МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ БУДІВНИЦТВА І АРХІТЕКТУРИ

Є.В. Бородавка

А.А. Бугров

Б.Ю. Волох

ГРАФІЧНІ ІНФОРМАЦІЙНІ ТЕХНОЛОГІЇ ТА ОБЧИСЛЮВАЛЬНА ГЕОМЕТРІЯ

Підручник для студентів технічних закладів вищої освіти, які навчаються за спеціальністю F6 – Інформаційні системи та технології

Київ 2026

УДК 004.92

ББК 32.973

Рецензенти: *А. О. Білощицький*, доктор технічних наук, професор, проректор з науки та інновацій Astana IT University (Казахстан)
Т. А. Гончаренко, доктор технічних наук, професор, завідувачка кафедри інформаційних технологій Київського національного університету будівництва і архітектури
С. В. Палій, кандидат технічних наук, доцент, доцент кафедри інформаційних систем та технологій

Рекомендовано Вченою радою Київського національного університету будівництва і архітектури як навчальний підручник для студентів технічних закладів вищої освіти, які навчаються за спеціальністю F6 — Інформаційні системи та технології.

Затверджено на засіданні Вченої ради Київського національного університету будівництва і архітектури, протокол №___ від _____ 2026 р.

Бородавка Є.В.

К 80 Графічні інформаційні технології та обчислювальна геометрія. Підручник / Є.В. Бородавка, А.А. Бугров, Б.Ю. Волох. — К.: Компринт, 2026. — 175 с.:іл.

В підручнику розглянуто базові двовимірні та тривимірні алгоритми обчислювальної геометрії, що використовуються в системах комп'ютерного проєктування та моделювання. Також розглянуто деякі алгоритми обробки зображень, що можуть бути корисними в розробці систем генерації зображень з використанням штучного інтелекту.

УДК 004.92

ББК 32.973

ЗМІСТ

Вступ

Цей підручник створено для студентів. Що навчаються в магістратурі за спеціальністю F6 «Інформаційні системи та технології», але він може бути корисним для всіх студентів галузі знань F «Інформаційні технології».

В підручнику розглянуті вибіркові теми з обчислювальної геометрії, комп'ютерної графіки, обробки зображень та комп'ютерного зору.

Підручник складається з 15 розділів, кожен з яких відповідає конкретній темі лекції, що викладаються студентам в межах однойменного курсу.

Головне завдання, яке стояло перед авторами — подати в доступній формі як прості класичні алгоритми, так і більш складні алгоритми і методи, що вимагають просунутих знань з математики в цілому, та геометрії зокрема.

Переважна більшість прикладів, а також ілюстрацій до них розроблена і виконана авторами особисто, проте деякі з них запозичені з класичних підручників та науково-популярних сайтів, які перелічені в списку використаних джерел.

Сподіваємося, що цей підручник стане корисним не тільки студентам КНУБА, а й студентам інших закладів вищої освіти України.

1. ЗАДАЧІ З ТОЧКАМИ ТА ПРЯМИМИ

Алгоритми, що наведені нижче, прості з математичної точки зору і базуються на елементарних постулатах аналітичної геометрії. Але ці алгоритми складають основу для більш складних алгоритмів, що забезпечують відтворення цікавих зображень.

1.1. РІВНЯННЯ ПРЯМОЇ, ЩО ЗАДАНА ДВОМА ТОЧКАМИ

Нехай координати точок P_1 та P_2 дорівнюють (x_1, y_1, w_1) та (x_2, y_2, w_2) відповідно. Точка з координатами (x, y, w) буде колінеарна точкам P_1 та P_2, якщо її координати лінійно залежні від координат цих точок. Це означає, що визначник матриці, стовпці якого задають координати трьох точок, що розглядаються, повинен дорівнювати 0.

$$\det\begin{bmatrix} x & x_1 & x_2 \\ y & y_1 & y_2 \\ w & w_1 & w_2 \end{bmatrix} = 0.$$

Це рівняння задає пряму, що проходить крізь дві точки P_1 та P_2. Розкривши визначник — отримаємо рівняння:

$$x \cdot (y_1 \cdot w_2 - w_1 \cdot y_2) + y \cdot (w_1 \cdot x_2 - x_1 \cdot w_2) + w \cdot (x_1 \cdot y_2 - y_1 \cdot x_2) = 0.$$

В подальшому нам часто доведеться застосовувати цей спосіб подання прямої як на площині, так і в просторі.

Загальне рівняння прямої на площині подається у наступному вигляді:

$$A \cdot x + B \cdot y + C \cdot w = 0.$$

Таким чином ми можемо визначити кожен з коефіцієнтів за наступними формулами:

$$A = (y_1 \cdot w_2 - w_1 \cdot y_2)$$
$$B = (w_1 \cdot x_2 - x_1 \cdot w_2)$$
$$C = (x_1 \cdot y_2 - y_1 \cdot x_2).$$

Після перетворення з однорідних координат в Евклідові (w=1) ми отримаємо наступні вирази:

$$A = (y_1 - y_2)$$
$$B = (x_2 - x_1)$$
$$C = (x_1 \cdot y_2 - y_1 \cdot x_2).$$

Для двох паралельних прямих справджується наступна умова:

$$\frac{A_1}{A_2} = \frac{B_1}{B_2} \neq \frac{C_1}{C_2}$$

1.2. Взаємне розташування прямих і точок

Вирішення багатьох задач, пов'язаних із визначенням видимості об'єктів зображення, а також задач відсікання, потребує визначення взаємного розташування прямих і точок.

Домовленість. Всі відрізки прямих при відсутності спеціальних приміток орієнтовані: у негоризонтальних відрізків стрілки спрямовані знизу вгору, у горизонтальних – зліва направо. Іншими словами, якщо (x_1, y_1, w_1) — координати кінцевої точки, що поставлена першою, (x_2, y_2, w_2) — координати кінцевої точки, що поставлена другою, і якщо значення y_1/w_1 не дорівнює значенню y_2/w_2, то значення y_1/w_1 менше значення y_2/w_2. У випадку рівності (тобто горизонтальності відрізка) значення x_1/w_1 менше за значення x_2/w_2 (рис. 1.1).

Визначення. Будемо вважати, що точка P міститься праворуч від відрізка прямої L, якщо вона розміщена праворуч від спостерігача, що переміщується вздовж цієї прямої від першої кінцевої точки до другої.

Визначення. Будемо вважати, що точка P затуляє пряму L (чи відрізок прямої L'), якщо горизонтальна пряма, що проходить крізь точку P, перетинає пряму L (чи відрізок прямої L') у точці, значення координати x якої менше значення відповідної координати точки P.

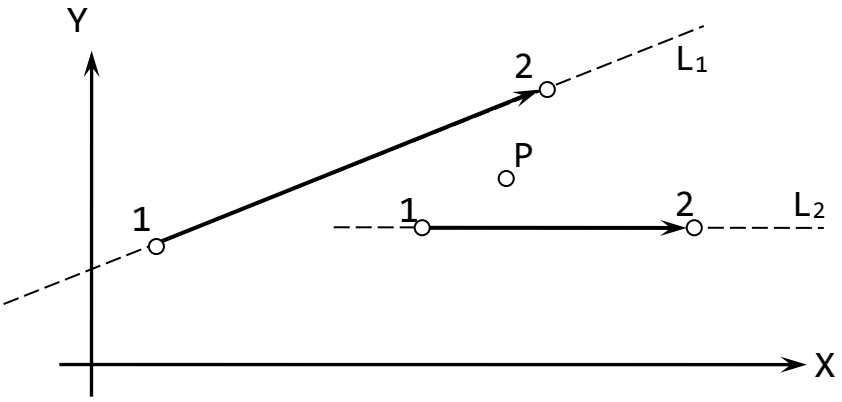


Рисунок 1.1. Взаємне розташування відрізків прямих і точки

Розшифровка позначень, які використані в домовленості та визначеннях: точка P розташована праворуч від прямої L_1 і ліворуч від прямої L_2; крім того, точка P затуляє пряму L_1.

Відношення «затуляє» не виконується, якщо точка P лежить на прямій L, і не визначено чи горизонтальна пряма L. Для відрізка прямої це відношення не визначене в одному з трьох наступних випадків:

- точка P розміщується вище верхньої кінцевої точки відрізка;
- точка P розміщується нижче за нижню кінцеву точку відрізка прямої;
- точка P лежить на горизонтальному відрізку прямої.

Якщо орієнтація відрізка прямої відповідає домовленості і відношення «затуляє» визначено, то воно стає еквівалентним відношенню «міститься праворуч від».

Твердження. Нехай X, Y і W — координати точки P, а (x_1, y_1, w_1) та (x_2, y_2, w_2) — кінцеві точки відрізка прямої L. Якщо значення W, w_1та w_2 — додатні, то точка P знаходиться праворуч відносно прямої, що визначається відрізком прямої L, у тому, і тільки у тому випадку, якщо виконується нерівність:

$$X(y_1w_2 - w_1y_2) + Y(w_1x_2 - x_1w_2) + W(x_1y_2 - y_1x_2) < 0. \qquad (1.1)$$

При переході від однорідних координат до евклідових твердження зберігає істинність, якщо прийняти W=w_1=w_2=1. Зокрема, точка P міститься праворуч від прямої, якщо виконується нерівність:

$$X(y_1 - y_2) + Y(x_2 - x_1) + (x_1y_2 - y_1x_2) < 0.$$

Фактично ми підставляємо координати точки P в рівняння прямої і визначаємо знак отриманого значення. Якщо воно менше 0, то точка знаходиться праворуч, якщо більше — ліворуч, якщо 0 — точка на прямій.

1.3. Перетин відрізків прямих

Якщо два відрізка прямих задані точками P_1, P_2, P_3 та P_4, то вони перетинаються у тому і тільки у тому випадку, коли при підстановці у рівняння прямої, що з'єднує точки P_3 та P_4, координати точок P_1 та P_2 результати мають різні знаки. Аналогічна умова діє і для випадку, коли точки P_1, P_2 та P_3, P_4 обмінюються ролями. Тобто, необхідно визначити знаки наступних чотирьох величин:

$$S_1 = x_1(y_3w_4 - w_3y_4) + y_1(w_3x_4 - x_3w_4) + w_1(x_3y_4 - y_3x_4);$$
$$S_2 = x_2(y_3w_4 - w_3y_4) + y_2(w_3x_4 - x_3w_4) + w_2(x_3y_4 - y_3x_4);$$
$$S_3 = x_3(y_1w_2 - w_1y_2) + y_3(w_1x_2 - x_1w_2) + w_3(x_1y_2 - y_1x_2);$$
$$S_4 = x_4(y_1w_2 - w_1y_2) + y_4(w_1x_2 - x_1w_2) + w_4(x_1y_2 - y_1x_2).$$

Умова перетину вимагає, щоб S_1 та S_2 мали різні знаки, так само як і S_3 та S_4.

Якщо умови виконані, то координати точки перетину можна визначити, розв'язавши пару лінійних рівнянь:

$$x(y_1w_2 - w_1y_2) + y(w_1x_2 - x_1w_2) + w(x_1y_2 - y_1x_2) = 0;$$
$$x(y_3w_4 - w_3y_4) + y(w_3x_4 - x_3w_4) + w(x_3y_4 - y_3x_4) = 0. \qquad (1.2)$$

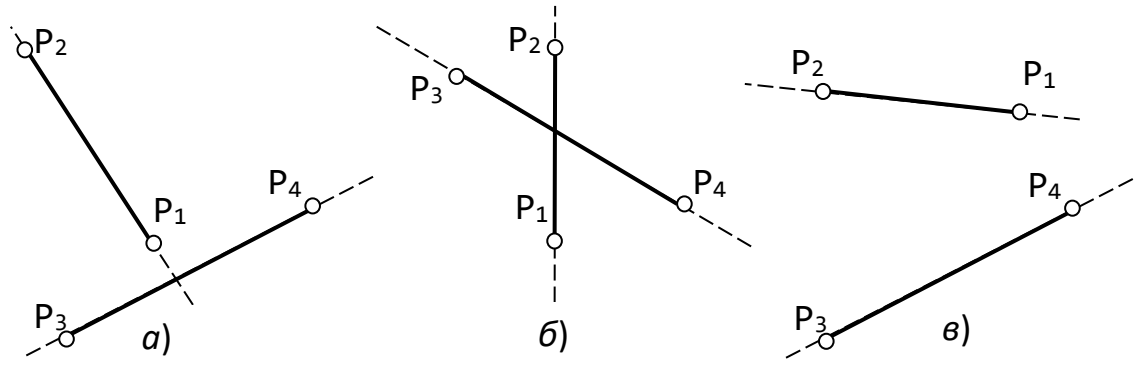


Рис. 1.2. Ілюстрація до співвідношень знаків величин, які визначаються рівняннями: *а*) $S_1S_2>0$ і $S_3S_4<0$; *б*) $S_1S_2<0$ і $S_3S_4<0$; *в*) $S_1S_2>0$ і $S_3S_4>0$

Умову перетину можна записати у компактному вигляді:

$$S_1 = \det(P_1, P_3, P_4)\,;\; S_2 = \det(P_2, P_3, P_4)\,;\; S_1 S_2 < 0;$$
$$S_3 = \det(P_3, P_1, P_2)\,;\; S_4 = \det(P_4, P_1, P_2)\,;\; S_3 S_4 < 0.$$

Аналогічно рівняння (1.2) приймають вигляд:

$$\det(P, P_1, P_2) = 0;\; \det(P, P_3, P_4) = 0.$$

Якщо одна чи декілька величин S_i мають нульові значення, то має місце вироджений випадок. Наприклад, якщо $S_1=0$, то це означає, що точка P_1 знаходиться на прямій між точками P_3 та P_4, у випадку, якщо $S_3S_4<0$, або на цій же прямій але вище (лівіше) або нижче (правіше) відрізка P_3P_4. Де конкретно лежить точка перетину, залежить від знаків величин S_3 та S_4.

1.4. Лівий і правий повороти двох відрізків

Щоб мати уявлення про поняття лівого і правого повороту двох відрізків, розглянемо приклад (рис. 1.3).

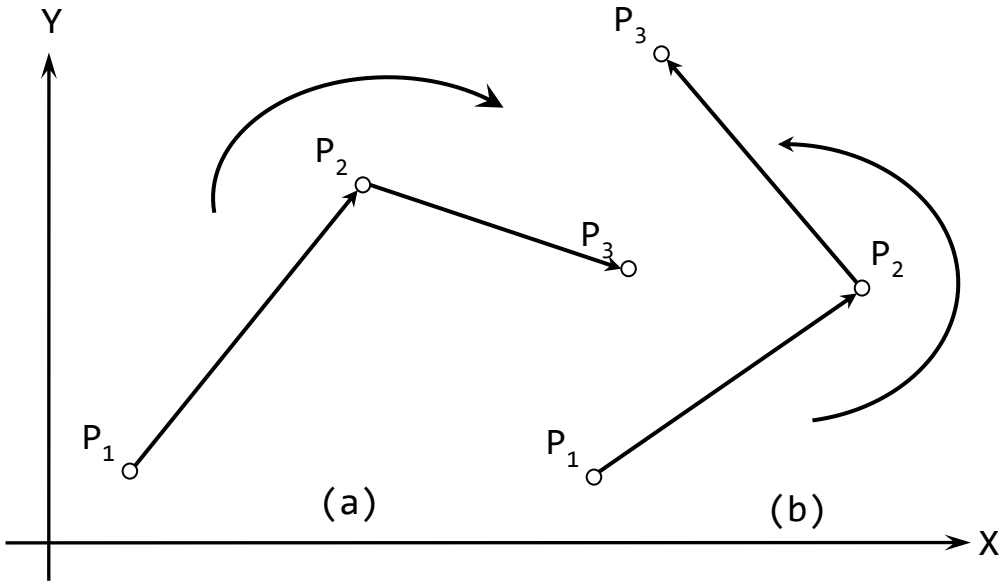


Рис. 1.3. (a) — правий поворот, (b) — лівий поворот

Обидві ламані на рис. 1.3 мають по три точки — P_1, P_2, P_3 та по два відрізки — P_1P_2 і P_2P_3. Точка P_2 є спільною для обох відрізків. На рис. 1.3 (a) відрізок P_2P_3 утворює правий поворот в точці P_2, а на рис. 1.3 (b) —лівий. Візуально не складно визначити який поворот є лівим, а який — правим. Але нам необхідно мати математичний апарат для визначення цього факту. Щоб його знайти, спочатку розглянемо задачу взаємного розташування двох точок.

Задача. Задані дві точки $P_1(x_1, y_1)$ і $P_2(x_2, y_2)$. Визначити, чи знаходиться точка P_1 у напрямку за годинниковою стрілкою від точки P_2 відносно початку системи координат (рис. 1.4).

Наївний спосіб розв'язання цієї задачі зводиться до порівняння кутів, що утворюють прямі проведені через початок системи координат і відповідні точки.

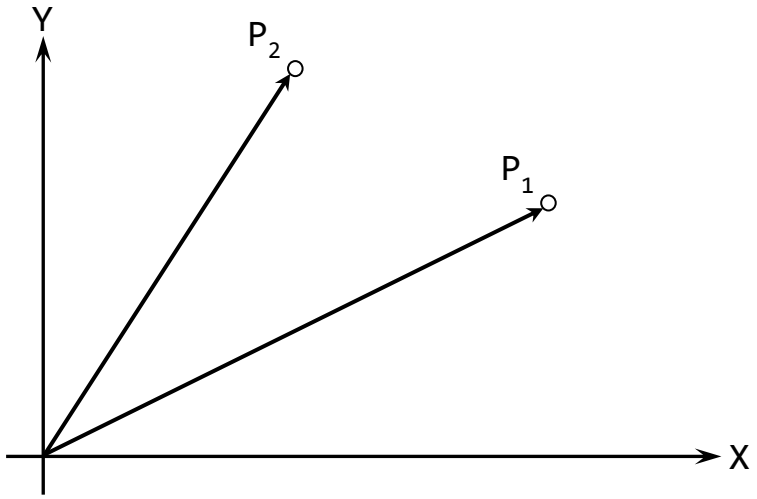


Рис. 1.4. Взаємне розташування двох точок

Значно більш простим і швидким способом розв'язання цієї задачі є використання векторного добутку векторів P_1 і P_2. Векторним добутком двох векторів є третій вектор, що має напрям перпендикулярний до площини, в якій знаходяться вектори, що множаться. В нашому випадку вектор-результат буде мати нульові координати X та Y, а координата Z буде рахуватися по формулі:

$$x_1 \cdot y_2 - x_2 \cdot y_1.$$

Якщо значення координати Z додатне, значить точка P_1 лежить у напрямку за годинниковою стрілкою від точки P_2 відносно початку координат — так, як це зображено на рис. 1.4. В протилежному випадку Z буде від'ємне. Якщо ж Z=0, то це означає, що вектори P_1 і P_2 колінеарні (лежать на одній прямій).

Тепер повернемося до задачі про лівий і правий повороти двох відрізків. У нас є три точки — $P_1(x_1, y_1)$, $P_2(x_2, y_2)$, $P_3(x_3, y_3)$ і два відрізки — P_1P_2 і P_2P_3. Побудуємо два вектори: $P_1P_2=(x_2-x_1, y_2-y_1)$ і $P_1P_3=(x_3-x_1, y_3-y_1)$ (рис. 1.5).

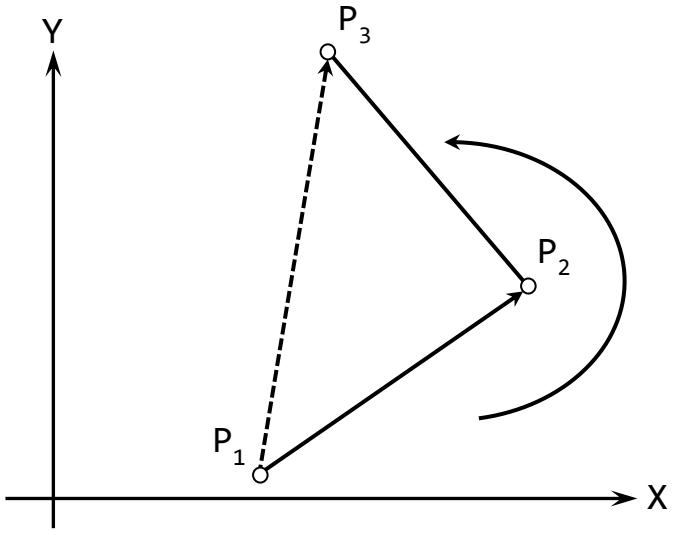


Рис. 1.5. Визначення лівого повороту за векторами

Таким чином задача про визначення повороту звелась до задачі визначення взаємного розташування точок P_2 і P_3. Для цього знаходимо векторний добуток $P_1P_2 \times P_1P_3=(x_2-x_1)*(y_3-y_1)-(x_3-x_1)*(y_2-y_1)$. У випадку додатного значення, точка P_2 лежить за годинниковою стрілкою від точки P_3 відносно початку в точці P_1. Тобто відрізок P_2P_3 робить лівий поворот відносно відрізка P_1P_2. В протилежному випадку, коли значення векторного добутку від'ємне, відрізок P_2P_3 робить правий поворот. У випадку нульового значення обидва відрізки лежать на одній прямій. Цей метод визначення правого і лівого поворотів буде використовуватися ще в багатьох алгоритмах, які ми будемо розглядати далі.

1.5. Тінь відрізка

При вирішенні ряду задач, зв'язаних з розділенням видимих та невидимих об'єктів зображення, необхідно оцінювати взаємне розташування відрізків прямих.

Визначення. Будемо вважати, що відрізок прямої a затіняє чи загороджує (затуляє) відрізок b, якщо він не перетинає відрізок b і, що найменше, одна з його точок загороджує відрізок b.

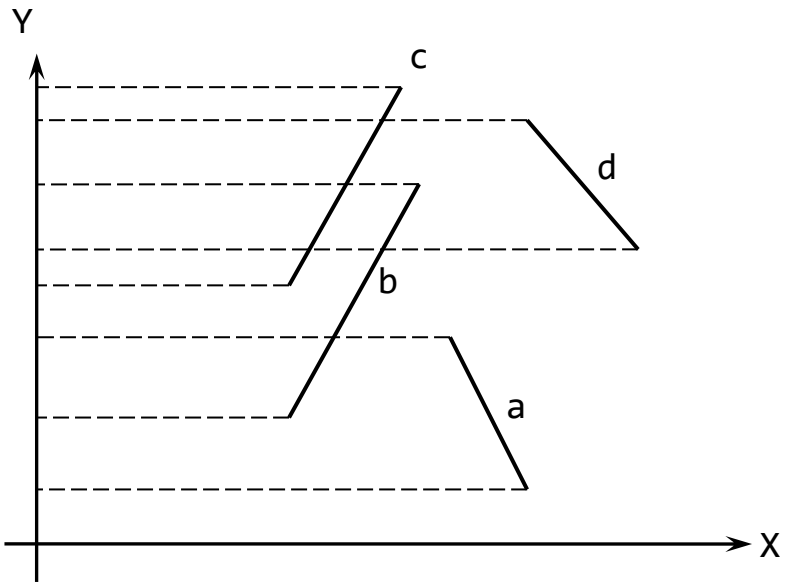


Рис. 1.6. Ілюстрація до затінення відрізків прямих

На рис 1.6. відрізок a затіняє відрізок b, відрізок b затіняє відрізок c, але відрізок a не затіняє відрізок c, відрізок d затіняє відрізки b та c, але не затіняє відрізок a.

Твердження. Відрізок прямої a затіняє відрізок прямої b у тому і тільки у тому випадку, якщо виконуються наступні умови:

$$y_1^a \le y_2^b;\ y_1^b \le y_2^a;$$

$$x_1^a\left(y_1^b - y_2^b\right) - y_1^a\left(x_1^b - x_2^b\right) + x_1^b y_2^b - y_1^b x_2^b < 0.$$

1.6. Відстань від точки до прямої

Нехай задані точка C(x_c,y_c)та пряма AB, де A(x_a,y_a), B(x_b,y_b) та потрібно знайти відстань від точки до прямої (рис. 1.7).

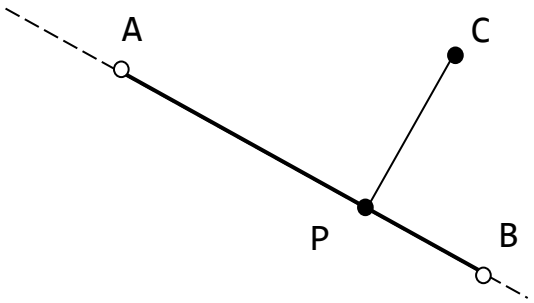


Рис. 1.7. Ілюстрація до задачі знаходження відстані від точки до прямої

Алгоритм розв'язання задачі знаходження відстані від точки до прямої наступний.

1. Знаходимо довжину відрізка AB:

$$l = \sqrt{(x_a - x_b)^2 + (y_a - y_b)^2}.$$

2. Опустимо з точки C перпендикуляр на AB. Точку P перетину цього перпендикуляра з прямою можна подати параметрично:

$$P = A + r(B - A),$$

де

$$r = \frac{(y_a - y_c)(y_a - y_b) - (x_a - x_c)(x_b - x_a)}{l^2}.$$

3. Положення точки C на цьому перпендикулярі буде задаватися параметром s, де s<0 означає, що C знаходиться ліворуч від AB, s>0, що C – праворуч від AB і s=0 означає, що C лежить на AB. Для обчислення s скористаємося наступною формулою:

$$s = \frac{(y_a - y_c)(x_b - x_a) - (x_a - x_c)(y_b - y_a)}{l^2},$$

тоді шукана відстань:

$$PC = sl.$$

Запитання для самоконтролю

1. Запишіть рівняння прямої у вигляді матриці.
2. Дайте визначення точки, що лежить праворуч від відрізка прямої.
3. Яка умова перетину двох відрізків?
4. Які можливі варіанти взаємного розташування двох відрізків?
5. Як визначити лівий чи правий поворот робить один відрізок відносно іншого?

2. ЗАДАЧІ З ПРЯМОКУТНИКАМИ

В цьому розділі будуть розглянуті деякі прості алгоритми для обробки прямокутників.

2.1. ПЕРЕТИН ПРЯМОКУТНИКІВ

Два ізотетичних прямокутника перетинаються тоді і тільки тоді, коли у них є хоча б одна спільна точка. Прямокутники називаються ізотетичними якщо їх сторони паралельні, тобто обидва прямокутники повернуті на однаковий кут відносно осі OX.

Найпростішим є одновимірний випадок, де «прямокутники» — це інтервали на прямій лінії. Оскільки d-вимірний прямокутник є декартовим добутком d інтервалів, кожен з яких визначений на відповідній координатній осі, то два d-вимірних прямокутника перетинаються тоді і тільки тоді , коли перетинаються їх проєкції на вісь j=1…d.

Нехай задані два інтервали $R'=[x'_1, x'_2]$ і $R''=[x''_1, x''_2]$. Умова $R' \cap R'' \neq \varnothing$ еквівалентна одній з наступних взаємовиключних умов:

$$x'_1 \leq x''_1 \leq x'_2;\ x''_1 \leq x'_1 \leq x''_2. \qquad (2.1)$$

Чотири можливі ситуації взаємного розташування кінців інтервалів, що відповідають умові $R' \cap R'' \neq \varnothing$ фактично охоплюється співвідношеннями (2.1). Тому для перевірки перетину R' і R'' достатньо перевірити факт потрапляння або лівого кінця R' всередину R'', або лівого кінця R'' всередину R'.

2.2. ЗОВНІШНІЙ КОНТУР ОБ'ЄДНАННЯ ПРЯМОКУТНИКІВ

В загальному випадку, контур об'єднання прямокутників може мати $O(N^2)$ ребер. Але для підмножини контуру, що називається *зовнішнім*, це не так.

Зовнішнім контуром об'єднання ізотетичних прямокутників F називається кордон між F та нескінченною областю площини.

Покажемо тепер, що зовнішній контур має O(N) ребер. Для цього розглянемо дві множини зовнішнього контуру, перша з яких називається *нетривіальним* контуром.

Нетривіальним контуром об'єднання ізотетичних прямокутників F називається множина таких контурних циклів, кожен з яких містить щонайменше одну вершину будь-якого з заданих прямокутників.

Приклади цих трьох об'єктів — контуру, нетривіального контуру і зовнішнього контуру подані на рис. 2.1.

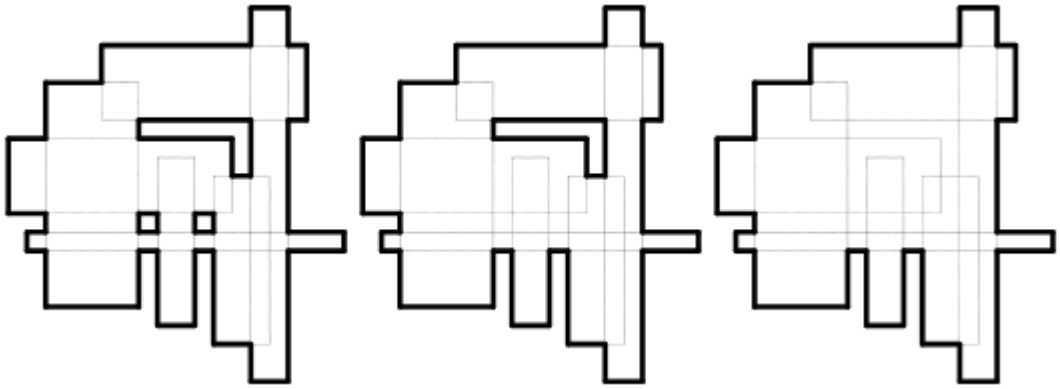


Рис. 2.1. Приклади: контуру, нетривіального контуру і зовнішнього контуру для об'єднання прямокутників

Для подальшого викладу зручно вважати кожне ребро таким, що складається з двох дуг, що орієнтовані в протилежних напрямах і лежать по різні сторони від заданого ребра (рис. 2.2).

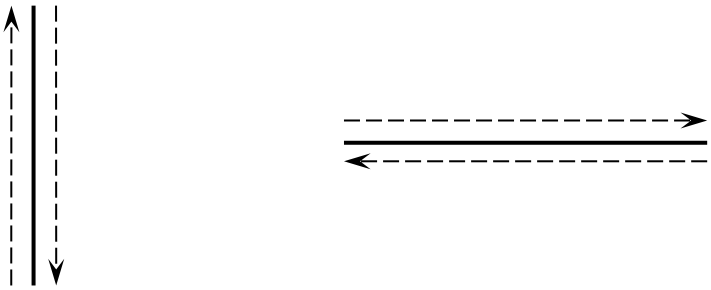


Рис. 2.2. Погодження відносно напрямів дуг для ребра

Ребра прямокутників R_1, ..., R_N розбивають площину на області, одна з яких нескінченна. Після введення допоміжних дуг, межі всіх областей цього розбиття породжують набір *орієнтованих ланцюгів*, що складаються з дуг. Подібний ланцюг називається *зовнішнім* або *внутрішнім* залежно від того, орієнтований він за годинниковою стрілкою чи проти. Дуга в ланцюзі називається *кінцевою*, якщо вона містить кінцеву точку того заданого відрізку (сторони прямокутника), якому вона належить. Ланцюг називається *нетривіальним* або *тривіальним* в залежності від того, містить він чи ні кінцеву дугу. Множина нетривіальних ланцюгів для прикладу з рис. 2.1 проілюстрована на рис. 2.3.

Задача. Заданий набір з N ізотетичних прямокутників. Знайти *нетривіальний* контур їх об'єднання.

Задача. Заданий набір з N ізотетичних прямокутників. Знайти *зовнішній* контур їх об'єднання.

Безпосередньо видно справедливість наступного ланцюжка включень:

Зовнішній контур $\subseteq$ *нетривіальний контур* $\subseteq$ *нетривіальні ланцюги*.

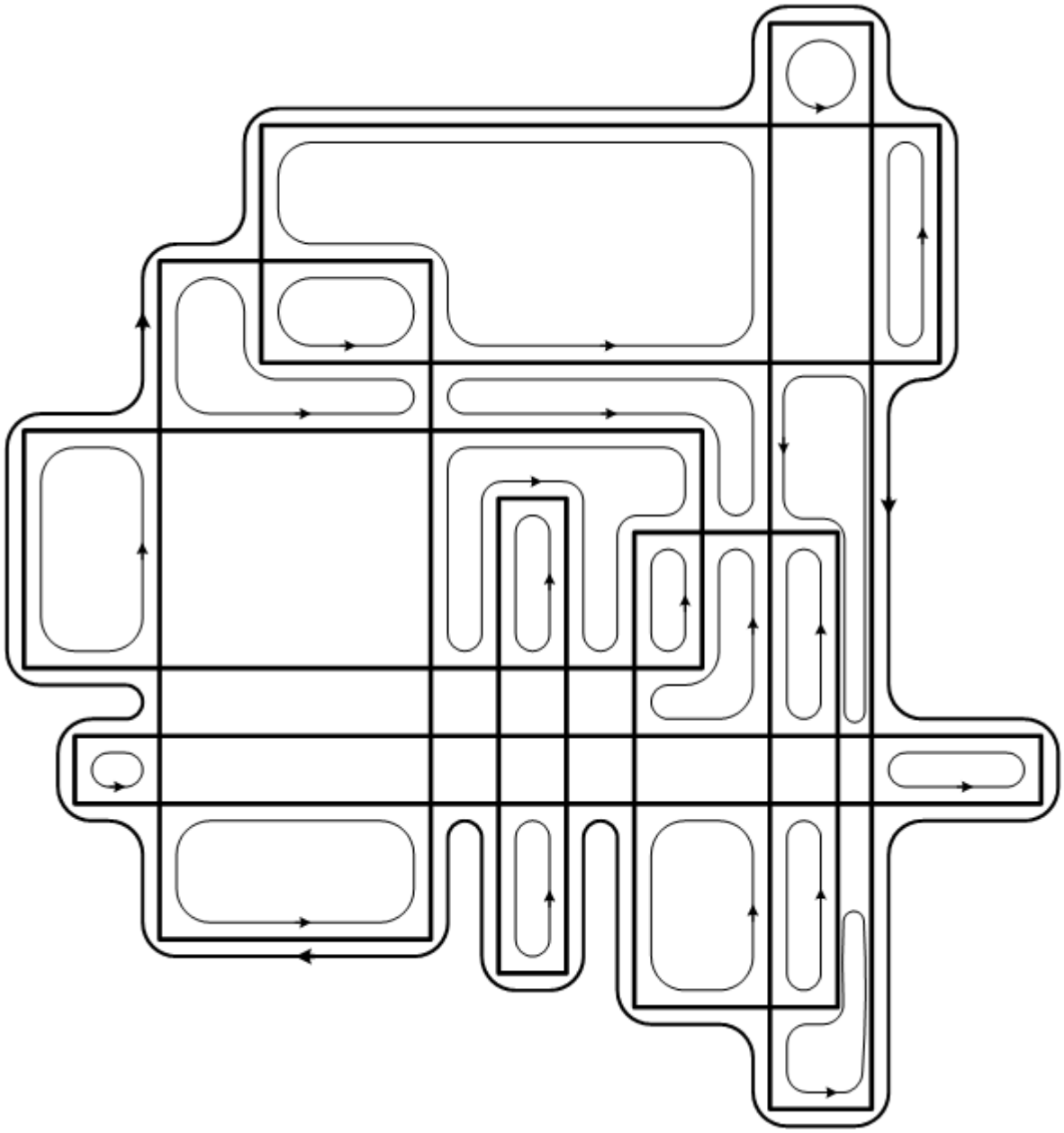


Рис. 2.3. Нетривіальні ланцюги для об'єднання прямокутників з рис. 2.1

Важлива властивість цих ланцюгів у тому, що число дуг в нетривіальних ланцюгах рівне O(N).

Тепер опишемо метод побудови зовнішнього контуру об'єднання ізотетичних прямокутників. Основною частиною процедури обходу є механізм руху, згідно якого ми вирушаємо з поточної вершини v_1 (рис. 2.4) і просуваємося по поточному відрізку l_1 в заданому напрямі, причому може виникнути одна з наступних двох ситуацій.

1. Існує такий відрізок l_2, найближчий до v_1, який перетинає l_1 і заходить всередину області, що лежить зліва від l_1. В цьому випадку робимо лівий поворот, тобто точка перетину v_2 стає поточною вершиною, а l_2 — поточним відрізком.
2. Досягнута кінцева точка v_3 відрізку l_1, що співпадає з кінцевою точкою відрізку l_2 (вочевидь l_2 не заходить всередину області, що лежить зліва від

l_1). В цьому випадку робимо правий поворот, тобто v_3 стає поточною вершиною, а l_2 — поточним відрізком.

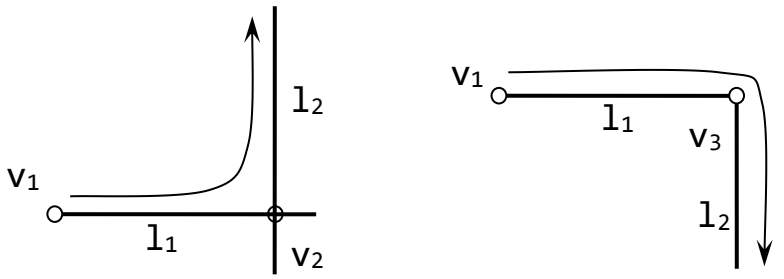


Рис. 2.4. Ситуації, що виникають в алгоритмі на етапі руху

Описаний вище крок руху можна реалізувати шляхом пошуку на двох відповідних геометричних структурах, що називаються *горизонтальною* і *вертикальною картами суміжності*, які будуть зараз викладені. Обмежимося горизонтальною картою суміжності (ГКС), оскільки всі міркування повною мірою застосовні до вертикальної карти суміжності.

Розглянемо множину V, що складається з вертикальних відрізків, які є вертикальними сторонами заданих прямокутників (рис. 2.5). Через кожну кінцеву точку p кожного відрізку з V проведемо пару горизонтальних променів: один вправо і один вліво; кожен з цих променів або закінчується на найближчому до p вертикальному відрізку, або якщо подібного перетину не існує, промінь продовжується в нескінченність.

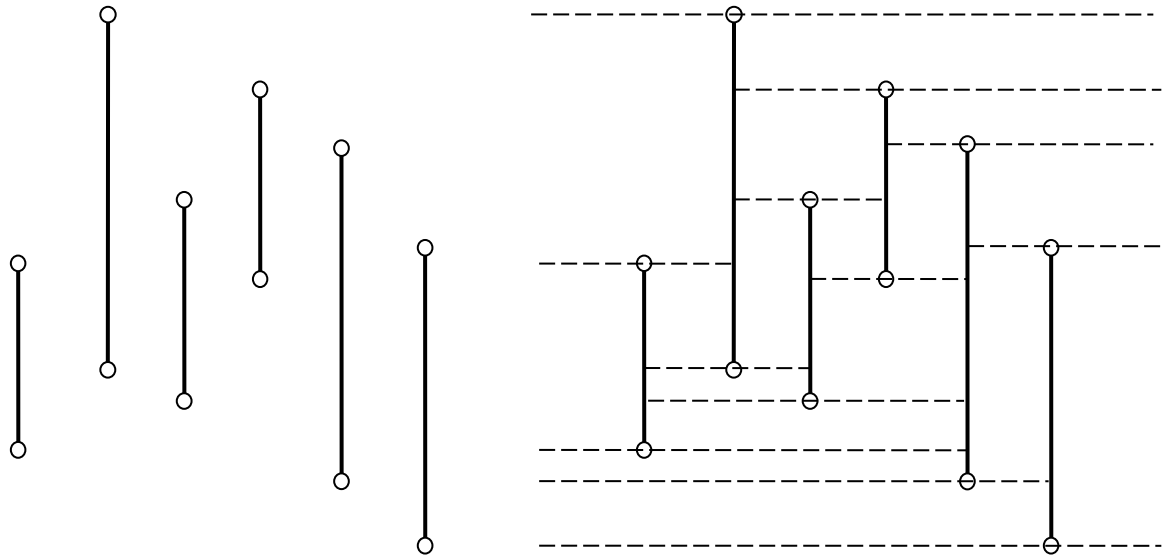


Рис. 2.5. Множина вертикальних відрізків V та результуюча ГКС

Таким чином площина розбивається на області, дві з яких є півплощинами, а всі інші — прямокутниками, можливо нескінченними в одному або обох горизонтальних напрямах. Ці області називатимемо «узагальненими прямокутниками». Кожен узагальнений прямокутник є класом еквівалентності для точок на заданій площині по відношенню до їх горизонтальної суміжності вертикальним відрізкам (звідси і назва «горизонтальна карта суміжності»).

Якщо передбачити, що поточний відрізок l_1 горизонтальний (рис. 2.4), то локалізація поточної вершини v_1 в одній з областей заданої карти (тобто визначення того узагальненого прямокутника, який містить v_1) означає визначення вертикальних сторін цієї області. Легко переконатися, що це все, що потрібно для реалізації кроку руху. Дійсно, припустимо, що v_1 — (x_1, y_1), а l_1 лежить на прямій $y=y_1$ в інтервалі $[x_1, x_3]$.

Локалізуємо точку (x_1, y_1) в ГКС і отримуємо абсцису x_2 для найближчого вертикального відрізку, що лежить праворуч від (x_1, y_1). Якщо $x_2 \leq x_3$, то треба зробити лівий поворот (випадок 1); якщо $x_2 > x_3$, то треба зробити правий поворот (випадок 2). Три випадки розташування поточного відрізку (зліва, зверху і знизу від поточної вершини), що залишилися, обробляються абсолютно аналогічним чином.

Для завершення побудови необхідно видалити ті ланцюги, які не відділяють F від його зовнішнього простору.

Запитання для самоконтролю

1. Як встановити перетин двох прямокутників?
2. Що таке нетривіальний контур об'єднання прямокутників?
3. Що таке зовнішній контур об'єднання прямокутників?
4. Наведіть алгоритм побудови зовнішнього контуру об'єднання прямокутників.
5. Що таке горизонтальна та вертикальна карти суміжності? Для чого вони використовуються?

3. ЗАДАЧІ З БАГАТОКУТНИКАМИ

В цьому розділі розглянуті деякі алгоритми, що застосовуються до багатокутників.

3.1. ОБЧИСЛЕННЯ ПЛОЩІ БАГАТОКУТНИКА

Нехай заданий багатокутник P, що утворений вершинами (v_0, v_1,..., v_n). Для того, щоб знайти його площу S(P), можна скористатися наступною формулою:

$$S(P) = \frac{1}{2}\sum_{i=0}^{n-1}\left(v_{i,x} \cdot v_{i+1,y} - v_{i,y} \cdot v_{i+1,x}\right). \qquad (3.1)$$

Формула (3.1) підраховує площу багатокутника зі знаком, що залежить від орієнтації його вершин. У випадку, коли вершини впорядковані в напрямку проти годинникової стрілки S(P)<0.

3.2. ПОЛОЖЕННЯ ТОЧКИ ВІДНОСНО БАГАТОКУТНИКА

Якщо вершини багатокутника впорядковані за годинниковою стрілкою, то точка знаходиться усередині цього багатокутника, якщо вона завжди знаходиться праворуч від спостерігача, що здійснює обхід сторін багатокутника у відповідності до порядку вершин. Якщо багатокутник опуклий, то ліва частина нерівності (1.1) має від'ємне значення для всіх сторін багатокутника. У цьому випадку вирішення задачі є тривіальним.

Якщо багатокутник не є опуклим, то знайти вирішення вже не так просто. Можна показати, що будь-яка точка, що міститься усередині неопуклого багатокутника G, належить опуклому багатокутнику, створеному сторонами багатокутника G та їх продовженнями. Але побудова подібних багатокутників є важкою задачею. Під час вирішення деяких прикладних задач, вони можуть задаватися заздалегідь, як це буває, наприклад, у випадках, коли неопуклий багатокутник побудований як деяке об'єднання опуклих багатокутників. Витрата зусиль на побудову таких багатокутників може бути виправдана і у тому випадку, коли необхідна перевірка розташування відносно неопуклого багатокутника значної кількості точок.

Інший спосіб вирішення, що застосовується до будь-якого багатокутника, передбачає проведення крізь точку P прямої, визначення її перетину зі всіма сторонами багатокутника G та використання критерію парності (рис. 3.1). Якщо відомо, що перша точка відповідної прямої знаходиться за межами області, що розглядається, то виконавши обхід цієї прямої, можна визначити, шляхом підрахування кількості перетинів, які відрізки розміщені усередині області. Якщо кількість перетинів — непарна, то відповідний відрізок знаходиться всередині області (відрізки AB та CD), у протилежному випадку — за межами області (відрізок BC). У випадку дотичної прямої — точку дотику необхідно враховувати два рази.

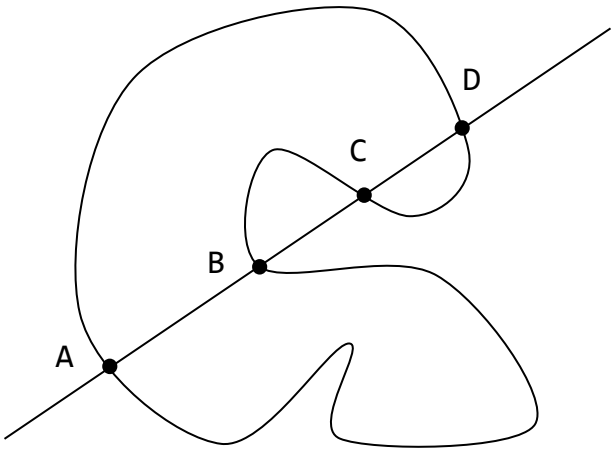


Рис. 3.1. Приклад використання принципу перевірки на парність для визначення точок, які лежать всередині замкненої кривої

3.3. Розрізання відрізка прямої опуклим багатокутником

Нехай заданий плоский опуклий багатокутник $P(A_1, A_2,\ldots, A_n)$. Необхідно визначити, яка частина відрізка прямої L, що задана точками $P_1(x_1, y_1, w_1)$ та $P_2(x_2, y_2, w_2)$ знаходиться всередині цього багатокутника.

Спочатку необхідно визначити взаємне розташування вершин багатокутника та відрізка прямої. Для цього координати кожної вершини багатокутника по черзі підставимо в рівняння прямої L, що проходить через точки P_1 та P_2:

$$S_i = \det(A_i, P_1, P_2)\,; i = \overline{1,n};$$

$$S_i = X_i(y_1 w_2 - w_1 y_2) + Y_i(w_1 x_2 - x_1 w_2) + W_i(x_1 y_2 - y_1 x_2).$$

Отримані результати S_i можуть відповідати одному з наступних випадків.

1. Всі значення S_i — ненульові та мають один і той же знак. Отже, всі вершини багатокутника розміщені по один бік від відрізка прямої L і жодна з частин прямої не міститься всередині багатокутника.
2. Одне із значень S_i дорівнює 0, а всі інші значення мають один і той же знак. Відповідно, пряма проходить крізь одну з вершин багатокутника, а всі інші його вершини розміщені по один бік від цієї прямої. Даний випадок еквівалентний випадку 1.
3. Два значення S_i дорівнюють 0, а всі інші значення мають один і той же знак. Оскільки багатокутник P — опуклий, це може означати лише те, що дві сусідні вершини багатокутника знаходяться на відрізку прямої L. Одна із сторін багатокутника розміщена на тій же прямій, що і відрізок прямої L.
4. Одне чи два значення S_i дорівнюють 0, а інші значення мають різні знаки. Відповідно, відрізок прямої L перетинає багатокутник P, проходячи крізь одну чи дві його вершини. Оскільки багатокутник P — опуклий, знаки змінюються лише двічі. Цей випадок буде розглядатися разом з наступним.
5. Всі S_i мають ненульові значення і їх знаки різні. Це означає, що при обході багатокутника по периметру знак змінюється двічі. Для позначення пар

вершин, на яких відбувається зміна знаку, будуть використовуватись символи A_j, A_{j+1} та A_k, A_{k+1}. У випадку 4 величина S_i може приймати ненульове значення у одній з вершин однієї чи обох пар. Відрізок прямої, що з'єднує вершини першої пари, позначається L_1, а відрізок прямої, що з'єднує вершини другої пари — L_2. Відмітимо, що не має значення, яка пара назначається першою, а яка другою; те ж саме відноситься до відповідних позначень, якщо прийнята процедура обробки точно виконується.

Після встановлення перетину прямої, що містить відрізок L з багатокутником (два останніх випадки), необхідно визначити положення точок P_1 та P_2 відносно відрізків прямих L_1 та L_2, тобто знаки наступних чотирьох величин:

$$U_1 = \det(P_1, A_j, A_{j+1});$$
$$U_1 = x_1(Y_jW_{j+1} - W_jY_{j+1}) + y_1(W_jX_{j+1} - X_jW_{j+1}) + w_1(X_jY_{j+1} - Y_jX_{J+1}); \quad (3.2)$$

$$U_2 = \det(P_2, A_j, A_{j+1});$$
$$U_2 = x_2(Y_jW_{j+1} - W_jY_{j+1}) + y_2(W_jX_{j+1} - X_jW_{j+1}) + w_2(X_jY_{j+1} - Y_jX_{J+1}); \quad (3.3)$$

$$U_3 = \det(P_1, A_k, A_{k+1});$$
$$U_3 = x_1(Y_kW_{k+1} - W_kY_{k+1}) + y_1(W_kX_{k+1} - X_kW_{k+1}) + w_1(X_kY_{k+1} - Y_kX_{k+1}); \quad (3.4)$$

$$U_4 = \det(P_2, A_k, A_{k+1});$$
$$U_4 = x_2(Y_kW_{k+1} - W_kY_{k+1}) + y_2(W_kX_{k+1} - X_kW_{k+1}) + w_2(X_kY_{k+1} - Y_kX_{k+1}). \quad (3.5)$$

Якщо орієнтація сторін багатокутника впорядкована за годинниковою стрілкою, величина U_i має від'ємний знак у рівняннях (3.2)–(3.5), коли точка лежить всередині багатокутника. Значення U_i дорівнює 0, якщо одна з точок P_1 і P_2 лежить на одній із сторін багатокутника. На рисунку 3.2 подані різні варіанти розміщення точок P_1 і P_2 відносно двох перетнутих ребер багатокутника з нанесеними знаками U_i.

Кінцеві точки відрізка прямої L, що знаходиться всередині багатокутника позначаються Q_1 та Q_2 і у трьох випадках (б, в, г) знаходяться як перетин двох відрізків.

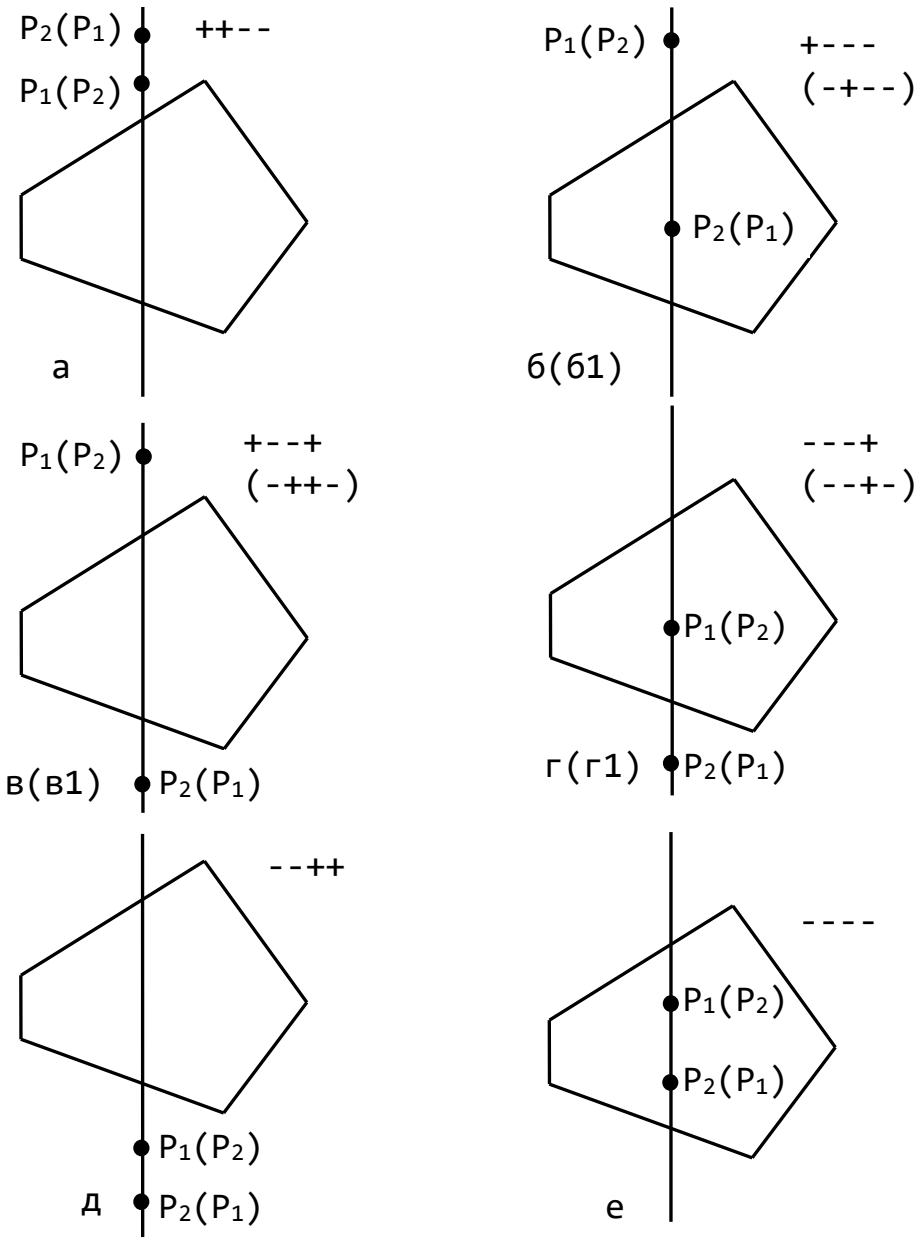


Рис. 3.2. Шість варіантів розрізання відрізка прямої багатокутником

Запитання для самоконтролю

1. Як визначити положення точки відносно опуклого багатокутника?
2. На якому принципі базується визначення положення точки відносно довільного багатокутника?
3. Яка послідовність дій для визначення фрагменту відрізка, що знаходиться всередині багатокутника?

4. ДВОВИМІРНЕ ВІДСІКАННЯ

Відсікання в комп'ютерній графіці це видалення зі списку побудови тих об'єктів або їх частин, що не потрапляють в область інтересу (viewport) . У випадку двовимірного відсікання viewport це деякий багатокутник (найчастіше прямокутник) (рис. 4.1).

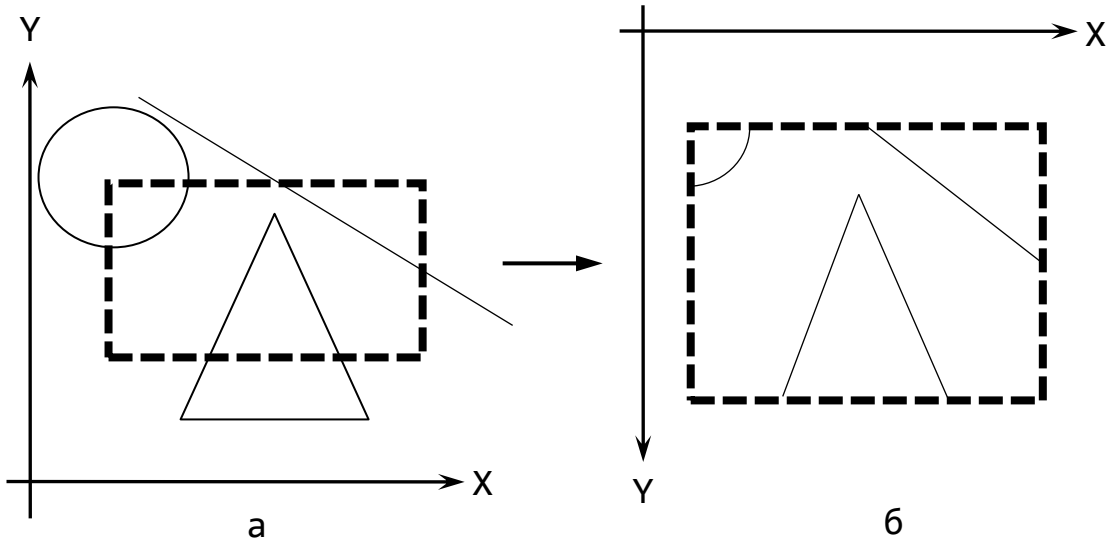


Рис. 4.1. Двовимірне відсікання: (а) — viewport в світовій системі координат; (б) — відображення viewport в приладовій системі координат

В загальному випадку двовимірне відсікання зводиться до пошуку перетину відрізків, що утворюють контури складніших об'єктів, і границь багатокутника, що описує viewport. Після цього необхідно визначити, яка частина об'єкта потрапляє в область інтересу і залишити в списку побудови лише ті частини відрізків, що видимі для користувача.

Якщо вирішувати цю задачу наївним методом через пошук точки перетину двох відрізків, як це було описано в першому розділі, то такий спосіб буде вкрай неефективний. Тому були розроблені спеціалізовані алгоритми для двовимірного і тривимірного відсікання. Ми буде розглядати приклади для двовимірного випадку але вони можуть бути застосовані також і для тривимірного простору.

4.1. АЛГОРИТМ КОЕНА-САЗЕРЛЕНДА

Цей алгоритм відсікання був розроблений американськими вченими Дані Коеном та Айваном Сазерлендом протягом 1967-1968 років у Гарварді.

Цей алгоритм дозволяє визначити випадок, коли відрізок лежить повністю поза прямокутником шляхом виконання побітової операції AND. Основою алгоритму є розбиття простору на дев'ять частин, кожна з яких отримує чотирьохбітний двійковий код (рис. 4.2).

