систему координат*, называемую также *мировой*. В дальнейшем все представления и преобразования объектов будем производить в правосторонней системе координат.

Трехмерные геометрические преобразования описываются матрицами размера 4×4 и являются обобщением двумерных преобразований:

- параллельный перенос:

$$T(t_x, t_y, t_z) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ t_x & t_y & t_z & 1 \end{pmatrix},$$

- масштабирование:

$$S(S_x, S_y, S_z) = \begin{pmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

Элементарный трехмерный поворот выполняется в одной из координатных плоскостей вокруг соответствующей главной оси. Например, двумерный поворот в плоскости XY относительно начала координат соответствует трехмерному повороту вокруг оси Z. Ниже приведены матрицы трехмерных поворотов:

- поворот вокруг оси Z:

$$R_z(\theta) = \begin{pmatrix} \cos\theta & \sin\theta & 0 & 0 \\ -\sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix},$$

- поворот вокруг оси X:

$$R_x(\theta) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & \sin\theta & 0 \\ 0 & -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix},$$

- поворот вокруг оси Y:

$$R_y(\theta) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

Как и в случае двух измерений, умножение вектора p на матрицу M выгодно производить с учетом ее структуры:

$$p' = pR + t.$$

Такое вычисление включает всего 9 умножений (против 16-ти при умножении без учета структуры матрицы M).

Упражнения

2-1. Напишите анимационную программу, которая отображает колесо, катящееся слева направо с равномерной скоростью. Угловая скорость вращения колеса должна соответствовать линейной скорости. Какие линейные преобразования описывают траекторию точки, расположенной на колесе? На оси колеса?

2-2. Поместите колесо из предыдущего упражнения на линию нижней полуокружности. Рассчитайте закон изменения скорости движения колеса при ускоренном движении слева направо и обратно.

2-3. Сделайте движения колеса из предыдущего упражнения затухающими.

2-4. Реализуйте твиннинг: плавное «превращение» одного векторного изображения в другое. Для этого вершине p каждого отрезка исходного изображения поставьте в соответствие вершину q второго изображения. Затем создайте серию из $n + 1$ кадра, в каждом из которых положение исходной вершины p перемещается по линейному закону в положение вершины q:

$$p_i = (q - p)i/n + p,\ i = 0, \ldots, n.$$

Глава 7

Построение плоских проекций

Основное внимание в этой главе будет уделено процессу формирования изображений трехмерных объектов. Процесс вывода трехмерной информации по существу более сложен, чем аналогичный двумерный процесс. В двумерном случае достаточно произвести отсечение в мировом окне и выполнить два параллельных переноса и масштабирование, переносящие плоский объект из окна в мировых координатах в окно на экране дисплея. В трехмерном же случае возникает проблема: у экрана дисплея на одно измерение меньше, чем у отображаемого объекта, поэтому таких преобразований, как перенос и масштабирование, недостаточно. Несоответствие между пространственными объектами и плоскими изображениями устраняется путем введения проекций, которые отображают трехмерные объекты на двумерную плоскость, называемую *картинной*.

Пусть заданы картинная плоскость и не принадлежащая ей точка c (рис. 7.1). *Проекцией* произвольной точки p в пространстве на

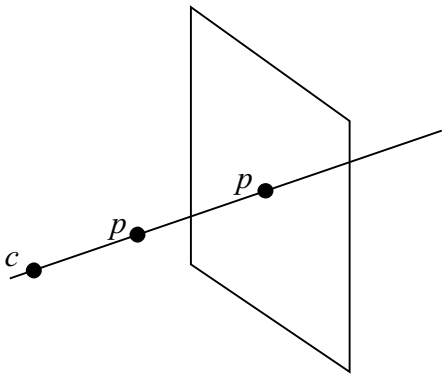


Рис. 7.1. Проекция точки на картинную плоскость

картинную плоскость с центром в точке c называется точка пересечения картинной плоскости $\mathcal{Z}$ с прямой, проходящей через p и c. Если прямая (p, c) параллельна картинной плоскости, то проекция не определена. Прямая (p, c) и точка c называются, соответственно, *проектором* и *центром проекции*.

Проекции на плоскость, использующие в качестве проекторов прямые, называются *плоскими геометрическими проекциями* (примером проекций другого типа являются географические карты). Если расстояние от центра проекции до проекционной плоскости конечно, то проекция называется *центральной*, в противном случае – *параллельной*. Поскольку проекции окружающих нас предметов на сетчатку глаза являются центральными (хотя и не плоскими!), мы воспринимаем центральную проекцию как более реалистичную.

7.1. Математическое описание проекций

Возникает вопрос: как формально описать проецирование? Будем исходить из предположения, что проецируемый объект задан своими координатами в правосторонней мировой системе координат. Наша цель – научиться изображать объекты, используя *произвольную* картинную плоскость и *произвольные* центр проекции (для центральных проекций) или направление проецирования (для параллельных проекций).

Построение плоских проекций напоминает процесс фотографирования. В фотоаппарате центром проекции является фокус объектива, в качестве картинной плоскости выступает поверхность фотопленки. Поскольку при проецировании картинная плоскость может быть расположена как перед, так и за центром проекции (с той лишь разницей, что при переходе через центр проекции изображение переворачивается на 180°), для упрощения вычислений картинную плоскость располагают перед центром проекции. В этом случае видимая в кадре область пространства, называемая *видимым объемом*, представляет собой четырехгранную пирамиду с вершиной в фокусе объектива и со сторонами, проходящими через вершины прямоугольного окна, ограничивающего область кадра (рис. 7.2). При увеличении фокусного расстояния пирамида вытягивается и в пределе превращается в параллелепипед. Задать проекцию – значит описать форму и пространственную ориентацию видимого объема.

Таким образом, исходными данными для проецирования являются:

- картинная плоскость;
- окно на картинной плоскости;
- центр проекции (для центральной проекции) или направление проецирования (для параллельной проекции).

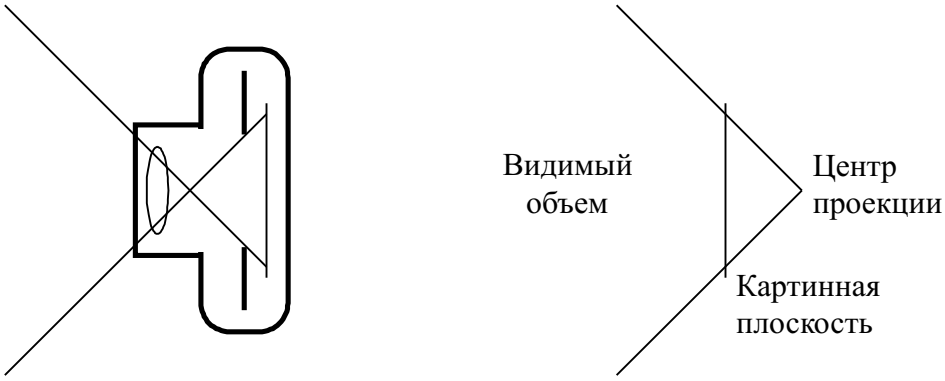


Рис. 7.2. Видимый объем

Картинную плоскость удобнее всего задавать через точку и вектор нормали (рис. 7.3). Обозначим через VRP некоторую точку на картинной плоскости, называемую *опорной* (view reference point), а через VPN – нормаль к картинной плоскости (view plane normal). Картинная плоскость может располагаться самым произвольным образом относительно проецируемых объектов: пересекать их, проходить спереди, сзади и т. д.

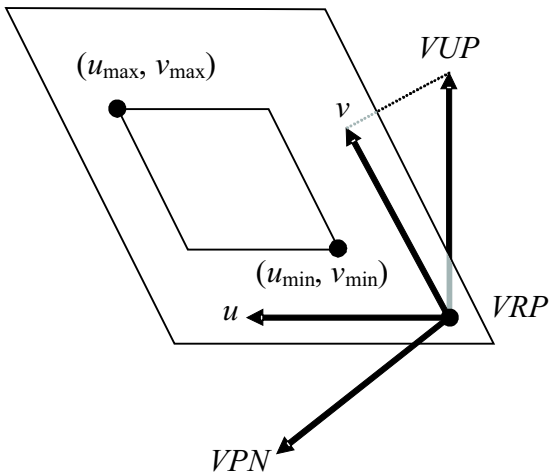


Рис. 7.3. Система видовых координат и окно на картинной плоскости

Окно задается в специальной системе координат на картинной плоскости, называемой *системой координат uv*. Начало этой системы координат расположено в точке VRP. Направление оси v (вертикальной оси) задается так называемым *вектором вертикали* VUP по следующему правилу: проекция вектора VUP на картинную плоскость совпадает с осью v. (Часто вектор вертикали полагают совпадающим по направлению с осью y мировой системы координат.) Ось u задается так, чтобы тройка векторов (u, v, VPN) задавала левостороннюю систему координат, называемую *видовой*.

После определения осей u и v окно на картинной плоскости задается координатами левой нижней и правой верхней вершин: $(u_{\min}, v_{\min})$ и $(u_{\max}, v_{\max})$.

Заметим, что опорная точка VRP и оба направляющих вектора – VPN и VUP – задаются в правосторонней мировой системе координат. Как мы уже говорили, видимый объем в случае *центральной* проекции определяется окном на картинной плоскости, центром проекции COP и *передней* и *задней секущими* плоскостями (рис. 7.4). Для параллельной проекции вместо центра проекции задается вектор DOP, указывающий направление проецирования (рис. 7.5).

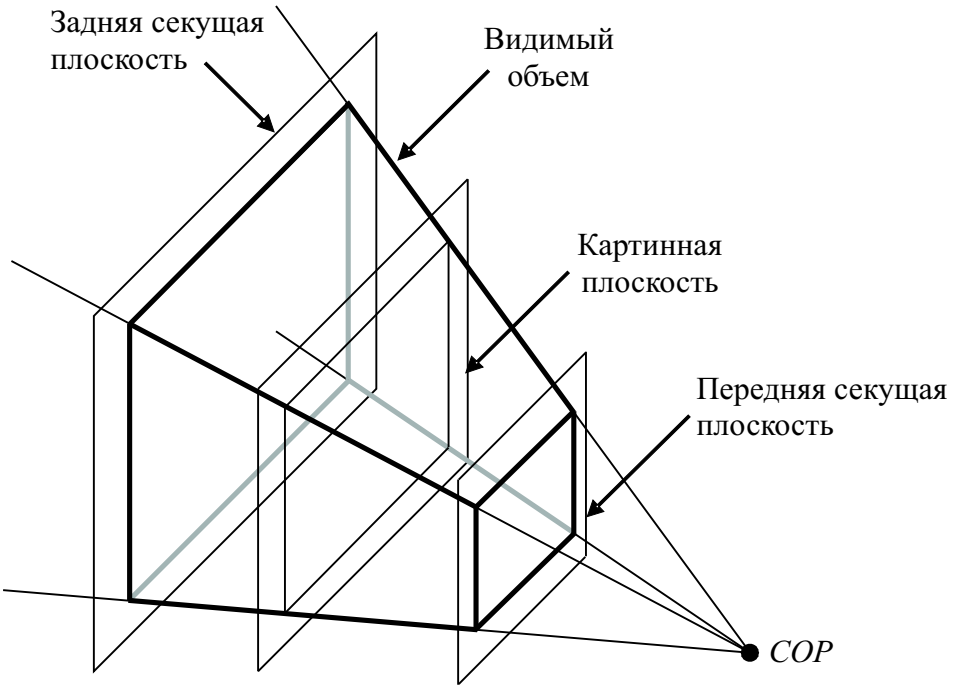


Рис. 7.4. Видимый объем для центральной проекции

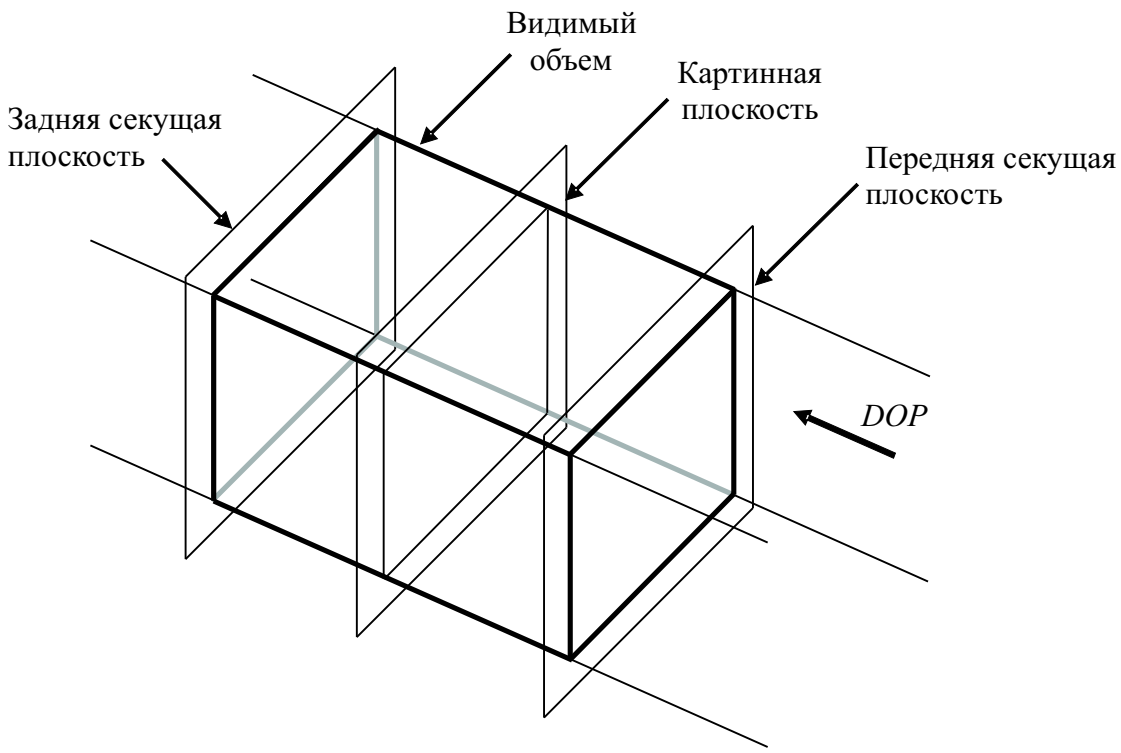


Рис. 7.5. Видимый объем для параллельной проекции

Ограничение видимого объема передней и задней секущей плоскостями производится для ограничения числа выводимых объектов, которое может оказаться слишком большим. Кроме того, для центральных проекций примитивы, расположенные слишком далеко от картинной плоскости, сливаются в одну точку, а расположенные слишком близко разрастаются до гигантских размеров и также не несут какой-либо информации об объекте. Передняя и задняя секущие плоскости параллельны картинной плоскости и задаются числами со знаком, соответственно, F и B, указывающими расстояния до нее, причем положительными считаются расстояния, задаваемые в направлении нормали к картинной плоскости.

Таким образом, видимый объем в случае центральной проекции представляет собой усеченную пирамиду, а в случае параллельной проекции – параллелепипед, со сторонами, параллельными вектору DOP, который в общем случае может оказаться непараллельным вектору нормали VPN.

Подведем итоги. Для задания произвольной проекции – параллельной или центральной – необходимо указать следующие параметры:

1. Опорную точку VRP.
2. Вектор нормали VPN.
3. Вектор вертикали VUP.
4. Координаты окна в системе координат uv: $(u_{\min}, v_{\min}), (u_{\max}, v_{\max})$.
5. Центр проекции COP (в случае центральной проекции) или вектор DOP, задающий направление проецирования (в случае параллельной проекции).
6. Числа F и B, задающие положения передней и задней секущих плоскостей.

7.2. Виды и свойства параллельных проекций

При параллельном проецировании параллельные прямые остаются параллельными. Вдоль координатных осей может происходить уменьшение длины. Углы сохраняются только на тех гранях проецируемого объекта, которые параллельны картинной плоскости. (Понятно поэтому, что чертежнику легче изображать параллельные проекции, чем центральные, однако для графической процедуры, вычисляющей проекцию, эти задания по сложности практически одинаковы.)

Параллельные проекции подразделяются на два основных класса: *ортографические* и *косоугольные*. Параллельные проекции, для которых нормаль к картинной плоскости параллельна направлению проецирования, называются ортографическими, в противном случае – косоугольными.

Ортографические проекции делятся на *ортогональные* (вид спереди, вид сбоку и вид сверху) и *аксонометрические*. При ортогональных проекциях (рис. 7.6) картинная плоскость перпендикулярна одной из главных координатных осей (т. е. направление проецирования задается одной из осей координат). Недостатком этих проекций является неясная (а иногда и неоднозначная) представимость формы объекта.

При аксонометрии картинная плоскость не перпендикулярна координатным осям. Здесь происходит укорачивание длин вдоль каждой координатной оси, и истинное расстояние всегда можно измерить, умножив длину проекции на соответствующий коэффициент (для каждой оси – свой).

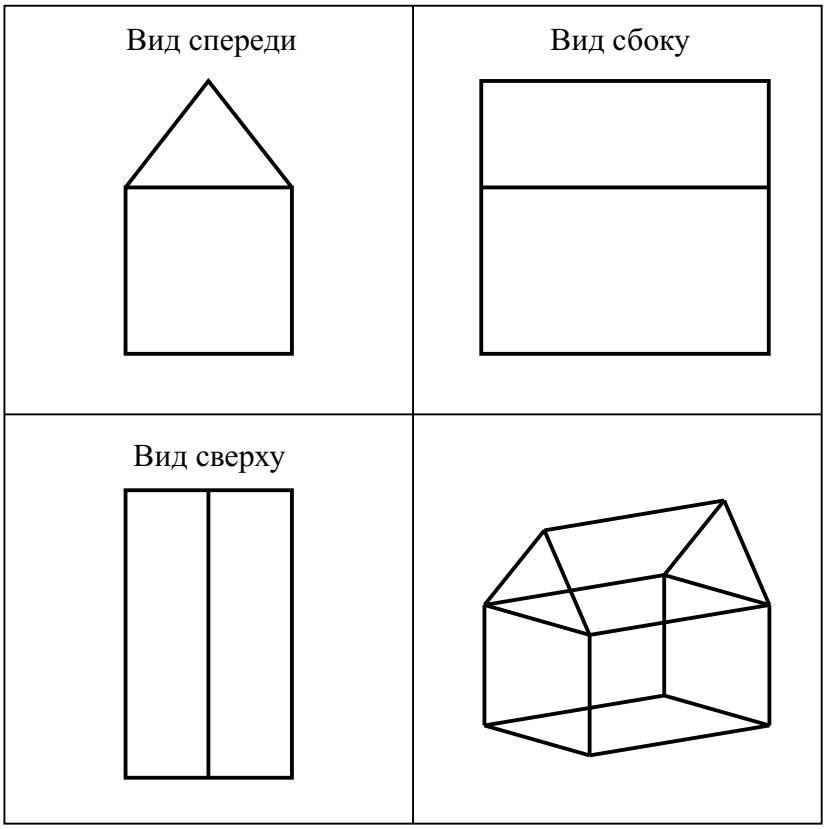


Рис. 7.6. Ортогональные проекции

Из бесконечного числа аксонометрий выделяют *изометрическую* проекцию (рис. 7.7), для которой нормаль к картинной плоскости параллельна вектору с координатами $(\pm 1, \pm 1, \pm 1)$. Существует, таким образом, ровно восемь направлений проецирования для изометрии – для каждого октанта системы координат. Здесь укорачивание длин одинаковое для всех трех координатных осей. В качестве упражнения читателю предлагается вычислить значение этого коэффициента.

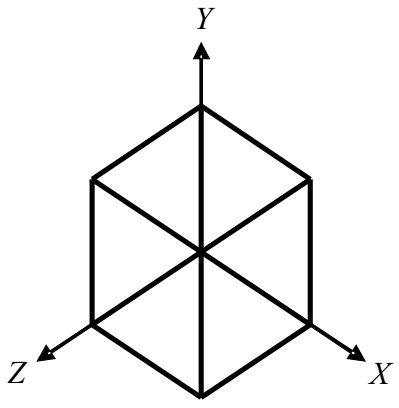


Рис. 7.7. Изометрия куба

Среди косоугольных проекций выделяют два вида: *кавалье* (cavalier) и *кабине* (cabinet). (В отечественной литературе употребляются названия, соответственно, *военная перспектива* и *кабинетная проекция*.) Для обоих видов косоугольной проекции картинная плоскость перпендикулярна главной

координатной оси, но направление проецирования не параллельно этой оси (рис. 7.8). Поэтому для граней объекта, параллельных картинной плоскости, длины и углы сохраняются, для других же граней для измерения длин необходимо производить масштабирование.

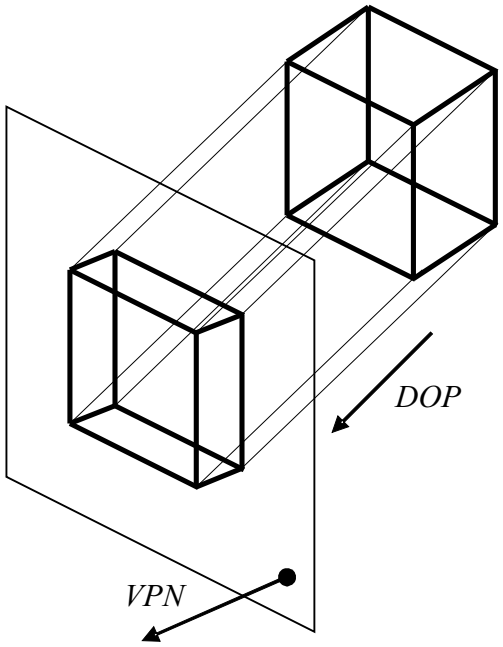


Рис. 7.8. Косоугольная проекция

Для проекции *кавалье* направление проецирования составляет 45° с картинной плоскостью, поэтому проекция отрезка, перпендикулярного картинной плоскости, имеет ту же длину. Ясно, что может быть бесконечно большое число проекций кавалье (как и число прямых, образующих 45° с картинной плоскостью), но в качестве стандарта приняты такие, для которых проекции прямых, перпендикулярных картинной плоскости, образуют с проекцией оси x углы 45° и 30°. Для проекции *кабине* направление проецирования составляет угол, равный $\mathrm{arcctg}(1/2)$. Следовательно, отрезки, перпендикулярные картинной плоскости, после проецирования уменьшаются вдвое (в этом смысле проекция кабине является более реалистической по сравнению с проекцией кавалье). В качестве стандартных проекций кабине приняты такие, для которых проекции отрезков также лежат под углом 45° и 30° к проекции оси x.

На рис. 7.9 приведены все описанные выше виды параллельных проекций и соответствующие им параметры (через Z обозначается картинная плоскость, вектора VPN и DOP считаются единичными).

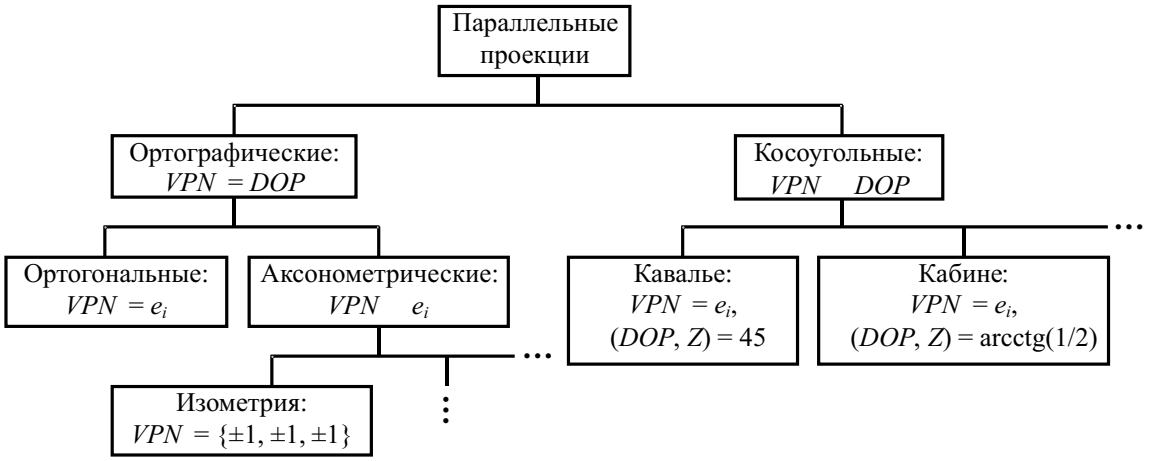


Рис. 7.9. Виды параллельных проекций

7.3. Виды и свойства центральных проекций

Центральные проекции порождают эффект, называемый перспективным укорачиванием, который состоит в том, что размер центральной проекции объекта обратно пропорционален расстоянию от центра проекции до объекта. Центральная проекция кажется более реалистичной, но не пригодна для передачи точной формы и размеров объекта. Углы сохраняются только на тех гранях, которые параллельны картинной плоскости. Проекции параллельных линий в общем случае непараллельны.

Нетрудно доказать, что центральные проекции произвольного множества параллельных прямых, не параллельных картинной плоскости, будут сходиться в точке на картинной плоскости, называемой *точкой схода*. Если эти прямые параллельны одной из главных координат осей, то их точка пересечения называется *главной точкой схода*. Таким образом, существуют ровно три такие точки. В зависимости от расположения картинной плоскости, а следовательно, и от числа главных точек схода центральные проекции делятся на *одноточечные, двухточечные* и *трехточечные* (рис. 7.10).

7.4. Вычисление проекций

Процесс вывода трехмерной графической информации предусматривает три этапа: отсечение элементов сцены относительно видимого объема, проецирование и переход к координатам физического устройства. Возникает вопрос: каким образом выполнить

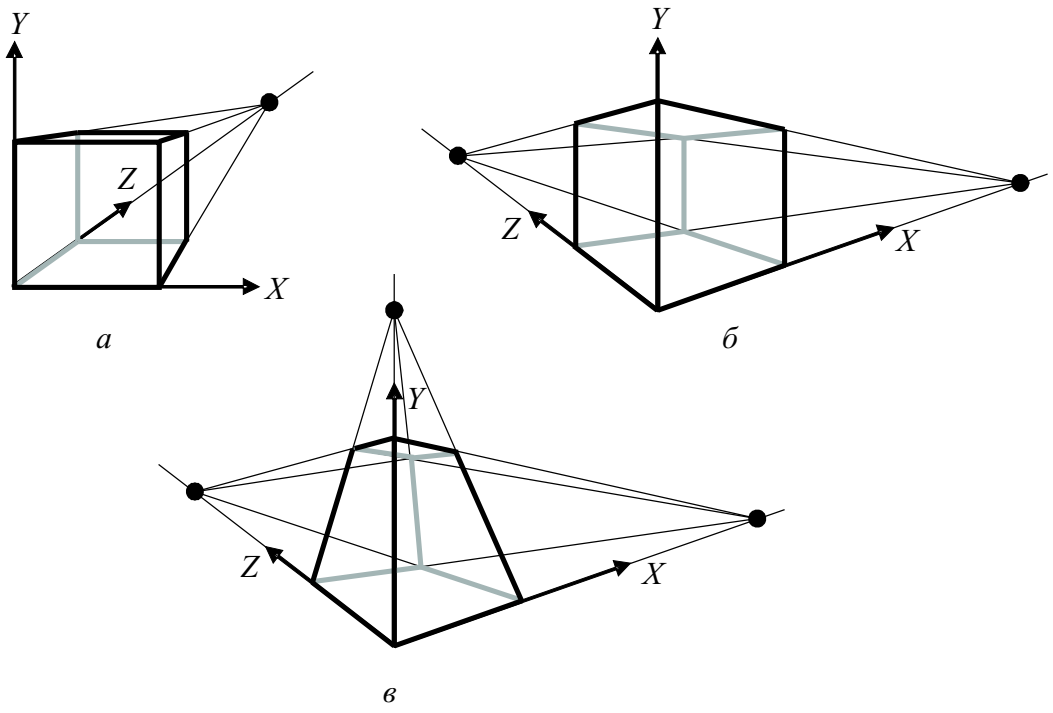


Рис. 7.10. Центральные проекции куба: *а* – одноточечная, *б* – двухточечная, *в* – трехточечная

эффективное отсечение объекта и вычислить координаты его проекции? Тривиальный подход состоит в том, чтобы сначала произвести отсечение отрезков и граней объекта относительно видимого объема, затем спроецировать их на картинную плоскость, вычисляя пересечения проекторов с картинной плоскостью, и наконец преобразовать координаты точек на картинной плоскости в двумерные координаты устройства. Нетрудно видеть, что при таком подходе потребуется большой объем вычислений, повторяемых для многих отрезков и граней.

Существует более эффективный способ построения проекции, идея которого состоит в следующем. С помощью геометрического преобразования, *не искажающего координаты проекции относительно системы координат uv на картинной плоскости*, видимый объем трансформируется таким образом, чтобы картинная плоскость совпала с плоскостью xy мировой системы координат, вектора u и v совпали по направлению с ортами e_1 и e_2, а проекторы стали параллельны оси z (рис. 7.11).

После такого преобразования проекция q' точки $p' = (x', y', z')$ будет иметь координаты $(x', y', 0)$, совпадающие с координатами

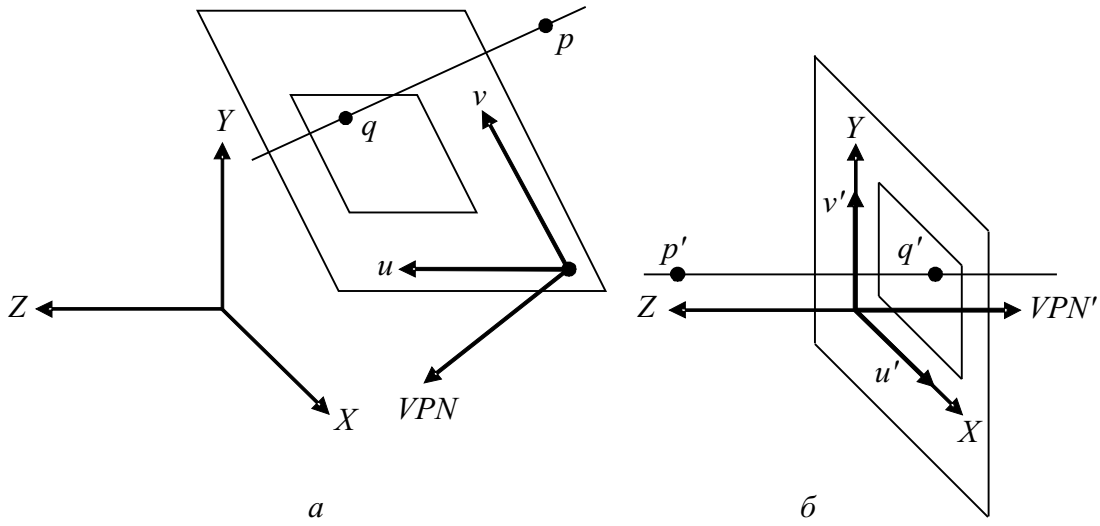


Рис. 7.11. Картинная плоскость и видовая система координат до (*а*) и после (*б*) нормирующего преобразования

(u, v) в видовой системе координат, и проецирование сведется просто к отбрасыванию нулевой z-координаты. Для центральных проекций требуются дополнительные несложные вычисления, но суть основного преобразования именно в этом. Описанное геометрическое преобразование называется *нормирующим*. В результате его применения видимый объем в случае параллельной проекции преобразуется в единичный куб, а в случае центральной – в правильную усеченную пирамиду, рассмотренную ранее в разделе 4.2.2. Такие регулярные видимые объемы называются *каноническими*. Помимо упрощения проецирования они позволяют также применить эффективные алгоритмы отсечения относительно регулярных областей.

