ВЫЧИСЛИТЕЛЬНАЯ ГЕОМЕТРИЯ В С

ВТОРОЕ ИЗДАНИЕ

ДЖОЗЕФ О'РУРК

САМБРИДЖ Е
UNIVERSITYPAESS

Содержание

Предисловие

Вычислительная геометрия в широком смысле - это изучение алгоритмов решения геометрических задач на компьютере. В данном тексте акцент сделан на проектировании таких алгоритмов, несколько меньше внимания уделено анализу производительности. В ряде случаев проектирование было доведено до уровня рабочих программ на языке Си, которые подробно обсуждаются.

Существует множество видов геометрии, и то, что стало известно как "вычислительная геометрия", рассматриваемая в этой книге, - это в первую очередь дискретная и комбинаторная геометрия. Таким образом, многоугольники играют в этой книге гораздо большую роль, чем области с кривыми границами. Большая часть работы по непрерывным кривым и поверхностям относится к "геометрическому моделированию" или "твердотельному моделированию" - области, имеющей свои собственные конференции и тексты, отличные от вычислительной геометрии. Конечно, существует значительное дублирование, и нет никаких фундаментальных причин для такого разделения областей; более того, кажется, что они в некоторой степени сливаются.

На момент написания этой статьи области вычислительной геометрии всего двадцать лет, если за ее начало диссертацию М. И. Шамоса (Shamos 1978). Сейчас существуют ежегодные конференции, журналы, тексты и процветающее сообщество исследователей с общими интересами.

Охваченные темы

1 К "основным" проблемам вычислительной геометрии относятся разбиение многоугольников (включая триангуляцию), выпуклые корпуса, диаграммы Вороного, расположение линий, геометрический поиск и планирование движения. Этим темам посвящены главы данной книги. Область не настолько устоялась, чтобы этот список можно было считать консенсусом; другие исследователи определяют ядро по-другому.

Многие учебники содержат гораздо больше материала, чем можно охватить за один семестр. Это не такой учебник. Обычно за один семестр в 40 аудиторных часов студенты проходят около 80 % текста, а аспиранты - весь текст. Чтобы затронуть каждую из основных тем, я считаю необходимым колебаться в уровне детализации, описывая только одни алгоритмы и подробно описывая другие. Какие алгоритмы описываются в общих чертах, а какие подробно - это мой личный выбор, который я могу обосновать только своим опытом работы в аудитории.

Пререквизиты

Материал, представленный в этом тексте, должен быть доступен студентам с минимальной подготовкой.

Для математики достаточно дискретной математики, исчисления и линейной алгебры. На самом деле очень

' Например, Хоффманн (1989) и Мортенсон (1990).

В тексте не используется практически никаких вычислений или линейной алгебры, и предприимчивый студент может освоить **все необходимое** на лету. В области информатики достаточно курса программирования и знакомства со структурами данных (Computer Science I и II во многих школах). Я не предполагаю наличия курса по алгоритмам, только знакомство с нотацией "big-0". Я преподаю этот материал студентам младших и старших курсов колледжа, в основном специализирующимся на информатике и математике.

Спешу добавить, что книга может быть плодотворно изучена теми, кто вообще не имеет опыта программирования, просто пропустив все разделы, посвященные реализации. Те, кто знает какой-либо язык программирования, но не C, смогут легко оценить обсуждение реализации, даже если не смогут прочитать код. Весь код доступен как на Java, так и на C, хотя в основном тексте рассматривается только C.

Когда я преподаю этот материал студентам, изучающим информатику и математику, я предлагаю им на выбор проекты, которые позволяют тем, кто имеет навыки программирования, писать код, а тем, кто имеет теоретические склонности, избегать программирования.

Хотя книга написана для студентов старших курсов, по моему опыту, она может стать основой для сложного курса для аспирантов. Я старался сочетать элементарные объяснения со ссылками на новейшую литературу. В сносках приводятся технические подробности и цитаты. В ряде упражнений ставятся открытые проблемы. Нетрудно дополнить текст научными статьями, взятыми из 300 библиографических ссылок, что позволит эффективно довести материал до уровня аспирантуры.

Реализации

Не все алгоритмы, рассматриваемые в книге, снабжены реализациями. В книгу включен полный код для двенадцати алгоритмов:[2]

- Площадь многоугольника.
- Триангулирование многоугольника.
- Выпуклый корпус в двух измерениях.
- Выпуклый корпус в трех измерениях.
- Триангуляция Делоне.
- Пересечение сегментов/лучей-сегментов.
- Пересечение сегмента и лучевого треугольника.
- Точка в многоугольнике.
- Точка в многограннике.
- Пересекающиеся выпуклые многоугольники.
- Свертка Минковского с выпуклым многоугольником.
- Многозвенная робота.

Исследователи в промышленности, пришедшие в эту книгу за рабочим кодом для своих любимых алгоритмов, могут быть разочарованы: Они могут искать алгоритм нахождения минимального охватывающего круга для набора точек и найти его в качестве упражнения.[3] Представленный код следует рассматривать как примеры программ по геометрии. Я надеюсь, что выбрал репрезентативный набор алгоритмов для реализации; остается много места для студенческих проектов.

Распределение также включает в себя генерацию случайных точек в кубе (рис. 4.14), случайных точек на сфере (рис. 4.15) и унитарно распределенных точек на сфере (изображение на обложке книги). "Упражнение 5.5.6[1 2].

Весь код на языке Си в книге доступен по анонимному ftp с адреса cs . smi th . edu (131.229.222.23), в каталоге /pub/compgeom.[4] Я регулярно обновляю файлы в этом каталоге, исправляя ошибки и внося улучшения. Java-версии всех программ находятся в том же каталоге.

Упражнения

В текст включено около 250 упражнений. Они варьируются от простых расширений текста до довольно сложных задач и "открытых" проблем. Последние являются захватывающей особенностью такой новой области; студенты могут достичь границы знаний так быстро, что даже студенты старших курсов могут надеяться решить проблемы, которые еще никому не удалось разгадать. Действительно, я написал несколько работ с участием студентов в результате их работы над поставленными мною домашними заданиями.[5] Не все открытые проблемы обязательно трудны; некоторые просто ожидают должного внимания.

Упражнения изредка помечены "[легко]" или "[сложно]", но отсутствие этих пометок не должно означать, что ни то, ни другое не применимо. Упражнения с пометкой "[программирование]" требуют навыков программирования, а упражнения с пометкой "[открытые]" являются нерешенными проблемами, насколько мне известно на данный момент. Я постарался указать авторов отдельных задач, где это уместно. Преподаватели могут связаться со мной для получения частичного руководства по решению.

Улучшения второго издания

По закону природы вторые издания длиннее первых, и эта книга не стала исключением: Она примерно на пятьдесят страниц длиннее, содержит пятьдесят новых упражнений, тридцать новых рисунков и восемьдесят дополнительных библиографических ссылок. Весь код первого издания значительно улучшен: Все программы теперь выводятся в формате Postscript, все переведены на язык Java, многие из них стали проще и/или логически чище, большинство более устойчивы к вырождениям и численным ошибкам, а большинство работает (иногда) значительно быстрее. Код триангуляции многоугольников и код триангуляции Делоне теперь имеют *размерность* $0(n^2)$.

Включены четыре новые программы: для вычисления триангуляции Делоне из трехмерного выпуклого корпуса (раздел 5.7.4), для пересечения луча с треугольником в 3-пространстве (раздел 7.3), для определения, находится ли точка внутри многогранника (раздел 7.5), для вычисления свертки (суммы Минковского) выпуклого многоугольника с общим многоугольником (раздел 8.4.4), а также код генерации точек, с помощью которого было получено изображение обложки.

Включены новые разделы, посвященные разбиению на монотонные горы (раздел 2.3), рандомизированной триангуляции (раздел 2.4. l), конечному(?) алгоритму плоского выпуклого корпуса (раздел 3.8.4), рандомизированному выпуклому корпусу в трех измерениях (раздел 4.5), структуре данных twin edge (раздел 4.4), пересечению отрезка и треугольника (раздел 7.3), проблема "точка в многограннике" (раздел 7.5), алгоритм Бентли-Отмана для пересечения отрезков (раздел 7.7), вычисление булевых операций между двумя многоугольниками (раздел 7.8), рандомизированное трапециевидное разложение для нахождения точки (раздел 7.11.4), вычисление свертки Минковского (раздел 8.4.4) и список источников для дальнейшего чтения (глава 9).

Подключайтесь к f tp c s . smi th . edu и используйте имя anon ous. Или получите доступ к файлам через ht tp : / / c s . smi th . edu / orourke .

Материал из одной статьи включен в раздел 7.6.

Значительно улучшены другие разделы, в том числе посвященные QuickHull (раздел 3.4), алгоритму Грэхема (раздел 3.5.5), переполнению объема (раздел 4.3.5), построению трианов Делоне с помощью преобразования параболоидов (раздел 5.7.4), задаче "точка в полигоне" (раздел 7.4), пересечению двух отрезков (раздел 7.2) и реализации пересечения выпуклых полигонов (раздел 7.6.1).

Благодарности

Я получил более шестисот сообщений по электронной почте от читателей первого издания этой книги, и я отчаялся точно распределить их заслуги по многим конкретным вкладам в это издание. Я глубоко ценю предложения следующих людей, многие из которых являются моими профессиональными коллегами, двадцать девять - моими бывшими студентами, но с большинством из которых я знаком только в электронном виде: Панкадж Агарвал, Кристи Андерсон, Билл Болдуин, Майкл Болдуин, Пьер Бошемин, Эд Болсон, Хелен Камерон, Джоан Кэннон, Рой Чиен, Сатьян Кург, Гленн Дэвис, Адлай ДеПано, Мэтью Диас, Тамала Дирлам, Дэвид Добкин, Сьюзен Дорвард, Скот Драйсдейл, Герберт Эдельсбруннер, Джон Эллис, Уильям Флис, Стив Форчун, Роберт Фрачкевич, Рейнальдо Гарсия, Шармилли Гош, Кароль Гитлин, Джейкоб Э. Goodman, Michael Goodrich, Horst Greiner, Suleyman Guleyupoglu, Eric Haines, Daniel Halperin, Eszter Hargittai, Paul Heckbert, Claudio Heckler, Paul Heffernan, Kevin Hemsteter, Christoph Hoffmann, Rob Hoffmann, Chun-Hsiung Huang, Кнут Хунстад, Ферран Уртадо, Джоан Хатчинсон, Андрей Тонс, Крис Джонстон, Мартин Джонс, Эми Йозефчик, Мартин Кершер, Эд Кнорр, Ник Корнеенко, Джон Катчер, Юджин Ли, Дэвид Лубинский, Джо Малкевич, Мишель Маурер, Майкл МакКенна, Томас Майер, Уолтер Мейер, Саймон Майкл, Джессика Миллер, Энди Мирзаян, Джозеф Митчелл, Аделин Нг, Сонбин Парк, Ирена Пащенко, Октавия Петровичи, Мадхав Понамги, Ари Раппопорт, Дженнифер Риппел, Кристофер Сондерс, Кэтрин Шевон, Питер Шорн, Вадим Шапиро, Томас Шермер, Пол Шорт, Сол Симхон, Стив Скиена, Кеннет Слоан, Стивен Смолдерс, Эван Смит, Шэрон Солмс, Тед Стерн, Илеана Стрейну, Винита Субраманиан, Дж.W.H. Tangelder, Yi Tao, Seth Teller, Godfried Toussaint, Christopher Van Wyk, Gert Vetger, Jim Ward, Susan Weller, Wendy Welsh, Rephael Wenger, Gerry Wiener, Bob Williamson, Stacia Wyman, Min Xu, Dianna Xu, Chee Yap, Amy Yee, Wei Yinong, Lilla Zollei, и участники семинара Faculty Advancement in Mathematics 1992. Приношу извинения за неизбежные пропуски. Лорен Каулз из Кембриджа была идеальным редактором. Я получил щедрую поддержку от Национального научного фонда на мои исследования в области вычислительной геометрии,

Последний раз по гранту CCR-9421 h70.

Джозеф О'Рурк
orourkeW cs.smith.edu
http://cs.smith.edns orourke
*Колледж **Смита**, Массачусетс*
23 декабря 1997 г.

Примечание на обложке:

На обложке изображен выпуклый корпус из 5000 точек, распределенных по спиральной кривой на поверхности сферы. Оно было сгенерировано в результате выполнения кодов spi ral . c и chu 11 . c, распространяемых вместе с этой книгой: spi ra1 5000 -r 10 00| chu 11 .

1

Триангуляция полигонов

1.1. ТЕОРЕМЫ ХУДОЖЕСТВЕННОЙ ГАЛЕРЕИ

1.1.1. Polygons

Большая часть вычислительной геометрии выполняет свои вычисления на геометрических объектах, известных как многоугольники. Многоугольники - это удобное представление для многих объектов реального мира; удобное как в том, что абстрактный многоугольник часто является точной моделью реальных объектов, так и в том, что им легко манипулировать вычислительно. Примеры их использования включают в себя представление формы отдельных букв для автоматического распознавания символов, препятствия, которое необходимо обойти в среде робота, или части твердого объекта для отображения на графическом экране. Однако многоугольники могут быть довольно сложными объектами, и часто возникает необходимость рассматривать их как состоящие из более простых частей. Это приводит к теме этой и следующей главы: разбиение многоугольников на части.

Определение многоугольника

Многоугольник - это область плоскости, ограниченная конечным набором отрезков прямых, образующих простую замкнутую кривую. Определить точное значение фразы "простая замкнутая кривая", к сожалению, довольно сложно. Тополог сказал бы, что это гомеоморфный образ круга,[2] что означает, что это определенная деформация круга. Мы пока обойдемся без топологии и подойдем к определению более простым способом, следующим образом.

Пусть *0. - , *2' - - - - t Un-j - *n* точек на плоскости. Здесь и во всей книге вся индексная арифметика будет mod n, подразумевая циклическое упорядочение точек, причем o следует за r , так как *(n* - 1) - l- 1 it= 0 (mod n). Пусть eo= *0* , *e*= ui . , *ei* - *,*i+1*' - -.
e,- = .- - n отрезков, соединяющих точки. Тогда эти отрезки ограничивают многоугольник iff[3]

1. Пересечением каждой пары отрезков, смежных в циклическом упорядочении, является единственная общая точка: *e,* O ei+= Ui+ t, для всех i= 0, ... , it - 1.
2. Несмежные отрезки не пересекаются: *e,* O e,= B, для всех j *i*+ l.

[1] *Отрезок ab* - это замкнутое подмножество прямой, заключенное между двумя точками a и *b,* которые называются его *конечными точками.* Подмножество замкнуто в том смысле, что оно включает в себя конечные точки. (Многие авторы используют *ab* для обозначения этого отрезка).

2A *окружность* - это одномерное точек. Мы термин *"диск"* для обозначения двумерного множества точек. область, ограниченная окружностью.

'"Iff" означает "если и только если" - удобное сокращение, популяризированное Халмосом (1985, с. 403).

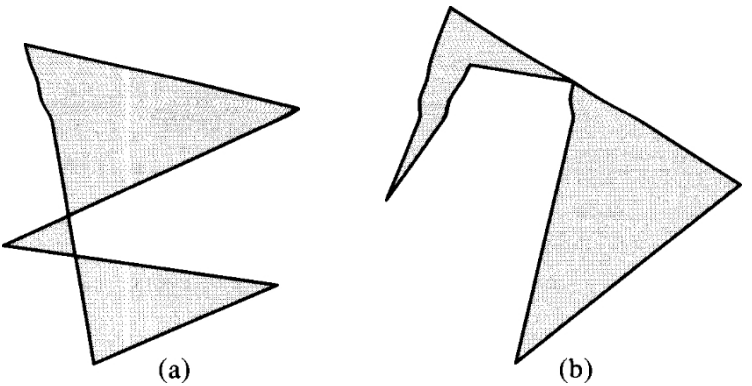


РИСУНОК 1.1 Непростые многоугольники.

Причина, по которой эти отрезки определяют *кривую*, заключается в том, что они соединены конец в конец; причина, по которой кривая является *замкнутой*, заключается в том, что они образуют цикл; причина, по которой замкнутая кривая является *простой*, заключается в том, что не смежные отрезки не пересекаются.

Точки Ui называются *вершинами* многоугольника, а отрезки ei - его вершинами. *граней.* Заметим, что многоугольник с n вершинами имеет n ребер.