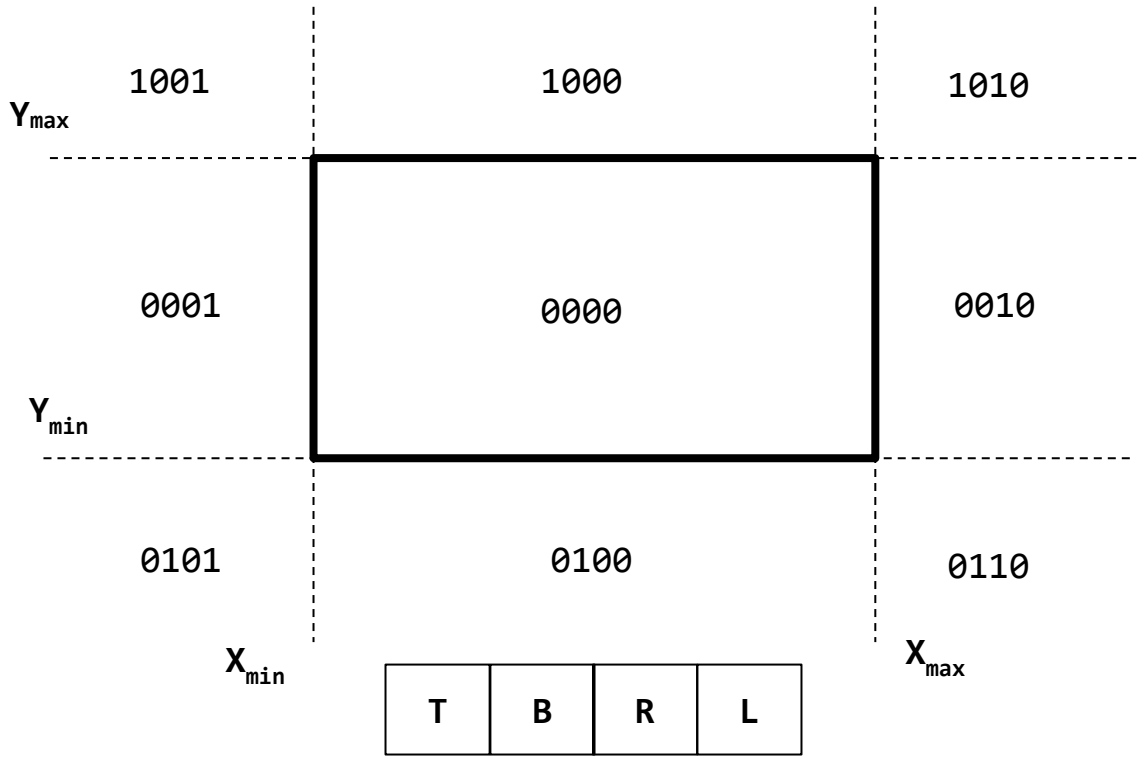


Рис. 4.2. Розбиття простору на 9 частин

Перший біт ліворуч позначає простір, що знаходиться вище прямокутника, який визначає область інтересу. Другий біт позначає простір, що знаходиться нижче прямокутника. Третій і четвертий біти позначають простори, що знаходяться відповідно праворуч і ліворуч від прямокутника.

4.1.1. Формальний опис алгоритму для одного відрізка

1. Для початкової $P_0(x_0,y_0)$ та кінцевої $P_1(x_1,y_1)$ точок відрізка розрахувати коди регіонів, в яких вони знаходяться. Це легко робиться шляхом порівняння відповідних координат точок і мінімальних та максимальних значень координат прямокутника.
2. Якщо обидва коди складаються лише з нулів, це означає, що відрізок знаходиться повністю всередині прямокутника. Відрізок додається у список відображення і алгоритм закінчує роботу для цього відрізка.
3. Перевіряємо результат побітової операції AND між двома кодами. Якщо результат операції не дорівнює нулю це означає, що відрізок повністю лежить за межею прямокутника і він не додається до списку відображення. Алгоритм закінчує свою роботу для цього відрізка.
4. Оскільки результат побітової операції AND між двома кодами нульовий, отже відрізок потенційно перетинає межі прямокутника. Обираємо не нульовий код кінця відрізку (як мінімум один не буде нульовим) і в ньому шукаємо першу одиницю. Знаходимо точку перетину з відповідним ребром прямокутника:

- для T: $x=x_0+(x_1-x_0)*(Y_{max}-y_0)/(y_1-y_0)$;

- для B: $x=x_0+(x_1-x_0)*(Y_{min}-y_0)/(y_1-y_0)$;
- для R: $y=y_0+(y_1-y_0)*(X_{max}-x_0)/(x_1-x_0)$;
- для L: $y=y_0+(y_1-y_0)*(X_{min}-x_0)/(x_1-x_0)$.

5. Заміняємо кінцеву точку відрізка на розраховану і переходимо до першого пункту.

Таким чином, в найгіршому випадку, алгоритм зробить не більше як чотири відсікання для одного відрізка.

4.1.2. Приклад виконання алгоритму

В якості прикладу розглянемо випадок коли відрізок перетинає дві сусідні сторони прямокутника (рис. 4.3).

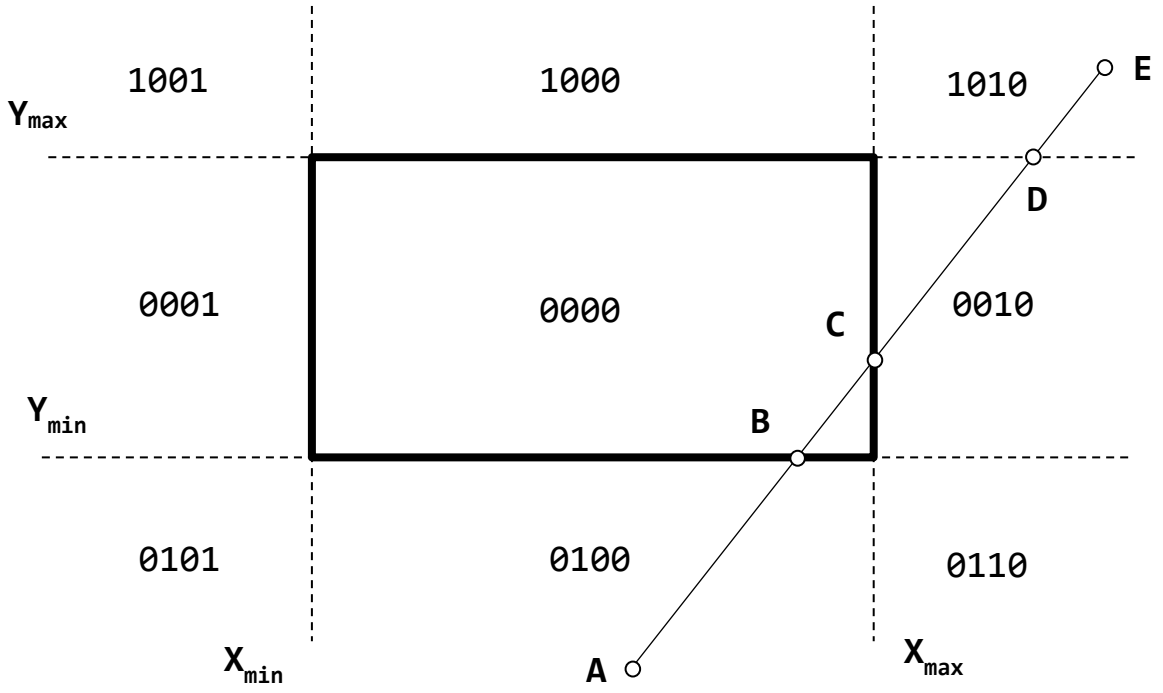


Рис. 4.3. Відрізок AE перетинає дві сусідні сторони прямокутника

Крок 1. Рахуємо коди для кінцевих точок відрізка: A(0100) та E(1010). Вони обидва не нульові, а отже перевіряємо результат побітової операції AND між ними: A & E = 0100 & 1010 = 0000. Отриманий результат говорить про те, що можливий перетин відрізка зі сторонами прямокутника. Отже необхідно обрати ненульовий код і знайти в ньому першу одиницю. Обираємо код для точки E(1010). Перша одиниця в ньому стоїть в позиції T, а отже необхідно шукати перетин відрізка з верхнім горизонтальним ребром прямокутника. В результаті ми отримаємо точку D, яка стає новою кінцевою точкою відрізка (рис. 4.4).

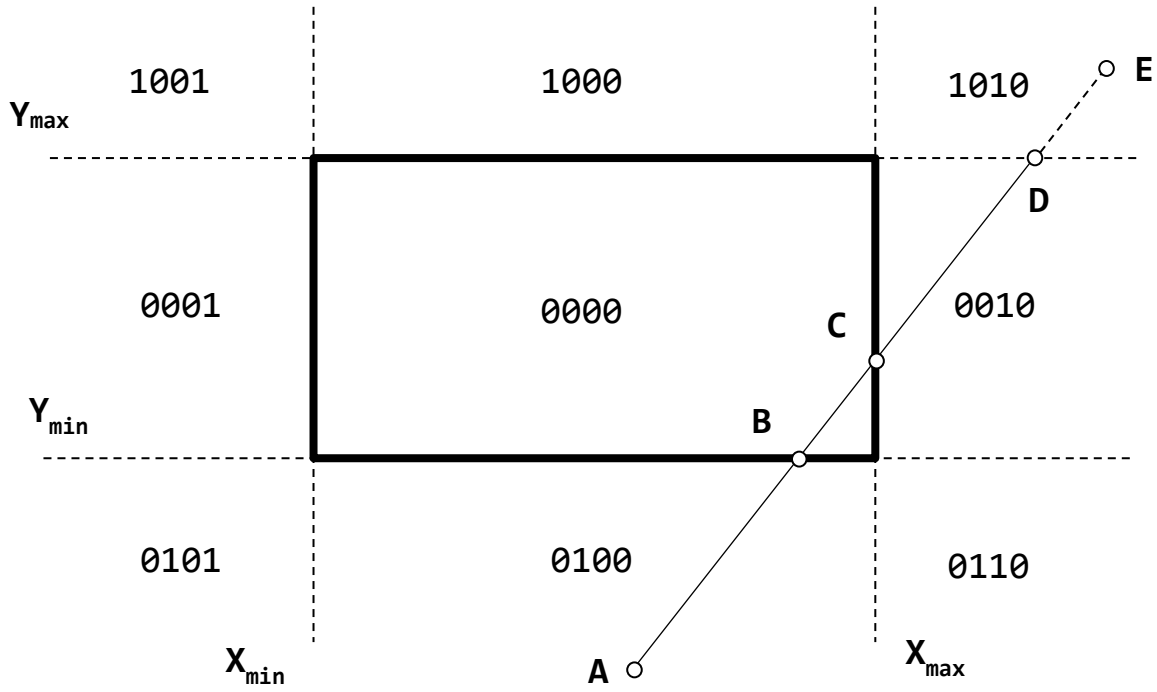


Рис. 4.4. Відрізок AE після першого відсікання

Крок 2. Рахуємо код для нової кінцевої точки відрізка D(0010). Коди початкової та кінцевої точок не нульові, отже перевіряємо їх оператором AND. В результаті отримуємо таке значення: A & D = 0100 & 0010 = 0000. Отже знову необхідно обрати ненульовий код і обрати в ньому позицію одиниці. Обираємо код для точки D(0010). Перша одиниця в ньому стоїть в позиції R, а отже необхідно шукати перетин відрізка з правим вертикальним ребром прямокутника. В результаті ми отримаємо точку C, яка стає кінцевою точкою відрізка (рис. 4.5).

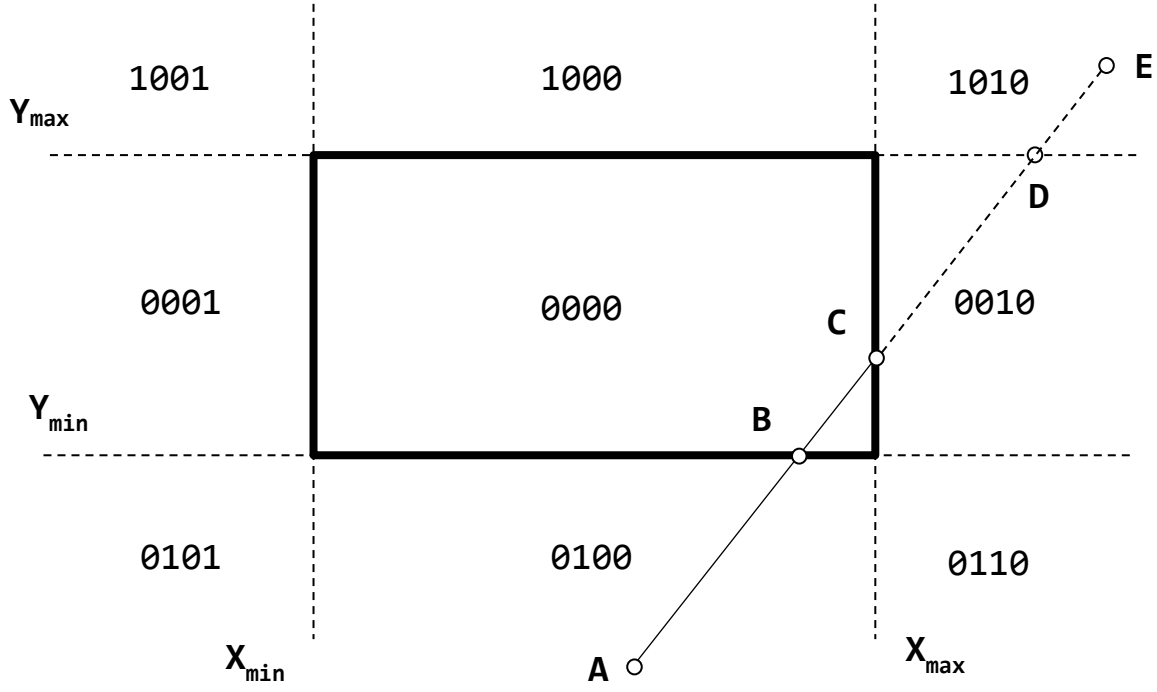


Рис. 4.5. Відрізок AE після другого відсікання

Крок 3. Рахуємо код для нової кінцевої точки відрізка C(0000). Код однієї точки нульовий, а іншої ні. В такому випадку операція AND завжди дає нульове значення, а отже необхідно шукати точку перетину. Розглядаємо ненульовий код A(0100) та знаходимо в ньому позицію одиниці. Вона знаходиться в позиції B, отже необхідно шукати перетин відрізка з нижнім горизонтальним ребром прямокутника. В результаті ми отримаємо точку B, яка стає новою кінцевою точкою відрізка (рис. 4.6).

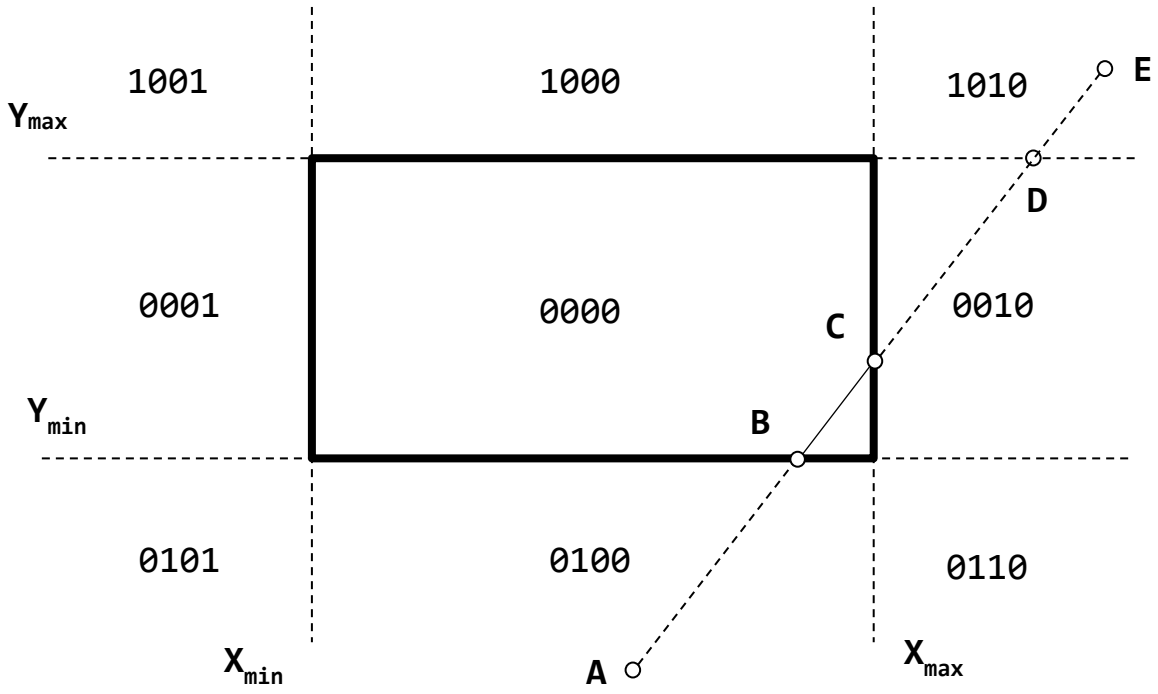


Рис. 4.6. Відрізок AE після третього відсікання

Крок 4. Рахуємо код для нової кінцевої точки відрізка B(0000). Коди обох точок B та C нульові, а отже відрізок BC знаходиться повністю всередині прямокутника і саме він повинен бути доданий в список відображення. На цьому алгоритм закінчує свою роботу.

4.1.3. Переваги та недоліки

Як і будь-який інший алгоритм, розглянутий алгоритм також має свої недоліки:

- фіксований порядок перевірки на першу одиницю іноді призводить до непотрібної роботи;
- область відсікання повинна бути прямокутною.

До переваг алгоритму слід віднести наступні:

- легкий в реалізації;
- орієнтований на більшість приладів, оскільки їх екрани є прямокутними;
- легко розширюється на тривимірний простір.

4.2. Алгоритм Сайруса-Бека

Цей алгоритм робить відсікання відрізків довільним опуклим багатокутником. Запропонований в 1978 році Майком Сайрусом і Джеєм Беком як ефективніша заміна алгоритму Коена-Сазерленда, який робить кілька ітерацій для відсікання одного відрізка і працює лише з прямокутниками.

Основою алгоритму Сайруса-Бека є використання параметричного рівняння прямої:

$$P(t) = P_0 + t \cdot (P_1 - P_0),$$

де P_0 — початкова точка відрізка, P_1 — кінцева точка відрізка, t — параметр $[0:1]$.

Алгоритм перевіряє, знаходиться перетин відрізка і ребра багатокутника в середині багатокутника чи ні. Для цієї перевірки використовується скалярний добуток двох векторів: вектора, що побудований із точки на ребрі багатокутника і точки на відрізку та вектора нормалі до ребра багатокутника (рис. 4.7).

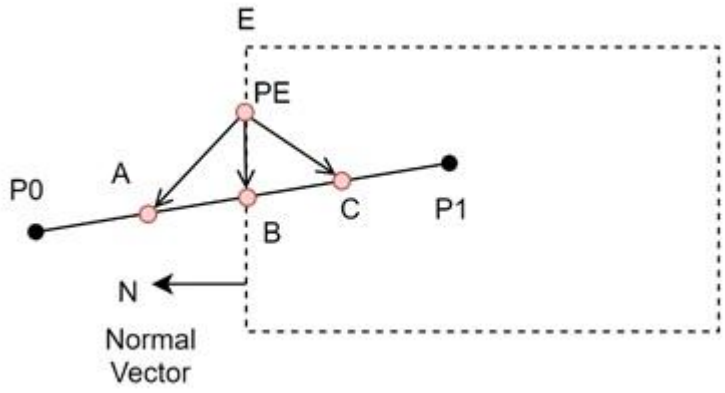


Рис. 4.7. Вектор нормалі та варіанти векторів від ребра до відрізка

На рисунку 4.7 вектор нормалі (N) це перпендикуляр до ребра багатокутника. В загальному випадку необхідно розрахувати нормалі до всіх ребер багатокутника.

Вектор відстані ($P(t)-P_E$) це вектор, що утворюється з точки на ребрі (P_E) (зазвичай це точка початку або кінця ребра) та точки на відрізку ($P(t)$) — точка з параметром t, що перевіряється.

Скалярний добуток цих двох векторів може бути додатнім, від'ємним або нульовим. В залежності від знаку цього добутку ми визначаємо положення точки з параметром t на відрізку: знак «+» — точка за межами багатокутника відсікання (точка A на рис. 4.7); знак «-» — точка всередині багатокутника відсікання (точка C на рис. 4.7). Якщо значення нульове, то це означає, що точка лежить на ребрі багатокутника(точка B на рис. 4.7).

Яким чином скалярний добуток нам допомагає визначити розташування точки? Позначимо вектор відстані ($P(t)-P_E$) через D. За визначенням скалярний добуток двох векторів це:

$$N \cdot D = |N| \cdot |D| \cdot \cos\theta.$$

Якщо кут між векторами 90°, тоді косинус дорівнює 0 і увесь добуток стає нульовим. Таким чином ми визначаємо, що точка перетину лежить на ребрі. Для кута >90° косинус буде від'ємним, а отже точка лежить всередині багатокутника, коли скалярний добуток <0. Відповідно для кута <90° косинус буде додатнім, і це означає, що точка лежить за межами багатокутника, коли добуток >0.

Тепер розглянемо, яким чином знайти параметри t для кожного ребра. Замінимо P(t) у векторі відстані на рівняння прямої через початкову і кінцеву точки відрізка і підставимо у векторний добуток. Оскільки ми шукаємо параметр t для точки перетину відрізка з ребром, то векторний добуток повинен бути нульовим:

$$N \cdot [P_0 + t \cdot (P_1 - P_0) - P_E] = 0,$$

після розкриття дужок отримаємо наступне:

$$N \cdot (P_0 - P_E) + N \cdot t \cdot (P_1 - P_0) = 0.$$

З цієї формули виведемо чому дорівнює параметр t:

$$N \cdot t \cdot (P_1 - P_0) = -N \cdot (P_0 - P_E),$$

тоді

$$t = \frac{N \cdot (P_E - P_0)}{N \cdot (P_1 - P_0)}.$$

Зауважте, що в нас ділення двох скалярних добутків векторів, а тому ми не можемо просто скоротити вектор N в чисельнику та знаменнику.

Для коректної роботи алгоритму, необхідне виконання таких умов для вхідних даних:

- $N \neq 0$ — вектор нормалі до кожного ребра не повинен бути нульовим;
- $(P_1 - P_0) \neq 0$ — відрізок не повинен бути точкою;
- $N \bullet (P_1 - P_0) \neq 0$ — відрізок не повинен бути паралельним до ребра. У такому випадку шукати параметр t для цього ребра не потрібно, оскільки точки перетину ребра і відрізка не існує.

4.2.1. ФОРМАЛЬНИЙ ОПИС АЛГОРИТМУ

1. Розрахувати нормалі до кожного ребра багатокутника.
2. Розрахувати вектор відрізка для відсікання.
3. Для кожного ребра побудувати вектор від початкової або кінцевої точки на відрізку для відсікання до початкової або кінцевої точки ребра і розрахувати скалярний добуток цього вектора на вектор нормалі до цього ж ребра.
4. Знайти скалярний добуток вектора нормалі до поточного ребра і вектора, що утворений відрізком відсікання.
5. Векторний добуток з п.3. поділити на векторний добуток з п.4 — це і буде параметр t для цього ребра.

6. Параметри t, що розраховані для всіх ребер, розбиваємо на дві групи: вхідні та вихідні. До вхідних додаємо ті, у яких скалярний добуток з п.4. був від'ємним, а до вихідних — всі інші. До вхідних також додаємо значення 0, а до вихідних — значення 1.
7. Серед всіх вхідних параметрів t_{in} обираємо максимальне, що більше або дорівнює нулю, а серед всіх вихідних t_{out} — мінімальне, що менше або дорівнює одиниці.
8. Якщо значення вхідного параметра t_{in} більше за значення вихідного t_{out}, то відкидаємо відрізок як такий, що лежить за межею багатокутника. В інакшому випадку будуємо відрізок:

$$P_b = P_0 + t_{in} \cdot (P_1 - P_0);$$
$$P_e = P_0 + t_{out} \cdot (P_1 - P_0).$$

4.2.2. Приклад виконання алгоритму

В якості прикладу розглянемо випадок коли відрізок перетинає одне ребро прямокутника (рис. 4.8). Замість прямокутника можна розглядати будь-який опуклий багатокутник.

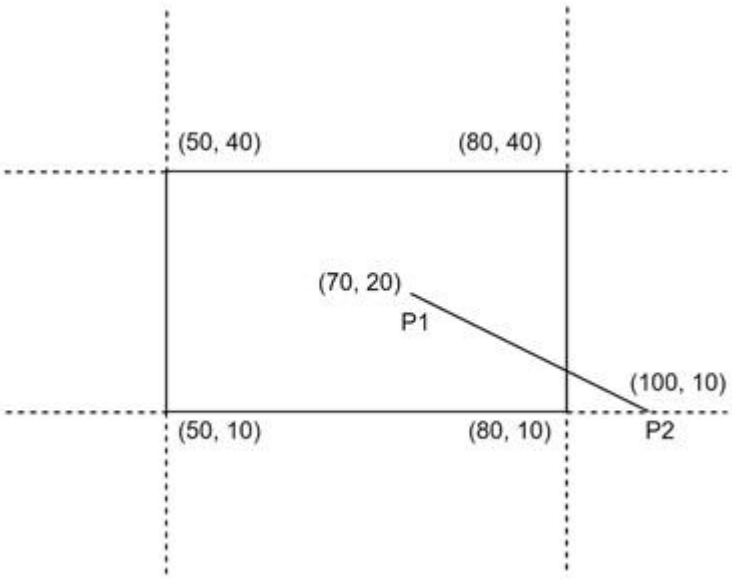


Рис. 4.8. Відрізок P_1P_2 перетинає сторону прямокутника

Крок 1. Рахуємо нормалі до кожного ребра багатокутника. Для лівого ребра це буде N_L=(-1, 0), для правого — N_R=(1, 0), для нижнього — N_B=(0, -1), для верхнього — N_T=(0, 1) (рис. 4.9).

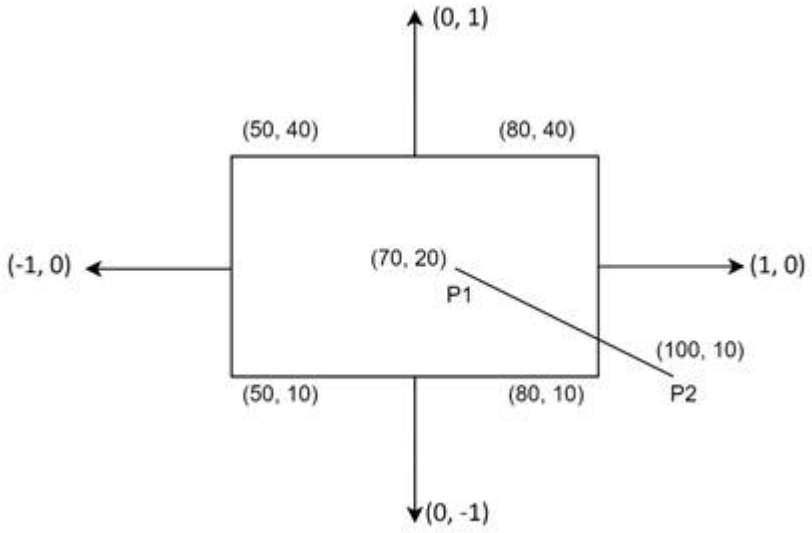


Рис. 4.9. Нормалі до сторін прямокутника

Крок 2. Розраховуємо вектор відрізка відсікання:

$$P_2 - P_1 = (100 - 70, 10 - 20) = (30, -10).$$

Крок 3. Будуємо вектори від початкових точок ребер до початкової точки відрізка (P_1) та знаходимо їх скалярні добутки з відповідними нормалями:

$$N_L \cdot (P_\text{L} - P_1) = (-1, 0) \cdot (50 - 70, 10 - 20) = 20 - 0 = 20;$$
$$N_R \cdot (P_\text{R} - P_1) = (1, 0) \cdot (80 - 70, 10 - 20) = 10 - 0 = 10;$$
$$N_B \cdot (P_\text{B} - P_1) = (0, -1) \cdot (50 - 70, 10 - 20) = 0 + 10 = 10;$$
$$N_T \cdot (P_\text{T} - P_1) = (0, 1) \cdot (50 - 70, 40 - 20) = 0 + 20 = 20.$$

Крок 4. Знаходимо скалярні добутки нормалей кожного ребра багатокутника з вектором відрізку відсікання:

$$N_L \cdot (P_2 - P_1) = (-1, 0) \cdot (30, -10) = -30 - 0 = -30;$$
$$N_R \cdot (P_2 - P_1) = (1, 0) \cdot (30, -10) = 30 - 0 = 30;$$
$$N_B \cdot (P_2 - P_1) = (0, -1) \cdot (30, -10) = 0 + 10 = 10;$$
$$N_T \cdot (P_2 - P_1) = (0, 1) \cdot (30, -10) = 0 - 10 = -10.$$

Крок 5. Знаходимо параметри t для кожного ребра багатокутника:

$$t_L = \frac{N_L \cdot (P_L - P_1)}{N_L \cdot (P_2 - P_1)} = \frac{20}{-30} = -\frac{2}{3};$$
$$t_R = \frac{N_R \cdot (P_R - P_1)}{N_R \cdot (P_2 - P_1)} = \frac{10}{30} = \frac{1}{3};$$
$$t_B = \frac{N_B \cdot (P_B - P_1)}{N_B \cdot (P_2 - P_1)} = \frac{10}{10} = 1;$$
$$t_T = \frac{N_T \cdot (P_T - P_1)}{N_T \cdot (P_2 - P_1)} = \frac{20}{-10} = -2;$$

Крок 6. Вхідними параметрами будуть ті, для яких скалярний добуток на четвертому кроці був від'ємним. Всі інші будуть вихідними. Отже вхідні параметри t_L і t_T, а вихідні — t_R і t_B.

Крок 7. Серед всіх вхідних параметрів обираємо максимальне серед тих які більші або дорівнюють нулю, а серед вихідних мінімальне серед тих, які менші або дорівнюють одиниці. Отже вхідним параметром стає $t_{in}=0$ оскільки обидва кандидати менші за нуль, а вихідним — $t_{out}=t_R=1/3$.

Крок 8. Оскільки $t_{in}<=t_{out}$ отже відрізок знаходиться в середині багатокутника між цими параметрами:

$$P_b = P_1 + t_{in} \cdot (P_2 - P_1) = (70, 20) + 0 \cdot (30, -10) = (70, 20);$$

$$P_e = P_1 + t_{out} \cdot (P_2 - P_1) = (70, 20) + \frac{1}{3} \cdot (30, -10) = (80, 16\frac{2}{3}).$$

Таким чином ви знайшли початкову і кінцеву точки відрізка після відсікання.

4.2.3. Переваги та недоліки

До переваг алгоритму слід віднести наступні:

- більш ефективний за алгоритм Коена-Сазерленда, оскільки не робить ітеративні відсікання;
- може використовуватися для вікна відсікання у вигляді будь-якого опуклого багатокутника.

Основним недоліком алгоритму є те, що він може використовуватися лише у двовимірному просторі.

4.3. Алгоритм Ляна-Барського

Цей алгоритм використовує той самий принцип, що і алгоритм Сайруса-Бека, але оптимізований під відсікання саме прямокутником у двовимірному випадку. Розробили алгоритм китайський вчений Лян Юдун та американець Браян Барський у 1984 році. Пізніше, в 1992, вони його удосконалили і з тих пір він не змінювався.

Оскільки в нас замість багатокутника використовується прямокутник, то ми можемо значно простіше порахувати всі чотири параметри t для кожного ребра. Позначимо координати прямокутника через X_{min}, Y_{min}, X_{max} та Y_{max}, а координати відрізка через $P(x_1, y_1)$ і $Q(x_2, y_2)$ відповідно (рис. 4.10).

Запишемо параметричне рівняння для обох координат через параметр t:

$$X = x_1 + t \cdot (x_2 - x_1);$$

$$Y = y_1 + t \cdot (y_2 - y_1).$$

Нагадаємо, що параметр t знаходиться на проміжку $[0;1]$, а точка, що відповідає знайденому параметру t, повинна бути розташованою всередині прямокутника.

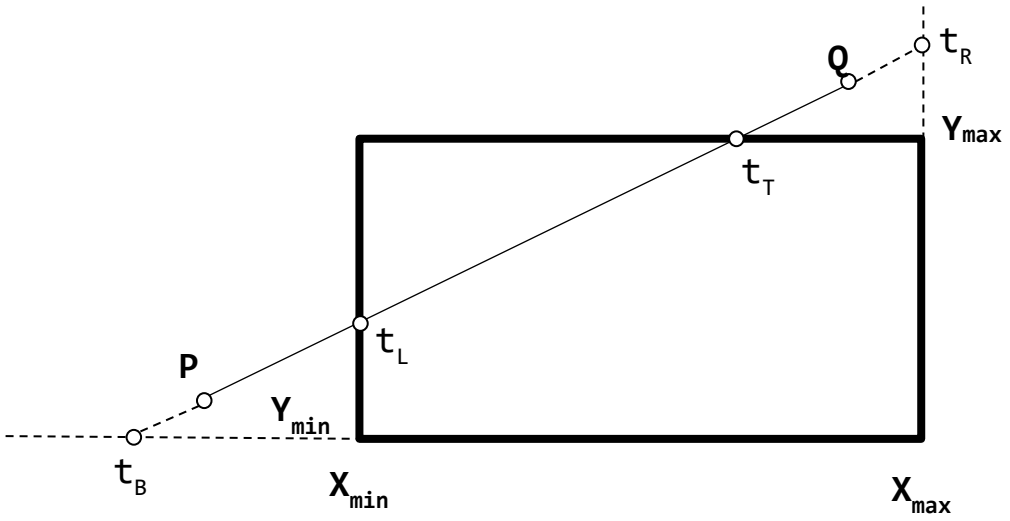


Рис. 4.10. Позначення до алгоритму Ляна-Барського

Щоб точка лежала в межах прямокутника, необхідне виконання таких чотирьох нерівностей:

$$X_{MIN} \le x_1 + t \cdot (x_2 - x_1);$$
$$X_{MAX} \ge x_1 + t \cdot (x_2 - x_1);$$
$$Y_{MIN} \le y_1 + t \cdot (y_2 - y_1);$$
$$Y_{MAX} \ge y_1 + t \cdot (y_2 - y_1).$$

Запишемо всі чотири нерівності у наступній формі:

$$t \cdot p_k \le q_k, \text{ де k=(L|R|T|B).}$$

Таким чином, ми отримаємо наступні формули для p і q:

$$p_L = (x_1 - x_2);\ q_L = (x_1 - X_{MIN});$$
$$p_R = (x_2 - x_1);\ q_R = (X_{MAX} - x_1);$$
$$p_B = (y_1 - y_2);\ q_B = (y_1 - Y_{MIN});$$
$$p_T = (y_2 - y_1);\ q_T = (Y_{MAX} - y_1).$$

Формули для розрахунку параметрів t будуть наступними:

$$t_L = \frac{q_L}{p_L};\ t_R = \frac{q_R}{p_R}; t_B = \frac{q_B}{p_B}; t_T = \frac{q_T}{p_T}.$$

Якщо відрізок паралельний одному з ребер прямокутника, то значення p для такого ребра буде дорівнювати нулю. Якщо значення p<0, то відповідний параметр t є вхідним, у зворотному випадку — вихідним. Якщо p=0 та q<0 це означає, що відрізок повністю за межами прямокутника. Якщо p=0 та q>0 — відрізок повністю знаходиться всередині прямокутника.

З чотирьох параметрів t потрібно обрати два — вхідний і вихідний. Для значення p<0 ми обираємо вхідний параметр як максимальний параметр, що більший за нуль. Для значення p>0 ми обираємо вихідний параметр як мінімальний

параметр, що менший за одиницю. Якщо в результаті вхідний параметр виявився більшим за вихідний то це означає, що відрізок лежить за межами прямокутника.

4.3.1. ФОРМАЛЬНИЙ ОПИС АЛГОРИТМУ

1. Встановити $t_{min}=0$ і $t_{max}=1$.
2. Розрахувати параметри t для всіх ребер прямокутника. Відкинути ті, що менші нуля та більші одиниці.
3. Серед параметрів, що залишилися визначити максимальний вхідний параметр (в якого $p<0$) та присвоїти його змінній t_{min} і мінімальний вихідний параметр (в якого $p>0$) та присвоїти його змінній t_{max}.
4. Якщо $t_{min}<t_{max}$ то будуємо відрізок за такими координатами (X_b,Y_b)–(X_e,Y_e):

$$X_b = x_1 + t_{min} \cdot (x_2 - x_1);\ Y_b = y_1 + t_{min} \cdot (y_2 - y_1);$$
$$X_e = x_1 + t_{max} \cdot (x_2 - x_1);\ Y_e = y_1 + t_{max} \cdot (y_2 - y_1).$$

5. Якщо $t_{min}>t_{max}$ то відрізок лежить повністю поза прямокутником і він відкидається.

4.3.2. ПРИКЛАД ВИКОНАННЯ АЛГОРИТМУ

Для прикладу розглянемо два випадки — коли відрізок перетинає прямокутник і коли не перетинає. Першим розглянемо приклад з перетином. Нехай прямокутник має такі координати: $X_{min}=0$, $Y_{min}=0$, $X_{max}=15$, $Y_{max}=10$, а координати відрізка $P(-5, 3)$ і $Q(20, 9)$ (рис. 4.11).

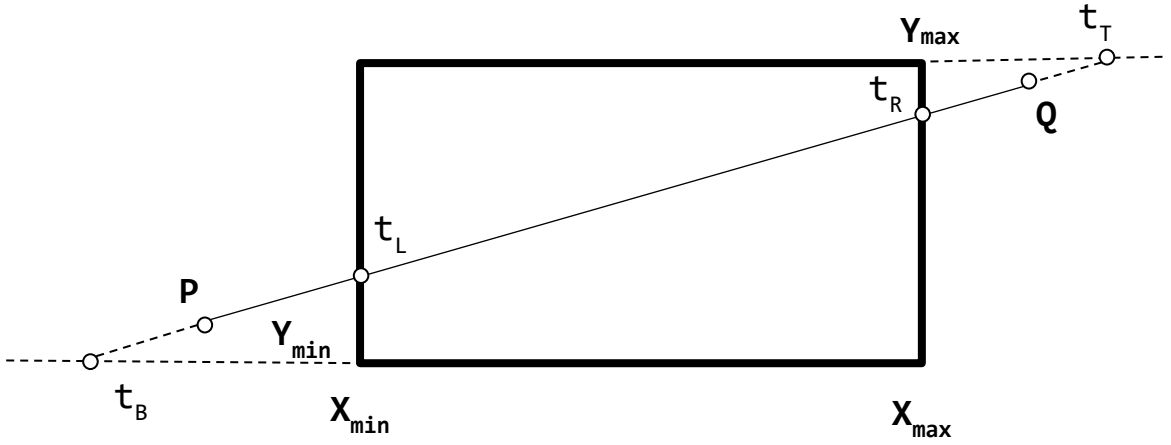


Рис. 4.11. Приклад алгоритму Ляна-Барського з перетином

Крок 1. Рахуємо параметри p та q для кожного ребра прямокутника:

$$p_L = (x_1 - x_2) = (-5 - 20) = -25;\ q_L = (x_1 - X_{MIN}) = (-5 - 0) = -5;$$
$$p_R = (x_2 - x_1) = (20 + 5) = 25;\ q_R = (X_{MAX} - x_1) = (15 + 5) = 20;$$
$$p_B = (y_1 - y_2) = (3 - 9) = -6;\ q_B = (y_1 - Y_{MIN}) = (3 - 0) = 3;$$
$$p_T = (y_2 - y_1) = (9 - 3) = 6;\ q_T = (Y_{MAX} - y_1) = (10 - 3) = 7.$$

Крок 2. Рахуємо параметри t для кожного ребра прямокутника:

$$t_L = \frac{q_L}{p_L} = \frac{-5}{-25} = \frac{1}{5};\ t_R = \frac{q_R}{p_R} = \frac{20}{25} = \frac{4}{5};$$

$$t_B = \frac{q_B}{p_B} = \frac{3}{-6} = -\frac{1}{2}; t_T = \frac{q_T}{p_T} = \frac{7}{6}.$$

Крок 3. Визначаємо максимальний вхідний параметр (в якого p<0) та присвоюємо його змінній t_{min}. В нас два таких параметри: t_L і t_B. Максимальний з них, що більший за 0 це параметр t_L, отже $t_{min}=t_L=1/5$.

Крок 4. Визначаємо мінімальний вихідний параметр (в якого p>0) та присвоюємо його змінній t_{max}. В нас два таких параметри: t_R і t_T. Мінімальний з них, що менший за 1 це параметр t_R, отже $t_{max}=t_R=4/5$.

Крок 5. Оскільки $t_{min}<t_{max}$ то будуємо відрізок за такими координатами (X_b,Y_b)–(X_e,Y_e):

$$X_b = -5 + \frac{1}{5}\cdot(20+5) = 0;\ Y_b = 3 + \frac{1}{5}\cdot(9-3) = 4\frac{1}{5};$$

$$X_e = -5 + \frac{4}{5}\cdot(20+5) = 15;\ Y_e = 3 + \frac{4}{5}\cdot(9-3) = 7\frac{4}{5}.$$

Тепер розглянемо приклад де відрізок лежить за межами прямокутника. Нехай прямокутник має такі координати: $X_{min}=0$, $Y_{min}=0$, $X_{max}=10$, $Y_{max}=10$, а координати відрізка P(-12, 4) і Q(6, 14) (рис. 4.12).

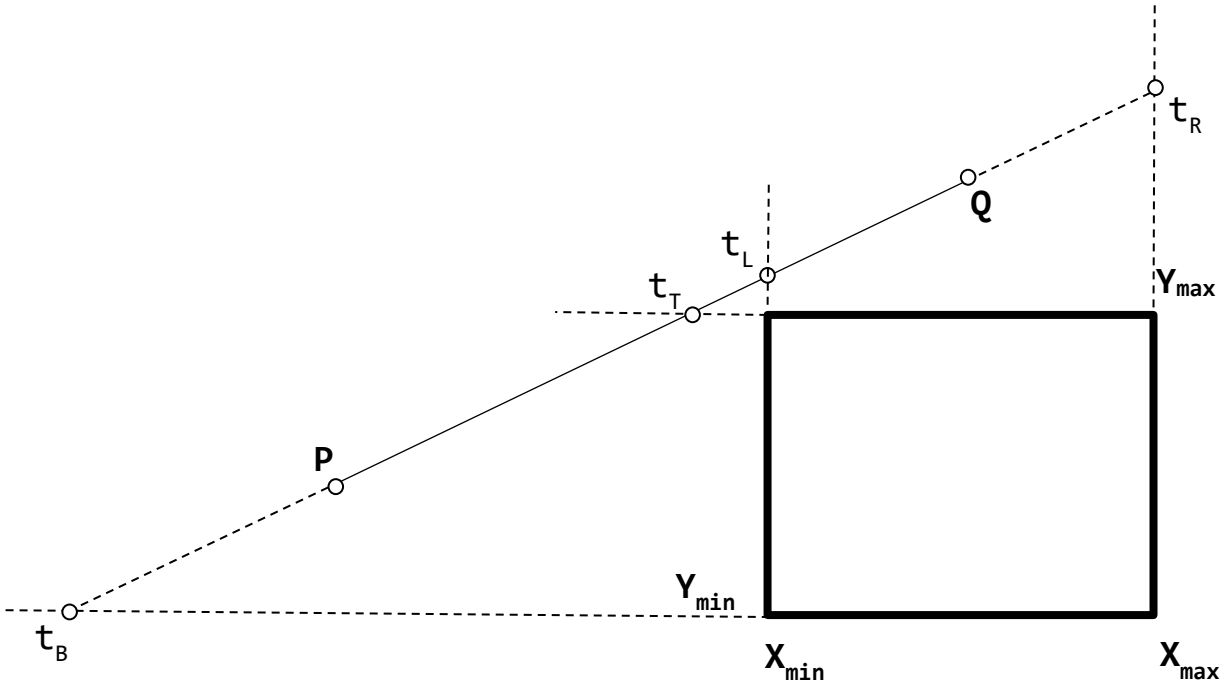


Рис. 4.12. Приклад алгоритму Ляна-Барського без перетину

Крок 1. Рахуємо параметри p та q для кожного ребра прямокутника:

$$p_L = (x_1 - x_2) = (-12 - 6) = -18;\ q_L = (x_1 - X_{MIN}) = (-12 - 0) = -12;$$
$$p_R = (x_2 - x_1) = (6 + 12) = 18;\ q_R = (X_{MAX} - x_1) = (10 + 12) = 22;$$
$$p_B = (y_1 - y_2) = (4 - 14) = -10;\ q_B = (y_1 - Y_{MIN}) = (4 - 0) = 4;$$
$$p_T = (y_2 - y_1) = (14 - 4) = 10;\ q_T = (Y_{MAX} - y_1) = (10 - 4) = 6.$$

Крок 2. Рахуємо параметри t для кожного ребра прямокутника:

$$t_L = \frac{q_L}{p_L} = \frac{-12}{-18} = \frac{2}{3};\ t_R = \frac{q_R}{p_R} = \frac{22}{18} = \frac{11}{9};$$
$$t_B = \frac{q_B}{p_B} = \frac{4}{-10} = -\frac{2}{5};\ t_T = \frac{q_T}{p_T} = \frac{6}{10} = \frac{3}{5}.$$

Крок 3. Визначаємо максимальний вхідний параметр (в якого p<0) та присвоюємо його змінній t_{min}. В нас два таких параметри: t_L і t_B. Максимальний з них, що більший за 0 це параметр t_L, отже $t_{min}=t_L=2/3$.

Крок 4. Визначаємо мінімальний вихідний параметр (в якого p>0) та присвоюємо його змінній t_{max}. В нас два таких параметри: t_R і t_T. Мінімальний з них, що менший за 1 це параметр t_R, отже $t_{max}=t_T=3/5$.

Крок 5. Оскільки $t_{min}>t_{max}$ то відрізок лежить повністю поза прямокутником і він відкидається.

4.3.3. Переваги та недоліки

До переваг алгоритму слід віднести наступні:

- значно швидший, ніж алгоритм Сайруса-Бека для прямокутних вікон, завдяки оптимізації;
- ефективний для відсікання відрізків для прямокутних областей;
- уникає обчислення точок перетину з непаралельними лініями, зменшуючи таким чином кількість операцій ділення.

Основним недоліком алгоритму є те, що він є ефективним лише для прямокутних вікон.

Запитання для самоконтролю

1. Який з розглянутих алгоритмів є найефективнішим для прямокутних областей відсікання?
2. Який з розглянутих алгоритмів був винайдений найпершим?
3. Який з розглянутих алгоритмів був винайдений найпізніше?
4. Який з розглянутих алгоритмів може бути розширеним на тривимірний простір?
5. Який з розглянутих алгоритмів може застосовуватися для довільних опуклих прямокутників?

5. Алгоритми побудови відрізка та кола

Алгоритми побудови відрізка — графічні алгоритми апроксимації відрізка на дискретному графічному пристрої (растеризація), наприклад, моніторі або принтері.

Відрізок, що заданий початковою та кінцевою точками у віконній системі координат має бути перетворений в послідовність пікселів, які необхідно зафарбувати, щоб отримати лінію. Аналогічним чином будується і коло (еліпс) — визначаються пікселі, що лежать на колі.

5.1. Алгоритм цифрового диференційного аналізатора

Цифровий диференційний аналізатор (ЦДА) — це простий алгоритм растеризації) відрізка, що в якості вхідних даних використовує координати початкової та кінцевої точок відрізка. Координати проміжних пікселів обчислюються в циклі з використанням значення приросту.

Може бути реалізований з використанням дійсних або цілих чисел.

5.1.1. Формальний опис алгоритму

Розглянемо алгоритм для відрізка, що лежить в першій половині першої координатної чверті.

1. Введіть початкову (x_1,y_1) та кінцеву (x_2,y_2) точки відрізка прямої.
2. Обчисліть різницю між координатами x та y кінцевих точок як dx та dy відповідно.
3. Обчисліть нахил прямої як $m=dy/dx$.
4. Встановіть початкову точку прямої як (x_1,y_1).
5. Запустіть цикл за координатами x відрізка від початкової до кінцевої, збільшуючи їх на одиницю кожного разу, та обчисліть відповідну координату y за формулою $y=y_1+m(x-x_1)$.
6. Побудуйте піксель в обчисленій координаті (x,y).
7. Повторюйте кроки 5 та 6, доки не буде досягнуто кінцевої точки (x_2,y_2).

Алгоритм для відрізка в другій половині першої чверті буде аналогічним, просто необхідно змінити формулу обчислення нахилу прямої на $m=dx/dy$ і рухатися в циклі за координатою y обчислюючи координату x за формулою $x=x_1+m\cdot(y-y_1)$.

Для інших координатних чвертей алгоритм працює аналогічно.

Оскільки нахил прямої це дійсне число, то результат розрахунку координати пікселя потрібно округлити, щоб отримати ціле значення.

Можна використовувати операцію побітового зсуву, щоб замінити ділення і працювати лише з цілими числами імітуючи арифметику дійсних чисел з фіксованою точкою.

5.1.2. Приклад виконання алгоритму

Нехай заданий відрізок у віконній системі координат з початковою точкою (0,0) і кінцевою точкою (8,6) (рис. 5.1). Для побудови використаємо алгоритм ЦДА.

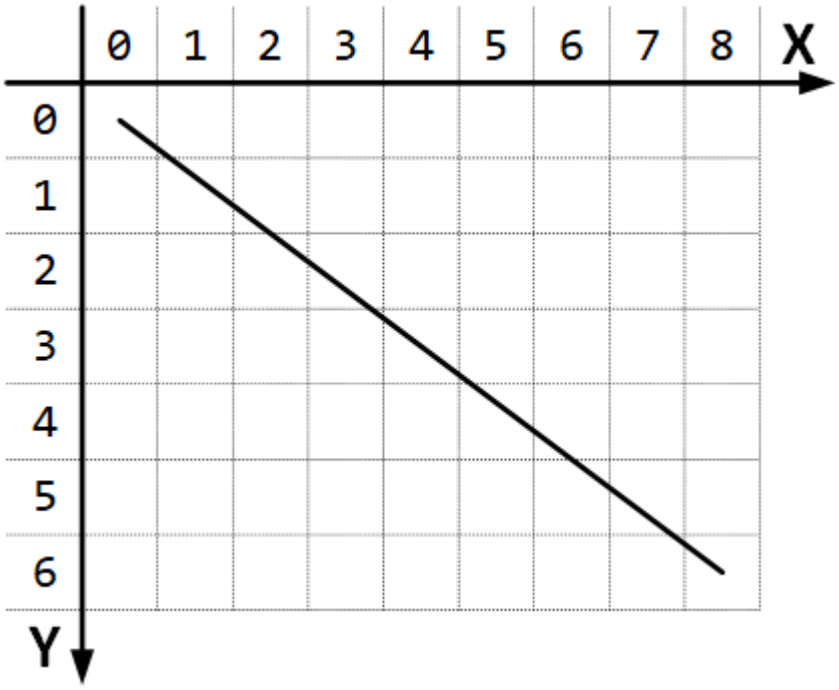


Рис. 5.1. Відрізок на піксельній решітці

Крок 1. Обчислюємо dx=(8-0)=8 та dy=(6-0)=6.

Крок 2. Обчислюємо нахил прямої m=dy/dx=6/8=0,75.

Крок 3. Встановлюємо початкову точку (0,0) і запускаємо цикл за координатою x від 0 до 8 включно (рис. 5.2).

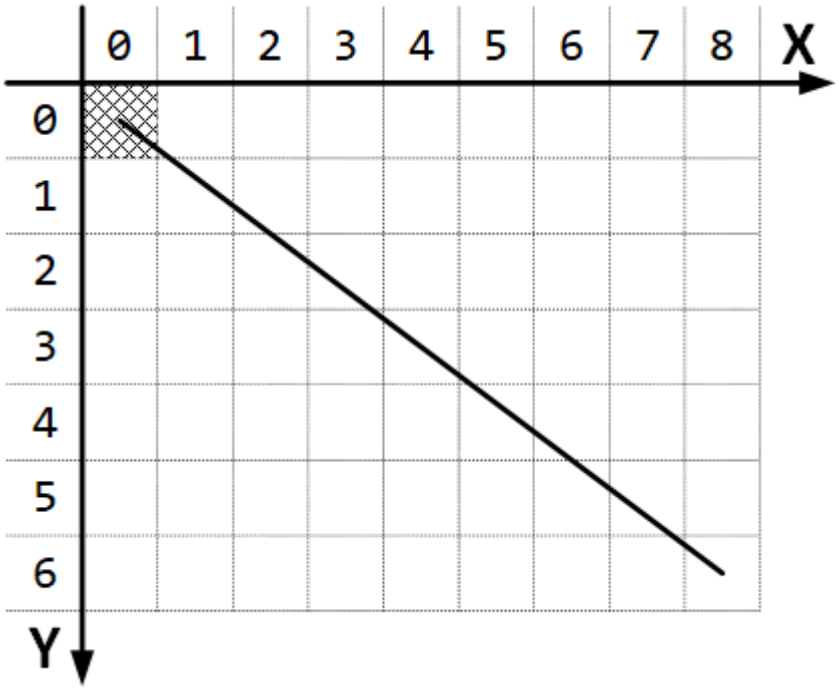


Рис. 5.2. Початкова точка на піксельній решітці

Ітерація №1. Координата x=1, а координата y розраховується за формулою: y=0+0,75·(1-0)=0,75. Після округлення отримуємо піксель (1,1) (рис. 5.3).

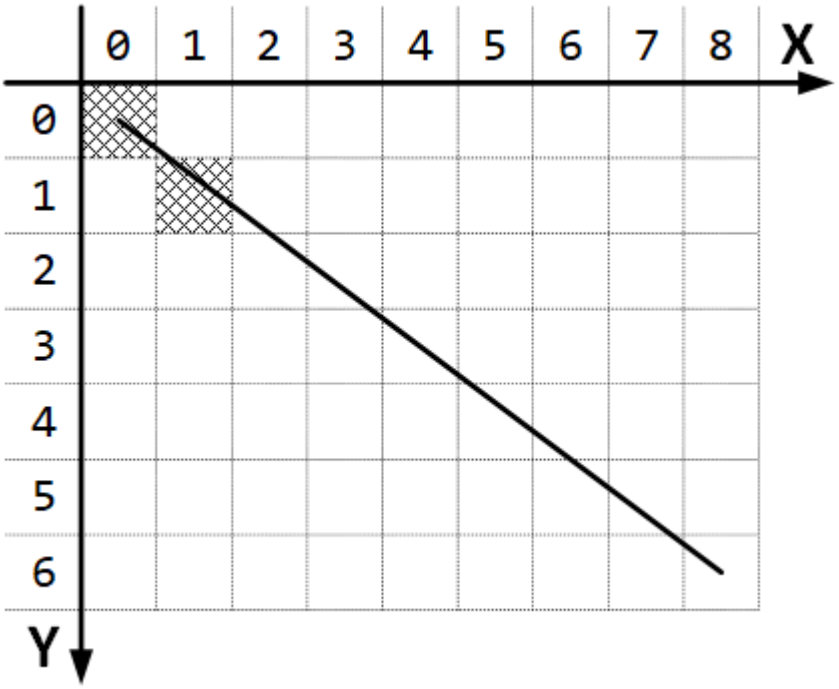


Рис. 5.3. Пікселі відрізка після ітерації №1

Ітерація №2. Координата x=2, а координата y=0+0,75·(2-0)=1,5. Після округлення отримуємо піксель (2,2) (рис. 5.4).

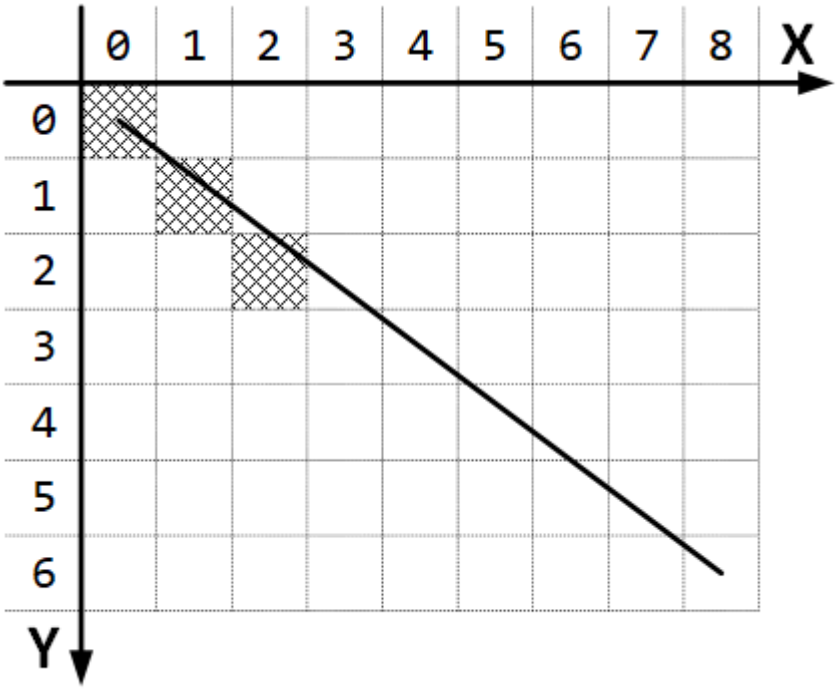


Рис. 5.4. Пікселі відрізка після ітерації №2

Ітерація №3. Координата x=3, а координата y=0+0,75·(3-0)=2,25. Після округлення отримуємо піксель (3,2) (рис. 5.5).

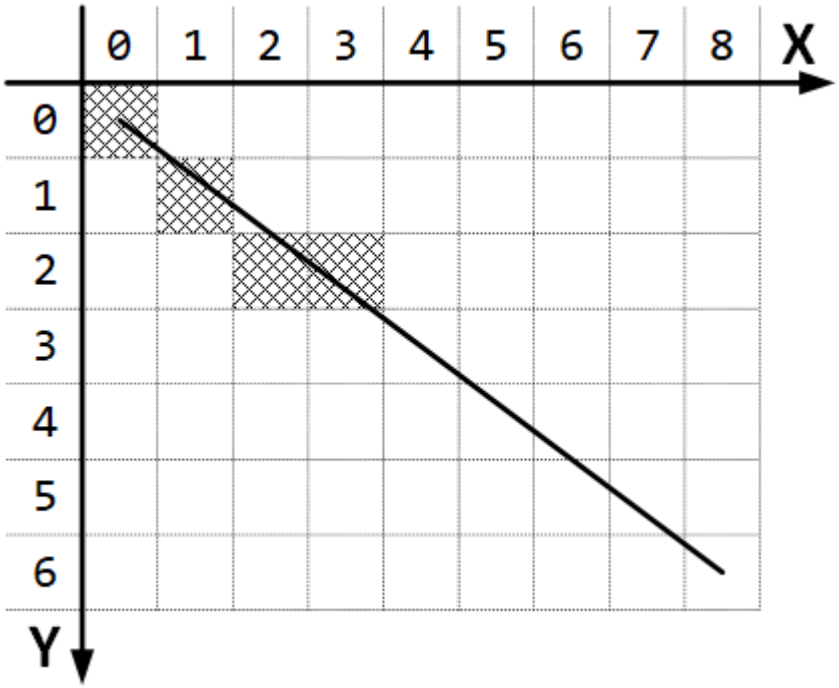


Рис. 5.5. Пікселі відрізка після ітерації №3

Ітерація №4. Координата x=4, а координата y=0+0,75·(4-0)=3. Після округлення отримуємо піксель (4,3) (рис. 5.6).