Прежде чем приступить к описанию нормирующего преобразования, рассмотрим свойства матрицы, реализующей композицию трехмерных поворотов, лежащих в основе нормирующего преобразования.

7.4.1. Свойства матрицы поворота

Результатом произвольной последовательности трехмерных поворотов является матрица вида

$$A = \begin{pmatrix} & & & 0 \\ & R & & 0 \\ & & & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

При этом ее подматрица

$$R = \begin{pmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{pmatrix}$$

обладает следующими свойствами:

1. Столбцы $r_x = (r_{11}, r_{21}, r_{31})$, $r_y = (r_{12}, r_{22}, r_{32})$ и $r_z = (r_{13}, r_{23}, r_{33})$ матрицы R являются взаимно ортогональными единичными векторами. Матрицы, обладающие таким свойством, называются *ортогональными*.

2. При повороте, заданном матрицей R, вектора r_x, r_y и r_z отобразятся, соответственно, в единичные орты e_1, e_2 и e_3 главных координатных осей x, y и z. Это свойство оказывается полезным, если необходимо построить матрицу поворота, результат которого известен заранее.

Для ортогональной матрицы R справедливо равенство $R^{-1} = R^T$. В частности, для матрицы R двумерного поворота

$$R^{-1}(\theta) = R^T(\theta) = R(-\theta).$$

Рассмотрим преобразование поворота, применяемое при вычислении проекций. Пусть даны опорная точка VRP и два вектора: VPN и VUP (рис. 7.12, *а*). В соответствии с описанными выше преобразованиями видимого объема необходимо выполнить следующие действия:

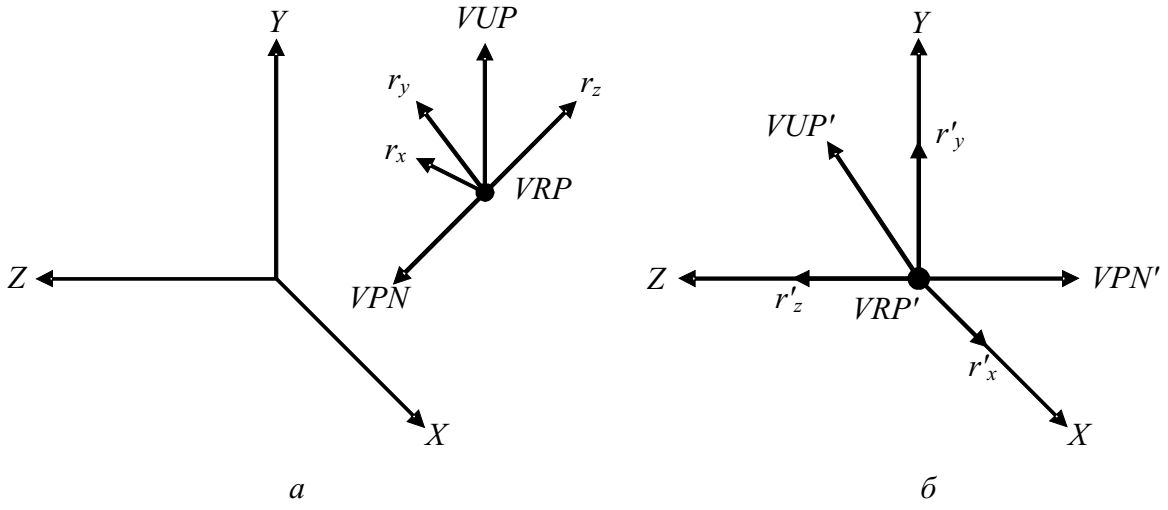


Рис. 7.12. Точка VRP и вектора VPN и VUP до (*а*) и после (*б*) нормирующего преобразования

1) перенести точку VRP в начало координат;

2) произвести поворот, в результате которого вектор VPN совместится с вектором $-e_3$, а вектор VUP окажется в неотрицательной полуплоскости YZ (рис. 7.12, *б*).

Поскольку результат преобразования известен, для вычисления матрицы поворота можно воспользоваться свойством ортогональных матриц, согласно которому достаточно вычислить ее вектора-столбцы r_x, r_y и r_z, переходящие после преобразования в единичные орты e_1, e_2 и e_3.

Третий столбец матрицы R – вектор r_z – получаем непосредственно:

$$r_z = -\frac{VPN}{||VPN||} \quad - \quad \text{единичный направляющий орт, который необходимо совместить с } e_3.$$

Как следует из условия задачи, вектор r_x должен быть ортогонален обоим векторам VPN и VUP. Следовательно, его можно вычислить как векторное произведение этих векторов:

$$r_x = \frac{VPN \times VUP}{||VPN \times VUP||} \quad - \quad \text{единичный вектор, перпендикулярный плоскости векторов } VPN \text{ и } VUP, \text{ который необходимо совместить с } e_1.$$

И наконец, $r_y = r_x \times r_z$. Заметим, что знак векторного произведения при вычислении r_x выбирался так, чтобы вектора (r_x, r_y, r_z) образовывали *левую* тройку.

Таким образом, искомая матрица вычисляется как произведение следующих двух матриц:

$$M = T(-VRP) \cdot \begin{pmatrix} r_{x1} & r_{y1} & r_{z1} \\ r_{x2} & r_{y2} & r_{z2} \\ r_{x3} & r_{y3} & r_{z3} \end{pmatrix} = T \cdot R.$$

Перейдем теперь к вычислению параллельных проекций. Напомним, что оно выполняется в три этапа: вычисление и применение нормирующего преобразования, отсечение по границам канонического объема и переход к координатам физического устройства. Рассмотрим подробно каждый из этих этапов.

7.4.2. Вычисление параллельной проекции

Наша задача – найти матрицу $N_{\text{парал}}$ нормирующего преобразования, переводящего произвольный видимый объем в единичный куб

$$\begin{aligned} 0 \leq\ & x\ \leq 1, \\ 0 \leq\ & y\ \leq 1, \\ 0 \leq\ & z\ \leq 1. \end{aligned} \tag{7.1}$$

На рис. 7.13 изображен видимый объем до начала преобразования.

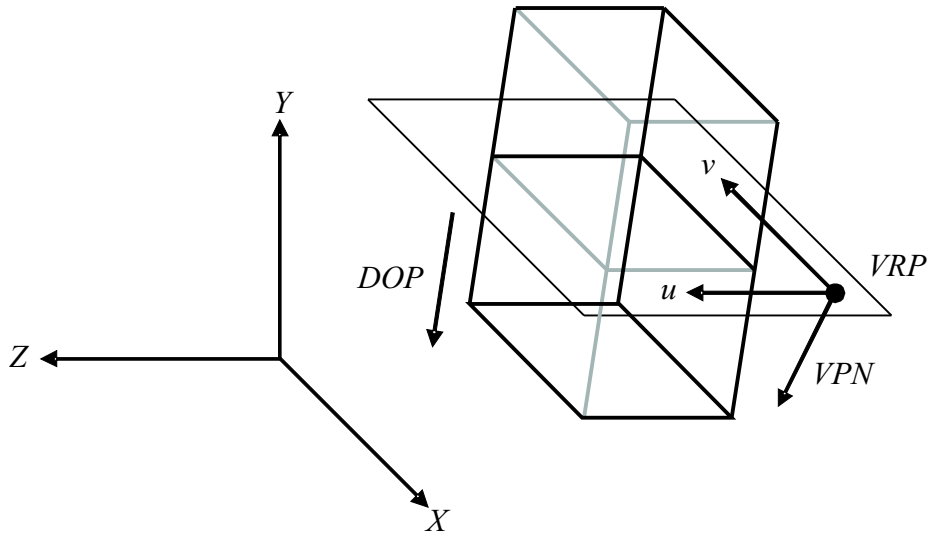


Рис. 7.13. Видимый объем до применения нормирующего преобразования

Это преобразование реализуется за несколько шагов:

1. Перенос точки VRP в начало мировой системы координат.
2. Поворот, в результате которого вектор VPN совпадет по направлению с вектором $-e_3$, а проекция вектора VUP на картинную плоскость совпадает по направлению с e_2.
3. Переход от правосторонней (мировой) системы координат к левосторонней (видовой).
4. Сдвиг, в результате которого вектор DOP станет параллельным вектору VPN.
5. Перенос и масштабирование видимого объема для преобразования его в единичный куб (7.1).

Первые два шага рассмотрены в разделе 7.4.1. Они реализуются умножением на произведение матриц $T(-VRP) \cdot R$.

Шаг 3 – переход от мировой системы координат к видовой – выполняет зеркальное отображении видимого объема относительно плоскости XY. Матрица этого преобразования следующая:

$$T_{\text{пл}} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}. \tag{7.2}$$

Обозначим через DOP' вектор направления проецирования после применения к нему преобразований R и $T_{\text{пл}}$: $DOP' = DOP \cdot R \cdot T_{\text{пл}}$ (рис. 7.14, *а*). В преобразование DOP не входит начальный параллельный перенос, поскольку преобразуется направление, а не положение

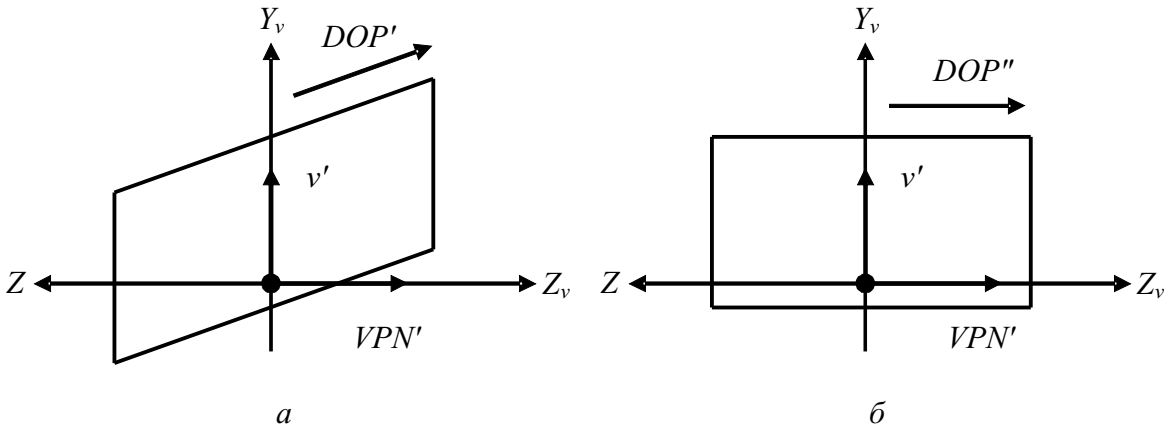


Рис. 7.14. Проекция видимого объема на плоскость $Y_v\ Z_v$ до (*а*) и после (*б*) применения шага 4

вектора. На шаге 4 выполняется *сдвиг* видимого объема вдоль оси Z_v, в результате которого все его грани, параллельные DOP', становятся параллельными оси Z_v. В общем виде матрица сдвига имеет вид

$$SH_z(a,b) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ a & b & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

Чтобы найти a и b, заметим, что в результате преобразования сдвига вектор DOP' становится параллельным оси Z_v. Отсюда получаем уравнение

$$DOP' \cdot SH_z(a,b) = (0,0,DOP'_z,1) \quad \text{или}$$

$$a = -\frac{DOP'_x}{DOP'_z}, \quad b = -\frac{DOP'_y}{DOP'_z}.$$

Заметим, что для ортографической проекции преобразования сдвига не требуется, поскольку вектора DOP и VPN параллельны, и матрица SH_z превращается в единичную. Результат выполнения шага 4 показан на рис. 7.14, *б*.

Шаг 5 предусматривает перенос и масштабирование видимого объема с целью преобразования его в единичный куб (7.1). После сдвига видимый объем определяется следующей системой неравенств:

$$\begin{aligned} u_{\min} \le\ & x\ \le u_{\max}, \\ v_{\min} \le\ & y\ \le v_{\max}, \\ F \le\ & z\ \le B. \end{aligned}$$

Параллельный перенос, совмещающий левый нижний передний угол видимого объема с началом координат, реализуется матрицей

$$T_{\text{парал}} = T(-u_{\min}, -v_{\min}, -F),$$

а масштабирование, переводящее видимый объем в единичный куб:

$$S_{\text{парал}} = S\left(\frac{1}{u_{\max} - u_{\min}}, \frac{1}{v_{\max} - v_{\min}}, \frac{1}{B - F}\right).$$

Результат выполнения шага 5 показан на рис. 7.15.

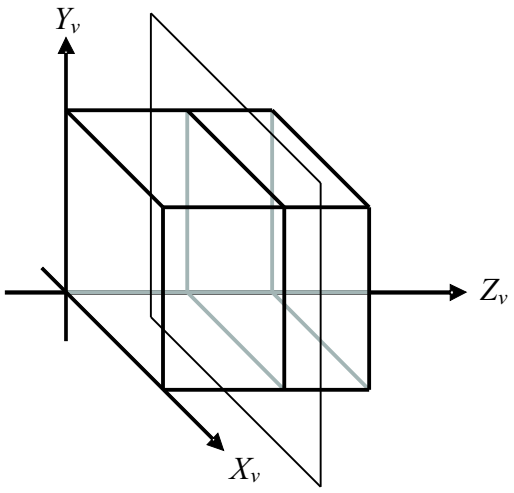


Рис. 7.15. Видимый объем после выполнения шага 5

Объединив все перечисленные преобразования, получаем матрицу нормирующего преобразования:

$$N_{\text{парал}} = T \cdot R \cdot T_{\text{пл}} \cdot SH_z(a,b) \cdot T_{\text{парал}} \cdot S_{\text{парал}}.$$

7.4.3. Вычисление центральных проекций

В случае центральной проекции видимый объем необходимо преобразовать к каноническому объему, определяемому неравенствами:

$$\begin{aligned} -z \le\ & x\ \le z, \\ -z \le\ & y\ \le z, \\ z_{\min} \le\ & z\ \le 1. \end{aligned} \tag{7.3}$$

На рис. 7.16 изображен видимый объем для центральной проекции до начала преобразований.

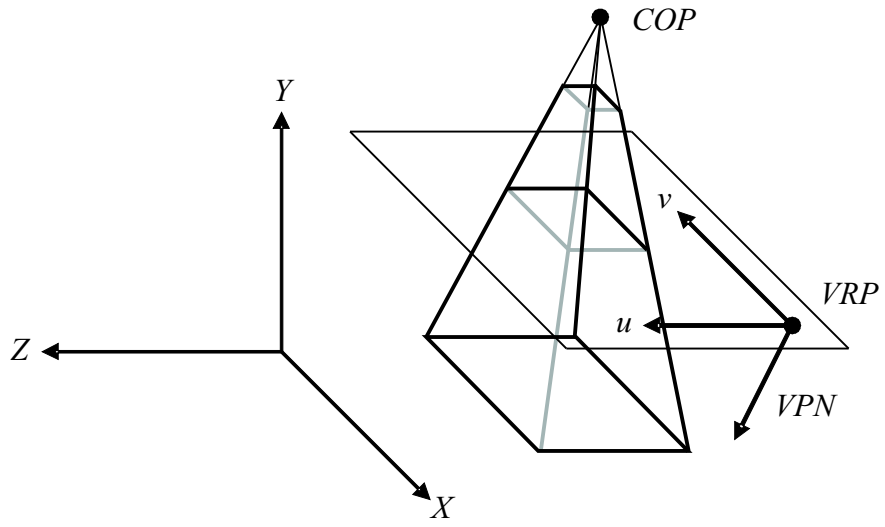


Рис. 7.16. Видимый объем до применения нормирующего преобразования

Вычисление нормирующего преобразования, задаваемого матрицей $N_{\text{центр}}$, также проводится за пять шагов:

1. Перенос центра проекции в начало проекции координат.
2. Поворот, в результате которого вектор VPN совпадет по направлению с вектором $-e_3$, а проекция вектора VUP на картинную плоскость совпадает по направлению с e_2.
3. Переход от правосторонней (мировой) системы координат к левосторонней (видовой).

4. Сдвиг, при котором ось видимого объема становится параллельной вектору VPN.

5. Масштабирование видимого объема, при котором видимый объем принимает вид усеченной прямоугольной пирамиды, определяемой неравенствами (7.3).

Шаги 1 и 2 реализуются умножением на матрицу $T(-COP) \cdot R$. Переход к видовой системе координат (X_v, Y_v, Z_v) реализуется умножением на матрицу $T_{\text{пл}}$ (7.2).

После указанных преобразований ось видимого объема, т. е. ось пирамиды, проходящая через начало координат и центр окна, в общем случае не совпадает с осью z_v (рис. 7.17, *а*). Целью сдвига является

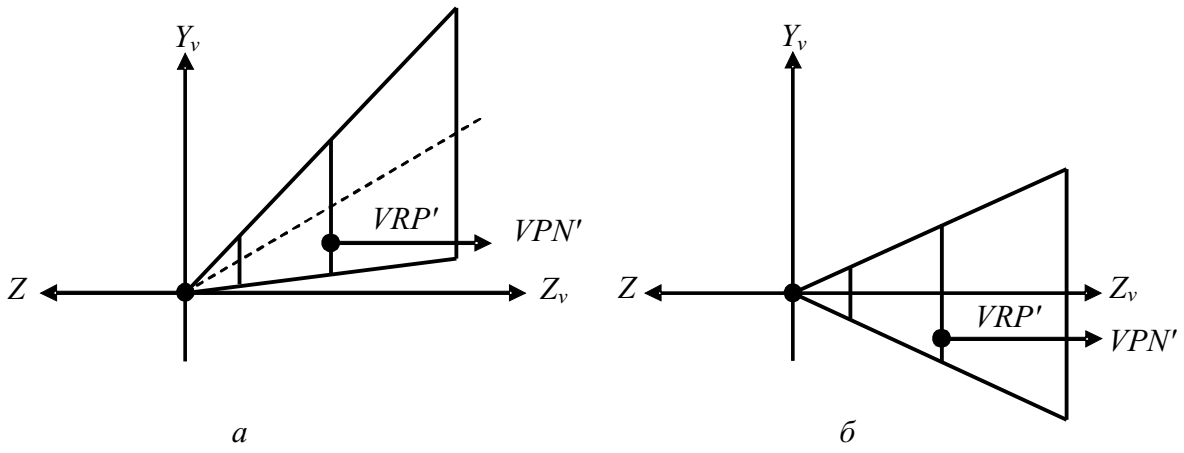


Рис. 7.17. Видимый объем до (*а*) и после (*б*) преобразования сдвига

совмещение оси видимого объема с осью z_v. Заметим, что в результате сделанных преобразований координаты окна в системе координат uv не изменились, поэтому координаты его центра в этой системе равны

$$\left(\frac{u_{\min} + u_{\max}}{2}, \frac{v_{\min} + v_{\max}}{2} \right).$$

Положение же начала координат системы uv, т. е. вектора VRP, относительно мировой системы координат изменилось и равно

$$VRP' = VRP \cdot T(-COP) \cdot R \cdot T_{\text{пл}}.$$

Таким образом, ось видимого объема проходит через точку с координатами

$$W_x^c = VRP'_x + \frac{u_{\min} + u_{\max}}{2},$$

$$\begin{aligned} W_y^c &= VRP'_y + \frac{v_{\min} + v_{\max}}{2}, \\ W_z^c &= VRP'_z. \end{aligned}$$

Для совмещения центра окна с точкой $(0, 0, VRP'_z)$ применим преобразование сдвига, задаваемое матрицей $Sh(a, b)$ с коэффициентами

$$a = -\frac{W_x^c}{W_z^c} \quad \text{и} \quad b = -\frac{W_y^c}{W_z^c}.$$

В результате применения преобразования сдвига окно (а следовательно, и видимый объем) центрируются относительно оси z (рис. 7.17, *б*).

На шаге 5 производится масштабирование вдоль всех трех осей, выполняемое в два этапа:

1. Масштабирование по осям x_v и y_v для совмещения верхних и боковых плоскостей видимого объема с плоскостями $x_v = z_v$ и $y_v = z_v$.
2. Однородное масштабирование вдоль всех трех осей для совмещения задней секущей плоскости видимого объема с плоскостью $z_v = 1$.

Из рис. 7.17, *б* видно, что в результате первого масштабирования полширины окна по осям x_v и y_v должны стать равными VRP'_z. Таким образом, масштабные коэффициенты по осям x_v и y_v соответственно равны

$$\frac{2VRP'_z}{(u_{\max} - u_{\min})} \quad \text{и} \quad \frac{2VRP'_z}{(v_{\max} - v_{\min})}.$$

Для выполнения второго этапа необходимо, чтобы задняя секущая плоскость, ограничивающая видимый объем, совпала с плоскостью $z_v = 1$ и при этом наклон боковых сторон не изменился. Этого можно добиться с помощью однородного масштабирования с коэффициентом $1/(VRP'_z + B)$. Таким образом, результирующая матрица масштабирования $S_{\text{центр}}$ будет иметь коэффициенты:

$$\begin{aligned} S_x &= \frac{2VRP'_z}{(u_{\max} - u_{\min})(VRP'_z + B)}, \\ S_y &= \frac{2VRP'_z}{(v_{\max} - v_{\min})(VRP'_z + B)}, \\ S_z &= \frac{1}{(VRP'_z + B)}. \end{aligned}$$

После такого масштабирования передняя секущая плоскость и картинная плоскость перейдут в новые положения

$$z_{\min} = \frac{VRP'_z + F}{VRP'_z + B} \quad \text{и} \quad z_{\text{карт}} = \frac{VRP'_z}{VRP'_z + B},$$

а видимый объем преобразуется в правильную усеченную пирамиду, ограниченную плоскостями (7.3) (рис. 7.18).

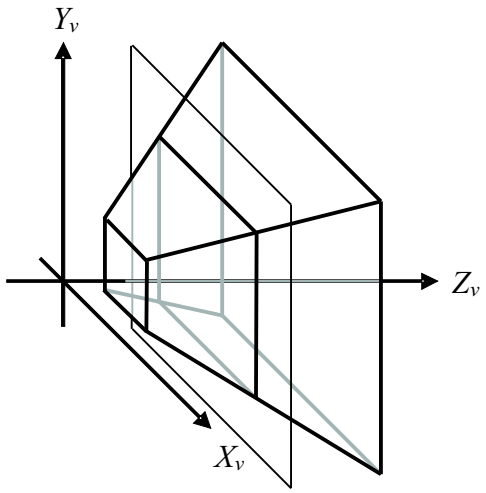


Рис. 7.18. Видимый объем после применения нормирующего преобразования

В итоге получаем матрицу нормирующего преобразования для случая центральной проекции

$$N_{\text{центр}} = T(-COP) \cdot R \cdot T_{\text{пл}} \cdot Sh_z \cdot S_{\text{центр}}.$$

7.4.4. Отсечение по границе канонического объема

Второй этап вычисления параллельных проекций предполагает отсечение объекта по границе видимого объема. Здесь используется любой из известных алгоритмов отсечения, рассмотренный в разделе 4.2. При этом в зависимости от представления объекта (в виде каркасной модели или сплошной поверхности) производится отсечение отрезков или граней. Следует помнить, что для корректного удаления невидимых поверхностей отсечение необходимо проводить именно на этом этапе, особенно если камера расположена *внутри* объекта (например, при моделировании внутреннего осмотра помещения).

7.4.5. Переход к координатам физического устройства

Преобразование в физические координаты проводится в три этапа.

1. Для центральных проекций выполняется преобразование *центрального проецирования* на картинную плоскость. Координаты проекции $p' = (x'_v, y'_v, z'_v)$ точки $p = (x_v, y_v, z_v)$ на картинную плоскость вычисляются из отношений для подобных треугольников (рис. 7.19):

$$\frac{x'_v}{z_{\text{карт}}} = \frac{x_v}{z_v}, \quad \frac{y'_v}{z_{\text{карт}}} = \frac{y_v}{z_v} \quad \text{или}$$

$$x'_v = x_v \frac{z_{\text{карт}}}{z_v}, \quad y'_v = y_v \frac{z_{\text{карт}}}{z_v}.$$

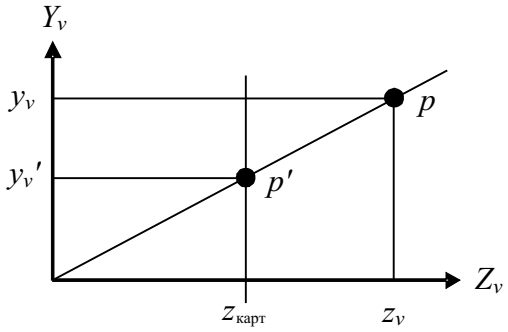


Рис. 7.19. Центральное проецирование на картинную плоскость

Это преобразование можно представить в виде матрицы

$$M_{\text{центр}} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & \dfrac{1}{z_{\text{карт}}} \\ 0 & 0 & 0 & 0 \end{pmatrix}.$$

Результат умножения точки p на матрицу $M_{\text{центр}}$ даст точку

$$(x_v, y_v, z_v, 1) \cdot M_{\text{центр}} = \left(x_v, y_v, z_v, \frac{z_v}{z_{\text{карт}}} \right),$$

каноническое представление которой имеет вид

$$(x'_v, y'_v, z'_v, 1) = \left(x_v \frac{z_{\text{карт}}}{z_v},\ y_v \frac{z_{\text{карт}}}{z_v},\ \ z_{\text{карт}},\ 1 \right).$$

Преобразование центрального проецирования отображает все точки видимого объема на картинную плоскость. При этом теряется информация о глубине изображения, которая используется при удалении невидимых линий и поверхностей. Чтобы сохранить значения глубины, вместо матрицы $M_{\text{центр}}$ центрального проецирования применяют матрицу M *перспективного преобразования*:

$$M = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \dfrac{1}{1-z_{\min}} & 1 \\ 0 & 0 & \dfrac{-z_{\min}}{1-z_{\min}} & 0 \end{pmatrix}.$$

Матрица M отображает точку $(x_v, y_v, z_v, 1)$ в точку с координатами

$$(x'_v, y'_v, z'_v, 1) = \left(\frac{x_v}{z_v},\ \frac{y_v}{z_v},\ \frac{z_v - z_{\min}}{z_v(1-z_{\min})},\ 1\right),$$

видимый объем (7.3) преобразует в объем, описываемый неравенствами

$$\begin{aligned} -1 \le\ & x_v \le 1, \\ -1 \le\ & y_v \le 1, \\ 0 \le\ & z_v \le 1, \end{aligned} \tag{7.4}$$

а центр проекции переводит в бесконечно удаленную точку на отрицательной полуоси Z_v (рис. 7.20). Произведение матриц $M \cdot T(1,1,0) \cdot S\left(\frac{1}{2}, \frac{1}{2}, 1\right)$ преобразует видимый объем (7.4) в единичный куб (7.1).

2. *Ортографическое проецирование* на плоскость $z_v = 0$, при котором точка $p = (x_v, y_v, z_v)$ отображается в точку $p' = (x_v, y_v, 0)$. По сути, на этом шаге происходит переход от трехмерных координат (x_v, y_v, z_v) к координатам (x_v, y_v) на картинной плоскости.

3. *Преобразование в физические координаты.* После ортографического проецирования все части объекта, попавшие в видимый объем, переводятся в двумерные отрезки и многоугольники в единичном квадрате, то есть представляются в нормированных

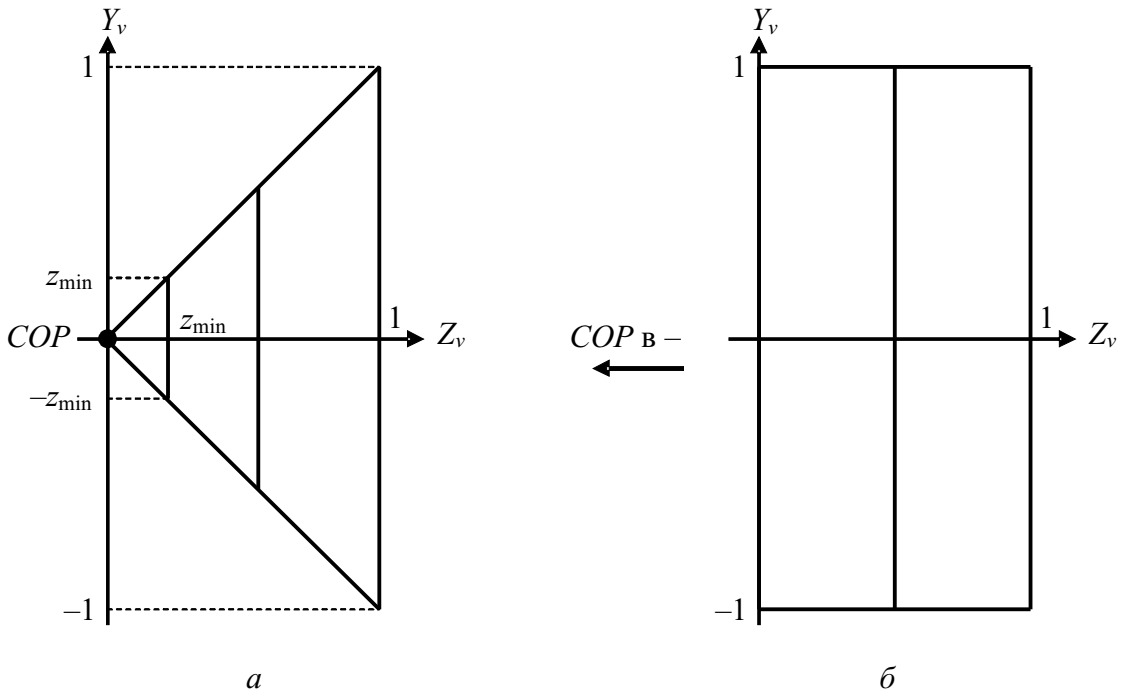


Рис. 7.20. Видимый объем для центральной проекции до (*а*) и после применения перспективного преобразования (*б*)

координатах. После этого изображение может быть выведено в некоторое окно на плоттере, экране дисплея и т. д. с помощью композиции преобразований масштабирования и переноса

$$S(x_{\max} - x_{\min}, y_{\max} - y_{\min}) \cdot T(-x_{\min}, -x_{\min}).$$

Здесь $(x_{\min}, y_{\min}), (x_{\max}, y_{\max})$ – координаты окна в поле вывода физического устройства. Заметим, что если окно в поле вывода пропорционально окну, задающему видимый объем, то изображение строится без искажений.