Важной теоремой топологии является теорема Жордана о кривых:

Теорема 1.1.1 (теорема Жордана о кривой). *Каждая простая замкнутая кривая делит плоскость* на две *составляющие.*

Это кажется очевидным и не требующим доказательства, но на самом деле точное доказательство довольно сложно.[4] Мы примем это как данность. Две части плоскости называются *внутренней* и *внешней* сторонами кривой. Внешняя часть не ограничена, а внутренняя ограничена. Это обосновывает наше определение многоугольника как области, ограниченной совокупностью отрезков. Заметим, что мы определяем многоугольник *P* как замкнутую область плоскости. Часто под многоугольником понимают только сегменты, ограничивающие область, а не саму область. Мы будем использовать обозначение fi *P* для обозначения границы *P* - это обозначение заимствовано из топологии.[5] По нашему определению, *b P* С *P.*

На рисунке 1.1 показаны два не простых многоугольника. Для обоих объектов на рисунке отрезки удовлетворяют условию (1) выше (смежные отрезки имеют общую точку), но не условию (2): несмежные отрезки пересекаются. Такие объекты часто называют многоугольниками, а многоугольники, удовлетворяющие условию (2), - *простыми* . Так как в этой книге мы будем мало использовать не простые многоугольники, мы откажемся от лишнего модификатора.

Мы будем перечислять вершины многоугольника против часовой стрелки, чтобы при движении вдоль границы, посещая вершины в таком порядке (*обход границы),* внутренняя часть многоугольника всегда находилась слева от вас.

'См., например, Henle (1979, pp. 100-3). Теорема датируется 1877 годом.
некотором граница области похожа на производную, поэтому имеет смысл использовать символ частичной производной fi.

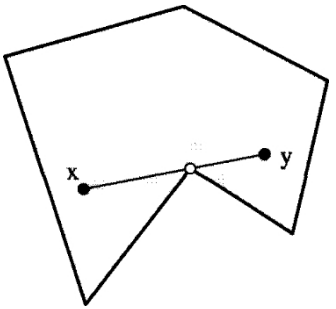


РИСУНОК 1.2 Падающий контакт прямой видимости.

1.1.2. Теорема о художественной галерее

Определение проблемы

Мы изучим увлекательную проблему, поставленную Клее[6], которая естественным образом приведет нас вопросу о триангуляции, наиболее важному разбиению многоугольников. Представьте себе помещение художественной галереи, план которого смоделировать многоугольником *из n* вершин. Клее спрашивает: сколько стационарных охранников необходимо для охраны помещения? Каждый охранник считается неподвижной точкой, которая может видеть во всех направлениях, то есть имеет дальность видимости 2s.[7] Конечно, охранник не может видеть сквозь стену комнаты. Эквивалентной формулировкой является вопрос о том, сколько точечных светильников необходимо для полного освещения комнаты. Прежде чем попытаться ответить на этот вопрос, уточним проблему Клее.

Видимость

Чтобы уточнить понятие видимости, скажем, что точка z может *видеть* точку y (или y видна x), если замкнутый отрезок xy нигде не является внешним по отношению к многоугольнику *P: x y* C *P.* Обратите внимание, что это определение допускает падающий контакт линии зрения с вершиной, как показано на рисунке 1.2. Альтернативное, не менее разумное определение гласит, что вершина может блокировать видимость; скажем, что *x* имеет *ясную видимость* для y, если xy C *P* и xy C fi *P* C (x, y). Мы будем иногда использовать это альтернативное определение в упражнениях (Упражнения 1.l.4[2] и [3]). Охранник - это точка. Считается, что множество стражей *покрывает* многоугольник, если каждая точка в многоугольнике видна какому-то стражу. Сами охранники не блокируют видимость друг друга.

Обратите внимание, что мы можем потребовать, чтобы охранники видели только точки fi *P,* ведь, предположительно, там находятся картины! Это интересный вариант, рассмотренный в упражнении 1.1.4[l].

Максимум над минимумом Формула

Теперь мы точно сформулировали большую часть задачи Клее, за исключением фразы "". Если говорить кратко, то задача состоит в том, чтобы найти максимальное по всем многоугольникам с *n* вершинами минимальное количество охранников, необходимых для покрытия многоугольника. Эта формулировка max-over-min непонятна новичкам, но она довольно часто используется в математике, поэтому уделим время ее тщательному объяснению.

Поставлена в 1973 году, о чем сообщает Honsberger (1976). Материалы этого раздела (и другие материалы по теме) можно найти в O'Rourke (1987).

[7]Для обозначения углов мы будем использовать радианы. u радианов = 180°.

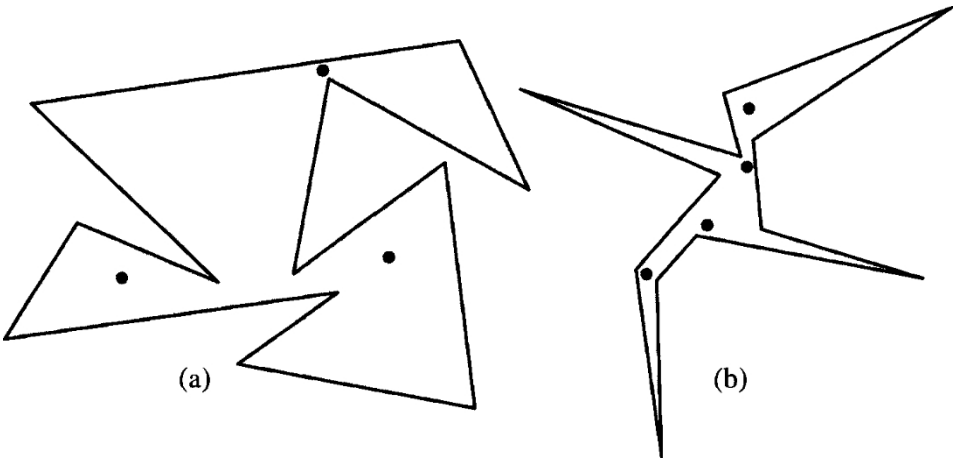


РИСУНОК 1.3 Два многоугольника с n= 12 вершинами: (a) требует 3 защит; (b) требует 4 защит.

Для любого заданного фиксированного многоугольника существует некоторое **минимальное** количество стражей, *необходимое* для полного покрытия. Так, на рисунке 1.3(a) видно, что для покрытия этого многоугольника с двенадцатью вершинами необходимо три стража, хотя расположение этих трех стражей может быть достаточно свободным. Но разве три стража - это самое большее, что требуется для всех возможных многоугольников с двенадцатью вершинами? Нет: многоугольник на рисунке 1.3(b), также с двенадцатью вершинами, требует четырех охранников. Каково наибольшее число охранников, которое требуется любому многоугольнику с двенадцатью вершинами? В конце концов мы покажем, что для любого многоугольника из двенадцати вершин всегда *достаточно* четырех стражей. это и является целью вопроса Клее: Выразите в виде функции от *n* наименьшее охранников, достаточное для покрытия любого многоугольника из *n* вершин. Иногда говорят, это число стражей необходимо и достаточно для покрытия: необходимо, потому что для *некоторых* многоугольников требуется по крайней мере такое количество стражей, а достаточно, потому что такого количества всегда достаточно для *любого* многоугольника.

Прежде чем исследовать проблему дальше, формализуем ее. Пусть *g(P)* - наименьшее число охранников, необходимое для покрытия многоугольника *P: g(P)*= mind *(S : S* покрывает *P]* |, где *S* - множество точек, а *S* - кардинальность' *S.* Пусть n - многоугольник из *n* вершин. *G(n)* - это максимум *g n)* Над всеми многоугольниками *с n* вершинами: *G{n)*= TTiax *p g(n-)* Klee's Проблема заключается в определении функции *G(n).* Может быть, не сразу очевидно, что *G(n)* определена для каждого n: по крайней мере, можно предположить, что для некоторого многоугольника не хватит конечного числа охранников. К счастью, *G(n)* конечна для всех *n,* как мы увидим. Но можно ли выразить ее в виде простой формулы или же она должна быть представлена в виде бесконечной таблицы значений, менее ясно.

Эмпирическое исследование

Достаточность. Конечно, всегда необходим хотя бы один страж. В наших обозначениях это дает нижнюю границу на *G{n):* l< *G(n).* Кажется очевидным, что для любого многоугольника достаточно *n* охранников: размещение охранника в каждой вершине обязательно покроет многоугольник. Это

Кардинальность множества - это количество его элементов.

обеспечивает верхнюю границу: $G_{\text{}}(n)$ у *n.* Но даже не совсем ясно, что достаточно *n* охранников. По крайней мере, это требует доказательства. Оно оказывается верным, оправдывая интуицию, но этот успех интуиции омрачается тем, что та же самая интуиция не работает в трех измерениях: Стражи, расставленные в каждой вершине многогранника, не обязательно покрывают многогранник! (См. упражнение 1.1.4[6]).

Существует множество задач, похожих на арт-галерею, и для большинства из них проще всего сначала нижнюю границу на *G(n)*, найдя общие примеры, показывающие, что иногда необходимо большое число охранников. Когда кажется, что никакая изобретательность не поможет увеличить необходимое число, пора переходить к доказательству того, что это число также достаточно. Именно так мы и поступим.

Необходимость для малых n. Для малых значений *n* можно угадать значение *G(n)* с небольшим исследованием. Очевидно, что каждый треугольник требует только одного охранника, поэтому *G(3)*= 1.

Четырехугольники можно разделить на две группы: выпуклые четырехугольники и четырехугольники с рефлекторной вершиной. Интуитивно понятно, что многоугольник является выпуклым, если у него нет вмятин. Это важное понятие будет подробно рассмотрено в главе 3. Вершина называется *рефлексивной*[9], если ее внутренний угол строго больше u; в противном случае вершина называется выпуклой.'[0] Выпуклый четырехугольник имеет четыре выпуклые вершины. Четырехугольник может иметь не более одной рефлексивной вершины по причинам, которые станут очевидны в разделе 1.2. Как видно из рисунка 1.4(a), даже четырехугольники с рефлексивной вершиной могут быть покрыты одним защитником, установленным вблизи этой вершины. Таким образом, G(4) = 1.

Для пятиугольников ситуация менее ясна. Конечно, выпуклый пятиугольник нуждается только в одной защите, а пятиугольник с одной рефлексивной вершиной - только в одной защите по той же причине, что и четырехугольник. Пятиугольник может иметь две рефлексивные вершины. Они могут быть либо смежными, либо разделенными выпуклой вершиной, как на рисунках l.4(c) и (d); в каждом случае достаточно одного защитника. Поэтому G(5) = 1.

Шестиугольники могут потребовать двух охранников, как показано на рисунке l.4(e) и (D. Небольшой эксперимент может привести к убеждению, что больше двух никогда не нужно, так что G(6) = 2.

Необходимость [n/3j

В этот момент читатель может перейти к обобщению рисунка l.4(f) для больших значений *n.* Рисунок 1.5 иллюстрирует конструкцию для *n* 12; обратите внимание на связь с рисунком l.4(9. Эта "гребенка" состоит из *k* зубцов, каждый из которых состоит из двух ребер, а соседние зубцы разделены ребром. Ассоциируя каждый зубчик с разделяющим ребром справа от него, а нижнее ребро - с крайним правым зубчиком, мы видим, что гребень из *k* зубчиков имеет *n* =- 3k ребер (и, следовательно, вершин). Поскольку для каждого зубца требуется своя защита, на этом примере мы устанавливаем, что *n/3< G(n)* для *n* -- 3k. Именно это я имел в виду, говоря, что общий пример может быть использован для установления нижней границы *G(n).*

Часто его называют *вогнутым,* но сходство слов "вогнутый" и "выпуклый" приводит к путанице, поэтому я буду использовать "рефлекс".

'0Некоторые авторы используют слово "выпуклый" для обозначения того, что я буду называть *строгой выпуклостью* - внутренний угол строго меньше чем н.

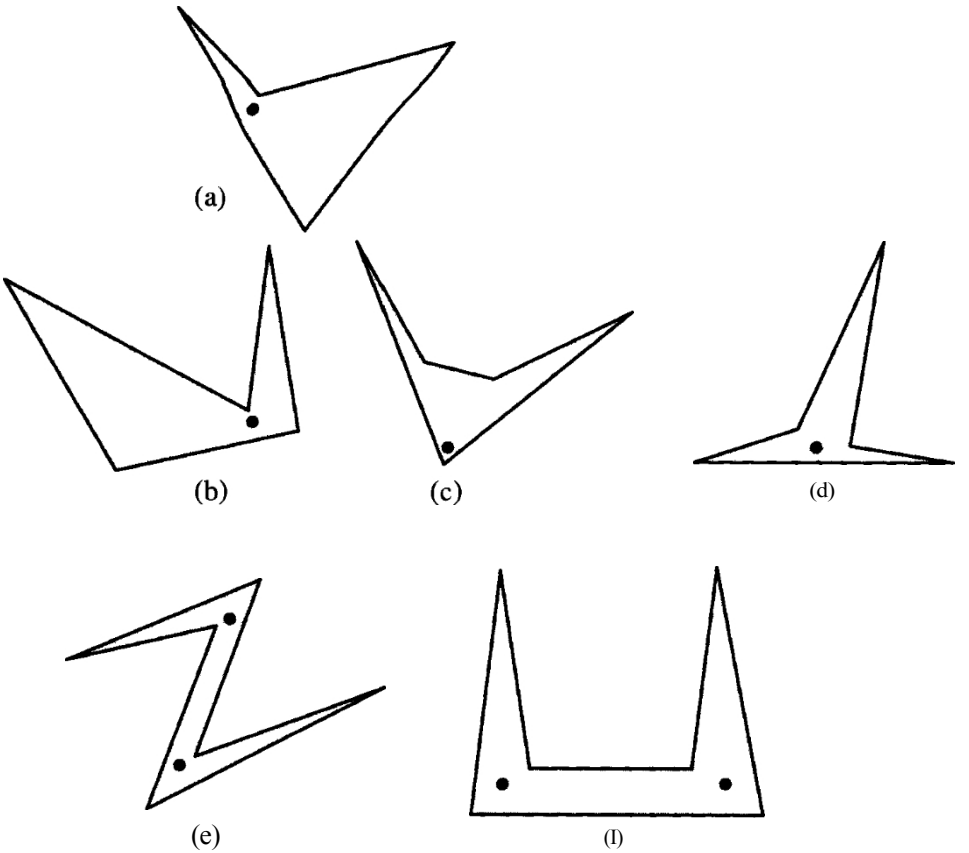


РИСУНОК 1.4 Многоугольники *с n*= 4, 5, 6 вершинами.

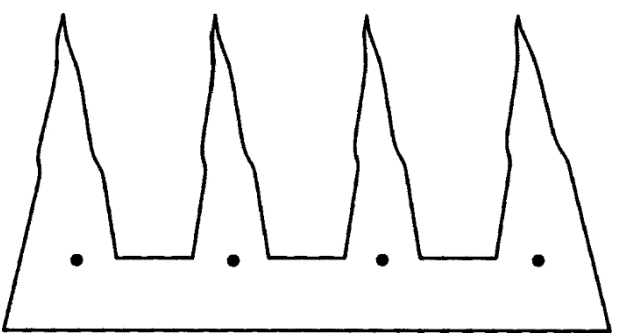


РИСУНОК 1.5 Гребень Шваталя для n= 12.

Заметив, что G(3) = *G(4)*= G(5), предположить, что *G(n)* - *{n/ 3) ,"* и на самом деле эта гипотеза оказывается верной. Это обычный способ ответа на подобные математические вопросы: Сначала ответ предполагается после эмпирического исследования,

' ' (zj - *floor* of z: наибольшее целое число меньшее или равное z. Функция floor отбрасывает дробную часть положительного вещественного числа.

и только после этого, имея перед собой определенную цель, доказать результат. Теперь перейдем к доказательству.

1.1.3. Доказательство достаточности Фиска

Первое доказательство того, что *G(n)*= n/3j, принадлежит Чваталю (1975). Его доказательство основано на индукции: Предполагая, что для всех *n*< *N* требуется n/3j защит, он доказывает ту же формулу для *n* -- *N*, аккуратно удаляя многоугольника так, чтобы уменьшить его вершин, применяя гипотезу индукции, а затем снова присоединяя удаленную часть. Доказательство распадается на несколько случаев и является довольно тонким.

Три года спустя Фиск нашел очень простое доказательство, занимающее всего одну журнальную страницу (Fisk 1978). Мы представим доказательство Фиска здесь.

Диагонали и триангуляция

Доказательство Фиска в значительной степени зависит от разбиения многоугольника на треугольники с диагоналями. ***Диагональ*** многоугольника *P* - это отрезок прямой между двумя его вершинами a и *b*, которые хорошо видны друг другу. Напомним, что это означает, что пересечением замкнутого отрезка *ab* с fi *P* является именно множество {a, *b* . Другой способ сказать это - открытый отрезок от a до *b* не пересекает fi *P,* поэтому диагональ не может иметь пастбищного контакта с границей.