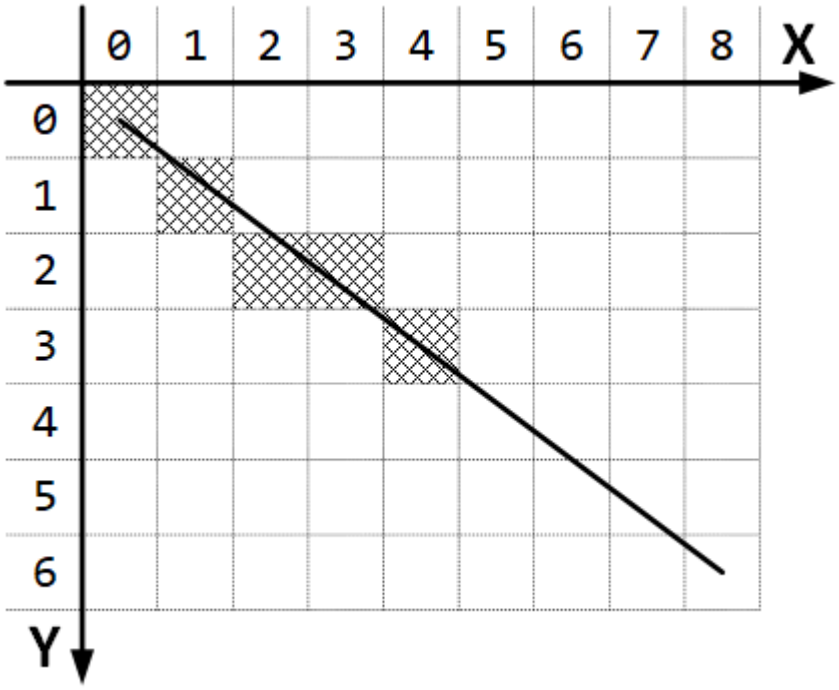


Рис. 5.6. Пікселі відрізка після ітерації №4

Ітерація №5. Координата x=5, а координата y=0+0,75·(5-0)=3,75. Після округлення отримуємо піксель (5,4) (рис. 5.7).

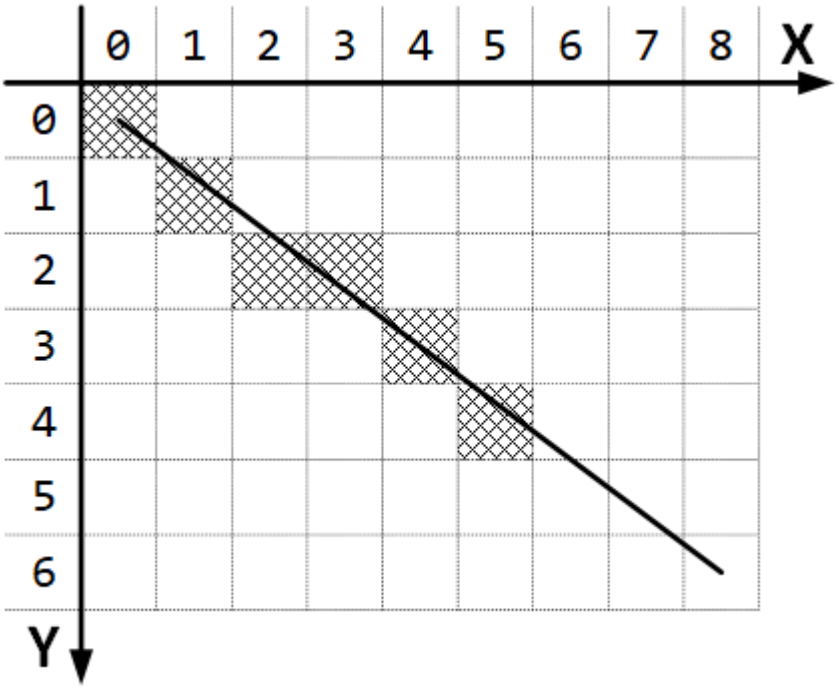


Рис. 5.7. Пікселі відрізка після ітерації №5

Ітерація №6. Координата x=6, а координата y=0+0,75·(6-0)=4,5. Після округлення отримуємо піксель (6,5) (рис. 5.8).

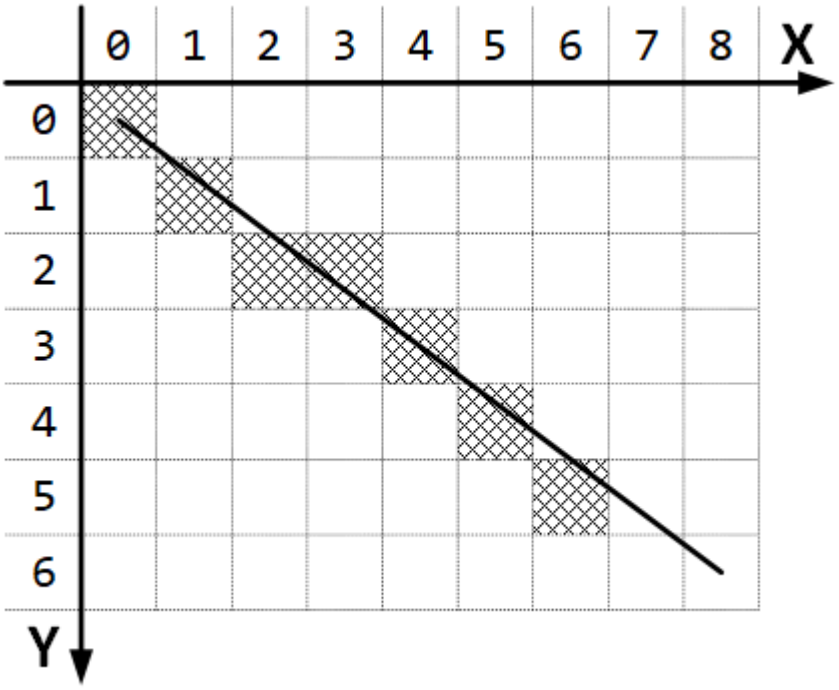


Рис. 5.8. Пікселі відрізка після ітерації №6

Ітерація №7. Координата x=7, а координата y=0+0,75·(7-0)=5,25. Після округлення отримуємо піксель (7,5) (рис. 5.9).

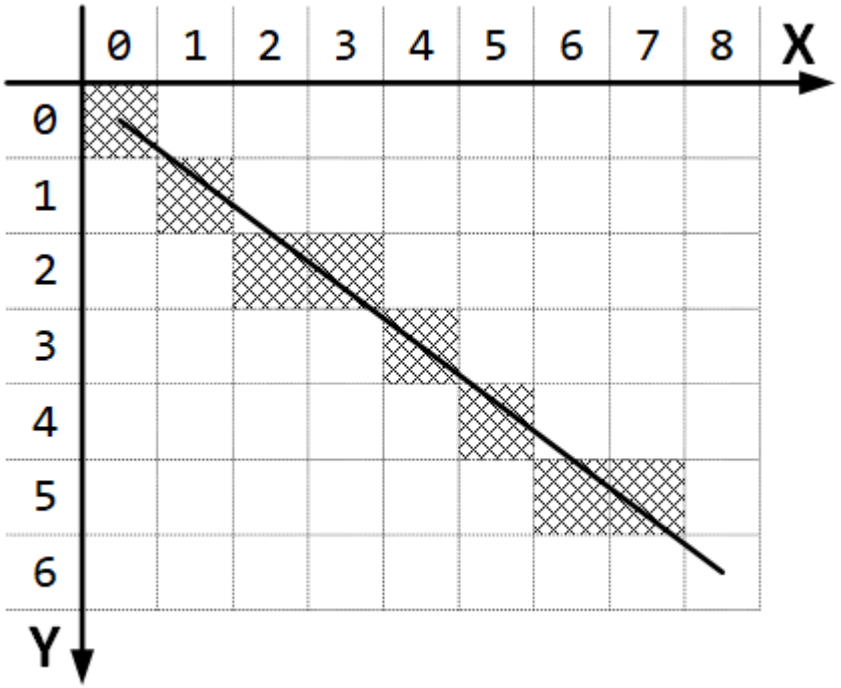


Рис. 5.9. Пікселі відрізка після ітерації №7

Ітерація №8. Координата x=8, а координата y=0+0,75·(8-0)=6. Після округлення отримуємо піксель (8,6) (рис. 5.10).

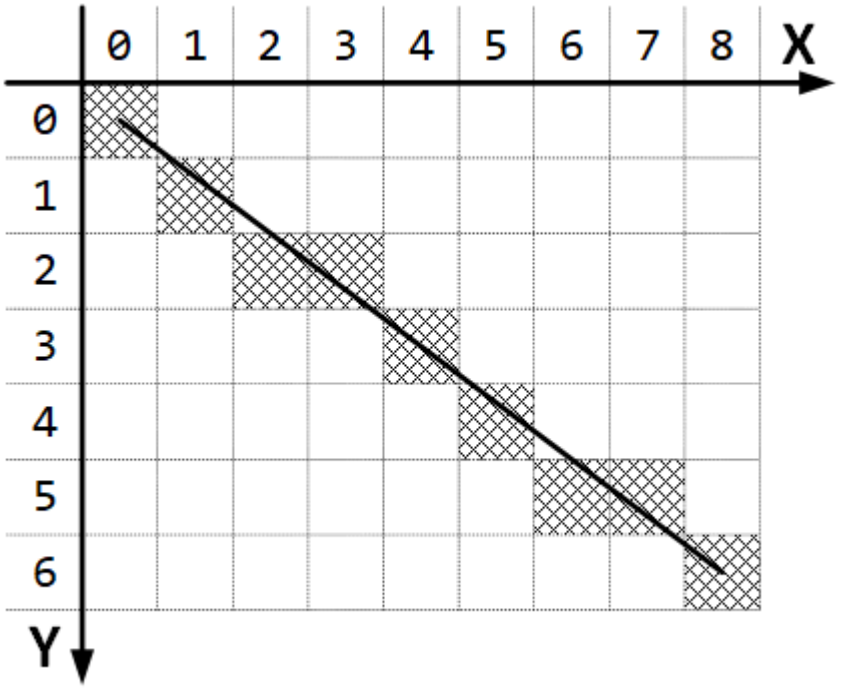


Рис. 5.10. Пікселі відрізка після ітерації №8

Алгоритм досягнув кінцевої точки відрізка, а тому закінчив свою роботу. У підсумку ми отримали пікселі, що утворюють заданий відрізок.

В практичні й реалізації варто замінити формулу $y=y_1+m(x-x_1)$ на ітераційний варіант: $y_i=Y_{i-1}+m$, де Y_{i-1} — ціла координата y пікселя, що обрахована на попередній ітерації циклу.

5.1.3. Переваги та недоліки

До переваг алгоритму відносять:

- простоту і легкість реалізації;
- відсутність повільних операцій множення;
- кращу швидкість в порівнянні з використанням параметричного рівняння прямої.

Недоліки алгоритму наступні:

- використовує повільну операцію округлення (цього можна уникнути, якщо використовувати імітацію операцій з фіксованою точкою);
- координати кінцевої точки можуть бути неточними;
- через операцію округлення здійснюється накопичення помилки.

5.2. Алгоритм Брезенхейма для побудови відрізка

Алгоритм Брезенхейма це фундаментальний метод у комп'ютерній графіці для апроксимації відрізка прямої лінії за допомогою дискретних пікселів, що забезпечує пряму та плавну растеризацію відрізка на піксельному дисплеї. Був винайдений Джеком Брезенхеймом у 1962 році та опублікований у 1965 році.

Алгоритм був розширений для побудови кіл, еліпсів та кубічних і квадратичних кривих Безьє.

Під час розгляду прикладу для алгоритму ЦДА, ми побачили, що на кожному кроці координата x завжди збільшується на 1, а координата y може збільшуватися на 1 або ні. Тобто фактично нам потрібно якимось чином вирішити який з двох варіантів обрати: $y_i=y_{i-1}$ чи $y_i=y_{i-1}+1$.

Як і алгоритм ЦДА, алгоритм Брезенхейма використовує нахил прямої, але з іншою метою. Брезенхейм запропонував на кожній ітерації рахувати накопичувальну помилку відхилення від нахилу прямої і на підставі цього ухвалювати рішення про те, який наступний піксель обрати. Початкове значення помилки (dp) приймається -1/2. В залежності від того який знак помилки на поточній ітерації обирається один із двох пікселів для побудови лінії.

5.2.1. Формальний опис алгоритму

Розглянемо алгоритм для відрізка, що лежить в першій половині першої координатної чверті (тобто в першому октанті).

1. Введіть початкову (x_1,y_1) та кінцеву (x_2,y_2) точки відрізка прямої.
2. Обчисліть різницю між координатами x та y кінцевих точок як dx та dy відповідно.
3. Обчисліть нахил прямої як $m=dy/dx$.
4. Встановіть початкову точку прямої як (x_1,y_1), а початкову помилку $dp_0=-1/2$.

5. Запустіть цикл за координатами x відрізка від початкової до кінцевої, збільшуючи їх на одиницю кожного разу, та оберіть відповідну координату y_i в залежності від знаку поточної помилки dp_i. Якщо $dp_i<0$, то $y_i=y_{i-1}$, інакше $y_i=y_{i-1}+1$. Якщо помилка $dp_i\geq 0$, тоді від неї потрібно відняти 1.
6. Побудуйте піксель в обчисленій координаті (x,y).
7. Розрахуйте оновлену помилку для наступного кроку за формулою $dp_{i+1}=dp_i+m$. Повторюйте кроки 5 та 6, доки не буде досягнуто кінцевої точки (x_2,y_2).

Як бачимо з опису алгоритму, в ньому використовуються дійсні числа, адже нахил прямої обчислюється як ділення двох відстаней, причому меншої на більшу. Але оскільки всі числа, що використовуються у формулах є цілими не комплексними числами, то Брезенхейм запропонував виконати просту арифметичну дію і помножити всі змінні в рівняннях на значення 2dx. Тоді ми отримаємо наступні формули:

- початкова помилка: $dp_0=-1/2\cdot(2dx)=-dx$;
- нахил прямої: $m=dy/dx\cdot(2dx)=2dy$;
- наступна помилка: $dp_{i+1}=dp_i+m=dp_i+2dy$;
- виправлена помилка: $dp_i=dp_i-1=dp_i-2dx$.

Тепер всі формули містять лише операції додавання або віднімання цілих чисел.

5.2.2. Приклад виконання алгоритму

Нехай заданий відрізок у віконній системі координат з початковою точкою (1,2) і кінцевою точкою (12,6) (рис. 5.11).

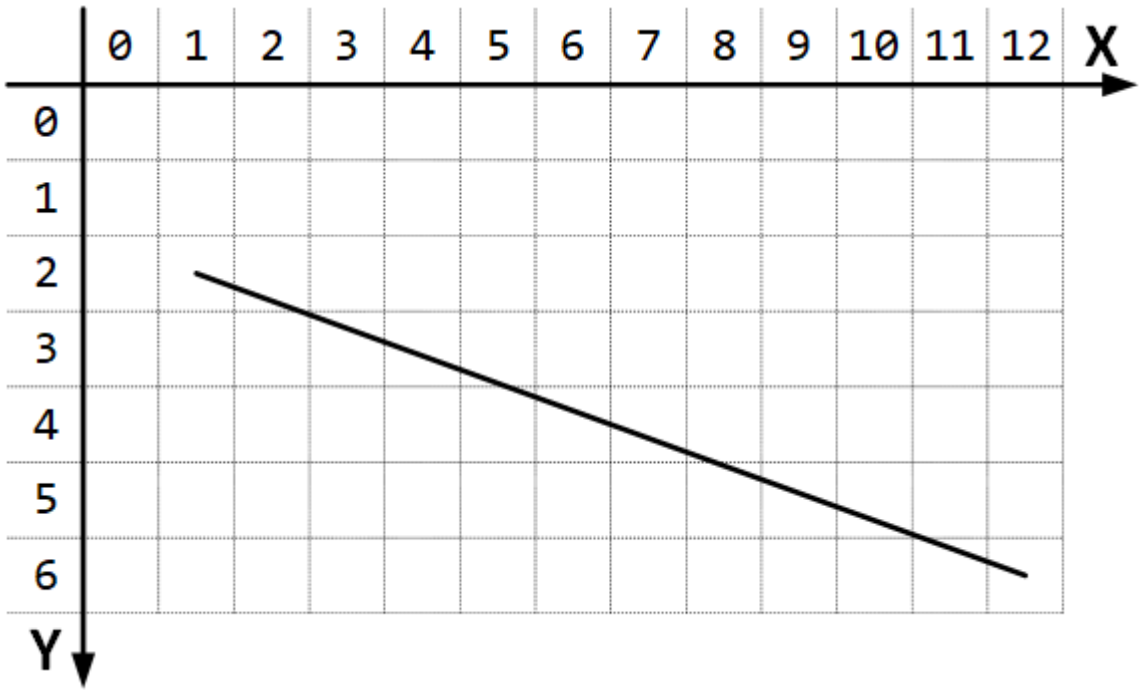


Рис. 5.11. Відрізок на піксельній решітці

Крок 1. Обчислюємо dx=(12-1)=11 та dy=(6-2)=4.

Крок 2. Обчислюємо нахил прямої m=2dy=2·4=8 та задаємо початкову помилку dp_0=-dx=-11.

Крок 3. Встановлюємо початкову точку (1,2) і запускаємо цикл за координатою x від 1 до 12 включно (рис. 5.12). x_0=1, y_0=2.

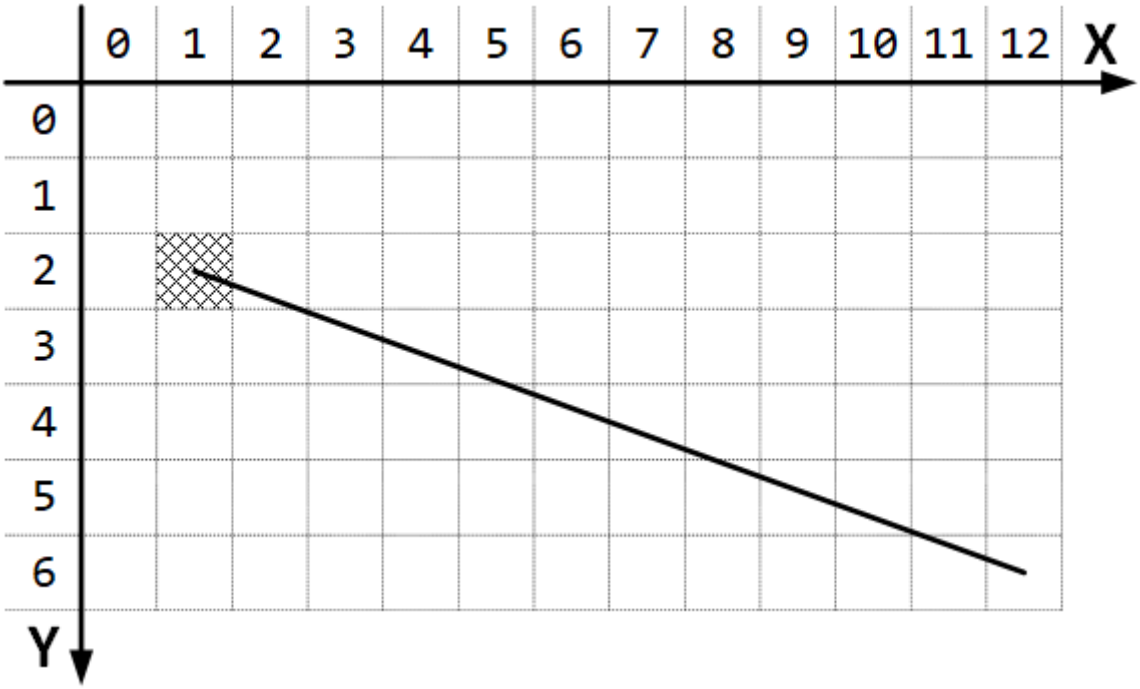


Рис. 5.12. Початкова точка на піксельній решітці

Ітерація №1. Розраховуємо оновлену помилку: dp_1=dp_0+2dy=-11+8=-3. Оскільки оновлена помилка менша за 0, то поточні координати пікселя будуть: x_1=x_0+1=1+1=2, y_1=y_0=2 (рис. 5.13).

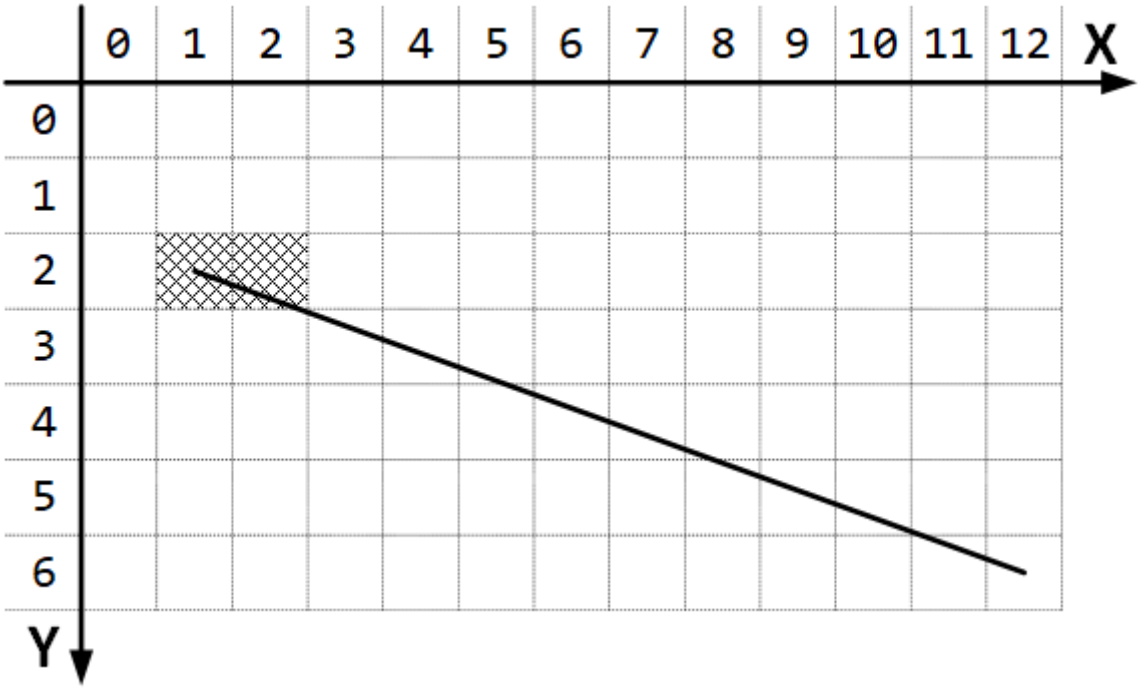


Рис. 5.13. Пікселі відрізка після ітерації №1

Ітерація №2. Розраховуємо оновлену помилку: $dp_2=dp_1+2dy=-3+8=5$. Оскільки оновлена помилка більша за 0, то поточні координати пікселя будуть: $x_2=x_1+1=2+1=3$, $y_2=y_1+1=2+1=3$ (рис. 5.14). Оскільки похибка не від'ємна, то її потрібно оновити: $dp_2=dp_2-2dx=5-22=-17$.

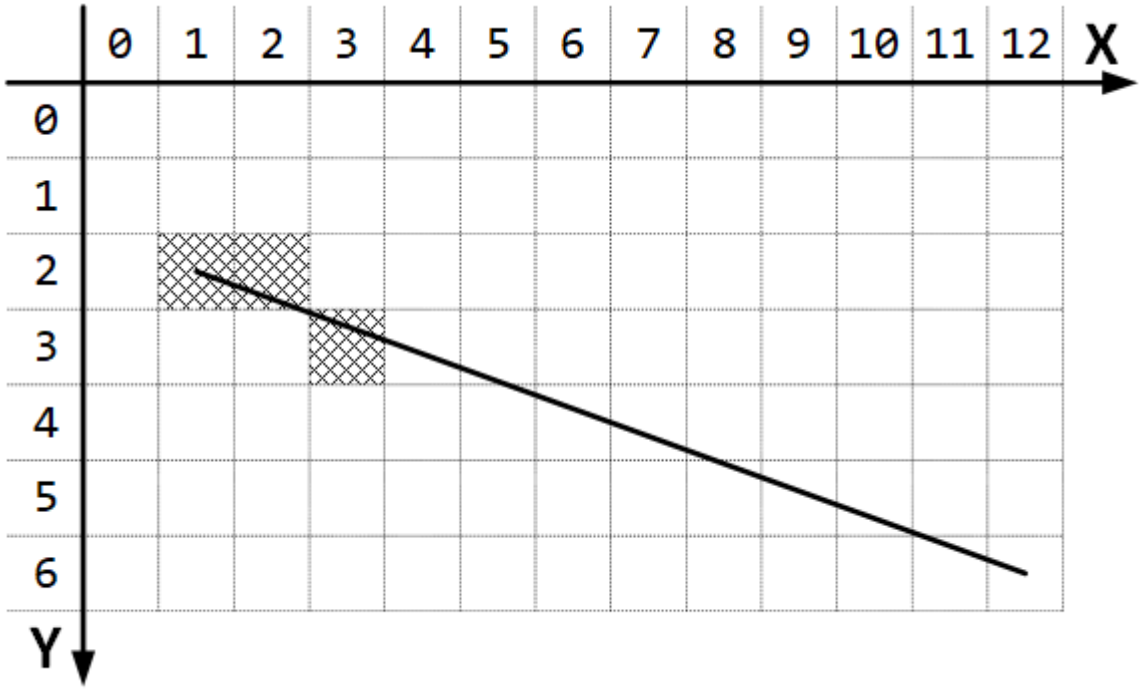


Рис. 5.14. Пікселі відрізка після ітерації №2

Ітерація №3. Розраховуємо оновлену помилку: $dp_3=dp_2+2dy=-17+8=-9$. Оскільки оновлена помилка менша за 0, то поточні координати пікселя будуть: $x_3=x_2+1=3+1=4$, $y_3=y_2=3$ (рис. 5.15).

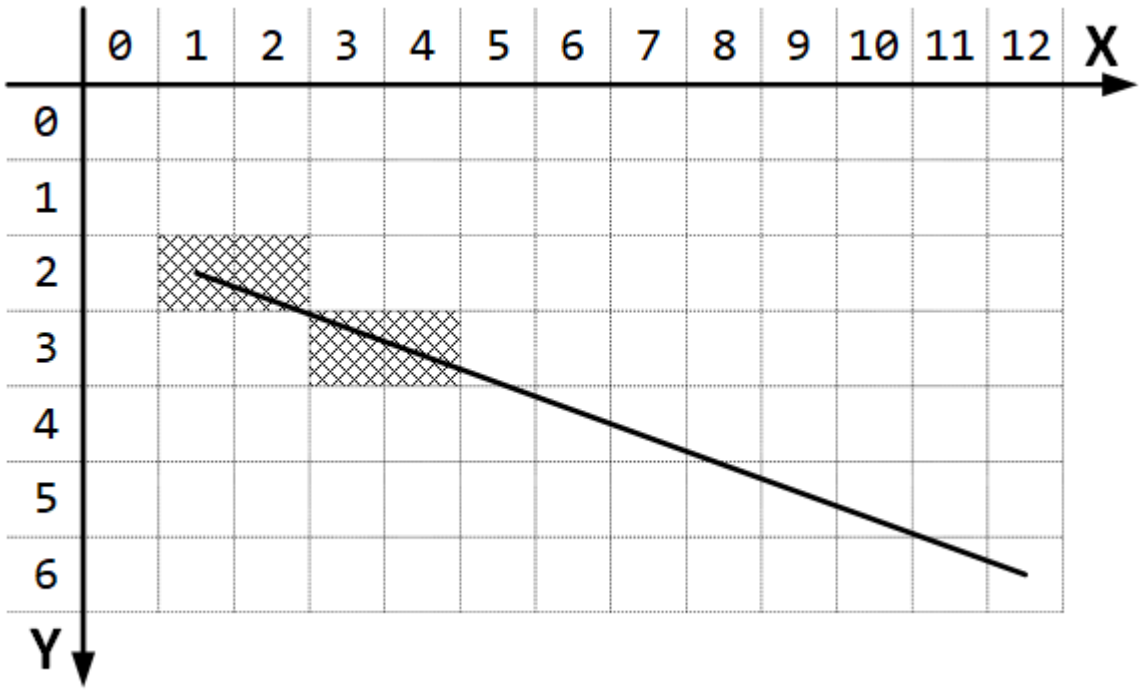


Рис. 5.15. Пікселі відрізка після ітерації №3

Ітерація №4. Розраховуємо оновлену помилку: $dp_4=dp_3+2dy=-9+8=-1$. Оскільки оновлена помилка менша за 0, то поточні координати пікселя будуть: $x_4=x_3+1=4+1=5$, $y_4=y_3=3$ (рис. 5.16).

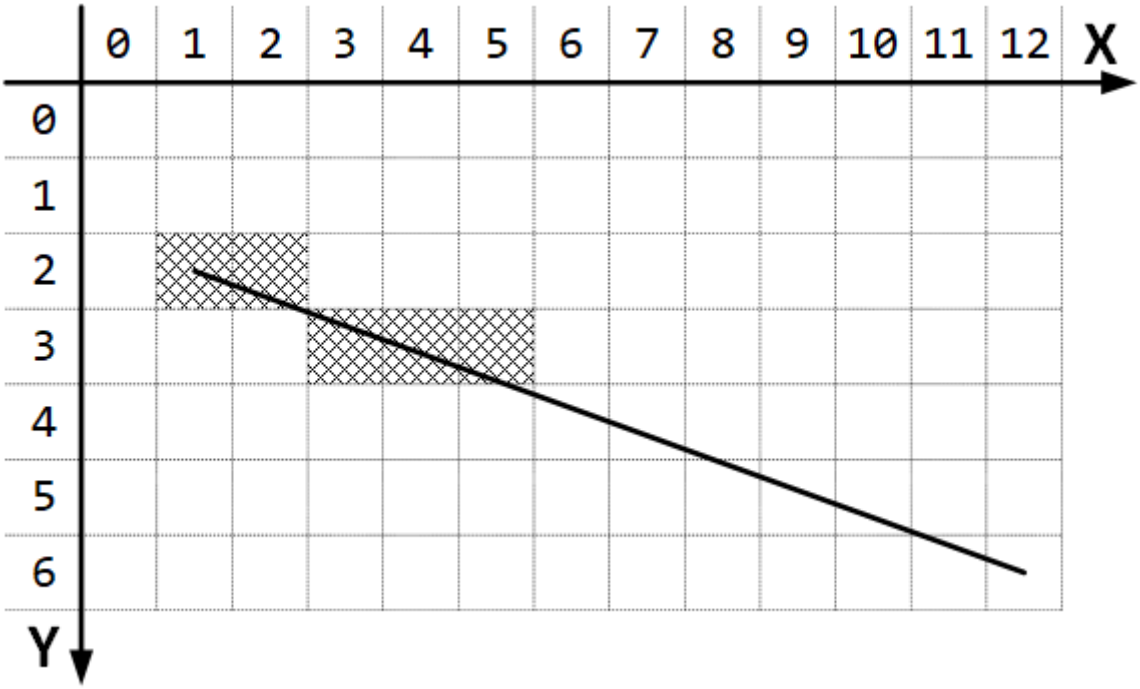


Рис. 5.16. Пікселі відрізка після ітерації №4

Ітерація №5. Розраховуємо оновлену помилку: $dp_5=dp_5+2dy=-1+8=7$. Оскільки оновлена помилка більша за 0, то поточні координати пікселя будуть: $x_5=x_4+1=5+1=6$, $y_5=y_4+1=3+1=4$ (рис. 5.17). Оскільки похибка не від'ємна, то її потрібно оновити: $dp_5=dp_5-2dx=7-22=-15$.

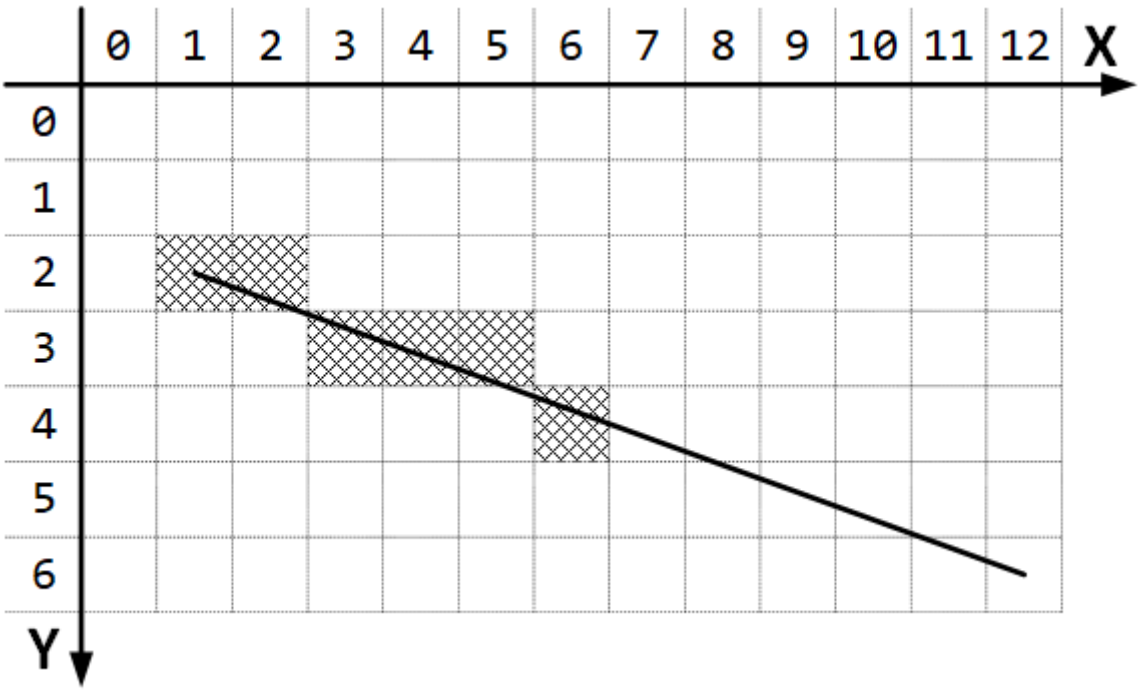


Рис. 5.17. Пікселі відрізка після ітерації №5

Ітерація №6. Розраховуємо оновлену помилку: $dp_6=dp_5+2dy=-15+8=-7$. Оскільки оновлена помилка менша за 0, то поточні координати пікселя будуть: $x_6=x_5+1=6+1=7$, $y_6=y_5=4$ (рис. 5.18).

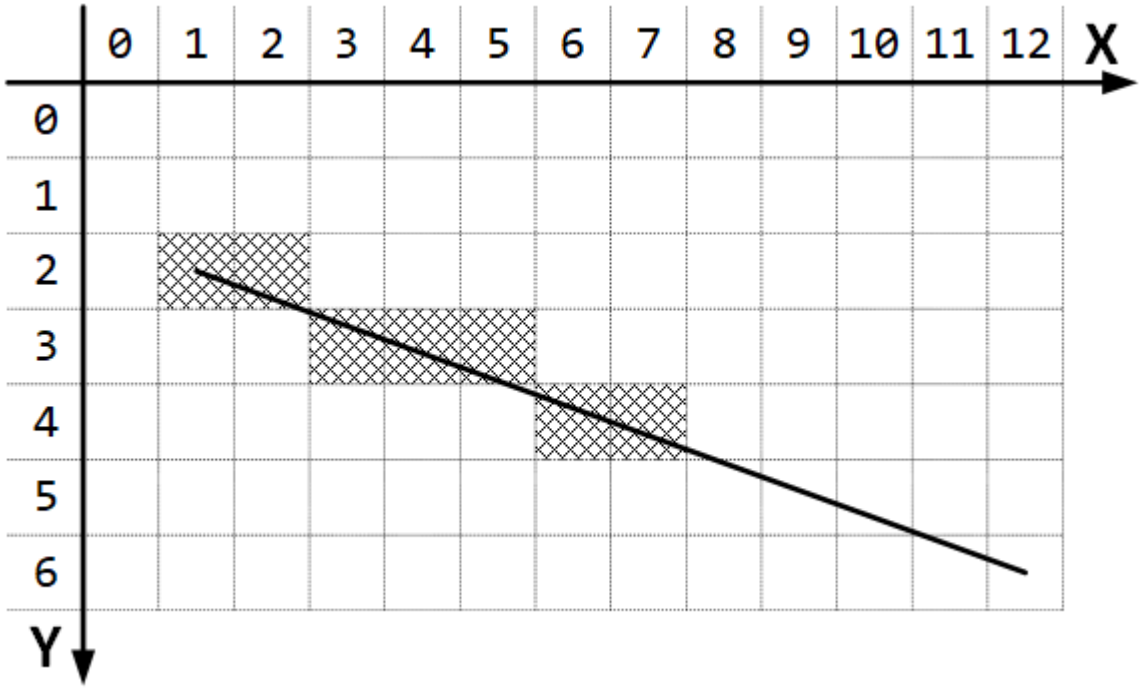


Рис. 5.18. Пікселі відрізка після ітерації №6

Ітерація №7. Розраховуємо оновлену помилку: $dp_7=dp_7+2dy=-7+8=1$. Оскільки оновлена помилка більша за 0, то поточні координати пікселя будуть: $x_7=x_6+1=7+1=8$, $y_7=y_6+1=4+1=5$ (рис. 5.19). Оскільки похибка не від'ємна, то її потрібно оновити: $dp_7=dp_7-2dx=1-22=-21$.

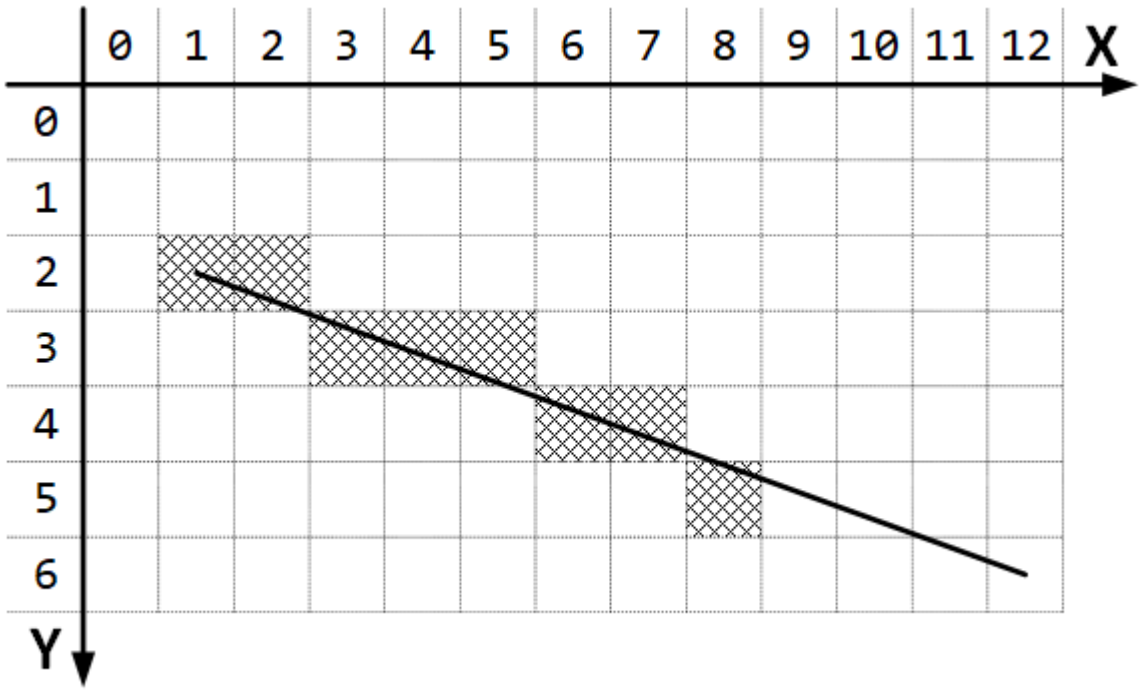


Рис. 5.19. Пікселі відрізка після ітерації №7

Ітерація №8. Розраховуємо оновлену помилку: $dp_8=dp_7+2dy=-21+8=-13$. Оскільки оновлена помилка менша за 0, то поточні координати пікселя будуть: $x_8=x_7+1=8+1=9$, $y_8=y_7=5$ (рис. 5.20).

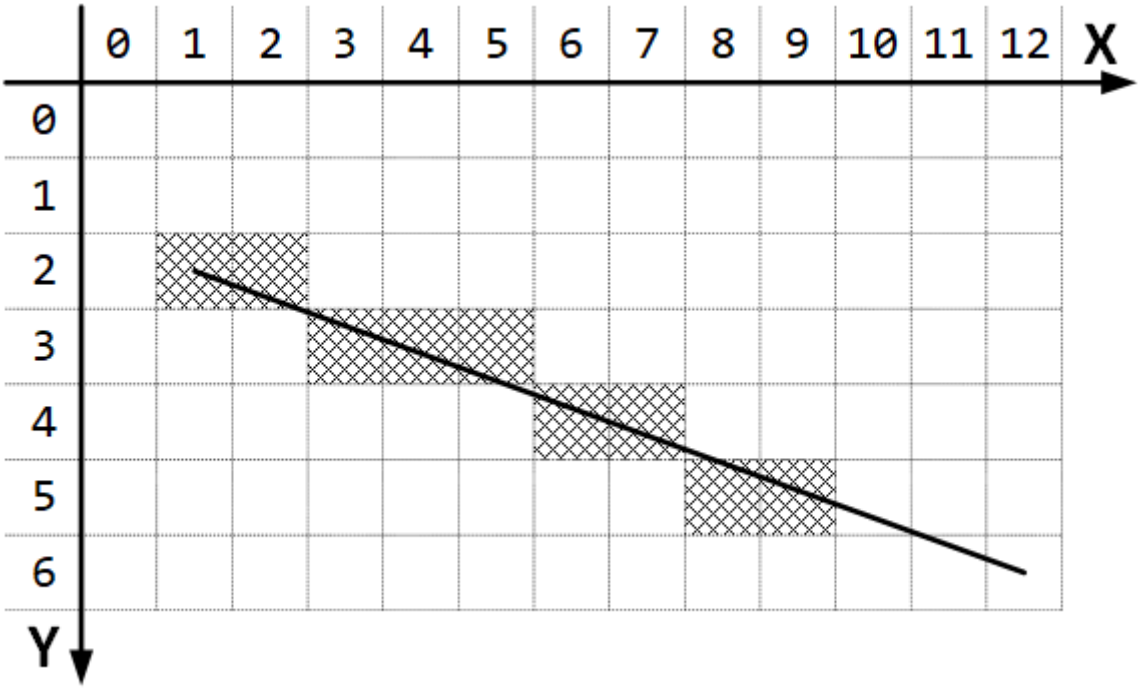


Рис. 5.20. Пікселі відрізка після ітерації №8

Ітерація №9. Розраховуємо оновлену помилку: $dp_9=dp_8+2dy=-13+8=-5$. Оскільки оновлена помилка менша за 0, то поточні координати пікселя будуть: $x_9=x_8+1=9+1=10$, $y_9=y_8=5$ (рис. 5.21).

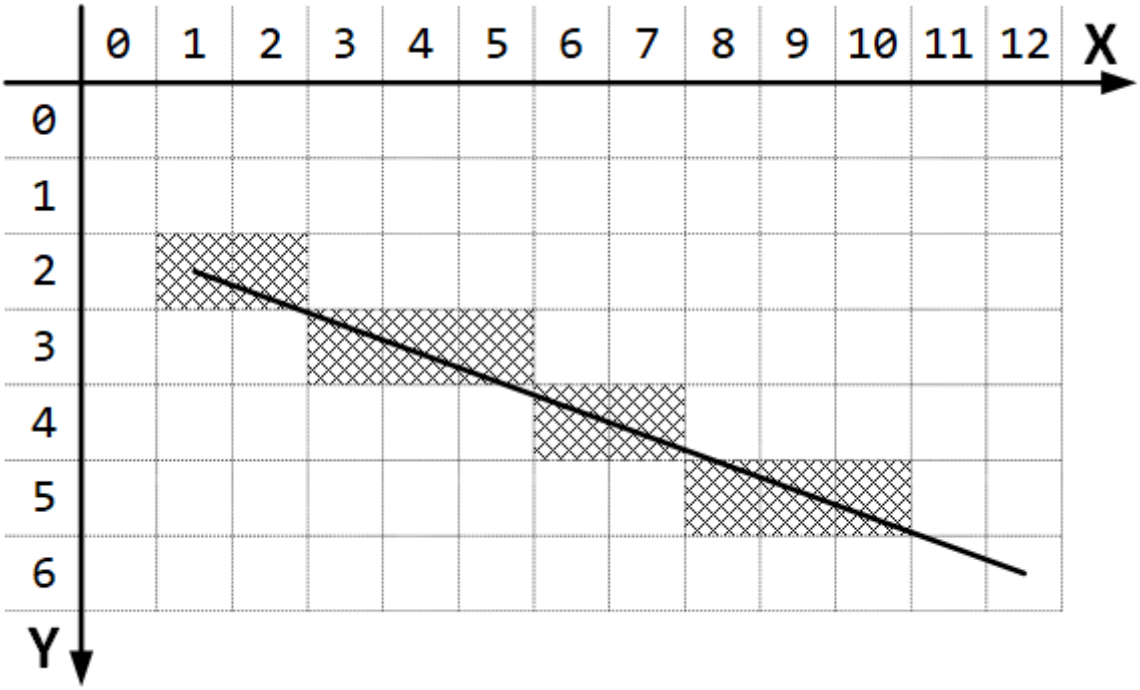


Рис. 5.21. Пікселі відрізка після ітерації №9

Ітерація №10. Розраховуємо оновлену помилку: $dp_{10}=dp_9+2dy=-5+8=3$. Оскільки оновлена помилка більша за 0, то поточні координати пікселя будуть: $x_{10}=x_9+1=10+1=11$, $y_{10}=y_9+1=5+1=6$ (рис. 5.22). Оскільки похибка не від'ємна, то її потрібно оновити: $dp_{10}=dp_{10}-2dx=3-22=-19$.

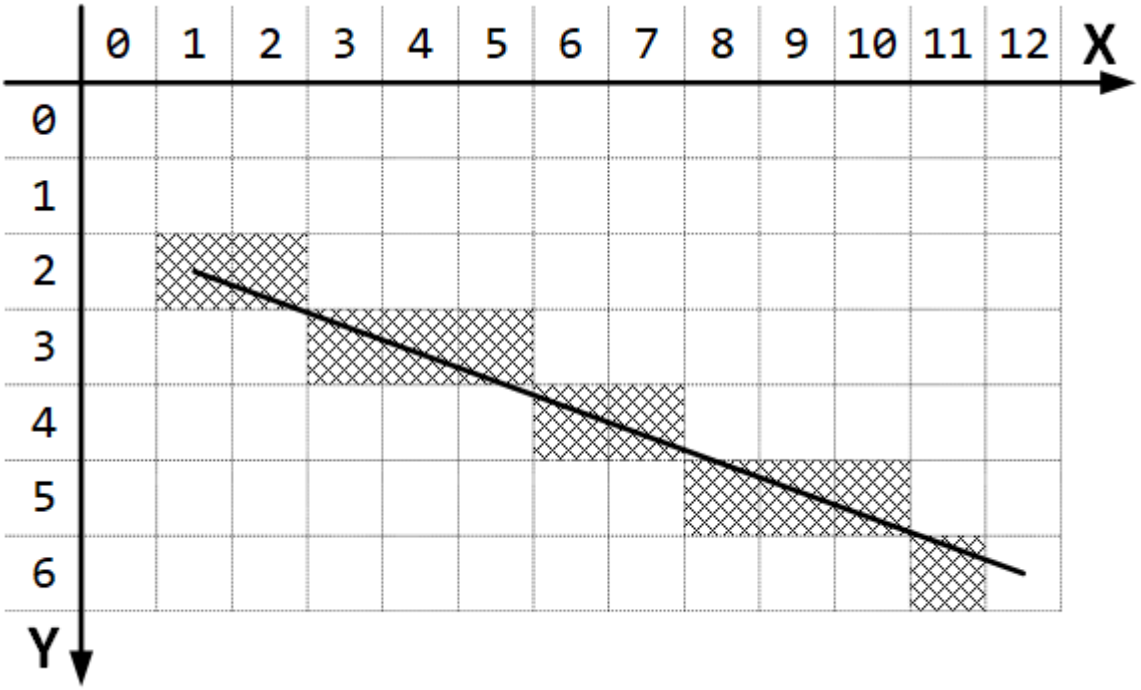


Рис. 5.22. Пікселі відрізка після ітерації №10

Ітерація №11. Розраховуємо оновлену помилку: $dp_{11}=dp_{10}+2dy=-19+8=-11$. Оскільки оновлена помилка менша за 0, то поточні координати пікселя будуть: $x_{11}=x_{10}+1=11+1=12$, $y_{11}=y_{10}=6$ (рис. 5.23).

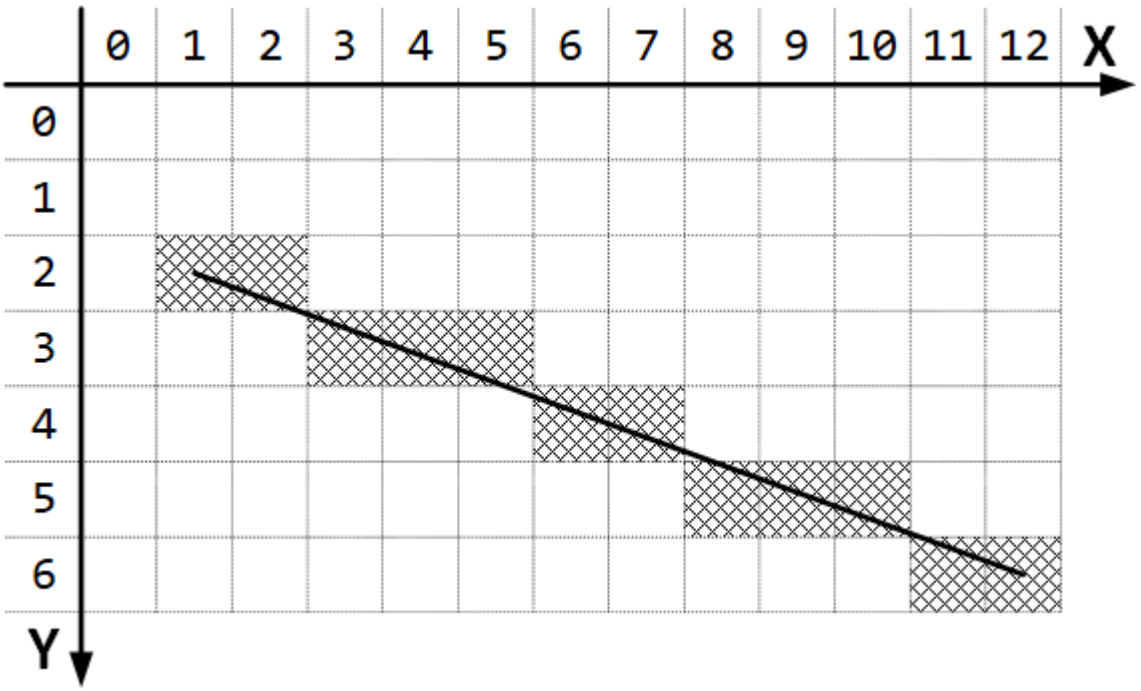


Рис. 5.23. Пікселі відрізка після ітерації №11

Алгоритм досягнув кінцевої точки відрізка, а тому закінчив свою роботу. У підсумку ми отримали пікселі, що утворюють заданий відрізок.

5.3. Алгоритм Брезенхейма для побудови кола

У растр потрібно розкладати не лише лінійні, але й інші, більш складні функції. Розкладанню конічних перерізів, тобто кіл, еліпсів, парабол, гіпербол, було присвячено значну кількість робіт. Один з найбільш ефективних і простих для розуміння алгоритмів побудови кола належить Брезенхейму. Для початку зауважимо, що згідно з цим алгоритмом, достатньо згенерувати тільки одну четверту чи навіть одну восьму частину кола. Інші його частини можуть бути отримані послідовними відображеннями.

Для пояснення алгоритму розглянемо першу чверть кола з центром у початку лівосторонньої системи координат. Зауважимо, що якщо робота алгоритму починається в точці (R,0), то під час генерації кола за годинниковою стрілкою в першому квадранті X є монотонно спадною функцією аргументу Y. Аналогічно, якщо вихідною точкою є (0,R), то під час генерації кола проти годинникової стрілки Y буде монотонно спадною функцією аргументу X. У нашому випадку вибирається генерація за годинниковою стрілкою з початком у точці (R,0). Передбачається, що центр кола і початкова точка знаходяться точно в точках растра.

Отже, для будь-якої заданої точки на колі, під час генерації за годинниковою стрілкою, існує тільки три варіанти вибрати наступний піксель, що найкращим чином наближає коло. Це здійснити елементарний крок в одному з наступних напрямків відносно до поточного `i`-го пікселя: горизонтально ліворуч (H_i), по діагоналі (S_i) та вертикально вниз (V_i) (рис. 5.24).

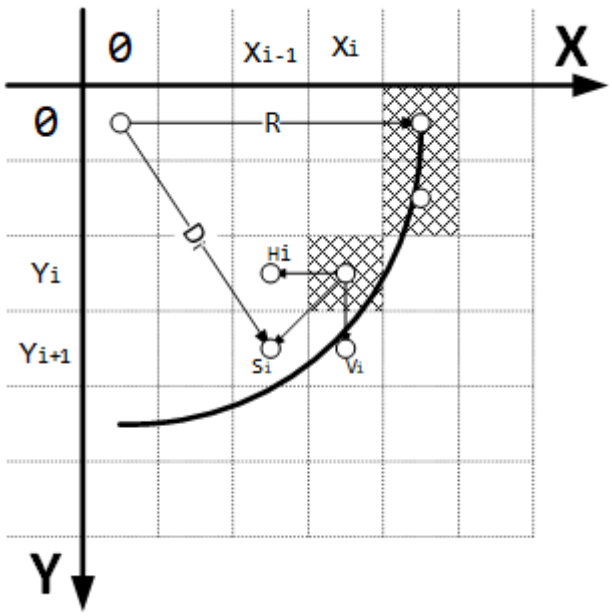


Рис. 5.24. Частина кола в першому квадранті (R — радіус кола)

Згідно з алгоритмом Брезенхейма для кола, вибирається піксель, для якого квадрат відстані між одним з цих положень і колом є мінімальним, тобто $\min(h_i,s_i,v_i)$, де $h_i=|x^2_{i-1}+y^2_i-R^2|$, $v_i=|x^2_i+y^2_{i+1}-R^2|$ і $s_i=|x^2_{i-1}+y^2_{i+1}-R^2|$.

Щоб ефективно вирішити задачу визначення наступного пікселя, спочатку перевіряється квадрат відстані до діагонального пікселя (D_i). В залежності від того, чи лежить він всередині чи зовні реального кола, додатково перевіряється різниця квадратів відстаней від кола до пікселів у горизонтальному і діагональному напрямках, або у вертикальному і діагональному напрямках. Якщо ж виявиться, що діагональний піксель лежить на колі, тоді однозначно обирається він. Але щоб зменшити кількість умовних розгалужень цей випадок можна розглядати разом із одним з двох попередніх.

Квадрат відстані до діагонального пікселя від поточного i-го пікселя визначається за формулою:

$$D_i = x_{i-1}^2 + y_{i+1}^2 - R^2. \quad (5.1)$$

Нас цікавить не абсолютне значення, а лише знак цієї величини. Якщо $D_i<0$, то діагональна точка (x_{i-1},y_{i+1}) знаходиться всередині кола. Отже ми маємо обрати між діагональним (S_i) та вертикальним (V_i) пікселями. Для цього перевіримо різницю квадратів відстаней від кола до пікселів у вертикальному і діагональному напрямках:

$$d_v = v_i - s_i = \left|x_i^2 + y_{i+1}^2 - R^2\right| - \left|x_{i-1}^2 + y_{i+1}^2 - R^2\right|. \quad (5.2)$$

Якщо $d_v<0$, то відстань від кола до діагонального пікселя більша, ніж до вертикального. І навпаки, якщо $d_v>0$ — відстань до вертикального пікселя більша. Таким чином, у разі $d_v<0$ вибираємо V_i, а за $d_v>0$ — S_i. У випадку, коли $d_v=0$, тобто коли відстань від кола до обох пікселів однакова, вибираємо вертикальний крок. Кількість обчислень, необхідних для оцінки величини d_v, можна скоротити, якщо зауважити, що $x^2_i+y^2_{i+1}-R^2\geq 0$, а $x^2_{i-1}+y^2_{i+1}-R^2<0$ у разі коли діагональний піксель лежить всередині кола, а вертикальний за межами кола. Тоді можна прибрати модуль і спростити вираз:

$$d_v = x_i^2 + y_{i+1}^2 - R^2 + x_{i-1}^2 + y_{i+1}^2 - R^2 = 2 \cdot y_{i+1}^2 + x_i^2 + x_{i-1}^2 - 2 \cdot R^2.$$

Замінимо x_i на $x_{i-1}+1$, тоді $x^2_i=(x_{i-1}+1)^2=x^2_{i-1}+2\cdot x_{i-1}+1$. Підставимо у формулу для d_v і отримаємо наступне:

$$d_v = 2 \cdot \left(x_{i-1}^2 + y_{i+1}^2 - R^2\right) + 2 \cdot x_{i-1} + 1.$$

Тепер в дужках стоїть вираз D_i і ми можемо спростити формулу повернувши x_i замість $x_{i-1}+1$:

$$d_v = 2 \cdot D_i + 2 \cdot x_{i-1} + 1 + 1 - 1 = 2 \cdot (D_i + x_i) - 1.$$

Таким чином для розрахунку значення d_v ми використовуємо вже пораховане значення D_i, що значно спрощує розрахунки.