Упражнения

2-1. Напишите программу, моделирующую облет произвольного предмета на заданной высоте по окружности с заданным радиусом вокруг оси Y. Для изображения предмета используйте центральную проекцию с центром в точке, в которой находится наблюдатель.

2-2. В предыдущем упражнении замените центральную проекцию на проекцию кавалье или кабине.

Глава 8

Удаление невидимых поверхностей

В процессе формирования изображения трехмерный объект может быть расположен таким образом, что часть его поверхности оказывается заслоненной от наблюдателя непрозрачными поверхностями или другими объектами. Если невидимые поверхности отображаются вместе с видимыми, то это приводит к неоднозначности в представлении объектов сцены, не говоря уже о потере реалистичности. Пример на рис. 8.1 демонстрирует такую неоднозначность.

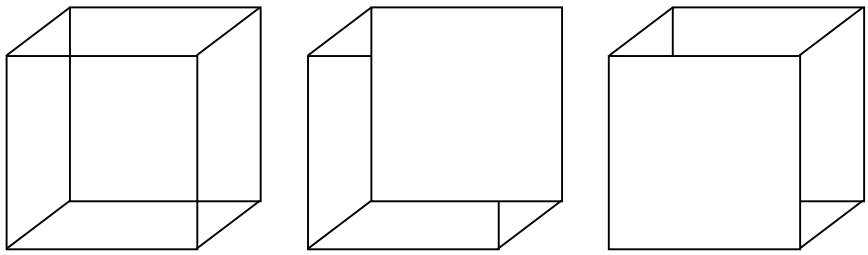


Рис. 8.1. Удаление невидимых поверхностей позволяет избавиться от неоднозначности в представлении объекта

Было время, когда алгоритмы удаления невидимых поверхностей представляли собой узкое место большинства графических систем: на них приходился максимум вычислительных затрат при генерации изображений. Архитектура современных компьютеров позволяет перенести значительный объем вычислений на аппаратный уровень и тем самым снижает остроту проблемы. Тем не менее с точки зрения техники вычислений эффективное удаление невидимых поверхностей остается наиболее сложной задачей, достойной отдельного изучения.

8.1. Алгоритм сортировки по глубине

Алгоритмы удаления невидимых поверхностей могут работать в различных системах координат. Часть действий выполняется в видовой системе координат непосредственно после применения нормирующего преобразования $N_{\text{парал}}$ или $N_{\text{центр}}$, другая часть – в системе координат физического устройства непосредственно в ходе вывода изображения.

Алгоритм сортировки по глубине работает в пространстве изображения и в пространстве объекта. Общая его схема следующая:

1. Грани всех объектов сцены упорядочиваются по убыванию их максимальных z_v-координат.

2. Упорядоченный список просматривается, некоторые грани меняются местами в списке и, при необходимости, разбиваются на части с целью разрешения неопределенностей, возникающих при пересечении проекций граней.

3. Проекции граней преобразуются в растровую форму в порядке их следования в списке.

Таким образом, основная идея состоит в том, чтобы производить закраску граней по мере их приближения к наблюдателю. При этом более близкие грани закрасят более удаленные, удалив тем самым невидимые поверхности. Очевидно, такой способ возможен только на растровых устройствах с регенерацией изображения.

Простое упорядочение на первом шаге еще не гарантирует, что данный многоугольник не закрывает последующий. Примеры подобных неопределенностей представлены на рис. 8.2. Для их разрешения на втором шаге алгоритма производятся дополнительные сравнения многоугольников. Для уменьшения операций сравнения вводится понятие *оболочки многоугольника*. Обозначим через $R(P)$ минимальный охватывающий параллелепипед многоугольника P со сторонами, параллельными координатным осям. *X-оболочкой* многоугольника P называется множество точек (x_v, y_v, z_v) таких, что

$$R_x^{\min}(P) \le x_v \le R_x^{\max}(P),$$

где $R_x^{\min}(P)$ и $R_x^{\max}(P)$ обозначают минимальную и максимальную x_v-координату параллелепипеда $R(P)$ (рис. 8.3). Аналогично вводится понятие Y- и Z-оболочки. С использованием этих оболочек второй шаг алгоритма состоит в следующем: каждый многоугольник P,

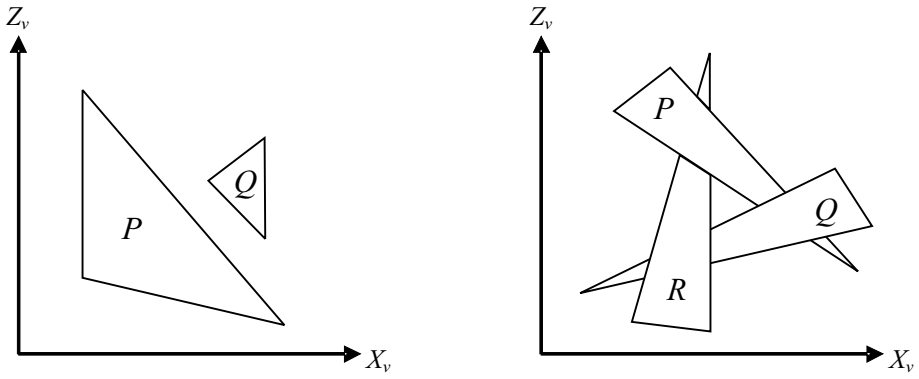


Рис. 8.2. Упорядочение по координате z_v не гарантирует линейного упорядочения многоугольников относительно наблюдателя

начиная с первого в списке S, сравнивается со всеми последующими многоугольниками, Z-оболочка которых пересекает Z-оболочку P. Сравнение двух многоугольников P и Q включает пять тестов, которые выполняются в порядке возрастания сложности:

1. $XH(P) \bigcap XH(Q) = \emptyset$.
2. $YH(P) \bigcap YH(Q) = \emptyset$.
3. Многоугольник P целиком лежит с той стороны от плоскости многоугольника Q, которая дальше от наблюдателя (рис. 8.4, *а*).
4. Многоугольник Q целиком лежит с той стороны от плоскости многоугольника P, которая ближе к наблюдателю (рис. 8.4, *б*).

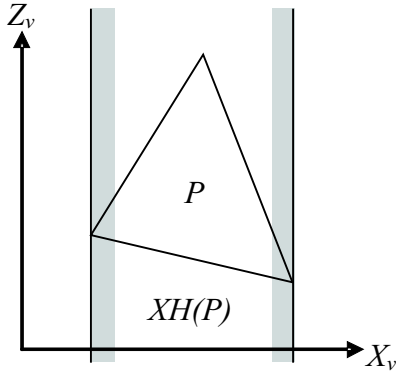
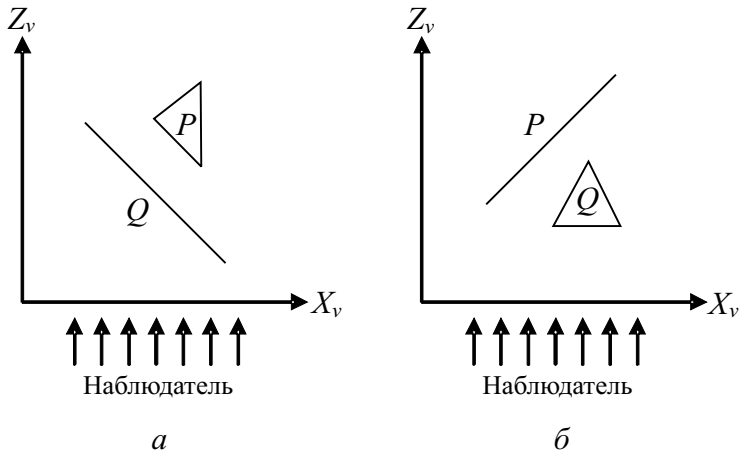


Рис. 8.3. X-оболочка многоугольника



а *б*

Рис. 8.4. Взаимное расположение граней P и Q, при котором выполняются тесты 3 (*а*) и 4 (*б*). Обратите внимание, что в случае *б* тест 3 не выполняется

5. Проекции многоугольников на плоскость X_vY_v не пересекаются (самый трудоемкий тест).

Если во всех пяти тестах получен отрицательный ответ, то предполагается, что многоугольник P действительно закрывает многоугольник Q и должен, таким образом, рисоваться первым. В этом случае многоугольники меняются местами в списке S.

При этом не исключена следующая коллизия: поскольку не существует плоскости, разделяющей P и Q, то повторное сравнение этих многоугольников может привести к зацикливанию алгоритма. Его можно избежать, если запретить повторное перемещение многоугольника, уже однажды помещенного в начало списка, пометив его. Вместо повторного перемещения помеченный многоугольник P рассекается плоскостью, содержащей Q, на два многоугольника P' и P'', где P' и наблюдатель расположены по разные стороны относительно плоскости многоугольника Q. Таким образом, P' закрывается многоугольником Q и должен рисоваться первым, а P'' вставляется в список S в соответствие с его максимальной z_v-координатой.

Ниже представлена схема алгоритма, выполняющего сортировку граней. Грани хранятся в приоритетной очереди F. Функция ZHullsOverlap(P, Q) возвращает TRUE, если пересекаются Z-оболочки граней P и Q. Все пять тестов выполняются с помощью функции Test(P, Q). Процедура SplitByPlane(P, Q, P_1, P_2) выполняет разбиение многоугольника P плоскостью многоугольника Q на две части – P_1 и P_2.

```
SortByDepth(F)
 1  Отсортировать грани по убыванию максимальной
    z_v-координаты и поместить их в очередь F
 2  P ← head[F]
 3  while P ≠ nil do
 4      Q ← next[P]
 5      while Q ≠ nil and ZHullsOverlap(P, Q) do
 6          if Test(P, Q) then
 7              Q ← next[Q]
 8          else if mark[Q] = false then
10              mark[Q] ← true
11              Swap(F, P, Q)
12              Q ← next[P]
13          else
14              SplitByPlane(P, Q, P_1, P_2)
15              P ← P_1
16              Insert(F, P_2)
17              Q ← next[Q]
18      P ← next[P]
```

Результатом работы алгоритма является упорядоченный список проекций граней, которые затем преобразуются в растровую форму, начиная с начала списка. Такая последовательность закраски обладает существенным недостатком: один и тот же пиксел может закрашиваться несколько раз. Это негативно сказывается на скорости алгоритма, особенно при построении реалистических изображений, когда процедура формирования цвета пиксела является поэтому весьма дорогостоящей операцией, включающей учет освещенности, оптических свойств поверхности грани и атмосферы.

Существует, однако, другой, более эффективный способ закраски, который состоит в следующем. Многоугольники выводятся в обратном порядке, начиная от самых ближних к наблюдателю. Непосредственно перед формированием цвета пиксела проверяется его текущий цвет. Если он отличается от цвета фона, то это означает, что пиксел был закрашен ранее цветом многоугольника, стоящего ближе к наблюдателю. В этом случае новое вычисление цвета не производится, рассматривается следующий пиксел и т. д.

8.2. Алгоритм, использующий z-буфер

Алгоритм сортировки по глубине и некоторые другие алгоритмы удаления невидимых поверхностей основаны на упорядочении граней объекта по критерию их удаленности от точки наблюдения. Поскольку в общем случае линейное упорядочение невозможно, требуются дополнительные действия, такие как разбиение многоугольников на части, их перестановку и т. п., что усложняет алгоритм и требует дополнительных вычислительных затрат.

Описанный ниже алгоритм отличается от других тем, что обходится без всякой сортировки. Он работает в пространстве изображения и выполняет удаление невидимых частей многоугольников непосредственно в процессе их растеризации. Каждому пикселу (x, y) на экране ставится в соответствие элемент матрицы Z, в котором хранится значение z_v-координаты (*глубины*) многоугольника в точке (x, y). Матрица Z называется *z-буфером*. Перед началом работы алгоритма z-буфер заполняется достаточно большими значениями, а экран заполнить цветом фона. Если глубина многоугольника Q в точке (x, y) больше, чем глубина многоугольника P в этой точке, то значение z-буфера $Z[x, y]$ полагается равным глубине многоугольника P, а пиксел (x, y) закрашивается в цвет этого многоугольника (рис. 8.5). Особенностью алгоритма является то, что многоугольники выводятся в произвольном порядке, что значительно упрощает алгоритм и повышает его быстродействие.

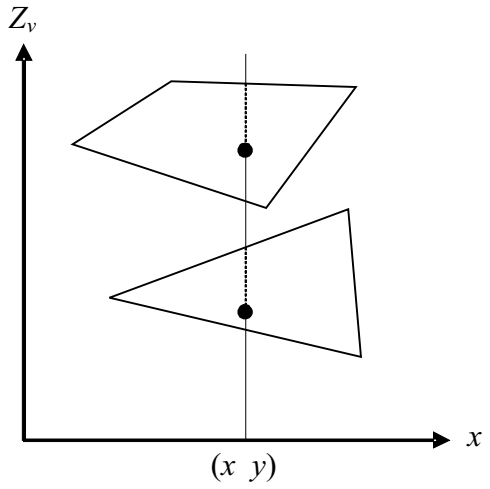


Рис. 8.5. Z-буфер содержит значение глубины многоугольника, ближайшего к наблюдателю в данной точке

Ниже представлена схема алгоритма. Исходные грани хранятся в очереди F. Каждая вершина грани хранит координаты ее проекции и значение глубины в этой вершине.

```
ZBuffer(F)
 1  while F ≠ ∅ do
 2      P ← Pop(F)
 3      for (по каждому пикселу) (x, y) ∈ P
 4          z ← Depth(P, x, y)
 5          if Z[x, y] > z then
 6              Z[x, y] ← z
 7              color ← Color(P, x, y)
 8              DrawPixel(x, y, color)
```

При растровой развертке грани вычисляется ее глубина в каждом пикселе, поэтому это вычисление желательно выполнять максимально быстро. Пусть уравнение плоскости, содержащей грань, имеет вид

$$Ax + By + CZ_v + D = 0,$$

откуда

$$Z_v = \frac{-D - Ax - By}{C}.$$

При выводе растровой строки y-координата пикселов остается неизменной, поэтому можно получить простую рекуррентную формулу для вычисления глубины каждого следующего пиксела на основе глубины предыдущего

$$Z_v(x+1, y) = Z_v(x, y) - \frac{A}{C},$$

где константа A/C вычисляется один раз для каждой грани.

Современные видеокарты включают поддержку z-буфера на аппаратном уровне. Это означает, что устранен наиболее существенный недостаток метода – высокие затраты памяти. Хотелось бы констатировать, что остаются лишь достоинства – простота и надежность, однако ложка дегтя имеется и здесь: при неудачном исходном расположении граней (см. упражнения) метод может тратить слишком много времени на вычисление цвета пикселов, которые в итоге окажутся невидимыми.

8.3. Алгоритм построчного сканирования

В следующем алгоритме объединены идеи двух предыдущих методов: в нем выполняется сортировка и растровая развертка с учетом глубины, но не всех многоугольников, а только тех, которые пересекают текущую строку растра.

Проекция объектов сцены может быть представлена в виде графа T, вершины и ребра которого совпадают с вершинами и ребрами граней выводимых объектов. До начала вывода изображения на экран в качестве препроцессорной обработки должны быть созданы следующие структуры данных:

- *Список ребер* E, который содержит ребра всех многоугольников. Каждое ребро e хранит следующую информацию:
  - *а*) указатели вершины и грани, инцидентные e;
  - *б*) x_0 – x-координата точки пересечения с текущей строкой растра;
  - *в*) δx – приращение x-координаты точки пересечения, которое добавляется к x_0 при переходе к следующей строке.
- *Список граней* F, содержащий номера вершин граней, а также информацию, позволяющую вычислить цвет грани в любой точке.

Алгоритм построчного сканирования является модификацией алгоритма Бентли – Оттмана (см. раздел 5.2.3), где в качестве заметающей прямой выступает строка растра. Напомним, что ребра, пересекаемые заметающей прямой, называются *активными*. Мы будем хранить их в динамической поисковой структуре данных E_A в порядке возрастания абсцисс точек пересечения со строкой растра. В интервале между двумя критическими точками ребра графа T не пересекаются, поэтому сканирующая строка может быть разбита на интервалы, внутри которых только одна грань является ближайшей к наблюдателю (рис. 8.6).

Закраска граней производится от интервала к интервалу с одновременным нахождением грани с наименьшей глубиной. Для упорядочения граней внутри интервала можно использовать значение z_v-координаты грани в серединной точке интервала. Таким образом, наряду со структурой данных E_A, реализующей статус сканирующей строки, должен существовать динамический контейнер F_A, который содержит *активные* грани и выполняет вставку и удаление за

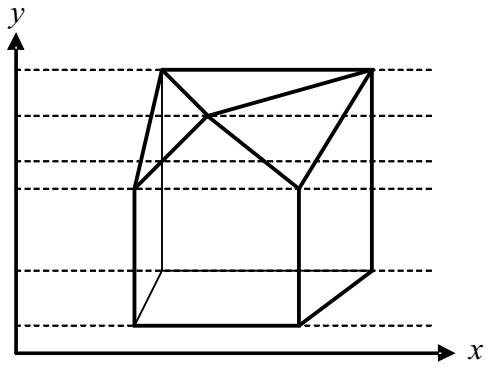


Рис. 8.6. При движении заметающей прямой в интервале между ординатами вершин ребра не пересекаются и поэтому могут быть упорядочены по возрастанию абсцисс точек пересечения со строкой растра

время $O(\log)$, а также поиск минимального элемента за время $O(1)$. Структурой данных, удовлетворяющей указанным требованиям, является очередь с приоритетом.

Итак, для записи алгоритма построчного сканирования нам потребуются три очереди с приоритетом:

1. V – для хранения вершин графа T, упорядоченных по возрастанию y-координаты, и для вставки найденных точек пересечения ребер.

2. E_A – для реализации статуса заметающей прямой, в котором хранятся активные ребра графа T, пересекающие текущую строку растра и упорядоченные по возрастанию x-координаты.

3. F_A – для хранения граней, пересекающих текущий интервал строки и упорядоченных по убыванию z_v-координаты.

Схема алгоритма представлена ниже. Рассмотрим его работу по шагам. В начале очередь V содержит только вершины ребер T, но в дальнейшем в нее могут быть вставлены найденные точки пересечения ребер. Процедура RetrieveActiveEdges(E, v_1, E_A) вставляет в очередь E_A ребра, инцидентные вершине v_1 и лежащие ниже нее, затем удаляет инцидентные ребра, лежащие выше v_1, и переставляет местами ребра в E_A, если v_1 является их точкой пересечения (строка 4). Эта же процедура проверяет на пересечение ребра, которые оказываются соседними в E_A в результате вставок и удалений, и при необходимости вставляет в V найденную точку пересечения.

Процедура RetrieveIntervals(E_A, y) в строке 7 вычисляет x-координаты точек пересечения активных ребер со строкой y, используя вычисленные ранее значения x_0 для каждого ребра и приращения δx.

Процедура RetrieveActiveFacets(F_A, e) обновляет список активных граней, вставляя в F_A грани, инцидентные ребру e и лежащие правее него, затем удаляет грани, лежащие левее e.

В завершение процедура DrawLine закрашивает интервал цветом грани f, ближайшей к наблюдателю (предполагается, что все точки грани имеют один цвет).

```
ScanLine(T)
 1  Занести вершины графа T в очередь V
 2  v1 ← Pop(V)
 3  while V ≠ ∅ do
 4      RetrieveActiveEdges(E, v1, EA)
 5      v2 ← Pop(V)
 6      for y ← y[v1] to y[v2] do
 7          RetrieveIntervals(EA, y)
 8          e1 ← head[EA]
 9          while next[e1] ≠ nil do
10              e2 ← next[e1]
11              RetrieveActiveFacets(FA, e1)
12              f ← head[FA]
13              DrawLine(y, x0[e1], x0[e2], color[f])
14              e1 ← e2
15      v1 ← v2
```

8.4. Алгоритм Варнока

Алгоритм построчного сканирования основан на трех вложенных одномерных сортировках: сначала вершины упорядочиваются по убыванию Y-координаты, затем в каждой новой вершине ребра упорядочиваются по возрастанию X-координат точек пересечения со сканирующей строкой. И наконец, внутри строки поддерживается упорядочение граней по глубине. Алгоритм Варнока основан на *двумерном* упорядочении, подобном тому, которое используется при региональном поиске с помощью квадратичного дерева (см. раздел 2.2.2). Алгоритм работает в пространстве изображения. Его идея состоит в рекурсивном разбиении прямоугольной области вывода на четыре одинаковые подобласти до тех пор, пока не станет возможным легко определить

видимость граней, проекции которых пересекают подобласть. По мере того как размеры подобласти сокращаются, ее пересекает все меньшее число многоугольников, так что в конце концов решение о закраске подобласти будет принято.

Проекция многоугольника может располагаться относительно прямоугольной области одним из четырех способов (рис. 8.7).

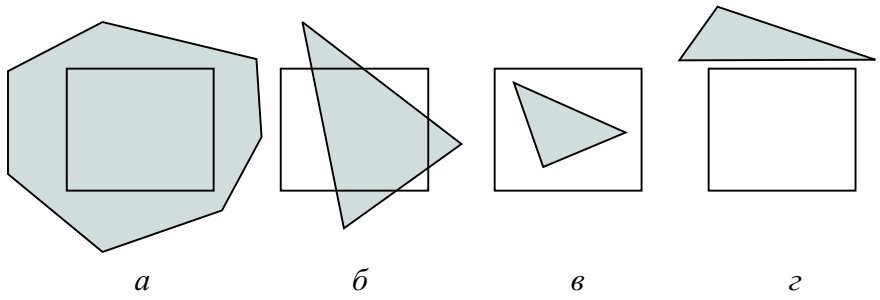


Рис. 8.7. Охватывающий (*а*), пересекающий (*б*), внутренний (*в*) и внешний (*г*) многоугольники

а) Многоугольник полностью включает в себя подобласть (*охватывающий* многоугольник).

б) Многоугольник имеет непустое пересечение с подобластью (*пересекающий* многоугольник).

в) Многоугольник целиком находится внутри подобласти (*внутренний* многоугольник).

г) Многоугольник целиком находится снаружи подобласти (*внешний* многоугольник).

Поскольку относительно одной и той же подобласти различные многоугольники могут располагаться различным способом, выделяют четыре случая, когда относительно этой подобласти можно принять решение о закраске (и прекращении дальнейшего ее разбиения):

1. Ни один из многоугольников не пересекается с подобластью. В этом случае подобласть окрашивается в цвет фона.

2. Существует либо только один пересекающий многоугольник, либо только один внутренний. Тогда внутренний многоугольник или часть пересекающего, попавшая в подобласть, преобразуются в растровую форму.

3. Существует лишь один охватывающий многоугольник, и нет ни одного пересекающего или внутреннего. Подобласть окрашивается в цвет этого многоугольника.

4. Существует по крайней мере один охватывающий, а также несколько пересекающих или внутренних. В этом случае необходимо определить, лежит ли какой-либо из охватывающих многоугольников ближе всех к наблюдателю. Для этого вычисляются и сравниваются значения z_v-координат граней во всех четырех вершинах подобласти. Если такой охватывающий многоугольник существует, то подобласть закрашивается в цвет этого многоугольника. Заметим, что значения глубины вычисляются не для самих граней (проекции внутренних и пересекающих могут вообще не содержать вершин подобласти), а для плоскостей, содержащих эти грани.

Если ни один из четырех тестов не позволил принять решение, то подобласть пропорционально разбивается на четыре части, и к каждой из них применяются такие же тесты.

После очередного разбиения необходимо идентифицировать лишь внутренние и пересекающие многоугольники: внешние и охватывающие для предыдущей подобласти остаются такими же и для ее частей.

Процесс разбиения имеет естественный предел – разрешение экрана. Если подобласть уменьшилась до размеров пиксела, а решение принять нельзя, то в центре этого пиксела вычисляется z_v-координата для всех многоугольников и пиксел закрашивается цветом многоугольника, имеющего минимальную z_v-координату (при этом можно продолжить виртуальные разбиения для устранения лестничного эффекта).

Упражнения

2-1. Оцените трудоемкость алгоритма сортировки по глубине. Проанализируйте варианты взаимного расположения граней и установите максимальное количество возможных разбиений одной грани относительно плоскостей других граней. Как это количество влияет на асимптотическую оценку скорости работы алгоритма?

2-2. При каком исходном расположении граней метод, использующий z-буфер, работает особенно медленно? Оцените отношение минимальной и максимальной производительности метода в зависимости от числа граней.

2-3. Реализуйте метод сортировки по глубине и метод, использующий z-буфер. Сравните их быстродействие для предметов и сцен различной сложности.

2-4. Используя оценку трудоемкости алгоритма Бентли – Оттмана, получите аналогичную оценку для алгоритма построчного сканирования.

2-5. Сравните быстродействие:

а) метода построчного сканирования и метода, использующего z-буфер;

б) метода, использующего z-буфер, и метода Варнока.

2-6. Какие ограничения на взаимное расположение предметов сцены связаны с числом двоичных разрядов, отведенных на хранение глубины в z-буфере?

2-7. Разработайте эффективный метод вычисления глубины точки на поверхности сферы на основе глубины в соседних пикселах.

Глава 9

Введение в реалистическую графику

Алгоритмы, рассмотренные в предыдущих главах, не позволяют получать реалистические изображения трехмерных объектов, подобные тем, которые мы видим на фотографиях. Для построения таких изображений требуется учет всех параметров сцены: количество и атрибуты источников света, описание материала изображаемых объектов, состояние атмосферы и т. д. Знание этих параметров необходимо для вычисления интенсивности световой энергии, претерпевающей различные изменения на пути от источника света к наблюдателю. Однако в силу многообразия факторов, влияющих на конечное значение интенсивности, их полный учет невозможен, и в модели освещения участвуют только наиболее важные параметры. Рассмотрим несколько моделей освещения в порядке возрастания их сложности.

9.1. Простая модель освещения

Световая энергия, падающая на поверхность, может быть поглощена, отражена или пропущена. В природе не существует материалов, полностью поглощающих, отражающих или пропускающих свет, поэтому можно говорить только о частичном поглощении, отражении или пропускании света.

Количество поглощенной, отраженной и пропущенной световой энергии, помимо прочих факторов, зависит от длины волны света. Если освещенная поверхность поглощает почти весь свет видимого диапазона, то она выглядит черной, а если только малая его часть, то белой или зеркальной. Если поглощается свет с определенной длиной волны, то поверхность выглядит цветной.

Отражение от поверхности может быть *диффузным* или *зеркаль-*

ным. При диффузном отражении свет, полученный от источника, поглощается поверхностью, а затем испускается с одинаковой интенсивностью во всех направлениях (рис. 9.1). Такие атрибуты поверхности, как цвет и текстура, проявляются благодаря именно диффузному отражению, поскольку цвет диффузно отраженного света определяется оптическими свойствами материала поверхности.

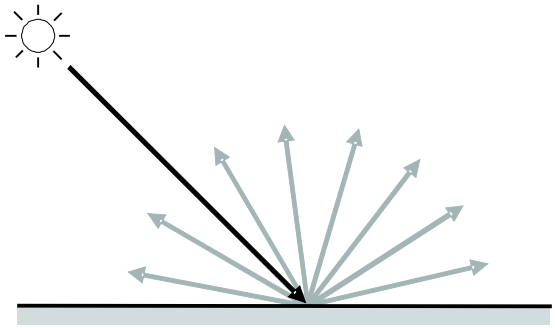


Рис. 9.1. Падающий и отраженный свет при диффузном отражении

Зеркальное отражение происходит строго в определенном направлении: вектор R отраженного света лежит в одной плоскости с вектором L, направленным на источник света, и вектором N нормали к поверхности в точке падения луча, угол между L и N равен углу между N и R (рис. 9.2).

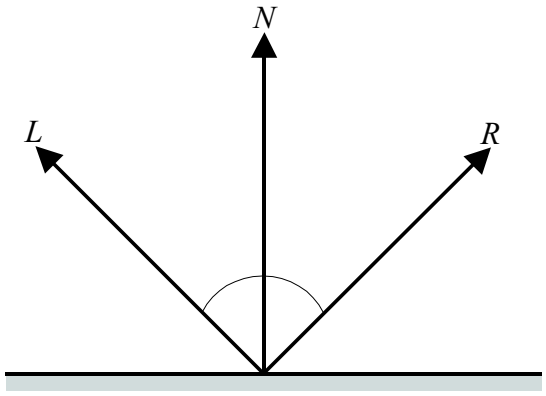


Рис. 9.2. Падающий и отраженный луч при зеркальном отражении

Помимо прямого света, падающего на поверхность непосредственно из источника, в формировании суммарной интенсивности отраженного света принимает участие также *рассеянный* свет, который достигает

поверхности, отражаясь от атмосферы или других объектов сцены. Как и при диффузном отражении, рассеянный свет отражается от поверхности во всех направлениях с одинаковой интенсивностью.

Таким образом, суммарная интенсивность отраженного света складывается из трех компонент:

$$I = R_a + R_d + R_s,$$

где R_a – интенсивность отражения рассеянного света, а R_d и R_s – интенсивности диффузного и зеркального отражения.

Если обозначить через I_a интенсивность рассеянного света, то

$$R_a = I_a k_d, \tag{9.1}$$

где k_d – коэффициент диффузного отражения, зависящий от свойств материала поверхности и изменяющийся в пределах от 0 до 1. Для белой поверхности $k_d = 1$, для черной – $k_d = 0$.