Назовем две диагонали *непересекающимися*, если их пересечение является подмножеством их конечных точек: У них нет общих внутренних точек. добавить к многоугольнику как больше непересекающихся диагоналей, то его внутренняя часть будет разбита на треугольники. Такое разбиение называется *триангуляцией* многоугольника. Диагонали можно добавлять в произвольном порядке, лишь они были законными и непересекающимися. В общем случае существует множество способов триангулировать заданный многоугольник. На рисунке 1.6 показаны две триангуляции многоугольника *с n*== 14 вершинами.

Доказательство того, что каждый многоугольник может быть триангулирован, мы отложим до раздела 1.2, а пока будем просто предполагать существование триангуляции.

Три раскраски

Чтобы доказать достаточность (n/3j) для *любого* многоугольника, доказательство должно работать для **арбитрного** многоугольника. Итак, пусть произвольный многоугольник *P с n* вершинами. Первый шаг доказательства Фиска - триангулировать *P.* Второй шаг - "вспомнить", что полученный граф может быть 3-цветным. Нам нужно объяснить, что это за граф и что значит 3-цветный.

Пусть G - граф, связанный с триангуляцией, дуги которого - ребра многоугольника и диагонали триангуляции, а узлы - вершины многоугольника. Именно такой граф использовал Фиск. Ак-раскраска графа - это назначение *k* цветов вершинам графа таким образом, что никакие две вершины, соединенные дугой, не имеют одинакового цвета. Фиск утверждает, что каждый триангуляционный граф может быть трехцветным. Мы снова отложим доказательство этого утверждения, но небольшие эксперименты должны сделать его правдоподобным. Трехцветные триангуляции на рисунке 1.6 показаны на рисунке 1.7. Начиная, скажем, с вершины, обозначенной стрелкой, и раскрашивая ее треугольник произвольным образом тремя цветами, остальная часть раскраски полностью принудительна: Других свободных вариантов нет. Грубо говоря, причина, по которой это всегда работает, заключается в том, что принудительный выбор никогда не повторяет предыдущий выбор; а причина, по которой этого никогда не происходит, заключается в том, что базовая фигура является многоугольником (без отверстий, по определению).

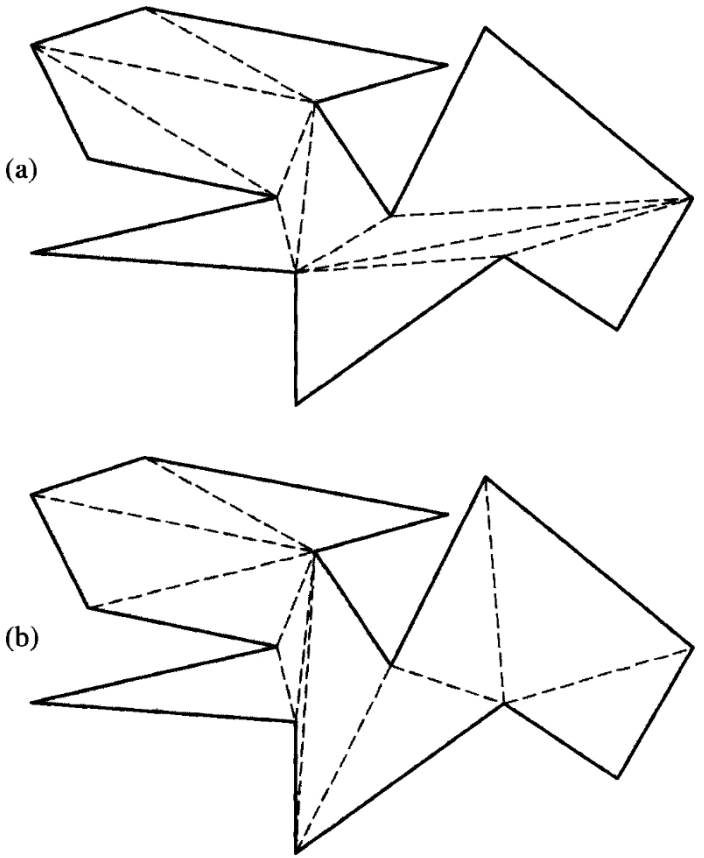


РИСУНОК 1.6 Две триангуляции многоугольника с n= 14 вершинами.

Третий шаг доказательства Фиска заключается в томчто размещение охранников во всех вершинах, которым присвоен один цвет, гарантирует видимое покрытие многоугольника. Он рассуждает следующим образом. Пусть красный, зеленый и синий - цвета, используемые в 3-раскраске. Каждый треугольник должен иметь каждый из трех цветов в трех своих вершинах. Таким образом, каждый треугольник имеет красный узел в одном из углов. Предположим, что в каждом красном узле стоят охранники. Тогда каждый треугольник имеет стража в одной из сторон. Очевидно, что треугольник покрыт стражем в одном из своих углов. Таким образом, каждый треугольник покрыт. Наконец, совокупность треугольников в триангуляции полностью покрывает многоугольник. Таким образом, весь многоугольник покрыт, если в красных узлах установлены охранники. Аналогично, весь многоугольник покрыт, если охранники размещены в зеленых или синих узлах.

В четвертом, последнем шаге доказательства Фиска применяется "принцип голубятни": Если *n* объектов помещены в *k* голубиных ям, то хотя бы одна яма должна содержать не более *n/ k* объектов. Ведь если бы каждая из *k* дырок содержала более *n/ k* объектов, общее количество объектов превысило бы *n.* В нашем случае *n* объектов - это узлы графа триангуляции, а *k* дырок - это 3 цвета. Принцип гласит, что один цвет должен быть использован не более *n/3* раз. Поскольку *n* - целое число, мы можем заключить, что один цвет используется не более ri/3j раз. Теперь у нас есть доказательство достаточности: Просто поместите

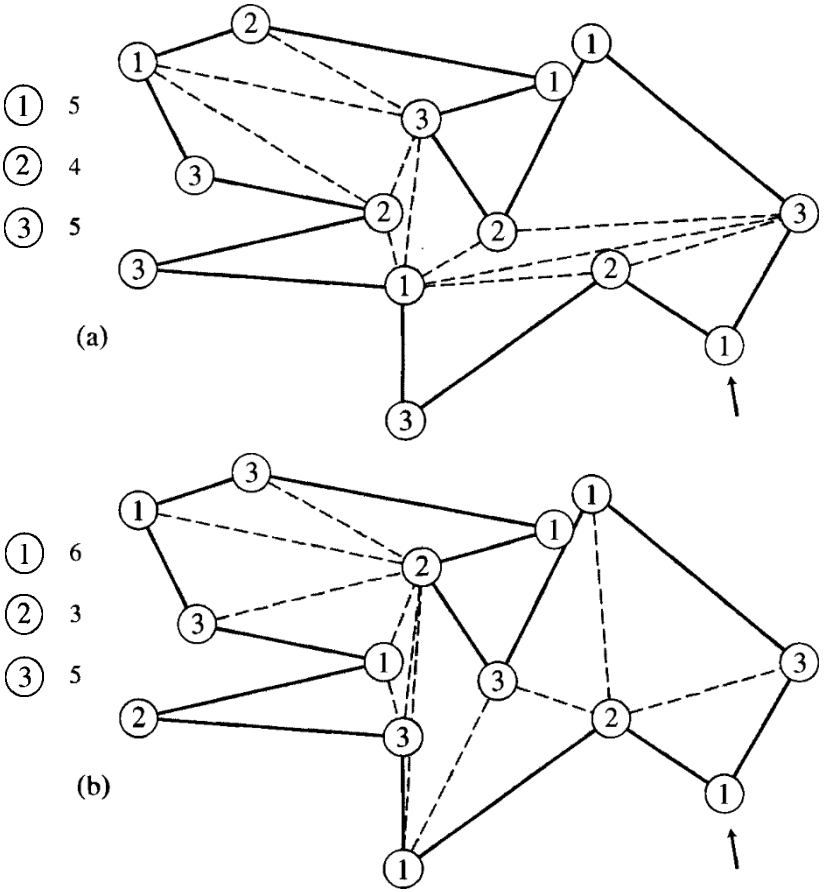


РИСУНОК 1.7 Две трехцветные раскраски многоугольника с n= 14 вершинами, основанные на триангуляциях показано на рисунке 1.6.

охраняет узлы, окрашенные наименее часто используемым цветом в 3-раскраске. Мы гарантируем, что это покроет многоугольник не более чем *G{n)*= t "g3j цветами.

Если вы не находите этот аргумент красивым (или хотя бы очаровательным), то вам не понравится многое в этой книге!

На рисунке 1.7 *n* 14, поэтому *(n/3* 4. На рисунке (a) цвет 2 используется четыре раза; на рисунке (b) этот же цвет используется только три раза. Обратите внимание, что аргумент о трех цветах не всегда приводит к наиболее эффективному использованию защит.

1.1.4. Упражнения

1. *Охрана стен.* Постройте многоугольник *P* и расставьте стражников так, чтобы стражники видели все точки d *P,* но была хотя бы одна внутренняя точка *P*, которую не видит ни один стражник.
2. *Четкая видимость, охранники.* Каков ответ на вопрос Клее о ясной видимости (раздел 1.1.2)? Более конкретно, пусть G'(n) - наименьшее число *точечных стражей*, достаточное для того, чтобы ясно видеть каждую точку в любом многоугольнике из n вершин. Точечные стражи - это стражи, которые могут стоять в любой точке *P,* - они отличаются от *вершинных стражей*, которые могут стоять только в вершинах.

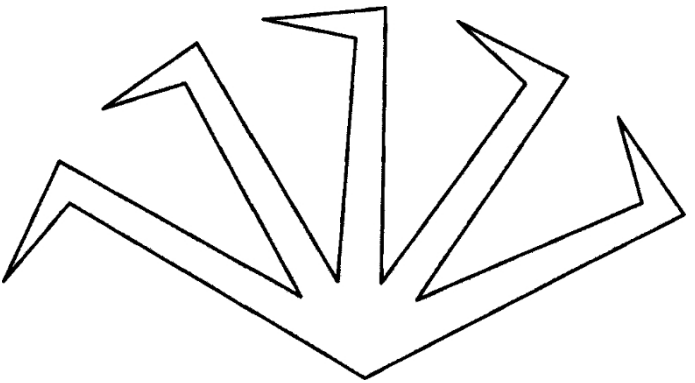
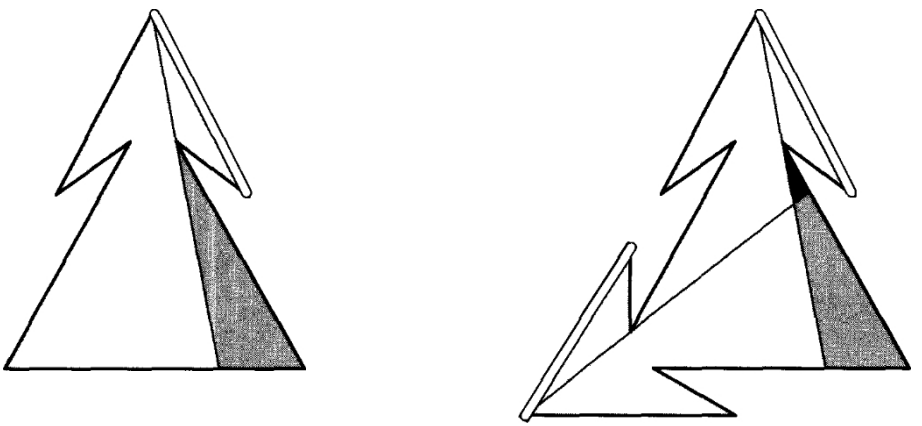


РИСУНОК 1.8 ii/4j необходимы защитные накладки на края (Туссен).



n=7, g=2 n=11, g=3

РИСУНОК 1.9 **Два** многоугольника, для которых требуется [(n+ 1)/4j ограждений ребер.

Ясновидящие охранники сильнее или слабее обычных охранников? Какое соотношение между G'(n) и G(n) следует из их относительной силы? (Определение G(n) дано в разделе 1.1.2) Устанавливает ли доказательство Фиска достаточность [n/3j для четкой видимости? Попробуйте точно определить G'(n).

3. *Ясная видимость, вершинные стражи* (Томас Шермер). Ответ на вопрос 2, но для *вершинных* охранников: охранники ограничены вершинами.

4. *Стражи краев* [открыто]. *Краевой страж* - это страж, который может патрулировать одно ребро *е* многоугольника. Точка *у е Р* покрыта охранником, если существует некоторая точка z e *е* такая, что z может видеть *у.* Другой способ посмотреть на это - представить себе флуоресцентный светильник, длина которого совпадает с е. Часть *Р*, освещаемая этим светом, - множество точек, покрытых охранником.

   Туссен показал, что иногда требуется bn/4j защит ребер, что демонстрирует многоугольник "полусвастика", показанный на рисунке 1.8 (O'Rourke 1987, p. 83). Он предположил, что bn/4j достаточно, за исключением нескольких малых значений n. Это странное исключение обусловлено двумя многоугольниками "со стрелками", показанными на рис. 1.9, которые, похоже, не обобщаются. Эти примеры взяты из Shermer (1992).

   Докажите или опровергните предположение Туссена.

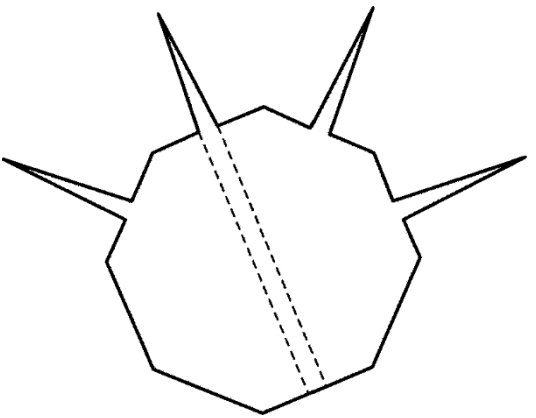


РИСУНОК 1.10 n /5j необходимо защищать края (Туссен).

5. *Охрана краев в звездчатых многоугольниках* [открыто]. *Звездный многоугольник* - это многоугольник, который может быть покрыт единственной (точечной) защитой. Туссен доказал, что для покрытия звездного многоугольника иногда требуется n /5j краевых защит, на примере, показанном на рисунке 1.10 (O'Rourke 1987, p. 119). Гипотеза о том, что n/5j всегда достаточно, оказалась ложной для n = 14 (Subramaniyam & Diwan 1991), но в остальном мало что известно. Докажите или опровергните, что n /5+ *c* достаточно для некоторой постоянной *c* > 0.

6. *Стражи в многогранниках.* Спроектируйте многогранник так, чтобы стражи, установленные в каждой вершине, не закрывали полностью его внутреннюю часть. Многогранник - это трехмерная версия многоугольника, состоящая из многоугольных граней и заключающая в себе объем. Точное определение дается в главе 4 (раздел 4.1). См. O'Rourke (1987, раздел 10.2.2).

1.2. : ТЕОРИЯ

В этом разделе мы докажем, что каждый многоугольник имеет триангуляцию, и установим некоторые основные свойства триангуляций. В последующих разделах (1.4-1.6.5) мы обсудим алгоритмы построения триангуляций.

Естественной реакцией на вопрос "Должен ли каждый многоугольник иметь триангуляцию?" является другой вопрос: "Как многоугольник может *не* иметь триангуляции?". Действительно, он не может не иметь ! Но если вам кажется, что это слишком очевидно для доказательства, рассмотрите эквивалентный вопрос в трех измерениях: Там естественное обобщение ложно! См. O'Rourke (1987, p. 253-4).

1.2.1. Существование диагонали

Ключом к существования триангуляции является доказательство существования ди-агоналей. Как только мы это получим, остальное будет легко. Для доказательства нам понадобится еще один, еще более очевидный факт: у каждого многоугольника должен быть хотя бы один строго выпуклый Vertexq2

Лемма 1.2.1. *Каждый многоугольник должен иметь хотя бы одну строго выпуклую вершину.*

'2Напомним, что (нестрогая) выпуклая вершина может быть коллинеарна со своими соседними вершинами.

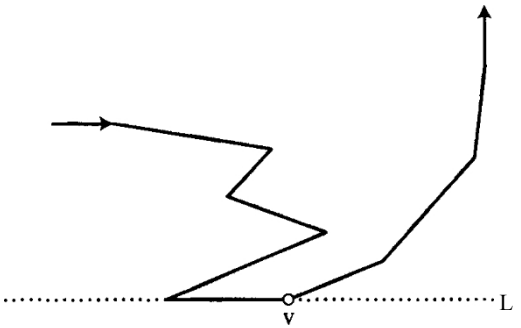


РИСУНОК 1.11 Самая правая нижняя вершина должна быть строго выпуклой.