Якщо ж вертикальний піксель також лежить в середині кола, тоді $x^2_i+y^2_{i+1}-R^2<0$ і $x^2_{i-1}+y^2_{i+1}-R^2<0$. В цьому випадку у разі $d_v<0$ вибираємо V_i, а за $d_v>0$ — S_i так само, як і раніше. Отже нам достатньо порахувати d_v за формулою:

$$d_v = 2 \cdot (D_i + x_i) - 1. \quad (5.3)$$

Якщо $D_i>0$, то діагональна точка (x_{i-1},y_{i+1}) знаходиться за межами кола. Отже ми маємо обрати між діагональним (S_i) та горизонтальним (H_i) пікселями. Для цього перевіримо різницю квадратів відстаней від кола до пікселів у горизонтальному і діагональному напрямках:

$$d_h = s_i - h_i = \left|x_{i-1}^2 + y_{i+1}^2 - R^2\right| - \left|x_{i-1}^2 + y_i^2 - R^2\right|. \quad (5.4)$$

Якщо $d_h<0$, то відстань від кола до горизонтального пікселя більша, ніж до діагонального. І навпаки, якщо $d_h>0$ — відстань до діагонального пікселя більша. Таким чином, у разі $d_h<0$ вибираємо S_i, а за $d_h>0$ — H_i. У випадку, коли $d_h=0$, тобто коли відстань від кола до обох пікселів однакова, вибираємо діагональний крок.

Застосувавши подібний підхід, що і для вертикального пікселя, отримаємо спрощену формулу:

$$d_h = 2 \cdot (D_i - y_i) - 1. \quad (5.5)$$

5.3.1. ФОРМАЛЬНИЙ ОПИС АЛГОРИТМУ

Розглянемо алгоритм для дуги кола радіусом R з центром в точці (0,0), що лежить в першій координатній чверті лівосторонньої системи координат.

1. Задати початкову точку з координатами (R,0): $x_i=R$, $y_i=0$.
2. Якщо $x_i\le0$ закінчити побудову.
3. За формулою (5.1) порахувати параметр $D_i=(x_i-1)^2+(y_i+1)^2-R^2$.
4. Якщо $D_i\le0$, то рахуємо параметр d за формулою (5.3) $d=2\cdot(D_i+x_i)-1$, а тоді в залежності від знаку d рахуємо:

- $y_i=y_i+1$, $x_i=x_i$, якщо $d\le0$;
- $y_i=y_i+1$, $x_i=x_i-1$, якщо $d>0$.

5. Якщо $D_i>0$, то рахуємо параметр d за формулою (5.5) $d=2\cdot(D_i-y_i)-1$, а тоді в залежності від знаку d рахуємо:

- $y_i=y_i+1$, $x_i=x_i-1$, якщо $d\le0$;
- $y_i=y_i$, $x_i=x_i-1$, якщо $d>0$.

6. Повторюємо з кроку 2.

Після побудови цієї чверті кола її можна віддзеркалити відносно однієї з осей координат і отримати півколо. Потім віддзеркалити півколо і отримати повне коло.

5.3.2. *Приклад виконання алгоритму*

Розглянемо приклад побудови чверті кола з радіусом 4. В якості початкової вибираємо точку (4,0) і рухаємося за годинниковою стрілкою. Поточний піксель $P_1(4,0)$ його ми точно зафарбовуємо. Наступними кандидатами є три пікселі: горизонтальний ліворуч $H_1(3,0)$, діагональний $S_1(3,1)$ та вертикальний вниз $V_1(4,1)$ (рис. 5.25).

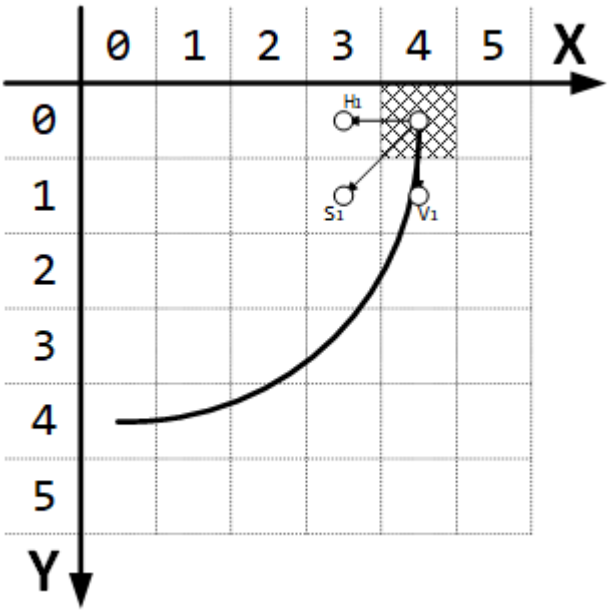


Рис. 5.25. Вибір координат пікселя P_2

Ітерація 1. Для того, щоб з'ясувати, який з цих пікселів вибрати, на першому кроці, скористаємось формулою (5.1), і, таким чином, обчислимо різницю між квадратами відстаней від центра кола до діагонального пікселя $S_1(3,1)$ і від центра до точки на колі R^2. Унаслідок отримаємо:

$$D_1 = 3^2 + 1^2 - 4^2 = 9 + 1 - 16 = -6.$$

Оскільки значення $D_1<0$, (діагональний піксель знаходиться всередині кола), то в якості наступного слід вибрати або піксель $S_1(3,1)$ або піксель $V_1(4,1)$. Для того, щоб визначити, який з цих пікселів найкращим чином наближає коло переходимо до кроку номер два, згідно з яким, за формулою (5.3), обчислюємо різницю квадратів відстаней від кола до пікселів у вертикальному та діагональному напрямках:

$$d = 2 \cdot (D_1 + 4) - 1 = -12 + 8 - 1 = -5.$$

Отримане значення менше за 0, а отже наступним пікселем обираємо вертикальний піксель $P_2=V_1(4,1)$.

Ітерація 2. Для визначення координат точки P_3, аналогічним чином, в якості кандидатів розглядаємо пікселі, що розташовані горизонтально ліворуч, по діагоналі та вертикально вниз відносно точки P_2 (рис. 5.26).

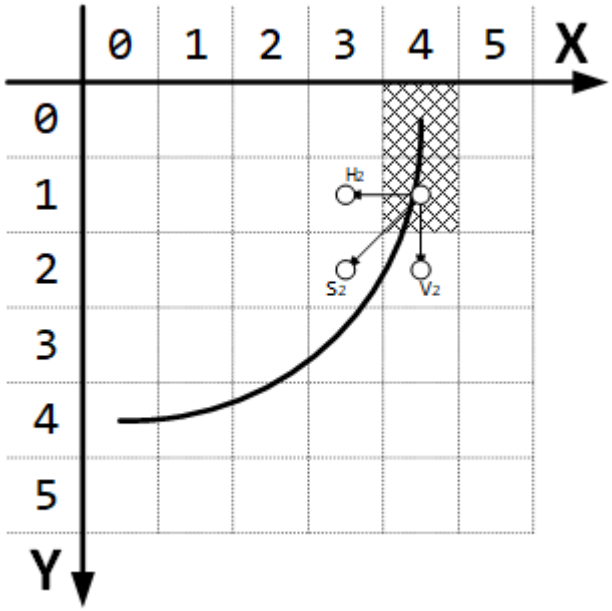


Рис. 5.26. Вибір координат пікселя P_3

Знову за формулою (5.1) рахуємо різницю між квадратами відстаней від центра кола до діагонального пікселя $S_2(3,2)$ і від центра до точки на колі R^2. Унаслідок отримаємо:

$$D_2 = 3^2 + 2^2 - 4^2 = 9 + 4 - 16 = -3.$$

Оскільки значення $D_2<0$, (діагональний піксель знаходиться всередині кола), то в якості наступного слід вибрати або піксель $S_2(3,2)$ або піксель $V_2(4,2)$. Для того, щоб визначити, який з цих пікселів найкращим чином наближає коло переходимо до кроку номер два, згідно з яким, за формулою (5.3), обчислюємо різницю квадратів відстаней від кола до пікселів у вертикальному та діагональному напрямках:

$$d = 2 \cdot (D_2 + 4) - 1 = -6 + 8 - 1 = 1.$$

Отримане значення більше за 0, а отже наступним пікселем обираємо діагональний піксель $P_3=S_2(3,2)$.

Ітерація 3. Для визначення координат точки P_4, в якості кандидатів розглядаємо пікселі, що розташовані горизонтально ліворуч, по діагоналі та вертикально вниз відносно точки P_3 (рис. 5.27).

Знову за формулою (5.1) рахуємо різницю між квадратами відстаней від центра кола до діагонального пікселя $S_3(2,3)$ і від центра до точки на колі R^2. Унаслідок отримаємо:

$$D_3 = 2^2 + 3^2 - 4^2 = 4 + 9 - 16 = -3.$$

Оскільки значення $D_3<0$, (діагональний піксель знаходиться всередині кола), то в якості наступного слід вибрати або піксель $S_3(2,3)$ або піксель $V_3(3,3)$. Для того, щоб визначити, який з цих пікселів найкращим чином наближає коло переходимо до кроку номер два, згідно з яким, за формулою (5.3), обчислюємо

різницю квадратів відстаней від кола до пікселів у вертикальному та діагональному напрямках:

$$d = 2 \cdot (D_3 + 3) - 1 = -6 + 6 - 1 = -1.$$

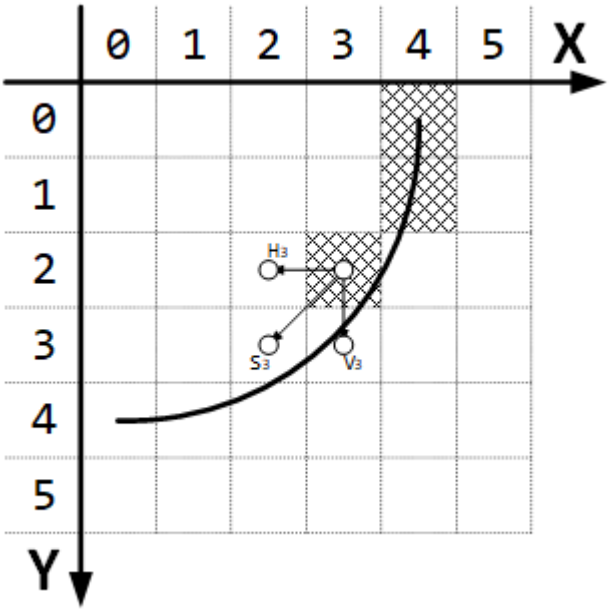


Рис. 5.27. Вибір координат пікселя P_4

Отримане значення менше за 0, а отже наступним пікселем обираємо вертикальний піксель P_4=V_3(3,3).

Ітерація 4. Для визначення координат точки P_5, в якості кандидатів розглядаємо пікселі, що розташовані горизонтально ліворуч, по діагоналі та вертикально вниз відносно точки P_4 (рис. 5.28).

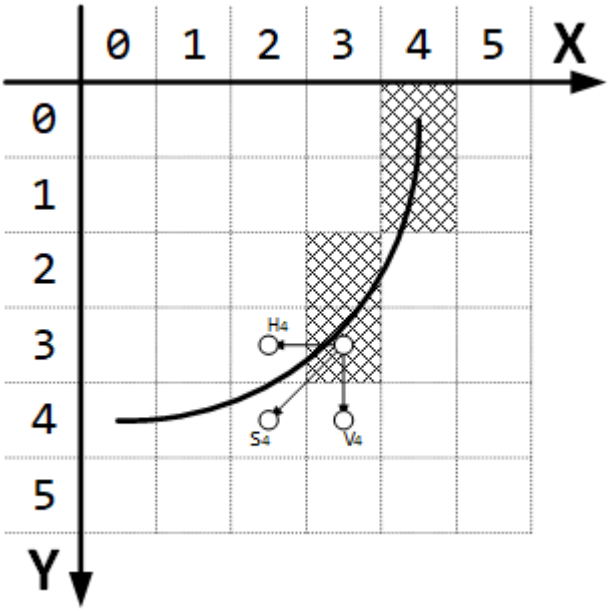


Рис. 5.28. Вибір координат пікселя P_5

Знову за формулою (5.1) рахуємо різницю між квадратами відстаней від центра кола до діагонального пікселя $S_4(2,4)$ і від центра до точки на колі R^2. Унаслідок отримаємо:

$$D_4 = 2^2 + 4^2 - 4^2 = 4 + 16 - 16 = 4.$$

Оскільки значення $D_4>0$, (діагональний піксель знаходиться за межею кола), то в якості наступного слід вибрати або піксель $S_4(2,4)$ або піксель $H_4(2,3)$. Для того, щоб визначити, який з цих пікселів найкращим чином наближає коло переходимо до кроку номер два, згідно з яким, за формулою (5.5), обчислюємо різницю квадратів відстаней від кола до пікселів у горизонтальному та діагональному напрямках:

$$d = 2 \cdot (D_4 - 3) - 1 = 8 - 6 - 1 = 1.$$

Отримане значення більше за 0, а отже наступним пікселем обираємо горизонтальний піксель $P_5=H_4(2,3)$.

Ітерація 5. Для визначення координат точки P_6, в якості кандидатів розглядаємо пікселі, що розташовані горизонтально ліворуч, по діагоналі та вертикально вниз відносно точки P_5 (рис. 5.29).

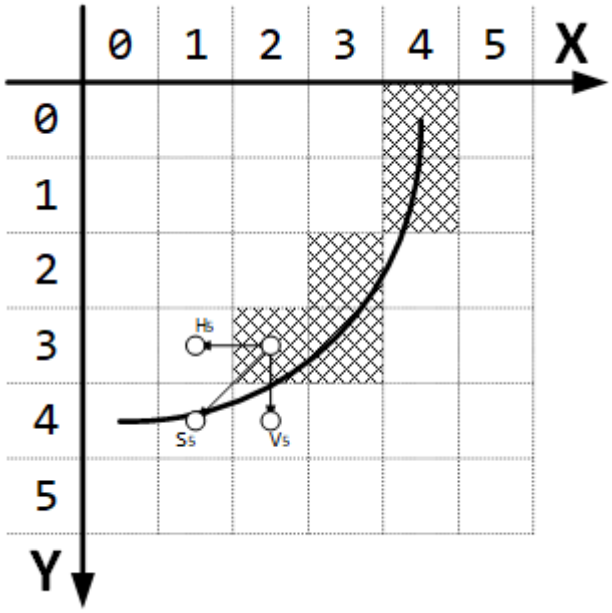


Рис. 5.29. Вибір координат пікселя P_6

Знову за формулою (5.1) рахуємо різницю між квадратами відстаней від центра кола до діагонального пікселя $S_5(1,4)$ і від центра до точки на колі R^2. Унаслідок отримаємо:

$$D_5 = 1^2 + 4^2 - 4^2 = 1 + 16 - 16 = 1.$$

Оскільки значення $D_5>0$, (діагональний піксель знаходиться за межею кола), то в якості наступного слід вибрати або піксель $S_5(1,4)$ або піксель $H_5(1,3)$. Для того, щоб визначити, який з цих пікселів найкращим чином наближає коло переходимо до кроку номер два, згідно з яким, за формулою (5.5), обчислюємо

різницю квадратів відстаней від кола до пікселів у горизонтальному та діагональному напрямках:

$$d = 2 \cdot (D_5 - 3) - 1 = 2 - 6 - 1 = -5.$$

Отримане значення менше за 0, а отже наступним пікселем обираємо діагональний піксель $P_6=S_5(1,4)$.

Ітерація 6. Для визначення координат точки P_7, в якості кандидатів розглядаємо пікселі, що розташовані горизонтально ліворуч, по діагоналі та вертикально вниз відносно точки P_6 (рис. 5.30).

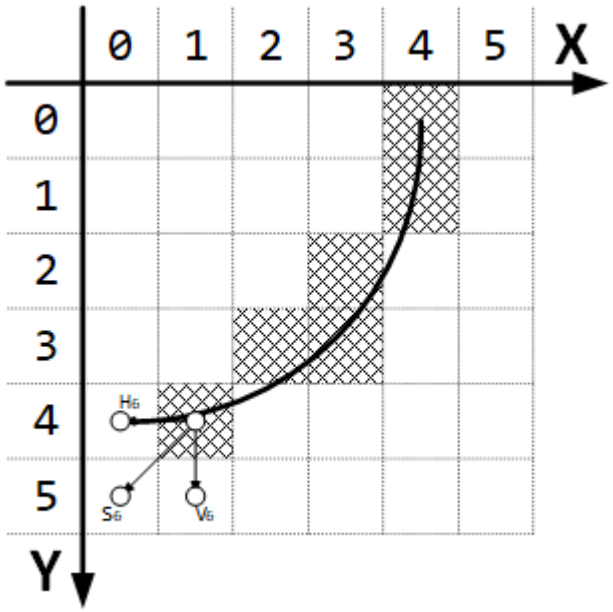


Рис. 5.30. Вибір координат пікселя P_7

Знову за формулою (5.1) рахуємо різницю між квадратами відстаней від центра кола до діагонального пікселя $S_6(0,5)$ і від центра до точки на колі R^2. Унаслідок отримаємо:

$$D_6 = 0^2 + 5^2 - 4^2 = 0 + 25 - 16 = 9.$$

Оскільки значення $D_6>0$, (діагональний піксель знаходиться за межею кола), то в якості наступного слід вибрати або піксель $S_6(0,5)$ або піксель $H_6(0,4)$. Для того, щоб визначити, який з цих пікселів найкращим чином наближає коло переходимо до кроку номер два, згідно з яким, за формулою (5.5), обчислюємо різницю квадратів відстаней від кола до пікселів у горизонтальному та діагональному напрямках:

$$d = 2 \cdot (D_6 - 4) - 1 = 18 - 8 - 1 = 9.$$

Отримане значення більше за 0, а отже наступним пікселем обираємо горизонтальний піксель $P_7=H_6(0,4)$.

Значення координати X стало дорівнювати 0, а отже алгоритм закінчив свою роботу і ми побудували чверть кола в першому квадранті (рис. 5.31).

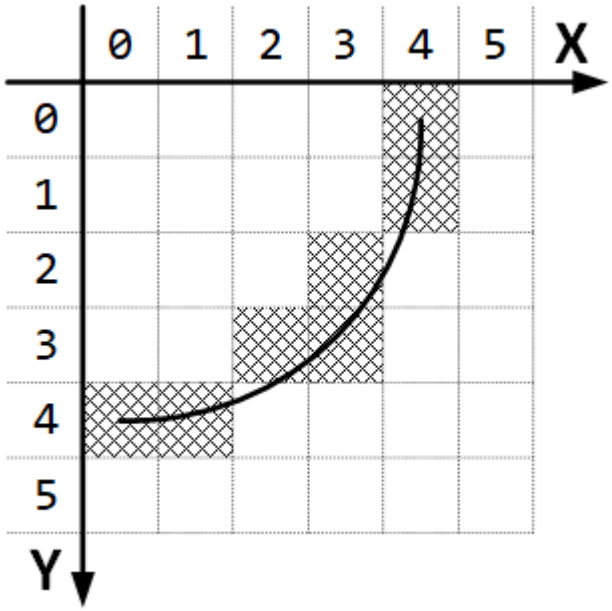


Рис. 5.31. Результат побудови чверті кола

Як вже зазначалося, для побудови повного кола доцільно використати операцію віддзеркалення побудованої чверті кола.

Запитання для самоконтролю

1. Навіщо потрібні алгоритми растеризації?
2. Які алгоритми побудови відрізка ви знаєте?
3. В чому перевага алгоритму Брезенхейма над алгоритмом ЦДА?
4. Для побудови яких фігур можна модифікувати алгоритм Брезенхейма?
5. Яку мінімальну частину кола достатньо побудувати алгоритмом Брезенхейма щоб відтворити повне коло?

6. Опуклі оболонки

Опукла оболонка множини точок S це найменша опукла множина, що містить S. Щоб наглядно уявити собі це поняття для кінцевої множини точок S припустимо, що ця множина охоплена великою розтягнутою гумовою стрічкою. Коли стрічка звільняється, то вона приймає форму опуклої оболонки.

6.1. Алгоритм побудови опуклої оболонки на площині

Для того, щоб знайти опуклу оболонку кінцевої множини точок, потрібно виконати наступні два кроки.

1. Визначити крайні точки.
2. Упорядкувати ці точки так, щоб вони утворювали опуклий багатокутник.

Визначення. Точка p опуклої множини S називається крайньою, якщо не існує пари точок (a, b) $\in$ S таких, що p лежить на відкритому відрізку ab.

Необхідна теорема, яка дозволить нам перевіряти, чи є деяка точка крайньою.

Теорема. Точка p не є крайньою плоскої випуклої множини S лише тоді, коли вона лежить в деякому трикутнику, вершинами якого є точки з S, але сама вона не є вершиною цього трикутника (рис. 6.1).

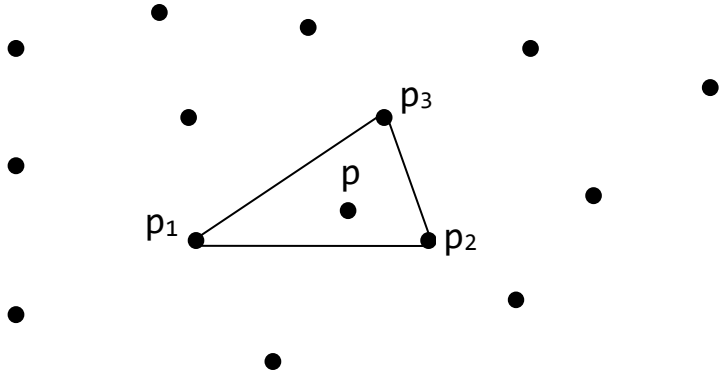


Рис. 6.1. Ілюстрація до визначення не крайньої точки

Ця теорема подає ідею алгоритму видалення точок, які не є крайніми. Є $O(N^3)$ трикутників, що визначаються N точками множини S. Перевірка належності точки заданому трикутнику може бути виконана за деяке постійне число операцій, так що за час $O(N^3)$ можна визначити, чи є точка крайньою. Повторення цієї процедури для всіх N точок множини S вимагає часу $O(N^4)$. Хоча такий алгоритм неефективний, він простий в ідейному плані і показує, що визначення крайніх точок може бути виконане за кінцеву кількість кроків.

Коли крайні точки визначені, потрібно їх якось впорядкувати, щоб отримати опуклу оболонку. Сенс цього порядку визначається наступними теоремами.

Теорема. Промінь, що виходить з внутрішньої точки обмеженої опуклої фігури F, перетинає кордон F точно в одній точці.

Теорема. Послідовні вершин опуклого багатокутника розташовуються в порядку, що відповідає зміні кута відносно будь-якої внутрішньої точки.

Уявіть промінь, що виходить з деякої внутрішньої точки q багатокутника P і що обходить вершини багатокутника P в порядку руху проти годинникової стрілки, починаючи з положення, що співпадає за напрямом з позитивним напрямом осі X системи координат. Протягом руху від вершини до вершини полярний кут променю монотонно збільшується. Саме це малося на увазі, коли говорилося про те, що вершини багатокутника P «впорядковані» (рис. 6.2).

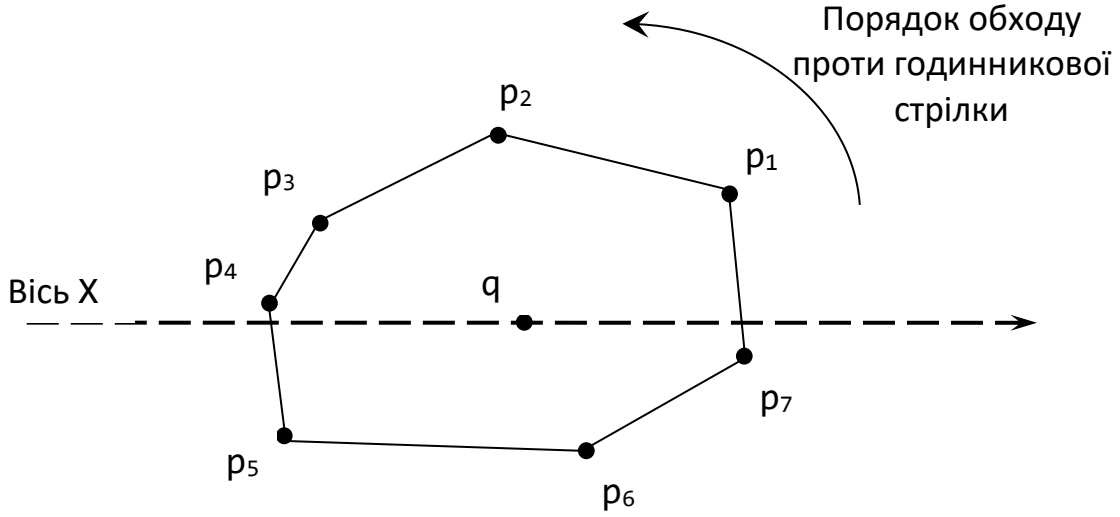


Рис. 6.2. Вершини багатокутника P впорядковані відносно точки q

Якщо задані крайні точки деякої множини, то його випуклу оболонку можна знайти, вибравши точку q, про яку відомо, що вона є внутрішньою точкою оболонки, і упорядкувавши потім крайні точки відповідно до полярного кута відносно q. У якості точки q можна взяти центроїд множини крайніх точок: добре відомо, що центроїд множини точок є внутрішньою точкою опуклої оболонки.

Центроїд множини з N точок в двовимірному просторі може бути легко обчислений за O(N*k) арифметичних операцій. Грехем запропонував інший метод знаходження внутрішньої точки, відмітивши, що цілком достатньо узяти центроїд будь-яких трьох не колінеарних точок.

6.2. Алгоритм Грехема

Алгоритм з часом виконання $O(N^4)$ не дозволить обробляти дуже великі набори даних. Якщо часові характеристики повинні бути покращені, то це можна зробити, або усунувши надлишкові обчислення в наявному алгоритмі, або вибравши інший теоретичний підхід. У цьому підрозділі ми досліджуємо наш алгоритм з точки зору наявності в ньому непотрібних обчислень.

6.2.1. Теоретичні аспекти алгоритму

Чи так необхідно перевіряти всі трикутники, що визначені множиною з N точок, щоб дізнатися, чи лежить деяка точка в якомусь із них? Якщо ні, то є надія, що

крайні точки можна знайти за час, менший ніж $O(N^4)$. Грехем в одній з перших робіт у 1972 році, спеціально присвячених питанню розробки ефективних геометричних алгоритмів, показав, що виконавши заздалегідь сортування точок, крайні точки можна знайти за лінійний час. Використаний ним метод став дуже потужним засобом в області обчислювальної геометрії.

Припустимо, що внутрішня точка вже знайдена, а координати інших точок тривіальним чином перетворені так, що знайдена внутрішня точка виявилася на початку координат. Упорядкуємо лексикографічно N точок відповідно до значення полярного кута і відстані від початку координат.

Під час виконання сортування не потрібно обчислювати дійсну відстань між двома точками, оскільки потрібно лише порівняти дві величини. Можна працювати з квадратом відстані, уникаючи тим самим операції видобування квадратного кореня, але даний випадок ще простіший. Порівняння відстаней необхідно виконувати лише у випадку, якщо дві точки мають один і той же полярний кут. Але тоді вони лежать на одній прямій з початком координат, і порівняння в цьому випадку тривіальне.

Подавши впорядковані точки у вигляді двічі зв'язаного кільцевого списку отримуємо ситуацію, що показана на рис. 6.3. Зверніть увагу: якщо точка не є вершиною опуклої оболонки, то вона є внутрішньою точкою для деякого трикутника (0pq), де p і q — послідовні вершини опуклої оболонки. Суть алгоритму Грехема полягає в однократному перегляді впорядкованої послідовності точок, в процесі якого видаляються внутрішні точки. Ті точки, які залишилися є вершинами опуклої оболонки, що подані в потрібному порядку.

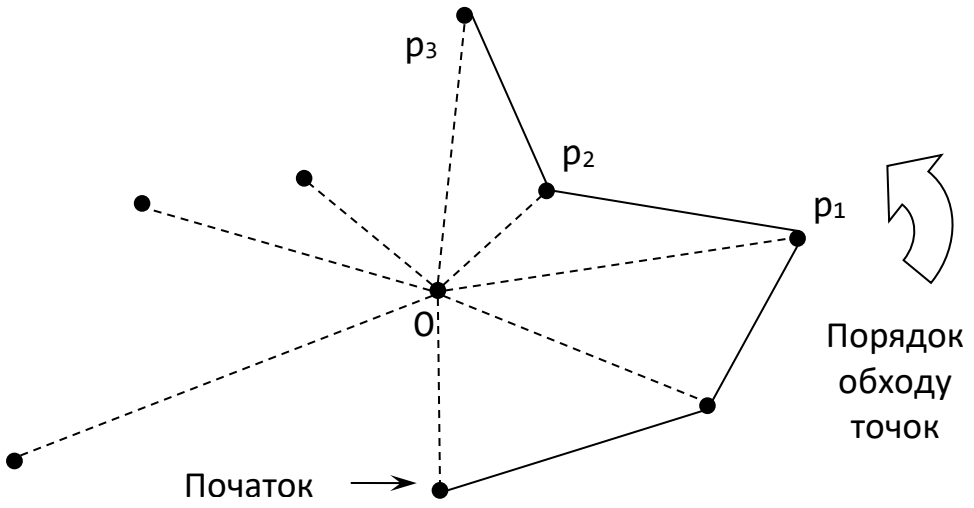


Рис. 6.3. Ілюстрація до алгоритму Грехема

Перегляд починається з точки, що помічена як початок, в якості якої можна взяти найбільш праву з найменшою ординатою точку з даної множини, яка завідомо є вершиною опуклої оболонки. Трійки послідовних точок багато разів перевіряються в порядку обходу проти годинникової стрілки з метою визначити, утворюють чи ні вони кут, більший або рівний π. Якщо внутрішній кут $p_1p_2p_3$

більше або рівний π, то говорять, що $p_1p_2p_3$ утворюють «правий поворот», інакше вони утворюють «лівий поворот».

З опуклості багатокутника безпосередньо виходить, що при його обході робитимуться лише ліві повороти. Якщо $p_1p_2p_3$ утворюють правий поворот, то p_2 не може бути крайньою точкою, оскільки вона є внутрішньою для трикутника (Op_1p_3). Залежно від результату перевірки кута, що утворюється поточною трійкою точок, можливі два варіанти продовження перегляду:

- $p_1p_2p_3$ утворюють правий поворот. Видалити вершину p_2 і перевірити трійку $p_0p_1p_3$.
- $p_1p_2p_3$ утворюють лівий поворот. Продовжити перегляд, перейшовши до перевірки трійки $p_2p_3p_4$.

Перегляд завершується тоді, коли обійшовши всі вершини, знову приходимо у вершину початок. Відмітимо, що вершина початок ніколи не видаляється, оскільки вона є крайньою точкою і тому при відході назад після видалення точок, ми не зможемо піти далі за точку, попередню точці початок. Простий аналіз показує, що такий перегляд виконується лише за лінійний час. Перевірка кута може бути виконана за фіксоване (постійне) число операцій. Після кожної перевірки відбувається або просування на одну точку (випадок 2), або видаляється точка (випадок 1).

Оскільки множина містить лише N точок, то просування вперед не може відбуватися більш N разів, як не може бути видалено і більше, ніж N точок. Розглянутий метод обходу контуру багатокутника є настільки корисним, що надалі ми називатимемо його алгоритмом обходу Грехема (анг. Graham Scan). Опукла оболонка N точок на площині може бути знайдена за час O(N log N) з просторовою складністю O(N) та використанням лише арифметичних операцій і порівнянь.

6.2.2. Модифікація Ендрю

В алгоритмі Грехема використовуються полярні координати, що може бути не зручно в деяких системах. Ендрю запропонував алгоритм, що дозволяє запобігти цьому.

Нехай на площині задана множина з N точок. Визначимо спочатку його ліву і праву крайні точки l і r (рис. 6.4) та побудуємо пряму, що проходить через ці точки. Точки, що залишилися, розбиваються на дві підмножини (нижню і верхню) залежно від того, по яку сторону від прямої вони розташовуються — нижче або вище прямої. Нижня підмножина породжує ламану (нижню оболонку, або H-оболонку), монотонну відносно осі X. Так само верхня підмножина породжує аналогічну ламану (верхню оболонку, або B-оболонку), а об'єднання цих двох ламаних дає опуклу оболонку заданої множини.

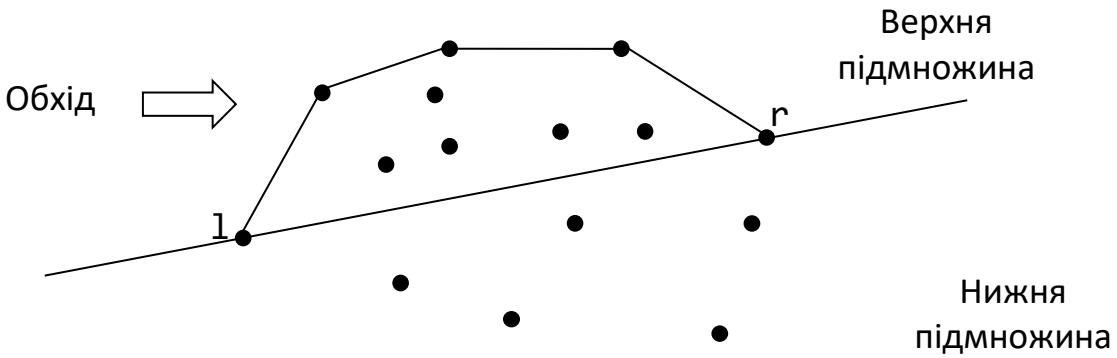


Рис. 6.4. Ілюстрація до алгоритму Ендрю

Розглянемо побудову верхньої оболонки. Точки впорядковуються відповідно до зростання абсциси і до отриманої послідовності застосовується метод обходу Грехема. За такого підходу відпадає необхідність в тригонометричних операціях. Відмітимо, що пропонований підхід є не що інше, як окремий випадок використання початкового методу Грехема, коли точка q, що співпадає з початком координат, вибирається нескінченно віддаленою у від'ємному напрямку ($-\infty$) по осі Y, так що в цьому випадку впорядкованість за абсцисою збігається з впорядкованістю за полярним кутом. Не дивлячись на те що, як було показано, алгоритм Грехема є оптимальним, як і раніше є багато причин для продовження дослідження задачі про опуклу оболонку.

1. Розглянутий алгоритм є оптимальним в найгіршому випадку, але ми не вивчили його поведінку в середньому.
2. Алгоритм застосовний лише для двовимірного простору і не має узагальнення на випадок просторів вищої розмірності.
3. Алгоритм не є відкритим алгоритмом, оскільки всі точки множини мають бути відомі до початку роботи алгоритму.
4. За можливості паралельної обробки більш ефективним є рекурсивний алгоритм, що допускає розбиття початкового завдання і даних на менші підзадачі.

6.2.3. Формальний опис алгоритму

Грехем у своїй роботі виклав теоретичні аспекти алгоритму, а тепер ми розглянемо яким чином його реалізувати практично. Деякі кроки оригінального алгоритму можна опустити під час реалізації, а сортування за полярними кутами можна виконати без знаходження самих кутів. Виконання алгоритму складається з таких п'яти кроків.

1. Серед заданої множини точок знайти точку P_0, з найменшою координатою Y. Якщо таких точок декілька, то обрати серед них ту, що має найменшу координату X. Цей крок має часову складність O(N).
2. Відсортувати всі точки за полярним кутом відносно початкової точки. Значення кутів не потрібно рахувати — нас цікавить лише орієнтація точок

одна відносно одної, тобто яка з двох точок, що розглядаються, утворює більший кут з віссю X. Тут ми можемо використати будь-який алгоритм сортування з часовою складністю O(N log N).

3. Перевіряємо відсортований список на наявність колінеарних точок. Якщо знайдені колінеарні точки, то залишаємо в списку лише точку, що є найбільш віддаленою від початкової точки P_0. Цей крок має часову складність O(N).
4. Перші дві точки у відсортованому списку завжди є точками опуклої оболонки. Створюємо стек для точок опуклої оболонки і додаємо в нього перші дві точки зі списку.
5. Для кожної наступної точки в списку перевіряємо який поворот вона утворює з двома точками на вершині стеку. Якщо поворот лівий, то додаємо поточну точку в стек, якщо правий — видаляємо точку зі стеку. Продовжуємо цю перевірку для всіх точок у списку. Цей крок має часову складність O(N).

Структури даних, які необхідні для виконання алгоритму це список для відсортованих вершин і стек для вершин опуклої оболонки.

Для сортування точок за полярним кутом і знаходження поворотів використовується один і той же метод, що був розглянутий в параграфі 1.4 цього підручника.

Для знаходження найвіддаленішої з колінеарних точок використовується проста формула Евклідової відстані між точками: $(x_1 - x_2)^2 + (y_1 - y_2)^2$.

6.3. Алгоритм Джарвіса

Багатокутник з однаковим успіхом можна задати впорядкованою множиною як його ребер так і його вершин. В завданні про опуклу оболонку ми до цих пір звертали увагу головним чином на ізольовані крайні точки. А що коли замість цього спробувати визначити ребра опуклої оболонки, чи приведе такий підхід до створення практично придатного алгоритму? Якщо задана множина точок, то досить важко швидко визначити чи є деяка точка крайньою. Проте якщо задані дві точки, то безпосередньо можна перевірити, чи є відрізок, що їх сполучає, ребром опуклої оболонки.

6.3.1. Теоретичні аспекти алгоритму

Теорема. Відрізок 1, що визначений двома точками, є ребром опуклої оболонки тоді і лише тоді, якщо всі інші точки заданої множини лежать на 1 або з однієї сторони від нього (рис. 6.5).

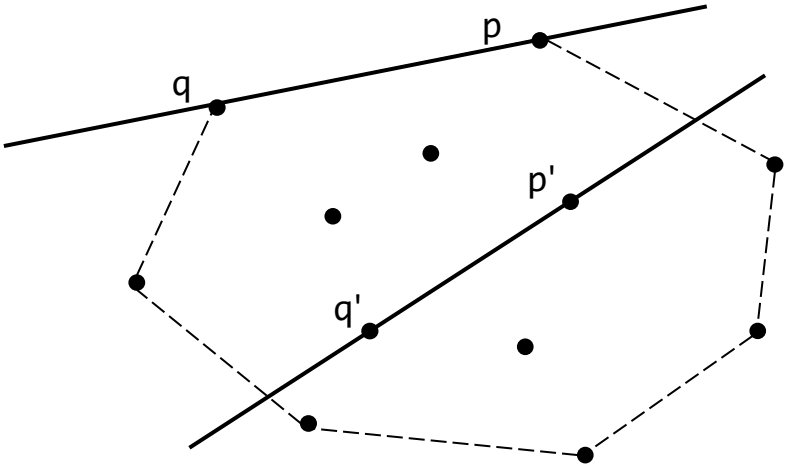


Рис. 6.5. Ребро опуклої оболонки

Ребро опуклої оболонки не може розділяти множину точок на частини:

- pq є ребром опуклої оболонки, оскільки всі точки множини розташовуються по одну сторону від нього;
- p'q' не є ребром опуклої оболонки, тому що по обидві сторони від нього є точки.

Джарвіс відмітив, що цей алгоритм можна поліпшити, якщо врахувати наступний факт. Якщо встановлено, що відрізок pq є ребром оболонки, то повинне існувати інше ребро з кінцем в точці q. В його роботі 1973 року показано, як використати цей факт, щоб зменшити необхідний час до $O(N^2)$.

Алгоритм Джарвіса обходить опуклу оболонку по колу (звідси і назва — обхід методом Джарвіса), породжує в потрібному порядку послідовність крайніх точок, по одній точці на кожному кроці (рис. 6.6). Таким чином будується частина опуклої оболонки (ламана лінія) від найменшої в лексикографічному порядку точки (p_1 на рис. 6.6) до найбільшої в лексикографічному порядку точки (p_4 на тому ж рисунку). Побудова опуклої оболонки завершується знаходженням іншої ламаної, такої, що йде з найбільшої в лексикографічному порядку точки в найменшу в лексикографічному порядку точку. Зважаючи на симетричність цих двох етапів необхідно змінити на протилежні напрямки осей координат і мати справу тепер з полярними кутами, найменшими відносно негативного напряму осі X.

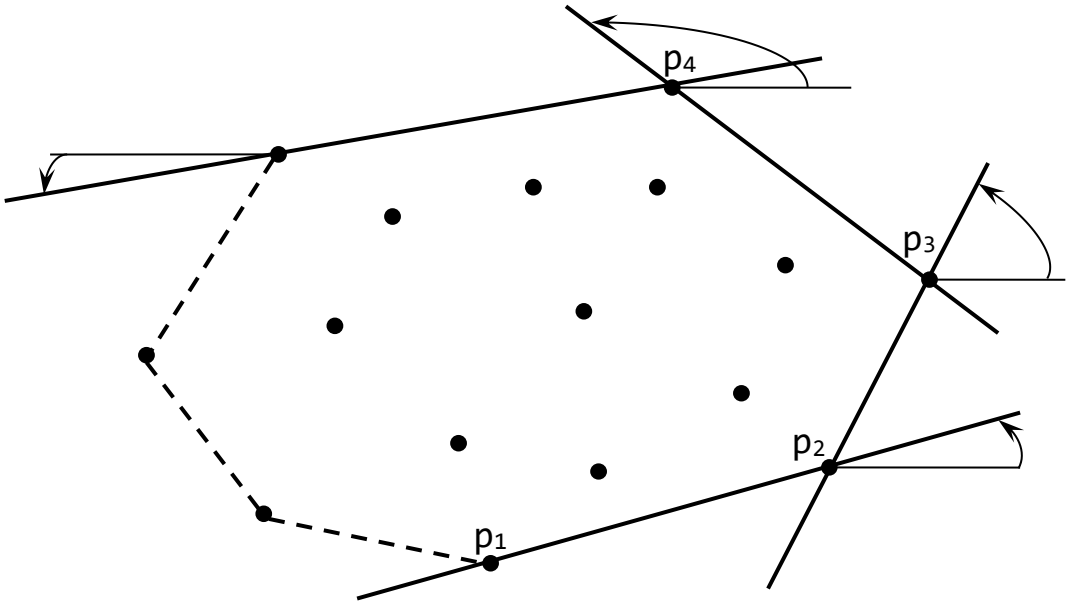


Рис. 6.6. Ілюстрація до алгоритму Джарвіса

Алгоритм Джарвіса знаходить послідовні вершини оболонки шляхом багатократного обчислення кута повороту. Кожна нова вершина визначається за час O(N).

Найменший кут може бути знайдений з використанням лише арифметичних операцій та порівнянь, не вдаючись до явного обчислення значень полярних кутів. Оскільки всі N точок множини можуть лежати на його опуклій оболонці (бути її вершинами), а алгоритм Джарвіса витрачає на знаходження кожної точки оболонки лінійний час, то час виконання алгоритму в найгіршому випадку рівний $O(N^2)$, що гірше, ніж в алгоритму Грехема. Якщо в дійсності число вершин опуклої оболонки рівне h, то час виконання алгоритму Джарвіса буде O(hN), і він дуже ефективний, коли заздалегідь відомо, що значення h мале. Наприклад, якщо оболонка заданої множини є багатокутником з довільним постійним числом сторін, то її можна знайти за лінійний час відносно числа точок.

Інше доречне тут зауваження полягає в тому, що ідея пошуку послідовних вершин оболонки за допомогою багатократного використання процедури визначення мінімального кута інтуїтивно асоціюється із загортанням двовимірного предмету. Насправді алгоритм Джарвіса можна розглядати як двовимірний варіант підходу, заснованого на ідеї «загортання подарунку» запропонованого Чандом і Капуром ще до появи роботи Джарвіса. Метод «загортання подарунку» застосовний також у випадку просторів, де розмірність більша за два.

6.3.2. ФОРМАЛЬНИЙ ОПИС АЛГОРИТМУ

Виконання алгоритму Джарвіса схоже на виконання алгоритму сортування «бульбашкою» — в обох випадках ми знаходимо найменший елемент і додаємо його у відсортований список. Якщо ми виконуємо обхід методом Джарвіса за

годинниковою стрілкою, то щоразу шукаємо найбільш ліву точку серед всіх і додаємо її в список вершин опуклої оболонки.

Послідовність виконання алгоритму Джарвіса включає чотири кроки, що подані нижче.

1. В заданій множині точок знаходимо точку з найменшою координатою X (позначимо її p). Оскільки ця точка буде гарантовано точкою опуклої оболонки, то додаємо її в список точок опуклої оболонки.
2. Знаходимо найбільш ліву точку відносно точки p. Для цього знаходимо точку, що розміщена після p (назвемо її q) та перевіряємо чи утворює правий поворот з'єднання відрізків pq та qj (де j це будь-яка інша точка множини). Якщо утворений поворот є правим, тоді точка j стає точкою q. Таким чином точка q рухається доки не стане в найбільш лівою. Після перегляду всіх точок, додаємо точку q в список вершин опуклої оболонки.
3. Тепер точка q стає точкою p і повторюється крок 2.
4. Повторюємо кроки 2 і 3 поки не досягнемо початкової точки.

Структури даних, які необхідні для виконання алгоритму це списки для відсортованих вершин і вершин опуклої оболонки.

Для знаходження поворотів використовується той самий метод, що і в алгоритмі Грехема.

6.4. Швидкий метод побудови опуклої оболонки

Швидкий метод розбиває множину S з N точок на дві підмножини, кожна з яких міститиме одну з двох ламаних, з'єднання яких дає багатокутник опуклої оболонки. Початкове розбиття множини визначається прямою, що проходить через дві точки l та r, що мають відповідно найменшу і найбільшу абсциси (рис. 6.7), так само, як і у варіанті алгоритму Грехема, запропонованому Ендрю.

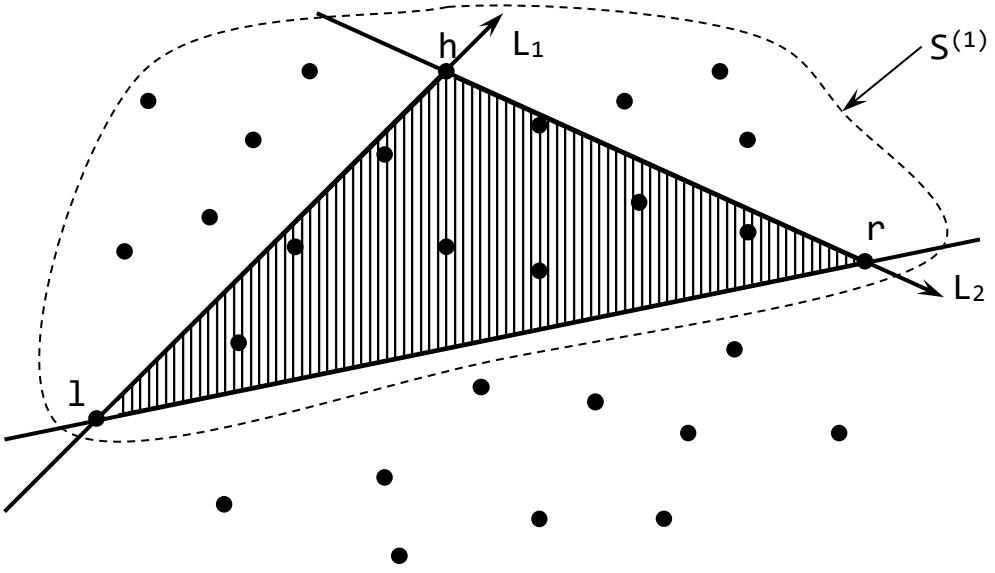


Рис. 6.7. Ілюстрація до швидкого методу побудови опуклої оболонки

Позначимо через $S^{(1)}$ підмножину точок, розташованих вище або на прямій, що проходить через l та r, а через $S^{(2)}$ симетричним чином визначена підмножина точок, розташованих нижче або на тій же самій прямій.

На кожному подальшому кроці обробка множин, подібних $S^{(1)}$ і $S^{(2)}$, виконується наступним чином (для конкретності ми розглянемо множину $S^{(1)}$ на рис. 6.7). Визначимо точку h, для якої трикутник (hlr) має максимальну площу серед всіх трикутників {(plr): $p \in S^{(1)}$}, а якщо таких точок більше ніж одна, то вибираємо ту з них, в якої кут (hlr) більший. Відмітимо, що точка h гарантовано належить опуклій оболонці. Дійсно, якщо провести через точку h пряму, паралельну відрізку lr, то вище цієї прямої не виявиться жодної точки множини S. Можливо, окрім точки h на цій прямій виявляться інші точки з множини S, але, згідно із зробленим нами вибором, h є найбільш лівою з них. Отже точка h не може бути подана у вигляді опуклої комбінації двох інших точок множини S.

Потім будуються дві прямі: одна L_1, направлена з l в h, інша L_2 — з h в r. Для кожної точки множини $S^{(1)}$ визначається її положення відносно цих прямих. Ясно, що жодна з точок не знаходиться одночасно ліворуч як від L_1, так і від L_2, крім того, всі точки, розташовані праворуч від обох прямих, є внутрішніми точками трикутника (lrh) і тому можуть бути видалені з подальшої обробки. Точки, що розташовані зліва від L_1 або на ній (і розташовані праворуч від L_2), утворюють множину $S^{(1,1)}$; аналогічно утворюється множина $S^{(1,2)}$. Утворені множини $S^{(1,1)}$ та $S^{(1,2)}$ передаються на наступний рівень рекурсивної обробки. Цей метод використовує процедури обчислення площі трикутника та визначення положення точки відносно прямої. Кожна з цих процедур вимагає декількох операцій складання і множення.

Алгоритм в середньому має час роботи $O(N \log N)$, а в найгіршому — $O(N^2)$.

6.5. Алгоритм апроксимації опуклої оболонки

Замість вибору алгоритму побудови опуклої оболонки на підставі його складності в середньому, можна піти по альтернативному шляху, розробляючи алгоритми, що будують апроксимації для реальної опуклої оболонки, розмінюючи тим самим точність на простоту та ефективність алгоритму. Такий алгоритм міг би, зокрема, бути дуже корисним для застосунків, де необхідно швидко отримати рішення, жертвуючи заради цього навіть точністю. Так, наприклад, це є прийнятним в прикладній статистиці, де результати спостережень не є точними, а відомі з деякою точністю.

Розглянемо один з таких алгоритмів для плоского випадку. Основна ідея цього алгоритму проста і полягає в тому, щоб виокремити з множини точок деяку підмножину, опукла оболонка якої і слугуватиме апроксимацією опуклої оболонки заданої множини точок. Проте конкретна схема виокремлення апроксимуючої підмножини точок, яка буде обрана далі, ґрунтується на моделі обчислень, відмінній від тих, що розглядалися раніше. А саме, в цьому розділі

буде передбачатися, що функція взяття цілої частини входить до складу множини примітивних операцій.

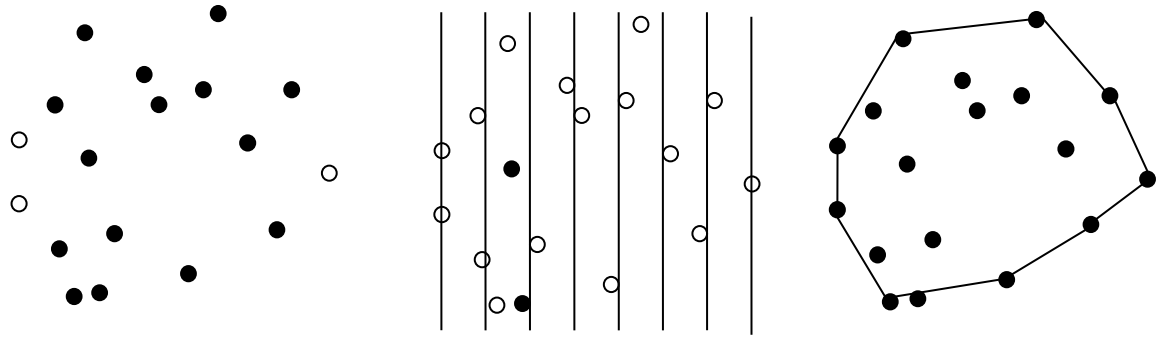


Рис. 6.8. Ілюстрація до алгоритму апроксимації випуклої оболонки: (а) – задана множина точок на площині, в якій виокремлюють найбільш ліві та найбільш праві точки; (б) – розбиття множини точок на частини в залежності від належності їх до однієї з k смуг та визначення в кожній смузі точок з екстремальною ординатою; (в) – наближена опукла оболонка.

На першому кроці алгоритму шукають мінімальне і максимальне значення координати X точок множини (рис. 6.8, а), а потім вертикальна смуга між ними розбивається на смуги рівної ширини. Ці k смуг утворюють послідовність «ящиків», в яких будуть розподілені N точок заданої множини S (для цього якраз і знадобиться функція взяття цілої частини). Після цього в кожній із смуг шукають дві точки, що мають мінімальне і максимальне значення координати Y (рис. 6.8, б). Крім того, вибираються точки з екстремальними значеннями координати X. Якщо одне й те ж екстремальне значення по координаті X мають відразу декілька точок, то з них вибираються точки лише з мінімальною і максимальною координатами Y. Таким чином, результуюча множина (S^*), містить не більше ніж 2k+4 точок. Нарешті, будується опукла оболонка множини S^*, яка є апроксимацією оболонки заданої множини (рис. 6.8, в). Відзначимо, що оболонка яка вийшла, насправді є лише апроксимацією: у прикладі на рис. 6.8 (в) одна з точок заданої множини лежить поза побудованою оболонкою.

Описаний тут коротко метод надзвичайно простий в реалізації. Вказані k смуг подаються масивом з (k+2) елементів (нульовий і (k+1)-й елементи містять дві точки з екстремальними значеннями координати X відповідно X_{min} та X_{max}). Щоб визначити смугу, в яку потрапляє деяка точка p, необхідно відняти від X_p мінімальне значення координати X (X_{min}) і розділити різницю, що вийшла, на (1/k)-у частину різниці значень координат X екстремальних точок. Взявши цілу частину від отриманого результату, отримаємо номер смуги, яка містить точку. Можна паралельно шукати мінімум і максимум в кожній смузі, оскільки для кожної точки перевіряється, чи виходить вона за межі поточних значень максимуму та мінімуму. Якщо це має місце, відбувається необхідна зміна в масиві. І нарешті, можна вибрати зручний в цьому випадку алгоритм побудови опуклої оболонки: очевидно, що найбільш зручним є варіант алгоритму Грехема, запропонований Ендрю. Відмітимо, що точки множини S^* майже впорядковані

по значенню координати X. Щоб отримати повністю впорядковану множину, необхідно лише порівняти в кожній смузі значення координати X двох точок множини S^*, що належать цій смузі.

Запитання для самоконтролю

1. Як ви розумієте поняття «опукла оболонка»?
2. Які основні кроки будь-якого алгоритму побудови опуклої оболонки?
3. Опишіть алгоритм побудови опуклої оболонки за методом Грехема.
4. Опишіть алгоритм побудови опуклої оболонки за методом Джарвіса.
5. На чому ґрунтується побудова опуклої оболонки швидким методом?
6. Опишіть алгоритм побудови апроксимації опуклої оболонки.

7. ТРІАНГУЛЯЦІЇ

Тріангуляція — планарний граф, всі внутрішні області якого є трикутниками.

Задача. На площині задано N точок. З'єднати їх відрізками, що не перетинаються таким чином, щоб кожна з областей усередині опуклої оболонки цієї множини точок була трикутником.

Граф тріангуляції множини з N точок, є планарним, має не більш 3*N-6 ребер. Результатом розв'язку сформульованої вище задачі повинен бути мінімум — список цих ребер.

Відмітимо важливість тріангуляції для чисельних застосунків, що пов'язані з інтерполяцією поверхонь, як в задачах графічного відображення даних, так і в задачах чисельного аналізу. Приклад використання: алгоритм побудови перетину багатогранників потребує виконання попередньої тріангуляції поверхні багатогранників.

7.1. ЖАДІБНА ТРІАНГУЛЯЦІЯ

«Жадібний» метод — це такий метод, під час якого ніколи не скасовуються те, що вже було зроблено раніше. Таким чином, жадібний метод тріангуляції послідовно породжує ребра тріангуляції (по одному за раз) і завершує цей процес після того, як породжена необхідна кількість ребер, яка повністю визначається розміром множини точок та його опуклої оболонки. Якщо мета полягає у мінімізації сумарної довжини ребер, то все, що можна зробити, використавши жадібний метод, це застосувати локальний критерій, додаючи на кожному етапі найменше з можливих ребер, сумісне з раніше породженими ребрами, тобто таке, що не перетинає жодне з них.

Алгоритм «жадібної тріангуляції» виконується в такій послідовності:

1. Для кожної точки визначається відстань (квадрат відстані) до інших точок.
2. Список утворених таким чином ребер (R0) сортується за зростанням довжини ребер.
3. Перше ребро (з мінімальною довжиною) записується в список ребер TIN-моделі (RT).
4. Для кожного ребра множини R0 проводиться тест перетину з ребрами множини RT. Якщо ребро не перетинається з ребрами множини RT, то воно додається до множини RT. Для N точок процес завершується коли кількість ребер в множині RT досягнула числа (3*N-6) або після завершення тестування всіх ребер множини R0.
5. На множині ребер RT формуються трикутники вузловим методом, який зводиться до пошуку ребер інцидентних кожній точці заданої множини та суміжних ребер до них, які й утворюють підмножину трикутників відповідної точки вузла. Кожний трикутник в цій моделі подається у вигляді (i, j, k) — індексів початкових точок, які є вершинами трикутника.

Часова складність жадібного алгоритму складає $O(N^2 \log N)$. У зв'язку з цим на практиці цей алгоритм майже не застосовується.

7.2. Тріангуляція Делоне

Широке застосування в комп'ютерній графіці отримала тріангуляція, що названа на честь радянського математика *Бориса Миколайовича Делоне*. Він в 1934 році у праці, присвяченій пам'яті математика Г. Ф. Вороного, сформулював теорему про порожнисту кулю, яка стосовно двовимірного простору формулюється так: *система взаємозв'язаних трикутників, що не перекриваються, має найменший діаметр, якщо жодна із вершин не попадає в середину жодного з кіл, описаних навколо утворених трикутників*.

Тріангуляція Делоне — це випукла тріангуляція, що задовольняє умові Делоне, сформульованій в теоремі про порожнисту кулю (рис. 7.1).

Рис. 7.1. Тріангуляція Делоне

Тріангуляція Делоне добре збалансована — її трикутники прямують до рівносторонніх, а система трикутників завжди має опуклу границю.