Интенсивность диффузной составляющей отраженного света вычисляется с помощью закона *косинусов Ламберта*:

$$R_d = I_l k_d \cos\theta, \quad 0 \le \theta \le \frac{\pi}{2},$$

где I_l – интенсивность света, падающего на поверхность, θ – угол между направлением на источник света и нормалью к поверхности. Условимся, что вектор L, задающий направление на источник света, и вектор N нормали к поверхности в точке падения луча имеют единичную длину. Тогда закон косинусов Ламберта может быть записан в векторном виде:

$$R_d = I_l k_d (L, N). \tag{9.2}$$

Поверхность, изображенная с помощью модели (9.2), кажется матовой, поскольку в ней отсутствует зеркальная составляющая. В ней также отсутствует учет расстояния от источника света до поверхности и от поверхности до наблюдателя, вследствие чего проекции поверхностей, расположенных под одним и тем же углом к источнику света, будут закрашены одинаково, их изображения могут сливаться. Обозначим через E интенсивность источника, а через D расстояние от источника до поверхности. Тогда

$$I_l = \frac{E}{D^2}. \tag{9.3}$$

Подстановка (9.3) в (9.2) даст заметный эффект в случае, когда источник расположен вблизи от освещаемых объектов. Для удаленного

источника, такого как солнце, значение D^2 и вместе с ним значение I_l для различных объектов сцены будут изменяться настолько незначительно, что D^2 обычно заменяют на константу D_0, и в модель включают учет расстояния d от поверхности до наблюдателя (т. е. центра проекции)

$$R_d = \frac{Ek_d}{D_0 + d}(L, N). \tag{9.4}$$

Объединив (9.1) и (9.4), получим простую модель освещения для диффузного отражения

$$I = I_a k_d + \frac{Ek_d}{D_0 + d}(L, N). \tag{9.5}$$

Для цветных поверхностей интенсивность диффузного отражения вычисляется отдельно для красной, зеленой и синей цветовой компоненты:

$$\begin{aligned} I_r &= I_{ar}k_{dr} + \frac{E_r k_{dr}}{D_0 + d}(L, N), \\ I_g &= I_{ag}k_{dg} + \frac{E_g k_{dg}}{D_0 + d}(L, N), \\ I_b &= I_{ab}k_{db} + \frac{E_b k_{db}}{D_0 + d}(L, N). \end{aligned}$$

9.2. Зеркальная составляющая

В результате зеркального отражения на поверхности появляются световые блики, причем цвет бликов определяется цветом источника света, а не поверхности.

Интенсивность зеркального отражения зависит от угла падения луча, длины волны падающего света и оптических свойств материала поверхности. Угол отражения от идеально отражающей поверхности равен углу θ падения луча (рис. 9.3). Если угол α между нормалью N к поверхности в точке падения луча и направлением S на наблюдателя не равен θ, то наблюдатель не увидит отраженный свет. Если же поверхность неидеальна, то количество света, достигшее наблюдателя, зависит от пространственного распределения зеркальной составляющей отраженного света. У гладких поверхностей оно более узкое, у шероховатых – более широкое. Именно так и появляются блики.

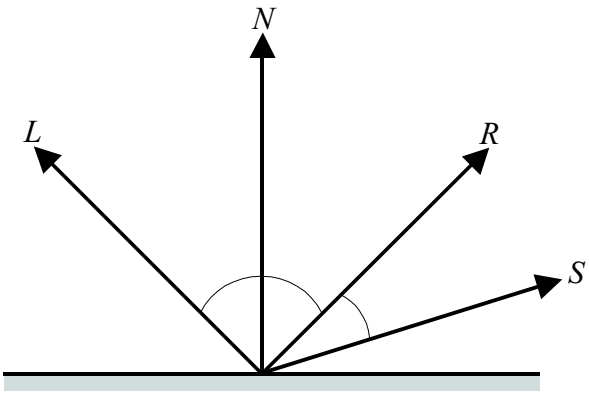


Рис. 9.3. Зеркальная составляющая отраженного света

Зеркальная составляющая описывается с помощью модели Фонга:

$$I_s = E\omega(\theta, \lambda)\cos^p \alpha,$$

где $\omega(\theta, \lambda)$ – коэффициент отражения, зависящий от свойств материала и определяемый углом падения и длиной волны, p – параметр, задающий гладкость поверхности: для идеального отражателя значение p стремится к бесконечности, для шероховатой поверхности – близко к единице. На рис. 9.4 представлены графики функции $\cos^p \alpha$ для различных значений параметра p.

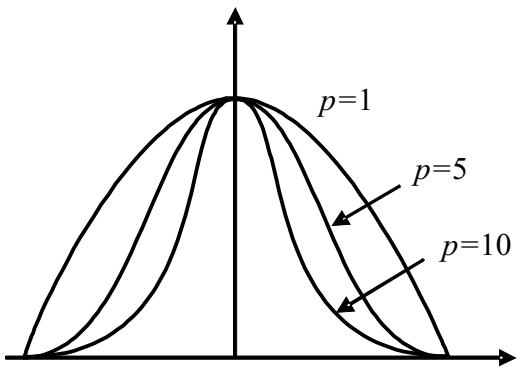


Рис. 9.4. Зависимость значения функции $\cos^p \alpha$ от параметра p

Функция $\omega(\theta, \lambda)$ имеет довольно сложное аналитическое представление, и на практике ее заменяют константой k_s, значение которой определяют экспериментально.

С учетом зеркальной составляющей простая модель освещения

выглядит следующим образом:

$$I = I_a k_d + E\,\frac{k_d(L,N) + k_s(R,S)^p}{D_0 + d}.$$

Как вычислить координаты вектора отраженного луча R? Вектор R лежит в одной плоскости с векторами L и N и имеет единичную длину. Рассмотрим вектор v равный разности проекции вектора L на вектор N и вектора L (рис. 9.5). Тогда $R = L + 2v$. Но $v = (L,N)N - L$, поэтому

$$R = L + 2((L,N)N - L) = 2(L,N)N - L.$$

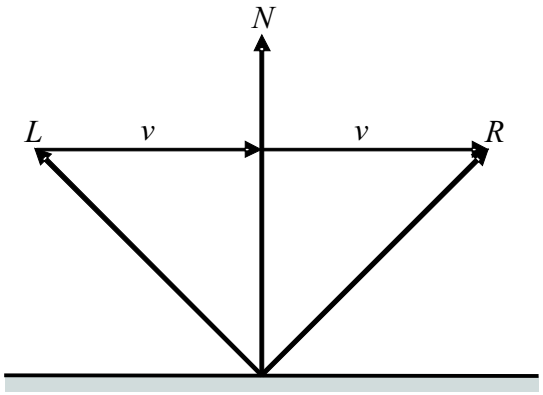


Рис. 9.5. Вычисление вектора отраженного луча

9.3. Направленный свет

Точечные источники света испускают свет с одинаковой интенсивностью во всех направлениях. Свет от *направленных* источников распространяется преимущественно в одном заданном направлении.

Направленный свет моделируется как отражение от точечной идеально зеркальной поверхности при освещении ее точечным источником (рис. 9.6), что позволяет изменять направление света независимо от источника.

Если угол β между направлением $-L$, обратным к направлению на направленный источник света, и нормалью N_L к поверхности источника равен нулю, то объект освещается, как от обычного точечного источника. В противном случае интенсивность освещения

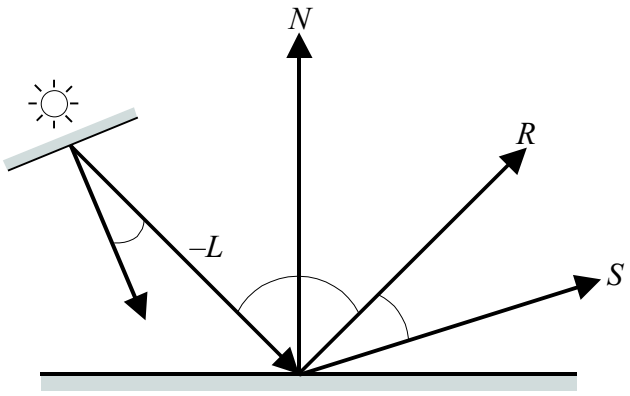


Рис. 9.6. Распределение световой энергии при освещении из направленного источника

направленным светом равна

$$I_L = E\cos^q\beta = -E\ (N_L, L)^q,$$

где q определяет степень пространственной концентрации направленного источника: при $q = 20$ получаем узко направленный свет, подобный тому, который испускает прожектор, значение $q = 1$ соответствует заливающему свету.

Теперь простую модель освещения можно дополнить с учетом возможности освещения несколькими направленными источниками:

$$I = I_a k_d - \sum_{i=1}^{k} E_i\ (N_{L_i}, L_i)\frac{k_d(L_i, N) + k_s(R, S)^p}{D_0 + d}.$$

Упражнения

2-1. Реализуйте простую модель освещения для цветных поверхностей.

2-2. Реализуйте простую модель освещения для нескольких источников разного цвета. Выполните тесты для белых и цветных поверхностей.

Глава 10

Диаграмма Вороного

В этой главе мы обсудим вопросы, связанные с отношением *геометрической близости*, заданным на множестве объектов на плоскости. В качестве примера рассмотрим работу авиационной диспетчерской службы, которой в каждый момент времени известны координаты всех находящихся в небе объектов в радиусе действия радиолокационной станции. Необходимо установить, какие два объекта расположены наиболее близко друг к другу. Или: какая из заправочных станций является ближайшей к автомобилю, который находится в движении и, значит, непрерывно меняет свое местоположение? Требуется минимизировать время ответа на запрос (очевидно, в случае с диспетчерской службой такое требование является критическим).

Задачи геометрической близости имеют одну привлекательную особенность: в них необходимо в полной мере учитывать геометрическую природу объектов, принимая во внимание не только их взаимное расположение, но также форму, размер и относительное расстояние.

Рассмотренные ниже алгоритмы пополнят нашу коллекцию классическими подходами к обработке геометрической информации. Мы познакомимся с универсальной комбинаторно-геометрической структурой, создание которой позволяет получить эффективное решение для широкого класса задач.

10.1. Задачи геометрической близости

Ниже перечислены некоторые из задач, связанные с понятием геометрической близости. Пусть на плоскости задано множество S из n точек.

1. **Ближайший сосед**. Для заданной точки $z \notin S$ найти ближайшую к ней точку множества S при условии, что допускается предобработка (рис. 10.1).

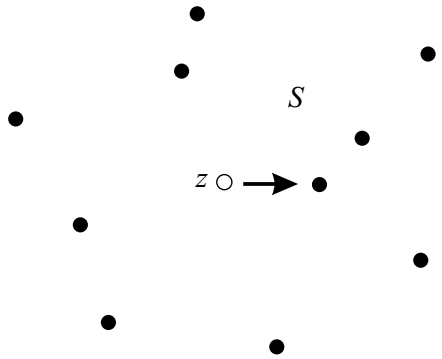


Рис. 10.1. Задача о ближайшем соседе

2. **Ближайшая пара**. Найти две точки множества S, расстояние между которыми наименьшее среди всех пар точек из S.
3. **Все ближайшие пары**. Для каждой точки множества S указать ближайшую к ней точку этого множества.
4. **Евклидово минимальное остовное дерево**. Построить дерево с вершинами в точках множества S с минимальной суммарной длиной ребер.
5. **Триангуляция**. Построить максимальное по мощности множество непересекающихся отрезков с концами в S (рис. 10.2, *а*).
6. **(k ближайших соседей)**. Найти k точек множества S, ближайших к заданной точке $z \notin S$.

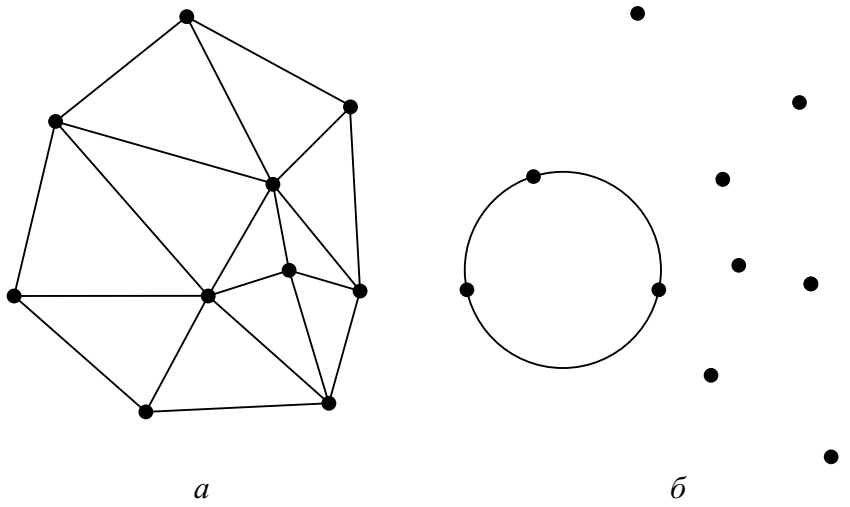


Рис. 10.2. Триангуляция точек на плоскости (*а*) и наибольшая пустая окружность с центром внутри выпуклой оболочки (*б*)

7. **(Наименьшая охватывающая окружность)**. Найти наименьшую окружность, охватывающую точки множества S.

8. **(Наибольшая пустая окружность)**. Найти наибольшую окружность с центром внутри выпуклой оболочки множества S, не содержащую внутри себя ни одной точки из S (рис. 10.2, *б*). Распространенными интерпретациями задачи являются выбор площадки для строительства вредного производства или поиск местоположения для торгового центра, максимально удаленного от конкурирующих объектов. Очевидно, в обоих случаях искомая точка – это центр наибольшей пустой окружности.

Этот список можно продолжить и далее (см., например, [28] главы 5 и 6), но мы на этом остановимся, поскольку для общего представления о характере изучаемых проблем этого достаточно. В следующих разделах, где мы займемся решением некоторых из перечисленных задач, нам потребуется знание нижних оценок.

Нижние оценки

1. К задаче «ближайший сосед» сводится задача ПОИСК В УПОРЯДОЧЕННОЙ ПОСЛЕДОВАТЕЛЬНОСТИ: *задана последовательность действительных чисел* $x_1, \ldots, x_n$ *такая, что* $x_1 \leq \ldots \leq x_n$*; определить, присутствует ли в ней заданное число* z. Известно, что для ответа на этот вопрос в рамках модели дерева решений[1] требуется не менее $\Omega(\log n)$ сравнений. Но эту же задачу можно решить, поставив в соответствие каждому числу x_i точку на числовой прямой с координатами $(x_i, 0)$ и выполнив поиск ближайшего соседа для точки $(z, 0)$. Сведение выполнено за константное время, поэтому нижняя оценка $\Omega(\log n)$ справедлива также для задачи «ближайший сосед».

2. К задаче о ближайшей паре сводится задача ЕДИНСТВЕННОСТЬ ЭЛЕМЕНТОВ с известной нижней оценкой $\Omega(n \log n)$: *заданы* n *действительных чисел, все ли они различны*? Поставим в соответствие каждому числу точку на числовой прямой, затем решим для полученного множества точек задачу о ближайшей паре и проверим, совпадают найденные ближайшие точки или нет. Как и в предыдущем случае, сведение выполнено за время $O(1)$, поэтому для задачи о ближайшей паре имеем нижнюю оценку $\Omega(n \log n)$.

[1] См. [20], раздел 1.5.

3. Для задачи «все ближайшие пары» получить нижнюю оценку особенно легко, поскольку к ней очевидным образом сводится задача «ближайшая пара». Действительно, если перечислить все ближайшие пары, то найти среди них самую ближайшую можно за время $O(n)$. Таким образом, поиск всех ближайших пар не может быть выполнен быстрее, чем за время $\Omega(n \log n)$. Казалось бы, этот результат не столько очевиден, сколько бесполезен – ведь интерес представляют только достижимые нижние оценки или приближенные к верхним. Но, как ни странно, в нашем случае полученная оценка достижима! Ниже будет показано, что наиближайшая пара и все ближайшие пары могут быть найдены за оптимальное время $\boldsymbol{\theta}(n \log n)$.

4. Евклидово минимальное остовное дерево и триангуляция n точек на плоскости не могут быть построены быстрее, чем за время $\Omega(n \log n)$. Предлагаем читателю доказать этот факт самостоятельно, использовав в качестве эталонной задачу с известной нижней оценкой задачу сортировки n действительных чисел. Для нахождения наименьшей охватывающей и наибольшей пустой окружности требуется не менее, соответственно, $\Omega(n)$ и $\Omega(n \log n)$ времени.

10.2. Области близости

Понятно, что при решении задач близости необходимо каким-то образом учитывать расстояние между точками множества S. Желательно также иметь структуру данных, реализующую отношение близости и позволяющую выполнять запросы на выборку точек, «расположенных рядом». Сформулируем проблему в терминах метода локусов, с которым мы уже сталкивались в разделе 2.1.

Задача 16. *На плоскости задано множество S из n точек. Для каждой точки $p \in S$ определить геометрическое место точек (локус) $V(p)$, расстояние от которых до p не превосходит расстояния до любой другой точки множества S.*

Обозначим через $H(p, q)$ полуплоскость, содержащую точку p и определяемую серединным перпендикуляром к отрезку $\overline{p, q}$. Если множество S состоит только из этих двух точек, то $V(p) = H(p, q)$. В общем случае при $n > 2$ искомое множество $V(p)$ есть пересечение $n - 1$ полуплоскости и представляет собой выпуклую многоугольную

область с не более чем $n-1$ стороной

$$V(p) = \bigcap_{p \neq q} H(p, q).$$

Эта область называется *многоугольником Вороного* точки p, *доменом Вороного* или просто *доменом* (рис. 10.3, *а*). Для заданного множества точек S разбиение плоскости на домены $V(p), p \in S$ задает прямолинейный граф $\mathrm{Vor}(S)$, называемый *диаграммой Вороного множества* S (рис. 10.3, *б*). Вершины и ребра этого графа называются, соответственно, *вершинами* и *ребрами* диаграммы Вороного.

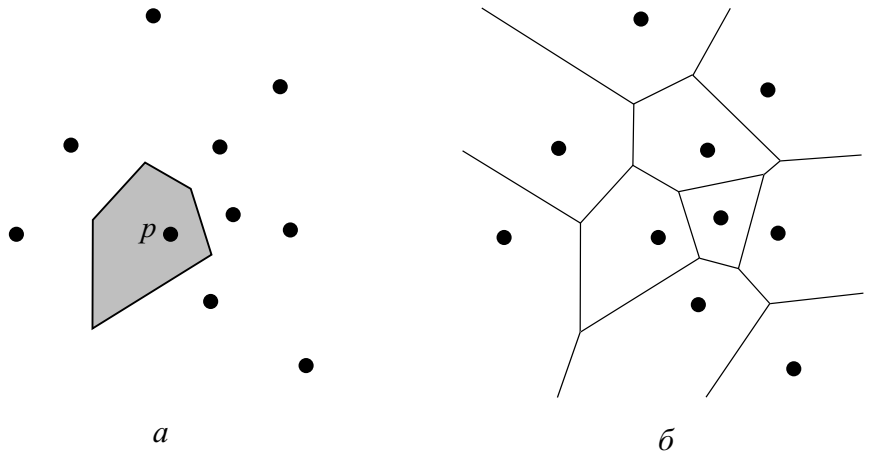


Рис. 10.3. Многоугольник Вороного (*а*) и диаграмма Вороного (*б*)

Заметим, что по определению ближайшим соседом точки $z \in V(p)$ является точка p, а ребра смежных доменов естественным образом определяют отношение близости точек множества S. Таким образом, есть все основания полагать, что диаграмма Вороного и есть та самая структура данных, которая обеспечивает эффективное решение задач геометрической близости. Ниже мы формально докажем это интуитивное предположение, но сначала рассмотрим важные свойства диаграммы Вороного.

10.3. Свойства диаграммы Вороного

Для простоты изложения предположим, что никакие четыре точки множества S не лежат на одной окружности.

Свойство 1. *Любая вершина диаграммы Вороного является точкой пересечения в точности трех ее ребер.*

Доказательство. Действительно, пусть вершина v есть точка пересечения ребер $e_1, \ldots, e_k$, которые лежат на серединных перпендикулярах к парам $(p_1, p_2), \ldots, (p_k, p_1)$ точек множества S. В силу предположения, $k \leq 3$. Если $k = 2$, то оба ребра e_1 и e_2 лежат на одном серединном перпендикуляре к точкам p_1 и p_2, поэтому v не может являться их точкой пересечения. ∎

Таким образом, любая вершина диаграммы Вороного равноудалена от некоторых трех точек p_1, p_2 и p_3, образующих три смежных домена $V(p_1)$, $V(p_2)$ и $V(p_3)$. Обозначим через $v(p_1, p_2, p_3)$ общую вершину этих доменов.

Свойство 2. *Для любой вершины $v = v(p_1, p_2, p_3)$ диаграммы Вороного множества S окружность $C(v)$ с центром в этой вершине, проходящая через p_1, p_2 и p_3, не содержит внутри себя никаких других точек множества S.*

Доказательство. Предположим, что $C(v)$ содержит некоторую точку p_4 множества S. Поскольку v расположена к p_4 ближе, чем к p_1, p_2 и p_3, она лежит строго внутри домена $V(p_4)$ и поэтому не может являться вершиной диаграммы Вороного. ∎

Свойство 3. *Ближайшая пара точки $p \in S$ определяет ребро домена $V(p)$ диаграммы Вороного множества S.*

Доказательство. Пусть точка q является ближайшей к точке p, и серединный перпендикуляр к отрезку $\overline{p, q}$ не определяет ребро в домене $V(p)$ (рис. 10.4). Тогда $\overline{p, q}$ пересекает в точке u некоторое ребро этого домена, соответствующее паре (p, r). Имеем:

$$d(p, q) > 2d(p, u) \geq 2d(p, v) = d(p, r),$$

то есть точка q не является ближайшей к p. ∎

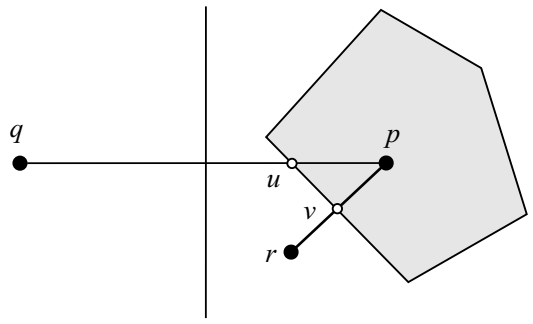


Рис. 10.4. Ближайшая пара точки p определяет ребро домена $V(p)$

Следующие три свойства приведем без доказательства.

Свойство 4. *Домен $V(p)$ диаграммы Вороного множества точек S является неограниченным тогда и только тогда, когда точка p является вершиной выпуклой оболочки S.*

Если соединить отрезками точки множества S, соответствующие смежным доменам, то получится прямолинейный граф, который является двойственным к диаграмме Вороного (рис. 10.5).

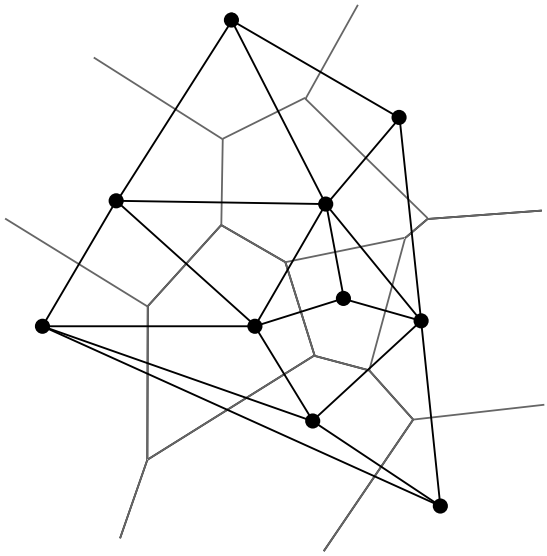


Рис. 10.5. Триангуляция Делоне

Свойство 5. *Граф, двойственный к диаграмме Вороного, является триангуляцией.*

Определенная таким образом триангуляция называется *триангуляцией Делоне*. Заметим, что для данного множества точек диаграмма Вороного определяется единственным образом, поэтому двойственная к ней триангуляция также определяется однозначно.

Поскольку диаграмма Вороного является планарным графом, для нее справедлива формула Эйлера, устанавливающая линейные соотношения между числом ребер, вершин и граней. Более того, можно точно вычислить число вершин и ребер диаграммы Вороного, зная число точек множества S и число вершин ее выпуклой оболочки.

Свойство 6. *Диаграмма Вороного множества S из n точек имеет ровно $v = 2n - c - 2$ вершины и $e = 3n - c - 3$ ребра, где $c = |\mathrm{conv}S|$.*

Из этих соотношений следует равенство $3v = 2e - c$. Представляет интерес также следующее замечание. Число всех доменов равно n, а число ребер всех доменов равно $2e \approx 6n$, то есть на каждый домен приходится в среднем 6 ребер, а каждая вершина триангуляции Делоне инцидентна в среднем 6 ребрам.

Теперь мы можем приступить к описанию алгоритмов построения диаграммы Вороного. Найдем сначала нижнюю оценку сложности задачи. Из свойства 3 следует, что к построению диаграммы Вороного сводится задача «ближайший сосед». Кроме того, к ней тривиально сводится задача сортировки множества действительных чисел. Таким образом, диаграмма Вороного не может быть построена за время меньшее, чем $\Omega(n \log n)$.

Заметим, что диаграмму Вороного можно построить, последовательно формируя домены Вороного для каждой точки из S. Каждый домен строится, в свою очередь, как пересечение n полуплоскостей за время $O(n \log n)$. Таким образом, тривиальный подход дает алгоритм сложности $O(n^2 \log n)$.

Ниже мы рассмотрим два алгоритма с оптимальным быстродействием $\boldsymbol{\Theta}(n \log n)$, а именно: алгоритм типа «разделяй и властвуй» и алгоритм Форчуна (Fortune). Таким образом, построение *всех* доменов Вороного асимптотически является не более сложной задачей, чем построение одного домена!

10.4. Алгоритм типа «разделяй и властвуй»

Определимся со структурами данных. Очевидно, нам потребуются контейнеры для координат исходных точек множества S и вершин диаграммы. Поскольку множество S имеет фиксированный размер, контейнером может служить статический массив или вектор (в терминах раздела 1.6). Тогда для ссылки на точку можно использовать ее порядковый номер. Для хранения ребер и доменов существует множество различных подходов, выбор которых определяется алгоритмом построения и эффективностью будущих запросов к диаграмме Вороного. Будем предполагать, что выбранную структуру данных можно преобразовать в любую другую за линейное время, поэтому вопрос об эффективности запросов пока оставим в стороне.

Как и любой другой планарный граф, диаграмма Вороного может быть представлена *реберным списком с двойными связями*. В этом списке каждое ребро содержит две ссылки p_1 и p_2 на концевые вершины, а также две ссылки e_1 и e_2 на смежные ребра, следующие за данным ребром при обходе вокруг концевой вершины по часовой стрелке (рис. 10.6).

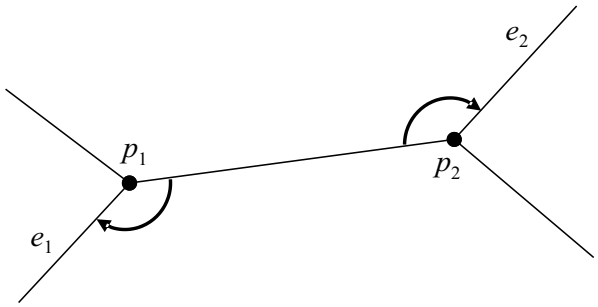


Рис. 10.6. Реберный список с двойными связями

10.4.1. Общая схема

Метод «разделяй и властвуй» включает три шага:

1. Разделить множество S на два примерно равных по можности подмножества S_1 и S_2.
2. Рекурсивно построить $\mathrm{Vor}(S_1)$ и $\mathrm{Vor}(S_2)$.
3. Построить $\mathrm{Vor}(S)$, объединив $\mathrm{Vor}(S_1)$ и $\mathrm{Vor}(S_2)$.

Понятно, что основную сложность представляет шаг 3, который необходимо выполнить за линейное время (иначе мы не получим алгоритм с оценкой $O(n \log n)$). Прежде всего разберемся, как диаграммы $\mathrm{Vor}(S_1)$ и $\mathrm{Vor}(S_2)$ связаны с $\mathrm{Vor}(S)$.

Определение 1. *Цепь C (то есть последовательность ребер вида $(p_1, p_2), (p_2, p_3), \ldots, (p_{k-1}, p_k)$) называется монотонной относительно прямой l, если любая прямая, перпендикулярная l, пересекает C не более чем в одной точке.*

Обозначим через $\boldsymbol{\sigma}(S_1, S_2)$ множество ребер $\mathrm{Vor}(S)$, которые являются общими для пар смежных доменов $V(p_1)$ и $V(p_2)$, где $p_1 \in S_1$, а $p_2 \in S_2$. Можно строго доказать [28], что $\boldsymbol{\sigma}(S_1, S_2)$ состоит из циклов

и цепей графа $\mathrm{Vor}(S)$, не имеющих общих ребер. Кроме того, если множества S_1 и S_2 линейно разделимы некоторой прямой l, то $\boldsymbol{\sigma}(S_1, S_2)$ представляет собой единственную цепь, монотонную относительно этой прямой.

Теперь мы получили воможность уточнить действия, производимые на шаге 3.

Утверждение 5. *Пусть множества S_L и S_R линейно разделимы вертикальной прямой и при этом все точки S_L расположены левее точек S_R. Обозначим через $\boldsymbol{\sigma}^L(S_L, S_R)$ и $\boldsymbol{\sigma}^R(S_L, S_R)$ две неограниченные многоугольные области, на которые плоскость разбивается ломаной $\boldsymbol{\sigma}(S_L, S_R)$. Тогда*

$$\mathrm{Vor}(S) = \big(\mathrm{Vor}(S_L) \cap \boldsymbol{\sigma}^L(S_L, S_R)\big) \cup \big(\mathrm{Vor}(S_R) \cap \boldsymbol{\sigma}^R(S_L, S_R)\big) \cup \boldsymbol{\sigma}(S_L, S_R).$$

Получаем следующую уточненную схему алгоритма:

1. Разделить множество S на два примерно равных по можности подмножества S_L и S_R, использовав для этого медиану по x-координате.
2. Рекурсивно построить $\mathrm{Vor}(S_L)$ и $\mathrm{Vor}(S_R)$.
3. Построить разделяющую ломаную $\boldsymbol{\sigma}(S_L, S_R)$.
4. Удалить части ребер $\mathrm{Vor}(S_L)$ справа от ломаной $\boldsymbol{\sigma}$ и части ребер $\mathrm{Vor}(S_R)$ слева от нее.
5. Включить в $\mathrm{Vor}(S)$ ребра ломаной $\boldsymbol{\sigma}$.

Таким образом, задача сводится к эффективному нахождению разделяющей ломаной $\boldsymbol{\sigma}$.

10.4.2. Построение разделяющей ломаной

Из свойства 4 диаграммы Вороного следует, что неограниченные сегменты ломаной $\boldsymbol{\sigma}$ (верхний и нижний) определяются серединными перпендикулярами к опорным отрезкам для выпуклых оболочек $CH(S_L)$ и $CH(S_R)$ (рис. 10.7). Построение ломаной $\boldsymbol{\sigma}$ начнем с верхнего неограниченного сегмента (потратив на нахождение верхнего опорного отрезка не более $O(n)$ действий), затем, переходя от сегмента к сегменту, будем продвигаться вниз, пока не достигнем нижнего опорного отрезка, и завершим построение ломаной, добавив в нее нижний неограниченный сегмент.