Доказательство. Если ребра многоугольника ориентированы так, что их направление указывает на обход против часовой стрелки, то строго выпуклая вершина - это поворот налево для идущего по границе, а рефлексивная вершина - поворот направо. Внутренняя часть многоугольника всегда находится слева от этого гипотетического пешехода. Пусть L - прямая, проходящая через наименьшую вершину r многоугольника *P,* наименьшую по минимальной координате y относительно системы координат; если имеется несколько наименьших вершин, пусть r будет самой правой. Внутренняя часть *P* должна быть выше
L. Ребро, следующее за r, должно лежать выше *L.* См. рис. 1.11. Вместе эти условия означают, что пешеход поворачивает налево в точке r и, следовательно, u - строго выпуклая вершина.

Это доказательство можно использовать для построения эффективного теста на ориентацию многоугольника (упражнение 1.3.9[3]).

Лемма 1.2.2 (Мейстерс). *Каждый многоугольник из n* 4 *вершин имеет диагональ.*

Доказательство. Пусть r - строго выпуклая вершина, существование которой гарантируется леммой l.2.l. Пусть a и *b* - вершины, смежные с v. Если *ab* - диагональ, то мы закончили. Предположим, что *ab* не является диагональю. Тогда либо *ab* внешняя к *P,* либо она пересекает fi *P.* В любом , поскольку *n*> 3, замкнутый треугольник *kavb* содержит по крайней мере одну вершину *P*, отличную от a, r, *b.* Пусть z - ближайшая к v вершина *P* в *kavb*, где расстояние измеряется ортогонально прямой, проходящей через *ab.* Таким образом, z - первая вершина в *kavb*, в которую попадает прямая L, параллельная *ab*, двигаясь от r к *ab.* См. рисунок 1.12.

Теперь мы утверждаем, что rz - диагональ *P.* Ибо ясно, что внутренность *kavb*, пересекающаяся с полуплоскостью, ограниченной L и включающей v (заштрихованная область на рисунке), не содержит точек *b P.* Поэтому rx не может пересекать *b P* иначе как в точках r и x, а значит, она - диагональ. O

Теорема 1.2.3 (Триангуляция). *Каждый многоугольник P из n вершин может быть разбит на треугольники добавлением (нуля или более) диагоналей.*

Доказательство. Доказательство проводится по индукции. Если *n* -- 3, то многоугольник является треугольником, и теорема выполняется тривиально.

Пусть *n*> 4. Пусть *d -- ab* - диагональ *P,* что гарантируется леммой 1.2.2. Поскольку *d* по определению пересекает *b P* только в конечных точках, то он разбивает *P* на два многоугольника,

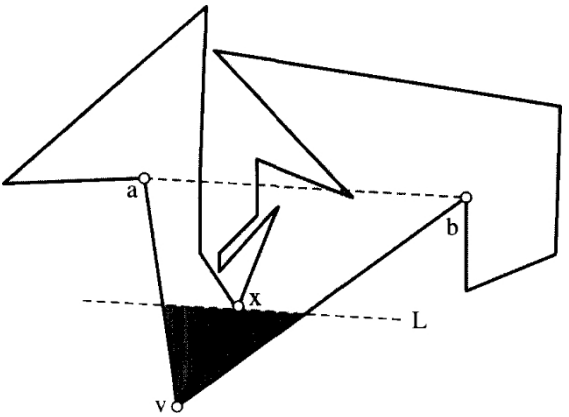


РИСУНОК 1.12 **rx** должен быть диагональю.

каждая из которых использует *d* в качестве ребра, и каждая из которых имеет меньше n вершин; см. рис. 1.13. Причина меньшего количества вершин в каждом из них заключается в том, что этом не добавляется ни одной вершины, и очевидно, что в каждой части есть хотя бы одна вершина в дополнение к *a* и *b.* Применение гипотезы индукции к двум подполигонам завершает доказательство. n

1.2.2. Свойства триангуляций

Хотя в общем случае может существовать большое количество различных способов триангуляции заданного многоугольника (упражнение 1.2.5[4]), все они имеют одинаковое количество диагоналей и треугольников, что легко устанавливается тем же аргументом, что и в Теореме 1.2.3:

Лемма 1.2.4 (Количество диагоналей). *Каждая триангуляция многоугольника P из n вершин использует n* - 3 *диагонали и состоит из n* - 2 *треугольников.*

Доказательство. Доказательство проводится по индукции. Оба утверждения тривиально верны для n - 3.

Пусть n > 4. Разобьем *P* на два многоугольника *P* и P2 с диагональю *d* -- *ab.* Пусть эти два многоугольника имеют n; и "2 вершин соответственно. Имеем, что ii + -2 = - + 2, так как *a* и *b* учитываются в обоих - и -2 Применяя гипотезу индукции к

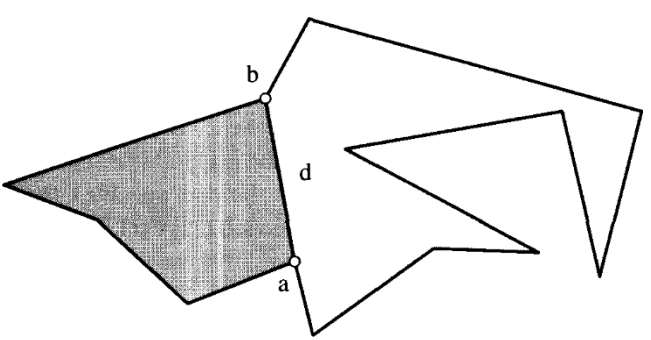


РИСУНОК 1.13 Диагональ разделяет многоугольник на два меньших многоугольника.

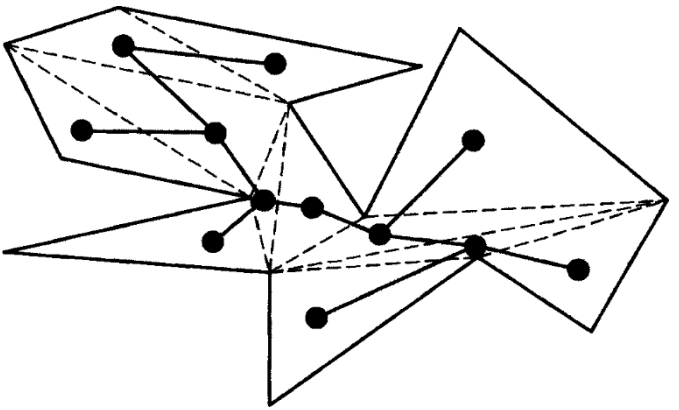


РИСУНОК 1.14 Двойная триангуляция.

подполигонов, мы видим, что всего существует (n; - 3) + t- - 3) + 1 = n - 3 диагонали, причем последний +1 член равен *d.* И существует (n t - 2) -l- (n_2 - 2) = ii - 2 треугольника. p

Корольлярий 1.2.5 (Сумма углов). *Сумма **внутренних** углов многоугольника из n вершин равна (n* - 2)a.

Доказательство. По лемме 1.2.4 существует n - 2 треугольника, и каждый из них вносит вклад u во внутренние углы. n

1.2.5. Двойная триангуляция

Важным понятием в теории графов является "двойник" графа. Нам не понадобится это понятие во всей его общности, мы будем определять конкретные дуальные графы по мере необходимости. В частности, изучение дуала триангуляции позволяет выявить полезную структуру триангуляции. *Двойник T* триангуляции многоугольника - это граф с узлом, связанным с каждым треугольником, и дугой между двумя узлами, если их треугольники имеют общую диагональ. См. Рисунок 1.14.

Лемма 1.2.6. *Двойка T триангуляции является* свободной,[3] *с каждой вершиной степени не более трех.*

Доказательство. То, что каждая вершина имеет степень не более трех, следует из того факта, что у треугольника не более трех общих сторон.

Предположим, что *T* не является деревом. Тогда в нем должен быть цикл *C.* Если этот цикл нарисовать в виде пути u на плоскости, соединив прямыми отрезками середины диагоналей, общих для треугольников, вершины которых составляют *C* (чтобы путь был конкретным), то он должен заключать в себе некоторые вершины многоугольника: а именно по одной конечной точке каждой диагонали, пересекаемой u. Но тогда u должен заключать в себе и точки, внешние по отношению к многоугольнику, так как эти заключенные вершины лежат на fi *P.*
Это противоречит простоте многоугольника. O

[3]* *Дерево* - это связный граф без циклов.

Узлы степени один - это листья *T,* узлы степени два лежат на путях дерева, а узлы степени три - это точки ветвления. Обратите внимание, что *T* является бинарным деревом, если его корень находится в любом узле степени один или два! Учитывая повсеместное распространение двоичных деревьев в информатике, такое соответствие между дуалами триангуляции и двоичными деревьями является удачным и часто может быть использовано (упражнение 1.2.5[7]).

Лемма 1.2.6 приводит к простому доказательству "теоремы о двух ушах" Мейстерса (Meisters 1975), которая, хотя и проста, но весьма полезна. Три последовательные вершины многоугольника *a, b,* с образуют *ухо* многоугольника, если *ac* - диагональ; h - *вершина* уха. Два уха *не пересекаются*, если внутренности их треугольников расходятся.

Теорема 1.2.7 (теорема Мейстерса о двух ушах). *Каждый многоугольник из n>* 4 *вершин имеет по крайней мере два непересекающихся уха.*

Доказательство. Листовой узел в дуальной триангуляции соответствует уху. Дерево из двух или более узлов (по лемме 1.2.4 дерево имеет (n - 2)> 2 узла) должно иметь не менее двух листьев.

1.2.4. Доказательство 5-раскраски

Эта теорема, в свою очередь, приводит к простому доказательству трехцветности триангуляционных графов. Идея состоит в том, чтобы удалить для индукции граф, который, поскольку он "взаимодействует" только на своей одной диагонали, может быть последовательно раскрашен.

Теорема 1.2.8 (5-раскраска). *Граф триангуляции многоугольника P может быть трехцветным.*

Доказательство. Доказательство проводится по индукции от числа вершин n. Очевидно, треугольник может быть трехцветным.

Предположим, что n> 4. По теореме 1.2.7, у *P* есть ухо *Гейба,* с вершиной уха *b.* Сформируйте новый многоугольник *P′*, отрезав ухо: То есть замените последовательность *abc в d P на ac в d P′. У P′ n* - I вершин: В *нем* не хватает только *b.* Примените гипотезу индукции, чтобы раскрасить *P′* в 3 цвета. Теперь верните ухо на место, раскрасив *b* цветом, не используемым в *a* и с. Это и есть 3-раскраска *P.* р

1.2.5. Упражнения

1. *Внешние углы* [легко]. Чему равна сумма внешних углов многоугольника с n вершинами?
2. *Реализация триангуляций.* Доказать или опровергнуть: Каждое двоичное дерево реализуемо как триангуляция, двойственная многоугольнику.
3. *Экстремальные триангуляции.* Какие многоугольники имеют наименьшее различных триангуляций? Могут ли многоугольники иметь уникальные триангуляции? Какие многоугольники имеют наибольшее количество различных триангуляций?
4. *Количество триангуляций* [сложно]. Сколько существует различных триангуляций выпуклого многоугольника *с n* вершинами?
5. *Четырехугольники. Ортогональный многоугольник* состоит из ребер, которые пересекаются ортогонально (например, горизонтальные и вертикальные ребра). Определите понятие "четырехугольника" ортогонального многоугольника - четырехсторонней версии уха - и ответьте на вопрос, каждый ли ортогональный многоугольник имеет четырехугольник согласно вашему определению.

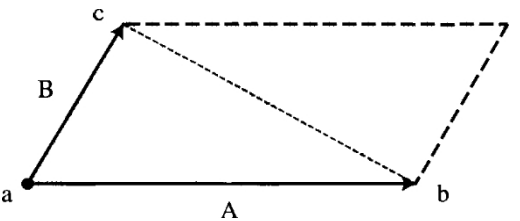


РИСУНОК 1.15 Параллелограмм с поперечным произведением.

6. *Есть ли у невыпуклых полигонов рот?* (Пьер Бошемин). Определите, что три последовательные вершины *a, b,* с **многоугольника** образуют *рот*, если *b* **рефлекторно и замкнутый *kabc* не содержит вершин, отличных от** трех **его углов. Докажите или опровергните: Каждый невыпуклый** многоугольник имеет **рот.**
7. *Вращения деревьев.* Для тех, кто знает, что повороты деревьев используются для балансировки бинарных деревьев:" Интерпретируйте повороты деревьев в терминах триангуляций многоугольников.
8. *Диагонали триангуляции.* Задав список диагоналей многоугольника, образующего триангуляцию, причем каждая диагональ задается индексами конечных точек против часовой стрелки, разработайте алгоритм построения дуального дерева триангуляции. [сложно]: Достичь времени *O(n)* за счет пространства *O* 'n^2).

1.3. ПЛОЩАДЬ МНОГОУГОЛЬНИКА

В этом разделе мы рассмотрим вопрос о том, как вычислить площадь многоугольника. Хотя этот вопрос интересен сам по себе, наша цель - подготовить почву для вычисления вложений в полуплоскости, пересечений между отрезками прямых, отношений видимости и, в конечном счете, привести к алгоритму триангуляции в разделе 1.6.5.

1.5.1. Площадь треугольника

Площадь треугольника равна половине основания, умноженной на высоту. Однако эта формула не совсем удобна, если нам нужна площадь треугольника *T*, тремя вершинами которого являются произвольные точки *a, b, c.* Обозначим эту площадь как *A(T).* Основание находится легко: *a - b* ,[5], но высота не так легко находится из координат, если только треугольник не ориентирован так, что одна из сторон параллельна одной из осей.

1.3.2. Кросс-продукт

Из линейной алгебры мы знаем, что величина поперечного произведения двух векторов равна площади параллелограмма, который они определяют: Если *A* и *B* - векторы, то *A* x *B* - площадь параллелограмма со сторонами *A* и *B,* как показано на рис. 1.15. Поскольку любой треугольник можно рассматривать как половину параллелограмма, это немедленный метод вычисления площади из координат. Пусть *A -- b - a* и *B c - a.* Тогда площадь равна половине длины *A* x *B.* Поперечное произведение можно вычислить из

"См., например, Cormen, Leiserson & Rivest (1990, pp. 265-7).
" |a - *b* - длина вектора *a - b,* иногда пишется ||n - *b* .

следующий определитель, где i, j и 'k - единичные векторы в направлениях *x, y* и e соответственно:

$$\begin{matrix} i &] & 'k \\ \text{Ao} & A & A\,2 - (Ai\,Bz - A\,y\,B)i + - (Az\,Bo - Ao\,By)j - \text{l} - (At\,Bi - Ai\,Bobk. \\ \text{По ссылке} & \text{Bi} & \text{Bz} \end{matrix} \tag{1.1}$$

Для двумерных векторов *Az* =- Bz= 0, поэтому вычисления сводятся к (oBi - *A* Bo) *"k:* Поперечное произведение - это вектор, нормальный (перпендикулярный) к плоскости треугольника. Таким образом, площадь определяется следующим образом

$$A[T) \qquad {}_{2}(AoB\ \text{i} - A\ By)\ .$$

Подстановка *A*= b - *a* и *B* -- *c* - *a* дает

$$2A(T) - \text{ao}/2] - a\,|\,bo + a\,C \quad - \text{age}] - \text{l} - \text{t'o}\hat{}\text{i} - Clb\,j \tag{1.2}$$

$$(\text{два} \quad 2\text{o})(<\text{i} - a\ \text{i}\ 1 - DC - ao1\ b\,j - \quad \text{i}) \tag{1.3}$$

Это позволяет достичь нашей непосредственной цели: получить выражение для площади треугольника как функции координат его вершин.

1.5.5. Детерминантная форма

Существует другой способ представления кросс-продукта, который формально идентичен, но обобщается на более высокие размерности.[16]

Выражение, полученное выше (уравнение 1.3), является значением определителя 3 x 3 трех координат точки, при этом третья координата заменяется на 1:'[7]

$$\begin{matrix} \text{ao a} & & 1 \\ \text{bo} & bf & 1 \\ <\text{o} & \hat{}\text{i} & 1 \end{matrix} \qquad (bq - \text{o})(< \text{i} - a\ ) - (\hat{}\text{o} - ao)(b - a\ ) - 2A\ T). \tag{1.4}$$

Этот определитель изучается в упражнении l.6.8[l]. Мы подведем итог в лемме.