Тріангуляція Делоне є двоїстою до діаграм Вороного — якщо ми візьмемо центри описаних кіл навколо трикутників Делоне і з'єднаємо їх, то отримаємо *діаграму Вороного*, яка складається з *багатокутників Вороного*.

Багатокутник Вороного — геометричне місце точок на площині, що знаходяться ближче до заданої точки P, ніж до будь-якої іншої точки P_i. Сукупність цих багатокутників утворює діаграму Вороного (рис. 7.2).

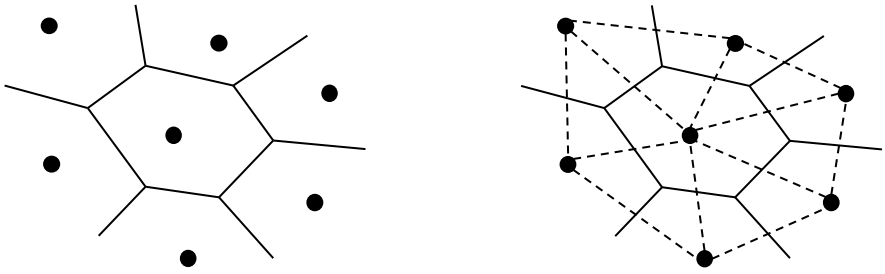


Рис. 7.2. Діаграма Вороного

7.2.1. Перевірка умови Делоне

Для побудови тріангуляції Делоне необхідно перевіряти отримані пари трикутників на виконання умови Делоне. На практиці використовується декілька способів такої перевірки:

- перевірка через рівняння описаного кола;
- перевірка з описаним колом, що вирахуване раніше;
- перевірка суми протилежних кутів;
- модифікована перевірка суми протилежних кутів.

Розглянемо детально перевірку через рівняння описаного кола. Рівняння кола, що проходить через три точки a, b, c можна записати у вигляді визначника:

$$\begin{bmatrix} x_a & y_a & x_a^2+y_a^2 & 1 \\ x_b & y_b & x_b^2+y_b^2 & 1 \\ x_c & y_c & x_c^2+y_c^2 & 1 \\ x & y & x^2+y^2 & 1 \end{bmatrix}.$$

Щоб визначити розташування довільної точки d відносно заданого кола, необхідно підставити її координати в рівняння кола:

$$\Delta = \begin{bmatrix} x_a & y_a & x_a^2+y_a^2 & 1 \\ x_b & y_b & x_b^2+y_b^2 & 1 \\ x_c & y_c & x_c^2+y_c^2 & 1 \\ x_d & y_d & x_d^2+y_d^2 & 1 \end{bmatrix} = \begin{bmatrix} x_a-x_d & y_a-y_d(x_a-x_d^2) & y_a-y_d^2 \\ x_b-x_d & y_b-y_d(x_b-x_d^2) & y_b-y_d^2 \\ x_c-x_d & y_c-y_d(x_c-x_d^2) & y_c-y_d^2 \end{bmatrix}.$$

Якщо $\Delta=0$, це означає, що точка d лежить на колі, утвореному точками a, b, c. Якщо $\Delta>0$, точка d належить колу, якщо $\Delta<0$ — не належить.

7.2.2. Алгоритми побудови тріангуляції Делоне

Існує досить велика кількість алгоритмів побудови тріангуляції Делоне. Багато з них використовують операцію *фліпу*.

Фліп — перебудова двох трикутників по найменшому суміжному ребру (рис. 7.3).

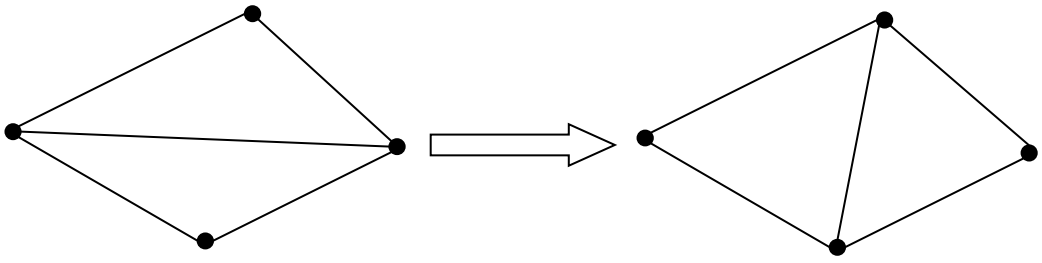


Рис. 7.3. Фліп ребра суміжних трикутників

Всі алгоритми побудови тріангуляції Делоне можна розділити на чотири категорії:

1. Ітеративні алгоритми — засновані на покроковому додаванні точок в уже побудовану тріангуляцію.
2. Алгоритми злиття — засновані на розбитті заданої множини точок на підмножини, з наступною тріангуляцією кожної з них та злиттям результатів.

3. Алгоритми прямої побудови — будують тільки ті трикутники, які задовольняють умові Делоне і тому не потребують перебудови.
4. Двопрохідні алгоритми — будь-яким алгоритмом спочатку будується тріангуляція, а на наступному проході здійснюється перевірка трикутників.

Кожна з вказаних категорій розділяється на декілька підкатегорій, які в свою чергу можуть поділятися далі.

Розглянемо один з алгоритмів прямої побудови, що називається покроковим, або *інкрементальним*.

Для спрощення алгоритму тріангуляції зробимо кілька припущень стосовно набору точок S:

- для існування тріангуляції необхідно, щоби набір S містив мінімум 3 неколінеарні точки;
- ніякі 4 точки не повинні знаходитись на одному колі, якщо це твердження неістинне, то можна побудувати декілька різних тріангуляцій Делоне;
- ребра знаходяться по правилу лівої орієнтації (всі точки знаходяться праворуч, порожня множина ліворуч).

Алгоритм працює шляхом постійного нарощування поточної тріангуляції по одному трикутнику за один крок. Спочатку поточна тріангуляція складається з єдиного ребра оболонки, після закінчення роботи алгоритму поточна тріангуляція стає тріангуляцією Делоне. На кожній ітерації алгоритм шукає новий трикутник, який підключається до границі поточної тріангуляції.

Визначення границі залежить від наступної схеми класифікації ребер тріангуляції Делоне щодо поточної тріангуляції. Кожне ребро може бути *сплячим*, *живим* або *мертвим*:

- *спляче* — ребро тріангуляції Делоне, що ще не було виявлено алгоритмом;
- *живе* — ребро, що виявлено, але для нього відома лише одна прилегла область;
- *мертве* — ребро, що виявлено і відомі обидві області, що прилягають до нього.

Спочатку живим є єдине ребро, що належить опуклій оболонці — до нього прилягає необмежена площина, а всі інші ребра сплячі. Під час роботи алгоритму ребра зі сплячих стають живими, а потім мертвими. Границя на кожному етапі складається з набору живих ребер.

На кожній ітерації вибирається будь-яке одне з ребер границі та піддається обробці, що полягає в пошуку невідомої області, до якої належить ребро. Якщо ця область виявиться трикутником, що визначається кінцевими точками ребра та деякою третьою вершиною (V), то ребро стає мертвим, оскільки тепер відомі обидві області, що прилягають до нього. Кожне з двох інших ребер трикутника

переводяться в наступний стан: із сплячого в живе або з живого в мертве. Тоді вершина (V) буде називатися *спряженою* з ребром. В протилежному випадку, якщо невідома область виявляється нескінченною площиною, то ребро просто вмирає. У цьому випадку ребро не має спряженої вершини.

Пошук спряженої вершини можна порівняти з надуванням плоскої бульбашки на ребрі (рис. 7.4).

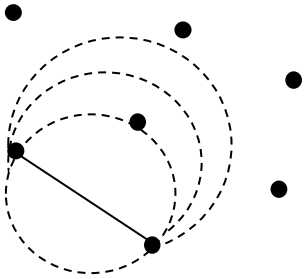


Рис. 7.4. Знаходження спряженої вершини для ребра

Для реалізації інкрементального алгоритму необхідно використовувати наступні функції:

- пошук крайнього ребра (побудова опуклої оболонки);
- побудова перпендикуляра до відрізка;
- тест кола;
- тест перевірки наявності точок зліва від ребра (тест орієнтації);
- тест стану ребра тріангуляції (визначення скільки разів використовувалося ребро);
- побудова кола на хорді;
- побудова трикутника з лівою орієнтацією.

Наведений алгоритм має час виконання $O(N^2)$, що далеко не найкращий в порівнянні з іншими алгоритмами.

7.3. Тріангуляція багатокутників

В попередніх підрозділах розглядалася тріангуляція на множині точок. Але часто виникає задача тріангуляції поверхонь, що задані багатокутниками.

Задачу тріангуляції багатокутників можна розбити на три категорії:

1. Тріангуляція опуклих багатокутників.
2. Тріангуляція неопуклих багатокутників.
3. Тріангуляція багатокутників з отворами.

Описана класифікація включає всі алгоритми тріангуляції багатокутників від найпростішого до найскладнішого. Розглянемо деякі найпростіші алгоритми тріангуляції багатокутників.

7.3.1. Тріангуляція опуклих багатокутників

Якщо відомо, що багатокутник опуклий, то є два основних методи його тріангуляції:

- розщеплення хордою на два багатокутника, і подальше рекурсивне розщеплення кожного утвореного багатокутника доки не залишаться лише трикутники;
- послідовне «відрізання» трикутників хордами з однієї вершини.

Ілюстрація обох методів подана на рис. 7.5.

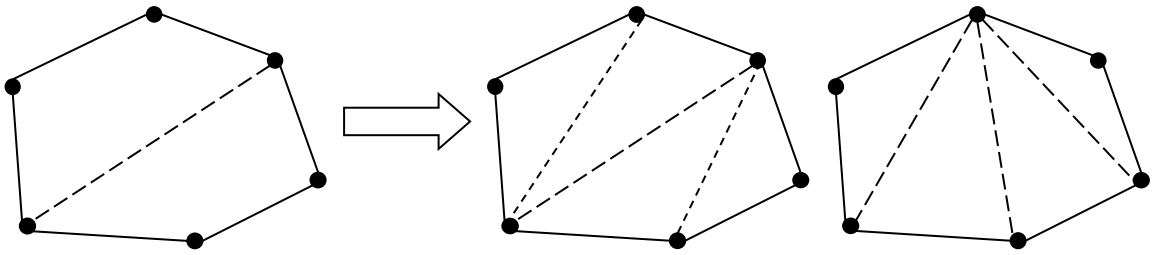


Рис. 7.5. Тріангуляція опуклих багатокутників

Обидва описаних методи доволі прості в реалізації, але працюють лише з опуклими багатокутниками.

7.3.2. Тріангуляція неопуклих багатокутників

Якщо багатокутник неопуклий, то для його тріангуляції можна застосувати два підходи:

- розбити його на опуклі багатокутники і тріангулювати одним з алгоритмів описаних раніше;
- застосувати алгоритм тріангуляції до заданого неопуклого багатокутника без розбиття на опуклі.

Найпростішим методом розбиття неопуклого багатокутника на опуклі є *відсікання*. Його алгоритм наступний.

1. Перенести систему координат в першу вершину багатокутника.
2. Повернути систему координат таким чином, щоб вісь X співпадала з ребром багатокутника.
3. Знайти сторони багатокутника, які лежать нижче осі X, та відсікти їх.
4. Перейти до наступної вершини і повторювати алгоритм рекурсивно доти, доки не залишаться лише опуклі багатокутники.

Одним з методів тріангуляції неопуклого багатокутника є алгоритм послідовної побудови трикутників на трьох сусідніх вершинах з перевіркою отриманого результату (рис. 7.6).

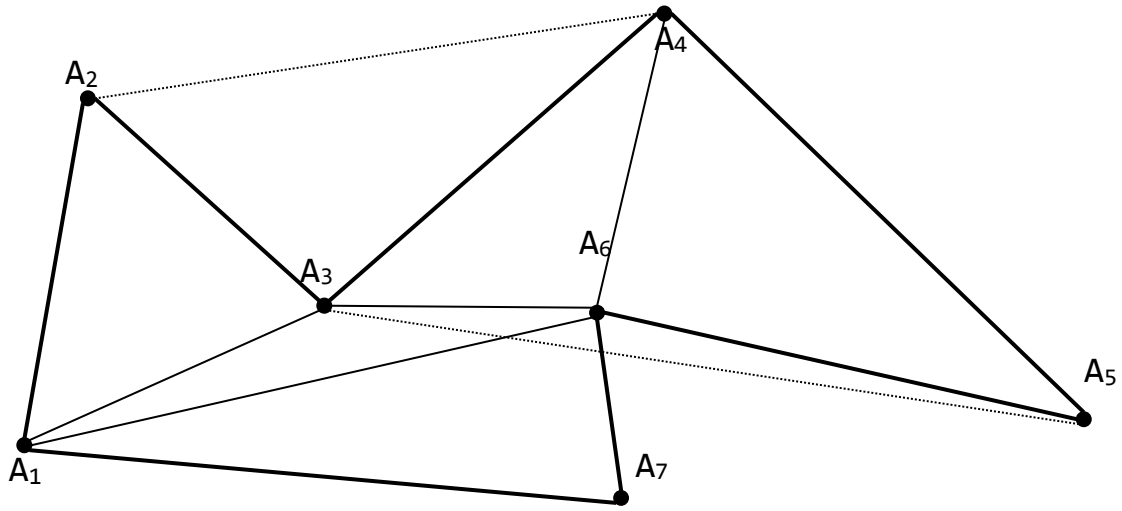


Рис. 7.6. Тріангуляція неопуклого багатокутника

Спочатку всі вершини багатокутника впорядковуємо за годинниковою стрілкою. Тоді для побудови тріангуляції можна використати наступний алгоритм.

1. Беремо послідовно три вершини A_i, A_{i+1}, A_{i+2}.
2. Перевіряємо, чи утворюють вектори A_iA_{i+1} та $A_{i+1}A_{i+2}$ правий поворот. Робимо це за методом, що описаний в параграфі 1.4.
3. Перевіряємо, чи не потрапляє в середину трикутника $A_iA_{i+1}A_{i+2}$ будь-яка із вершин, що залишилися.
4. Якщо умови 2 і 3 виконуються, то будуємо трикутник по точкам $A_iA_{i+1}A_{i+2}$. Вершину A_{i+1} виключаємо із розгляду. Наступним розглядаємо трикутник $A_iA_{i+2}A_{i+3}$.
5. Якщо хоч одна з умов 2 чи 3 не виконалась, то переходимо до розгляду вершин A_{i+1}, A_{i+2}, A_{i+3}.
6. Повторюємо з кроку 1, доки не залишаться лише 3 вершини, які утворять останній трикутник.

Алгоритми тріангуляції багатокутників з отворами набагато складніші і в цьому підручнику не розглядаються.

Запитання для самоконтролю

1. Що таке тріангуляція?
2. Які види тріангуляції ви знаєте?
3. В чому перевага тріангуляції Делоне?
4. Які методи використовуються для перевірки трикутників на відповідність їх умові Делоне?
5. Які категорії задачі тріангуляції багатокутників ви знаєте?

8. Моделювання кривих

Моделювання кривої — процес відтворення форми кривої заданою множиною дискретних точок.

Для побудову кривих використовують два основних способи моделювання: *інтерполяцію* та *апроксимацію*.

Інтерполяція — побудова кривої, що проходить крізь контрольні точки.

Апроксимація — наближення кривої, гарантує проходження синтезованої форми через задані точки (крива необов'язково проходить крізь задані точки, але задовольняє деяку задану властивість відносно цих точок).

Неперервна крива — крива, яка не має розривів. Такі криві відносять до класу C^0. В загальному випадку неперервність C^n означає, що неперервні функція, та її перші n похідні.

8.1. Інтерполяція

Задача. Задано $f(x_i)=y_i$, $x_i<x_{i+1}$, $i\in[1,n]$ (рис. 8.1). Побудувати функцію $f(x)$, що задовольняє наведеній умові. Будемо використовувати такі терміни: x_i — вузли інтерполяції, пари (x_i,y_i) — базові точки, різниця між «сусідніми» точками (x_i-x_{i-1}) — крок.

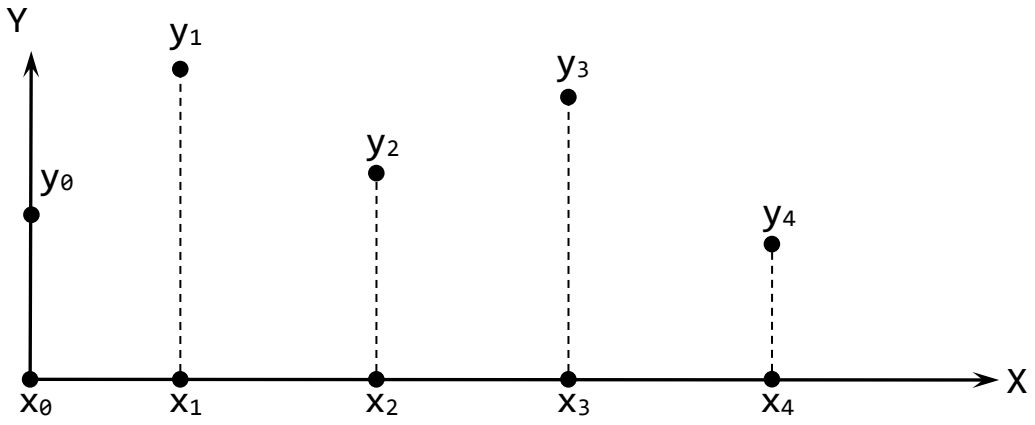


Рис. 8.1. Постановка задачі для інтерполяції

Існує багато різних способів інтерполяції. Вибір найбільш придатного алгоритму залежить від відповідей на питання: наскільки точний обраний метод, які затрати на його використання, наскільки гладкою є інтерполяційна функція, яку кількість точок даних вона вимагає і т.д.

На практиці найчастіше застосовують інтерполяцію многочленами. Це зв'язано перш за все з тим, що многочлени легко розраховувати та легко аналітично знаходити їх похідні.

8.1.1. Лінійна інтерполяція

Розглянемо функцію інтерполяції, яка задається окремо на кожному відрізку $[x_i,x_{i+1}]$, $i\in[0,n-1]$, що дозволяє краще враховувати локальну поведінку необхідної функції та уникнути громіздких обчислень (оскільки на кожному з відрізків функція інтерполяції має по можливості простий вигляд).

Лінійна інтерполяція – це інтерполяція функції алгебраїчним двочленом $P_1(x)=kx+c$ у точках x_0 та x_1, які належать відрізку [a,b] (рис. 8.2).

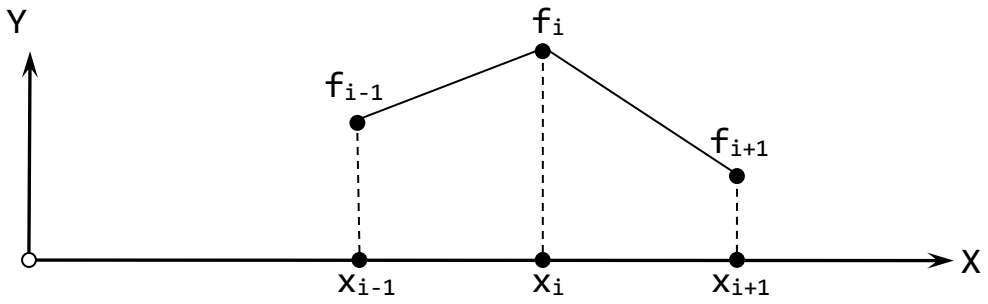


Рис. 8.2. Лінійна інтерполяція

З геометричної точки зору це означає заміну функції f прямою, що проходить через точки $(x_i,f(x_i))$ та $(x_{i+1},f(x_{i+1}))$.

Рівняння такої прямої має вигляд:

$$\frac{y-f(x_i)}{f(x_{i+1})-f(x_i)}=\frac{x-x_i}{x_{i+1}-x_i},$$

звідси для $x\in[x_i,x_{i+1}]$ маємо:

$$f(x)\approx y=f(x_i)+\frac{f(x_{i+1})-f(x_i)}{x_{i+1}-x_i}(x-x_i).$$

Лінійна інтерполяція виконується на кожному проміжку між двома точками всієї множини. Результуюча інтерполяція кривої є об'єднання окремих частинок, тому іноді також називається лінійна інтерполяція частинами. Така крива є неперервною, тобто відноситься до класу C^0. Основним недоліком лінійної інтерполяції є низька гладкість отриманої кривої.

8.1.2. Інтерполяційний многочлен Лагранжа

Інтерполяційний многочлен Лагранжа — многочлен мінімальної степені, що приймає задані значення у заданому наборі точок. Для n+1 пар чисел $(x_0,y_0),(x_1,y_1)\ldots(x_n,y_n)$, де всі x_i різні, існує єдиний многочлен L(x) степені не більшої за n, для якого $L(x_i)=y_i$.

У найпростішому випадку n=1 — це лінійний многочлен, графік якого — пряма, що проходить через дві задані точки.

Лагранж запропонував спосіб обчислення таких многочленів:

$$L(x) = \sum_{j=0}^{n} y_j l_j(x),$$

де базисні поліноми визначаються за формулою:

$$l_j(x) = \prod_{i=0, j\neq i}^{n} \frac{(x - x_i)}{(x_i - x_j)}.$$

Базисні поліноми мають наступні властивості:

- це поліноми степені n;
- $l_j(x_i)=1$;
- $l_j(x_i)=0$ коли $i\neq j$.

Звідси випливає, що L(x), як лінійна комбінація $l_j(x)$, може мати степінь не більшу за n, та $L(x_j)=y_j$ (рис. 8.3).

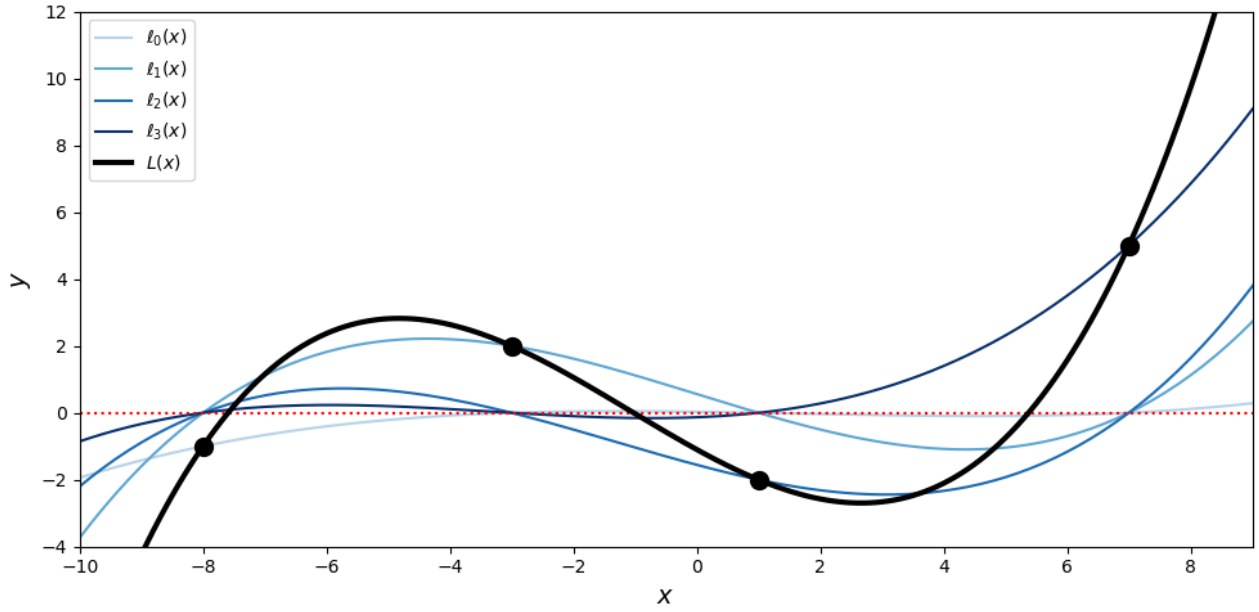


Рис. 8.3. Приклад многочлена Лагранжа

Недоліки використання многочлена Лагранжа наступні:

- потребує значного об'єму обчислень для знаходження значень функції у довільній точці (громіздкість розрахунків);
- невизначена поведінка побудованої функції між вузлами;
- необхідність повного перерахунку у разі додавання нової точки.

8.1.3. Інтерполяційний многочлен Ньютона

Інтерполяційний многочлен Ньютона — застосовується для побудови многочлена n-ї степені, який співпадає в (n+1) точці зі значеннями невідомої шуканої функції y=f(x) (рис. 8.4).

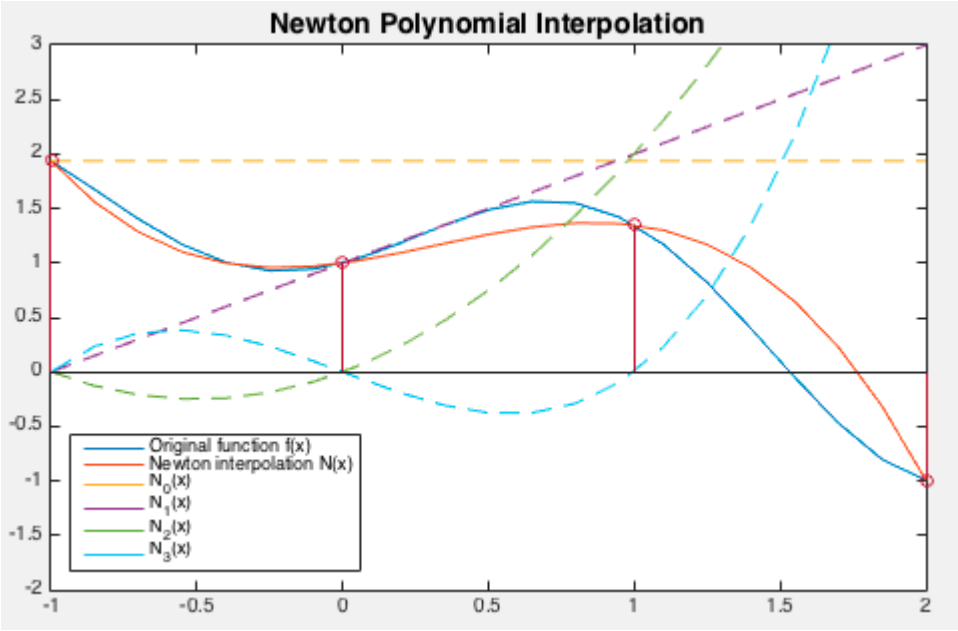


Рис. 8.4. Приклад многочлена Ньютона

Побудова многочлена Ньютона заснована на понятті *розділених різниць*. Розділеними різницями першого порядку називають вирази такого виду:

$$f(x_0, x_1) = \frac{f(x_1) - f(x_0)}{x_1 - x_0};\ f(x_1, x_2) = \frac{f(x_2) - f(x_1)}{x_2 - x_1}.$$

Розділеними різницями другого порядку називають вирази такого виду:

$$f(x_0, x_1, x_2) = \frac{f(x_1, x_2) - f(x_0, x_1)}{x_2 - x_0};\ f(x_1, x_2, x_3) = \frac{f(x_2, x_3) - f(x_1, x_2)}{x_3 - x_1}.$$

Якщо продовжити аналогічні побудови, то для (n+1)-го порядку ми отримаємо наступну формулу розділених різниць:

$$f(x_0, x_1, \dots, x_n, x_{n+1}) = \frac{f(x_1, \dots, x_{n+1}) - f(x_0, \dots, x_n)}{x_{n+1} - x_0}.$$

Якщо перетворити відношення різниць через значення функції, то отримаємо наступну формулу:

$$f(x_0, x_1, \dots, x_n) = \sum_{i=0}^{n} \frac{f(x_i)}{(x_i - x_0) \dots (x_i - x_{i-1})(x_i - x_{i+1}) \dots (x_i - x_n)}.$$

Тепер запишемо інтерполяційний многочлен Ньютона використовуючи отримані формули:

$$L_n(x) = f(x_0) + (x - x_0)f(x_0, x_1) + (x - x_0)(x - x_1)f(x_0, x_1, x_2) + \cdots$$
$$+ (x - x_0)(x - x_1) \dots (x - x_{n-1})f(x_0, x_1, \dots, x_n).$$

На відміну від многочлена Лагранжа, многочлен Ньютона не потребує повного перерахунку у випадку додавання нової точки, оскільки попередні результати від цього не змінюються. Всі інші недоліки многочлена Лагранжа можна застосувати і до многочлена Ньютона.

8.1.4. Сплайни

Сплайн — частинний поліном степені K з неперервною похідною степені K-1 у точках з'єднання сегментів (рис. 8.5).

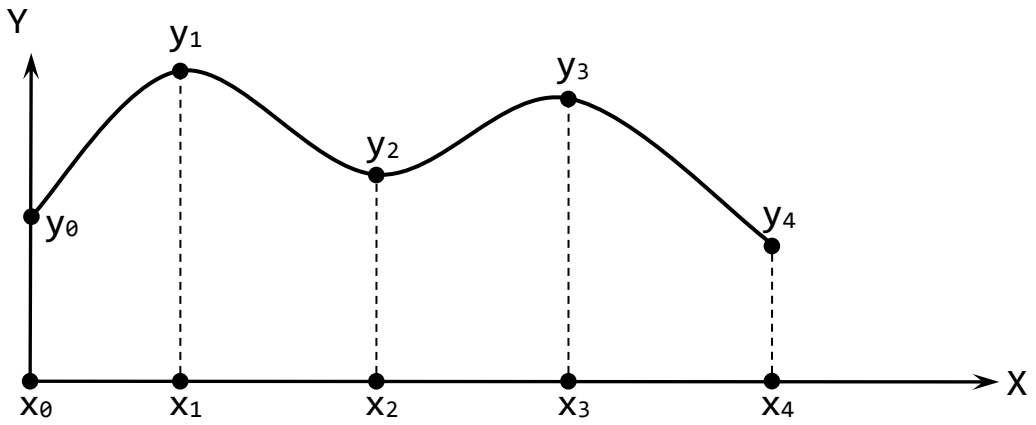


Рис. 8.5. Сплайн

Термін сплайн походить від англійського слова *spline*, що означає гнучку смужку сталі, яку застосовували креслярі для проведення плавних кривих під час побудови обводів кораблів або літаків.

В подальшому будемо розглядати кубічні сплайни, тобто сплайни форма яких задається кубічним поліномом.

Розглянемо кубічний сплайн для побудови функції однієї змінної. Нехай на площині задана послідовність точок (x_i,y_i), $i\in[0,n]$, причому $x_0<x_1<\ldots<x_n$. Визначимо функцію y=S(x), яка повинна задовольняти наступним умовам:

- функція повинна проходити через всі задані точки $S(x_i)=y_i$, $i\in[0,n]$;
- функція повинна мати неперервну першу та другу похідну на всьому відрізку $[x_0,x_n]$;
- на кожному відрізку $[x_{i-1},x_i]$ функція є многочленом третьої степені.

Для однозначного визначення сплайну перерахованих умов недостатньо, потрібно накласти деякі додаткові умови. *Природнім* кубічним сплайном називають сплайн, що задовольняє граничним умовам виду:

$$S''(a) = S''(b) = 0.$$

Теорема. Для будь-якої функції f і будь-якого розбиття відрізка [a,b] існує рівно один природній сплайн S(x), що задовольняє перерахованим вище умовам.

Приклад кубічного сплайну подано на рис. 8.6.

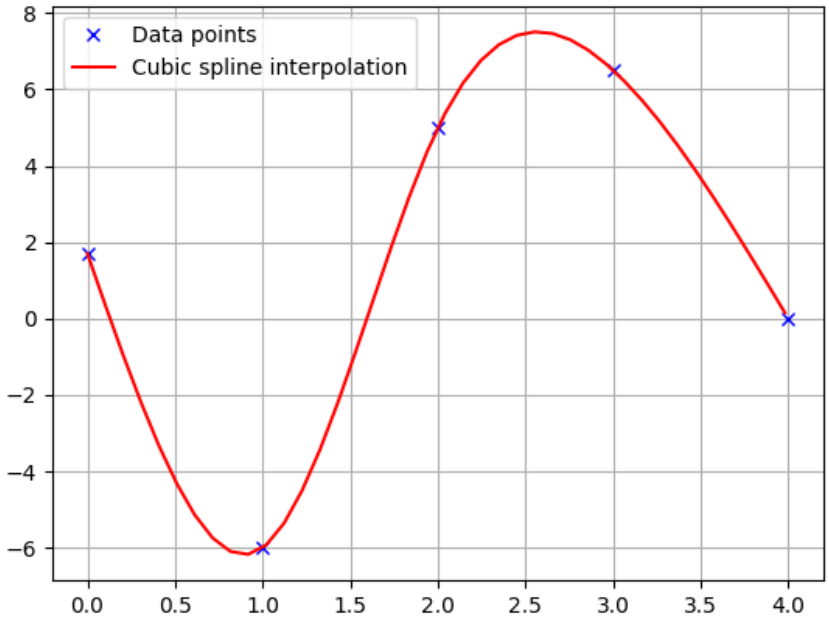


Рис. 8.6. Приклад кубічного сплайну

Запишемо функцію кубічного сплайну для відрізка $[x_i,x_{i+1}]$ у вигляді многочлена третьої степені:

$$S_i(x) = a_i + b_i(x - x_i) + \frac{c_i}{2}(x - x_i)^2 + \frac{d_i}{6}(x - x_i)^3. \qquad (8.1)$$

Тоді перші три коефіцієнти будуть відповідати значенням функції в точці та першим двом похідним відповідно:

$$S_i(x_i) = a_i;\ S_i'(x_i) = b_i;\ S_i''(x_i) = c_i.$$

Ми маємо чотири невідомих коефіцієнти, тому необхідно скласти систему з чотирьох рівнянь і розв'язати її.

Запишемо умови неперервності кубічного сплайну в `i`-й точці:

$$S_i(x_{i-1}) = S_{i-1}(x_{i-1});\ S_i'(x_{i-1}) = S_{i-1}'(x_{i-1});\ S_i''(x_{i-1}) = S_{i-1}''(x_{i-1}). \qquad (8.2)$$

Таким чином ми отримали три рівняння. Четвертим запишемо умову інтерполяції у вигляді:

$$S_i(x_{i-1}) = f(x_{i-1}). \qquad (8.3)$$

Об'єднавши рівняння (8.2) та (8.3) в систему та позначивши через h_i різницю (x_i - x_{i-1}), отримаємо наступні формули для знаходження коефіцієнтів сплайну:

$$\begin{aligned} a_i &= f(x_i); \\ h_i c_{i-1} + 2(h_i + h_{i+1})c_i + h_{i+1}c_{i+1} &= 6\left(\frac{f_{i+1} - f_i}{h_{i+1}} - \frac{f_i - f_{i-1}}{h_i}\right); \\ d_i &= \frac{c_i - c_{i-1}}{h_i}; \\ b_i &= -\frac{1}{2}h_i c_i - \frac{1}{6}h_i^2 d_i + \frac{f_i - f_{i-1}}{h_i}. \end{aligned}$$

Як бачимо з рівнянь, найпростіше знайти коефіцієнти a_i — вони дорівнюють значенням функції у вузлах інтерполяції. Коефіцієнти b_i та d_i вираховуються через коефіцієнти c_i, які й потрібно знайти. Враховуючи, що $c_0=c_n=0$, коефіцієнти c_i можна знайти за допомогою *методу прогонки* для тридіагональної матриці (матриці Якобі).

Метод прогонки використовується для вирішення систем лінійних рівнянь наступного виду:

$$A_i x_{i-1} + C_i x_i + B_i x_{i+1} = F_i.$$

Тоді розв'язок такої системи можна отримати за формулою:

$$x_i = \alpha_{i+1} x_{i+1} + \beta_{i+1}, \text{де } i = n-1, n-2, \dots, 1.$$

Коефіцієнти α_{i+1} та β_{i+1} знаходяться за наступними формулами:

$$\alpha_{i+1} = \frac{-B_i}{A_i \alpha_i + C_i};\ \beta_{i+1} = \frac{F_i - A_i \beta_i}{A_i \alpha_i + C_i}.$$

Конкретно для кубічного сплайну отримуємо $x_i=c_i$, а коефіцієнти A_i, B_i, C_i та F_i відповідають наступним значенням:

$$\begin{aligned} A_i &= h_i; \\ B_i &= h_{i+1}; \\ C_i &= 2(h_i + h_{i+1}); \\ F_i &= 6\left(\frac{f_{i+1} - f_i}{h_{i+1}} - \frac{f_i - f_{i-1}}{h_i}\right). \end{aligned}$$

Знайшовши за методом прогонки коефіцієнти c_i легко знаходимо інші коефіцієнти та підставивши їх у рівняння (8.1) отримуємо значення функції в шуканій точці.

8.2. Апроксимація

В попередньому підрозділі ми розглянули інтерполяцію різними методами, закінчивши розгляд інтерполяцією кубічним сплайном. Для апроксимації також використовуються сплайни, але у відповідності до визначення, вони не обов'язково проходять через всі задані точки. Найбільш розповсюджені методи апроксимації це використання кривих Без'є та B-сплайнів.

8.2.1. Криві Безьє

При вирішення задачі апроксимації типовим є використання кривих Безьє. Це пов'язано з їх зручністю як для аналітичного опису, так і для наочної геометричної побудови (стосовно комп'ютерної графіки це означає, що користувач може задавати форму кривої інтерактивно).

Криві Безьє були запроваджені в 1962 році *П'єром Безьє* з автомобілебудівної компанії «Рено», хоча ще в 1959 році використовувались *Полем де Кастельє* з компанії «Сітроен», але його дослідження не публікувались і приховувались компанією як комерційна таємниця до кінця 1960-х.

Наочний метод побудови цих кривих було запропоновано саме *де Кастельє* (de Casteljau) в 1959 році. Побудуємо криву по 3 опорним точкам (рис. 8.7). Метод де Кастельє заснований на розбиванні відрізків, що з'єднують задані початкові точки відносно t (значення параметра), а потім в рекурсивному повторенні цього процесу для отриманих відрізків.

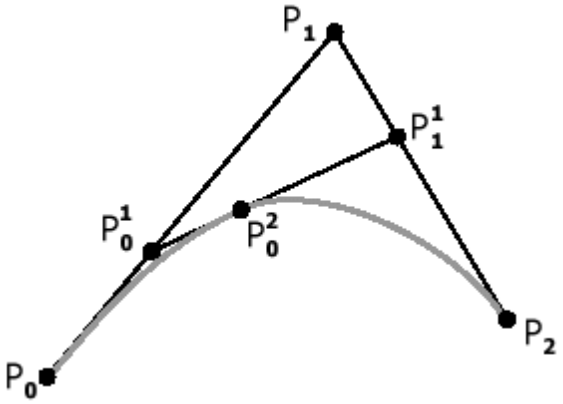


Рис. 8.7. Крива Безьє побудована за трьома опорними точками

Позначимо опорні точки через P_i, $i \in [0,2]$, початок кривої розмістимо у точці P_0 ($t=0$), а кінець у точці P_2 ($t=1$), для кожного $t \in [0,1]$ знайдемо точку P_0^2.

$$\begin{aligned} P_0^1 &= (1-t)P_0 + tP_1; \\ P_1^1 &= (1-t)P_1 + tP_2; \\ P_0^2 &= (1-t)P_0^1 + tP_1^1 = (1-t)^2 P_0 + 2t(1-t)P_1 + t^2 P_2. \end{aligned} \qquad (8.4)$$

Таким чином, із формул (8.4) випливає, що ми отримали криву другого порядку.

Тепер аналогічним чином побудуємо криву Без'є за чотирма опорними точками (рис. 8.8).

$$\begin{aligned} P_0^1 &= (1-t)P_0 + tP_1; \\ P_1^1 &= (1-t)P_1 + tP_2; \\ P_2^1 &= (1-t)P_2 + tP_3; \\ P_0^2 &= (1-t)P_0^1 + tP_1^1 = (1-t)^2 P_0 + 2t(1-t)P_1 + t^2 P_2; \\ P_1^2 &= (1-t)P_1^1 + tP_2^1 = (1-t)^2 P_1 + 2t(1-t)P_2 + t^2 P_3; \\ P_0^3 &= (1-t)P_0^2 + tP_1^2 = (1-t)^3 P_0 + 3t(1-t)^2 P_1 + 3t^2(1-t)P_2 + t^3 P_3. \end{aligned}$$

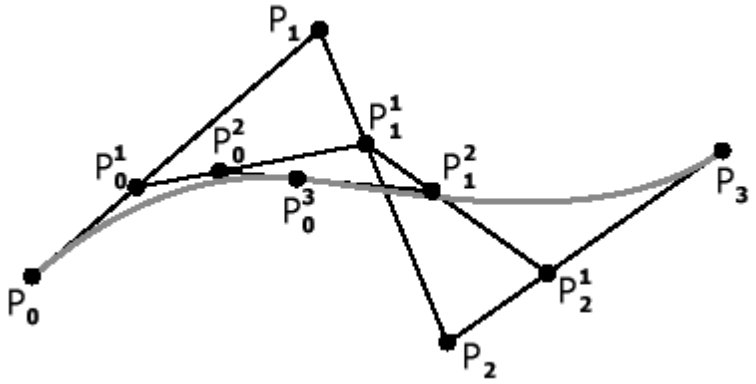


Рис. 8.8. Крива Безьє побудована за чотирма опорними точками

Для чотирьох точок ми отримали криву третього порядку. Можна продовжувати подібні побудови і для більшої кількості вузлів, отримуючи аналогічні результати. Матрична форма запису кривої Безьє третього порядку наступна:

$$B(t) = [t^3 \quad t^2 \quad t \quad 1] M_B \begin{bmatrix} P_0 \\ P_1 \\ P_2 \\ P_3 \end{bmatrix},$$

де M_B називається базисною матрицею Безьє.

$$M_B = \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}.$$

Загальне аналітичне подання кривої Безьє з n+1 опорною точкою:

$$P(t) = \sum_{i=0}^{n} P_i B_i^n(t),$$

де P_i — опорні вершини, $B_i^n(t)$ — базисні многочлени Бернштейна n-го степеня (вагові функції Безьє/Бернштейна). Вони розраховуються за наступною формулою:

$$B_i^n(t) = C_i^n t^i (1-t)^{n-i},$$

де C_i^n — біноміальні коефіцієнти. Вони розраховуються за формулою:

$$C_i^n(t) = \frac{n!}{i!\,(n-i)!}.$$

Також існує рекурсивна формула побудови кривих Безьє:

$$B_{P_0 P_1 \dots P_n}(t) = (1-t) B_{P_0 P_1 \dots P_{n-1}}(t) + t B_{P_1 P_2 \dots P_n}(t).$$

Криві Безьє мають наступні властивості:

1. Інваріантність відносно афінних перетворень.
2. Інваріантність відносно лінійних замін параметризації t=(x-a)/(b-a)*b.
3. Крива Безьє належить опуклій оболонці опорних точок (виходить з геометричного способу побудови). *Висновок:* Якщо всі опорні точки знаходяться на одній прямій, то крива Безьє вироджується у відрізок, що з'єднує ці точки.
4. Крива Безьє проходить через P_0 та P_n.
5. Симетричність: якщо розглядати контрольні точки у протилежному порядку, то крива не зміниться.
6. Степінь многочлена, що подає криву в аналітичному вигляді на 1 менше числа опорних точок.
7. Вектори дотичних у точках P_0 та P_n колінеарні P_0P_1 та $P_{n-1}P_n$, відповідно.

8.2.2. B-СЛАЙНИ

З математичної точки зору крива, що задана вершинами багатокутника, залежить від інтерполяції чи апроксимації, що встановлює зв'язок між кривою та багатокутником. Тут основою є вибір базисних функцій. Базис Бернштейна породжує криві Безьє, але він має дві властивості, які обмежують гнучкість кривих. По-перше, кількість вершин багатокутника жорстко задають порядок многочлена. Наприклад, багатокутник з шести точок завжди породжує криву п'ятого порядку.

Друге обмеження витікає з глобальної природи базису Бернштейна. Будь-яка точка, що лежить на кривій Безьє залежить від всіх визначальних вершин, тому зміна будь-якої однієї вершини чинить вплив на всю криву. Локальні впливи на криву неможливі.

Існує неглобальний базис, що має назву базис B-сплайну, він включає базис Бернштейна як частковий випадок. B-сплайни неглобальні, оскільки з кожною вершиною B_i зв'язана своя базисна функція.

Нехай P(t) визначає криву як функцію від параметра t, тоді B-сплайн має вигляд:

$$P(t) = \sum_{i=1}^{n+1} B_i \, N_i^k(t), \;\; t_{min} \le t \le t_{max}, \;\; 2 \le k \le n+1,$$

де B_i є n+1 вершина багатокутника, а N_i^k — нормалізовані функції базису B-сплайну. Відмітимо, що на відміну від кривих Безьє вершини визначаючого багатокутника нумеруються від 1 до n+1.

Для i-ї нормалізованої функції базису порядку k (степені k-1) функції базису $N_i^k(t)$ визначаються рекурсивними формулами Кокса-де Бура:

$$N_i^1(t) = \begin{cases} 1, & x_i \le t \le x_{i+1} \\ 0, & \text{інакше} \end{cases},$$
$$N_i^k(t) = \frac{(t - x_i) N_i^{k-1}(t)}{x_{i+k-1} - x_i} + \frac{(x_{i+k} - t) N_{i+1}^{k-1}(t)}{x_{i+k} - x_{i+1}}. \qquad (8.5)$$

Величини x_i — це елементи вузлового вектора, які задовольняють співвідношенню $x_i \le x_{i+1}$. Параметр t змінюється від t_{min} до t_{max} вздовж кривої P(t). Вважається, що 0/0=0.

Формально B-сплайн визначається як поліноміальний сплайн порядку k (степені k-1), оскільки він задовольняє наступним умовам:

- функція P(t) є поліномом степені k-1 на кожному інтервалі $x_i \le t < x_{i+1}$;
- P(t) та її похідні порядку 1,2,…,k-2 неперервні вздовж всієї кривої.

Так, наприклад, B-сплайн четвертого порядку — це часткова кубічна крива.

Із того, що B-сплайн задається базисом B-сплайну, одразу визначається ще декілька властивостей:

- сума базисних функцій B-сплайну для будь-якого значення t:

$$\sum_{i=1}^{n+1} N_i^k(t) \equiv 1;$$

- кожна базисна функція додатна або рівна нулю для всіх значень параметра, тобто $N_i^k \geq 0$;
- окрім k=1, всі базисні функції мають рівно один максимум;
- максимальний порядок кривої дорівнює кількості вершин визначального багатокутника;
- крива володіє властивістю зменшення варіації. Крива перетинає будь-яку пряму не частіше, ніж її визначальний багатокутник;
- загальна форма кривої повторює форму визначального багатокутника;
- щоб застосувати до кривої будь-яке афінне перетворення, необхідно застосувати його до вершин визначаючого багатокутника;
- крива лежить всередині випуклої оболонки визначаючого багатокутника.

Вибір вузлового вектору чинить істотний вплив на базисні функції B-сплайну $N_i^k(t)$ та, відповідно, на сам B-сплайн. Єдина вимога до вузлового вектора: $x_i \leq x_{i+1}$, тобто це монотонно зростаюча послідовність дійсних чисел. Зазвичай використовуються три типи вузлових векторів: рівномірні, відкриті рівномірні (чи відкриті) та нерівномірні. Розмір вузлового вектору n+k+1.

В рівномірному вузловому векторі окремі значення розподілені на однаковій відстані. Наприклад:

[0 1 2 3 4] або [-0.2 -0.1 0 0.1 0.2].

В загальному випадку рівномірні вузлові вектори починаються в нулі та збільшуються на 1 до деякого максимального значення або нормуються в діапазоні від 0 до 1 рівними десятковими значеннями. Наприклад:

[0 0.25 0.5 0.75 1].

Для заданого порядку k рівномірні вузлові вектори породжують періодичні рівномірні функції базису, для яких

$$N_i^k(t) = N_{i-1}^k(t-1) = N_{i+1}^k(t+1).$$

Тобто кожна функція базису — це паралельне переміщення іншої базисної функції (рис. 8.9).

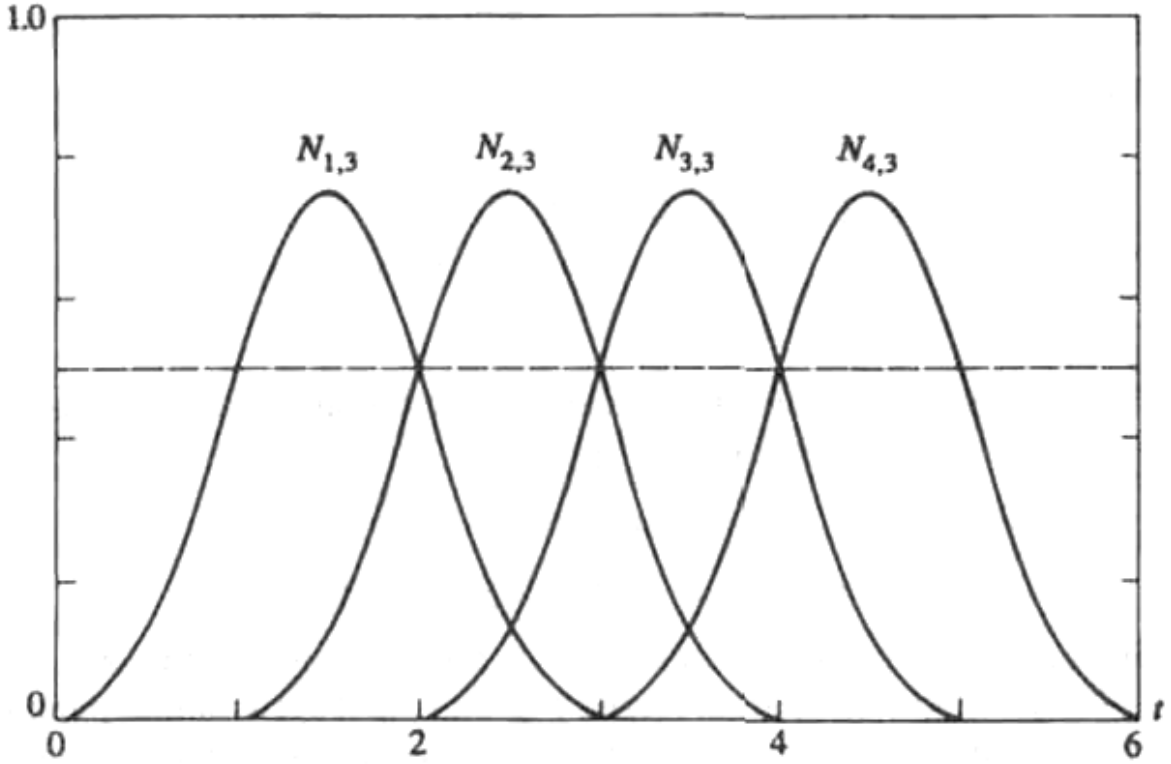


Рис. 8.9. Базисні функції періодичного рівномірного B-сплайну, n+1=4, k=3, [X]=[0 1 2 3 4 5 6]

У відкритого рівномірного вузлового вектора кількість однакових вузлових значень на кінцях рівне порядку k базисної функції B-сплайну. Внутрішні вузлові точки розподілені рівномірно. Декілька прикладів з цілим приростом:

$$\begin{aligned}
&\mathrm{k}=2\ [0 \quad 0 \quad 1 \quad 2 \quad 3 \quad 4 \quad 4],\\
&\mathrm{k}=3\ [0 \quad 0 \quad 0 \quad 1 \quad 2 \quad 3 \quad 3 \quad 3],\\
&\mathrm{k}=4\ [0 \quad 0 \quad 0 \quad 0 \quad 1 \quad 2 \quad 2 \quad 2 \quad 2].
\end{aligned}$$

Формально відкритий вузловий вектор визначається як:

$$\begin{aligned}
&x_i = 0; && 1 \le i \le k;\\
&x_i = i - k; && k+1 \le i \le n+1;\\
&x_i = n - k + 2; && n+2 \le i \le n+k+1.
\end{aligned}$$

Отримувані базисні функції поводяться приблизно так само, як і криві Безьє. Фактично, якщо кількість вершин багатокутника дорівнює порядку базису B-сплайну та використовується відкритий рівномірний вузловий вектор, базис B-сплайну зводиться до базису Бернштейна. Звідси B-сплайн є кривою Безьє. В цьому випадку вузловий вектор — це просто k нулів, за якими йдуть k одиниць. Наприклад, для чотирьох вершин відкритий рівномірний вузловий вектор:

[0 0 0 0 1 1 1 1].

В результаті отримуємо криву Безьє — B-сплайн. Відповідні базисні функції зображені на рис. 8.10.

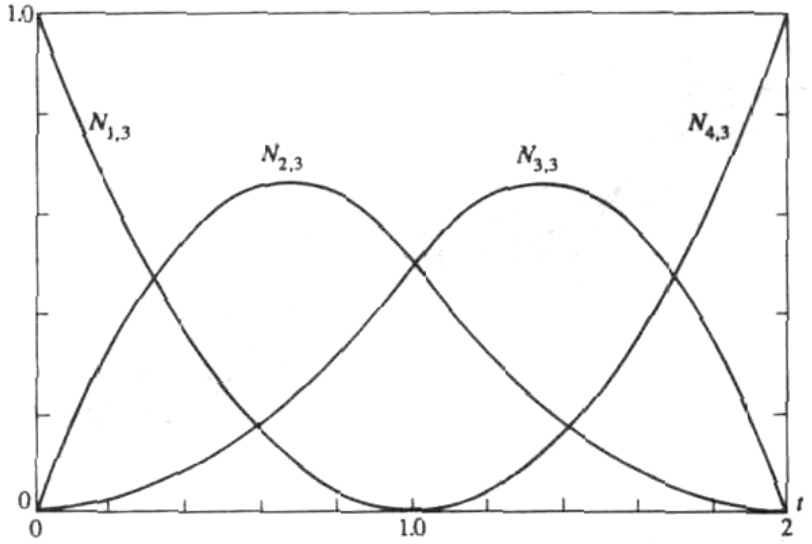


Рис. 8.10. Базисні функції відкритого рівномірного B-сплайну, n+1=4, k=3, [X]=[0 0 0 1 2 2 2]

Нерівномірні вузлові вектори відрізняються тим, що їх внутрішні вузлові величини розташовуються на різній відстані одна від одної та/або суміщаються. Вектори можуть бути періодичними або відкритими, наприклад:

[0 0 0 1 1 2 2 2] або [0 1 2 2 3 4] або [0 0.28 0.5 0.72 1].

На рис. 8.11 показані приклади нерівномірних базисних функцій B-сплайну порядку k=3. У відповідних вузлових векторів на кінцях знаходиться по k суміщених однакових значень. Відмітимо, що у нерівномірних базисів симетрія порушується. Окрім того, у суміщених вузлових значеннях в однієї з функцій з'являється злам. На рисунку видно, що положення зламу залежить від розташування суміщеного значення у вузловому векторі.

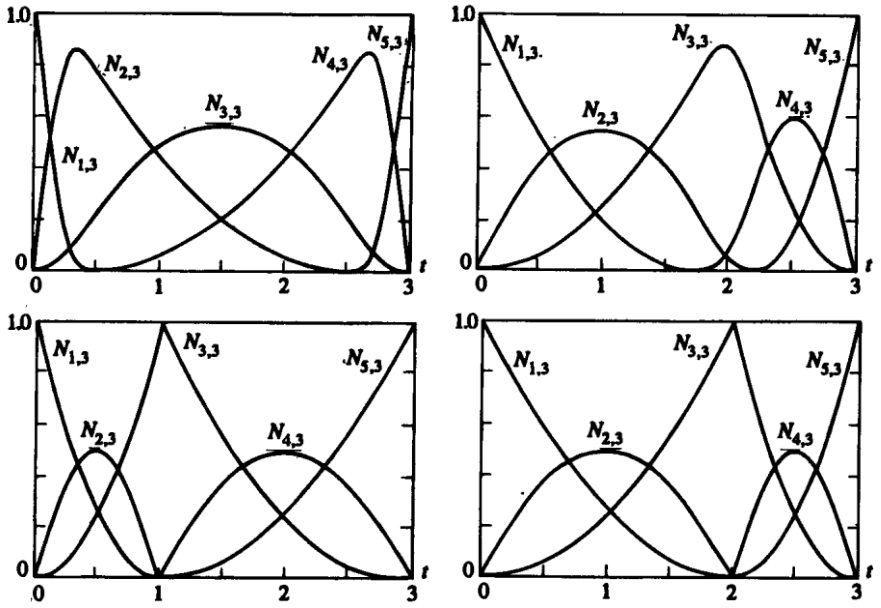


Рис. 8.11. Функції нерівномірного базису для n+1=5, k=3,

[X]=[0 0 0 0.4 2.6 3 3 3], [X]=[0 0 0 1.8 2.2 3 3 3]

[X]=[0 0 0 1 1 3 3 3], [X]=[0 0 0 2 2 3 3 3]

Вибір вузлового вектора впливає на вигляд базисних функцій B-сплайну, а значить і на форму B-сплайну.

Гнучкість B-сплайну дозволяє впливати на форму кривої різноманітними способами:

- змінюючи тип вузлового вектора: періодичний рівномірний, відкритий рівномірний і нерівномірний;
- змінюючи порядок k базисних функцій;
- змінюючи кількість і розташування вершин визначаючого багатокутника;
- використовуючи повторювані вершини;
- використовуючи повторювані вузлові значення у вузлових векторах.

Коротко розглянемо ці способи для всіх трьох видів B-сплайнів. Почнемо з відкритих періодичних B-сплайнів, оскільки вони за своїми властивостями багато в чому схожі з кривими Безьє.

Як уже зазначалося, якщо порядок B-сплайну дорівнює кількості вершин визначаючого багатокутника, то базис B-сплайну зводиться до базису Бернштейна, а сам B-сплайн стає кривою Безьє. У відкритого B-сплайну будь-якого порядку ($к \geq 2$) *перша* і *остання* точки кривої *збігаються* з відповідними вершинами багатокутника, а нахил кривої в першій і останній вершинах багатокутника дорівнює нахилу відповідних сторін багатокутника.

На рис. 8.12 зображені три відкритих B-сплайни різного порядку, задані одним набором з чотирьох вершин. Крива четвертого порядку — це крива Безьє — один кубічний поліноміальний сегмент. Крива третього порядку складається з двох параболічних сегментів, що з'єднуються в центрі другого відрізка з неперервністю C^1. Крива другого порядку співпадає з визначальним багатокутником. Вона складається з трьох лінійних сегментів, що з'єднуються в другій та третій вершинах з неперервністю C^0. Кут нахилу на кінцях, заданий нахилом сторін багатокутника, однаковий для всіх трьох кривих. Відмітимо також, що в міру зростання порядку кривої, вона все менше нагадує початковий багатокутник і стає більш гладкою.