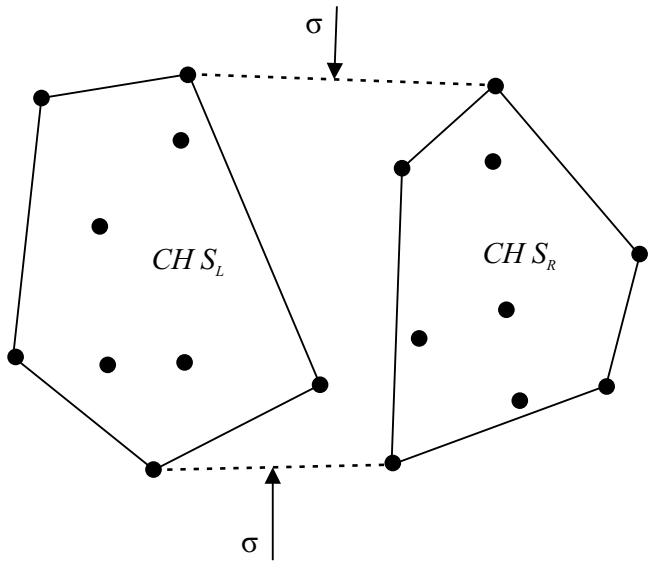


Рис. 10.7. Неограниченные сегменты ломаной σ определяются серединными перпендикулярами к опорным отрезкам для $CH(S_L)$ и $CH(S_R)$

Обозначим через e текущий (последний построенный) сегмент и через v его нижний конец. Изначально в качестве e выбираем верхний неограниченный сегмент (луч) e^+, который определяется серединным перпендикуляром к точкам p_L и p_R, задающим верхний опорный отрезок. Затем, двигаясь вниз вдоль e^+, мы встретим ребро домена $V(p_L)$ или $V(p_R)$. Пусть для определенности это будет ребро e_L домена $V(p_L)$, отделяющее этот домен от домена $V(q)$, где $q \in S_L$ (рис. 10.8). Что произойдет с ломаной в этом случае? Если до пере-

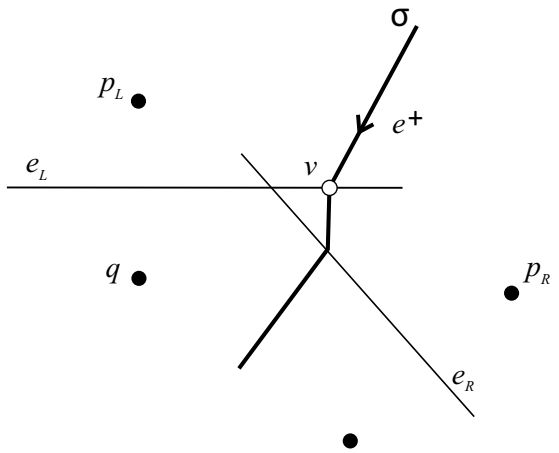


Рис. 10.8. При движении вдоль e^+ первым встретилось ребро домена $V(p_L)$

сечения с e_L сегмент e^+ целиком принадлежал доменам $V(p_L)$ или $V(p_R)$ и отстоял на равном расстоянии от p_L и p_R, то после пересечения точка q становится более близкой к e^+, чем p_L. Стало быть, в точке пересечения с ребром e_L ломаная σ должна изменить свое направление таким образом, чтобы расстояние от нее до ближайших точек S_L и S_R опять стало равным. Поэтому полагаем в качестве v точку пересечения луча e^+ с ребром e_L, в качестве p_L – точку q, а в качестве нового сегмента e ломаной – серединный перпендикуляр к обновленной паре (p_L, p_R). Затем итерация повторяется: ищем пересечение сегмента e с ребрами доменов $V(p_L)$ и $V(p_R)$ и т. д.

Заметим, что ломаная может иметь пересечение с несколькими ребрами одного и того же домена. Принимая во внимание, что все построение должно занять не более $O(n)$ времени, необходимо организовать перебор ребер домена таким образом, чтобы исключить повторные проверки для одного и того же ребра. Здесь важную роль играет тот факт, что ломаная σ является монотонной относительно вертикальной прямой, то есть ординаты ее вершин строго уменьшаются по мере ее построения. Предположим, что последнее пересечение ломаной с доменом $V(p_L)$ было зафиксировано с ребром e_L (рис. 10.9). Тогда поиск следующего пересечения с ломаной

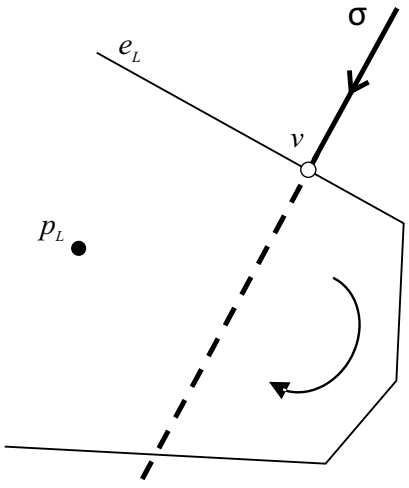


Рис. 10.9. Поиск пересечения с σ можно начинать с ребра, следующего за e_L в порядке обхода по часовой стрелке

нужно начинать с ребра, следующего за e_L в порядке обхода *по часовой стрелке*, и при этом гарантированно исключаются повторные проверки! Действительно, «возврат» к уже пройденному ребру возможен только для немонотонной цепи. Аналогичные рассуждения

справедливы для доменов диаграммы $\mathrm{Vor}(S_R)$: для них обход ребер необходимо производить *против часовой стрелки*.

Для компактной записи алгоритма нам потребуются несколько простых функций. Пусть функция $\mathrm{I}(e_1,\ e_2)$ возвращает точку пересечения ребер e_1 и e_2, а функция $\textsc{Pair}(p_1,\ e)$ возвращает пару точки p_1, определяемую серединным перпендикуляром e к отрезку $\overline{p_1, p_2}$. Через $\textsc{Bisector}(p_1,\ p_2)$ обозначим функцию, вычисляющую серединный перпендикуляр к отрезку $\overline{p_1, p_2}$. Ломаная будет строиться в виде последовательности сегментов: $\boldsymbol{\sigma} = e^+, e_1, \ldots, e_k, e^-$. Будем также предполагать, что ребра доменов диаграмм $\mathrm{Vor}(S_L)$ и $\mathrm{Vor}(S_R)$ хранятся в виде списков в порядке их обхода, соответственно, по часовой стрелке или наоборот.

Алгоритм:

$\textsc{VoronoySplittingPolyline}(S_L,\ S_R,\ \boldsymbol{\sigma})$

1 Построить верхний опорный отрезок p_L^+, p_R^+
и нижний опорный отрезок p_L^-, p_R^- для $CH(S_L)$ и $CH(S_R)$
2 $p_L \leftarrow p_L^+$
$p_R \leftarrow p_R^+$
$e \leftarrow \textsc{Bisector}(p_L,\ p_R)$
$v \leftarrow$ точка на e с большой ординатой
3 $\textsc{Push}(\boldsymbol{\sigma},\ e)$
$e_L \leftarrow head[V(p_L)]$
$e_R \leftarrow head[V(p_R)]$
4 **repeat**
5 **while** $\mathrm{I}(e, e_L) = \emptyset$ **do** $e_L \leftarrow next[e_L]$
6 **while** $\mathrm{I}(e, e_R) = \emptyset$ **do** $e_R \leftarrow next[e_R]$
7 **if** $\textsc{Dist}(v,\ \mathrm{I}(e,\ e_L)) < \textsc{Dist}(v,\ \mathrm{I}(e,\ e_R))$ **then**
8 $v \leftarrow \mathrm{I}(e,\ e_L)$
9 $p_L \leftarrow \textsc{Pair}(p_L,\ e_L)$
10 $e_L \leftarrow$ ссылка на это же ребро в новом домене $V(p_L)$
11 **else**
12 $v \leftarrow \mathrm{I}(e,\ e_R)$
13 $p_R \leftarrow \textsc{Pair}(p_R,\ e_R)$
14 $e_R \leftarrow$ ссылка на это же ребро в новом домене $V(p_R)$
15 $e \leftarrow \textsc{Bisector}(p_L,\ p_R)$
16 $\boldsymbol{\sigma} \leftarrow e$
17 **until** $p_L = p_L^-$ **and** $p_R = p_R^-$

Нетрудно заметить, что при нахождении пересечений с ломаной $\boldsymbol{\sigma}$ на шаге 5 и 6 ребра доменов просматриваются не более одного раза. Если учесть, что $\boldsymbol{\sigma}$ содержит не более $O(n)$ сегментов и суммарное число ребер всех доменов в диаграммах $\mathrm{Vor}(S_L)$ и $\mathrm{Vor}(S_R)$ также не превосходит $O(n)$, то получаем, что алгоритм строит ломаную за линейное время. Следствием этого утверждения является важный теоретический результат:

Теорема 7. *Диаграмму Вороного множества из n точек на плоскости можно построить за оптимальное время $O(n \log n)$.*

10.5. Алгоритм Форчуна

Алгоритм построения диаграммы Вороного, использующий технику заметающей прямой, был разработан намного позже рассмотренного классического алгоритма типа «разделяй и властвуй», и на то были серьезные причины. Дело в том, что в любом алгоритме заметания должна существовать возможность обнаружения критических точек *до* того, как через них проходит заметающая прямая. Например, в алгоритме Бентли – Оттмана (раздел 5.2.3) появление точки пересечения отрезков можно предсказать, просматривая соседние отрезки в структуре данных, задающей статус прямой. Для диаграммы Вороного ситуация иная: согласно общей схеме заметания выше прямой должна оставаться построенная часть диаграммы для уже пройденных точек, в то время как точка, порождающая вершину этой построенной части, может оказаться *ниже* прямой (рис. 10.10).

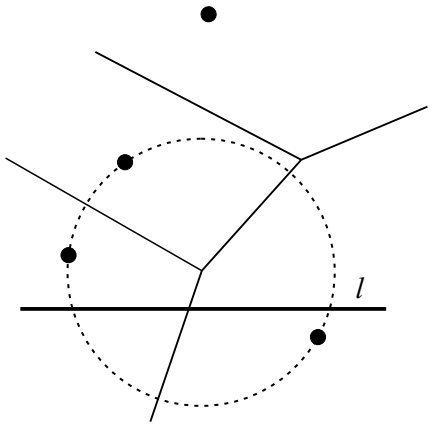


Рис. 10.10. Точка множества S, порождающая вершину построенной части диаграммы, может находиться ниже прямой

Форчун [9] решил проблему следующим образом: вместо прямого построения диаграммы Вороного он свел задачу к построению похожей геометрической структуры, из которой затем с помощью простого линейного алгоритма можно восстановить «настоящую» диаграмму Вороного. Определение этой новой структуры почти полностью совпадает с определением диаграммы Вороного (отличие выделено жирным шрифтом).

Пусть задано множество S точек на плоскости и прямая l. *Доменом Вороного – Форчуна* точки $p \in S$ называется геометрическое место точек, более близких к точке p, чем к другим точкам множества S и **прямой** l. Объединение доменов Вороного – Форчуна всех точек множества S называется *диаграммой Вороного – Форчуна* множества S.

Пусть заметающая прямая движется сверху вниз. Обозначим ее ординату через Y. Если множество S состоит из единственной точки p, то, по определению, для точки p с координатами (x_F, y_F), $y_F > Y$ домен Вороного – Форчуна представляет собой часть плоскости над параболой

$$y = \frac{1}{2}\left[\frac{(x_F - x)^2}{y_F - Y} + y_F + Y\right],$$

где точка (x_F, y_F) расположена в фокусе параболы, а заметающая прямая является ее директрисой (рис. 10.11, *а*). Для двух точек

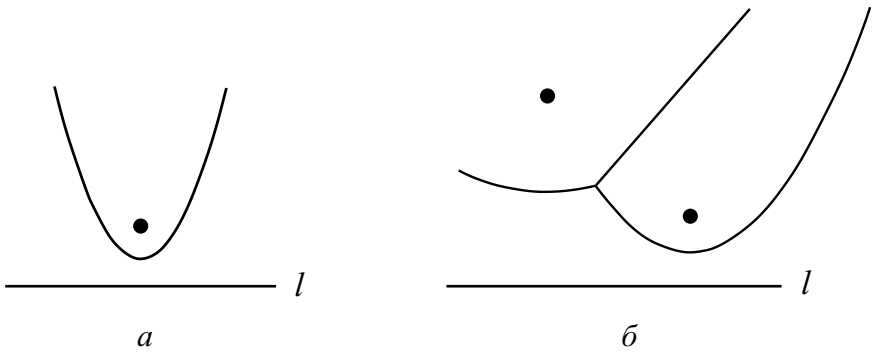


Рис. 10.11. Диаграммы Вороного – Форчуна для одной и двух точек

диаграмма усложняется, но не намного: друг от друга точки отделены серединным перпендикуляром, а от заметающей прямой – кусочно-непрерывной кривой, состоящей из двух дуг парабол и расположенной в нижней части диаграммы (рис. 10.11, *б*).

Таким образом, Форчуну удалось разрешить противоречие, о котором говорилось выше: по мере продвижения заметающей прямой

сверху вниз построенная часть диаграммы располагается целиком выше этой прямой и определяется исключительно пройденными точками. Теоретически диаграмма Вороного получается из диаграммы Вороного – Форчуна, если устремить ординату заметающей прямой в $-\infty$. На практике после прохождения всех точек множества S и построения всех вершин диаграммы нужную ее часть просто отсекают относительно прямоугольного окна, охватывающего все точки S.

Прежде чем приступить к построению диаграммы Вороного – Форчуна, исследуем ее структуру более подробно.

10.5.1. Береговая линия

Рассмотрим диаграмму на рис. 10.12. Кусочно-непрерывная кривая, задающая нижнюю границу диаграммы, имеет поэтическое название – *береговая линия* (beach line), поскольку ее форма внешне напоминает очертания морского побережья. Точки на береговой линии равноудалены от заметающей прямой и точек множества S, распо-

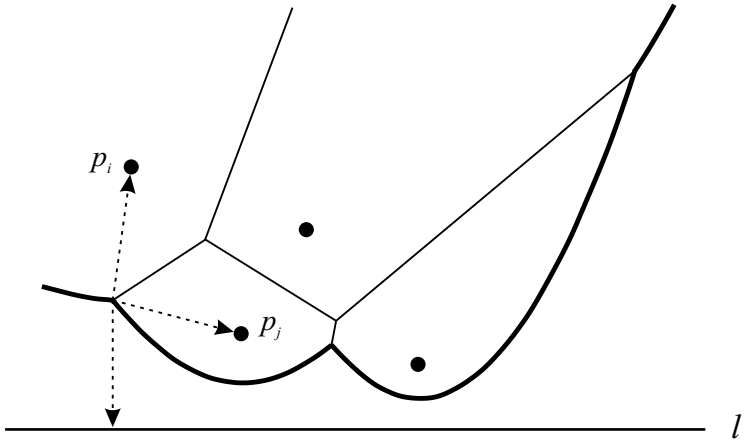


Рис. 10.12. Береговая линия и точки излома

ложенных в нижней части диаграммы. Дуги любых двух смежных парабол, образующих береговую линию, имеют общую точку, называемую *точкой излома* (break point), которая равноудалена от заметающей прямой и двух точек p_i и p_j множества S – фокусов соответствующих парабол. По определению, точка излома лежит на серединном перпендикуляре к отрезку $\overline{p_i, p_j}$, определяющему ребро в конечной диаграмме Вороного.

По мере продвижения заметающей прямой береговая линия изменяется: дуги парабол становятся менее выгнутыми, некоторые

из них увеличиваются в длине, другие, наоборот, сокращаются. Рис. 10.13 иллюстрирует этот процесс. Если фокусы p_1, p_2 и p_3 парабол соседних дуг образуют правый поворот (рис. 10.13, *а*), то при

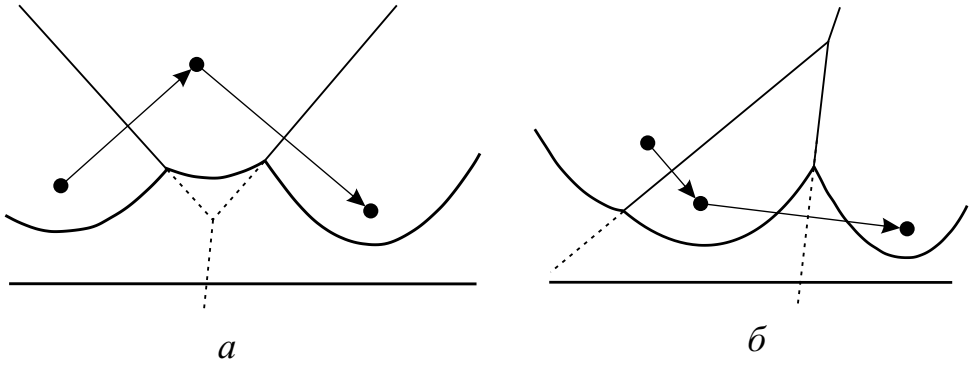


Рис. 10.13. Центральная дуга сужается, если фокусы парабол образуют правый поворот (*а*), и расширяется в противном случае (*б*)

смещении заметающей прямой вниз центральная дуга сужается и в конце концов может выродиться в точку, образовав тем самым вершину диаграммы Вороного. Однако это происходит не всегда: на пути следования заметающей прямой может оказаться другая критическая точка (такую ситуацию мы рассмотрим далее). Если p_1, p_2 и p_3 образуют левый поворот (рис. 10.13, *б*), то центральная дуга, наоборот, расширяется. В дальнейшем для краткости будем говорить о *правых* и *левых тройках* соседних дуг.

Проследим процесс образования береговой линии, начиная от точки p_1 с максимальной ординатой (случай, когда таких точек несколько, здесь не рассматривается). Предположим, что заметающая прямая l пересекла ординату точки p_1 и продолжает движение вниз (рис. 10.14, *а*). Текущая береговая линия представляет собой пока единственную параболу a_1. При достижении следующей точки p_2 создается дуга a_2 параболы с фокусом в p_2, которая в момент создания является вырожденной: она представляет собой отрезок с началом в p_2, ограниченный сверху береговой линией. Одновременно (это важно!) вместе с параболой создается ребро e_1 диаграммы Вороного, разделяющее домены точек p_1 и p_2 (рис. 10.14, *б*). В момент создания это ребро не имеет вершин и определяется только серединным перпендикуляром к отрезку $\overline{p_1, p_2}$.

Теперь нужно добавить новую дугу a_2 в береговую линию. Пока береговая линия состоит из единственной дуги a_1, это сделать

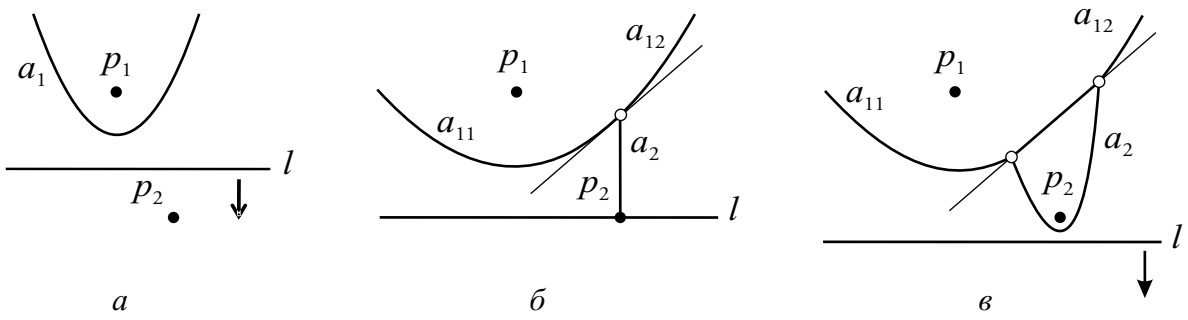


Рис. 10.14. Береговая линия: создана первая парабола (*а*), вставляется новая дуга (*б*), новая дуга расширяется по мере движения заметающей прямой (*в*)

несложно: a_1 разбивается на две части a_{11} и a_{12}, между ними вставляется новая дуга a_2, затем дуга a_1 удаляется. Заметим, что обе новые дуги a_{11} и a_{12} являются сегментами одной и той же параболы (рис. 10.14, *в*).

Далее процесс продолжается: новые точки множества S порождают новые дуги, которые вставляются в нужные места береговой линии. Здесь перед нами встают два вопроса:

1. Какие параметры дуги определяют ее положение в береговой линии?

2. Как определить дугу параболы в береговой линии, расположенную непосредственно над новой точкой p?

Ответить на первый вопрос не так просто, как может показаться на первый взгляд. Мы уже обратили внимание на то, что дуга параболы не определяется однозначно ее фокусом. И уж тем более абсцисса фокуса не определяет положение дуги в береговой линии. Второй вопрос подчеркивает необходимость разработки *эффективной* процедуры поиска нужной дуги. Действительно, в алгоритме заметания ожидается ровно n вставок новых парабол, поэтому мы вправе потратить на каждую вставку не более $O(\log n)$ времени.

Заметим, что дуги отделяются друг от друга точками излома, которые, в свою очередь, однозначно определяются ребрами диаграммы Вороного. Теперь становится понятно, почему ребра должны создаваться одновременно с созданием дуги параболы: ребра ограничивают дугу с обеих сторон и могут служить для хранения ссылок на соседние дуги. Интересным является факт, что эти ребра могут совпадать: в нашем примере новая дуга a_2 отделена от соседних дуг одним и тем же ребром e_1, которое порождает обе точки излома.

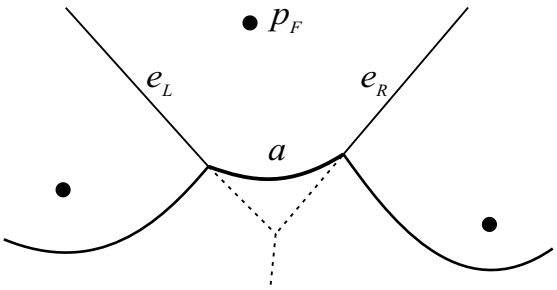


Рис. 10.15. Элементы дуги

Таким образом, дуга a параболы в составе береговой линии определяется следующими параметрами (рис. 10.15):

$p_F[a]$ – фокус (точка S),

$e_L[a]$ – ребро слева,

$e_R[a]$ – ребро справа.

Каждое ребро e содержит ссылки $p_1[e]$ и $p_2[e]$ на точки множества S, домены которых она разделяет. Это позволяет с помощью указанных параметров дуги за время $O(1)$ вычислить абсциссы точек излома, отделяющих данную дугу от соседних дуг справа и слева. Например, для вычисления абсциссы x_L левой точки излома относительно дуги a необходимо выполнить следующие действия:

```
FindLeftBreakPoint (a)
1  x_L ← nil
2  p_L ← LeftFocus(a)
3  if p_L ≠ nil then
4       x_L ← ArcIntersection(p_L, p_F[a], Y)
5  return x_L
```

Функция LeftFocus(a) в строке 2 возвращает фокус параболы дуги, расположенной слева от дуги a:

```
LeftFocus (a)
1  p ← nil
2  if p_1[e_L[a]] = p_F[a] then
3       p ← p_2[e_L]
4  else
5       p ← p_1[e_L[a]]
6  return p
```

Функция ArcIntersection$(p_L,\ p_F[a],\ Y)$ вычисляет абсциссу x_L одной из двух возможных точек пересечения двух парабол с заданными фокусами и общей директрисой. Порядок, в котором фокусы передаются в функцию, задает расположение сегментов двух парабол относительно друг друга и поэтому является существенным при вычислении нужной точки пересечения.

Таким образом, абсциссы точек излома задают отношение линейного порядка, которое можно использовать для упорядочения дуг в

составе береговой линии. Положение дуги однозначно определяется x-координатой левой точки излома, вычисляемой за время $O(1)$. Это позволяет использовать для хранения береговой линии контейнер типа *словарь*, поддерживающий вставку, удаление и поиск нужной дуги за время $O(\log n)$. Назовем этот контейнер *BeachLine*.

10.5.2. Отслеживание ребер

Итак, в начале алгоритма исходные точки множества S упорядочиваются по убыванию ординат и заносятся в очередь событий, которые называются событиями типа *точка*. Заметающая прямая движется дискретно от одного события к другому, при этом в береговую линию добавляются новые дуги.

Вернемся к нашему примеру. Мы увидели начало процесса создания ребра диаграммы, однако пока не ясно, в какой момент это ребро обретает начало и конец. Предположим, что следующая точка p_3 множества S расположена левее p_1 и ниже p_2 (рис. 10.16, *а*). Когда заметающая прямая достигает ее ординаты, создается новая дуга a_3,

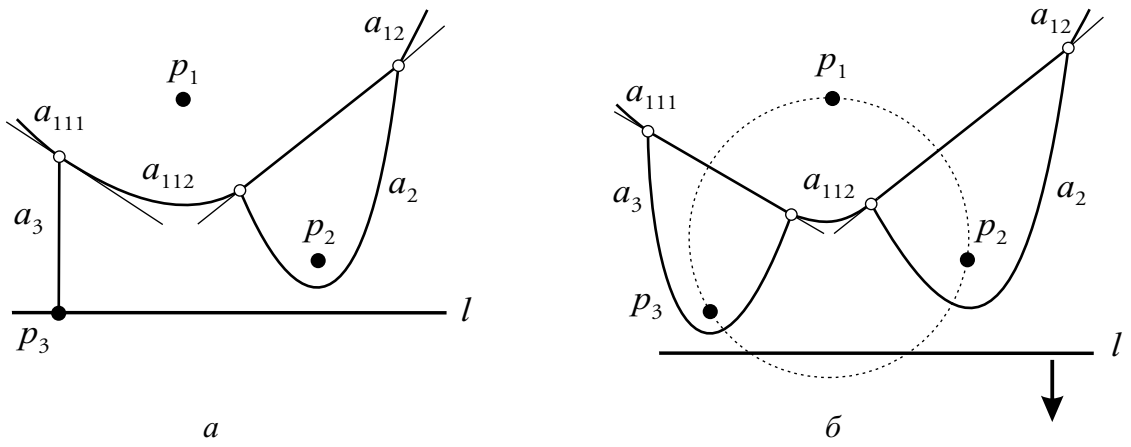


Рис. 10.16. Добавляется новая дуга a_3 (*а*). По мере приближения заметающей прямой к нижнему краю окружности дуга a_{112} сужается

которая разбивает дугу a_{11} на дуги a_{111} и a_{112} и располагается между ними. Одновременно создается очередное ребро e_2, отделяющее дугу a_3 от дуг a_{111} и a_{112}.

Продолжаем движение заметающей прямой вниз (рис. 10.16, *б*). Поскольку дуги a_3, a_{112} и a_2 образуют правую тройку, центральная дуга a_{112} непрерывно сужается и в какой-то момент вырождается в точку, образуя вершину v_1 диаграммы Вороного. Какую ординату при

этом будет иметь заметающая прямая? По определению, три фокуса p_3, p_1, p_2 и прямая l равноудалены от v_1, откуда следует, что l проходит через *нижнюю точку* описанной около p_3, p_1 и p_2 окружности с центром в v_1.

При наступлении этого события дуга a_{112} удаляется из береговой линии, точка v_1 завершает ребра e_1 и e_2 и одновременно становится началом нового ребра e_3, направление которого задает серединный перпендикуляр к отрезку p_3, p_2 (рис. 10.17). Таким образом, мы вводим

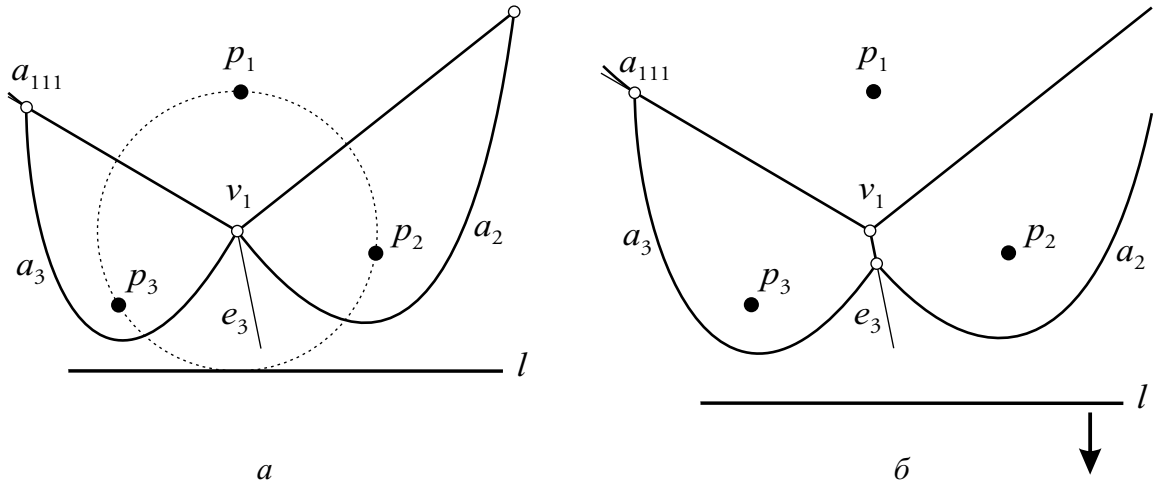


Рис. 10.17. В момент, когда заметающая прямая касается нижнего края окружности, описанной около p_3, p_1 и p_2, дуга a_{111} вырождается в вершину v_1 ($а$). Дуга a_{111} и соответствующая окружность удаляются, а точка v_1 становится началом нового ребра e_3 ($б$)

в рассмотрение событие нового типа – *окружность*. С каждым таким событием c должны быть связаны следующие параметры, которые можно вычислить за время $O(1)$ непосредственно при создании окружности:

$v[c]$	центр окружности (потенциальная вершина диаграммы),
$y[c]$	ордината нижнего края окружности,
$a[c]$	дуга, которая вырождается при наступлении события.

Каждая правая тройка соседних дуг береговой линии потенциально может породить вершину диаграммы Вороного. Поэтому, чтобы не пропустить создание вершины, всякий раз при создании новой правой тройки в очередь событий необходимо вставить соответствующее событие типа «окружность», а при разрушении существующей правой тройки – удалить событие из очереди.