Лемма 1.5.1. ***Площадь*** *треугольника T -= (a, b,* c) *в* ***два раза больше*** *площади треугольника T.*

$$2A[T) - /2 \quad \begin{matrix} \text{a}_{\text{O}} & a & 1 \\ & bj & \\ \text{co} & 1 & \end{matrix} \quad 1 - (\text{два} - \text{при})(\text{c}; - -\text{i}\ )\ 1 \qquad DC \quad - ao1(b - a\ ). \tag{1.5}$$

'6Обратите внимание, что операция кросс-продукции ограничена трехмерными векторами (или двумерными векторами с нулевой третьей координатой). Более точно рассматривать кросс-продукт внешнее произведение, производящее не другой вектор, а "бивектор". См., например, Koenderink (1990).

[7] Можно рассматривать каждый ряд как точку в "однородных координатах", при этом третья координата нормирована на 1.

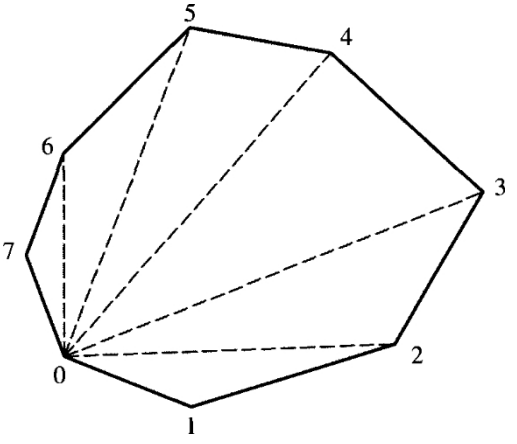


РИСУНОК 1.16 Триангуляция выпуклого многоугольника. Центр веера находится в точке 0.

1.5.4. Площадь выпуклого многоугольника

Теперь, когда у нас есть выражение площади треугольника, легко найти площадь любого многоугольника, сначала проведя его триангуляцию, а затем просуммировав площади треугольников. Однако было бы приятно избежать довольно сложного этапа триангуляции, и это действительно возможно. Прежде чем перейти к этому вопросу, мы рассмотрим выпуклые многоугольники, триангуляция которых тривиальна.

Каждый выпуклый многоугольник может быть триангулирован как "веер", со всеми диагоналями, приходящимися на общую вершину; это можно сделать с любой вершиной, служащей "центром" веера. См. рис. 1.16. Поэтому площадь многоугольника с вершинами u , - - - - ' n- l, обозначенными против часовой стрелки, может быть вычислена как

$$\mathcal{A}(P) = \mathcal{A}(v_0, v_1, v_2) + \mathcal{A}(v_0, v_2, v_3) + \cdots + \mathcal{A}(v_0, v_{n-2}, v_{n-1}). \tag{1.6}$$

Здесь -o - это центр вентилятора.

Мы подготовимся к результату, который докажем в Теореме 1.3.3 ниже, рассмотрев выпуклые и невыпуклые четырехугольники, где соответствующие соотношения очевидны.

1.5.5. Площадь выпуклого четырехугольника

Площадь выпуклого четырехугольника 9= *(a, b, c, d)* может быть записана двумя способами, в зависимости от двух различных триангуляций (см. рис. 1.17):

$$A(Q) = A(a, b, c) + A(a, c, d) = A(d, a, b) + A(d, b, c). \tag{1.7}$$

Записав выражения для площадей с помощью уравнения (1.2) для двух членов первой триангуляции, получим

$$2A(9) = \text{ally} - a\,by + '2]\,\hat{}o - 2o\hat{}\,i + bo<i - <obi$$
$$-1\text{-}ao\hat{}i - a\,'o - |- a\,i\,d - aod\,f + Modi - dogi \tag{1.8}$$

Обратите внимание, члены *a* 'o - ao появляются в *A(a, b,* c) и в *A(a, c, d)* с противоположными знаками, и поэтому они аннулируются. Таким образом, члены, "соответствующие" диагонали *ac*, отменяются;

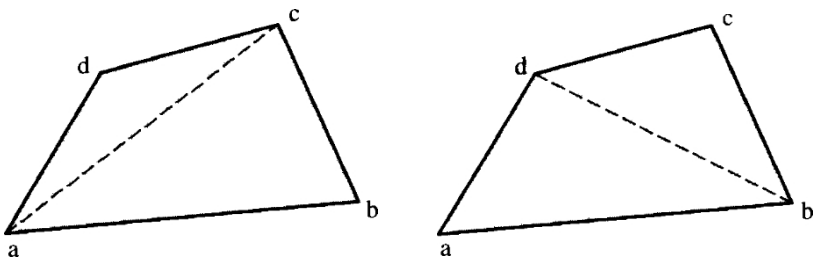


РИСУНОК 1.17 Две триангуляции выпуклого четырехугольника.

Аналогично отменяются члены, соответствующие диагональным *db* во второй триангуляции. Таким образом, мы приходим к точно такому же выражению независимо от триангуляции, как, конечно, и должно быть.

Обобщая, мы видим, что на каждое ребро многоугольника приходится два члена, а на внутренние диагонали - ни одного. Таким образом, если координаты вершины ri равны *xi* и *ii* - *wice, то площадь выпуклого многоугольника определяется следующим образом

$$2\,A(P) \sum_{f=0} (x_i y_{i+} - y_i x_{i+}). \qquad (1.9)$$

Вскоре мы увидим, что это уравнение справедливо и для невыпуклых многоугольников.

1.5.6. Площадь невыпуклого четырехугольника

Теперь предположим, что у нас есть невыпуклый четырехугольник 9= (n, *b, c, d)*, как показано на рисунке 1.18. Тогда существует только одна триангуляция, использующая диагональ *db.* Но мы только что показали, полученное алгебраическое выражение не зависит от выбранной диагонали, поэтому должно быть так, что уравнение

$$A(Q) - A(a, b, c) + A\{a, c, d)$$

по-прежнему верно, даже если диагональ *ac* внешняя по отношению к 9. Это уравнение имеет очевидную интерпретацию: *A(a, c, d)* отрицательно, и поэтому оно вычитается из окружающего

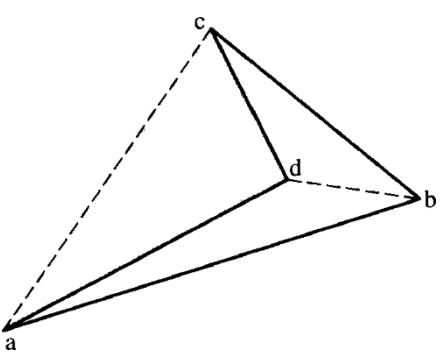


РИСУНОК 1.18 Триангуляция невыпуклого четырехугольника. Заштрихованная область *A(a, d,* c) является отрицательной.

треугольник *kabc.* И действительно, обратите внимание, что *(a, c, d)* - это путь по часовой стрелке, поэтому формула произведения кросс-продуктов показывает, что площадь будет отрицательной.

Явление, наблюдаемое с невыпуклым четырехугольником, является общим, что мы сейчас и продемонстрируем.

1.3.7. Площадь от произвольного центра

Теперь мы формализуем наблюдения предыдущих параграфов, которые затем будем использовать для получения площади невыпуклых многоугольников общего вида.

Обобщим метод суммирования площадей треугольников в триангуляции на суммирование площадей по произвольной, возможно, внешней точке *p.* Пусть *T kabc* треугольник с вершинами, ориентированными против часовой стрелки, и пусть *p* - любая точка на плоскости. Тогда мы утверждаем, что

$$A(T) = A(p, a, b) + A(p, b, c) \qquad A(p, c, a). \tag{1.10}$$

Рассмотрим рисунок 1.19. При *p* -- pi первый член уравнения (1.10), *A(p , a, b)*, отрицателен, поскольку вершины направлены по часовой стрелке, а два остальных члена положительны, поскольку вершины направлены против часовой стрелки. Теперь заметим, что *A(p , a, b)* вычитает ровно ту часть четырехугольника *(p , b, c, a)*, которая лежит вне *T,* оставляя общую сумму именно *A(T)*, как и утверждалось.

Аналогично, из *p* -- pz, как *A(pz, a, b)*, так и *ATP2, b,* c) отрицательны, потому что вершины расположены по часовой стрелке, и они удаляют из *A pz, , a),* которая положительна, ровно столько, сколько нужно, чтобы оставить *A(T).*

Все остальные позиции для *p* в плоскости, не являющейся внутренней для *T*, эквивалентны либо

p

в силу симметрии; и, конечно, уравнение справедливо, когда *p* является внутренним, как мы утверждали в

Раздел 1.3.4. Поэтому мы установили следующую лемму:

Лемма 1.3.2. ***Если*** *T -- kabc - треугольник с вершинами, ориентированными против часовой стрелки,* ***и***
p - любая точка на плоскости, тогда

$$A(T) = A(p, a, b) + A(p, b, c) \qquad A(p, c, a) . \tag{1.11}$$

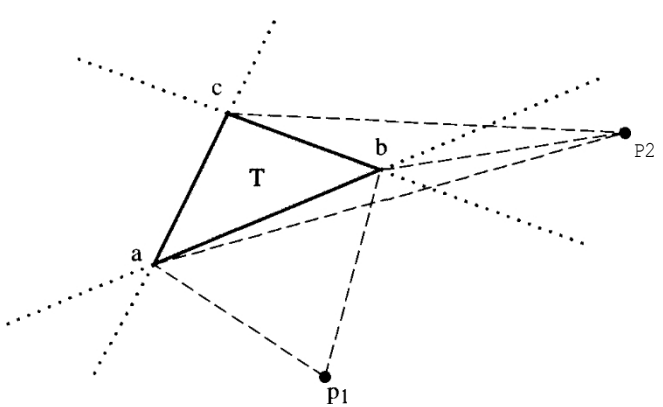


РИСУНОК 1.19 Площадь *T* на основе различных внешних точек *p* pz

Теперь мы можем обобщить предыдущую лемму и установить то же уравнение (в общем виде) для произвольных многоугольников.

Теорема 1.5.5 (Площадь многоугольника).[8] *Пусть многоугольник (выпуклый или невыпуклый) P* ***имеет*** *вершины* u , u , - - - ' n- I *, обозначенные против часовой стрелки, и пусть p - любая точка на плоскости. Тогда*

$$\mathcal{A}(P) = \mathcal{A}(p, v_0, v_1) + \mathcal{A}(p, v_1, v_2) + \mathcal{A}(p, v_2, v_3) + \cdots + \mathcal{A}(p, v_{n-2}, v_{n-1}) + \mathcal{A}(p, v_{n-1}, v_0). \tag{1.12}$$

!I i' i , yr), это выражение эквивалентно уравнениям

$$2\mathcal{A}(P) = \sum_{i=0}^{n-1} (x_i y_{i+1} - y_i x_{i+1}) \tag{1.13}$$

$$= \sum_{i=0}^{n-1} (x_i + x_{i+1})(y_{i+1} - y_i). \tag{1.14}$$

Доказательство. Мы доказываем уравнение суммы площадей по индукции от числа вершин n в P. Базовый случай, n= 3, устанавливается Леммой 1.3.2.

Предположим, что уравнение (1.12) справедливо для всех многоугольников с n - 1 вершинами, и пусть *P* - многоугольник из n вершин. Согласно теореме 1.2.7, у *P* есть "ухо". Перенумеруем вершины *P* так, чтобы *E* (n-2' o- i , r) было ухом. Пусть n-] - многоугольник, полученный при удалении *E.* По гипотезе индукции,

$$\mathcal{A}(P_{n-1}) = \mathcal{A}(p, v_0, v_1) + \cdots + \mathcal{A}(p, v_{n-3}, v_{n-2}) + \mathcal{A}(p, v_{n-2}, v_0).$$

По лемме 1.3.2,

$$\mathcal{A}(E) = \mathcal{A}(p, v_{n-2}, v_{n-1}) + \mathcal{A}(p, v_{n-1}, v_0) + \mathcal{A}(p, v_0, v_{n-2}).$$

Поскольку *A(P) - A(n -l)+ A(E),* имеем

A(P)= A(p, c , ci) - A+ + *p,* n-3 ' n-2)+ *A pp' n -2 '* 0)

*+ A(P, * -z, ^n -i)* *A(P. * -1. *o)+ A(p,* *o, ^n -z

Но заметим, что *A(p* o. n-z= - *A p. -n* -z, а не Отмена этих членов приводит к искомому уравнению.

Уравнение (1.13) получается путем разложения определителей и отмены членов, как объяснялось в разделе 1.3.5. Уравнение (1.14) можно считать эквивалентным, если перемножить и снова отменить члены. О

"Эту теорему можно рассматривать как дискретную версию теоремы Грина, которая связывает интеграл по границе области с интегралом по ее внутренней части: qp ui' *p du',* где 'zi "1-форма" (см., например, Buck & Buck (1965, p. 406) или Koenderink (1990, p. 99)).

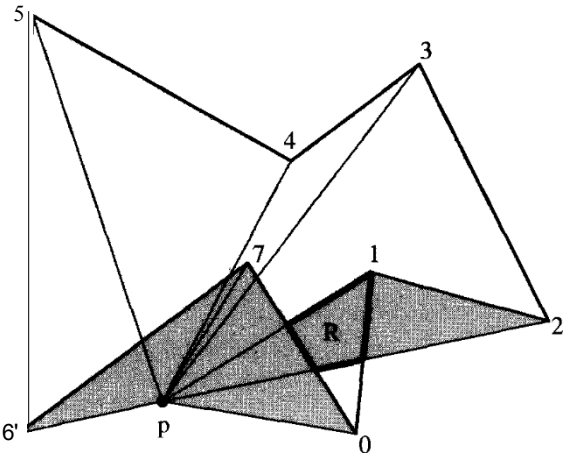


РИСУНОК 1.20 Вычисление площади невыпуклого многоугольника из точки *p.* Более темные треугольники ориентированы по часовой стрелке, поэтому их площадь отрицательна.

Уравнение (1.14) можно вычислить с помощью одного умножения и двух сложений на каждый член, в то время как уравнение (1.13) использует два умножения и одно сложение. Поэтому вторая форма более эффективна в большинстве реализаций.

На рисунке 1.20 треугольники *Up* 12, fi p67 и fip70 ориентированы по часовой стрелке, а остальные - против часовой. Можно считать, что треугольники против часовой стрелки присоединяют к каждой покрываемой ими точке заряд +1, тогда как треугольники по часовой стрелке присоединяют заряд - l. Тогда точки fi из *Up* 12, которые попадают внутрь многоугольника (обозначены на рисунке), получают заряд - l от этого треугольника по часовой стрелке; но fi также покрывается двумя треугольниками против часовой стрелки, fip01 и fip23. Поэтому fi имеет чистый заряд + 1. Аналогично, каждая точка внутри *P* имеет чистый заряд -J-1, а каждая точка снаружи - чистый заряд 0.

1.5.8. Объем в трех и более высоких измерениях

Одно из преимуществ детерминантной формулировки площади треугольника в лемме 1.3.1 заключается в том, что она распространяется непосредственно на более высокие измерения. В трехмерном пространстве объем тетраэдра *T* с вершинами *a, b, c, d* равен

$$6V(T) = \begin{vmatrix} ao & a & az & 1 \\ бо & би & bz & 1 \\ c0 & cl & cz & 1 \\ сдел\ ать & ди & dz & 1 \end{vmatrix} \qquad (1.15)$$

$$= -(az - dz)(b\ i - d)'ct - точка + \{a - d)(bz - dz) <o - d)$$

$$+ (2 - dz)(два\ \ do)(<i - d) - \{ao \qquad dQ)(bz - d\,y*z\{C\,f - d)$$

$$- (fi\ j - d\,f)\ bo - do)(Hz - d2) + (a0 - do)\ b\ j - d\backslash)\ <z - dz-) \qquad (1.16)$$

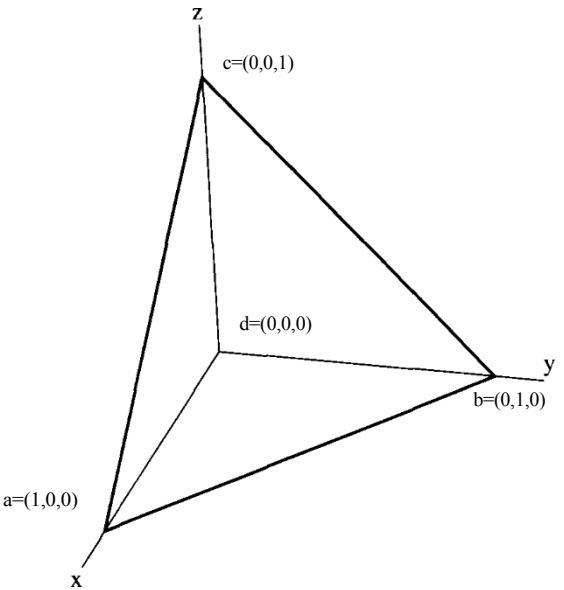


РИСУНОК 1.21 Тетраэдр в начале координат.