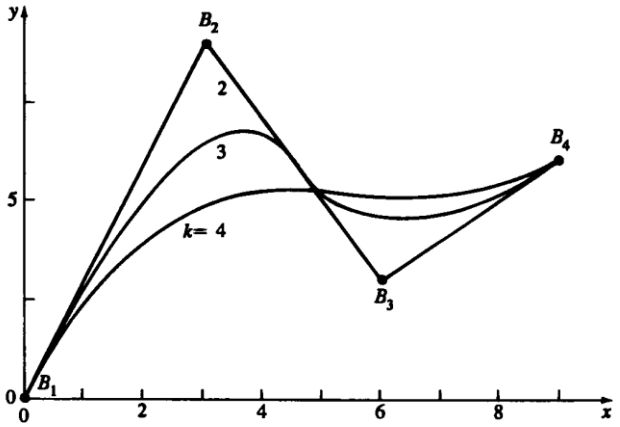


Рис. 8.12. Залежність форми відкритого B-сплайну від його порядку

Щоб сегменти кривої «притягувалися» до вершин визначального багатокутника, необхідно повторювати вершину декілька разів під час розрахунку В-сплайну. Чим більше разів повторюється вершина, тим ближче до неї «притягується» крива, і, за кратності k-1, вона співпадає з вершиною визначального багатокутника.

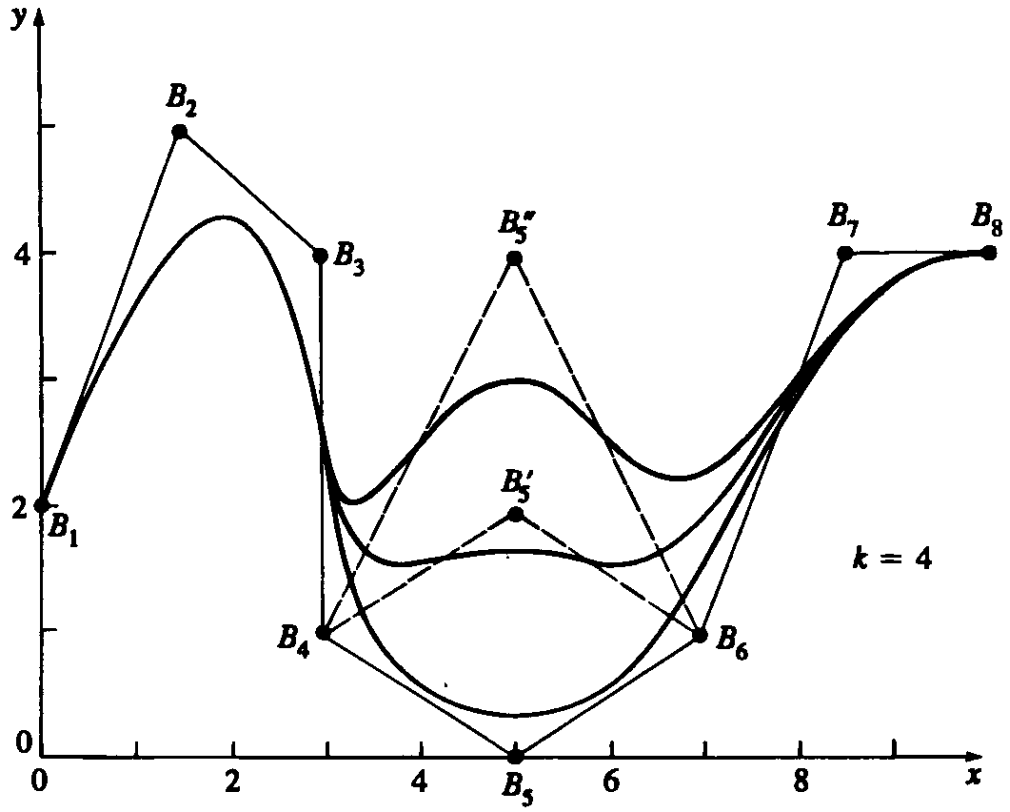


Рис. 8.13. Локальна корекція В-сплайну

На рис. 8.13 показані три В-сплайни четвертого порядку. Кожен визначальний багатокутник складається з восьми вершин. Криві відрізняються тим, що точка B_5 пересувається в B'_5 і B''_5. Пересування точки B_5 впливає на криву тільки локально: змінюються лише сегменти, що відповідають відрізкам B_3B_4, B_4B_5 та B_5B_6, B_6B_7.

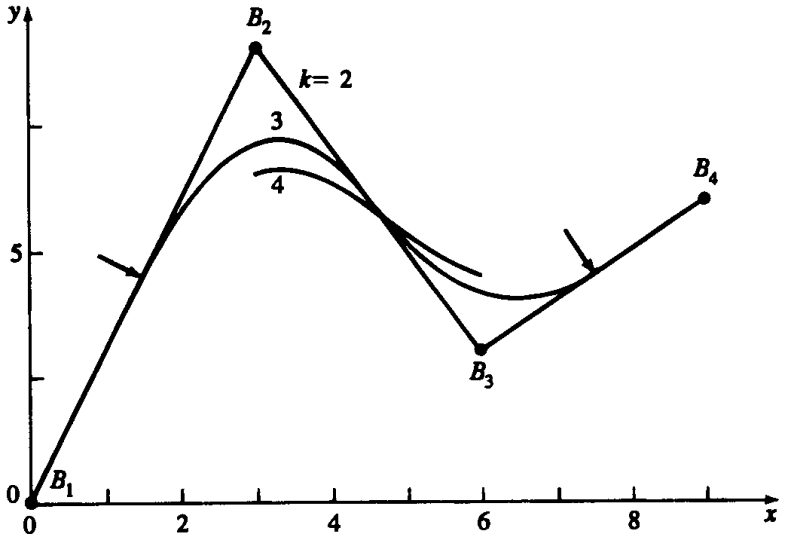


Рис. 8.14. Залежність форми періодичного В-сплайну від його порядку

Тепер розглянемо періодичні B-сплайни. На рис. 8.14 показані три періодичних B-сплайни різного порядку. Всі криві визначені тими самими вершинами, що і для відкритих B-сплайнів (див. рис. 8.12). Для k=2 B-сплайн знову збігається з визначальним багатокутником. Відзначимо, однак, що у періодичного B-сплайну для k>2 *перша* і *остання* точки на кривій *не збігаються* з першою і останньою точками багатокутника. Нахил в першій і останній точках також може відрізнятися від нахилу відповідних сторін багатокутника. Для k=3 B-сплайн починається в середині першого ребра і закінчується в середині останнього, як зазначено стрілками. Це відбувається через скорочення діапазону параметра для базисних функцій періодичного B-сплайну. Для k=2 періодичний вузловий вектор — `[0 1 2 3 4 5]` з діапазоном параметра `1<t<4`. Для k=3 періодичний вузловий вектор — `[0 1 2 3 4 5 6]` з діапазоном параметра `2<t<4`. Для k=4 періодичний вузловий вектор — `[0 1 2 3 4 5 6 7]` з діапазоном параметра `3<t<4`.

Порівняння отриманих результатів і результатів для відкритих вузлових векторів (див. рис. 8.12) показує, що криву можна визначити на повному діапазоні параметра, задаючи кратні вузлові значення на кінцях векторів. Водночас крива розтягується до кінців багатокутника.

В розглянутому випадку, крива четвертого порядку знову складається з єдиного кубічного сегмента; третього порядку — з двох параболічних сегментів, що з'єднані в середині другого ребра з неперервністю C^1; другого порядку — з трьох лінійних сегментів, з'єднаних в другій та третій вершах з неперервністю C^0. Збільшення порядку знову згладжує криву, але в той же час і вкорочує її.

Використання кратних вершин під час розрахунку періодичного B-сплайну дає той самий ефект, що і для відкритих B-сплайнів.

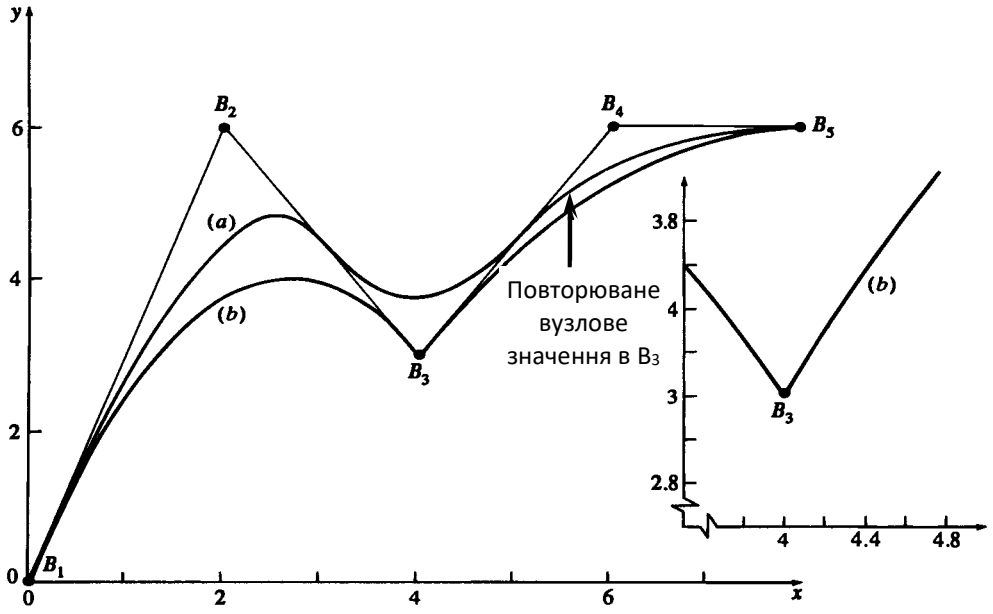


Рис. 8.15. Нерівномірні B-сплайни, k=3

Тепер розглянемо нерівномірні B-сплайни. На рис. 8.15 крива змінюється під впливом кратних внутрішніх вузлових значень. Верхня крива третього порядку (k=3) розрахована для відкритого вузлового вектора [0 0 0 1 2 3 3 3]. Базисні функції для цієї кривої зображені на рис. 8.10. Нижня крива третього порядку побудована з нерівномірним вузловим вектором [0 0 0 1 1 3 3 3]. Її базисні функції — на рис. 8.11.

З рис. 8.15 видно, що кратні внутрішні вузлові значення породжують злам у вершині B_3. Кратне значення породжує ребро нульової довжини, тому зменшується діапазон підтримки базисних функцій. Далі, кратні внутрішні вузлові значення, на відміну від кратних вершин багатокутника, знижують диференційованість базисної функції в x_i до C^{k-m-1}, де $m \leq k-1$ рівне кратності внутрішнього вузлового значення. Локально нерівномірна крива на (рис. 8.15) C^0 ($k-m-1=3-2-1=0$) неперервна в околиці B_3, що і призводить до появи кута.

В цілому нерівномірні B-сплайни не дуже відрізняються від рівномірних у разі невеликої зміни відносної відстані між вершинами.

8.2.3. Раціональні B-сплайни

Вперше в комп'ютерній графіці опис раціональних кривих і поверхонь був запропонований в роботі Кунса в 1967 році. В літературі широко відомі раціональні форми кубічних сплайнів і кривих Безьє, а також конічних перетинів. Тут ми розглянемо лише раціональні B-сплайни, оскільки вони складають загальноприйняту основу. Раціональні B-сплайни — це єдине точне математичне подання, що охоплює всі аналітичні форми — прямі, площини, конічні перерізи, що включають кола, криві довільної форми, квадрики та тривимірні поверхні, що використовуються в обчислювальній геометрії та комп'ютерній графіці.

Першим раціональні B-сплайни вивчив Веспрілл. Варто відзначити, що нерівномірні раціональні B-сплайни (*NURBS*) з 1983 р. є стандартом IGES. IGES — це стандарт обміну проєктною інформацією між системами комп'ютерного проєктування, а також між ними і системами автоматизації виробництва.

Раціональний B-сплайн це проєкція нераціонального (поліноміального) сплайна, визначеного в чотиривимірному (4D) однорідному координатному просторі, на тривимірний (3D) фізичний простір. Зокрема:

$$P(t) = \sum_{i=1}^{n+1} B_i^h N_i^k(t), \qquad (8.6)$$

де B_i^h — вершини багатокутника для нераціонально 4D B-сплайна в чотиривимірному просторі, $N_i^k(t)$ — функція базису нераціонального B-сплайну з рівняння (8.5).

Раціональний B-сплайн отримується після проєктування, тобто ділення на однорідну координату:

$$P(t) = \frac{\sum_{i=1}^{n+1} B_i h_i N_i^k(t)}{\sum_{i=1}^{n+1} h_i N_i^k(t)} = \sum_{i=1}^{n+1} B_i R_i^k(t), \quad (8.7)$$

де B_i –— вершини тривимірного багатокутника для раціонального B-сплайну, а R_i^k — базисні функції раціонального B-сплайну:

$$R_i^k(t) = \frac{h_i N_i^k(t)}{\sum_{i=1}^{n+1} h_i N_i^k(t)}. \quad (8.8)$$

тут $h_i \geq 0$ для всіх i.

Як видно з рівнянь (8.6)-(8.8), раціональні B-сплайни та їх базиси — це узагальнення нераціональних B-сплайнів та базисів. Вони успадковують майже всі аналітичні та геометричні властивості останніх. Зокрема:

- кожна функція раціонального базису додатна або рівна нулю для всіх значень параметрів, тобто $R_i^k \geq 0$;
- для будь-якого значення параметра t сума базисних функцій раціонального B-сплайну дорівнює одиниці, тобто:

$$\sum_{i=1}^{n+1} R_i^k(t) \equiv 1; \quad (8.9)$$

- окрім $k=1$ кожна раціональна базисна функція має рівно один максимум:
- раціональний B-сплайн порядку k (степені $k-1$) скрізь C^{k-2} неперервний;
- максимальний порядок раціонального B-сплайну дорівнює кількості вершин визначального багатокутника;
- раціональний B-сплайн володіє властивістю зменшення варіації;
- загальна форма раціонального B-сплайну повторює контури визначального багатокутника;
- будь-яке проєктивне перетворення раціонального B-сплайну виконується відповідним перетворенням вершин визначального багатокутника, тобто крива інваріанта відносно проєктного перетворення. Це більш сильна умова, ніж для нераціонального B-сплайну, який інваріантний тільки відносно афінного перетворення.

Нераціональні B-сплайни є частковим випадком раціональних. Раціональні B-сплайни — це чотиривимірне узагальнення нераціональних B-сплайнів, а їх базиси можна отримати за допомогою відкритих рівномірних, періодичних рівномірних і нерівномірних вузлових векторів.

Однорідні координати h_i (що також називаються вагами) в рівняннях (8.7) і (8.8) надають додаткові можливості вигину кривої; $h=1$ називається афінним

простором і відповідає фізичному простору. Для h=1 крива раціонального B-сплайну співпадає з кривою нераціонального.

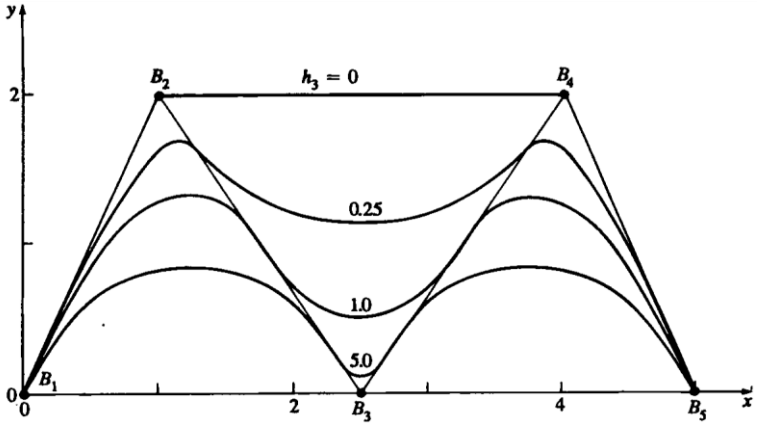


Рис. 8.16. Раціональні B-сплайни для n+1=5, k=3 з відкритим вузловим вектором [X]=[0 0 0 1 2 3 3 3], [H]=[1 1 h_3 1 1]

Раціональний B-сплайн для h_3=1 (рис. 8.16) збігається з відповідним нераціональним. Відзначимо, що для h_3=0 R_3^3 скрізь дорівнює нулю; тобто відповідна вершина B_3 не чинить ніякого впливу на форму відповідної кривої. Тому вершини визначального багатокутника B_2 і B_4 з'єднані прямою. Під час збільшення h_3 також зростає R_3^3 і, внаслідок рівняння (8.9), R_2^3 та R_4^3 зменшуються. На рис. 8.16 зображено вплив на відповідні раціональні B-сплайни. Зокрема зазначимо, що зі збільшенням h_3 крива наближається до B_3. Звідси, як уже зазначалося, випливає, що однорідні координати дають можливість збільшити гнучкість кривої. Однак для B-сплайнів більш високого порядку крива для h_3=0 не вироджується у відрізок прямої між точками B_2 і B_4.

Як і для нераціональних кривих, k-1 кратна вершина призводить до появи гострого кута або піку. Кратна вершина породжує ребра нульової довжини, тому існування кута не залежить від значень h_3>0, що відповідають їй.

Запитання для самоконтролю

1. Які способи моделювання кривих ви знаєте?
2. Дайте визначення поняттю «інтерполяція».
3. Дайте визначення поняття «апроксимація».
4. Назвіть основні методи інтерполяції.
5. Що таке сплайн?
6. Назвіть основні методи апроксимації.
7. Назвіть різновиди вузлових векторів для B-сплайнів. Які з них відповідають кривим Безьє і за якого порядку?
8. Що таке раціональний B-сплайн? Які його переваги?

9. МОДЕЛЮВАННЯ ПОВЕРХОНЬ

В обчислювальній геометрії існує декілька методів моделювання поверхонь. Перший метод — обертання плоских фігур навколо осі. В результаті отримується поверхня обертання. Другий метод — переміщення об'єкта вздовж деякої кривої. Такі поверхні називаються поверхнями перенесення. Третя група методів заснована на побудові поверхонь на заданій множині точок. Методи цієї групи розглянемо в цьому розділі більш детально.

9.1. БІЛІНІЙНІ ПОВЕРХНІ

Однією з найпростіших є білінійна поверхня. Вона конструюється із чотирьох кутових точок одиничного квадрата в параметричному просторі, тобто з точок P(0,0), P(0,1), P(1,1) і P(1,0). Будь-яка точка на поверхні визначається лінійною інтерполяцією між протилежними границями одиничного квадрата (рис. 9.1).

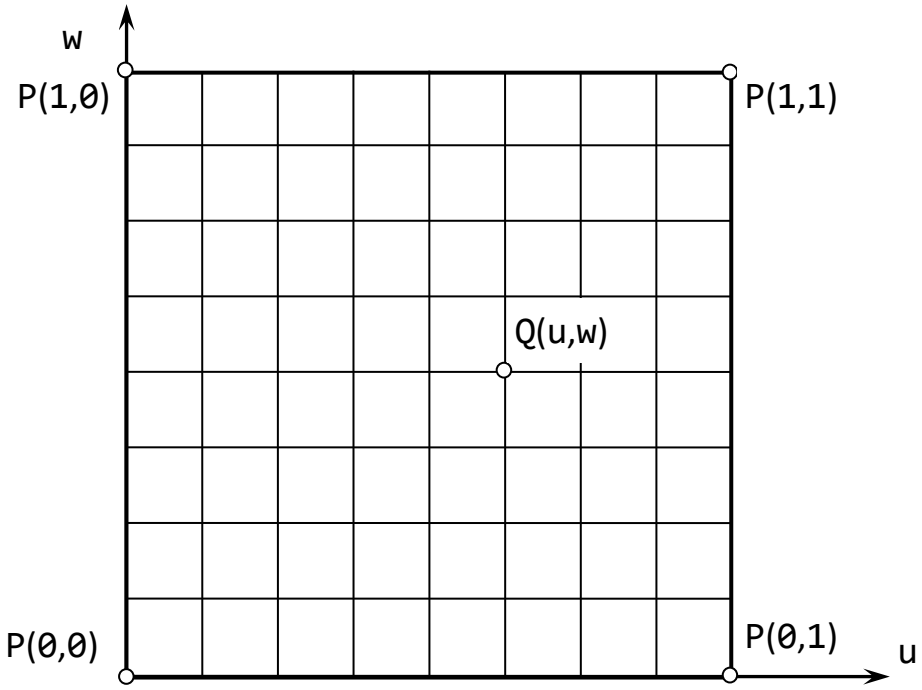


Рис. 9.1. Білінійна інтерполяція в параметричному просторі

Будь-яка точка всередині параметричного квадрата задається рівнянням:

$$Q(u,w) = P(0,0)(1-u)(1-w) + P(0,1)(1-u)w + P(1,0)u(1-w) + P(1,1)uw.$$

В матричному вигляді його можна записати так:

$$Q(u,w) = \begin{bmatrix} 1-u & u \end{bmatrix} \begin{bmatrix} P(0,0) & P(0,1) \\ P(1,0) & P(1,1) \end{bmatrix} \begin{bmatrix} 1-w \\ w \end{bmatrix}. \qquad (9.1)$$

Легко перевірити, що кутові точки належать поверхні, оскільки Q(0,0)=P(0,0).

Рівняння (9.1) задане в узагальненому матричному вигляді інтерпольованої поверхні, тобто складається з трьох матриць: матриці функцій змішування по одній з біпараметричних змінних; геометричної матриці, що містить початкові дані; матриці функцій змішування по іншій біпараметричній змінній.

Якщо координатні вектори точок, що визначають білінійну поверхню, задані в тривимірному об'єктному просторі, то і сама поверхня, після відображення параметричного простору в об'єктний, буде тривимірною. Якщо чотири визначаючих точки не лежать в одній площині, то і білінійна поверхня не лежатиме в жодній з площин. Дійсно, в загальному випадку вона сильно вигнута (рис. 9.2). На рисунку визначальні точки лежать на кінцях протилежних діагоналей одиничного куба. В результаті ми отримали гіперболічний параболоїд.

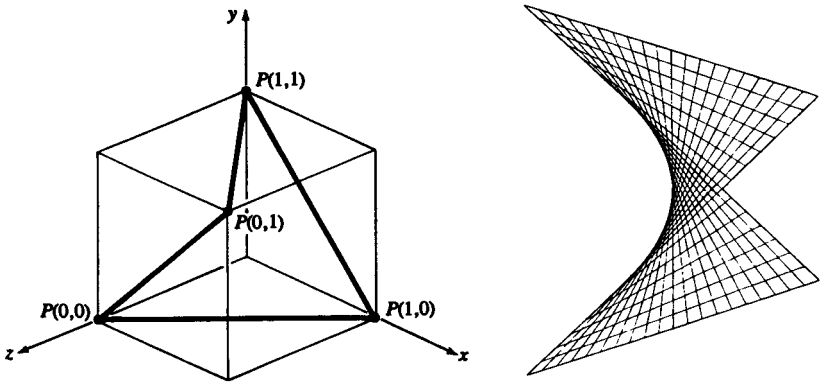


Рис. 9.2. Білінійна поверхня з визначальними точками на кінцях протилежних діагоналей одиничного куба

Кожна ізопараметрична лінія на білінійній поверхні є відрізком прямої. Така поверхня називається дволінійчастою.

9.2. Поверхні Безьє

Логічним розширенням кривих Безьє для двох параметрів є поверхні Безьє. Декартове або тензорне подання поверхні Безьє задається у вигляді:

$$Q(u,w) = \sum_{i=0}^{n}\sum_{j=0}^{m} B_{i,j} J_i^n(u) K_j^m(w), \qquad (9.2)$$

де $J_i^n(u)$ і $K_j^m(w)$ — базисні функції Бернштейна в параметричних напрямках u та w. Як і для кривих Безьє, розпишемо базисні функції Бернштейна для поверхні:

$$J_i^n(u) = C_i^n u^i (1-u)^{n-i};\ K_j^m(w) = C_j^m w^j (1-w)^{m-j},$$

де C_i^n та C_j^m — біноміальні коефіцієнти у відповідних параметричних напрямках. Вони розраховуються за формулами:

$$C_i^n(u) = \frac{n!}{i!\,(n-i)!};\ C_j^m(w) = \frac{m!}{j!\,(m-j)!}.$$

Елементи $B_{i,j}$ в формулі (9.2) є вершинами полігональної сітки, що визначає криву (рис. 9.3). Індекси n і m на одиницю менші кількості вершин багатогранника у напрямках u та w відповідно. Для чотирибічних частин поверхні визначальна полігональна сітка повинна бути топологічно прямокутною, тобто повинна мати однакову кількість вершин у кожному «ряді».

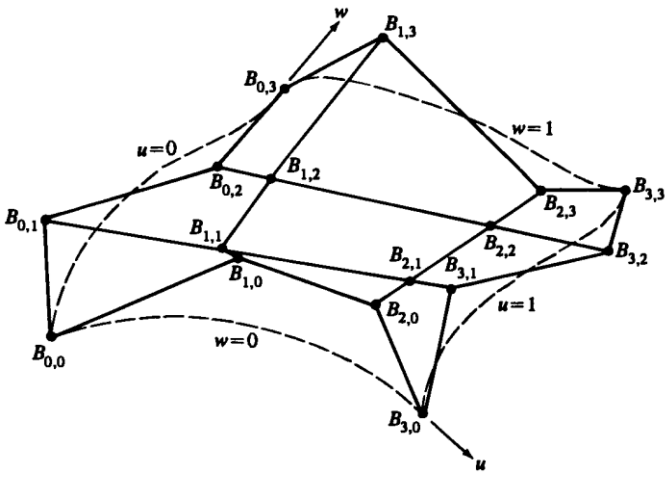


Рис. 9.3. Поверхня Безьє та вершини характеристичного багатогранника

Як і для кривих Безьє так і для поверхонь для змішування використовуються функції Бернштейна, тому багато властивостей поверхні відомі. Наприклад:

- степінь поверхні у кожному параметричному напрямку на одиницю менше за кількість вершин визначального багатогранника у цьому напрямку;
- гладкість поверхні у кожному параметричному напрямку на дві одиниці менше кількості вершин визначального багатогранника у цьому напрямку;
- у загальному вигляді поверхня відтворює форму визначальної полігональної сітки;
- співпадають тільки кутові точки визначальної полігональної сітки та поверхні;
- поверхня міститься усередині опуклої оболонки визначальної сітки;
- поверхня інваріантна по відношенню до афінного перетворення.

Розглянемо визначальну полігональні сітку для бікубічної поверхні Безьє розміром 4×4 (рис. 9.4).

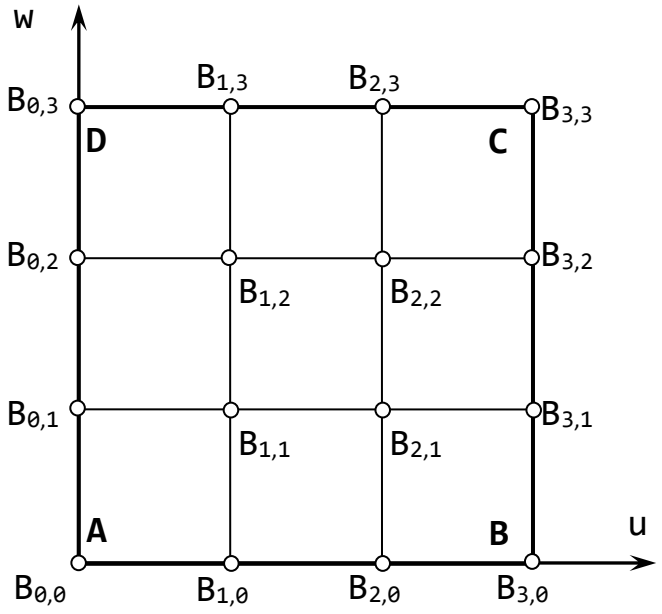


Рис. 9.4. Схема визначаючої полігональної сітки 4×4 для поверхні Безьє

Кожна з граничних кривих поверхні Безьє є кривою Безьє. Напрямок і величина векторів дотичних у кутових точках частин поверхні керуються положенням сусідніх точок вздовж сторін сітки. Вектори дотичних у напрямках u, w в точці A керуються вершинами полігональної сітки $B_{0,1}$та $B_{1,0}$ відповідно. Аналогічним чином, вершини полігональної сітки $B_{2,0}$ $B_{3,1}$ $B_{3,2}$ $B_{2,3}$ та $B_{1,3}$ $B_{0,2}$ керують дотичними векторами у кутових точках B, C, D відповідно. Чотири внутрішні вершини полігональної сітки $B_{1,1}$ $B_{2,1}$ $B_{2,2}$ $B_{1,2}$впливають на напрямок і величину векторів кручення у кутових точках A, B, C та D частин поверхні. Відповідно, користувач може керувати формою частини поверхні, не знаючи конкретних значень векторів дотичних та векторів кручення.

Поверхня Безьє не обов'язково повинна бути квадратною. Наприклад поверхня розміром 5×3 складається з поліноміальних кривих четвертої степені в параметричному напрямку u та із квадратичних поліноміальних кривих в напрямку w.

9.3. B-сплайн поверхні

Природнім розширенням поняття B-сплайну є декартовий добуток B-сплайн поверхні, що визначається за формулою:

$$Q(u,w) = \sum_{i=1}^{n+1} \sum_{j=1}^{m+1} B_{i,j} N_i^k(u) M_j^l(w), \qquad (9.3)$$

де $N_i^k(u)$ та $M_j^1(w)$ — базисні функції B-сплайну в біпараметричних напрямках u та w. Вони визначаються за аналогічними формулами Кокса-де Бура, що і для B-сплайну:

$$N_i^1(u) = \begin{cases} 1, & x_i \le u \le x_{i+1} \\ 0, & \text{інакше} \end{cases},$$

$$N_i^k(u) = \frac{(u - x_i) N_i^{k-1}(u)}{x_{i+k-1} - x_i} + \frac{(x_{i+k} - u) N_{i+1}^{k-1}(u)}{x_{i+k} - x_{i+1}};$$

$$M_j^1(w) = \begin{cases} 1, & y_i \le w \le y_{i+1} \\ 0, & \text{інакше} \end{cases},$$

$$M_j^l(w) = \frac{(w - y_j) M_j^{l-1}(w)}{y_{j+l-1} - y_j} + \frac{(y_{j+l} - w) M_{j+1}^{l-1}(w)}{y_{j+l} - y_{j+1}},$$

де x_i та y_j є елементами вузлових векторів, як і для B-сплайнів. В формулі (9.3) $B_{i,j}$ — вершини визначальної полігональної сітки. Для чотирикутних частин поверхні визначальна сітка повинна бути топологічно прямокутною. Індекси n і m на одиницю менші кількості вершин багатогранника у напрямках u та w відповідно.

Як і для B-сплайн кривих на форму B-сплайн поверхні суттєво впливають вузлові вектори [X] та [Y], причому використовуються *незамкнуті (відкриті)*, *періодичні* та *неоднорідні* вузлові вектори. Зазвичай для обох параметричних

напрямків приймають вузлові вектори одного й того ж типу, але це не обов'язково.

Оскільки для опису граничних кривих та для інтерпретації внутрішньої частини поверхні використовується базис B-сплайну, то відразу можна перерахувати деякі властивості B-сплайн поверхні:

- максимальний порядок поверхні у кожному параметричному напрямку дорівнює числу вершин визначального багатогранника у цьому напрямку;
- гладкість поверхні у кожному параметричному напрямку на дві одиниці менше кількості вершин визначального багатогранника у цьому напрямку, тобто C^{k-2} та C^{l-2} в напрямках u і w відповідно;
- поверхня інваріантна відносно афінного перетворення, тобто поверхня створюється за допомогою перетворення визначальної полігональної сітки;
- якщо кількість вершин полігональної сітки дорівнює порядку у кожному параметричному напрямку та внутрішні вузлові вершини відсутні, то B-сплайн поверхня перетворюється у поверхню Безьє;
- під час тріангуляції визначальна полігональна сітка створює плоску апроксимацію поверхні;
- поверхня міститься всередині опуклої оболонки, що задає полігональну сітку, яка створюється об'єднанням усіх опуклих оболонок k, l сусідніх вершин полігональної сітки.

З властивостей оболонки B-сплайн кривих слідує, що B-сплайн поверхні можуть містити плоскі області та лінії різкого порушення гладкості.

9.4. Раціональні B-сплайн поверхні

Декартовий добуток раціональної B-сплайн поверхні у чотиривимірному просторі однорідних координат задається формулою:

$$Q(u,w) = \sum_{i=1}^{n+1} \sum_{j=1}^{m+1} B_{i,j}^{h} N_i^k(u) M_j^l(w),$$

де $B^h_{i,j}$ — є 4D однорідними вершинами визначального багатогранника, а $N_i^k(u)$ та $M_j^1(w)$ — нераціональні базисні функції B-сплайну в біпараметричних напрямках u та w.

Проєктування у тривимірний простір за допомогою ділення на однорідну координату дає раціональну B-сплайн поверхню:

$$Q(u,w) = \frac{\sum_{i=1}^{n+1} \sum_{j=1}^{m+1} h_{i,j} B_{i,j} N_i^k(u) M_j^l(w)}{\sum_{i=1}^{n+1} \sum_{j=1}^{m+1} h_{i,j} N_i^k(u) M_j^l(w)} = \sum_{i=1}^{n+1} \sum_{j=1}^{m+1} B_{i,j} S_{i,j}(u,w), \qquad (9.4)$$

де $B_{i,j}$ — є 3D-точками визначальної полігональної сітки, $h_{i,j}$ — однорідні координати, а $S_{i,j}(u,w)$ — базисні функції від двох змінних раціональної B-сплайн поверхні. Базисні функції розраховуються за наступними формулами:

$$S_{i,j}(u,w) = \frac{h_{i,j} N_i^k(u) M_j^l(w)}{\sum_{i1=1}^{n+1} \sum_{j1=1}^{m+1} h_{i1,j1} N_{i1}^k(u) M_{j1}^l(w)}. \quad (9.5)$$

Зручно прийняти $h_{i,j} \geq 0$ для всіх i, j.

Тут важливо відмітити, що $S_{i,j}(u,w)$ не є добутком $R_i^k(u)$ та $R_j^l(w)$. Тим паче, $S_{i,j}(u,w)$ мають форму та аналітичні властивості схожі на функцію добутку $N_i^k(u)*M_j^l(w)$. Відповідно раціональні B-сплайн поверхні мають аналітичні та геометричні властивості схожі на нераціональні:

- сума базисних функцій раціональної поверхні для будь-яких значень u, w рівна:

$$\sum_{i=1}^{n+1} \sum_{j=1}^{m+1} S_{i,j}(u,w) \equiv 1;$$

- кожна базисна функція раціональної поверхні додатна або дорівнює нулю для всіх значень параметрів u, w, тобто $S_{i,j} \geq 0$;
- окрім випадку $k=0$ або $l=0$, кожна базисна функція має рівно один максимум;
- максимальний порядок раціональної B-сплайн поверхні в кожному параметричному напрямку рівний кількості вершин визначального багатокутника в цьому напрямку;
- раціональна B-сплайн поверхня порядку k, l (степені k-1, l-1) гладка у всіх точках C^{k-2}, C^{l-2};
- раціональна B-сплайн поверхня інваріантна відносно проєктивного перетворення, тобто будь-яке проєктивне перетворення може бути застосоване до поверхні шляхом його застосування до визначальної полігональної сітки. Ця умова більш строга, ніж для нераціональної B-сплайн поверхні;
- поверхня лежить всередині опуклої оболонки визначальної полігональної сітки, що утворюється об'єднанням всіх опуклих оболонок k, l сусідніх вершин полігональної сітки;
- під час тріангуляції визначальна полігональна сітка створює плоску апроксимацію поверхні;
- якщо кількість вершин визначальної полігональної сітки рівна порядку в кожному параметричному напрямку і дублювання внутрішніх вузлових величин нема, то раціональна B-сплайн поверхня є раціональною поверхнею Безьє.

З формул (9.4) і (9.5) ясно, що коли $h_{i,j}=1$, то $S_{i,j}(u,w)=N_i^k(u)M_j^l(w)$. Таким чином базисні функції раціональних В-сплайн поверхонь і самі поверхні перетворюються в їх нераціональні еквіваленти.

Для генерації раціональних В-сплайн поверхонь можуть використовуватися *незамкнутий однорідний*, *періодичний однорідний* та *неоднорідний* вузлові вектори. Типи вузлових векторів можуть змішуватися.

Приклад бікубічної (k=l=4) раціональної В-сплайн поверхні та її визначальної полігональної сітки для $h_{1,3}=h_{2,3}=0,1,5$ поданий на рис. 9.5.

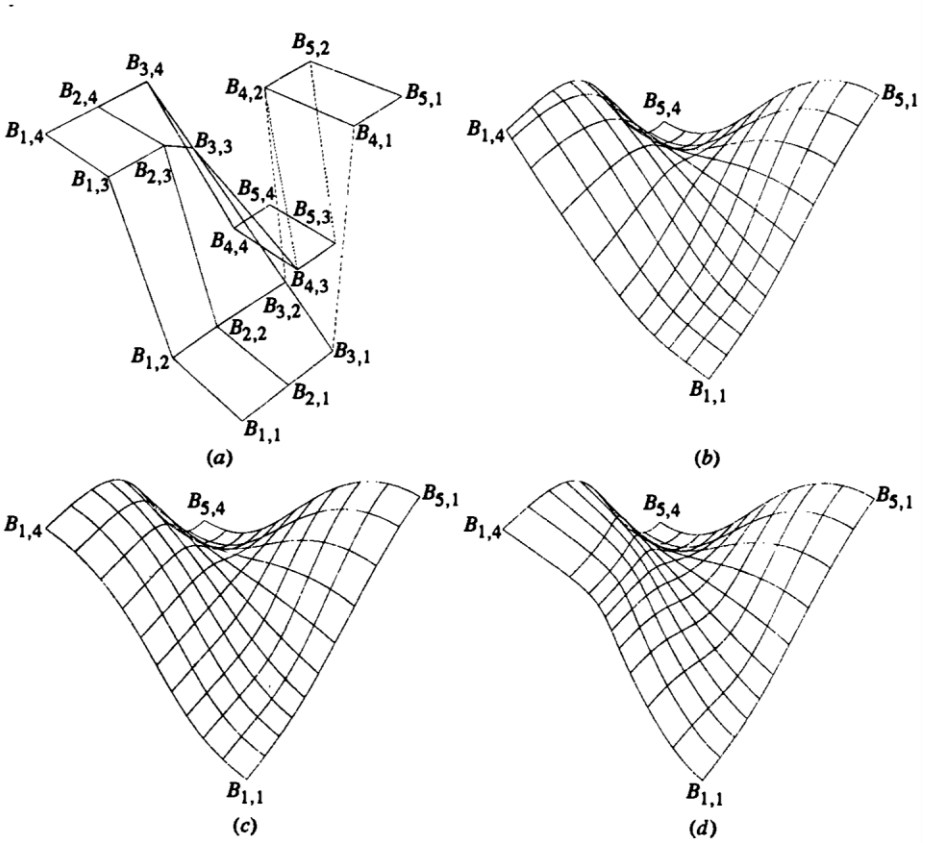


Рис. 9.5. Раціональні В-сплайн поверхні з n+1=5, m+1=4, k=l=4. (a) — визначальний багатогранник; (b) — $h_{1,3}=h_{2,3}=0$; (c) — $h_{1,3}=h_{2,3}=1$; (d) — $h_{1,3}=h_{2,3}=5$

Запитання для самоконтролю

1. Назвіть основні методи побудови поверхонь.
2. Яка поверхня є найпростіша? Який спосіб її побудови?
3. Назвіть основні властивості поверхні Безьє.
4. Назвіть основні властивості В-сплайн поверхні.
5. Яким чином отримуються раціональні В-сплайн поверхні.
6. Назвіть основні властивості раціональних В-сплайн поверхонь.

10. Способи подання полігональних моделей

Під час комп'ютерного моделювання суцільних об'єктів виникає питання способу подання їх моделей. Існують різні підходи до реалізації цієї задачі але ми зупинимося на граничних моделях (*Boundary Representation* — B-rep). За такого підходу суцільний об'єкт подається у вигляді поверхонь, що його обмежують. Зазвичай поверхня наближується набором граней (face). Границі граней подаються *ребрами* (edge). Частини кривої, що формують ребро закінчуються *вершинами* (vertex). Гранична модель, що має лише плоскі грані називається *полігональною*(рис. 10.1). Нижче розглянуті можливі способи подання полігональних моделей.

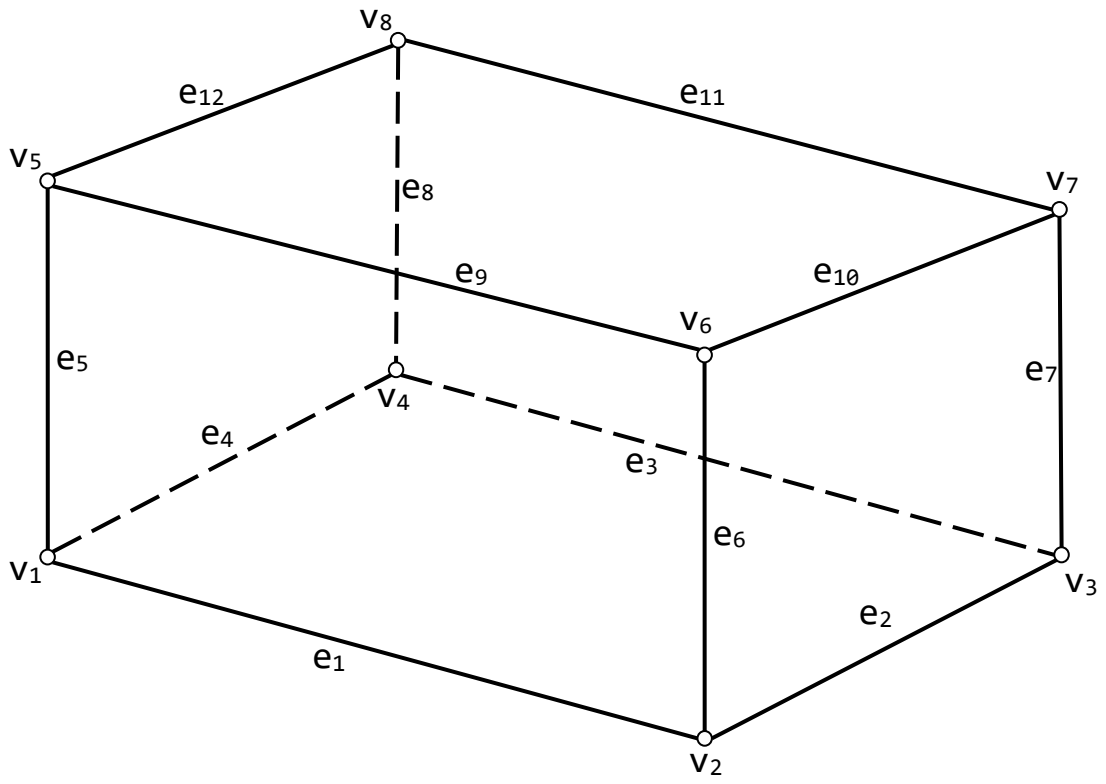


Рис. 10.1. Полігональна модель паралелепіпеда

10.1. Явне подання

В явному поданні об'єкт складається з набору граней, кожна з яких є полігоном, що складається з послідовності координат вершин. Наприклад:

Грані	Координати
f_1	$x_1y_1z_1$, $x_2y_2z_2$, $x_6y_6z_6$, $x_5y_5z_5$
f_2	$x_2y_2z_2$, $x_3y_3z_3$, $x_7y_7z_7$, $x_6y_6z_6$

Недоліки такого подання в тому, що по-перше, взаємовідношення граней задані неявно, а по-друге, координати кожної вершини з'являються стільки разів, скільки граней мають цю вершину.

10.2. Список вершин

Щоб обійти повторюваність координат вершин, можна виокремити їх в самостійну структуру. В такому випадку з гранями асоціюються не координати вершин, як в попередньому випадку, а їх індекси в масиві координат вершин. Наприклад:

Вершини	Координати	Грані	Вершини
v_1	$x_1y_1z_1$	f_1	$v_1v_2v_3v_4$
v_2	$x_2y_2z_2$	f_2	$v_6v_2v_1v_5$
v_3	$x_3y_3z_3$	f_3	$v_7v_3v_2v_6$
v_4	$x_4y_4z_4$	f_4	$v_8v_4v_3v_7$
v_5	$x_5y_5z_5$	f_5	$v_5v_1v_4v_8$
v_6	$x_6y_6z_6$	f_6	$v_8v_7v_6v_5$
v_7	$x_7y_7z_7$		
v_8	$x_8y_8z_8$		

Відмітимо, що список вершин впорядкований за годинниковою стрілкою, якщо дивитися ззовні паралелепіпеда. Таке подання корисно в багатьох алгоритмах, таких як видалення невидимих поверхонь і розрахунок освітленості полігональних граней. Однак в такому поданні залишаються багато недоліків явного. Наприклад, задача пошуку ребер інцидентних заданій вершині все одно вимагає повного перебору.

10.3. Список ребер

В такій моделі грань подається набором ребер і вершини грані визначаються через ребра. Наприклад:

Ребра	Вершини	Вершини	Координати	Грані	Ребра
e_1	v_1v_2	v_1	$x_1y_1z_1$	f_1	$e_1e_2e_3e_4$
e_2	v_2v_3	v_2	$x_2y_2z_2$	f_2	$e_9e_6e_1e_5$
e_3	v_3v_4	v_3	$x_3y_3z_3$	f_3	$e_{10}e_7e_2e_6$
e_4	v_4v_1	v_4	$x_4y_4z_4$	f_4	$e_{11}e_8e_7e_3$
e_5	v_1v_5	v_5	$x_5y_5z_5$	f_5	$e_{12}e_5e_4e_8$
e_6	v_2v_6	v_6	$x_6y_6z_6$	f_6	$e_{12}e_{11}e_{10}e_9$
e_7	v_3v_7	v_7	$x_7y_7z_7$		
e_8	v_4v_8	v_8	$x_8y_8z_8$		
e_9	v_5v_6				
e_{10}	v_6v_7				
e_{11}	v_7v_8				
e_{12}	v_8v_5				

Таким чином для кожного ребра ми задаємо напрямок. Наприклад ребро e_1 направлено (має позитивний напрямок) від точки v_1 до точки v_2. Грані також орієнтовані, тобто ребра задані за годинниковою стрілкою, якщо дивитися на паралелепіпед ззовні.

10.4. WINGED-EDGE REPRESENTATION

Ця модель є розвитком попередньої моделі у вигляді списку ребер. *Winged-Edge Representation* — «крилате подання», розширює список ребер шляхом додавання інформації про взаємне розташування граней (рис. 10.2).

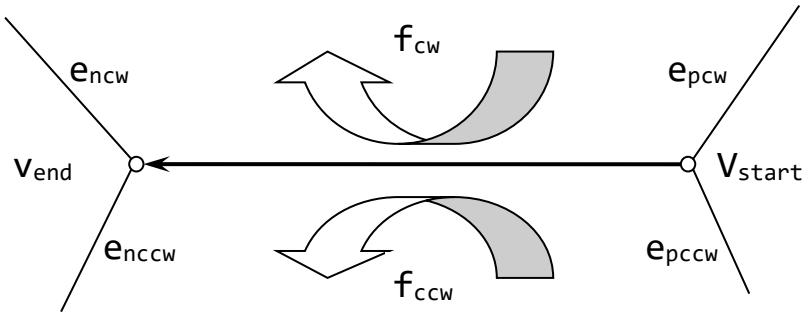


Рис. 10.2. «Крилате» подання

Оскільки кожне ребро з'являється точно в двох гранях, то рівно два ребра з'являються після нього в цих гранях. До того ж один раз ребро з'являється в позитивній орієнтації, а один раз — в негативній.

В «крилатому» поданні використовується асоціація ребра з наступними двома ребрами в гранях. Вони позначаються як ncw (next clockwise) і nccw (next counterclockwise). В цьому випадку ncw означає наступне ребро, що знаходиться в тій грані, де ребро з'являється в позитивному напрямку, а nccw — наступне ребро в іншій грані. Таким чином, починаючи з ребра, що прямо зв'язане з гранню, ми можемо отримати всі інші інцидентні цій грані ребра, слідуючи за посиланнями ncw і nccw. В найбільш загальному випадку в структуру включають також посилання на попередні ребра в сусідніх гранях. Тоді ми отримаємо наступну структуру:

```
struct TEdge
{
    TEdge Encw, Epcw, Enccw, Epccw;
    TFace Fcw, Fccw;
    TVertex Vstart, Vend;
};
```

В цій структурі Encw, Epcw — посилання на наступне і попереднє ребра в грані, куди ребро входить в позитивному напрямку. Аналогічно Enccw, Epccw — наступне і попереднє ребра в грані, що відповідає негативному напрямку ребра.

ЗАПИТАННЯ ДЛЯ САМОКОНТРОЛЮ

1. Що називають полігональною моделлю?
2. Яке найпростіше подання полігональної моделі?
3. Які недоліки подання у вигляді списку вершин?
4. Які переваги подання у вигляді списку ребер?
5. Які дані включені в «крилате» подання?

11. Геометричний пошук

Геометричний пошук — це пошук необхідної інформації серед даних, що описують геометричні об'єкти.

Пошукове повідомлення, відповідно до якого ведеться перегляд файлу, зазвичай іменується *запитом*. Від типу файлу і від набору допустимих запитів сильно залежатимуть організація першого і алгоритми обробки останніх. Один конкретний приклад дозволить переконатися, наскільки важливий цей аспект завдання.

Нехай є набір геометричних даних і потрібно дізнатися, чи володіють вони певною властивістю (наприклад опуклістю). В простому випадку, коли це питання виникає один раз, було б недоцільно проводити попередню обробку даних в надії прискорити виконання наступних запитів. Назвемо разовий запит такого типу *унікальним*. Проте будуть і запити, обробка яких повторюється багато разів на тому ж самому файлі. Такі запити назвемо *масовими*.

В останньому випадку, можливо варто розташувати інформацію відповідно до деякої структури, що полегшує пошук. Проте це можна виконати, лише витративши деякий ресурс, і аналіз треба зосередити на чотирьох різних аспектах його оцінки.

1. *Час запиту*. Скільки часу необхідно як в середньому, так і у найгіршому випадку для відповіді на один запит?
2. *Пам'ять*. Скільки пам'яті необхідно для структури даних?
3. *Час попередньої обробки*. Скільки часу необхідно для організації даних перед пошуком?
4. *Час коригування*. Вказаний елемент даних. Скільки часу буде потрібно на його включення в структуру даних або видалення з неї?

Різні варіанти витрат часу запиту, часу попередньої обробки і пам'яті продемонструємо на прикладах основних завдань геометричного пошуку. Серед них виокремлюють дві основні моделі.

1. Завдання *локалізації*, коли файл є розбиттям геометричного простору на області, а запит є точкою. Локалізація полягає у визначенні області, яка містить точку запиту.
2. Завдання *регіонального пошуку*, коли файл містить набір точок простору, а запит є деяка стандартна геометрична фігура, що довільно переміщується в цьому просторі (типовий запит в тривимірному просторі — куля або брус). Регіональний пошук полягає у вичитуванні (завдання звіту) або в підрахунку числа (завдання підрахунку) всіх точок всередині регіону (області) запиту.

11.1. Підрахунок кількості точок

Задача. Задані N точок на площині. Скільки з них лежить всередині заданого прямокутника, сторони якого паралельні координатним осям? Тобто скільки

точок (x, y) задовольняють нерівностям a≤x≤b, c≤y≤d для вказаних a, b, c і d (рис. 11.1)?

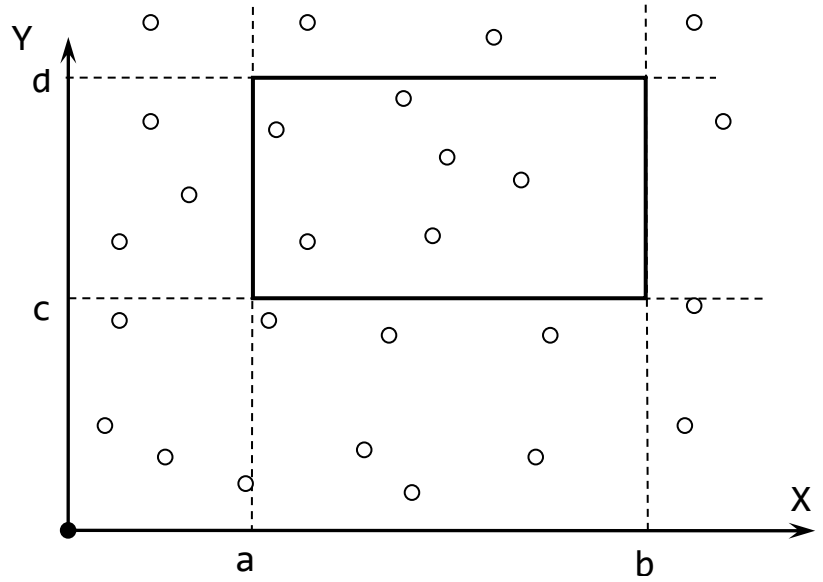


Рис. 11.1. Регіональний запит: скільки точок всередині прямокутника?

Вочевидь, що унікальний регіональний запит може бути оброблений (оптимально) за лінійний час, оскільки треба тільки перевірити кожну з N точок, аби побачити, чи задовольняє вона нерівностям, що задають прямокутник. Аналогічно необхідна лінійна витрата пам'яті, оскільки слід запам'ятати лише 2N координат. Немає жодних витрат на попередню обробку, а час коригування для нової точки дорівнює константі. Яку структуру даних можна використовувати для прискорення обробки масових запитів? Здається, що дуже важко знайти таке впорядкування точок, аби будь-який новий прямокутник міг бути легко з ним узгоджений. Ми не можемо також вирішити це завдання наперед для всіх можливих прямокутників, оскільки їх число нескінченне. Наступне рішення є прикладом використання методу локусів в геометричних задачах: запиту ставиться у відповідність точка в зручному для пошуку просторі, а цей простір розбивається на області (локуси), в межах яких відповідь не змінюється. Іншими словами, якщо вважати еквівалентними два запити, на яких отримуються однакові відповіді, то кожна область розбиття простору пошуку відповідає одному класу еквівалентності запитів.

Прямокутник сам по собі не досить зручний об'єкт; ми вважаємо за краще працювати з точками. Це означає, наприклад, що можна замінити запит з прямокутником чотирма підзадачами, по одній на кожну з його вершин, і поєднати їх вирішення для отримання остаточної відповіді. В цьому випадку підзадача, пов'язана з вершиною p, полягає у визначенні числа точок Q(p) заданої множини, які задовольняють нерівностям x≤x(p) і y≤y(p), тобто числа точок в лівому нижньому квадранті, який визначається вершиною p (рис. 11.2).

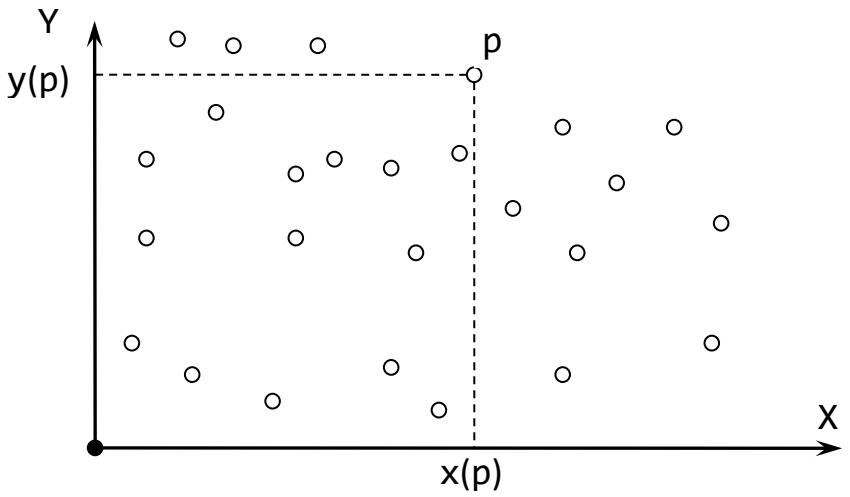


Рис. 13.2. Скільки точок «південно-західніше» p?

Поняття, з яким ми зустрілися тут — *векторне домінування*. Говорять, що точка (вектор) v домінує над w, тоді і тільки тоді, коли для всіх індексів i справджується умова $v_i \geq w_i$. На площині точка v домінує над w тоді і тільки тоді, коли w лежить в лівому нижньому квадранті, що визначається v. Тоді Q(p) — число точок, над якими домінує p. Зв'язок між домінуванням і регіональним пошуком показаний на рис. 11.3.

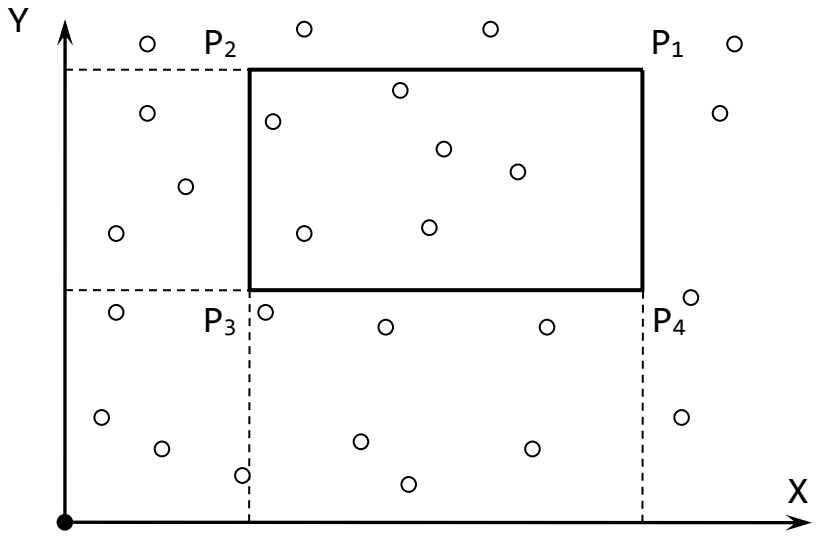


Рис. 11.3. Регіональний пошук у вигляді 4-х запитів про домінування

Кількість точок N в прямокутнику $P_1P_2P_3P_4$ визначається наступним чином:

$$N(P_1P_2P_3P_4) = Q(P_1) - Q(P_2) - Q(P_4) + Q(P_3).$$

Отже, задача регіонального пошуку зведена до задачі обробки запитів про домінування для чотирьох точок. Властивість, яка полегшує ці запити, в тому, що на площині існують області зручної форми, всередині яких число домінування Q є константою.