Разрушение тройки происходит в двух случаях: в результате добавления новой или удаления вырожденной существующей дуги. Для быстрого удаления окружности в параметры каждой дуги a следует добавить обратные ссылки $c_i[a]$ на окружности, построенные с использованием этой дуги. Нетрудно заметить, что всего может быть не более трех ненулевых ссылок: каждая ссылка соответствует правой тройке, в которой данная дуга расположена слева, в центре или справа.

10.5.3. Алгоритм: общая схема

Теперь мы можем записать общую схему алгоритма. Очереди событий разных типов удобно вести отдельно. Обозначим их соответственно через *Sites* и *Circles*.

```
VORONOY-FORTUNE-DIAGRAM (S)
 1  Поместить точки S в очередь Sites в порядке убывания ординат
 2  while Sites ≠ NIL do
 3      if Circles ≠ NIL and y[head[Circles]] > y[head[Sites]] then
 4          CIRCLEEVENT(head[Circles])
 5      else
 6          SITEEVENT(head[Sites])
 7  while Circles ≠ NIL do
 8      CIRCLEEVENT(head[Circles])
```

Обработка окружностей в конце алгоритма требуется для построения оставшихся вершин и завершения всех конечных ребер диаграммы.

10.5.4. Обработка события типа «точка»

Процедура SITEEVENT(p) выполняет обработку событий типа «точка». Она принимает на вход точку p, формирует дугу параболы с фокусом в p, создает ребро диаграммы и вставляет дугу в береговую линию.

```
SITEEVENT (p)
 1  a ← FIND(BeachLine, p)
 2  for i = 1 to 3 do
 3      DELETE(Circles, c_i[a])
 4  e ← CREATEEDGE(p_F[a], p, 0)
 5  SPLITARC(a, e, a_1, a_2)
 6  INSERTCIRCLE(a_1)
 7  INSERTCIRCLE(a_2)
```

Функция Delete в строке 3 удаляет из очереди *Circles* все окружности, связанные с дугой a. Затем функция CreateEdge(p_1, p_2, 0) создает неограниченное ребро, определяемое серединным перпендикуляром к отрезку $\overline{p_1, p_2}$ (нуль означает, что начало ребра не задано). Разбиение дуги a выполняется вызовом процедуры SplitArc. Дуга задается фокусом параболы и двумя ребрами, ограничивающими сегмент параболы слева и справа, поэтому разбиение дуги означает простое переопределение ссылок на граничные ребра:

```
SplitArc (a, e, a1, a2)
 1  pF[a1] ← pF[a]
 2  eL[a1] ← eL[a]
 3  eR[a1] ← e
 4  pF[a2] ← pF[a]
 5  eL[a2] ← e
 6  eR[a2] ← eR[a]
```

Вызовы процедур InsertCircle вставляют в очередь *Circles* новые события типа «окружность» для всех правых троек дуг, содержащих новые дуги a_1 и a_2 (кроме тройки a_1, a, a_2). Схема процедуры представлена ниже. В ней создается окружность, образованная правой тройкой, в которой заданная дуга a расположена посередине.

```
InsertCircle (a)
 1  ap ← prev[a]
 2  an ← next[a]
 3  if RightTurn(ap, a, an) then
 4      c ← CreateCircle(ap, a, an)
 5      Insert(Circles, c)
 6      c1[ap] ← c
 7      c2[a]  ← c
 8      c3[an] ← c
```

Процедура CreateCircle(a_p, a, a_n) (строка 4) вычисляет центр и ординату нижнего края окружности, описанной вокруг фокусов заданных дуг. Затем созданная окружность вставляется в очередь событий, и в дугах устанавливаются необходимые ссылки.

10.5.5. Обработка события типа «окружность»

Напомним, что обработка окружности выполняется в момент, когда заметающая прямая достигает нижнего края этой окружности. Наступление этого события означает, что одна из дуг выродилась в вершину

диаграммы Вороного и элементы построенной части диаграммы требуют перестройки.

```
CircleEvent (c)
 1  CompleteEdge(e_L[a[c]], v[c])
 2  CompleteEdge(e_R[a[c]], v[c])
 3  p_L ← LeftFocus(a)
 4  p_R ← RightFocus(a)
 5  e ← CreateEdge(p_L, p_R, v[c])
 6  for i = 1 to 3 do
 7      Delete(Circles, c_i[a[c]])
 8  Delete(BeachLine, a[c])
 9  InsertCircle(prev[a[c]])
10  InsertCircle(next[a[c]])
```

Вначале (строки 1 и 2) завершаются ребра, ограничивающие дугу $a[c]$. Затем создается новое ребро (строки 3–5) с началом в центре окружности $v[c]$. Далее из очереди окружностей удаляются все окружности, созданные с помощью дуги $a[c]$, а сама дуга удаляется из береговой линии. Удаление дуги приводит к перестройке береговой линии, поэтому анализируются вновь образованные последовательные тройки дуг и при необходимости создаются новые окружности.

В заключение остается добавить, что все действия внутри обработчиков SiteEvent и CircleEvent требуют не более $O(\log n)$ времени, поэтому общее время работы алгоритма Форчуна не превосходит оптимальную асимптотическую оценку $O(n \log n)$.

10.6. Решение задач геометрической близости с помощью диаграммы Вороного

Задавая отношение соседства точек на плоскости, диаграмма Вороного представляет собой мощный инструмент для эффективного решения задач геометрической близости, которые мы обсуждали в разделе 10.1. Рассмотрим некоторые из них.

Ближайший сосед

При условии, что построена диаграмма Вороного, поиск ближайшего соседа можно выполнить за оптимальное время $\Theta(\log n)$. Действительно, для всех внутренних точек домена $V(p)$ ближайшим соседом,

по определению, является точка p. Остается выяснить, какому из n доменов принадлежит точка запроса z, что можно сделать за время $O(\log n)$, например, с помощью метода детализации триангуляции (см. раздел 2.4.3).

Ближайшая пара и все ближайшие пары

Ключ к решению обеих задач дает свойство 3. Для каждой точки $p \in S$ необходимо перебрать ребра соответствующего домена и найти ближайшую к ней пару. Поскольку потребуется рассмотреть не более $O(n)$ ребер, решение каждой из задач может быть найдено за линейное время (при условии, что диаграмма Вороного построена).

Евклидово минимальное остовное дерево и триангуляция

Триангуляция Делоне может быть построена как прямолинейный граф, двойственный диаграмме Вороного. Хотя для построения триангуляции этот способ не самый эффективный, он дает оптимальную асимптотическую оценку. Для евклидова минимального остовного дерева (ЕМОД) в [28] приведена модификация известного алгоритма Прима, основанная на факте, согласно которому все ребра ЕМОД принадлежат триангуляции Делоне. Поскольку число ребер триангуляции линейно, это позволило организовать эффективное построение дерева за время $O(n \log n)$.

Наибольшая пустая окружность

Напомним, что требуется найти наибольшую окружность с центром внутри выпуклой оболочки множества S, не содержащую внутри себя ни одной точки из S. Поиск решения значительно упрощается, если в качестве потенциальных центров искомой окружности рассматривать только вершины диаграммы Вороного, расположенные внутри $\mathrm{conv}S$, и точки пересечения ребер диаграммы с ребрами выпуклой оболочки. Анализ всех вариантов требует не более $O(n)$ времени, что дает общую оценку трудоемкости $O(n \log n)$.

Решение ряда других задач геометрической близости связано с обобщениями диаграммы Вороного на множества подмножеств точек на плоскости и в многомерных пространствах, а также с переходом к неевклидовым метрикам. Многочисленные варианты этих задач и их решений можно найти в монографиях [28, 3].

Упражнения

10-1. Найдите нижнюю оценку сложности построения триангуляции и евклидова минимального остовного дерева.

10-2. Что представляет собой диаграмма Вороного:

а) для множества $n > 3$ коллинеарных точек?

б) для точек, расположенных в узлах регулярной сетки, стороны которой параллельны координатным осям?

10-3. Докажите, что точка на плоскости принадлежит ребру диаграммы Вороного множества S тогда и только тогда, когда она является центром окружности, проходящей через две точки множества S и не содержащей внутри себя других точек множества S.

10-4. Докажите свойство 6, используя равенство Эйлера для плоских графов.

10-5. Напишите программу построения диаграммы Вороного на основе существующей триангуляции Делоне.

10-6. Реализуйте алгоритм типа «разделяй и властвуй». Используйте для вычисления направления обхода точек устойчивый геометрический предикат, описанный в [15].

10-7. Реализуйте алгоритм Форчуна. Выполните тесты для случаев, когда все исходные точки расположены:

а) на одной горизонтальной прямой,

б) на одной вертикальной прямой,

в) на одной окружности.

Глава 11

Оптимальные триангуляции

Рассмотрим задачу интерполяции функции двух переменных:

Задача 17. *Задано множество S из n точек на плоскости и скалярная функция f, определенная на S. Для заданного класса $\mathcal{G}$ требуется найти функцию $g \in \mathcal{G}$, определенную на* $\mathrm{conv}(S)$*, такую что $g(p) = f(p)$ для любой точки $p \in S$.*

Если рассматривать значение $f(p)$ как третью координату, то задача 17 может иметь следующую геометрическую интерпретацию: необходимо построить поверхность, проходящую через заданное множество точек в пространстве.

Класс допустимых функций $\mathcal{G}$ определяется прикладными требованиями. Например, для моделирования рельефа земной поверхности достаточно, чтобы поверхность была непрерывной. При проектировании искусственных сооружений, таких как дорожный откос или пролет моста, на поверхность налагается требование *гладкости*, то есть непрерывности вектора градиента ∇g в каждой точке поверхности. В автомобилестроении и авиационной промышленности возникает необходимость в моделировании поверхностей с непрерывной функцией кривизны.

Для решения задачи интерполяции применяется подход, основанный на идее декомпозиции одной сложной задачи на несколько простых. Выпуклая оболочка S разбивается на непересекающиеся треугольные или четырехугольные домены с вершинами в точках S, затем искомая поверхность представляется в виде объединения фрагментов поверхностей, определенных на этих локальных доменах. Такой подход с успехом применяется при решении задач из других областей, в частности, в методе конечных элементов, где решение вариационной задачи ищется как совокупность функций, каждая из которых определена на своей локальной подобласти.

Форма и размер доменов существенно влияют на качество решения в целом. Таким образом, возникает задача поиска оптимального разбиения плоскости на треугольники или четырехугольники в соответствии с некоторым критерием, учитывающим количество, форму и размер элементов разбиения.

Если точки множества S расположены в узлах регулярной сетки, то разбиение плоскости представляет собой совокупность прямоугольных или квадратных доменов со сторонами, параллельными координатным осям. Методы интерполяции по регулярной сетке значений хорошо изучены (см. главу 12), однако в такой постановке задача встречается сравнительно редко. В общем случае точки, в которых значение функции известно, могут быть расположены нерегулярно или вовсе хаотично, что требует разработки иных подходов.

Если точки распределены произвольным образом, плоскость разбивается на непересекающиеся треугольные или четырехугольные домены с вершинами в этих точках. Если разбиение состоит только из треугольных доменов, то оно называется *триангуляцией* (рис. 11.1, *а*), если только из четырехугольных, то – *квадрангуляцией* (рис. 11.1, *б*). В

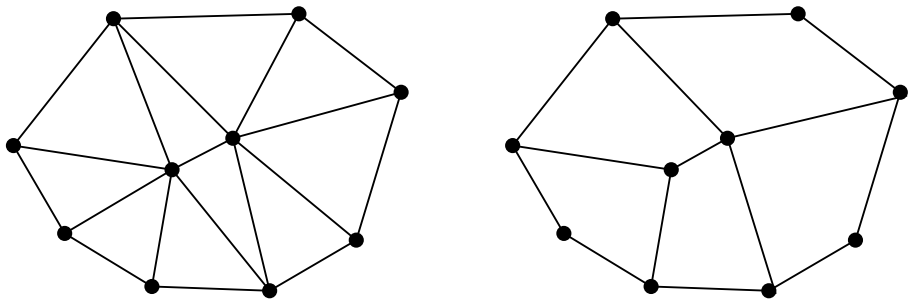


Рис. 11.1. Триангуляция (*а*) и квадрангуляция (*б*)

ряде случаев строятся разбиения смешанного типа, включающие домены обоих типов.

В следующих разделах мы рассмотрим критерии оценки качества разбиения плоскости применительно к задаче интерполяции. Будут описаны алгоритмы построения оптимальных триангуляций, не затрагивающие частные случаи, такие как триангуляции с ограничениями или триангуляции с точками Штейнера. Более полные обзоры можно найти в работах [4, 30]. Алгоритмы построения оптимальных квадрангуляций, их свойства и связь с триангуляциями описаны

в [14, 5, 18]. В статье [19] исследуются сети смешанного типа, включающие треугольные и четырехугольные домены.

11.1. Мера качества домена

Качество интерполяции существенно зависит от формы и размера доменов. Обозначим через T триангуляцию множества точек S. Пусть f – скалярная функция, определенная на conv$S$. Триангуляция T естественным образом задает кусочно-линейную непрерывную функцию g, интерполирующую f в точках множества S: каждому треугольнику триангуляции с вершинами (x_i, y_i), $1 \leq i \leq 3$, соответствует треугольник в пространстве с вершинами $(x_i, y_i, f(x_i, y_i))$, лежащий в плоскости

$$\begin{vmatrix} x & y & z & 1 \\ x_1 & y_1 & f(x_1, y_1) & 1 \\ x_2 & y_2 & f(x_2, y_2) & 1 \\ x_3 & y_3 & f(x_3, y_3) & 1 \end{vmatrix} = 0.$$

Дж. Шевчук [16] описал зависимость погрешности линейной интерполяции значений функции и ее градиента от формы и размера треугольного домена. Азевадо [2] получил аналогичные результаты для билинейной интерполяции и четырехугольных доменов. Обозначим через A площадь треугольника t, а через l_1, l_2 и l_3 – длины его сторон, упорядоченные по возрастанию. Через f и g будем обозначать функцию и ее линейную аппроксимацию над t. Тогда имеют место следующие оценки:

$$\|f - g\|_\infty \leq c_t \frac{l_{\max}^2}{6}, \tag{11.1}$$

$$\|\nabla f - \nabla g\|_\infty \leq c_t \frac{3 l_1 l_2 l_3}{4A}, \tag{11.2}$$

где c_t ограничивает сверху норму вектора второй производной f''.

Норма $\|f - g\|_\infty$ определяется как максимальная ошибка интерполяции над треугольником t:

$$\|f - g\|_\infty = \max_{p \in t} |f(p) - g(p)|.$$

Аналогично определяется ошибка вычисления градиента:

$$\|\nabla f - \nabla g\|_\infty = \max_{p \in t} |\nabla f(p) - \nabla g(p)|.$$

Если рассматривается *произвольная* функция f, то величина ошибки линейной интерполяции может быть сколь угодно большой, поэтому имеет смысл строить оценки для функций только определенного класса. Например, разумным будет предположение, что f является непрерывной и абсолютное значение второй производной f''_d по любому направлению d в любой точке треугольника t ограничено некоторой константой c_t.

Нас прежде всего интересует геометрическая интерпретация приведенных оценок. Вкратце вывод следующий: *абсолютная величина погрешности интерполяции определяется размером домена, в то время как погрешность интерполяции градиента зависит от его формы.*

В таких задачах, как вычисление векторов отражения и преломления луча в компьютерной графике, качественная аппроксимация наклона участка поверхности не менее важна, чем точность интерполяции входных данных. Поэтому рассмотрим более подробно, каким образом форма треугольника влияет на значение ошибки интерполяции градиента. Из оценки для ошибки градиента 11.3 следует, что она неограниченно возрастает при уменьшении площади треугольника, что, в свою очередь, означает, что наименее предпочтительными для интерполяции являются треугольники, содержащие тупые углы, близкие к 180°. Действительно, если зафиксировать периметр треугольника, то при стремлении тупого угла к 180° площадь A стремится к 0, в то время как длины сторон уменьшаются лишь незначительно. Заметим, что «длинные тонкие» треугольники с одним острым углом, близким к 0°, и короткой противолежащей стороной не дают аналогичного эффекта, поскольку при уменьшении острого угла площадь треугольника убывает пропорционально длине противолежащей стороны, что не приводит к общему росту оценки. Визуальная иллюстрация эффекта влияния формы треугольника на погрешность градиента приведена на рис. 11.2.

Исходя из оценок (11.1) и (11.3) можно вывести так называемую *меру качества* домена – скалярную функцию, зависящую от параметров домена и дающую возможность как сравнить два домена друг с другом, так и оценить всю сеть в целом.

В качестве примера рассмотрим три меры качества треугольного домена t, позволяющие строить качественные триангуляции с точки зрения минимизации погрешности линейной интерполяции:

$$
\begin{aligned}
M_1(t) &= \frac{A^3}{(l_1 l_2 l_3)^2}, \\
M_2(t) &= \frac{A}{(l_1 l_2 l_3)^{2/3}}, \\
M_3(t) &= \sin\theta_{\min} = \frac{2A}{l_2 l_3}.
\end{aligned}
$$

Вычислительные эксперименты показывают, что использование этих мер качества для построения триангуляции зачастую дает в результате одну и ту же триангуляцию. Поэтому при выборе меры качества принимают во внимание дополнительные критерии, такие как скорость вычисления.

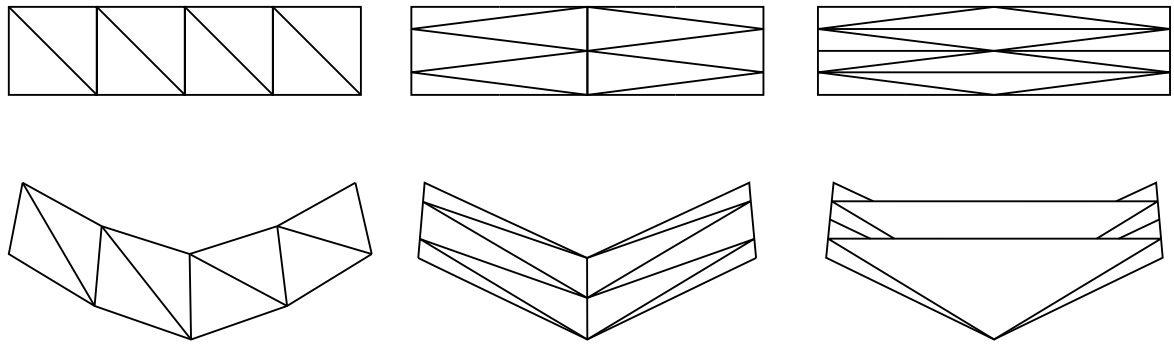


Рис. 11.2. Кусочно-линейная интерполяция точек на поверхности параболоида: наихудшее приближение градиента получено на треугольниках, содержащих тупые углы

Для нас особый интерес представляет мера $M_3(t)$, поскольку именно она имеет непосредственное отношение к алгоритмам построения триангуляции Делоне, описанным в следующих разделах.

11.2. Построение произвольной триангуляции

Прежде чем приступать к построению триангуляций с заданными свойствами, рассмотрим алгоритмы построения произвольной триангуляции, к качеству которой не предъявляется никаких требований и на которую не накладывается никаких ограничений.

Простейший алгоритм с оптимальной оценкой трудоемкости, строящий такую триангуляцию, использует обход Грехэма, описанный в разделе 3.5. Напомним, что на начальном этапе построения выпуклой

оболочки методом Грехэма точки множества S упорядочиваются в порядке возрастания полярного угла относительно крайней левой точки q. Дополним этот шаг и соединим отрезками точку q со всеми остальными точками (рис. 11.3, *а*). Далее, при обходе Грехэма будем

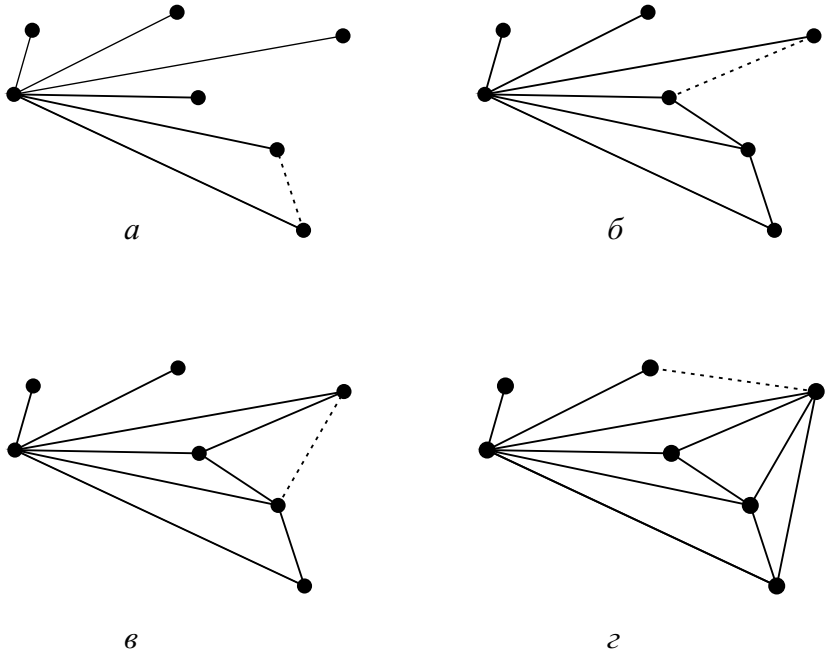


Рис. 11.3. Построение произвольной триангуляции с помощью обхода Грехэма

соединять ребром каждые две соседние точки (рис. 11.3, *б*), а также добавлять в триангуляцию новое ребро всякий раз, когда встречается правый поворот (рис. 11.3, *в, г*). Очевидно, трудоемкость обхода Грехэма остается линейной, откуда для построения произвольной триангуляции получаем алгоритм с оптимальной оценкой $\Theta(n \log n)$.

Как видно из рисунка, результатом работы алгоритма является триангуляция с длинными тонкими треугольниками, многие из которых имеют углы, близкие к 180°. Всему виной является способ упорядочения точек на начальном шаге. Даже если не принимать во внимание качество триангуляции, обработка точек, расположенных подобным образом, может являться причиной неустойчивой работы предиката, вычисляющего направление обхода для тройки последовательных точек.

Частично разрешить проблему позволяет другой способ упорядочения. Если вершины отсортированы в порядке возрастания абсцисс,

то обход Грехэма необходимо выполнять в два этапа – слева направо и обратно. Предлагаем читателю разработать модифицированный алгоритм Грехэма самостоятельно (см. упражнения).

11.3. Триангуляция Делоне

В разделе 10.3 триангуляция Делоне определена как прямолинейный граф, двойственный диаграмме Вороного. Из свойств диаграммы Вороного следует так называемое *условие Делоне*: окружность, описанная около любого треугольника триангуляции Делоне не содержит внутри себя других точек множества S. Точка (x, y) лежит строго внутри окружности, проходящей через три точки (x_1, y_1), (x_2, y_2) и (x_3, y_3), если выполняется неравенство[1]:

$$\begin{vmatrix} x & y & x^2+y^2 & 1 \\ x_1 & y_1 & x_1^2+y_1^2 & 1 \\ x_2 & y_2 & x_2^2+y_2^2 & 1 \\ x_3 & y_3 & x_3^2+y_3^2 & 1 \end{vmatrix} > 0. \qquad (11.3)$$

Из условия Делоне, в свою очередь, следует так называемое свойство *локальной равноугольности* [17]. Пусть два смежных треугольника триангуляции Делоне имеют общее ребро e, которое является диагональю выпуклого четырехугольника. Тогда значение наименьшего из шести углов обоих треугольников не уменьшится при замене диагонали e на альтернативную (рис. 11.4).

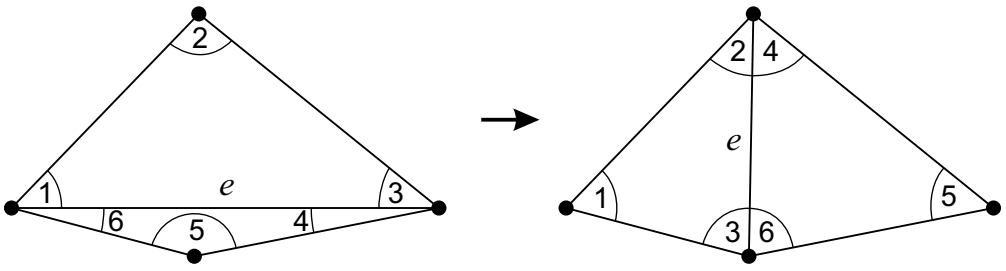


Рис. 11.4. Улучшение триангуляции по критерию Делоне: замена диагонали увеличивает наименьший из шести углов смежных треугольников

[1]Проверка этого условия лежит в основе большинства алгоритмов построения триангуляции Делоне. Однако нетрудно построить пример расположения четырех точек, когда из-за вычислительной погрешности знак определителя (11.3) вычисляется неверно. Проблеме устойчивости предиката (11.3), а также предиката, вычисляющего обход тройки точек, посвящен ряд исследований, и в настоящее время ее можно считать решенной [15, 1, 8].

Можно также доказать, что из всех триангуляций данного множества точек триангуляция Делоне имеет минимальную сумму площадей окружностей, описанных около каждого треугольника.

Существует любопытная связь между триангуляцией Делоне и трехмерной выпуклой оболочкой. Каждой точке (x, y) множества S поставим в соответствие точку (x, y, x^2+y^2) на поверхности параболоида. Выпуклая оболочка полученных точек состоит из верхней и нижней части: нижняя выпуклая оболочка включает грани, проходящие через плоскость, отделяющую точки S от бесконечно удаленной точки $(0, 0, -\infty)$ на отрицательной полуоси z. Можно доказать [6], что триангуляция Делоне множества S есть проекция нижней выпуклой оболочки на плоскость xy.

Ниже мы рассмотрим два алгоритма построения триангуляции Делоне: универсальный алгоритм замены диагоналей, позволяющий строить триангуляции с учетом различных мер качества, а также оптимальный алгоритм типа «разделяй и властвуй».

11.3.1. Алгоритм замены диагоналей

Рассмотрим произвольную триангуляцию T и выберем в ней два смежных треугольника (рис. 11.5). Если эти треугольники образуют

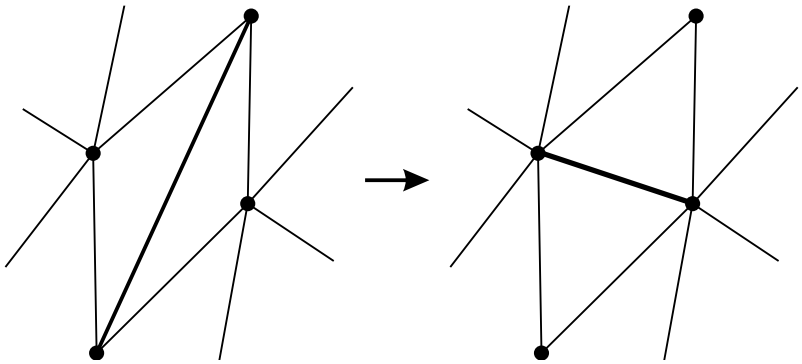


Рис. 11.5. Замена диагонали (флип)

выпуклый четырехугольник, то можно заменить его диагональ на альтернативную и тем самым перейти от триангуляции T к новой триангуляции T'. Такая операция называется *флип* (flip). Алгоритм замены диагоналей (*флип-алгоритм*) основан на улучшении существующей триангуляции с помощью последовательности флипов. Если анализ ребра показывает, что его замена приведет к улучшению

меры качества триангуляции, то флип выполняется, в противном случае ребро сохраняется на прежнем месте. Таким образом, для работы алгоритма требуется предикат Check(e), возвращающий true, если требуется замена ребра e.

В начале работы алгоритма все ребра триангуляции T помещаются в очередь с приоритетом Q. Первое ребро e извлекается из очереди, проверяется с помощью предиката Check(e) и при необходимости заменяется на альтернативное. В результате замены изменяется форма и взаимное расположение треугольников, формирующих данный четырехугольник, поэтому внешние ребра четырехугольника заносятся в очередь Q для последующей проверки. Алгоритм заканчивает работу, когда очередь не содержит ни одного ребра. Схема алгоритма представлена ниже.

```
FlipTriangulation (T)
 1  Поместить все ребра T в очередь Q
 2  while Q ≠ nil do
 3      e ← Pop(Q)
 4      if Check(e) then
 5          Flip(e)
 6          A ← GetAffectedEdges(T, e)
 7          for i ← 1 to 4 do
 8              if not Member(Q, A[i]) then
 9                  Push(Q, A[i])
```

Функция GetAffectedEdges(T, e) возвращает вектор A ребер, образующих выпуклый четырехугольник с диагональю e. Если триангуляция T хранится как реберный список с двойными связями, описанный в разделе 10.4, то указанные ребра можно получить за время $O(1)$.

Верхняя оценка работы алгоритма $O(n^2)$, но в среднем алгоритм показывает высокую производительность. Главным достоинством флип-алгоритма является его универсальность. Для реализации предиката Check(e) могут использоваться различные меры качества, в том числе рассмотренные в разделе 11.1.

Важным является следующий результат [17]. Обозначим через t_1, t_2 треугольники, смежные ребру e.

Теорема 8. *Результатом работы алгоритма замены диагоналей является триангуляция Делоне, если предикат* Check(e) *поддерживает условие локальной равноугольности, то есть определен таким образом, что замена диагонали e не уменьшает* $\min\{M_3(t_1), M_3(t_2)\}$.

11.3.2. Алгоритм типа «разделяй и властвуй»

Поскольку триангуляция Делоне может быть получена из диаграммы Вороного за линейное время, вопрос о существовании оптимального алгоритма не является актуальным. Тем не менее такой подход вряд ли можно считать удовлетворительным: триангуляция является более простой структурой, нежели диаграмма Вороного, значит и методы ее построения должны быть более простыми.

Описанный в [10] алгоритм строит триангуляцию в соответствии с классической схемой «разделяй и властвуй», то есть выполняет декомпозицию множества точек S на два примерно равных по мощности подмножества S_1 и S_2, рекурсивно строит триангуляции $T(S_1)$ и $T(S_2)$ и затем объединяет построенные триангуляции в одну триангуляцию $T(S)$. Чтобы общее время работы алгоритма не превысило $O(n \log n)$, шаг слияния должен быть выполнен за линейное время.