Этот объем подписан; он положителен, если *(a, b, c)* образуют контур против часовой стрелки, если смотреть со стороны, удаленной от *d,* так что нормаль к грани, определенная по правилу правой руки, направлена наружу. Например, пусть *a* - (1, 0, 0), *b* (0, 1, 0), c = (0, 0, l), a *d* (0, 0, 0). Тогда *(a, b,* c) находится снаружи против часовой стрелки; см. рис. 1.21. Подстановка в уравнение (1.15) дает определитель, равный 1, поэтому *(T j* $= \frac{1}{6}$. Это соответствует правилу умножения площади основания на высоту: $_{y}$ у- 1. Мы воспользуемся этой формулой объема позже для вычисления "выпуклого корпуса" точек в трех измерениях (глава 4).

Примечательно, что Теорема 1.3.3 обобщается также непосредственно: Объем многогранника может быть вычислен суммированием (знаковых) объемов тетраэдров, образованных произвольной точкой и каждой треугольной гранью многогранника (упражнение 4.7[7].) Здесь все грани должны быть ориентированы против часовой стрелки снаружи.

Более того, уравнение (1.15) обобщается на более высокие измерения *d,* давая объем d-мерного "симплекса" (обобщение тетраэдра на более высокие измерения), умноженный на константу d!

1.3.9. Упражнения

1. *Тройное произведение.* Интерпретируйте выражение определителя (уравнение (1.4)) для площади треугольника в терминах тройного векторного произведения.

   Если *A, B* и *C* - трехмерные векторы, то их тройное произведение *A* - *{B* x *C).* Это скаляр со значением, равным объему параллелепипеда, определяемому тремя векторами, определяемому в том же смысле, в каком два вектора определяют параллелограмм. Значение такое же, как и у определителя

$$A\text{-}'B \times C) - \begin{vmatrix} A_o & A & Az \\ B_o & B & Bz \\ C_0 & C & C_2 \end{vmatrix}.$$

Предполагая, что этот определитель является объемом параллелепипеда, докажите, что уравнение (1.4) в два раза площади указанного треугольника.

2. *Ориентация многоугольника: из области* [легко]. Если задан список вершин простого многоугольника в порядке обхода границ, как определить его ориентацию (по часовой стрелке против часовой стрелки), используя теорему 1.3.3?
3. *Ориентация многоугольника.* Используя доказательство леммы 1.2.1, разработайте более эффективный алгоритм ориентации многоугольника Gn.
4. *Объем куба.* Вычислите объем единичного куба (длина стороны l) с помощью аналога уравнения (1.12), используя одну вершину в качестве *р.*

1.4. ИТП

Оставшаяся часть главы представляет собой довольно длинное "отступление" в сторону проблем реализации. Цель - представить код для вычисления триангуляции. Это связано с обнаружением пересечения двух сегментов - на первый взгляд тривиальной задачей, которая часто реализуется неправильно. Мы будем подходить к определению пересечения сегментов, используя вычисление площадей из раздела 1.3. Начнем с нескольких вопросов представления.

1.4.1. Представление точки

Массивы против записей

Все точки будут представлены массивами с соответствующим количеством координат. Обычно точку представляют в виде записи с полями x и y, но это исключает использование циклов for для перебора координат.[9] Может показаться, что нет особой необходимости писать цикл for для перебора только двух индексов, но я нахожу его более понятным, и он, конечно, обобщается на более высокие измерения.

Целочисленные и вещественные числа

По возможности мы представлять координаты целыми числами, а не числами с плавающей точкой. Это позволит нам избежать проблемы погрешности округления чисел с плавающей точкой и написать код, который будет достоверно корректен в диапазоне значений координат. Числовые погрешности - важная тема, которая будет обсуждаться в разных местах книги (например, в разделах 4.3.5 и 7.2). Очевидно, что от привычки использовать целые числа придется отказаться, когда мы будем вычислять, например, точку пересечения двух отрезков прямой. Определения типов будут изолированы, так что модификация кода для работы с различными разновидностями типов координатных данных может быть выполнена в одном месте.

Определение типа точки

Все идентификаторы типов начинаются со строчной буквы L. Все определяемые константы полностью пишутся в верхнем регистре. Суффиксы i и d обозначают типы ink eger и doubt e соответственно. См. код l.l. В математических выражениях мы будем писать p_0 и pt для p [0] и p (1] .

[9] То есть, исключает его в большинстве языков программирования.

```
#define X 0
#define Y 1
typedef enum {FALSE, TRUE ) bool;

#de f ine   DIM 2                    /* Размерность точек */
Cypede I    и CPoink i DIM) ;        /* Тип целочисленной точки */
```

Код 1.1 Тип точки.

1.4.2. Представление многоугольника

Основные опции здесь - использовать массив или список, а если последний, односвязный или двусвязный, а также линейный или кольцевой.

Массивы привлекательны для ясности кода: Структура циклов и приращения индексов в массивах несколько понятнее, чем в списках. Однако вставка и удаление точек в массивах неуклюжи. Поскольку разрабатываемый нами код триангуляции будет отрезать уши, мы пожертвуем простотой, чтобы получить легкость удаления. В любом случае, нам придется использовать идентичные структуры для кода выпуклого корпуса в главах 3 и 4, так что вложенные здесь средства окупятся позже. Стремясь к такой общности, мы решили использовать для представления многоугольника двусвязный круговой список. Основная ячейка структуры данных представляет собой одну вершину, LVer cexs Lruc Our e, первичным полем данных которой является LPo ink. Для связи каждой вершины с соседними вершинами предусмотрены указатели next и prey. См. код 1.2. Для вывода на печать включен целочисленный индекс vnum, а другие поля (например, bool ear) будут добавляться по мере необходимости.

```
typedef struct tVertexStructure tsVertex;          /*Usedon/inNfW{} */
typedef tsVertex *tVertex;
struct tVertexStructure {

   int          vnum;                  /* Индекс */
   tPointi      v;                     /* Координаты */
   bool         ear;                   /* TRUE, если ухо */
   tVertex      следующий,
                предыдущий;

tVertex Вершины= NULL;                 /* "Глава" кругового списка. */
```

Код 1.2 Вершинная структура.

В любой момент времени поддерживается глобальная переменная ver t i ces, указывающая на некоторую ячейку вершины. Она будет служить "головой" списка при итеративной обработке. Циклы по всем вершинам будут иметь вид, показанный в коде 1.3. Следует быть осторожным, если при обработке в цикле удаляется ячейка, на которую указывает ver tex.

```
tVertex v;
v= вершины;

    /* Обработайте вершину v */
    v= v->next;
) while ( v != vertices );
```

Код 1.3 Цикл для обработки всех вершин.

Нам понадобятся две базовые процедуры обработки списков для вершинных структур: одна для выделения нового элемента (NEW) и другая для добавления нового элемента в список (ADD). Забегая вперед, скажем, что в последующих главах мы напишем их как макросы, причем NEW будет принимать тип в качестве одного параметра. Таким образом, процедуры можно использовать для разных типов. (Си не позволяет манипулировать переменными без учета типа, но макросы основаны на тексте и не обращают внимания на типы). См. код 1.4. ADD сначала проверяет, не является ли head не-NULL, и если да, то вставляет ячейку перед head; если нет, то head указывает на добавленную ячейку, которая затем единственной ячейкой в списке. В результате в серии ADDS n-я точка добавляется до Oth (головы), но после (n - l)-й точки.

```
#define EXIT FAILURE l char
*malloc();

#define NEW(p, type) \
          if ((p=(type *) malloc (sizeof(type)))) == NULL) {\

                     N          )памяти!\n");\
          r tt XI     :ILURE


#define ADD( head, p ) if ( head ) {\ p-
          >next = head;\
          p->prev= head->prev;\
          head->prev = p;\
          p->prev->next= p;\

     иначе (\
          =p;\
          head->next= head->prev= p;\



#define FREE(p)         if (p) {free ((char *) p); p= NULL; }
```

Код 1.4 Макросы NEW и ADD. (Обратные слеши продолжают строки, чтобы препроцессор не воспринимал их как командные строки). FREE используется в главах 3 и 4.

1.4.3. Код для зоны

Вычисление площади многоугольника теперь к простой реализации уравнений (1.12) или (1.13). Первый вариант, с *p* -- о, показан в коде 1.5.

Используются структуры данных и соглашения, установленные в предыдущем разделе.

```
int       Area2( tPointi a, tPointi b, tPointi c )

   возврат
             (blX] - a[Xl) * (c[Y] - a[Y1)
             (c[X] - a[X]) * (b[YJ - a[Y]);



int       AreaPoly2( void )

   int       сумма= 0;
   tVertex p, a;

   p = вершины;        /*£ied. */
   a = p->next;       /*iWoWng. */
   do {
       sum+= Area2( p->v, a->v, a->next->v ); a = a-
       >next;
   ) while ( a->next != vertices ); return sum;
```

Код 1.5 Ar e a2 и Ar e a Po ly2.

Существует интересная потенциальная проблема с Ar ea2: если координаты большие, то умножение координат может привести к переполнению целочисленного слова, о чем, к сожалению, не сообщает большинство реализаций на языке С. Для **Area2** мы использовали выражение из уравнения (1.3), а не из (1.2), поскольку в первом случае используется меньше умножений и умножаются разности координат. Тем не менее, вопрос остается открытым, и мы вернемся к нему в разделе 4.3.5. См. упражнение 1.6.4[l].

1.5. СЕГМЕНТ И МЕЖСЕКТОРНОЕ ПРОСТРАНСТВО

1.5.1. Диагонали

Наша цель - разработать код для триангуляции многоугольника. Ключевым шагом будет нахождение диагонали многоугольника - прямой линии между двумя вершинами r, и r,. Отрезок u, uj не будет диагональю, если он заблокирован частью границы многоугольника. Чтобы быть заблокированным, *i* j должен пересекать ребро многоугольника. Обратите внимание, что если **i*< j пересекает ребро *e* только в своей конечной точке, возможно, лишь пасуя перед границей, он все равно будет заблокирован, так как диагонали должны иметь четкую видимость.

Из определения диагонали (раздел 1.5.1) вытекает следующее:

Лемма 1.5.1. *Отрезок s v, v является диагональю P, если*

1. *для всех ребер e из P, которые не совпадают ни с v, ни с v, s и e не пересекаются: s C e B;*
2. *s является внутренним для P в окрестности v, и v .*

Условие (l) этой леммы сформулировано так, что "диагональность" отрезка можно определить без нахождения фактической точки пересечения между *s* и каждым e: Требуется только булевский предикат пересечения отрезков. Обратите внимание, что при более прямой реализации определения это не так: Диагональ пересекает ребра многоугольника только в конечных точках диагонали. Такая формулировка потребовала бы вычисления точек пересечения и последующего сравнения с конечными точками. Цель условия (2) - отличить внутренние диагонали от внешних, а также исключить коллинеарное пересечение с инцидентным ребром. Мы вернемся к этому условию в разделе 1.6.2. Теперь мы переходим к разработке кода для проверки условия непересечения.

1.5.2. Проблемы со склонами

Пусть r, U, = *ab* и *e cd.* Обычно при решении задачи о том, пересекаются ли *ab* и *cd*, нужно найти точку пересечения прямых *L* и *L2*, содержащих отрезки, решив два линейных уравнения в форме "наклон - перехват", а затем проверить, что точка попадает на отрезки. Этот метод явно будет работать, и его не так уж сложно закодировать. Но код грязен и склонен к ошибкам; требуется удивительное усердие, чтобы сделать его точно правильно. Есть два особых случая: вертикальный отрезок, наклон содержащей его прямой бесконечен, и параллельные отрезки, содержащие их прямые не пересекаются. Оба случая приводят к делению на ноль при вычислениях, чего необходимо избегать с помощью специального кода. Кроме того, проверка того, что точка пересечения попадает на отрезки, может привести к проблемам с точностью вычислений.

Чтобы обойти эти проблемы, мы полностью избегаем склонов.

1.S.3. Слева

Вопрос о том, пересекаются ли два отрезка, можно решить с помощью предиката Le f L, который определяет, находится ли точка слева от направленной прямой. Как Le fL используется для определения пересечения, будет показано в следующем разделе. Здесь мы сосредоточимся на самом Le fL.

Направленная прямая определяется двумя точками, заданными в определенном порядке *{a, b).* Если точка с находится слева от прямой, определяемой точками *(a, b),* то тройка *(a, b,* c) образует контур против часовой стрелки: Вот что значит быть слева от прямой. См. рис. 1.22.

Теперь связь с подписанной площадью окончательно ясна: с находится слева от *(a, b)*, если площадь треугольника против часовой стрелки, *A(a, b,* c), положительна. Поэтому мы можем реализовать предикат Le f C одним вызовом Ar ea2 (код 1.6).

Заметим, что Le f t можно реализовать, найдя уравнение прямой, проходящей через *a* и *b,* и подставив в уравнение координаты точки c. Этот метод был бы прост, но подвержен возражениям, связанным с особыми случаями, о которых говорилось ранее. Код района, напротив, не имеет особых случаев.

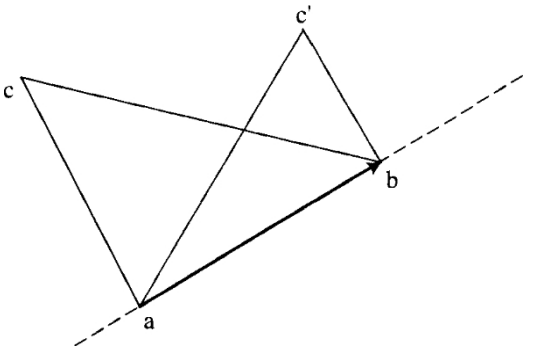


РИСУНОК 1.22 с находится слева от *ab*, если *kabc* имеет положительную площадь; *4 abc'* также имеет положительную площадь.

```
bool      Left( tPointi a, tPointi b, tPointi c )

   return Area2( a, b, c )> 0;


bool      Lefton( tPointi a, tPointi b, tPointi c )

   return Area2( a, b, c ) >- 0;


bool      Collinear( tPointi a, tPointi b, tPointi c

   return Area2( a, b, c )== 0;
```

Codel.6 Left.

Что происходит, когда с коллинеарно с *ab** Тогда определенный треугольник имеет нулевую площадь. Таким образом, мы имеем счастливое обстоятельство, что исключительная геометрическая ситуация соответствует исключительному числовому результату. Поскольку иногда будет полезно различать коллинеарность, мы пишем для этого отдельный предикат Co11 in ear predicate[20], а также Le f LOn, что дает нам эквивалент =, < и <; см. снова Код 1.6.

Обратите внимание, что в этих процедурах мы сравниваем удвоенную площадь с нулевой: Мы не сравниваем саму площадь. Причина в том, что площадь может быть не целым числом, а мы бы предпочли не покидать удобную область целых чисел.

1.5.4. Булево пересечение

Если два отрезка *ab* и *cd* пересекаются в своих внутренних частях, то с и *d* разделены прямой *L* , содержащей *ab:* с находится по одну сторону, а *d* - по другую. Аналогично, *a* и *b*

*Если в конкретном приложении требуются координаты с плавающей точкой, этот предикат нужно будет изменить, так как он зависит от точного равенства с Zero.

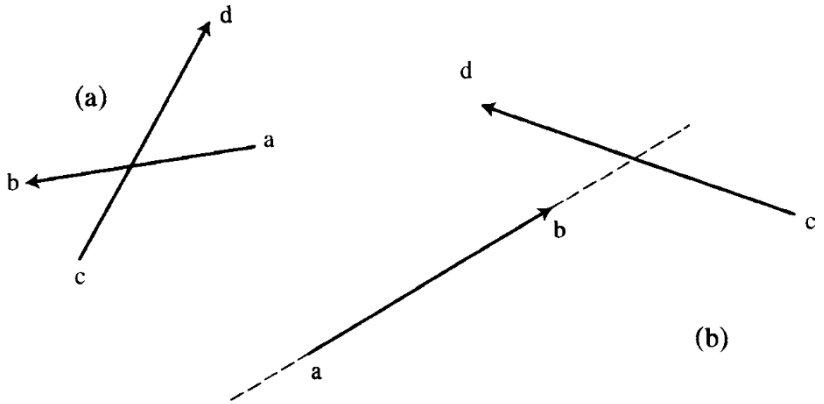


РИСУНОК 1.23 Два отрезка пересекаются (a), если их конечные точки разделены определенными прямыми; обе пары конечных точек должны быть разделены (b).