11.2. Локалізація точки

Задачу локалізації точки можна також назвати задачею про належність точки. Насправді, твердження «точка p лежить в області R» є синонімом до твердження «точка p належить області R». Обчислювальна складність цієї задачі безумовно буде залежати від природи простору та від способу його розбиття.

Ми вже коротко розглядали задачі локалізації точки відносно прямокутника (див. розділ 2.1) та багатокутника (див. розділ 3.2). Тепер більш докладно розглянемо задачу локалізації точки відносно будь-якого багатокутника.

Теорема. Належність точки z внутрішній області простого N-кутника P можна встановити за час O(N) без попередньої обробки.

Вирішення задачі локалізації точки відносно будь-якого багатокутника у випадку унікального запиту наступне:

- проводимо через точку z пряму l, що паралельна осі OX;
- в циклі по черзі перебираємо всі ребра, якщо ребро горизонтальне, то пропускаємо його, якщо ж ні, то перевіряємо його перетин з проведеною прямою l;
- якщо поточне ребро перетинає пряму l ліворуч від точки z, то збільшуємо лічильник на 1;
- перевіряємо значення лічильника: якщо воно непарне, то точка z лежить всередині багатокутника, інакше — зовні.

Очевидно, що цей простий алгоритм виконується за час O(N).

Для масових запитів спочатку розглянемо випадок, коли P — опуклий багатокутник. Пропонований метод використовує опуклість P, а саме властивість, що вершини опуклого багатокутника впорядковані за полярними кутами відносно будь-якої внутрішньої точки. Таку точку q можна легко знайти; наприклад, можна взяти центр мас (центроїд) трикутника, утвореного будь-якою трійкою вершин P. Тепер розглянемо N променів, що виходять з точки q і проходять через вершини багатокутника P (рис. 11.4).

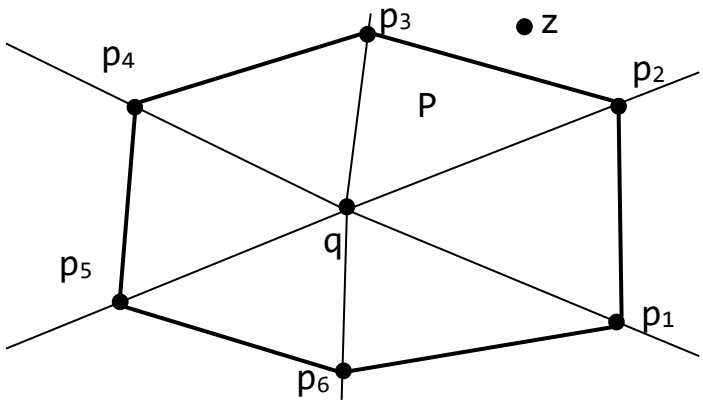


Рис. 11.4. Розбиття на клини для задачі про належність опуклому багатокутнику

Ці промені розбивають площину на N клинів. Кожен клин розбитий на дві частини одним з ребер багатокутника P. Одна з цих частин лежить цілком усередині P, інша — цілком зовні. Вважаючи q початком полярних координат, ми можемо відшукати той клин, де лежить точка z, провівши один раз двійковий пошук, оскільки промені слідують в порядку зростання їх кутів. Після знаходження клину залишається лише порівняти z з тим єдиним ребром з P, яке розрізає цей клин, і вирішити, чи лежить z всередині P.

Алгоритм реалізації описаного методу наступний:

- визначаємо методом двійкового пошуку клин, в якому лежить точка z. Точка z лежить між променями, що визначаються p_i та p_{i+1}, тоді і тільки тоді, коли кут (zqp_{i+1}) позитивний, а кут (zqp_i) від'ємний;
- якщо p_i та p_{i+1} знайдені, то z — внутрішня точка тоді і тільки тоді, коли кут $(p_ip_{i+1}z)$ від'ємний.

Зазначимо, що визначенню знака кута $(p_1p_2p_3)$ відповідає обчислення визначника матриці третього порядку, утвореної координатами цих точок. Конкретно, якщо прийняти $p_i=(x_i,y_i)$, то цей визначник дорівнює:

$$\begin{vmatrix} x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \\ x_3 & y_3 & 1 \end{vmatrix}.$$

Він дає подвоєну орієнтовану площу трикутника $(p_1p_2p_3)$, де знак плюс буде тоді і тільки тоді, коли обхід $(p_1p_2p_3)$ орієнтований проти годинникової стрілки. Отже, $(p_1p_2p_3)$ відповідає лівому повороту тоді і тільки тоді, коли цей визначник додатній.

Теорема. Час відповіді на запит про належність точки опуклому N-кутнику дорівнює $O(\log N)$ із витратою $O(N)$ пам'яті та $O(N)$ часу на попередню обробку.

Для можливості застосування двійкового пошуку необхідно, щоб вершини багатокутника були впорядковані за полярним кутом відносно деякої точки. Очевидно, що опуклість є тільки достатньою умовою для володіння цією властивістю. Насправді ж існує більш широкий клас простих багатокутників, що включає в себе і опуклі багатокутники, який володіє цією властивістю — це клас зірчастих багатокутників (рис. 11.5).

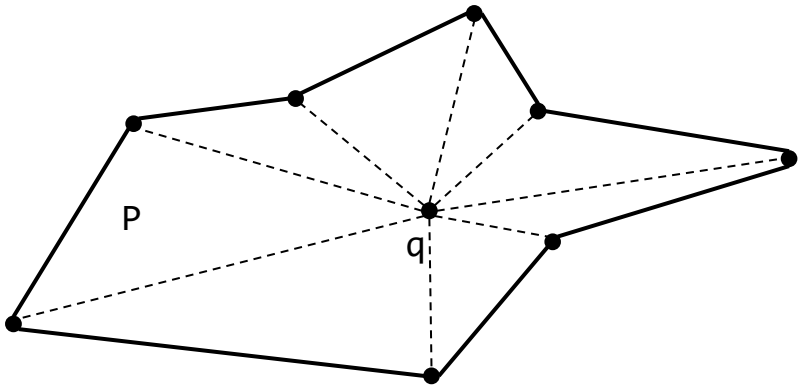


Рис. 11.5. Зірчастий багатокутник

Для визначення належності точки зірчастому багатокутнику можна використати попередній алгоритм для опуклого багатокутника.

Теорема. Час відповіді на запит про належність точки зірчастому N-кутнику дорівнює $O(\log N)$ із витратою $O(N)$ пам'яті та $O(N)$ часу на попередню обробку.

Тепер можна звернути увагу на прості багатокутники загального виду, які називатимемо звичайними. Існує ієрархія властивостей, строго впорядкована відношенням «бути підмножиною»:

$$\text{ВИПУКЛІСТЬ} \subset \text{ЗІРКОВІСТЬ} \subset \text{ЗВИЧАЙНІСТЬ}.$$

Ми тільки що побачили, що задача про належність зірчастому багатокутнику практично анітрохи не складніша за задачу про належність опуклому багатокутнику. Але що можна сказати про звичайний випадок? Один з підходів до цієї задачі підказаний тим, що кожен простий багатокутник є об'єднання деякого числа багатокутників спеціального вигляду — таких, як зірчасті або опуклі, або зрештою трикутників. Як розбити довільний багатокутник на трикутники розглядалося в розділі 7.3.

Запитання для самоконтролю

1. Які основні задачі виокремлюють в геометричному пошуку?
2. Що таке «векторне домінування»?
3. Опишіть алгоритм локалізації точки в опуклому багатокутнику?
4. Назвіть ієрархію властивостей багатокутників.

12. СТРУКТУРИ ПРОСТОРОВОЇ ІНДЕКСАЦІЇ

В попередньому розділі ми ознайомилися з поняттям геометричного пошуку і його основними задачами. В багатьох графічних застосунках часто постає задача локалізувати точку на заданій множині об'єктів. Найпростіший метод вирішення цієї задачі, це послідовний перебір всіх об'єктів по черзі і тестування точки на входження в них. Звичайно такий алгоритм є абсолютно не оптимальним, оскільки в найгіршому випадку нам доведеться протестувати всі об'єкти множини. Для збільшення швидкодії операцій геометричного пошуку використовуються різноманітні структури просторової індексації. Найбільш розповсюджені з них ми розглянемо в цьому розділі.

12.1. БАГАТОВИМІРНІ ДВІЙКОВІ ДЕРЕВА

Ці дерева ще називають kD-дерева (*k-Dimensional Tree*) . Ця абревіатура введена *Дональдом Кнутом* для k-вимірного дерева двійкового пошуку. Метод побудови цього дерева заснований на принципі *дихотомії*.

Дихотомія — послідовний розріз регіону (неважливо, скінченного чи нескінченного) на дві частини. У випадку двох вимірів всю площину можна вважати нескінченим прямокутником, що буде розрізаний спочатку на дві півплощини прямою, паралельною одній з осей. Потім кожна з цих півплощин може розрізатися ще раз прямою, що паралельна іншій осі, і т. д., змінюючи на кожному кроці напрямок лінії розрізу, наприклад від X до Y. Принцип вибору лінії розрізу: у відповідності з принципом, що використовується під час дихотомії, тобто принцип отримання приблизно рівної кількості елементів (точок) у кожній стороні розрізу.

Прямокутником будемо вважати таку область на площині, що визначається декартовим добутком $[x_1,x_2]\times[y_1,y_2]$, включно з граничними випадками, коли у кожній комбінації дозволяється: $x_1=-\infty$, $x_2=\infty$, $y_1=\infty$, $y_2=\infty$. Тому будемо вважати прямокутниками також необмежені (з одної чи двох сторін) смуги, будь-який квадрант, чи навіть всю площину.

Процес розбиття множини S шляхом розрізу площини краще за все проілюструвати в сукупності з побудовою двовимірного двійкового дерева T. З кожним вузлом v дерева T неявним чином зв'язуємо прямокутник R(v) та підмножину точок $S(v)\subseteq S$, що лежать всередині R(v). Явно, як фактичні параметри цієї структури даних, зв'яжемо з v одну обрану точку P(v) з S(v) та «січну пряму» l(v), що проходить крізь P(v) паралельно одній з координатних осей.

Процес починається з визначення кореня T. З R(корінь) співвідноситься вся площина, і вважається, що S(корінь)=S. Потім визначається точка $p\in S$, така що x(p) — медіана множини абсцис точок з S(корінь), та вважається що P(корінь)=p, а з l(корінь) співвідноситься пряма з рівнянням x=x(p). Точка p розбиває S на дві множини приблизно рівної потужності, що назначені нащадкам кореня. Процес

дроблення припиняється, коли знайдено прямокутник, що не містить всередині точок; відповідний йому вузол є листом дерева T.

Цей метод проілюстровано на прикладі (рис. 12.1) для множини з N=11 точок. Вузли трьох різних типів позначені різними графічними символами: колами — нелистові вузли з вертикальною лінією розрізу, квадратами — нелистові вузли з горизонтальною лінією розрізу, а точками — листя. Така структура даних має назву 2D-дерево (абревіатура виразу «двовимірне двійкове дерево пошуку»).

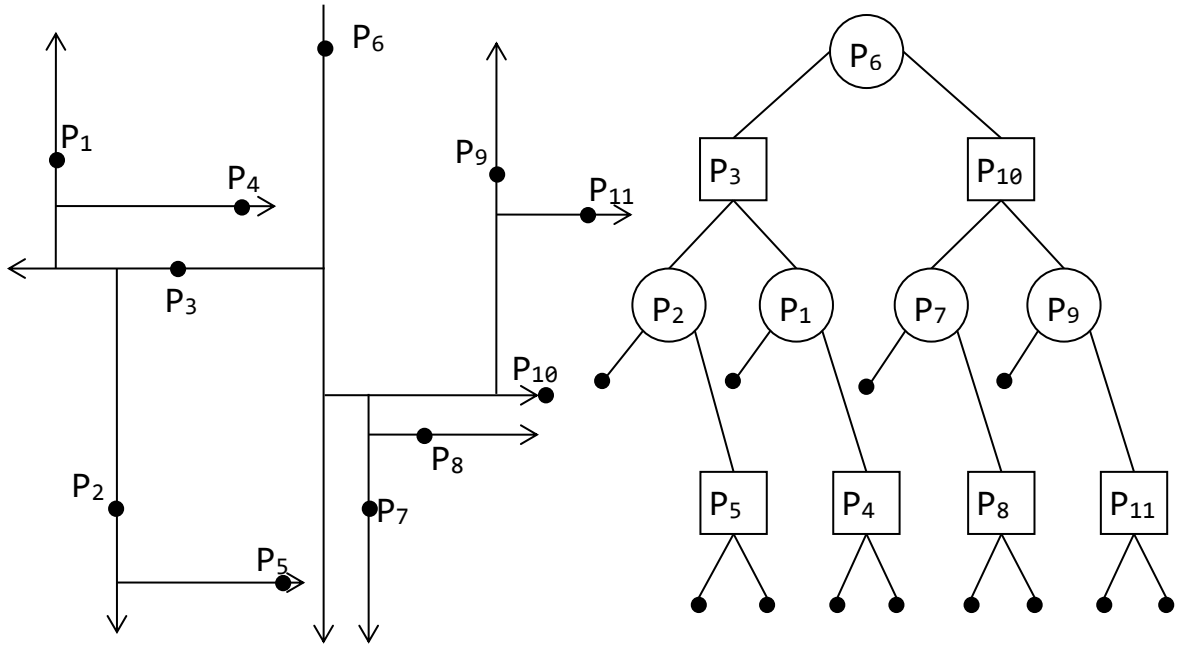


Рис. 12.1. Ілюстрація методу пошуку за допомогою двовимірного двійкового дерева

Пошук починається завжди з кореневого вузла і продовжується вглиб дерева. Ілюстрація регіонального пошуку подана на рис. 12.2.

Час запиту є пропорційним по відношенню до загальної кількості вузлів в T, які відвідує пошуковий алгоритм, оскільки у кожному вузлі цей алгоритм витрачає константний час. Якщо у вузлі v обирається точка P(v) тоді v *продуктивний* вузол; інакше, цей вузол — *непродуктивний*. Кожен вузол v з T відповідає узагальненому прямокутнику R(v).

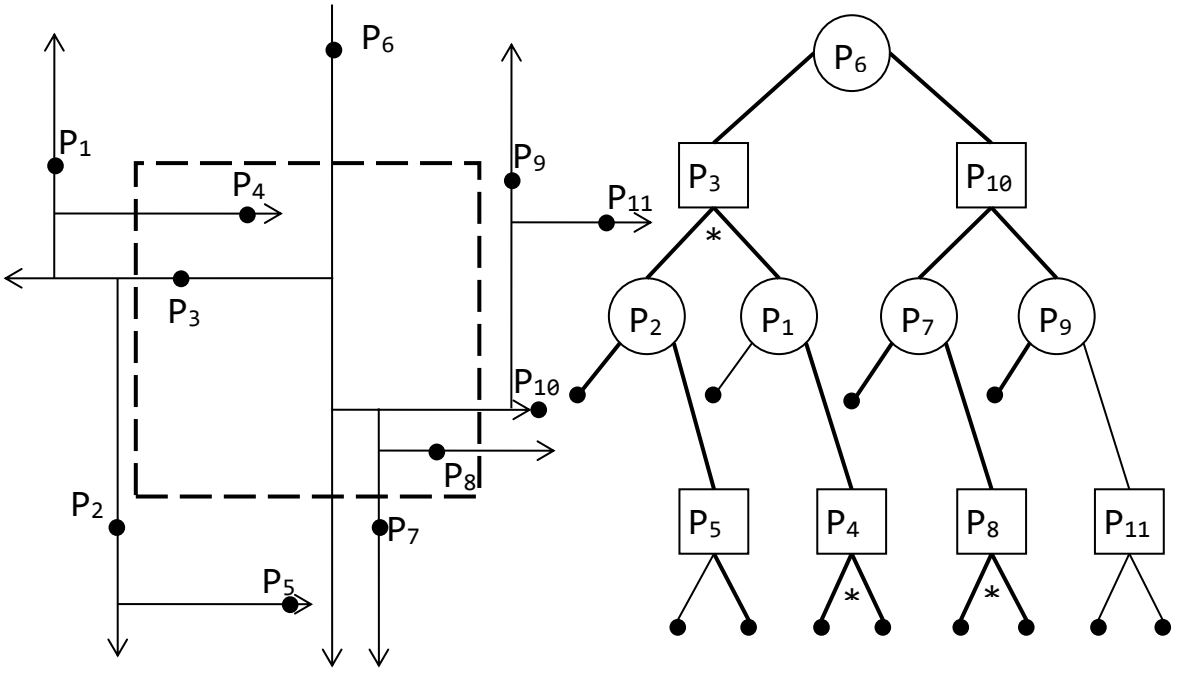


Рис. 12.2. Приклад регіонального пошуку на попередньо заданому файлі

Перетини регіону запиту D і подібного узагальненого прямокутника R(v) можуть бути віднесені до різних «типів» в залежності від числа сторін R(v), що мають непусті перетини з D (рис. 12.3). Єдиний тип перетину, який є завжди продуктивним — це тип 4; всі інші можуть бути непродуктивними.

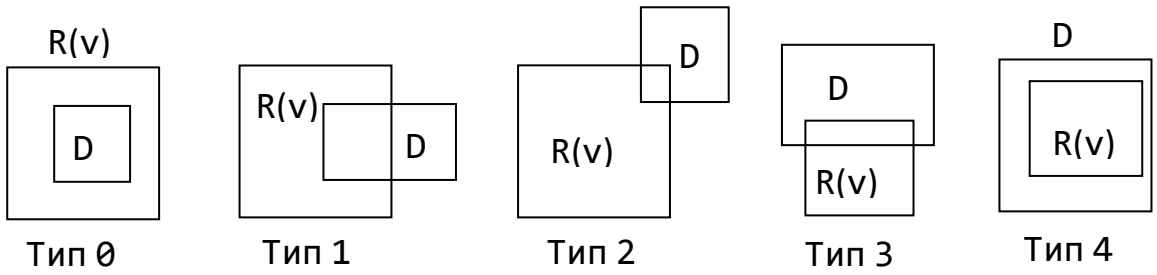


Рис. 12.3. Приклад різних типів перетинів

Оптимальні оцінки по витратам пам'яті і часу попередньої обробки в k-D-дереві, нажаль, не компенсують вкрай погану оцінку часу пошуку для найгіршого випадку.

12.2. Квадро-дерева

Квадро-дерева (*Quadtree* або *Q-tree*) — це дерева, які зберігають інформацію декомпозиції двовимірного простору, тобто квадрати простору. Кожен вузол квадро-дерева може мати не більше чотирьох нащадків. Подібна структура даних, завдяки своїй простоті, часто використовується для прискорення доступу до об'єктів двовимірної площини. На рисунку 12.4 наведено квадро-дерево та зв'язана з ним квадратична декомпозиція простору.

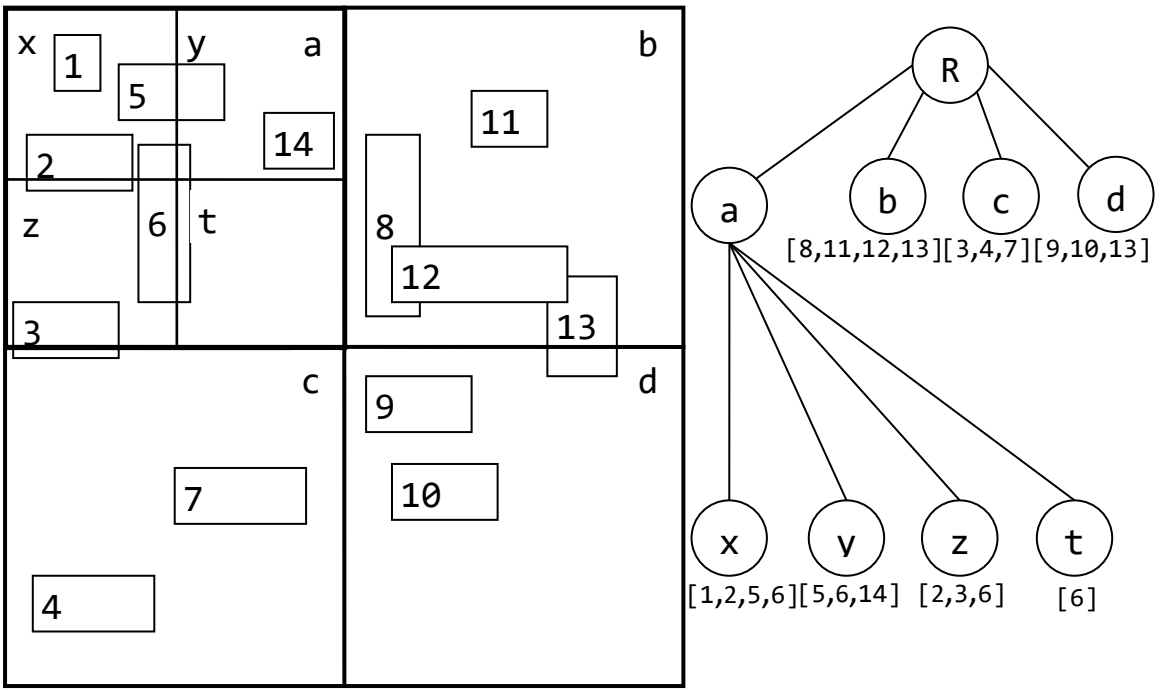


Рис. 12.4. Квадро-дерево

Обробка точкового запиту виконується наступним чином. Дерево перевіряють від кореня до листових сторінок, на кожному рівні серед чотирьох квадрантів обирають той, який містить точку P. Рисунок 12.5 ілюструє обробку точкового запиту.

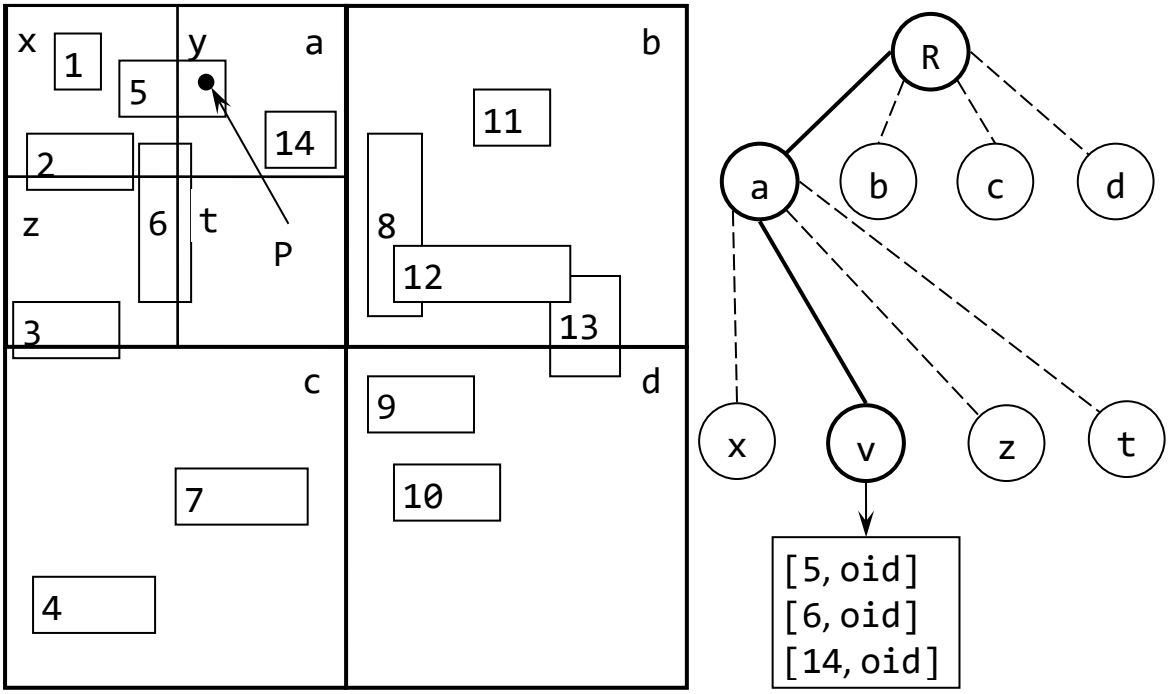


Рис. 12.5. Обробка точкового запиту в квадро-деревах

Розглянемо динамічну вставку прямокутників. Прямокутник повинен бути вставлений у кожен квадрант, який він перекриває. Перевіряються листові сторінки p даних квадрантів. Під час перевірки можливі два випадки: сторінка p заповнена та сторінка p незаповнена (об'єкт додається у дерево). У випадку

переповнення листової сторінки виконується розбиття квадранта. Після розбиття отримуємо чотири листові сторінки, додаємо об'єкт у ті квадранти які перетинаються з ним. Рисунок 12.6 ілюструє вставку об'єктів у квадро-дерево. Вставка об'єкта 16 не призводить до розбиття квадрантів, а вставка об'єкта 15 призводить до розбиття квадранта b на квадранти m, n, p та q. Об'єкт 15 додається у листові сторінки n та q. Об'єкт 16 додається у листові сторінки c і t.

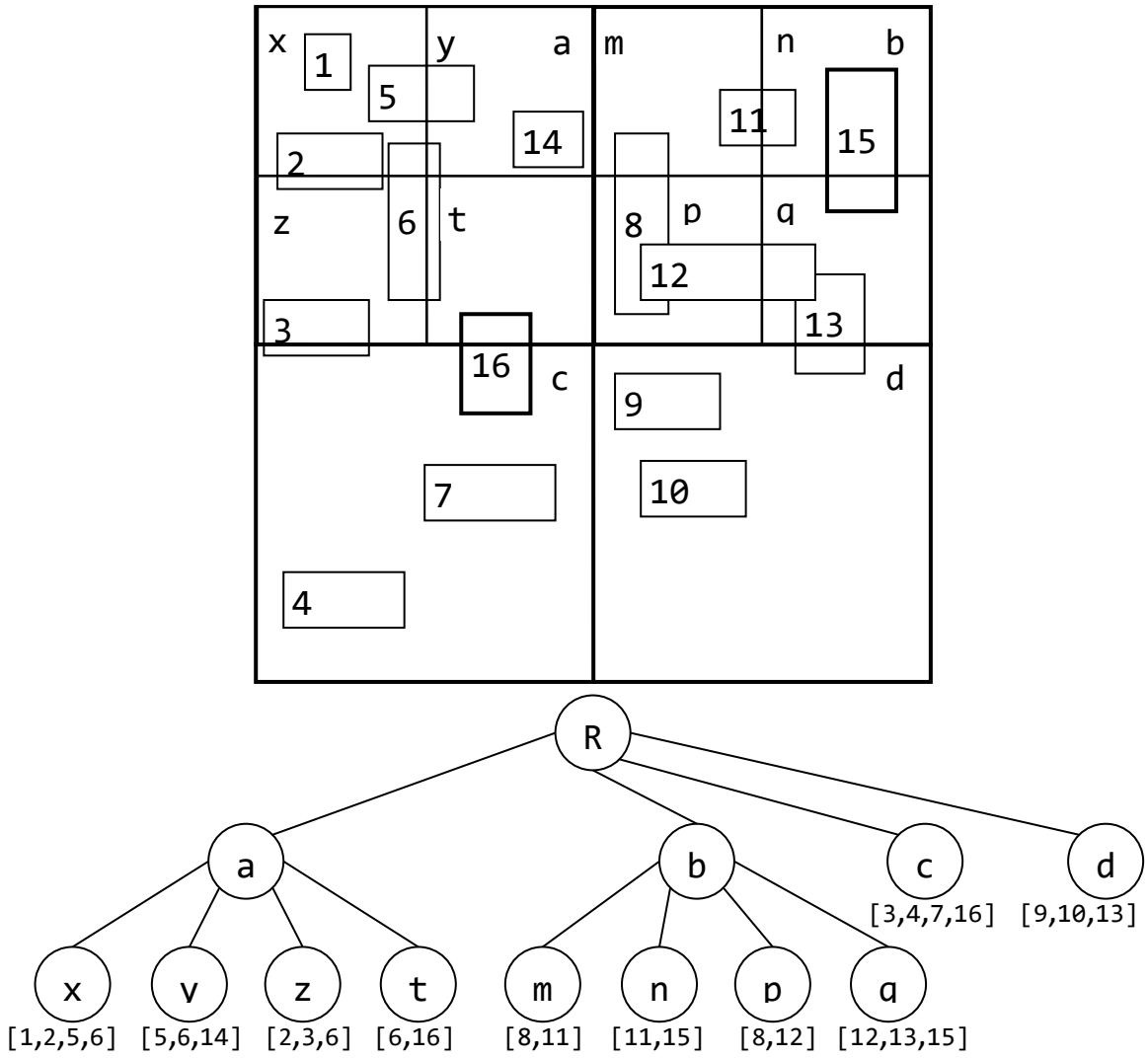


Рис. 12.6. Вставка об'єкта в квадро-дерево

Час запиту у квадро-деревах зв'язаний з глибиною дерева. В найгіршому випадку, вузол кожного піддерева знаходиться в окремій сторінці і кількість вводів-виводів дорівнює глибині дерева. Якщо колекція є статичною, до неї можна застосовувати більш ефективні методи розміщення вузлів дерева, але ситуація ускладнюється під час застосування динамічних колекцій. Крім того, подібно до структури фіксованої решітки, квадро-дерево потерпає від дублювання об'єктів в декількох листових сторінках. Коли розмір колекції є настільки великим, що розмір квадрантів дорівнює розміру індексованих прямокутників, дублювання об'єктів зростає експоненціально, що значно зменшує ефективність застосування подібної структури.

12.2.1. Криві розподілення

Крива розподілення визначає порядок комірок (квадрантів) двовимірної решітки. Перший тип кривої відомий як крива z-порядку (або код Мортона). Подібний порядок генерується наступним чином, корінь дерева не має позначки, кожен вузол дерева має позначку (0,1,2,3), отриману таким чином, що північно-західний нащадок вузла k має позначку k.0 (відповідно північно-східний — k.1, південно-західний — k.2 , південно-східний — k.3) (рис. 12.7). Існує можливість сортування комірок за лексикографічним порядком.

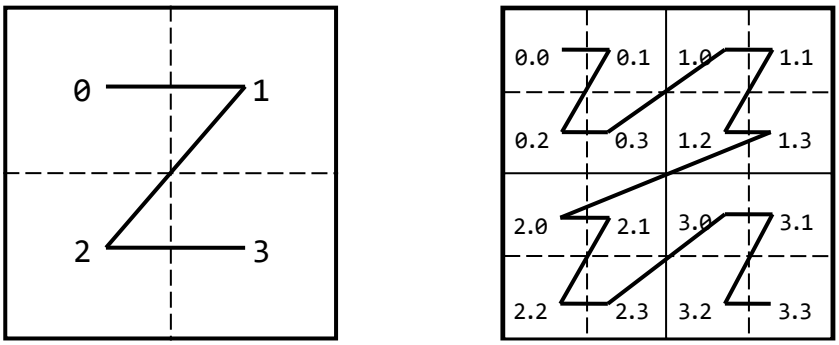


Рис. 12.7. Крива z-порядку

Крива Гільберта (Hilbert) — має форму П. На відміну від кривої z-порядку, крива Гілберта складається з частин однорідної довжини, це виключає переходи під час сканування від однієї комірки до іншої, якщо вони знаходяться одна від одної на значній відстані (рис 12.8).

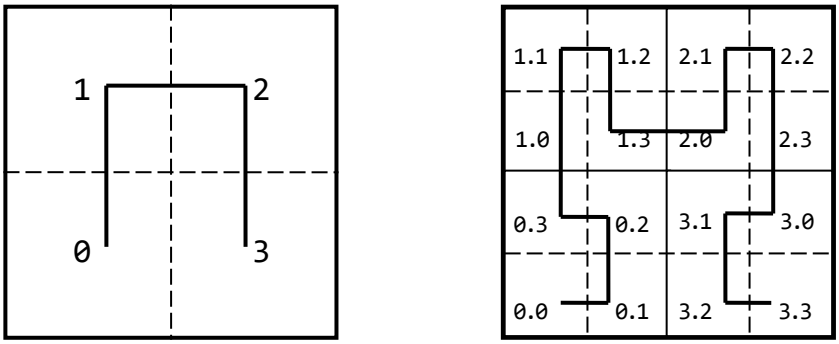


Рис. 12.8. Крива Гільберта

На рисунках 12.4-12.6 видно, що існують деякі ситуації, коли два об'єкти знаходяться близько один від одного на площині, але мають значну різницю в індексах, зумовлену порядком розподілення.

12.2.2. Лінійне квадро-дерево

Якщо об'єкт [mbb, oid] зв'язаний з вершиною квадро-дерева l та зберігається на сторінці з адресою p, то існує можливість індексування колекції пар (l, p) та створення B+ дерева. Де mbb (*minimal bounding box*) це мінімальний прямокутник, що визначає межі об'єкта, oid (*object id*) — ідентифікатор об'єкта. Таким чином ми отримаємо лінійне квадро-дерево.

Подібна структура забезпечує добре пакування позначок (індексів) квадро-дерева за допомогою B+ дерева. Пакування є динамічним, тобто воно зберігається під час видалення або вставки об'єктів у колекцію. Але подібна схема має проблему надмірності.

12.3. R-ДЕРЕВА

R-дерево — це гілчаста збалансована деревовидна структура з різною організацією внутрішніх та листових сторінок.

Інформація, що зберігається у R-деревах дещо відрізняється від тієї, що зберігається у бінарних деревах. В доповнення до ідентифікаторів просторових об'єктів, що знаходяться у листових сторінках, у R-деревах зберігається інформація про межі індексованого об'єкта. У випадку двовимірного простору зберігаються горизонтальні та вертикальні координати нижнього лівого та верхнього правого кутів найменшого прямокутника, який огинає об'єкт.

Структура R-дерева повинна відповідати наступним вимогам:

- для всіх внутрішніх вузлів дерева (окрім кореня), кількість нащадків знаходиться між m та M, де $m \in [0, M/2]$, m — мінімальна степінь вузла, M — максимальна степінь вузла;
- для кожного внутрішнього вузла визначена пара (dr, nodeid): dr — директивний прямокутник, nodeid — ідентифікатор вузла;
- для кожної листової сторінки визначена пара (mbb, oid): mbb — мінімальний прямокутник, який визначає межі просторового об'єкта, oid — ідентифікатор об'єкта;
- корінь має не менш ніж два виходи;
- усі листові сторінки знаходяться на одному рівні.

На рисунку 12.9 зображене R-дерево з $m=2$ та $M=4$. Індексована колекція зберігає 14 об'єктів. Директивні прямокутники вузлів a, b, c, d зображені пунктирною лінією.

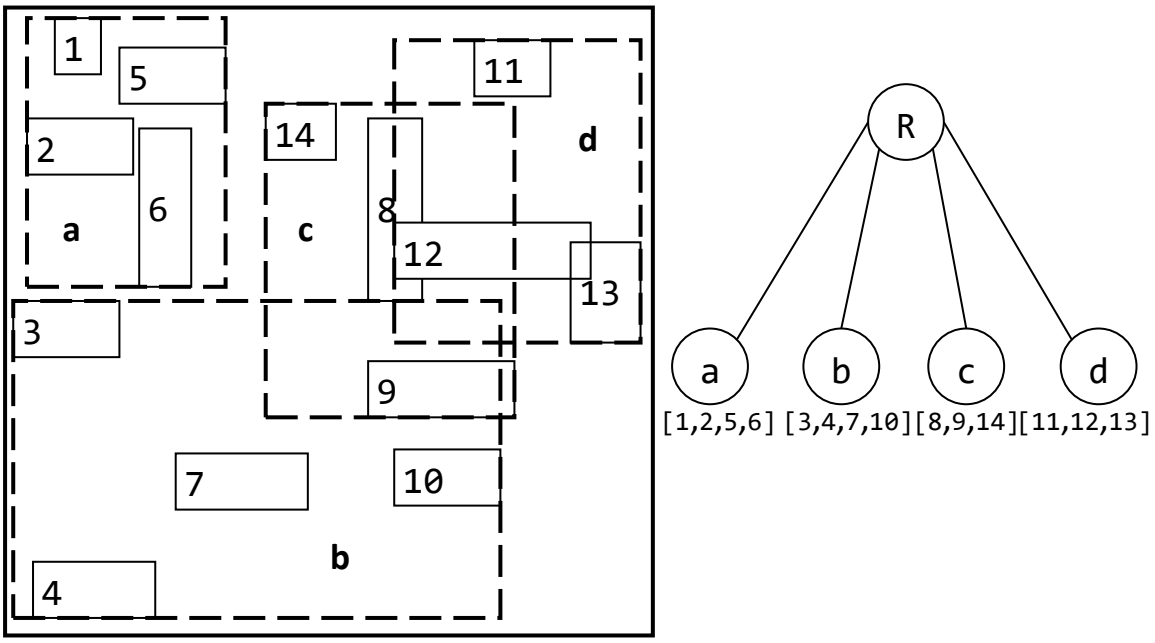


Рис. 12.9. R-дерево

Вказані вимоги до R-дерев повинні виконуватись після будь-якої перебудови дерева, викликаної динамічною вставкою чи видаленням об'єкта. Відмітимо те, що структура збалансованого дерева пристосовується до асиметрії розподілу даних. Регіон області пошуку заповнений великою кількістю об'єктів генерує велику кількість сусідніх листових сторінок.

M — максимальна степінь вузла, залежить від розміру нащадків вузла size(E) та можливості диску вмістити сторінку size(P): $M=\lfloor size(E)/size(P) \rfloor$. M може відрізнятися для листової та внутрішньої сторінки, в залежності від розміру ідентифікаторів nodeid (вузла) та oid (вершини). Визначення мінімальної кількості нащадків вузла — m залежить від стратегії розбиття вузлів.

R-дерево глибини d вміщує не менше ніж m об'єктів та не більше ніж M^{d+1}. І навпаки, глибина R-дерева, яке індексує колекцію N об'єктів, знаходиться між $\lfloor \log_{m}(N) \rfloor - 1$ та $\lfloor \log_{M}(N) \rfloor - 1$.

12.3.1. Пошук в R-деревах

Нижче розглянуто детальний алгоритм пошуку для точкового запиту. Функція точкового запиту виконується у два етапи. Спочатку, відбувається пошук всіх вузлів директивний прямокутник яких містить точку P. Розглядаються усі піддерева, оскільки точка може належати до перетину декількох прямокутників (рис. 12.10).

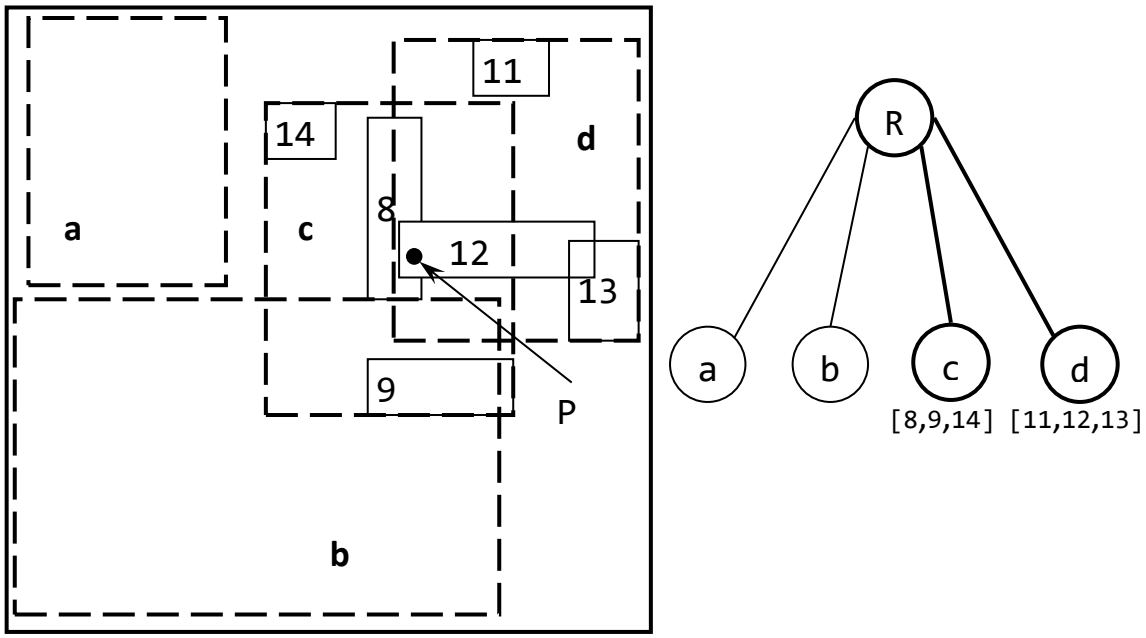


Рис. 12.10. Ілюстрація обробки точкового запиту в R-дереві

Процес повторюється на кожному рівні дерева доти, доки не будуть знайдені листові сторінки. Для кожного вузла N можливі дві ситуації:

- у вузлі ні один з прямокутників не містить у собі точку P — пошук завершується. Така ситуація можлива і за умови знаходження точки P в директивному прямокутнику вузла, точка P може знаходитися у так званому *мертвому просторі* вузла;
- точка P знаходиться в директивному прямокутнику одного чи декількох вузлів. Необхідно переглянути кожне піддерево.

Рисунок 12.10 ілюструє точковий запит, точка P належить об'єктам 8 та 12. Оброблюються три вузла: R, c, і d.

Для обходу R-дерева використовують рекурсивну функцію, яка на вхід отримує спочатку кореневий вузол, а тоді кожен наступний вузол нижчих рівнів.

Якщо точка належить тільки одному прямокутнику на кожному рівні, запит потребує d кроків для досягнення листової сторінки, де d — глибина дерева. Хоча подібна ситуація трапляється рідко, у більшості випадків необхідно розглянути невелику кількість шляхів від кореня до листових сторінок, а тому очікувана кількість кроків підпадає під логарифмічне розподілення. Нажаль, можливе виникнення ситуації, коли кількість кроків не буде підпадати під закон логарифмічного розподілення, так у найбільш несприятливому випадку всі директивні прямокутники можуть мати область перекриття до якої належить точка P. У подібній ситуації необхідно буде переглянути все дерево.

Алгоритм віконного запиту відрізняється лише тим, що предикат «містять точку P» змінений на предикат «перетинають вікно W», де W — параметри вікна запиту. Чим більше вікно, тим більша кількість вузлів, які необхідно розглянути.

12.3.2. Вставка об'єкта в R-дерево

Для того, щоб виконати операцію вставки об'єкта в існуюче дерево, необхідно обійти дерево зверху донизу, перевіряючи на кожному рівні, чи містить директивний прямокутник mbb об'єкта і, якщо містить, розглядати піддерева вузла до досягнення листової сторінки. Із декількох піддерев обирається те, чий директивний прямокутник потребує найменшого розширення.

Якщо листова сторінка не заповнена, то додаємо новий об'єкт [mbb, oid], а також, якщо це необхідно всі директивні прямокутники батьківських вузлів.

Якщо листова сторінка l, в яку необхідно вставити об'єкт, заповнена, виконується операція розбиття. Створюється нова листова сторінка l' та M+1 об'єктів розподіляється між l та l'. По закінченню розбиття необхідно модифікувати батьківський вузол та, якщо він заповнений, виконати операція розбиття.

Одна з важливих частин алгоритму — розбиття вузлів. Існує багато методів розбиття вузла. Будь-яке рішення, за якого m+1 знаходиться в одному вузлі (сторінці) та M+1-m-1 в іншому, може вважатися задовільним. Стратегія розбиття повинна відповідати наступним вимогам:

- мінімізація загальної області двох вузлів;
- мінімізація області перекриття двох вузлів.

Нажаль, ці вимоги не завжди сумісні, подібний випадок проілюстровано на рисунку 12.11.

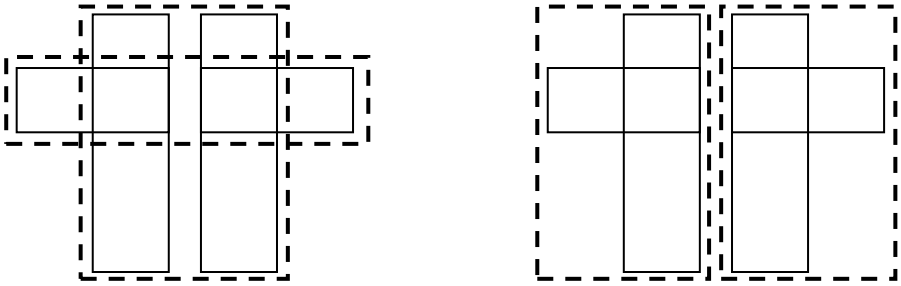


Рис. 12. 11. Розбиття з мінімальними областю та перекриттям

Метод вирішення «в лоб», під час використання стратегії, що враховує перший критерій, полягає у тому, щоб розглянути всі можливі прямокутники і вибрати серед них той, що мінімізує загальну область двох вузлів. Однак цей метод, не дивлячись на його простоту, виявляється дуже дорогим, виходячи з часу його виконання. Альтернативою може слугувати квадратичний алгоритм розбиття, з часом виконання квадратичним до M.

Квадратичний метод використовує наступну евристику. Спочатку дві групи об'єктів ініціалізуються, в них додаються об'єкти e та e' відстань і мертвий простір між якими максимальні. Мертвий простір визначається як сума областей mbb e та e' мінус сума областей e та e'. З об'єктів, що залишилися, кількістю M-2, до кожної групи додається той об'єкт, чиї розмір та положення мінімально

збільшать директивний прямокутник групи. У випадку рівності об'єкт додається до тієї групи, чия кількість об'єктів менша.

Дві частини алгоритму (ініціалізація групи та вставка елементів) квадратичні по відношенню до M. Якщо під час виконання останньої частини алгоритму одній з груп (наприклад G_1), була надана перевага над іншою групою (наприклад G_2), набір елементів, що залишилися E', повинен бути доданий в групу G_2, незалежно від їх місцезнаходження. Це може, у деяких випадках, призвести до поганого розподілення елементів. Очевидно, що це залежить від параметра m, який визначає мінімальну кількість елементів групи. Після проведених досліджень значення m=40%M, здається найбільш придатним для використання цього алгоритму.

Також існує лінійний алгоритм, який складається з вибору таких елементів, відстань між якими за значенням однієї з осей декартового простору є найбільшим, та розподілення елементів, що залишилися, у групу, mbb якої потребує найменшого розширення області під час вставки цього елемента. Лінійний алгоритм є більш простим та швидшим. Але, як було доведено в декількох експериментах, результат роботи цього алгоритму погіршує надмірний перетин граничних прямокутників груп.

12.3.3. R-ДЕРЕВА*

R*-дерева — це R-дерева, які надають дещо покращені класичні алгоритми. В R*-деревах оптимізація проводиться за наступними параметрами: область перекриття вузла, область вузла, область директивного прямокутника.

Нажаль не існує методів, які б одночасно оптимізували ці три параметри. Нижче подані два варіанти методів, які покращують алгоритми класичного R-дерева. Перший алгоритм — покращений алгоритм розбиття дерева.

Алгоритм розбиття R-дерева спочатку ініціалізує дві групи об'єктів, які знаходяться на максимальній відстані один від одного, а потім додають об'єкти до тієї чи іншої групи, виходячи з критерію мінімізації області покриття груп об'єктів. Підхід, який застосовується у R*-деревах, відрізняється тим, що розбиття відбувається тільки відносно якоїсь з осей (вертикальної чи горизонтальної). Переваги цього методу зображені на рисунку 12.12.

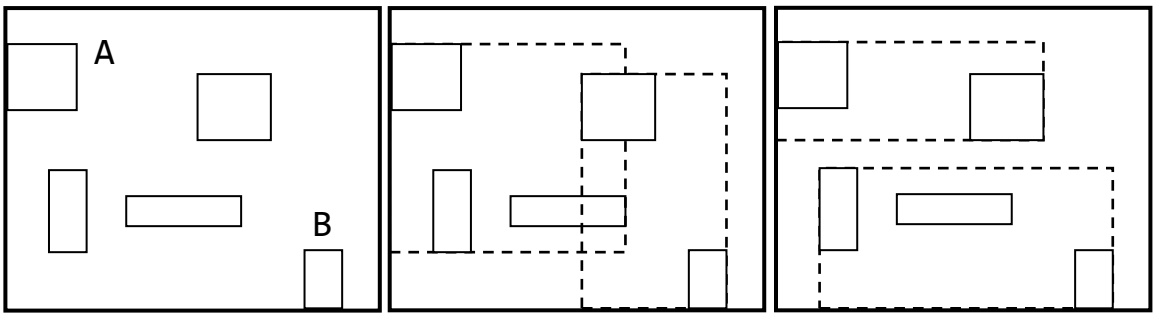


Рис. 12.12. Стратегії розбиття R-дерева та R*-дерева

Алгоритм розбиття R-дерева обирає спочатку два перших об'єкта A та B, та ініціалізує відповідно групи G_1 та G_2. Оскільки об'єкт A значно більший ніж об'єкт B, то алгоритм розбиття R-дерева на перших ітераціях буде додавати об'єкти до групи G_1, що у подальшому призведе до неоптимального розподілення об'єктів у дереві.

Алгоритм розбиття R*-дерева проводить розбиття відносно вісі X, що дозволяє уникнути перетину директивних прямокутників двох груп. Для знаходження вісі розбиття прямокутники сортують за мінімальним або максимальним параметром. Кількість операцій дорівнює 2(2(M-2m+2)). Враховуючи вісь, обирається розподілення об'єктів з мінімальним перекриттям. У випадку двох варіантів розподілення з однаковим перекриттям обирається варіант з мінімальною областю покриття.

Інший важливий алгоритм R*-дерев — алгоритм вставки, який базується на принципі примусової повторної вставки об'єкта (рис. 12.13).

На рисунку 12.13, що ілюструє стратегію реорганізації R*-дерев під час заповнення елементів, об'єкт 8 вставлений в існуюче дерево, що викликало переповнення вузла v. Алгоритм вставки R-дерева виконає місцеву реорганізацію, що призведе до небажаного перекриття сторінок дерева.

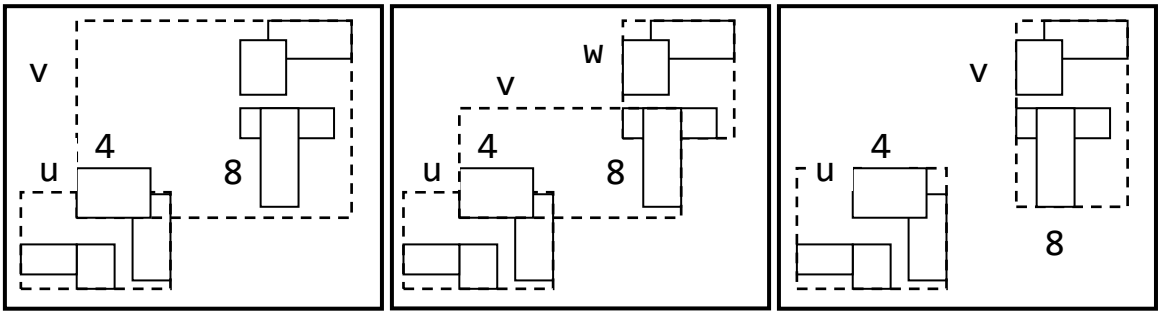


Рис. 12.13. Стратегія повторної вставки об'єкта: вставка об'єкта 8 (переповнення вузла v); розбиття R-дерева та реорганізація у R*-дереві

Алгоритм повторної вставки R*-дерева виконає наступну послідовність дій:

- видалить об'єкт 4 з сторінки v;
- вирахує нове граничне поле v;
- повторно вставить об'єкт 4, модифікувавши дерево, починаючи з кореня у вузол u;
- розмістить об'єкт 8 на сторінці v, не виконуючи подальшого розбиття.

Оскільки переповнення вузла може трапитись на будь-якому рівні дерева, видалений об'єкт повинен бути вставлений на тому ж самому рівні, де він був видалений. У випадку, коли після виконання повторної вставки вузол виявляється переповненим, для того, щоб уникнути нескінченного циклу, застосовується алгоритм розбиття дерева.

12.3.4. R+-ДЕРЕВА

R+-дерева — це своєрідний компроміс між R-деревами та kD-деревами. Вони уникають перекриття директивних прямокутників груп, шляхом вставки об'єкта в кілька листових сторінок, якщо це необхідно.

Різниця між R-деревами та R+-деревами наступна:

- листові сторінки не гарантовано заповнені хоча б на половину;
- внутрішні вузли не перекриваються;
- ідентифікатор об'єкта може зберігатися в більше, ніж одній листовій сторінці.

Переваги R+-дерев:

- оскільки листові сторінки не перекриваються між собою, виконання точкового запиту є швидким, оскільки всі об'єкти охоплюються щонайбільше одним прямокутником;
- під час пошуку переглядається менша кількість вузлів, ніж в аналогічному R-дереві.

Недоліки R+-дерев:

- оскільки прямокутники дублюються, R+-дерево може бути більш громіздкім, ніж R-дерево побудоване на тому ж наборі об'єктів;
- побудова і підтримка R+-дерев більш складна, ніж побудова і підтримка R-дерев та R*-дерев.

12.4. Z-ВПОРЯДКОВАНІ ДЕРЕВА

На відміну від інших типів дерев, цей тип працює не з прямокутниками, а з реальною геометрією об'єктів.

Алгоритм побудови включає наступний основний етап. Враховуючи геометрію об'єкта o та квадрант q, перевіряємо в які листові сторінки квадро-дерева побудованого в квадранті q попадає об'єкт. Проводимо декомпозицію об'єкта, тобто аналізуємо в які підквадранти потрапляє об'єкт o, та будуємо B+ дерево. Декомпозиція припиняється по досягненні максимальної глибини d.

Рисунок 12.14 ілюструє декомпозицію та апроксимацію об'єкта. Апроксимація об'єкта — список квадрантів {023, 03, 103, 12, 201, 210, 211, 300, 301, 302}.

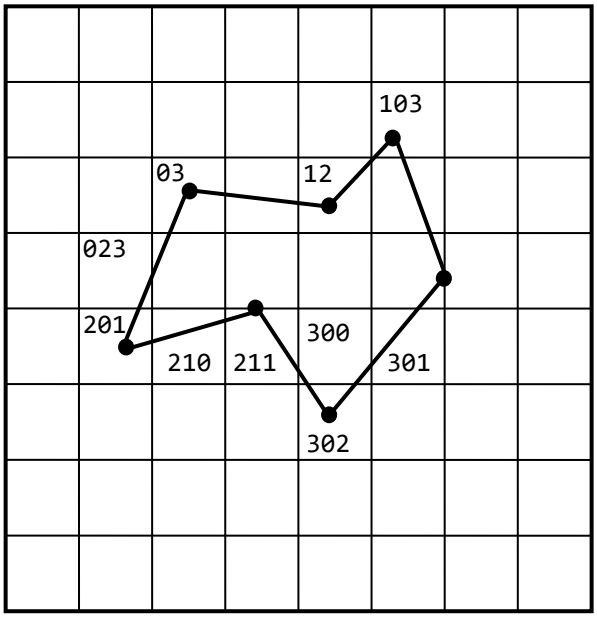
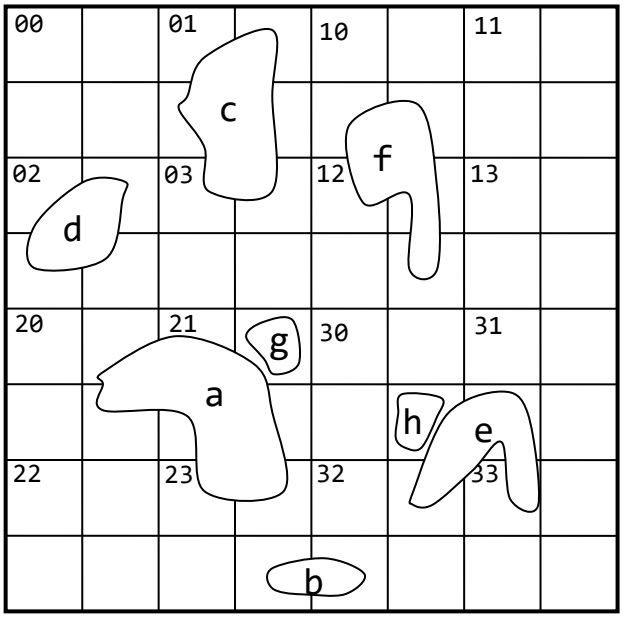


Рис. 12.14. Z-впорядкування та декомпозиція об'єкта

Колекція з восьми об'єктів разом з їх Z-впорядкованою декомпозицією подана на рисунку 12.15. Квадро-дерево має максимальну глибину d=3. Отримуємо набір об'єктів `[l,oid]`, где `l` — позначка комірки, і `oid` — ідентифікатор об'єкта, апроксимована форма якого містить чи перетинає комірку `l`.

Очевидно, що ця схема, допускає дублювання. Ідентифікатор об'єкта може знаходитись в багатьох комірках апроксимації його контуру. Так і навпаки, можлива ситуація, коли існує декілька пар, що мають одну і ту ж `l`, але різні ідентифікатори об'єктів. Подібна ситуація виникла для об'єктів e та h, які разом використовують комірку 303. Також можлива ситуація, коли декілька об'єктів припадають на одну і ту ж комірку, але на різних рівнях декомпозиції (об'єкти a та g).




```
a={201 203 21 230 231}
b={233 322}
c={01 030 031}
d={02}
e={303 312 321 330}
f={102 103 120 121 123}
g={211}
h={303}
```

Рис. 12.15. Набір об'єктів із Z-впорядкованою декомпозицією

На рисунку 12.16 подане B+ дерево, яке отримано від Z-впорядкованого набору даних (рис. 12.15). Ідентифікатори об'єктів доступні у вузлах B+ дерева.

Необхідно відмітити, що один і той же об'єкт може бути представлений у двох віддалених листових сторінках B+ дерева, завдяки деякому неминучому переходу під час Z-впорядкування. Наприклад, об'єкт b розподілений між двома комірками: ідентифікатор об'єкта зберігається у різних та не сусідніх листових сторінках B+ дерева.

Алгоритми обробки точкових та віконних запитів подібні до алгоритмів обробки запитів квадро-дерева. Єдина відмінність — збереження ідентифікаторів об'єктів (`oid`), які перетинаються з вікном пошуку.