Предположим, что множества S_L и S_R линейно разделимы вертикальной прямой (рис. 11.6). Шаг слияния можно представить как

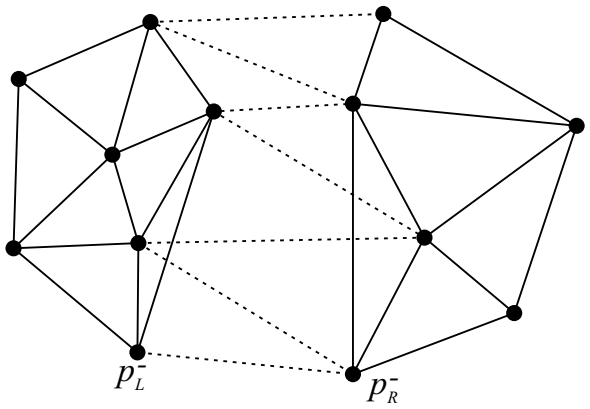


Рис. 11.6. Сшивка двух триангуляций Делоне

сшивку этих триангуляций по мере продвижения вдоль разделяющей прямой снизу вверх. В процессе сшивки часть существующих ребер может быть удалена, а часть новых добавлена. Вставка ребер начинается с нахождения нижней опорной прямой множества S, проходящей через точки $p_L^- \in S_L$ и $p_R^- \in S_R$. Ребро вставляется в триангуляцию и считается текущим *опорным* ребром. Далее новое опорное ребро вычисляется в три этапа:

1. Найти точку $q_L \in S_L$ такую, что для треугольника $t_L = (q_L, p_L, p_R)$ относительно множества S_L выполняется условие Делоне, то есть

окружность, описанная около t_L, не содержит других точек S_L (рис. 11.7, *а*).

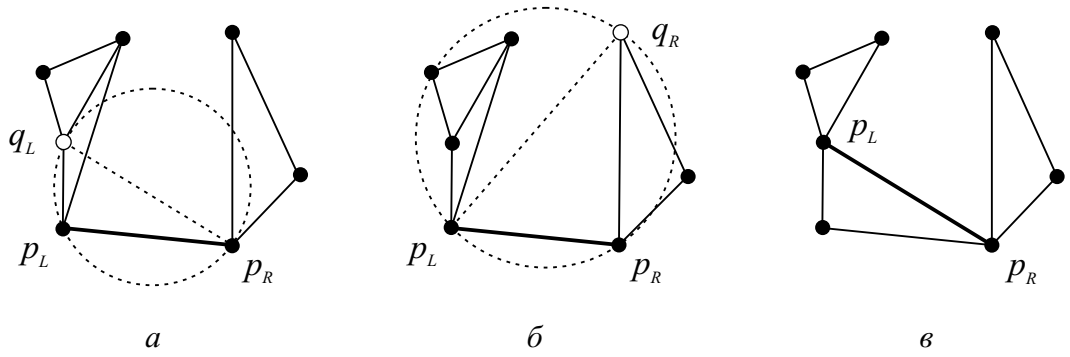


Рис. 11.7. Сшивка двух триангуляций: поиск нового опорного ребра

2. Аналогичным образом найти точку $q_R \in S_R$ такую, что для треугольника $t_R = (q_R, p_L, p_R)$ относительно множества S_R также выполняется условие Делоне (рис. 11.7, *б*).

3. Выбрать в качестве нового опорного ребра (q_L, p_R), если точка q_L лежит за пределами окружности, описанной около треугольника t_R, или (q_R, p_L), если q_R лежит за пределами окружности, описанной около t_L (рис. 11.7, *в*).

Сшивка заканчивается построением последнего опорного ребра, лежащего на верхней опорной прямой (p_L, p_R) множества S.

Последовательный анализ вершин-кандидатов для образования нового опорного ребра может потребовать в сумме $O(n^2)$ времени, если проверку условия Делоне относительно ребра $\overline{p_L, p_R}$ выполнять для всех точек S_L и S_R. К счастью, в этом нет необходимости: при поиске q_L требуется проверять только точки, смежные p_L, а при поиске q_R – только точки, смежные p_R. Мало того, не исключено, что потребуется просмотр не всех таких точек. В примере на рис. 11.8 условие Делоне проверяется сначала для точки b относительно треугольника (p_L, a, p_R). Если условие выполняется, то ребро $\overline{p_L, a}$ удаляется из $T(S_L)$. В противном случае следующая проверка выполняется для точки c относительно треугольника (p_L, b, p_R) и т. д. Проверки прекращаются, если для очередной точки выполняется условие Делоне или эта точка расположена правее направленной прямой (p_L, p_R).

Схема алгоритма приведена ниже. Ребра, «сшивающие» триангуляции $T(S_L)$ и $T(S_R)$, хранятся в очереди E.

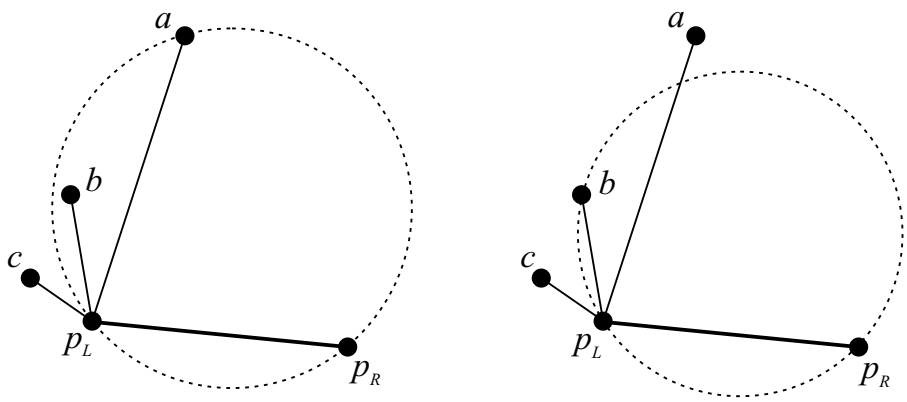


Рис. 11.8. Проверка условия Делоне для точек, смежных p_L

DivideAndConquerDelaunay (S)

1 Разбить множество S на S_L и S_R по x-медиане
2 $T_L \leftarrow$ BuildDelaunay(S_L)
3 $T_R \leftarrow$ BuildDelaunay(S_R)
4 Для conv(S_L) и conv(S_R) построить
нижний опорный отрезок $\overline{p_L^-, p_R^-}$ и
верхний опорный отрезок $\overline{p_L^+, p_R^+}$
5 $p_L \leftarrow p_L^-$
$p_R \leftarrow p_R^-$
6 Создать ребро $e = \overline{p_L, p_R}$
7 Insert(E, e)
8 **while** $p_L \neq p_L^+$ **and** $p_R \neq p_R^+$ **do**
9 $q_L \leftarrow head[$Adjacent$(T_L, p_L)]$
10 **while** $next[q_L] \neq$ nil **and not** IsValid(q_L, p_L, p_R) **do**
11 Delete$(T_L, \overline{p_L, q_L})$
12 $q_L \leftarrow next[q_L]$
13 $q_R \leftarrow head[$Adjacent$(T_R, p_R)]$
14 **while** $next[q_R] \neq$ nil **and not** IsValid(q_R, p_L, p_R) **do**
15 Delete$(T_R, \overline{p_R, q_R})$
16 $q_R \leftarrow next[q_R]$
17 **if** InCircle(q_R, q_L, p_L, p_R) **then**
18 $p_R \leftarrow q_R$
19 **else**
20 $p_L \leftarrow q_L$
21 Создать ребро $e = \overline{p_L, p_R}$
22 Insert(E, e)
23 $T \leftarrow T_L + T_R + E$
24 **return** T

Функция ADJACENT(T, p) возвращает список вершин, смежных p в триангуляции T. Вершины в списке упорядочены против часовой стрелки для T_L и по часовой стрелке для T_R. Функция ISVALID(q, p_L, p_R) определяет, может ли точка q быть выбрана в качестве вершины опорного ребра $\overline{p_L, p_R}$:

```
ISVALID (q, p_L, p_R)
 1  r ← next[q]                    ▷ вершина, следующая за q в списке
                                   ▷ вершин, смежных p_L или p_R
 2  return
 3      LEFTTURN(r, p_L, p_R) and
 4      INCIRCLE(r, q, p_L, p_R)
```

Функции LEFTTURN и INCIRCLE определяют, соответственно, ориентацию и условие Делоне (11.3) для заданных точек.

Число точек, смежных вершинам опорных ребер, в процессе слияния не превосходит удвоенного числа ребер обеих триангуляций, поэтому этот шаг может быть выполнен за линейное время.

На этапе декомпозиции необходимо учитывать характер распределения точек множества S. Часто удачной стратегией является поочередное разбиение множества S по осям абсцисс и ординат на каждой итерации.

11.4. Триангуляция минимального веса

Триангуляция, сумма длин ребер которой минимальна для данного множества точек, называется *триангуляцией минимального веса*, или *минимальной триангуляцией*.

Рассмотрим следующую меру качества треугольного домена:

$$M_4 = l_1 + l_2 + l_3.$$

Соответствующий предикат CHECK(e) для флип-алгоритма возвращает TRUE, если существующая диагональ выпуклого четырехугольника длиннее альтернативной. Мера M_4 не учитывает величину угла треугольника, что непосредственно отражается на ее пригодности для интерполяции поверхности: как показано на рис. 11.9, триангуляция минимального веса может содержать углы, сколь угодно близкие к 180°, откуда следует, что ошибка интерполяции градиента может быть чрезмерно большой.

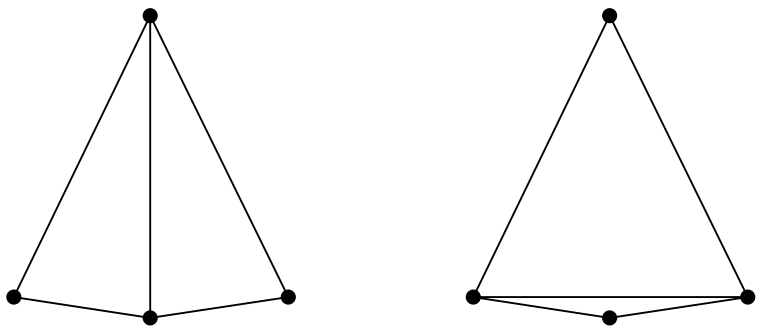


Рис. 11.9. В отличие от триангуляции Делоне (*а*) триангуляция минимального веса (*б*) может содержать треугольники с углами, сколь угодно близкими к 180°

Минимальная триангуляция вызывает у математиков самый живой интерес в связи с проблемой иного рода. В известной монографии Гэри и Джонсона [23], вышедшей в 1979 году, сформулирована следующая задача распознавания:

ТРИАНГУЛЯЦИЯ МИНИМАЛЬНОЙ ДЛИНЫ
Заданы набор $C = \{(a_i, b_i) : 1 \leq i \leq n\}$ *пар целых чисел (определяющих координаты* n *точек на плоскости) и положительное целое число* B*. Существует ли триангуляция множества точек, определяемого набором* C*, общая «дискретно-евклидова» длина которого не превосходит* B*?*

С той поры прошло не одно десятилетие, но, несмотря на самые серьезные исследования, проведенные силами мирового математического сообщества, задача по-прежнему остается открытой: не доказана ее NP-полнота, ровно как не найден полиномиальный алгоритм ее решения.

Алгоритм замены диагоналей с мерой качества домена M_4 и «жадный» алгоритм, добавляющий в триангуляцию ребра, отсортированные по возрастанию длины, в общем случае могут построить минимальную триангуляцию или близкую к ней. При этом легко построить пример, когда «жадный» алгоритм дает на выходе неоптимальное решение, неулучшаемое с помощью алгоритма замены диагоналей (рис. 11.10).

Непригодность простых градиентных алгоритмов была обоснована в работе [13], в которой построен пример расположения исходных точек,

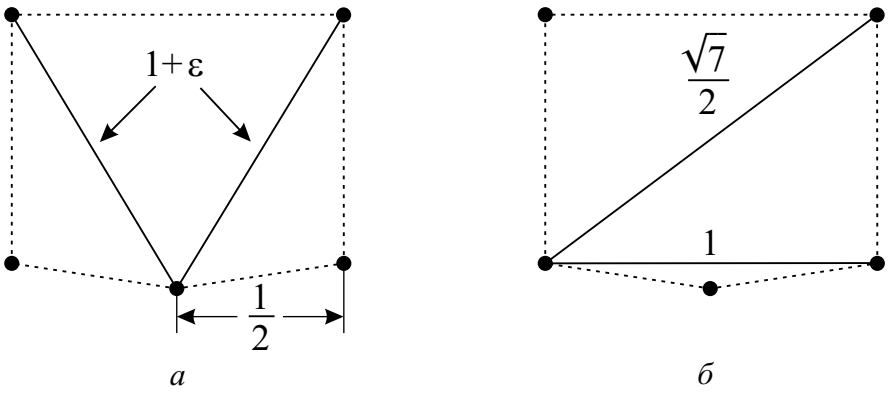


Рис. 11.10. Триангуляция минимального веса (*а*) и «жадная триангуляция» (*б*) с суммарной длиной внутренних ребер, соответственно, $2 + 2\varepsilon$ и $1 + \frac{\sqrt{7}}{2} \approx 2.33$, где ε – сколь угодно малое положительное число

такой что жадный алгоритм генерирует триангуляцию с суммарной длиной ребер в $O(n)$ больше минимальной.

Указанные негативные результаты относятся к общему случаю, когда множество S может быть произвольным. Если исходные точки расположены в вершинах выпуклого многоугольника, то задача значительно упрощается и имеет полиномиальное решение.

Алгоритм, предложенный в работе [11], основан на технике динамического программирования и непосредственно связан с задачей о порядке перемножения матриц (см. [25], глава 16). Пусть $P = v_0, \ldots, v_n$ – выпуклый многоугольник. Любая его триангуляция содержит все ребра выпуклой оболочки, поэтому длиной триангуляции будем считать суммарную длину хорд (внутренних ребер), соединяющих несоседние вершины. Для удобства для двух вершин $v_i, v_j \in P$ введем функцию $d_{i,j}$, равную расстоянию между v_i и v_j, если они несмежные, и равную 0 в противном случае. Отметим два важных факта, справедливых для триангуляции многоугольника с более чем тремя вершинами:

1. Любому ребру P смежна по крайней мере одна хорда.
2. Для любой хорды (v_i, v_j) существует вершина $v_k \in P$, такая что каждый из отрезков (v_i, v_k) и (v_k, v_j) является либо хордой, либо ребром P.

Обозначим через $w[i, j]$ длину минимальной триангуляции T_{ij} многоугольника $v_i, \ldots, v_j$, $0 \leq i < j \leq n$. Тогда длина минимальной

триангуляции P равна $w[0,n]$. Из приведенных фактов следуют рекуррентные формулы для вычисления $w[i,j]$:

$$w[i,j] = \begin{cases} 0, & i = j, \\ \min_{j<k<j}\{w[i,k] + w[k,j] + d_{i,k} + d_{k,j}\}, & j < j. \end{cases}$$

Действительно, любая триангуляция многоугольника P, включая минимальную, должна содержать треугольник t с основанием v_0, v_n и некоторой вершиной v_k (рис. 11.11). Значит, длина минимальной триангуляции будет состоять из суммарной длины ребер $\overline{v_0, v_k}$ и $\overline{v_k, v_n}$ треугольника t плюс длины минимальных триангуляций T_{0k} и T_{kn} многоугольников, примыкающих к t слева и справа. Схема алгоритма представлена ниже.

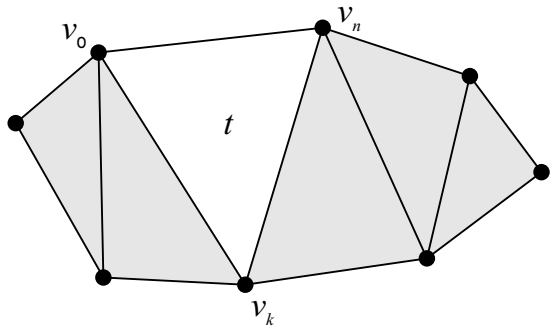


Рис. 11.11. Минимальная триангуляция содержит треугольник t и две примыкающие к нему минимальные триангуляции смежных многоугольников

```
MinWeightTriangulation (P)
 1  for i ← 0 to n do
 2      w[i,i] ← 0
 3      w[i,i+1] ← 0
 4      w[i,i+2] ← 0
 5  for m ← 3 to n do
 6      for i ← 0 to n − m
 7          j ← i + m − 1
 8          w[i,j] ← ∞
 9          for k ← i+1 to j − 1 do
10              q ← w[i,k] + w[k,j] + d(i,k) + d(k,j)
11              if q < w[i,j] then
12                  w[i,j] ← q
13                  e[i,j] ← k
14  return e
```

Вычисление длины $w[1,n]$ всей триангуляции организовано последовательно от вычисления $w[i,j]$ для всех пар вершин v_i, v_j, лежащих через две вершины друг от друга, затем через три и так далее (строка 5). Для каждой пары значения $w[i,j]$ запоминаются и используются на

следующей итерации для вычисления длины минимальной триангуляции большего размера.

Для пары вершин (v_i, v_j) элемент матрицы $e[i, j]$ хранит номер оптимальной вершины, соответствующей треугольнику с основанием $\overline{v_i, v_j}$. После окончания работы алгоритма множество E ребер минимальной триангуляции может быть получено с помощью вызова следующей рекурсивной процедуры:

```
RetrieveEdges (e, i, j, E)
 1  k ← e[i, j]
 2  if i < k − 1 then                    ▷ если v_i v_k – хорда
 3      Push(E, v_i v_k)
 4      RetrieveEdges(e, i, k, E)
 5  if k < j − 1 then
 6      Push(E, v_k v_j)
 7      RetrieveEdges(e, k, j, E)
```

Очевидно, алгоритм имеет полиномиальную сложность: время работы не превосходит $O(n^3)$ при затратах памяти $O(n^2)$.

Упражнения

3-1. Реализуйте модифицированный алгоритм Грехэма для построения произвольной триангуляции, описанный в разделе 11.2.

3-2. Доказать, что если на вход флип-алгоритма подается «жадная» триангуляция, то не будет произведено ни одной замены диагоналей.

3-3. *Графом Габриэля* множества точек S на плоскости называется прямолинейный граф, такой что для любого его ребра e окружность с диаметром e не содержит других точек множества S. Доказать, что ребра графа Габриэля являются ребрами триангуляции Делоне множества S.

3-4. *Графом относительного соседства* множества точек S на плоскости называется прямолинейный граф, такой что для любого его ребра e пересечение двух окружностей с радиусом e и центрами в вершинах этого ребра не содержит точек множества S. Доказать, что ребра графа относительного соседства являются ребрами графа Габриэля множества S.

Глава 12

Моделирование кривых и поверхностей

Эта глава является введением в область геометрического моделирования, называемую *конструированием форм*. В отличие от *моделирования сплошных тел* конструирование форм изучает модели объектов, представленных в виде поверхности, ограничивающей *полое* тело. Построение объектов такого типа производится в два этапа.

1. Создается *каркасная* (или *проволочная*) модель объекта, представляющая собой точки в пространстве вместе с соединяющими их отрезками или сегментами кривых (рис. 12.1). Методы построения каркасных моделей определяются областью применения. В частности, в топографических системах пространственное положение точек определяется на основе полевых измерений, а построение каркасной модели (триангуляции) производится автоматически. В машиностроении каркасные модели деталей создаются проектировщиком в интерактивном режиме, накапливаются в геомет-

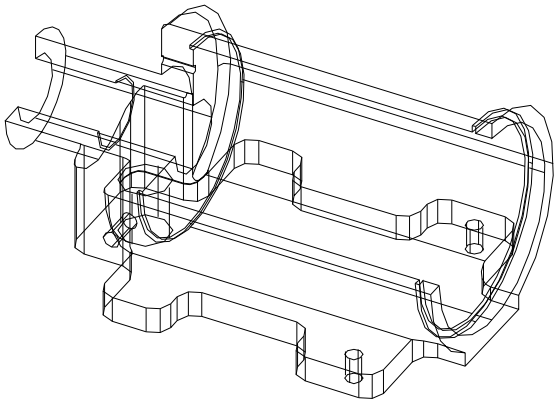


Рис. 12.1. Каркасная модель кожуха генератора

рической базе данных и служат затем для проектирования более сложных узлов.

2. На втором этапе аппроксимируется форма каждого куска поверхности, ограниченного кривыми каркасной модели. Коэффициенты функции, описывающей форму куска, вычисляются на основе граничных условий и требований к степени гладкости.

Если нас не интересуют физические характеристики объекта, то конструирование формы является наиболее удобным способом его представления для последующего графического отображения.

Существует два основных способа представления поверхностей в пространстве: в виде полигональной сетки и в виде параметрических нелинейных кусков. *Полигональная сетка* представляет собой совокупность ребер кусочно-линейной поверхности, состоящей из смежных плоских многоугольников. Частным случаем полигональной сетки является триангуляция (рис. 12.2). Полигональные сетки

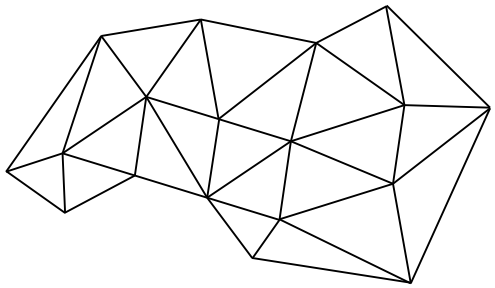


Рис. 12.2. Пример полигональной сетки

применяются как для точного описания формы многогранных поверхностей, так и для аппроксимации нелинейных поверхностей.

Параметрические нелинейные куски описывают координаты точек на поверхности с помощью трех уравнений (по одному для каждой координаты x, y и z). Границами кусков являются параметрические кривые (возможно, отрезки). Для представления поверхности с заданной точностью требуется значительно меньшее число нелинейных кусков, чем при аппроксимации полигональной сеткой.

В следующих разделах рассмотрены основные методы построения кривых и поверхностей в пространстве.

12.1. Представление кривых в пространстве

Существуют два основных способа представления кривых:

- как функций одной из переменных x, y или z;
- как функций независимого параметра t (параметрическое представление).

В первом случае одна из координат – например x – выбирается в качестве независимой переменной, и точка на кривой описывается в виде

$$x = x,\ y = y(x),\ z = z(x),\ x_{\min} \le x \le x_{\max}. \tag{12.1}$$

Такой способ обладает существенным недостатком: невозможно описать кривую, если касательная к ней в какой-либо точке перпендикулярна оси независимой переменной. По этой причине невозможно описать замкнутую или неоднозначную относительно независимой переменной кривую.

Параметрическое представление произвольного криволинейного сегмента P включает три функции независимого параметра t:

$$P(t) = (x(t), y(t), z(t)), \quad 0 \le t \le 1.$$

При таком представлении легко вычисляются координаты касательного вектора в произвольной точке $P(t_0)$ сегмента

$$R(t_0) = (x'(t_0), y'(t_0), z'(t_0)).$$

Кривая P называется *гладкой* на отрезке $0 \le t \le 1$, если функция $R(t)$ непрерывна в любой точке этого отрезка.

Наша цель описать гладкую кривую, состоящую из последовательности сегментов. Непрерывность кривой в точке соединения сегментов P_1 и P_2 обеспечивается условием

$$P_1(1) = P_2(0),$$

а ее гладкость в этой точке определяется сонаправленностью касательных векторов

$$R_1(1) = kR_2(0), \quad k > 0.$$

12.1.1. Общее представление кубических кривых

Ниже описаны методы построения параметрических кривых, имеющих общее представление

$$\begin{aligned} x(t) &= a_x t^3 + b_x t^2 + c_x t + d_x, \\ y(t) &= a_y t^3 + b_y t^2 + c_y t + d_y, \\ z(t) &= a_z t^3 + b_z t^2 + c_z t + d_z \end{aligned}$$

или в матрично-векторной форме

$$P(t) = TC, \tag{12.2}$$

где

$$T = (t^3, t^2, t, 1) \quad \text{и} \quad C = \begin{pmatrix} a_x & a_y & a_z \\ b_x & b_y & b_z \\ c_x & c_y & c_z \\ d_x & d_y & d_z \end{pmatrix}.$$

В таких обозначениях касательный вектор в точке t задается функцией $R(t) = \nabla TC$, где $\nabla T = (3t^2, 2t, 1, 0)$. Столбцы матрицы C будем обозначать, соответственно, C_x, C_y и C_z.

Описанные ниже методы построения кубических кривых основаны на различных формах представления исходных данных. Мы рассмотрим три из них: форму Эрмита, форму Безье и форму B-сплайнов.

12.1.2. Форма Эрмита

Исходными данными для построения формы Эрмита являются концевые точки сегмента и касательные вектора в этих точках (рис. 12.3). Требуется на основе этой информации найти

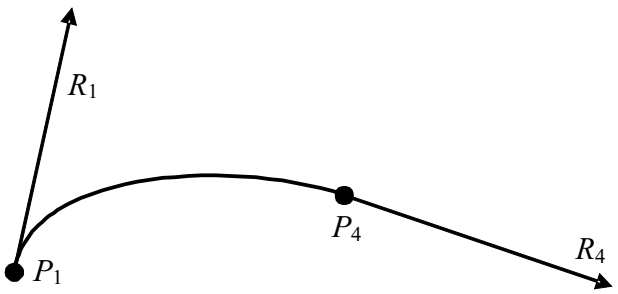


Рис. 12.3. Кубическая кривая в форме Эрмита

коэффициенты C_x, C_y и C_z кубических полиномов $x(t)$, $y(t)$ и $z(t)$.

Обозначим через P_1 и P_4 концевые точки сегмента, а через R_1 и R_4 касательные вектора в этих точках. Тогда начальные условия для вычисления коэффициентов кривой имеют вид

$$x(0) = P_{1x}, \quad x(1) = P_{4x}, \quad x'(0) = R_{1x}, \quad x'(1) = R_{4x}.$$

Выражения для $x(t)$ и $x'(t)$ запишем в матричном виде

$$x(t) = (t^3, t^2, t, 1)C_x.$$

Используя это представление, получим систему уравнений

$$\begin{array}{rcl} x(0) & = & (0,0,0,1)C_x \\ x(1) & = & (1,1,1,1)C_x \\ x'(0) & = & (0,0,1,0)C_x \\ x'(1) & = & (3,2,1,0)C_x \end{array} \quad \text{или} \quad \begin{pmatrix} P_1 \\ P_4 \\ R_1 \\ R_4 \end{pmatrix}_x = \begin{pmatrix} 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 \\ 3 & 2 & 1 & 0 \end{pmatrix} C_x,$$

откуда

$$C_x = \begin{pmatrix} 2 & -2 & 1 & 1 \\ -3 & 3 & -2 & -1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} P_1 \\ P_4 \\ R_1 \\ R_4 \end{pmatrix}_x = M_h G_{hx}.$$

Матрица M_h называется *матрицей Эрмита*, а вектор G_h – *геометрическим вектором Эрмита*. Для $y(t)$ и $z(t)$ справедливы аналогичные соотношения

$$C_y = M_h G_{hy} \text{ и } C_z = M_h G_{hz}.$$

Подставив их в (12.2), получим выражение для кубической формы Эрмита

$$P(t) = T M_h G_h.$$

Гладкость сплайновой кривой в месте сопряжения двух сегментов S^1 и S^2 достигается при условии, что выполняется соотношение $R_4^1(1) = kR_1^2(0)$, $k > 0$, где R_j^i – касательный вектор сегмента i в точке P_j. На рис. 12.4 изображены кривые Эрмита с сонаправленными касательными векторами разной длины.

12.1.3. Форма Безье

Для построения кубической формы Безье необходимо указать четыре управляющие точки сегмента P_1, P_2, P_3 и P_4. Эти точки называются

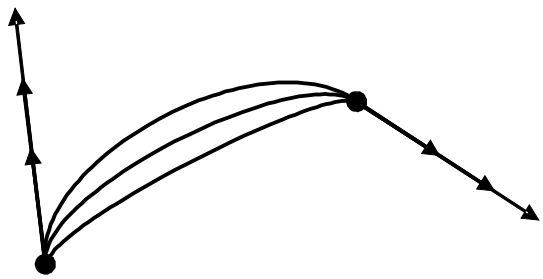


Рис. 12.4. Изменение длины касательного вектора влечет изменение формы кривой Эрмита, сохраняя ее наклон в концевой точке

точками Безье. Форма Безье отличается от формы Эрмита способом задания касательных векторов в конечных точках: направления касательных задаются отрезками (P_1, P_2) и (P_3, P_4). При этом имеют место равенства

$$R_1 = 3(P_2 - P_1) \text{ и } R_4 = 3(P_4 - P_3),$$

откуда легко найти соотношение между геометрической матрицей Безье G_b и геометрической матрицей Эрмита G_h

$$G_h = \begin{pmatrix} P_1 \\ P_4 \\ R_1 \\ R_4 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ -3 & 3 & 0 & 0 \\ 0 & 0 & -3 & 3 \end{pmatrix} = \begin{pmatrix} P_1 \\ P_2 \\ P_3 \\ P_4 \end{pmatrix} = M_{hb} G_b.$$

Подставим G_b в матричное выражение для $x(t)$

$$x(t) = T M_h G_{hb} = T M_h M_{hb} G_{bx}.$$

Обозначив $M_b = M_h M_{hb}$, получим выражение для формы Безье

$$x(t) = T M_b G_{bx},$$

где матрица

$$\begin{pmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix}$$

называется *матрицей Безье*.

Гладкость кривой Безье в месте сопряжения двух сегментов обеспечивается при условии сонаправленности касательных векторов в точке сопряжения

$$P_3^1 P_4^1 = k P_1^2 P_2^2,\ k > 0.$$

Преимуществом представления кубической кривой в форме Безье является более естественная связь между исходными данными (точками Безье) и формой кривой. Рис. 12.5 иллюстрирует зависимость формы кривой Безье от расположения управляющих точек.

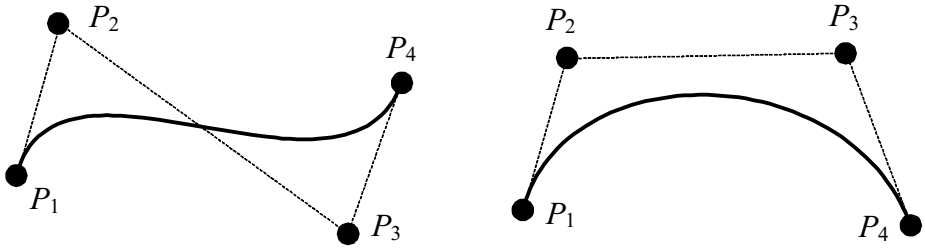


Рис. 12.5. Кривая Безье аппроксимирует ломаную, задаваемую управляющими точками

Кроме того, кривая Безье обладает следующим свойством: она целиком содержится внутри выпуклой оболочки точек Безье. Это свойство используется при проверке пересечения кривых и при отсечении: сначала проверяется, пересекаются ли выпуклые оболочки, и только затем, в случае их пересечения, вычисляются точки пересечения самих кривых.