разделить на Lz. линию, содержащую *cd.* См. рис. 1.23(a). Как видно из рисунка 1.23(b), ни одно из этих условий само по себе не гарантирует пересечения, но очевидно, что оба условия вместе достаточны. Это приводит к прямому коду для определения *правильного* пересечения, когда два отрезка пересекаются в точке, внутренней для обоих, если известно, что никакие три из четырех конечных точек не являются коллинеарными. Мы можем обеспечить выполнение этого условия неколлинеарности с помощью явной проверки; см. код 1.7.

```
bool IntersectProp( tPointi a, tPointi b, tPointi c, tPointi d)

   /* Исключите случаи импичмента. */
   if ( Collinear(a,b,c)
        Collinear(a,b,d)
        Collinear(c,d,a)     ||
        Collinear(c,d,b)     ||
                             ||
        return FALSE;


     return
            Xor( Left(a,b,c), Left(a,b,d) )
         && Xor( Left(c,d,a), Left(c,d,b) );

/* Исключительное или: T, если истинен ровно один аргумент. */
bool      Xor( bool x, bool y )

   /* Аргументы отрицаются, чтобы убедиться, что i6af они являются значениями 0/1. */
   вернуть !x       !y;
```

Code17 IntersectProp.

В этом коде есть досадная избыточность, поскольку четыре соответствующие области треугольника вычисляются по два раза. Эту избыточность можно устранить, вычисляя площади и сохраняя их в локальных переменных, или разработав другие примитивы, которые лучше подходят для решения этой задачи. Я бы возразил против хранения областей, поскольку тогда код не будет прозрачным. Но может оказаться, что код может быть спроектирован вокруг других примитивов более естественно. Оказывается, для триангуляции можно полностью удалить первое условие if, хотя тогда процедура больше не будет вычислять правильное пересечение (равно как неправильное). Этот вопрос рассматривается в упражнении l.6.4[2]. Я предпочитаю пожертвовать эффективностью ради ясности и оставить I nC er s ec CProp как есть, поскольку полезно смотреть за пределы непосредственной задачи программирования на возможные другие применения. В данном случае I nCer sec CProp - это именно та функция, которая необходима для вычисления явной видимости (раздел 1.1.2).

Здесь есть одна тонкость: Может возникнуть соблазн реализовать принцип exclusive-or, требуя, чтобы произведения соответствующих областей были строго отрицательными, гарантируя тем самым, что они имеют противоположный знак и ни одна из них не равна нулю:

```
                Area2(a,b,c)'Area2(a,b,d)< 0
    && Area2(c,d,a)'Area2(c,d,b)< 0;
```

Слабость этой формулировки в том, что произведение областей может привести к целочисленному переполнению слова! Таким образом, хитроумный трюк с экономией нескольких строк может скрыть губительную ошибку. Этой проблемы переполнения можно избежать, если Ar ea2 будет возвращать не истинную площадь, а + 1, 0 или - l (Sedgewick 1992, p. 350). Пока я предпочитаю возвращать площадь, поскольку это полезно в других контекстах - например, для вычисления площади многоугольника! В главе 4 мы пересмотрим это решение, когда будем обсуждать переполнение как общую проблему (код 4.23).

Неправильный перекресток

Наконец, мы должны разобраться с "особым случаем" неправильного пересечения двух отрезков, поскольку Лемма 1.5.1 требует, чтобы пересечение было абсолютно пустым, чтобы отрезок был диагональю. Неправильное пересечение возникает именно тогда, когда конечная точка одного отрезка (скажем, с) лежит где-то на другом (замкнутом) отрезке *ab.* См. рисунок 1.24(а).

Это может произойти только в том случае, если *a, b,* с коллинеарны. Но коллинеарность не является достаточным условием

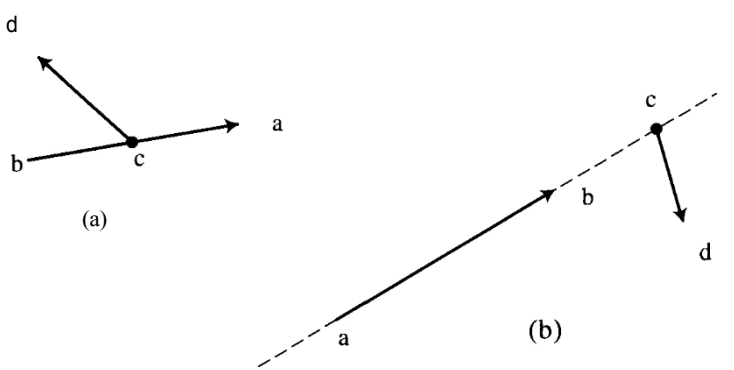


РИСУНОК 1.24 Неправильное пересечение двух отрезков (а); недостаточная коллинеарность (b).

для пересечения, как ясно из рисунка l.24(b). Нам нужно решить, находится ли с *между a* и *b.*

Взаимосвязь
Мы хотели бы вычислить этот предикат "между", не прибегая к склонениям, которые потребуют особых случаев. Поскольку мы будем проверять промежуточность с только тогда, когда узнаем, что она лежит на прямой, содержащей *ab,* мы можем воспользоваться этим знанием. Если *ab* не вертикальна, то с лежит на *ab*, если координата *x* с попадает в интервал, определяемый координатами *x a* и *b.* Если *ab* вертикальна, то аналогичная проверка по координатам у определяет промежуточность. См. код 1.8.

```
bool      Between( tPointi a, tPointi b, tPointi c )

    tPointi ba, ca;

    Если ( ! Collinear( a, b, c ) )
        возвращается FALSE;


   /* Если ab не вертикальна, проверьте промежуточность по x; el.se по y. */
   i€ ( a[X] != b[X] )
       return ((a[X] <= c[X]) && (c[X]<= b[X]))
                ((a[xl '= c[xl) && (c[xl '= b[xl)):
   else
       return ((a[Y]<= c[Y]) && (c[Y]<= b[Y]))
                ((a[Yl '= c[Y)) && c[Yl°= b[Yl));
```

Codel.8 Between.

1.5.5. Код перекрестка

Наконец, мы можем представить код для вычисления пересечения отрезков. Два отрезка пересекаются, если они пересекаются правильно или одна конечная точка одного отрезка лежит между двумя конечными точками другого отрезка. Поэтому проверка на неправильное пересечение осуществляется четырьмя вызовами Be tween; см. код 1.9. В упражнении 1.6.4[3] предлагается проанализировать неэффективность этой процедуры.

1.6. TR I A NG U L A TI ON: IM PL E ME NT A TI ON

1.6.1. Диагонали, внутренние или внешние

Разработав код пересечения сегментов, мы почти готовы к написанию кода для триангуляции многоугольника. Наша первая цель - найти диагональ многоугольника.

```
bool      Intersect( tPointi a, tPointi b, tPointi c, tPointi d )

  if            IntersectProp( a, b, c, d ) )
    возвращают TRUE;
         else if ( Between( a, b, c )
                   Between( a, b, d )
                   Между( c, d, a ) j
                     Между( c, d, b )

    возвращают TRUE;
  else       верните FALSE;
```

Codel.9 Intersect.

Напомним, что в лемме 1.5.1 диагонали характеризовались двумя условиями: не пересекаться с ребрами многоугольника и быть внутренними. Если пренебречь различием между внутренними и внешними диагоналями, то поиск диагоналей сводится к простому повторному применению InCersec t: Для каждого ребра е многоугольника, не связанного ни с одним из концов потенциальной диагонали *s,* посмотрите, пересекает ли *е* диагональ *s.* Как только пересечение обнаруженоизвестно, что s не является диагональю. Если ни одно ребро не пересекает *s,* то *s* может быть диагональю. Причина, по которой мы не можем сразу прийти к положительному выводу, заключается в том, что возможно, что одно из ребер, инцидентных конечной точке *s*, может быть коллинеарно с *s,* и это не будет обнаружено. Мы эту возможность в ближайшее время. Прямой код для обнаружения пересечений показан в коде 1.10.

```
bool    Diagonalie( tVertex a, tVertex b )

  tVertex c, cl;

  /* Для каждого ребра (c,ct) из P */
  c= вершины; do (
    cl= c->next;
    /* Пропустите ребра, инцидентные и или b */
         если (      ( c != a ) && ( cl != a )
                  && ( c != b ) && ( cl != b )
           && Intersect( a->v, b->v, c->v, cl->v )

        return FALSE; c
    = c->next;
  ) while ( c != vertices ); return
  TRUE;
```

Код1.10 Диагональ.

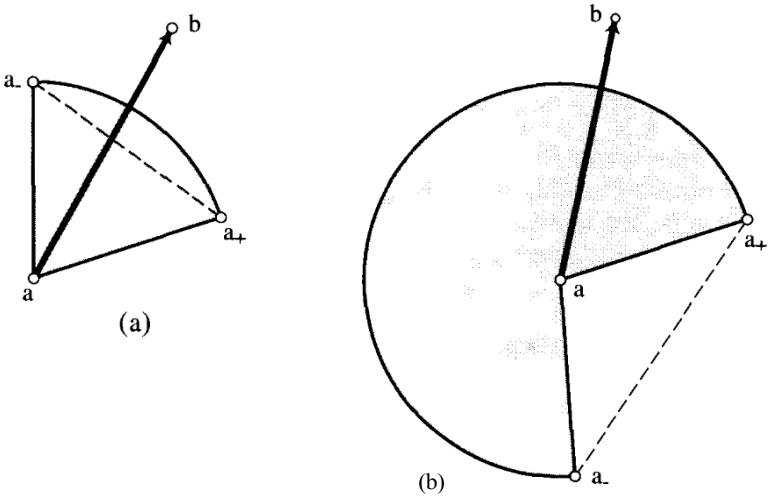


РИСУНОК 1.25 Диагональ s= *ub* находится в конусе, определяемом *as , a, a+:* (a) выпуклая; (b) рефлексивная. В (b) оба *a-* и *a+* лежат справа от *ab.*

1.6.2. InCone

Теперь перейдем ко второму условию леммы 1.5.1: Мы должны отличать внутренние диагонали от внешних; и мы должны позаботиться о ребрах, инцидентных конечным точкам диагоналей. Мы решаем эту задачу с помощью булевой процедуры InCone, которая определяет, лежит ли вектор *B* строго в открытом конусе против часовой стрелки между двумя другими векторами *A* и C. Два последних вектора лежат вдоль двух последовательных ребер многоугольника, а *B* лежит вдоль диагонали. Такой процедуры будет достаточно для определения диагоналей, о чем будет подробно рассказано ниже. Пока же мы сосредоточимся на проектировании **`InCone.`**

Это было бы простой задачей, если бы вершина конуса была выпуклым углом; что может быть рефлекторным, требует либо отдельного случая, либо некоторой сообразительности. Здесь мы остановимся на методе падежей, оставив сообразительность для упражнения 1.6.4[5].

Выпуклый случай на рисунке l.25(a). Фактический результат, выданный кодом, выглядит следующим образом. Из этого рисунка ясно, что s является внутренним в *P iR*, то есть внутренним в конусе, вершиной которого является a, а стороны проходят через *as* и *a+*. Это легко определить с помощью нашей функции Le f t: as должно находиться слева от *ab,* а *a+* должно находиться слева от *ba.* Оба левых угла должны быть строгими для *ab*, чтобы исключить коллинеарное пересечение с границами конуса.

На рис. 1.25(b) показано, что этих условий недостаточно для характеристики внутренних диагоналей, когда *a* рефлексивно: *a-* и *a+* могут быть как слева от внутренней диагонали, так и справа от нее, или одна может быть слева, а другая справа от . Но обратите внимание, что *внешняя сторона* окрестности *a* теперь является конусом, как в выпуклом случае, по той простой причине, что рефлексивная вершина выглядит как выпуклая вершина, если поменять местами внутреннюю и внешнюю. Поэтому в данном случае проще всего характеризовать s как внутреннюю, если она не является внешней: Это не тот случай, когда и *a+* находится слева или на *ab*, и *as* находится слева или на *ba.* Заметим, что на этот раз левая часть

должны быть неправильными, допускающими коллинеарность, поскольку мы отбрасываем диагонали, удовлетворяющие этим условиям.

Наконец, различие между выпуклыми и рефлекторными случаями легко достигается одним обращением к Le f t: a является выпуклым, если *a-* находится слева или на *aaz.* Обратите внимание, что если (as, *a, a)* коллинеарны, то внутренний угол при a равен x, который мы определили как выпуклый (раздел 1.1.2).

Код в Code 1.11 реализует вышеописанные идеи простым способом.

```
bool     InCone( tVertex a, tVertex b )

   CVer kex a0 , al;              /* a0,a,at - последовательные вершины. */

   al= a->next;
   a0= a->prev;

   /* Ifa - выпуклая вершина ... */
   if( Lefton( a->v, al->v, a0->v ) ) return
                  Left( a->v, b->v, a0->v )
               && Left( b->v, a->v, al->v );

   /* Если a - рефлекс: */
        return !(      Lefton( a->v, b->v, al->v ) &&
                     Lefton( b->v, a->v, a0->v ) );
```

Codelll InCone.

Хотя этот тест InCone прост, существует множество возможностей его неправильной реализации. Обратите внимание, что вся функция состоит из пяти вычислений площади с подписью, что иллюстрирует полезность этого вычисления.

1.6.3. Диагональ

Теперь мы разработали код для определения того, является ли a *b* диагональю: если Diagonal i e (a , b), InCone (a , b) и InCone (b , a) истинны. Вызовы I nCone служат как для того, чтобы убедиться, что *at* является внутренним, так и для того, чтобы охватить ребра, связанные с конечными точками, которые не рассматриваются в Di agona1i e. Казалось бы, больше нечего сказать на эту тему, но на самом деле есть выбор, как упорядочить вызовы функций. Как только вопрос , ответ находится сразу: I nConeS должна быть первой, потому что каждая из них - это вычисления с постоянным временем, выполняемые в окрестностях a и *b* без учета остальной части многоугольника, тогда как Di agona l i e включает цикл по всем n граням многоугольника. Если любой из вызовов I nCone возвращает **FALSE,** то (потенциально) дорогостоящая проверка Diagonal i e не выполняется. См. код 1.12.

```
bool      Диагональ( tVertex a, tVertex b )

   return InCone( a, b ) && InCone( b, a ) && Diagonalie( a, b );
```

Код1.12 Диагональ.

1.6.4. Упражнения

1. *Целочисленное переполнение.* На машине, которая ограничивает i nts до m2[n], насколько большими могут быть координаты
   a, b и c, чтобы избежать целочисленного переполнения при вычислении области 2 (код 1.5)?
2. Int er s ec t Prop. Подробно опишите, что именно вычисляет Int ers ec t Prop (код 1.7), если удалить условие if. Укажите, что после этого удаления Int ers ec C (код 1.9) по-прежнему работает правильно.
3. *Неэффективность в* I nder s ec t. Проследите (вручную) I nder s ec t (код 1.9) и определите наибольшее количество вызовов Ar e a2 (код 1.5), которое он может вызвать. Разработайте новую версию, позволяет избежать дублирования вызовов.
4. *Сохранение информации о пересечениях.* Разработайте схему, позволяющую не проверять одни и те же два отрезка на пересечение дважды. Проанализируйте временную и пространственную сложность нового алгоритма.
5. *Улучшение* InCone (Энди Мирзиан). Докажите, что *ab* находится в конусе в точке *a*, если ложно не более одного из этих трех Le f ts: Left(a, *a+ , b),* Left(a, *b, a),* Left(a, *a , a+).*
6. *Улучшение* D i agona l. Докажите, что любой из двух вызовов I nCone в D i agona l может быть удален без изменения результата.

1.6.5. Триангуляция путем удаления уха

Теперь мы готовы разработать код для нахождения триангуляции многоугольника. Один из методов - имитировать доказательство теоремы о триангуляции (Теорема 1.2.3): Найдите диагональ, разрежьте многоугольник на две части и выполните рекурсию по каждой из них. Мы увидим, что этот метод приводит к довольно неэффективному коду, и в итоге выберем метод, основанный на теореме Майстерса о двух ушах (Теорема 1.2.7). Но сначала мы проанализируем скорость рекурсивного метода. Мы будем использовать так называемую нотацию big-0, которая, как мы предполагаем, знакома читателю.[2']

Алгоритм, основанный на диагоналях

Метод, предложенный в Теореме 1.2.3, является $0(n^4)$ алгоритмом: Имеется (2) - $0(n^2)$ кандидатов в диагонали, и проверка каждого из них на диагональность стоит $0(n)$. Повторение этого $0(n3)$ вычисления для каждой из n - 3 диагоналей дает $0(n^4)$.

Мы можем ускорить этот процесс в n раз, используя теорему о двух ушах: мы не только знаем, что должна существовать внутренняя диагональ, и знаем, что должна существовать внутренняя диагональ

[2'] *O(f(n))* означает, что константа, умноженная на /(ri), является верхней границей для больших ri; (8 t^)) означает, что константа, умноженная на 8 (-), является нижней границей для бесконечно многих ri; O(/(ri)) означает, что и *O f(n))*, и f2 (/(ri)) выполнены. См., например, Cormen et al. (1990, глава 2), Albertson & Hutchinson (1988, раздел 2.8),
или Rawlins (1992, раздел 1.4).

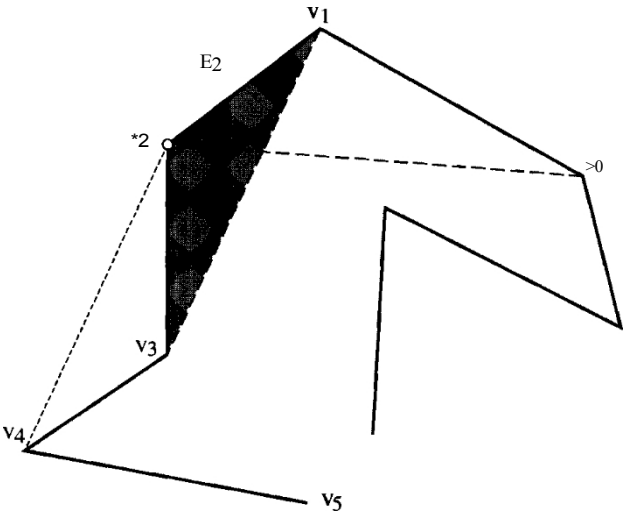


РИСУНОК 1.26 Обрезание уха $z= (-i -'. r). Здесь состояние уха изменяется с TRUE tO FALSE.

которая отделяет ухо. Существует только *0(n)* кандидатов на "диагональ уха": (ui , -+) для *i* -- 0, ... Это также упрощает рекурсию, так как есть только один фрагмент, по которому нужно рекурсировать: другой - ухо, треугольник, который, конечно, уже триангулирован. Таким образом, мы можем достичь наихудшей сложности *0(n')*.

Удаление ушей

Теперь мы улучшим приведенный выше алгоритм до $0(n^2)$. Поскольку один вызов Di agon a l стоит 0(n), для достижения $0(n^2)$ Diagonal может быть вызван только *0 'n)* раз. Ключевая идея, позволяющая улучшить этот алгоритм, заключается в том, что одного уха не сильно меняет многоугольник, в частностине меняет, много ли его вершин

являются потенциальными кончиками ушей. Для этого сначала нужно определить для каждой вершины **i,* ли она потенциальным кончиком уха в том смысле, что u, , nip - диагональ. Для этого уже используется $0(n^2)$, но этот дорогостоящий шаг не нужно повторять.

t (Up, " . 2, 3. -4) - пять последовательных вершин *P,* и предположим, что -2 является кончиком уха, а ухо Ez= (i , z, 3) удалено; см. рис. 1.26. Статус каких вершин как кончиков ушей может измениться? Только r и r3. Рассмотрим, например, вершину 4. Является ли она кончиком уха, зависит от того, является ли u3 диагональю. Удаление +2 оставляет конечные точки отрезка r3r неизменными. Конечно, это удаление не может перекрыть прежнюю линию видимости между этими конечными точками, если они могли видеть друг друга. Возможно, менее очевидно, что если они не могли видеть друг друга, не видят и после удаления . Но, как и в доказательстве леммы 1.2.2, есть только два случая, которые необходимо рассмотреть. Если r3 является внешним, то очевидно, что удаление p2 не может сделать его внутренним. Иначе fi(-3. 4, -5) должна содержать вершину, причем рефлексивную вершину (x на рис. 1.12). Но удаление удаляет только одну вершину, и она выпуклая. Поэтому статус 4 не изменится удаления , как и всех вершин, кроме u и -3, ушные диагонали которых инцидентны удаленной вершине 2.

Следствием этого является то, что после дорогостоящего шага инициализации состояние кончика уха может быть обновлено двумя вызовами Diagonal за итерацию. Это приводит к псевдокоду, показанному в Алгоритме 1.1, для построения триангуляции.

Алгоритм: ТРИАНГУЛЯЦИЯ
Инициализируйте состояние кончика уха для каждой вершины.
while *n*> 3 do
    Найдите кончик уха r2.
    Выведите диагональ r} r3.
    Удалить -2
    Обновите состояние наконечников ушей на сайтах i и r3.

Алгоритм 1.1 Алгоритм **триангуляции**.

Обратите внимание, что мы интерпретируем задачу "триангулировать многоугольник" как "вывести в произвольном порядке диагонали, образующие триангуляцию". Это сделано в первую очередь для удобства изложения. Часто в приложениях, связанных с триангуляцией, требуется более структурированный вывод: Например, может потребоваться информация о смежности треугольников в двойственном графе. Хотя получение более структурированного вывода не является более сложным точки зрения асимптотической сложности времени, оно часто значительно усложняет код. Мы не будем далее рассматривать эти альтернативные варианты вывода триангуляции.

Код триангуляции
Первая задача - инициализировать булевский флаг v->ear, который является частью структуры вершин (код 1.2). Это выполняется одним вызовом Di agona l для каждой вершины. См. раздел Ear Ini C, код 1.13.

```
void     EarInit( void )

   CVer tex    v0 ,   vl ,   v2;          /* три последовательные вершины */

   /* Инициализируйте vl-> ear для всех вершин. */
   vl= вершин;

     v2= vl->next;
     v0= vl->prev;
     vl-near= Diagonal( v0, v2 ); vl =
     vl->next;
   ) while ( vl != vertices );
```

Код 1.13 EarInit.

Основной код Tr iangu l at e состоит из двойного цикла. Внешний цикл удаляет одно ухо за итерацию, останавливаясь, когда n= 3. Внутренний цикл ищет ухо, проверяя

предварительно вычисленный флаг v2 ->ear, где - потенциальный кончик уха. Как только кончик уха найден, статус уха для u; и u3 обновляется вызовом Diagonal, печатается диагональ, представляющая основание уха, и ухо удаляется из полигона. Это удаление выполняется путем переключения указателей next и prey для u и u3. (В этот момент ячейка для -2 может быть освобождена, если она не используется в соседнем приложении). Необходимо соблюдать осторожность, чтобы -z не стал "головой" списка вершин, точкой доступа в кольцевой список. По этой причине головной указатель vertices перемещается на u3. См. код 1.14. Перед анализом времени мы рассмотрим пример
сложность.

```
void     Triangulate( void )

tVerCex v0 ,   vl ,   v2 ,   v3 ,   v4;          /* пять последовательных вершин */
инт      n= nver t i ces:                        /*количество вершин,' уменьшается до 3. */

EarInit()
/* Каждый шаг внешнего цикла удаляет одно ухо. */
wi l e          n> 3 ) (
   /* Внутренний цикл ищет ухо. */
   v2   = ver L i ces ;
   делать  {
      i f   { v2 - >ear ) (
         /* Ухо найдено. Заполнить переменные. */
         v3    = v2 - >nex t ; v4= v3 - >nex t ;
         vl= v2 - >pr ev ; v0   = vl - >pr ev ;

         /* (v1,v3) - диагональ */
         PrintDiagonal( vl, v3 );

         tUpdeeariofdiagonaendpointt      vl-
         >ear= Diagonal( v0, v3 ); v3->ear =
         Diagonal( vl, v4 );

         /* Отрезать ухо v2 */
         vl->next= v3;

         Глагол ices= v3 ;          /* /n case r/te /tead was v2. */

         b2akprev /* выход из внутреннего цикла, - возобновление
         внешнего цикла *7
       1/* конец, если ухо найдено */
         v2= v2->next;
   ) while ( v2 != vertices );
1 /* конец внешнего цикла while */
```

Код 1.14 Триангулируйте.

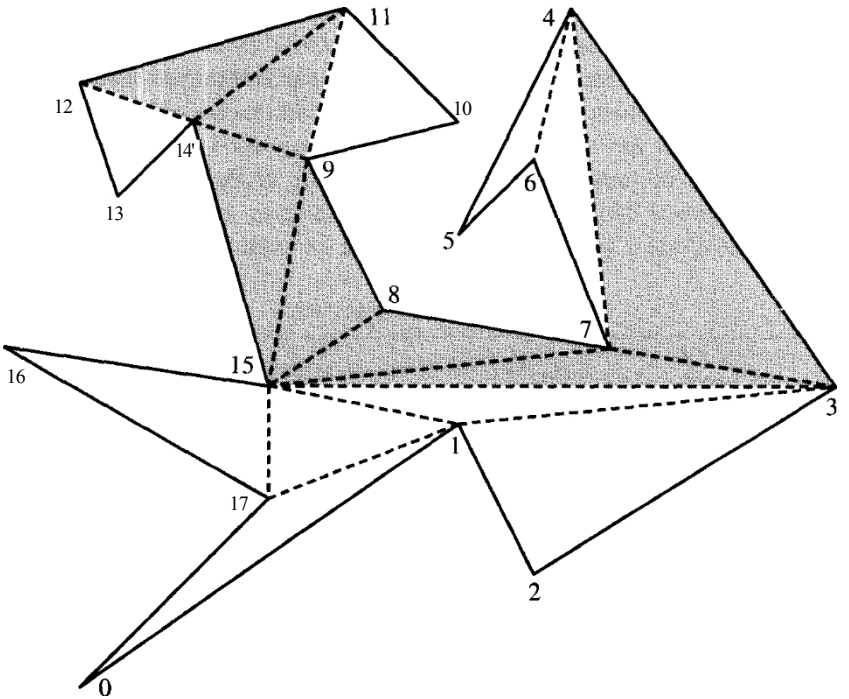


РИСУНОК 1.27 Многоугольник из 18 вершин и триангуляция, полученная программой Tr i angul a te. Темный подполигон - это остаток после вывода 9-й диагонали (15, 3). Координаты вершин приведены в таблице l.l.

1.6.6. Пример

На рис. 1.27 показан многоугольник и триангуляция, созданная простой основной программой (код 1.15). Код для чтения и печати прост и не будет здесь показан.[2]

```
main()

   ReadVertices(); PrintVertices();
   Triangulate();
```

Codell5 main.

Теперь пройдемся по выводу диагоналей для этого примера, представленного в табл. 1.2. u - кончик уха, поэтому первой диагональю будет (17, 1). ut - не уха, поэтому указатель v2 перемещается на z., которая является кончиком, печатая диагональ (1, 3) следующей. Ни u3

2* О том, как получить полный код, читайте в предисловии.

Таблица 1.1. Координаты вершин для многоугольника, изображенного на рисунке 1.27.

i	*(x, y)*	*i*	*(x, y)*
0	(0, 0)	9	(6, 14)
1	(10, 7)	10	(10, 15)
2	(12, 3)	l 1	(7, 10)
3	(20, 8)	12	(0, 16)
4	(13, 17)	13	(1, 13)
5	(10, 12)	14	(3, l5)
6	(12, 14)	15	(5, 8)
7	(14, 9)	16	(-2, 9)
8	(8, 10)	17	(5, 5)

ни 4 не является кончиком уха, поэтому выход на следующую диагональ, (4, 6), происходит только после достижения . Отрезок u3ug коллинеарен с u , поэтому следующее обнаруженное ухо появляется только после i Заштрихованный темным цветом подполигон на рис. 1.27 показывает оставшийся полигон после вывода диагонали (15, 3) (9-й). Еще одна коллинеарность, Ug с (rii i), не позволяет нам стать ухом после вырезания диагонали (15, 9).

1.6.7. Анализ

Теперь мы проанализируем временную сложность алгоритма. Как уже говорилось, ухо Ini t стоит $0\{n^l\}$. Внешний цикл Tr i angu la t e итерируется столько раз, сколько существует диагоналей, n - 3 = *0(n)*. Внутренний цикл поиска ушей также равен *0(n),* потенциально проверяя каждую вершину. Работа внутри внутреннего цикла равна *0{n):* Каждый из двух вызовов Diagonal может, в худшем случае, пройтись по всему многоугольнику, чтобы убедиться, что диагональ не заблокирована. Наивно полагать, что в этом случае временная сложность составит *0(n'), что* не дотягивает до обещанных $0(n^2)$. Более тщательный анализ покажет, что $0\{n^l\}$ - это правильное ограничение для Tri angu1aCe.

Рассмотрим пример на рис. 1.28. удаления u внутренний цикл выполняет поиск в прошлом -i 6 до достижения следующего кончика уха -7 Затем он должен пройти мимо Ug, . iz, прежде чем найдет кончик уха u 3. Этот пример показывает, что действительно внутренний цикл может проделать итерацию f2 (n) раз, прежде чем найдет ухо. Но обратите внимание, что два вызова диагонали *0(n)* внутри цикла вызываются только после того, как ухо найдено - они не вызываются в каждой итерации. Таким образом, хотя поверхностная структура кода предполагает сложность n x n x n= 0(n'), на самом деле это n x (n + n) - $0(n^d)$.

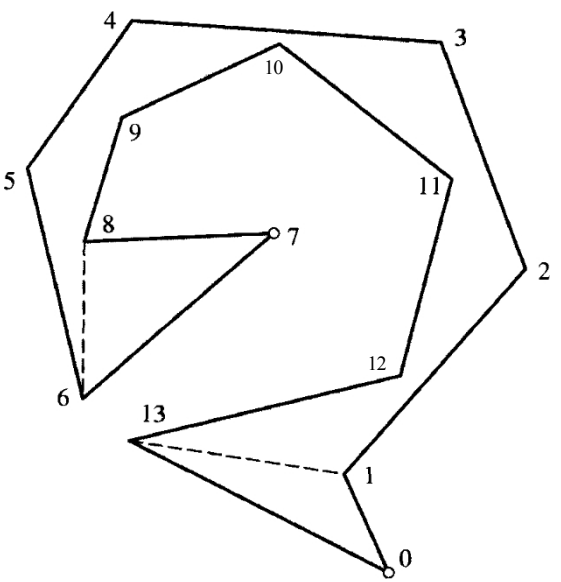
Хотя возможны и дальнейшие небольшие улучшения (упражнение 1.6.8[4]), снижение асимптотической временной сложности ниже квадратичной требует совсем других подходов, которые рассматриваются в следующей главе.

1.6.8. Упражнения

1. *Повторяющиеся проверки пересечений* [программирование]. Tr i angu1ate (код 1.14) часто проверяет пересечения одних и тех же сегментов/сегментов. Измените код так, чтобы можно было определить, сколько ненужных проверок пересечений сегментов/сегментов выполняется. Проверьте это на рисунке 1.27.

Таблица 1.2. Столбцы показывают порядок вывода диагоналей, заданных в виде пар индексов конечных точек.

Заказать	Диагональные индексы	Заказать	Диагональные индексы
1	(17, 1)	10	(3, 7)
2	(1, 3)	11	(11, 14)
3	(4, 6)	12	(I5, 7)
4	(4, 7)	13	(15, 8)
5	(9, 11)	14	(15, 9)
6	(12, 14)	15	(9, 14)
7	(I5, 17)		
8	(15, 1)		
9	(15, 3)		



ПИГУРА 1.28 Пример, который заставляет цикл внутреннего уха интенсивно искать следующее ухо.

2. *Выпуклые многоугольники* [легко]. Проанализируйте производительность Tr i angu laC e при работе с выпуклым многоугольником.

3. *Спираль.* Продолжите анализ рисунка 1.28: Продолжает ли Tr i angu1a t e обходить границу в поисках уха? , если многоугольник имеет *n* вершин, сколько полных обходов границы совершит указатель v2 до завершения?

4. *Ear fisr* (программирование]. Внутреннего цикла поиска в Tr i angu l at e можно избежать, соединив кончики ушей в их собственный (круговой) список, связав те вершины r, для которых v- >ear== TRUE, с указателями next ear и pr eve ar в структуре вершин. Тогда ухо для следующей итерации может быть найдено путем перехода к следующему уху в этом списке за тем которое только что было обрезано. Реализуйте это улучшение и посмотрите, заметно ли его ускорение на примере (возможно, похожем на рис. 1.28).

5. *Центр тяжести.* Разработайте алгоритм для вычисления центра тяжести многоугольника, предполагая, что он вырезан из материала с однородной плотностью. Центр тяжести - точка, которую можно рассматривать как вектор. Центр тяжести треугольника находится в его *центроиде*, координаты которого совпадают со средним значением координат вершин треугольника. Центр тяжести *z 'S)* любого множества *S*, которое является дизъюнктивным объединением множеств *A* и *B*, представляет взвешенную сумму центров тяжести этих двух частей. Пусть *w{S) -- w A) + w B)* - вес *S.* Тогда

$$\gamma(S) = \frac{w(A) \quad A) + w[B)\ [B)}{w\{S)}$$

Здесь вес каждого треугольника - это его площадь в предположении равномерной плотности.