В общем случае кривая Безье представляет собой полином порядка $n-1$, где n – число управляющих точек Безье:

$$P(t) = \sum_{i=0}^{n-1} P_i B_i^n(t),$$

где через $B_i^n(t)$ обозначены так называемые *полиномы Бернштейна*

$$B_i^n(t) = \left(\begin{array}{c} n \\ i \end{array} \right) t^i (1-t)^{n-i}.$$

Нетрудно убедиться, что значение $n = 4$ соответствует описанному выше представлению кубической кривой Безье.

12.1.4. Форма B-сплайнов

Для кривой, представленной в форме B-сплайнов, непрерывной является не только функция касательной $P'(t) = R(t)$, но и функция кривизны $P''(t)$. Таким образом, B-сплайн обладает большей степенью гладкости, нежели кривые Эрмита и Безье. Другим отличием является то, что B-сплайн является *аппроксимационным* сплайном (траектория кривой необязательно проходит через управляющие точки), тогда как кривые в форме Эрмита и Безье являются *интерполяционными* сплайнами (прохождение через управляющие точки обязательно). И наконец, третье отличие заключается в том, что B-сплайн аппроксимирует *произвольное* количество управляющих точек. Кривая в форме B-сплайна представлена на рис. 12.6. Сегмент B-сплайновой

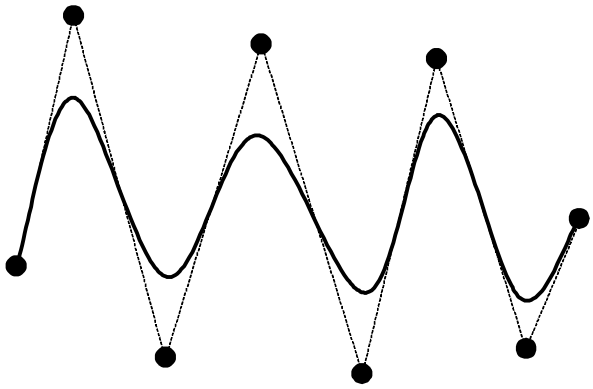


Рис. 12.6. Кривая в форме B-сплайна

кривой описывается формулой

$$P(t) = TM_sG_s,$$

где

$$M_s = \frac{1}{6}\begin{pmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 0 & 3 & 0 \\ 1 & 4 & 1 & 0 \end{pmatrix}.$$

B-сплайн аппроксимирует управляющие точки $P_1, \ldots, P_n$, и при этом каждой паре P_iP_{i+1} соседних точек соответствует своя геометрическая матрица G_s^i

$$G_s^1 = \begin{pmatrix} 2P_1 - P_2 \\ P_1 \\ P_2 \\ P_3 \end{pmatrix},\ G_s^i = \begin{pmatrix} P_{i-1} \\ P_i \\ P_{i+1} \\ P_{i+2} \end{pmatrix},\ 2 \le i \le n-2,$$

$$G_s^{n-1} = \begin{pmatrix} P_{n-2} \\ P_{n-1} \\ P_n \\ 2P_n - P_{n-1} \end{pmatrix}.$$

Как и кривая в форме Безье, B-сплайновый сегмент и вся составная кривая принадлежат выпуклой оболочке своих управляющих точек.

12.2. Построение поверхностей

Представление поверхностей также основано на введении осенезависимых параметров s и t, что обеспечивает однозначное представление замкнутых и многозначных поверхностей. Координаты точки на параметрической поверхности задаются вектором

$$P(s,t) = (x(s,t), y(s,t), z(s,t)),\ 0 \le s,t \le 1.$$

Если один из параметров является функцией другого, например, $t = f(s)$, то выражение $P(s, f(s))$ описывает кривую на поверхности.

В качестве исходных данных для построения куска параметрической поверхности используются граничные условия – кривые, задающие контур куска поверхности, и параметры поверхности в вершинах контура – координаты вершин, наклон, кривизна, кручение. Ниже будут рассмотрены кусочно-непрерывные поверхности (билинейные, линейчатые и линейные куски Кунса) и кусочно-гладкие поверхности со свойством непрерывности первых производных на границах сегментов (кубические куски Кунса и бикубические поверхности).

12.2.1. Билинейные поверхности

Исходными данными для построения билинейной поверхности являются четыре прямолинейных отрезка, в общем случае не принадлежащих одной плоскости и образующих замкнутый контур в пространстве. Угловые точки контура p_1, p_2, p_3 и p_4 совпадают с точками, соответственно, $P(0,0)$, $P(0,1)$, $P(1,0)$ и $P(1,1)$ поверхности

$P(s,t)$. Линейно проинтерполируем по параметру t противолежащие граничные отрезки $[P(0,0),P(0,1)]$ и $[P(1,0),P(1,1)]$:

$$\begin{aligned} P(0,t) &= tP(0,1)+(1-t)P(0,0), \\ P(1,t) &= tP(1,1)+(1-t)P(1,0). \end{aligned}$$

Теперь проинтерполируем по s отрезок между соответствующими точками $P(0,t)$ и $P(1,t)$ этих отрезков:

$$\begin{aligned} B(s,t) &= s(tP(1,1)+(1-t)P(1,0))+ \\ &+ (1-s)(tP(0,1)+(1-t)P(0,0))= \\ &= stP(1,1)+s(1-t)P(1,0)+ \\ &+ t(1-s)P(0,1)+(1-s)(1-t)P(0,0). \end{aligned}$$

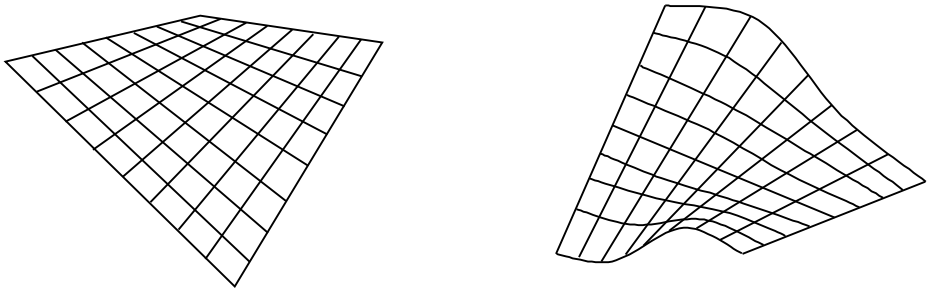
Введем вектор $L(u)=(1-u,u)$. Тогда выражение для билинейной поверхности $B(s,t)$ перепишется в виде

$$B(s,t)=L(s)\begin{pmatrix} P(0,0) & P(0,1) \\ P(1,0) & P(1,1) \end{pmatrix} L(t)^T.$$

Пример билинейной поверхности представлен на рис. 12.7, *а*.

12.2.2. Линейчатые поверхности

Линейчатые поверхности задаются на сегменте, две противоположные стороны которого являются кривыми, а две другие – прямолинейными отрезками (рис. 12.7, *б*). Как и в случае билинейных поверхностей,



а *б*

Рис. 12.7. Билинейная (*а*) и линейчатая (*б*) поверхности

проведем линейную интерполяцию по s между граничными кривыми $P(0,t)$ и $P(1,t)$:

$$P(s,t)=sP(1,t)+(1-s)P(0,t)=L(s)[P(0,t),P(1,t)].$$

Легко проверить, что поверхность удовлетворяет граничным условиям, то есть проходит через задающие ее кривые $P(0,t)$ и $P(1,t)$ и прямолинейные отрезки $P(s,0)$ и $P(s,1)$.

12.2.3. Линейные поверхности Кунса

Пусть теперь сегмент поверхности со всех четырех сторон ограничен кривыми $P(s,0)$, $P(s,1)$, $P(0,t)$ и $P(1,t)$. Если сложить линейчатые поверхности, ограниченные кривыми $P(s,0)$ и $P(s,1)$ и кривыми $P(s,0)$ и $P(s,1)$, то получим

$$Q(s,t) = tP(s,1) + (1-t)P(s,0) + sP(1,t) + (1-s)P(0,t).$$

Но на ребрах не выполняются граничные условия:

$$Q(0,t) = [tP(0,1) + (1-t)P(0,0)] + P(0,t).$$

Выражение в квадратных скобках равно разности между границей куска $Q(0,t)$ и кривой $P(0,t)$ и представляет собой отрезок с концами в угловых точках $P(0,0)$ и $P(0,1)$. Стало быть, для выполнения граничных условий вдоль всего контура необходимо вычесть из $Q(s,t)$ поверхность, ограниченную прямолинейными отрезками с концами в точках $P(0,0)$, $P(0,1)$, $P(1,0)$ и $P(1,1)$. Простейшей такой поверхностью является билинейная поверхность $B(s,t)$. Получаем, что поверхность, интерполирующая граничные кривые, может быть представлена как результат сложения двух линейчатых поверхностей и вычитания билинейной (рис. 12.8):

$$\begin{aligned} C(s,t) &= Q(s,t) - B(s,t) \\ &= L(s)[P(0,t), P(1,t)] + L(t)[P(s,0), P(s,1)] \\ &- L(s) \begin{pmatrix} P(0,0) & P(0,1) \\ P(1,0) & P(1,1) \end{pmatrix} L(t)^T. \end{aligned}$$

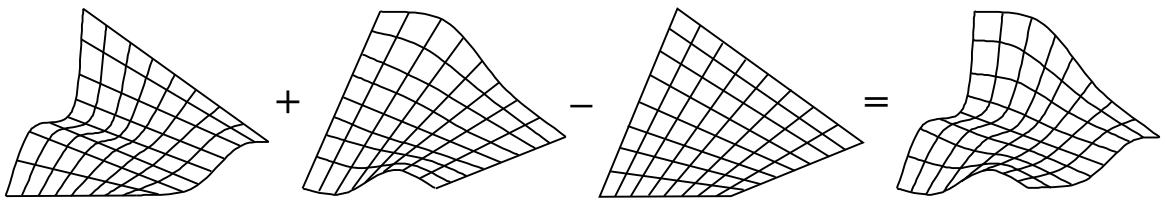


Рис. 12.8. Линейная поверхность Кунса есть результат сложения двух линейчатых поверхностей и вычитания билинейной

Нетрудно убедиться, что для линейной поверхности Кунса имеет место совпадение поверхности с граничными кривыми.

12.2.4. Кубические поверхности Кунса

Обобщим вектор-функцию $L(u)$ следующим образом:

$$F(u) = [1 - F_0(u), F_0(u)],$$

где функция $F_0(u)$ называется *интерполяционной функцией Кунса*. Обозначим $F_1(u) = 1 - F_0(u)$.

Пусть теперь для двух соседних кусков требуется обеспечить не только совпадение граничных кривых, но и совпадение наклона поверхности вдоль общей границы. В качестве интерполяционной функции Кунса рассмотрим функцию $F_0(u) = -2u^3 + 3u^2$, обладающую следующими свойствами (рис. 12.9):

$$F_0(0) = 0, \quad F_0(1) = 1, \quad \frac{dF_0}{du}(0) = \frac{dF_0}{du}(1) = 0.$$

Для функции F_1 имеем

$$F_1(0) = 1, \quad F_1(1) = 0, \quad \frac{dF_1}{du}(0) = \frac{dF_1}{du}(1) = 0.$$

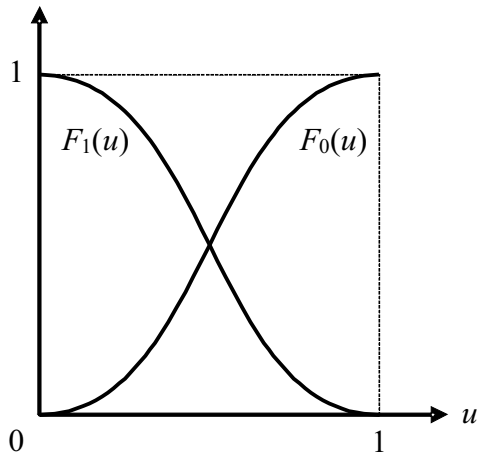


Рис. 12.9. Графики функций F_0 и F_1

Подставим $F(u)$ вместо $L(u)$ в выражение для линейного куска Кунса и рассмотрим поведение поверхности на границе сегмента.

Для $C(s,0)$ имеем

$$\begin{aligned} C(s,0) &= F_0(0)P(s,1) + F_1(0)P(s,0) + F_0(s)P(1,0) + F_1(s)P(0,0) - \\ &- F_0(s)F_0(0)P(1,1) - F_0(s)F_1(0)P(1,0) - \\ &- F_0(0)F_1(s)P(0,1) - F_1(s)F_1(0)P(0,0) = \\ &= P(s,0) + [F_0(s)P(1,0) + F_1(s)P(0,0)] - \\ &- [F_0(s)P(1,0) + F_1(s)P(0,0)] = \\ &= P(s,0). \end{aligned}$$

Таким образом, сегмент проходит через граничную кривую, причем поверхность, состоящая из двух кубических кусков Кунса с общей граничной кривой, непрерывна.

Найдем теперь условия, при которых имеет место непрерывность частных производных на границе двух соседних сегментов. Исследуем поведение соседних кусков $P_1(s,t)$ и $P_2(s,v)$ вдоль общей границы $P_1(s,0) = P_2(s,1)$.

Поскольку граница общая, то и производная по s совпадает вдоль нее для обоих кусков. Распишем производные по t и v:

$$\begin{aligned} \frac{\partial P_1}{\partial t}(s,t) &= \frac{\partial P_1}{\partial t}(s,0)\frac{dF_0}{dt}(t) + \frac{\partial P_1}{\partial t}(s,1)\frac{dF_1}{dt}(t) + \\ &+ \frac{\partial P_1}{\partial t}(0,t)F_0(s) + \frac{\partial P_1}{\partial t}(1,t)F_1(s) - \\ &- \frac{\partial P_1}{\partial t}(0,0)F_0(s)\frac{dF_0}{dt}(t) - \frac{\partial P_1}{\partial t}(1,0)F_1(s)\frac{dF_0}{dt}(t) - \\ &- \frac{\partial P_1}{\partial t}(0,1)F_0(s)\frac{dF_1}{dt}(t) - \frac{\partial P_1}{\partial t}(1,1)F_1(s)\frac{dF_1}{dt}(t). \end{aligned}$$

Поскольку в точке $P_1(s,0)$ кривой производные $\dfrac{dF_0}{dt}$ и $\dfrac{dF_1}{dt}$ равны нулю, имеем

$$\frac{\partial P_1}{\partial t}(s,0) = \frac{\partial P_1}{\partial t}(0,0)F_0(s) + \frac{\partial P_1}{\partial t}(1,0)F_1(s)$$

и, аналогично, для куска P_2

$$\frac{\partial P_2}{\partial v}(s,1) = \frac{\partial P_2}{\partial v}(0,1)F_0(s) + \frac{\partial P_2}{\partial v}(1,1)F_1(s),$$

откуда получаем достаточное условие гладкости кубической поверхности Кунса на границе соседних кусков:

$$\frac{\partial P_1}{\partial t}(0,0) = \frac{\partial P_2}{\partial v}(0,1),$$
$$\frac{\partial P_1}{\partial t}(1,0) = \frac{\partial P_2}{\partial v}(1,1).$$

12.2.5. Общее представление бикубических поверхностей

Бикубические поверхности имеют следующее общее представление:

$$\begin{aligned} P(s,t) &= a_{11}s^3t^3 + a_{12}s^3t^2 + a_{13}s^3t + a_{14}s^3 + \\ &+ a_{21}s^2t^3 + a_{22}s^2t^2 + a_{23}s^2t + a_{24}s^2 + \\ &+ a_{31}st^3 + a_{32}st^2 + a_{33}st + a_{34}s + \\ &+ a_{41}t^3 + a_{42}t^2 + a_{43}t + a_{44} \end{aligned}$$

или в матричной форме

$$P(s,t) = SCT^T,$$

где

$$s = (s^3\ s^2\ s\ 1), \quad t = (t^3\ t^2\ t\ 1), \quad C = \begin{pmatrix} a_{11} & \dots & a_{14} \\ \vdots & \ddots & \vdots \\ a_{41} & \dots & a_{44} \end{pmatrix},$$

причем коэффициенты матрицы C представляют собой трехмерные вектора вида $a_{ij} = (a_{ij}^x, a_{ij}^y, a_{ij}^z)$.

12.2.6. Форма Эрмита для бикубической поверхности

Форма Эрмита для бикубической поверхности задается с помощью управляющих точек и касательных векторов в вершинах.

Рассмотрим уравнение кубической кривой (для удобства выпишем выражение только для x-координаты)

$$x(s) = SM_hG_{hx}.$$

Перепишем его так, чтобы геометрическая матрица G_{hx} являлась функцией от t

$$x(s,t) = SM_h \begin{pmatrix} P_1(t) \\ P_4(t) \\ R_1(t) \\ R_4(t) \end{pmatrix}_x .$$

Представим каждую из кривых $P_i(t)$ также в виде многочлена Эрмита:

$$P_{1x}(t) = TM_h \begin{pmatrix} q_{11} \\ q_{12} \\ q_{13} \\ q_{14} \end{pmatrix}_x , \quad P_{4x}(t) = TM_h \begin{pmatrix} q_{21} \\ q_{22} \\ q_{23} \\ q_{24} \end{pmatrix}_x ,$$

$$R_{1x}(t) = TM_h \begin{pmatrix} q_{31} \\ q_{32} \\ q_{33} \\ q_{34} \end{pmatrix}_x , \quad R_{4x}(t) = TM_h \begin{pmatrix} q_{41} \\ q_{42} \\ q_{43} \\ q_{44} \end{pmatrix}_x$$

или в матричной форме

$$(P_1(t)\ P_4(t)\ R_1(t)\ R_4(t)) = TM_h \begin{pmatrix} q_{11} & \cdots & q_{41} \\ \vdots & \ddots & \vdots \\ q_{14} & \cdots & q_{44} \end{pmatrix} .$$

Транспонируем левую и правую части:

$$\begin{pmatrix} P_1(t) \\ P_4(t) \\ R_1(t) \\ R_4(t) \end{pmatrix} = \begin{pmatrix} q_{11} & \cdots & q_{14} \\ \vdots & \ddots & \vdots \\ q_{41} & \cdots & q_{44} \end{pmatrix}_x M_h T^T = Q_x M_h T^T$$

и подставим в выражение для $x(s,t)$

$$x(s,t) = SM_h Q_x M_h^T T^T .$$

Объединив полученное выражение с аналогичными выражениями для координат y и z, получим представление для бикубической поверхности в форме Эрмита

$$P(s,t) = SM_h Q M_h^T T^T .$$

Анализ происхождения коэффициентов матрицы Q позволяет установить, что она имеет следующую структуру:

$$Q_x = \begin{pmatrix} x(0,0) & x(0,1) & \frac{\partial x}{\partial t}(0,0) & \frac{\partial x}{\partial t}(0,1) \\ x(1,0) & x(1,1) & \frac{\partial x}{\partial t}(1,0) & \frac{\partial x}{\partial t}(1,1) \\ \frac{\partial x}{\partial s}(0,0) & \frac{\partial x}{\partial s}(0,1) & \frac{\partial^2 x}{\partial s \partial t}(0,0) & \frac{\partial^2 x}{\partial s \partial t}(0,1) \\ \frac{\partial x}{\partial s}(1,0) & \frac{\partial x}{\partial s}(1,1) & \frac{\partial^2 x}{\partial s \partial t}(1,0) & \frac{\partial^2 x}{\partial s \partial t}(1,1) \end{pmatrix} \tag{12.3}$$

Правая нижняя подматрица матрицы Q содержит значения *кручения* поверхности в каждой из четырех вершин, которые часто полагают равными нулю. Это приводит к тому, что в местах сопряжения четырех соседних сегментов бикубическая поверхность выглядит плоской. Чтобы этого избежать, кручение аппроксимируется на основе направлений касательных в соседних вершинах.

Бикубическая поверхность в форме Эрмита, как и кубическая поверхность Кунса, позволяет достичь непрерывности поверхности и касательных к ним при переходе от одного куска к другому. Из выражения (12.3) следует, что для гладкой сшивки кусков необходимо и достаточно, чтобы граничные кривые на общем ребре были одинаковыми для обоих кусков, а касательные вектора и кручения совпадали в общих вершинах (рис. 12.10).

12.2.7. Форма Безье для бикубической поверхности

Представим точку $P(s,t)$ на бикубической поверхности в виде

$$P(s,t) = SM_b \begin{pmatrix} P_1(t) \\ P_2(t) \\ P_3(t) \\ P_4(t) \end{pmatrix},$$

где

$$P_i(t) = TM_b \begin{pmatrix} p_{i1}(t) \\ p_{i2}(t) \\ p_{i3}(t) \\ p_{i4}(t) \end{pmatrix}.$$

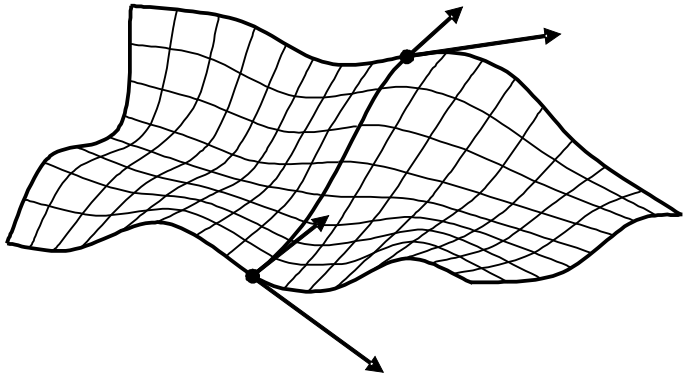


Рис. 12.10. Гладкое сопряжение бикубических кусков в форме Эрмита

В результате получаем выражение для формы Безье

$$P(s,t) = SM_b P M_b^T T^T.$$

Элементами геометрической матрицы являются 16 точек:

$$P = \begin{pmatrix} p_{11} & \cdots & p_{14} \\ \vdots & \ddots & \vdots \\ p_{41} & \cdots & p_{44} \end{pmatrix},$$

где управляющие точки p_{11}, p_{41}, p_{14} и p_{44} совпадают с угловыми точками сегмента $P(0,0)$, $P(1,0)$, $P(0,1)$ и $P(1,1)$ (рис. 12.11).

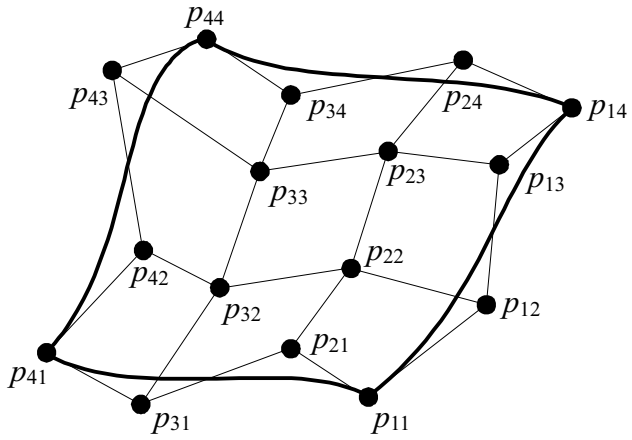


Рис. 12.11. Бикубическая поверхность и управляющие точки Безье

Выясним, при каком расположении управляющих точек двух соседних кусков Безье обеспечивается непрерывность и гладкость составной

поверхности. Как для формы Эрмита, C^0-непрерывность достигается, когда граничная кривая на общем ребре совпадает для обоих кусков. Это условие выполняется, если совпадают четыре управляющие точки, задающие общую границу.

Как показано на рис. 12.12, для достижения непрерывности касательного вектора в поперечном направлении относительно общего ребра (C^1-непрерывность) требуется, чтобы управляющие

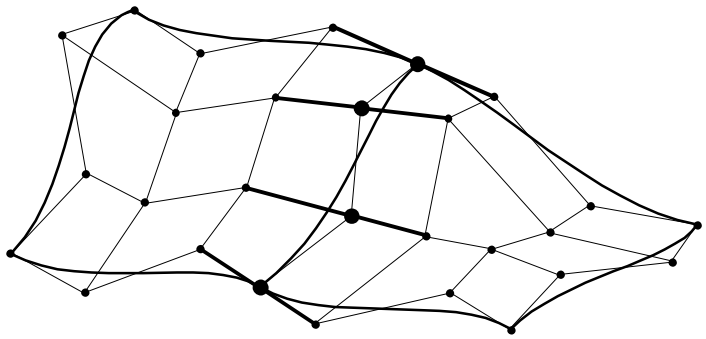


Рис. 12.12. Гладкая сшивка двух кусков Безье

точки, задающие наклон поверхности на границе куска, задавали коллинеарные отрезки по обеим сторонам от ребра.

Упражнения

2-1. Выпишите формулы для расчета первых производных в заданной точке поверхности с помощью координат вектора нормали в этой точке.

2-2. Для сетки четырехугольных доменов на плоскости с заданными значениями высоты в каждом узле разработайте алгоритм аппроксимации первых производных. Указание: используйте координаты векторов нормалей билинейных кусков смежных доменов.

2-3. Разработайте алгоритм для вычисления приближенного значения вторых производных в узлах сетки на основе известных значений первых производных.

Библиографические ссылки

1. *Avnaim F., Boissonnat J. -D., Devillers O.* et al. Evaluating signs of determinants using single-precision arithmetic // ALGORITHMICA. Vol. 17. № 2. P. 111–132.

2. *D'Azevado E. F.* Are bilinear quadrilaterals better than linear triangles? // SIAM Journal on Scientific Computing. 2000. Vol. 22. № 1. P. 198–217.

3. *De Berg M., van Kreveld M., Overmars M.* et al. Computational Geometry (Algorithms and Applications). Springer-Verlag, 1997.

4. *Bern M., Eppstein D.* Mesh generation and optimal triangulation // Computing in Euclidean Geometry, D.–Z. Du and F.K. Hwang, eds., World Scientific. 1992. P. 23–90.

5. *Bose P., Toussaint G. T.* Characterizing and efficiently computing quadrangulations of planar point sets // Computer Aided Geometric Design. 1997. Vol. 14. P. 763–785.

6. *Edelsbrunner H.* Algorithms in Combinatorial Geometry. Springer-Verlag, 1987.

7. *Farin G.* Curves and surfaces in CAGD. Academic Press, 1997.

8. *Fortune S., Van Wyk Ch. J.* Efficient Exact Arithmetic for Computational Geometry. Ninth Annual Symposium on Computational Geometry. 1993. P. 163–172.

9. *Fortune S.* A sweepline algorithm for Voronoi diagrams // Proceedings of the second annual symposium on Computational geometry. Yorktown Heights, New York, United States. 1986. P. 313–322.

10. *Guibas L., Stolfi J.* Primitives for the manipulation of general subdivisions and the computation of Voronoi diagrams // ACM Transactions on Graphics. 1985. Vol. 4. № 2. P. 74–123.

11. *Hu T. C., Shing M.T.* Computation of matrix chain product // SIAM Jounal on computing. Part I. 1982. 11(2). P. 362–373; Part II. 1984. 13(2). P. 228–251.

12. *Mielke B.* Integrated computer graphics. West Publishing company, 1991.

13. *Manacher G., Zobrist A.* Neither the Greedy Nor the Delaunay Triangulation of a Planar Point Set Approximates the Optimal Triangulation. Inf. Process. Lett. 1979. 9(1). P. 31–34.

14. *Ramaswami S., Ramos P., Toussaint G. T.* Converting triangulations to quadrangulations // Computational Geometry: Theory and Applications. 1998. Vol. 9. P. 257–276.

15. *Shewchuk J. R.* Adaptive precision floating-point arithmetic and fast robust geometric predicates. Preprint, School of computer science Carnegie Mellon University, Pittsburg PA 15231, 1996.

16. *Shewchuk J. R.* What Is a Good Linear Finite Element? Interpolation, Conditioning, Anisotropy, and Quality Measures. Preprint, Department of Electrical Engineering and Computer Sciences University of California at Berkeley, CA 94720, 2002.

17. *Sibson R.* Locally equiangular triangulations // Comput. J. 1977. Vol. 21. P. 243–245.

18. *Toussaint G. T.*, Quadrangulations of planar sets // Proceedings of 4th International Workshop on Algorithms and Data Structures (WADS'95), invited paper. 1995. P. 218–227.

19. *Vasilkov D. M.* Surface design based upon a combined mesh // IKM 2003. Bauhaus–Universität Weimar. 2003.

20. *Ахо А., Хопкрофт Дж., Ульман Дж.* Построение и анализ вычислительных алгоритмов. М.: Мир, 1979.

21. *Ахо А., Хопкрофт Дж., Ульман Дж.* Структуры данных и алгоритмы. М.: Вильямс, 2000.

22. *Вирт Н.* Алгоритмы и структуры данных. М.: Мир, 1989.

23. *Гэри М., Джонсон Д.* Вычислительные машины и труднорешаемые задачи. М.: Мир, 1982.

24. *Джосьютис Н.* C++ Стандартная библиотека. СПб.: Питер, 2004.

25. *Кормен Т., Лейзерсон Ч., Ривест Р.* Алгоритмы: построение и анализ. М.: МЦНМО, 1999.

26. *Котов В. М., Соболевская Е. П.* Разработка и анализ алгоритмов. Минск: БГУ, 2009.

27. *Ласло М.* Вычислительная геометрия и компьютерная графика на C++. М.: Бином, 1997.

28. *Препарата Ф., Шеймос М.* Вычислительная геометрия: Введение. М.: Мир, 1989.

29. *Роджерс Д., Адамс Дж.* Математические основы машинной графики. М.: Мир, 2001.

30. *Скворцов А. В.* Триангуляция Делоне и ее применение. Томск: Изд-во Томского университета, 1992.

31. *Страуструп Б.* Язык программирования C++. М.: Бином, 1999.

32. *Фоли Д., ван Дэм А.* Основы интерактивной машинной графики. М.: Мир, 1985.

33. *Хилл Ф.* OpenGL. Программирование компьютерной графики. Для профессионалов. СПб.: Питер, 2002.

34. *Шикин Е. В., Боресков А. В.* Компьютерная графика. М.: Диалог МИФИ, 1995.

Предметный указатель

Учебное издание

Васильков Дмитрий Михайлович

ГЕОМЕТРИЧЕСКОЕ МОДЕЛИРОВАНИЕ И КОМПЬЮТЕРНАЯ ГРАФИКА

ВЫЧИСЛИТЕЛЬНЫЕ И АЛГОРИТМИЧЕСКИЕ ОСНОВЫ

Курс лекций

В авторской редакции

Художник обложки *Т. Ю. Таран*
Технический редактор *Г. М. Романчук*
Корректор *Л. С. Мануленко*

Ответственный за выпуск *Т. М. Турчиняк*

Электронный ресурс 2,2 Мб
Режим доступа: http:/www.elib.bsu.by, ограниченный

Белорусский государственный университет.
ЛИ № 02330/0494425 от 08.04.2009.
Пр. Независимости,4, 220030, Минск.