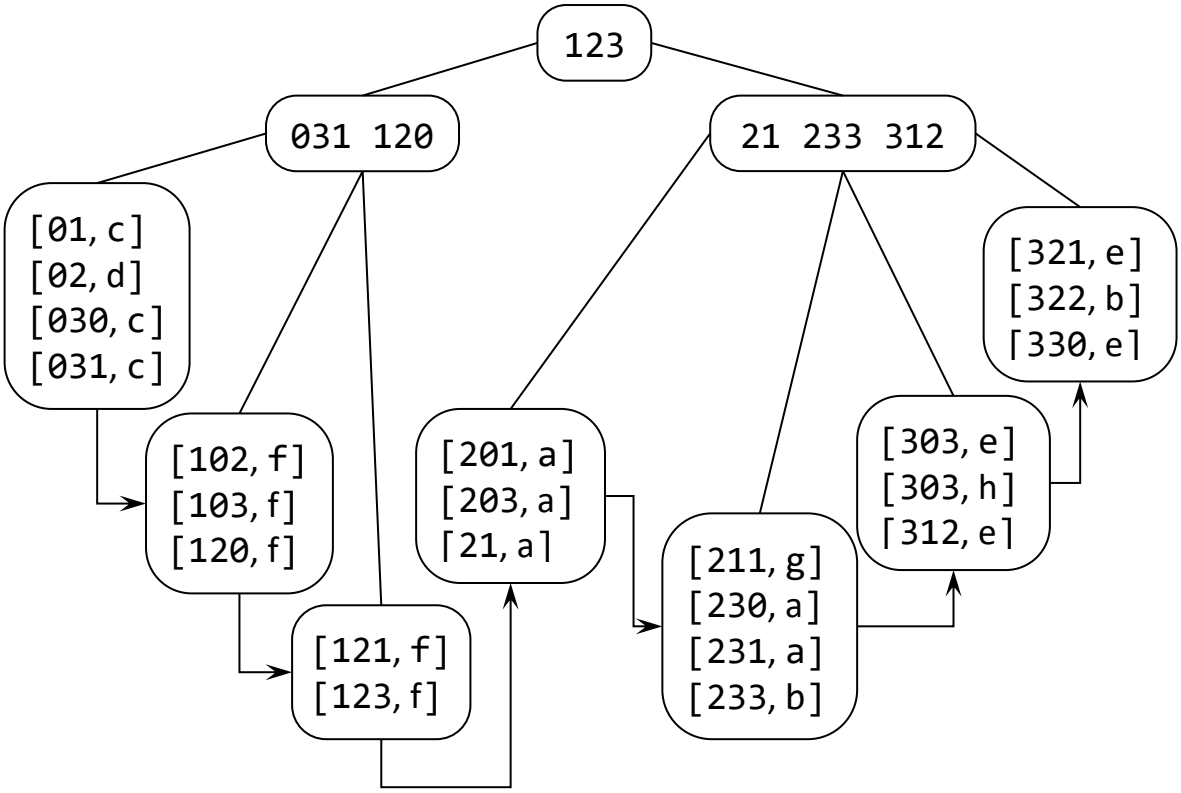


Рис. 12.16. B+ дерево побудоване на Z-впорядкованому наборі даних

Запитання для самоконтролю

1. Для чого використовуються структури просторової індексації?
2. На якому принципі засноване kD-дерево?
3. Який спосіб розбиття простору використовується в квадро-деревах?
4. Що таке криві розподілення? Які бувають їх види?
5. Що таке лінійне квадро-дерево?
6. Який принцип побудови R-дерев?
7. Що таке R* та R+-дерева? Які їх особливості та відмінності від R-дерев?
8. Які особливості Z-впорядкованих дерев?

13. ВИДАЛЕННЯ НЕВИДИМИХ ЛІНІЙ ТА ГРАНЕЙ

Однією з найважливіших задач тривимірної графіки є визначення, які частини об'єктів (ребра, грані), що знаходяться у тривимірному просторі, будуть видимі під час обраного способу проєктування, а які будуть закриті від спостерігача іншими об'єктами. В якості можливих видів проєктування традиційно розглядаються паралельне і центральне.

Саме проєктування відбувається на так звану картинну площину (екран): крізь кожну точку кожного об'єкта проводиться проєкційний промінь (проєктор) до картинної площини. Усі проєктори створюють пучок або паралельних променів (під час паралельного проєктування), або променів, що виходять з однієї точки (центральне проєктування). Перетин проєктора з картинною площиною дає проєкцію точки. Видимими будуть тільки ті точки, що розміщені вздовж напрямку проєктування найближче до картинної площини. Усі три точки P_1, P_2 та P_3 (рис. 13.1) знаходяться на одному і тому ж проєкторі, тобто проєктуються в одну і ту ж точку картинної площини.

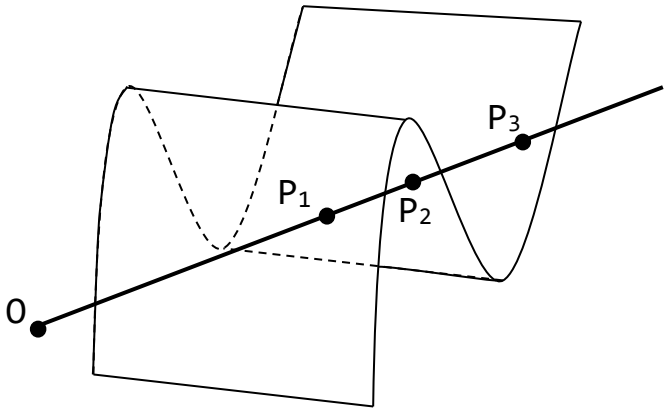


Рис. 13.1. Точки об'єкта на одному проєкторі

Але оскільки точка P_1 лежить ближче до картинної площини, ніж точки P_2 і P_3 та закриває їх під час проєктування, то з цих трьох точок саме вона є видимою.

Не дивлячись на простоту, як здається, завдання видалення невидимих ліній і поверхонь є досить складним і часто вимагає дуже великих об'ємів обчислень. Тому існує цілий ряд різних методів вирішення цієї задачі, включно з методами, які спираються на апаратні рішення.

Ці методи розрізняються за наступними основними параметрами:

- способу подання об'єктів;
- способу візуалізації сцени;
- простору, в якому проводиться аналіз видимості;
- вигляду отриманого результату (його точність).

Як можливі способи подання об'єктів можуть виступати аналітичні (явні і неявні), параметричні та полігональні.

Далі вважатимемо, що всі об'єкти подані набором опуклих плоских граней, наприклад трикутників (полігональний спосіб), які можуть перетинатися один з одним лише вздовж ребер.

Координати в заданому тривимірному просторі позначатимемо через (x, y, z), а координати в картинній площині — через (X, У). Також вважатимемо, що на картинній площині задана цілочисельна растрова решітка — множина точок (i, j), де i та j — цілі числа.

Якщо це особливо не обумовлено, вважатимемо для простоти, що проєктування здійснюється на площину XOY. Проєктування відбувається або паралельно осі OZ, тобто задається формулами: X=x та y=Y, або є центральним з центром, розташованим на осі OZ, і задається формулами:

$$X = \frac{x}{z}; Y = \frac{y}{z}.$$

Існують два різні способи зображення тривимірних тіл — *каркасне* (wireframe — малюються лише ребра) і *суцільне* (малюються зафарбовані грані). Тим самим виникають два типи задач — *видалення невидимих ліній* (ребер для каркасних зображень) і *видалення невидимих поверхонь* (граней для суцільних зображень).

Аналіз видимості об'єктів можна проводити як в початковому тривимірному просторі, так і на картинній площині. Це приводить до розділення методів на два класи:

- методи, що працюють безпосередньо в просторі самих об'єктів;
- методи, що працюють в просторі картинної площини, тобто працюють з проєкціями об'єктів.

Отримуваний результат є або набором видимих областей або відрізків, заданих з машинною точністю (має неперервний вигляд), або інформацію про найближчий об'єкт для кожного пікселя екрану (має дискретний вигляд).

Методи першого класу дають точне вирішення задачі видалення невидимих ліній і поверхонь, ніяк не прив'язане до растрових властивостей картинної площини.

Вони можуть працювати як з самими об'єктами, виділяючи ті їх частини, які видимі, так і з їх проєкціями на картинну площину, виділяючи на ній області, відповідні проєкціям видимих частин об'єктів, і, як правило, практично не прив'язані до растрових решіток та вільні від похибок дискретизації. Оскільки ці методи працюють з неперервними початковими даними і результати, що виходять, не залежать від растрових властивостей, то їх інколи називають *неперервними* (*continuous methods*) .

Простий варіант неперервного підходу полягає в порівнянні кожного об'єкта зі всіма іншими, що дає часові витрати, пропорційні N^2, де N — кількість об'єктів у сцені.

Проте слід мати на увазі, що неперервні методи, як правило, досить складні.

Методи другого класу — дискретні (*point-sampling methods*) дають наближене вирішення задачі видимості, визначаючи видимість лише в деякому наборі точок картинної площини — в точках растрових решіток. Вони дуже сильно прив'язані до растрових властивостей картинної площини і фактично полягають у визначенні для кожного пікселя тієї грані, яка є найближчою до нього уздовж напряму проєктування. Зміна роздільної здатності призводить до необхідності повного перерахунку всього зображення.

Простий варіант дискретного методу має часові витрати порядку CN, де C — загальна кількість пікселів екрана, а N — кількість об'єктів.

Всім методам другого класу традиційно властиві помилки дискретизації (*aliasing artifacts*). Проте, як правило, дискретні методи відрізняються простотою.

Окрім цього існує досить велика кількість змішаних методів, що використовують роботу як в об'єктному просторі, так і в картинній площині. Ці методи виконують частину роботи з неперервними даними, а частину — з дискретними.

Більшість алгоритмів видалення невидимих граней і поверхонь тісно пов'язана з різними методами сортування. Деякі алгоритми проводять сортування явно, в деяких воно присутнє в прихованому вигляді. Наближені методи відрізняються один від одного фактично лише порядком і способом проведення сортування.

Дуже поширеною структурою даних в задачах видалення невидимих ліній і поверхонь є різні типи дерев — BSP (*Binary Space Partition*), квадро (*Quad trees*), окто (*Oct trees*) та ін.

Методи, що практично застосовуються в теперішній час, в більшості є комбінаціями ряду простих алгоритмів, несучи в собі цілий ряд різного роду оптимізацій.

Вкрай важлива роль в підвищенні ефективності методів видалення невидимих ліній і граней відводиться використанню *когерентності* (*coherence* — зв'язність). Вирізняють декілька типів когерентності:

- когерентність в картинній площині — якщо піксель відповідає точці грані P, то швидше за все сусідні пікселі також відповідають точкам тієї ж грані;
- когерентність в просторі об'єктів — якщо об'єкт (грань) видимий (невидимий), то розташований поруч об'єкт (грань) швидше за все також є видимим (невидимим);
- в разі побудови анімації виникає третій тип когерентності — часова: грані, видимі в поточному кадрі, швидше за все будуть видимі і в наступному; аналогічно грані, невидимі в поточному кадрі, швидше за все будуть невидимі і в наступному.

Акуратне використання когерентності дозволяє помітно скоротити кількість перевірок і помітно підвищити швидкодію алгоритму.

13.1. Відсікання нелицьових граней

Розглянемо багатогранник, для кожної грані якого заданий одиничний вектор зовнішньої нормалі (рис. 13.2). Неважко помітити, що коли вектор нормалі грані n складає з вектором l, який задає напрям проєктування, тупий кут (вектор нормалі направлений від спостерігача), то ця грань завідомо не може бути видимою. Такі грані називаються *нелицьовими*. Якщо відповідний кут є гострим, грань називається *лицьовою*.

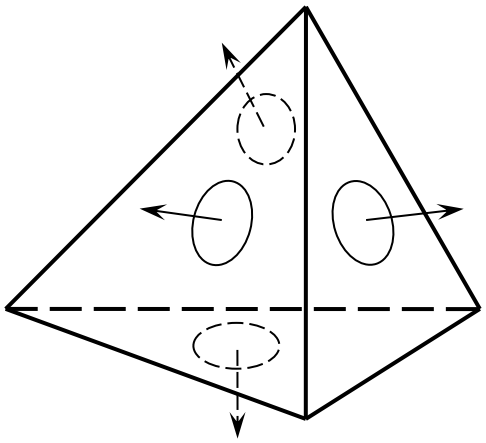


Рис. 13.2. Лицьові та нелицьові грані

Під час паралельного проєктування умову на кут можна записати у вигляді нерівності (n, l)<0, оскільки напрям проєктування від грані не залежить.

Під час центрального проєктування з центром в точці c вектор проєктування для точки p буде рівний l=c–p. Для визначення того, є задана грань лицьовою чи ні, досить узяти довільну точку цієї грані і перевірити виконання умови (n, l)<0. Знак цього скалярного добутку не залежить від вибору точки на грані, а визначається тим, в якому півпросторі відносно площини, що містить цю грань, лежить центр проєктування.

Оскільки під час центрального проєктування промінь проєктування залежить від грані (і не залежить від вибору точки на грані), то лицьова грань може стати нелицьовою, а нелицьова лицьовою навіть під час паралельного зсуву. Під час паралельного проєктування зсув не змінює кутів і те, чи є грань лицьовою чи ні, залежить лише від кута між нормаллю до грані і напрямом проєктування.

Відмітимо, що якщо по аналогії з визначенням належності точки багатокутнику, пропустити через довільну точку картинної площини промінь проєктування до об'єктів сцени, то число перетинів променя з лицьовими гранями буде дорівнювати числу перетинів променя з нелицьовими гранями.

У разі, коли сцена є одним опуклим багатогранником, видалення нелицьових граней повністю вирішує задачу видалення невидимих граней.

Хоча в загальному випадку запропонований підхід і не вирішує задачі видалення повністю, проте дозволяє приблизно вдвічі скоротити кількість граней внаслідок того, що нелицьові грані завжди невидимі; що ж до лицьових граней, то в

загальній ситуації частини деяких лицьових граней можуть бути закриті іншими лицьовими гранями (рис. 13.3).

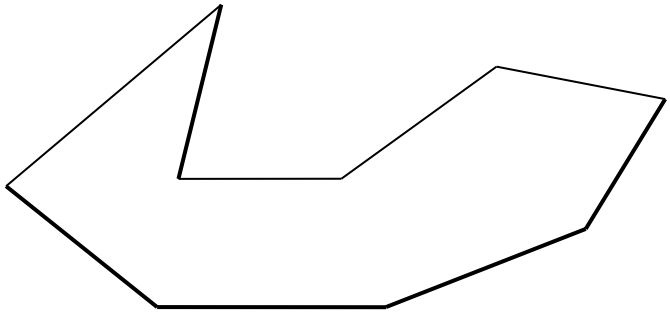


Рис. 13.3. Перекриття лицьової грані іншими лицьовими гранями

Ребра між нелицьовими гранями також завжди не видно. Проте ребро між лицьовою і нелицьовою гранями цілком може бути видимим.

13.2. Алгоритм Робертса

Першим алгоритмом видалення невидимих ліній був алгоритм Робертса, який потребує, щоб кожна грань була опуклим багатокутником. Спочатку відкидаються всі ребра, в яких обидва визначальні грані є нелицьовими (жодне з таких ребер завідомо не видиме). Наступним кроком є перевірка на закривання кожного з ребер, що залишилися, зі всіма лицьовими гранями багатогранника. Можливі наступні випадки (рис. 13.4):

- грань ребра не закриває;
- грань повністю закриває ребро (тоді воно видаляється із списку даних ребер);
- грань частково закриває ребро (в цьому випадку ребро розбивається на декілька частин, видимими з яких є не більше двох; саме ребро видаляється із списку, але в список перевірених ребер додаються ті його частини, які цією гранню не закриваються).

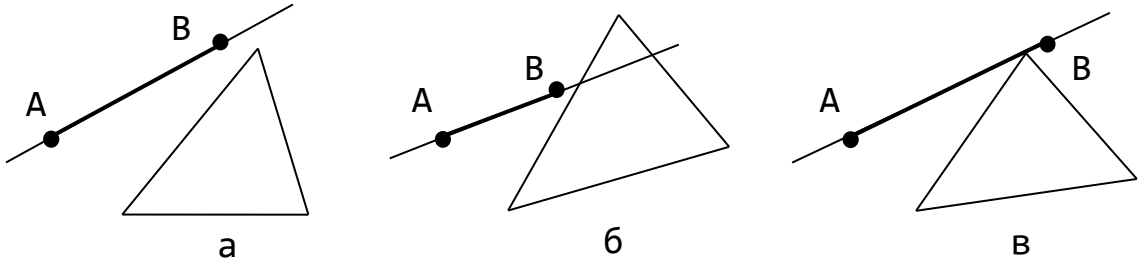


Рис. 13.4. Варіанти співвідношення грані і ребра

Розглянемо, як здійснюються ці перевірки.

Нехай задане ребро AB, де точка A має координати (x_a, y_a), а точка B — (x_b, y_b). Пряма, що проходить через відрізок AB, задається рівняннями:

$$x = x_a + t(x_b - x_a);\ y = y_a + t(y_b - y_a).$$

Причому сам відрізок відповідає значенням параметра `0<t<1`. Цю пряму можна задати неявним чином як `F(x, y)=0`, де:

$$F(x,y) = (y_b - y_a)(x - x_a) - (y_b - x_a)(y - y_a).$$

Передбачимо, що проєкція грані задається набором проєкцій вершин $P_1,...,P_k$ координатами (x_i,y_i) $i \in [1,n]$. Позначимо через F, значення функції F у точці P_i, розглянемо i-й відрізок проєкції грані P_iP_{i+1}. Цей відрізок перетинає пряму AB тоді і лише тоді, коли функція F набуває значення різних знаків на кінцях відрізка, а саме коли:

$$F_iF_{i+1} \le 0.$$

Випадок, коли $F_{i+1}=0$, відкидатимемо, аби двічі не зараховувати пряму, що проходить через вершину, для обох відрізків, що виходять з неї.

Отже, ми вважаємо, що перетин має місце в двох випадках:

$$F_i \ge 0;\ F_{i+1} < 0;\ \text{або}\ F_i \le 0;\ F_{i+1} > 0.$$

Точка перетину визначається співвідношеннями:

$$x = x_i + s(x_{i+1} - x_i);\ y = y_i + s(y_{i+1} - y_i), \text{де}\ s = \frac{F_i}{F_i - F_{i+1}}.$$

Звідси легко знаходиться значення параметра t:

$$t = \begin{cases} \dfrac{x - x_a}{x_b - x_a}, |x_b - x_a| \ge |y_b - y_a|, \\ \dfrac{y - y_a}{y_b - y_a}, |y_b - y_a| > |x_b - x_a|. \end{cases}$$

Можливі наступні випадки.

1. Відрізок не має перетину з проєкцією грані, окрім, можливо, однієї точки. Це може мати місце, коли:
   - пряма AB не перетинає ребра проєкції (рис. 13.4, а);
   - пряма AB перетинає ребро в двох точках t_1 і t_2, але або $t_1<0$, $t_2<0$ або $t_2>1$, $t_1>1$ (рис. 13.4, б);
   - пряма AB проходить через одну вершину, не зачіпаючи внутрішній простір трикутника (рис. 13.4, в).

Вочевидь, що в цьому випадку відповідна грань ніяк не може закривати собою ребро AB.

2. Проєкція ребра повністю міститься всередині проєкції грані (рис. 13.5, а). Тоді є дві точки перетину прямої AB і грані — $t_1<0<1<t_2$. Якщо грань лежить ближче до картинної площини, ніж ребро, то ребро повністю невидиме і видаляється.
3. Пряма AB перетинає ребра проєкції грані в двох точках і або $t_1<0<t_2<1$, або $0<t_1<1<t_2$ (рис. 13.5, б і в). Якщо ребро AB знаходиться далі від картинної площини, ніж відповідна грань, то воно розбивається на дві частини, одна з

яких повністю закривається гранню і тому відкидається. Проєкція другої частини лежить поза проєкцією грані і тому цією гранню не закривається.

4. Пряма AB перетинає ребра проєкції грані в двох точках, причому `0<t1<t2<1` (рис. 13.5, г). Якщо ребро AB лежить далі від картинної площини, ніж відповідна грань, то воно розбивається на три частини, середня з яких відкидається.

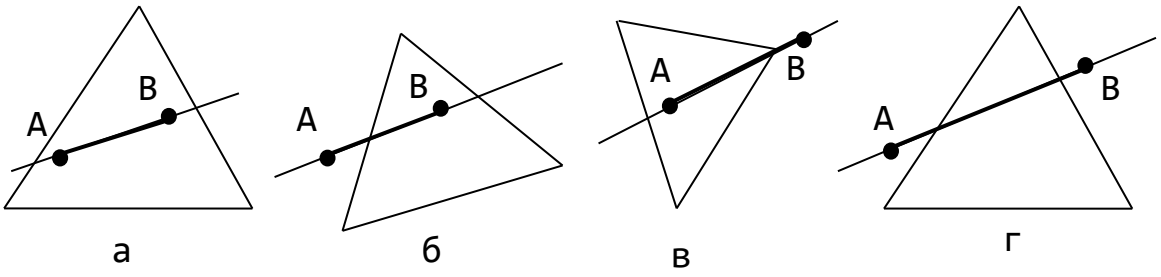


Рис. 13.5. Варіанти перетину ребра і грані

Для визначення того, що лежить ближче до картинної площини — відрізок AB (проєкція якого лежить в проєкції грані) або сама грань, через цю грань проводиться площина (n, p)+c=0 (n — нормальний вектор грані), що розбиває весь простір на два півпростори. Якщо обидва кінця відрізка AB лежать в тому ж півпросторі, в якому знаходяться спостерігачі, то відрізок лежить ближче до грані; якщо обидва кінця знаходяться в іншому півпросторі, то відрізок лежить далі. Випадок, коли кінці лежать в різних півпросторах, тут неможливий (це означало б, що відрізок AB перетинає внутрішню частину грані).

Якщо загальна кількість граней рівна N, то часові витрати для цього алгоритму складають $O(N^2)$. Кількість перевірок можна помітно скоротити, якщо скористатися розбиттям картинної площині.

Розіб'ємо видиму частину картинної площини (екран) на $N_1 x N_2$ рівних частин (клітинок) і для кожної клітинки A_{ij} побудуємо список всіх лицьових граней, чиї проєкції мають з цією клітинкою непустий перетин. Для перевірки довільного ребра на перетин з гранями відберемо спочатку всі ті клітинки, які проєкція цього ребра перетинає. Ясно, що перевіряти на перетин з ребром має сенс лише ті грані, які містяться в списках цих клітинок.

Як крок розбиття зазвичай вибирається O(l), де l — характерний розмір ребра в сцені. Для будь-якого ребра кількість граней, що перевіряються, практично не залежить від загального числа граней і сукупні часові витрати алгоритму на перевірку перетинів складають O(N), де N — кількість ребер в сцені.

Оскільки процес побудови списків полягає в переборі всіх граней, їх проєктуванні і визначенні клітинок, в які потрапляють проєкції, то витрати на складання всіх списків також складають O(N).

13.3. Метод трасування променів

Задача видалення невидимих граней є помітно складнішою, ніж задача видалення невидимих ліній, хоча б за загальним обсягом інформації. Якщо практично всі методи, що використовуються для видалення невидимих ліній, працюють в об'єктному просторі і дають точний результат, то для видалення невидимих поверхонь існує велика кількість методів, що працюють тільки в картинній площині, а також змішаних методів.

Найбільш природним методом для визначення видимості граней є метод трасування променів (варіант, що використовується тільки для визначення видимості, без відстежування відображених і заломлених променів зазвичай називається *ray casting*), в якому для кожного пікселя картинної площини визначається найближча до нього грань. Для цього через піксель пропускається промінь, знаходяться всі точки його перетину з гранями і серед них вибирається найближча. Цей алгоритм можна подати таким чином:

```
for all pixels
for all objects
compare z
```

Однією з переваг цього метода є простота, універсальність (він може легко працювати не тільки з полігональними моделями, можливе використання *Constructive Solid Geometry*) і можливість поєднання визначення видимості з розрахунком кольору пікселя.

Ще одним безперечним плюсом методу є велика кількість методів оптимізації, що дозволяють працювати з сотнями тисяч граней і які забезпечують часові витрати порядку O(C log N), де C — загальна кількість пікселів на екрані і N — загальна кількість об'єктів у сцені. Більш того, існують методи, що забезпечують практичну незалежність часових витрат від кількості об'єктів.

13.4. Метод Z-буфера

Одним з найпростіших алгоритмів видалення невидимих граней і поверхонь є, метод Z-буфера (буфера глибини), де для кожного пікселя, як і в методі трасування променів, знаходиться грань, найближча до нього уздовж напряму проєктування, проте тут цикли по пікселям і по об'єктам міняються місцями:

```
for all objects
for all covered pixels
compare z
```

Поставимо у відповідність кожному пікселю (x, y) картинної площини окрім кольору c(x, y), що зберігається у відеопам'яті, його відстань до картинної площини уздовж напряму проєктування z(x, y) (його глибину).

Для виводу на картинну площину довільної грані вона переводиться в растрове подання на картинній площині і потім для кожного пікселя цієї грані знаходиться

його глибина. У випадку, якщо ця глибина менша значення глибини, що зберігається в Z-буфері, піксель малюється і його глибина заноситься в Z-буфер.

Досить ефективним є поєднання растрової розгортки грані з виводом в Z-буфер. Водночас для обчислення глибини пікселів можуть застосовуватися інкрементальні методи, що вимагають всього декількох додавань на піксель.

Грань малюється послідовно рядок за рядком; для знаходження необхідних значень використовується лінійна інтерполяція (рис. 13.6).

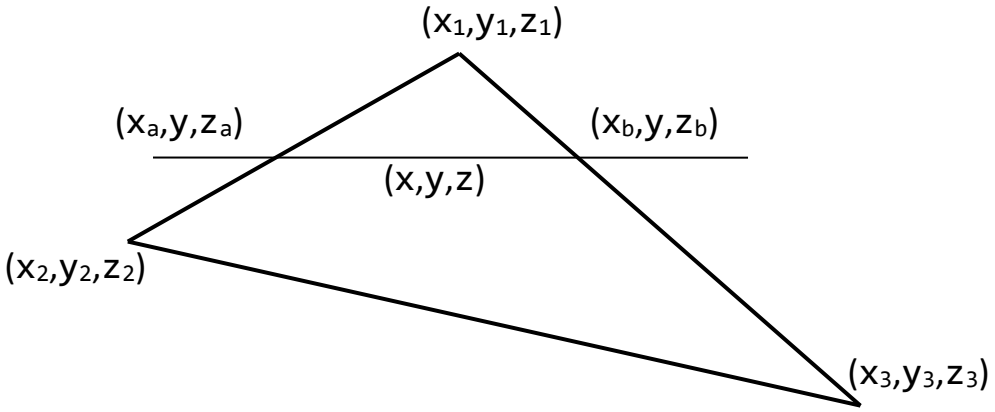


Рис. 13.6. Послідовний вивід грані методом Z-буфера

Глибина точки розраховується за наступними інтерполяційними формулами:

$$x_a = x_1 + (x_2 - x_1)\frac{y - y_1}{y_2 - y_1};\ x_b = x_1 + (x_3 - x_1)\frac{y - y_1}{y_3 - y_1};$$

$$z_a = z_1 + (z_2 - z_1)\frac{y - y_1}{y_2 - y_1};\ z_b = z_1 + (z_3 - z_1)\frac{y - y_1}{y_3 - y_1};$$

$$z = z_a + (z_b - z_a)\frac{x - x_a}{x_b - x_a}.$$

Фактично метод Z-буфера здійснює порозрядне сортування за x і y, а потім сортування за z, вимагаючи всього одного порівняння для кожного пікселя кожної грані.

Метод Z-буфера працює виключно в просторі картинної площини і не вимагає ніякої попередньої обробки даних. Порядок, в якому грані виводяться на екран, не грає ніякої ролі.

Для економії пам'яті можна намалювати не все зображення відразу, а малювати частинами. Для цього картинна площина розбивається на частини (звичайно це горизонтальні смуги) і кожна така частина оброблюється незалежно. Розмір пам'яті під буфер визначається розміром найбільшої з цих частин.

Більшість сучасних графічних станцій містять в собі графічну плату з апаратною реалізацією Z-буфера, часто включаючи і апаратну реалізацію (тобто перетворення зображення з координатного подання в растрове) граней разом із зафарбовуванням *Гуро*. Подібні карти забезпечують дуже високу швидкість рендирінгу аж до декількох мільйонів граней за секунду. Середні часові витрати

в них складають O(N), де N — загальна кількість граней. Одним з основних недоліків Z-буфера (крім великого об'єму потрібної під буфер пам'яті) є надмірність обчислень: здійснюється вивід всіх граней незалежно від того, видимі вони чи ні. І якщо, наприклад, піксель накривається десятьма різними лицьовими гранями, то для кожного відповідного пікселя кожній з цих десяти граней необхідно провести розрахунок кольору. Під час використання складних моделей освітленості (наприклад, моделі Фонга) і текстур ці обчислення можуть потребувати дуже великих часових витрат.

Розглянемо як приклад модель будівлі з кімнатами і всім, що знаходиться всередині них. Загальна кількість граней в подібній моделі може складати сотні тисяч і мільйони. Проте, знаходячись всередині однієї з кімнат цієї будівлі, спостерігач реально бачить тільки невелику частину граней (декілька тисяч). Тому вивід всіх граней є недозволенною витратою часу.

Існує декілька модифікацій методу Z-буфера, що дозволяють помітно скоротити кількість граней, що виводяться. Одним найбільш потужних і елегантних є метод *ієрархічного Z-буфера*.

Метод ієрархічного Z-буфера використовує відразу всі три типи когерентності в сцені — в картинній площині (Z-буфері), в просторі об'єктів і часову когерентність.

Назвемо грань прихованою (невидимою) по відношенню до Z-буфера, якщо для будь-якого пікселя картинної площини, що накривається цією гранню, глибина відповідного пікселя грані не менше значення в Z-буфері. Ясно, що виводити приховані грані немає сенсу, оскільки вони нічого не змінюють (вони завідомо не є видимими).

Куб (прямокутний паралелепіпед) назвемо прихованим по відношенню до Z-буфера, якщо всі його лицьові грані є прихованими по відношенню до цього Z-буфера.

Такий підхід спирається на когерентність в об'єктному просторі і дозволяє легко відкинути основну частину невидимих граней.

Для полегшення перевірки грані на приховану можна використовувати Z-*піраміду*. Її нижнім рівнем є сам Z-буфер. Для побудови наступного рівня пікселі об'єднуються в групи по 4 (2×2) і з їх глибин вибирається найбільша. Таким чином, наступний рівень виявляється теж буфером, але його розмір вже буде менший результуючого в 2 рази за кожним виміром. Аналогічно будуються і решта рівнів піраміди до тих пір, поки ми не дійдемо рівня, що складається з єдиного пікселя, що є вершиною Z-піраміди.

Першим кроком перевірки грані на прихованість буде порівняння її мінімальної глибини із значенням у вершині Z-піраміди. Якщо мінімальна глибина грані виявляється більше, то грань прихована. Інакше грань розбивається на 4 частини і порівняння проводиться на наступному рівні піраміди. Якщо на жодному з

проміжних рівнів прихованість грані встановити не вдалося, то здійснюється перехід до останнього рівня, на якому грань растеризується, і проводиться піксельне порівняння з Z-буфером. Найбільш простою є перевірка на вершині піраміди, найбільш трудомісткою — перевірка в її основі. Застосування Z-піраміди дозволяє використовувати когерентність в картинній площині — сусідні пікселі швидше за все відповідають одній і тій же грані. А отже, значення глибин в них відрізняється мало. Ясно, що чим раніше видима грань буде виведена, тим більше невидимих граней будуть відкинуті відразу ж. Висловлене міркування дозволяє використовувати когерентність за часом. Для цього ведеться список тих граней, які були видимі в поточному кадрі. Виведення наступного кадру починається з виведення саме цих граней (щоб уникнути їх повторного виведення, вони позначаються як вже виведені). Тільки після цього здійснюється виведення всього дерева.

13.5. Алгоритми впорядкування

Підхід, заснований на послідовному виведенні на екран в певному порядку всіх граней, може бути успішно використаний і для побудови складніших сцен.

Подібний алгоритм можна описати таким чином:

```
sort objects by z
for all objects
for all visible pixels paint
```

Тим самим методи впорядкування виносять порівняння за глибиною за межі циклів і проводять сортування граней явним чином.

Методи впорядкування є гібридними методами, що здійснюють порівняння і розбиття граней в об'єктному просторі, а для безпосереднього накладення однієї грані на іншу використовують растрові властивості дисплея.

Впорядкуємо всі лицьові грані так, щоб під час їх виведення в цьому порядку виходило коректне зображення сцени. Для цього необхідно, щоб для будь-яких двох граней P і Q та з них, яка під час виведення може закривати іншу, виводилася пізніше. Таке впорядкування звичайно називається *back-to-front*, оскільки спочатку виводяться дальші грані, а потім ближчі.

Існують різні методи побудови подібного впорядкування. Разом з тим нерідкі випадки, коли задані грані упорядкувати не можна (рис 13.7, а). Тоді необхідно провести додаткове розбиття граней так, щоб множину граней, що вийшла після розбиття, вже можна було впорядкувати.

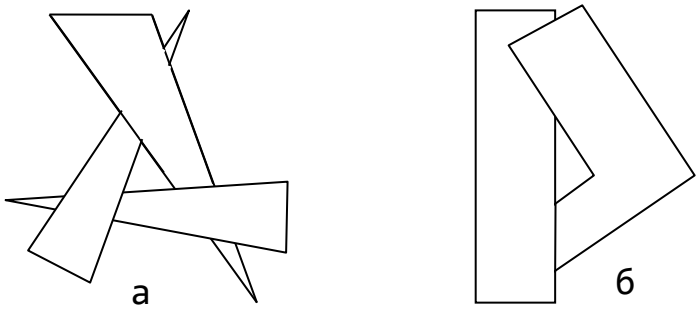


Рис. 13.7. Випадки неможливості впорядкування граней

Відмітимо, що будь-які дві опуклі грані, що не мають загальних внутрішніх точок, можна упорядкувати завжди. Для неопуклих граней це в загальному випадку невірно (рис. 13.7, б).

13.5.1. Метод сортування за глибиною. Алгоритм художника

Цей метод є найпростішим з методів, заснованих на впорядкуванні граней. Як художник спочатку малює дальші об'єкти, а потім поверх них ближчі, так і метод сортування за глибиною спочатку упорядковує грані в міру наближення до спостерігача, а потім виводить їх в цьому порядку.

Метод ґрунтується на наступному простому спостереженні: якщо для двох граней A і B найдальша точка грані A ближче до спостерігача (картинної площини), ніж найближча точка грані B, то грань B ніяк не може закрити грань A від спостерігача.

Тому якщо наперед відомо, що для будь-яких двох лицьових граней найближча точка однієї з них знаходиться далі, ніж найдальша точка іншої, то для впорядкування граней досить просто відсортувати їх за відстанню до спостерігача (картинної площини).

Проте таке не завжди можливо: може зустрітися така пара граней, що найдальша точка однієї знаходиться до спостерігача не ближче, ніж найближча точка іншої.

На практиці часто зустрічається наступна реалізація цього алгоритму: множина лицьових граней сортується за найближчою відстанню до картинної площини (спостерігача) і потім ці грані виводяться у порядку наближення до спостерігача. Як алгоритми сортування можна використовувати або швидке сортування, або порозрядне (*radix sort*).

Хоча подібний підхід і працює в переважній більшості випадків, проте можливі ситуації, коли просто сортування за відстанню до картинної площини не забезпечує правильного впорядкування граней (рис. 13.8) — так, грань B буде помилково виведена раніше, ніж грань A; тому після сортування бажано перевірити порядок, в якому грані виводитимуться.

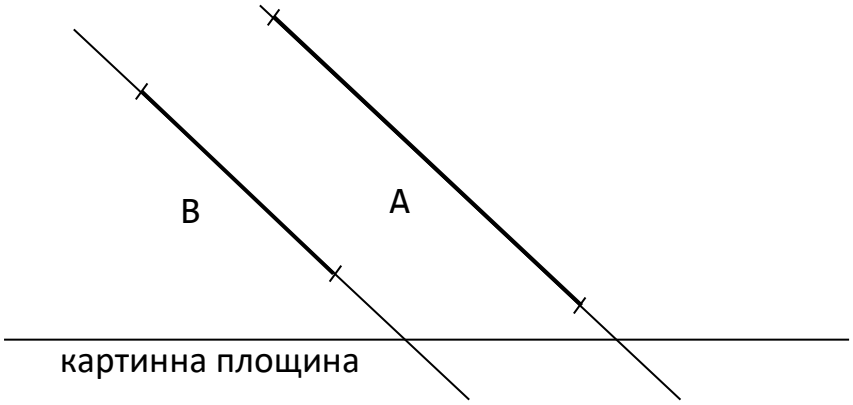


Рис. 15.8. Помилкове впорядкування граней

Пропонується наступний алгоритм цієї перевірки. Для простоти вважатимемо, що розглядається паралельне проєктування по осі OZ.

Перед виведенням чергової грані P слід переконатися, що ніяка інша грань Q, яка стоїть в списку пізніше, ніж P, і проєкція якої на вісь OZ перетинається з проєкцією грані P (якщо перетину немає, то порядок виведення P і Q визначений однозначно), не може закриватися гранню P. В цьому випадку грань P дійсно повинна бути виведена раніше, ніж грань Q.

Нижче приведені 4 тести в порядку зростання складності перевірки.

1. Чи перетинаються проєкції цих граней на вісь OX?
2. Чи перетинаються проєкції цих граней на вісь OY?

Якщо хоч б на одне із двох питань одержана негативна відповідь, то проєкції граней P і Q на картинну площину не перетинаються і, отже, порядок, в якому вони виводяться, не має значення. Тому вважатимемо, що грані P і Q впорядковані вірно.

Для перевірки виконання цих умов дуже зручно використовувати обмежуючі тіла.

У разі, коли обидва ці тести дали ствердну відповідь, проводяться наступні тести.

3. Чи знаходяться грань P і спостерігач по різні сторони від площини, що проходить через грань Q?
4. Чи знаходяться грань Q і спостерігач по одну сторону від площини, що проходить через грань?

Якщо хоч б на одне з цих питань одержана ствердна відповідь, то вважаємо, що грані P і Q впорядковані вірно, і порівнюємо P з наступною гранню.

У випадку, якщо жоден з тестів не підтвердив правильність впорядкування граней P і Q, перевіряємо, чи не слід поміняти ці грані місцями. Для цього проводяться тести, що є аналогами тестів 3 і 4 (очевидно, що знову проводити тести 1 і 2 не має сенсу):

3'. Чи знаходяться грань Q і спостерігач по різні сторони від площини, що проходить через грань P?

4'. Чи знаходяться грань P і спостерігач по одну сторону від площини, що проходить через грань Q?

У випадку, якщо жоден з тестів 3, 4, 3', 4' не дозволяє з упевненістю визначити, яку з цих двох граней потрібно виводити раніше, одна з них розбивається площиною, що проходить через іншу грань і питання про впорядкування цілої грані та частин розбитої грані легко вирішується за допомогою тестів 3 або 4 (3' або 4').

Можливі ситуації, коли не дивлячись на те, що грані P і Q впорядковані вірно, їх розбиття все ж таки буде проведено (алгоритм створює надмірне розбиття). Подібний випадок зображений на рис. 13.9, де для кожної вершини вказана її глибина.

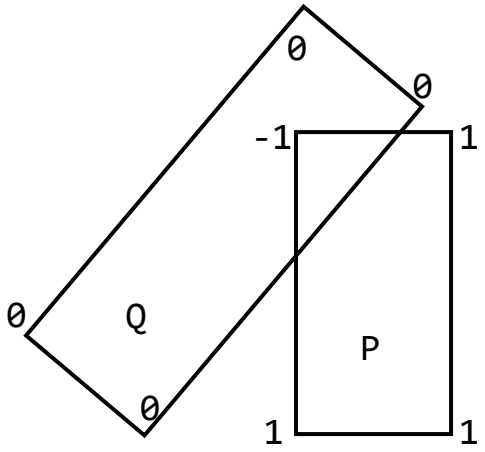


Рис. 13.9. Надмірне розбиття грані

Методу впорядкування властивий той же недолік, що і методу Z-буфера, а саме необхідність виведення всіх лицьових граней. Щоб уникнути цього, можна його модифікувати таким чином: грані виводяться в зворотному порядку — починаючи з найближчих і закінчуючи найдальшими (*front-to-back*). Під час виведення чергової грані малюються тільки ті пікселі, які ще не були виведені. Як тільки весь екран буде заповнений, вивід граней можна припинити.

Але тут потрібен механізм відстежування того, які пікселі були виведені, а які ні. Для цього можуть бути використані найрізноманітніші структури, від ліній горизонту до бітових масок.

13.5.2. Метод двійкового розбиття простору

Існує інший, досить елегантний і гнучкий спосіб впорядкування граней. Кожна площина в об'єктному просторі розбиває весь простір на два півпростори. Вважаючи, що ця площина не перетинає жодну з граней сцени, одержуємо розбиття множини всіх граней на дві множини (кластери), що не перетинаються. Кожна грань потрапляє в той або інший кластер залежно від того, в якому півпросторі щодо площини розбиття ця грань знаходиться.

Ясно, що жодна з граней, які лежать в півпросторі, що не містить спостерігача, не може закривати собою жодну з граней, які лежать в тому ж півпросторі, в якому знаходиться і спостерігач (з невеликими змінами це працює і для паралельного проєктування).

Для побудови правильного зображення сцени необхідно спочатку виводити грані з дальнього кластера, а потім з ближнього.

Застосуємо запропонований підхід для впорядкування граней всередині кожного кластера. Для цього виберемо дві площини, що розбивають кожний з кластерів на два підкластери.

Повторюючи описаний процес до тих пір, поки в кожному кластері, що вийшов, залишиться не більше однієї грані (рис. 13.10).

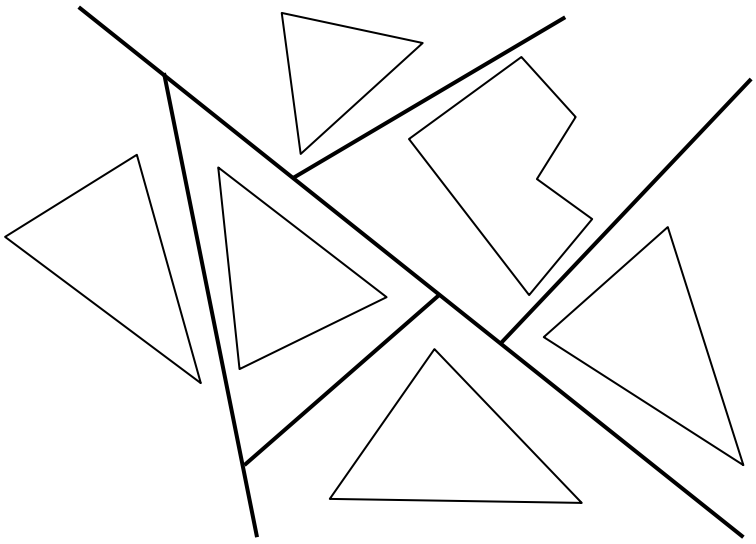


Рис. 13.10. Двійкове розбиття простору

Звичайно в якості площини розбиття вибирається площина, що проходить через одну з граней. Всі грані, що перетинаються цією площиною, розбиваються вздовж неї, а частини, що вийшли під час розбиття поміщаються у відповідні піддерева.

Для прикладу розглянемо сцену, що зображена на рис. 13.11.

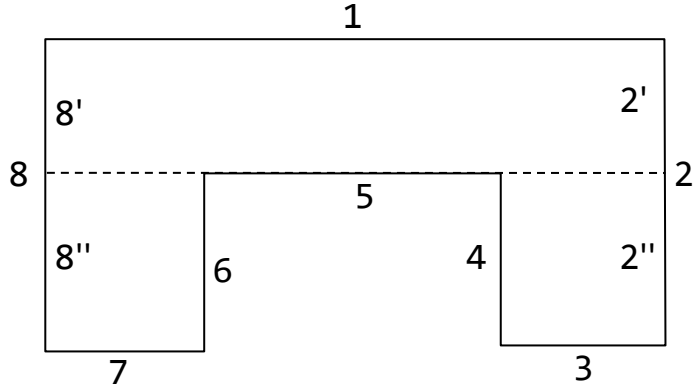


Рис. 13.11. Сцена для прикладу двійкового розбиття простору

Площина, що проходить через грань 5, розбиває грані 2 та 8 на частини 2', 2'', 8' та 8'', і вся множина граней (з урахуванням розбиття граней 2 і 8) розпадається на два кластери (1, 8',2') і (2'', 3, 4, 6, 7, 8''). Вибравши для першого кластера в якості площини, що розбиває — площину, що проходить через грань 6 розбиваємо його на два підкластери (7, 8') і (2', 3, 4). Кожне наступне розбиття відділятиме лише по одній грані з кластерів, що залишилися. В результаті одержимо наступне дерево (рис. 13.12).

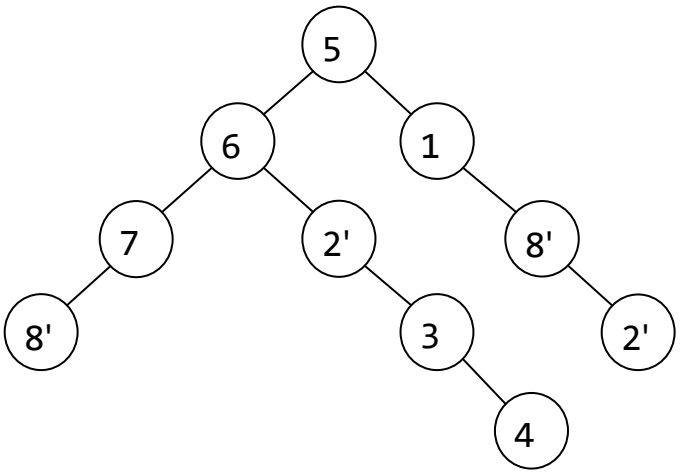


Рис. 13.12. BSP-дерево

Таким чином, процес побудови BSP-дерев (*Binary Space Partition*) полягає у виборі площини (грані), що розбиває, розбитті множини всіх граней на дві частини (це може вимагати розбиття граней на частини) і рекурсивного застосування описаної процедури до кожної з частин, що вийшли.

Зауваження. Якщо грань, що перевіряється, лежить в площині розбиття, то її можна помістити в будь-яку з частин. Існують варіанти методу, які з кожним вузлом дерева зв'язують список граней, які лежать в тій площині, що розбиває.

Природнім чином виникає питання про побудову оптимального дерева. Існує два основних критерії оптимальності:

- отримання максимально збалансованого дерева (коли для будь-якого вузла кількість граней в правому піддереві мінімально відрізняється від кількості граней в лівому піддереві); це забезпечує мінімальну висоту дерева (і відповідно найменшу кількість перевірок);
- мінімізація кількості розбиттів; однією із найбільш неприємних властивостей BSP-дерев є розбиття граней, що призводить до значного збільшення їх загальної кількості і, як наслідок, до росту витрат (пам'яті та часу) на зображення сцени.

На жаль, ці критерії, як правило, є взаємовиключними. Тому зазвичай вибирається деякий компромісний варіант. Наприклад, в якості критерію вибирається сума висоти дерева і кількість розбиттів із заданими вагами.

Однією з основних переваг цього методу є повна незалежність дерева від параметрів проєктування (положення центру проєктування, напряму

проєктування та ін.), що робить його досить зручним для побудови серій зображень однієї і тієї ж сцени з різних точок спостереження. Ця обставина призвела до того, що BSP-дерева стали широко використовуватися у ряді систем віртуальної реальності. Зокрема, видалення невидимих граней в широко відомих іграх DOOM, Quake і Quake II засновано на залученні саме BSP-дерев.

До недоліків методу BSP-дерев відносяться явно надмірна необхідність розбиття граней, особливо актуальна під час роботи з великими сценами, і не локальність BSP-дерев — навіть незначна локальна зміна сцени може спричинити за собою зміну практично всього дерева.

Внаслідок їх не локальності подання великих сцен у вигляді BSP-дерев виявляється занадто складним, оскільки призводить до занадто великої кількості розбиттів. Для боротьби з цим явищем можна розділити всю сцену на декілька частин, які можна легко впорядкувати між собою, і для кожної з цих частин побудувати своє BSP-дерево, що містить тільки ту частину сцени, яка потрапляє в цей фрагмент.

Запитання для самоконтролю

1. В чому полягає задача видалення невидимих ребер та граней?
2. В чому суть алгоритму відсікання нелицьових граней?
3. Опишіть послідовність алгоритму Робертса?
4. Коротко опишіть метод трасування променів.
5. Коротко опишіть метод Z-буфера. В чому його відмінність від алгоритму трасування променів?
6. Розкрийте суть алгоритму художника.
7. Опишіть метод двійкового розбиття простору.

14. ФІЛЬТРАЦІЯ ЗОБРАЖЕНЬ

Фільтрація зображень — це процес обробки зображень, що включає застосування фільтрів для покращення якості, видалення шуму (наприклад, шумів або імпульсних перешкод за допомогою медіанної фільтрації) або виділення певних ознак, і може виконуватись у просторовій (безпосередньо з пікселями) або частотній області. Основна мета — змінити або виділити певну інформацію в зображенні, наприклад, розмити його, підсилити краї або усунути небажані артефакти, щоб зробити зображення чистішим або зрозумілішим для аналізу чи сприйняття.

Невеликі групи пікселів зображення (наприклад 3x3 або 5x5) називаються ядром (kernel) і описуються простими матрицями. Ці ядра використовуються для операції згортки (convolution). Згортка виконується шляхом множення значень інтенсивності кольору пікселя та його сусідів на значення коефіцієнтів ядра з використанням ковзного вікна. Або простіше кажучи, коли кожен піксель на вихідному зображенні є функцією сусідніх пікселів (включно з собою) на вхідному зображенні, ядро є цією функцією.

Звичайно ядро не завжди квадратне, може бути і прямокутне, але зазвичай це квадрат зі стороною з непарної кількістю пікселів (3x3, 5x5, 7x7, 9x9 і т.д.).

Згортки також іноді називають фільтрами оскільки можна провести аналогію з фільтрацією сигналів. Фільтри бувають лінійні — ті, які зберігають форму вхідного сигналу, та нелінійні — ті, які змінюють форму вхідного сигналу.

Загальний вираз для опису згортки наступний:

$$g(x,y) = \omega \cdot f(x,y) = \sum_{i=-a}^{a} \sum_{j=-b}^{b} \omega(i,j) \cdot f(x-i, y-j), \qquad (14.1)$$

де g(x,y) — фільтроване зображення; f(x,y) — оригінальне зображення; ω — ядро фільтра. Кожен елемент ядра фільтра має індекс (i,j), де −a<=i<=a, а також −b<=j<=b.

14.1. ВИДАЛЕННЯ ШУМІВ МЕДІАННИМ ФІЛЬТРОМ

Медіанний фільтр — це нелінійний цифровий фільтр, який використовується для видалення шуму з сигналів та зображень, особливо імпульсного («сіль і перець»), шляхом заміни кожного пікселя (або значення сигналу) медіаною всіх значень у його вікні. Він ефективно зберігає краї, на відміну від середньозваженого фільтра. Процес включає сортування значень у вікні, а потім вибір середнього (медіанного) значення, що робить його стійким до викидів.

Розглянемо приклад використання медіанного фільтра для одновимірного зображення. Використаємо фільтр з ядром 1x3 для вхідного зображення розміром 1x10. Для обробки країв зображення використаємо повторення значень граничних пікселів (padding).Фільтр проходить ядром 1x3 і для поточної

вибірки з трьох пікселів виконує їх сортування з наступним обранням значення, що стоїть посередині (рис. 14.1).

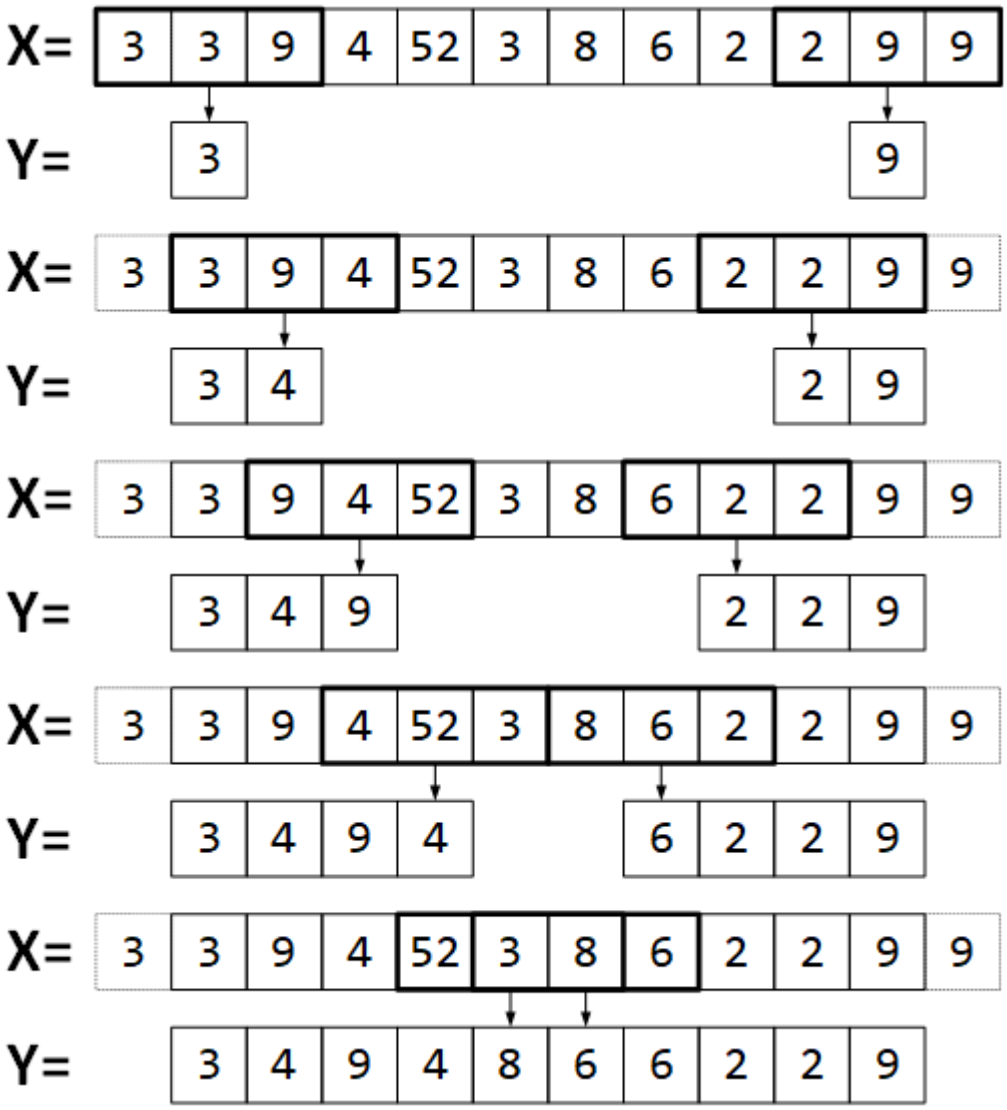


Рис. 14.1. Процес фільтрації одновимірного зображення

На рис. 14.1 показаний паралельний процес фільтрації, який просувається з обох сторін зображення. Зазвичай цей процес послідовний і відбувається зліва на право і згори до низу.

Тепер розглянемо двовимірний медіанний фільтр. Використаємо зображення розміром 6x6 пікселів та ядро фільтра розміром 3x3 (рис. 14.2).

В першому ядрі фільтра на рис. 14.2 після сортування отримаємо таку послідовність значень: [0 0 1 1 **1** 2 2 4 4]. Тут медіанне значення 1, а отже воно і буде результатом роботи медіанного фільтра для цього пікселя.

Останній піксель в зображенні потребував доповнення граничними пікселями. У підсумку була отримана така послідовність значень: [0 0 0 0 **2** 3 3 5 5].

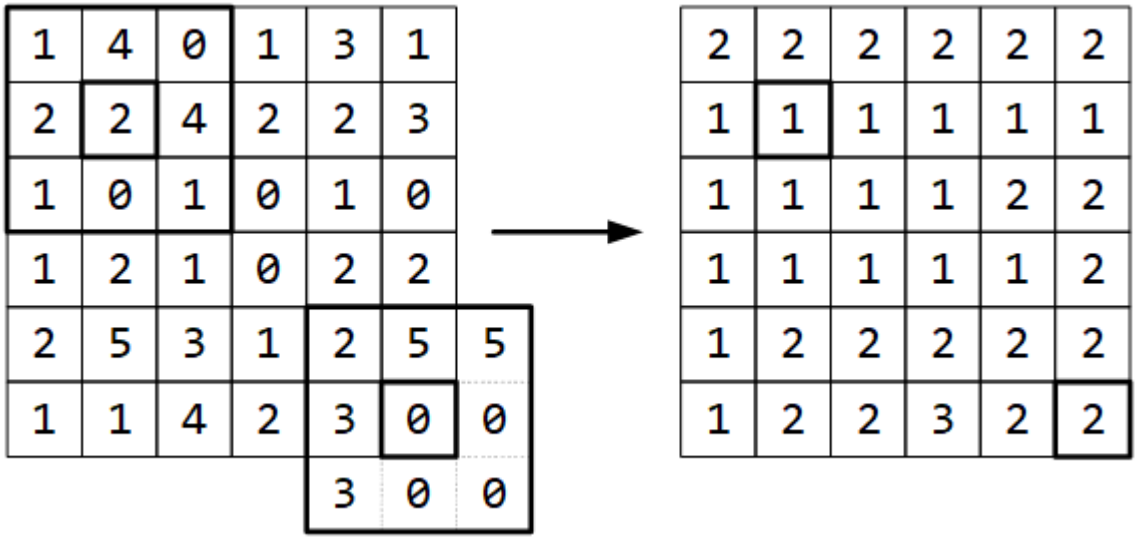


Рис. 14.3. Процес медіанної фільтрації двовимірного зображення

Переваги медіанного фільтра:

- чудово усуває імпульсні шуми («сіль і перець»), не розмиваючи значно краї зображення, на відміну від лінійних фільтрів;
- краще зберігає різкі переходи та контури об'єктів.

Медіанний фільтр застосовується в таких галузях:

- обробка зображень (фотографії, медичні зображення);
- цифрова обробка сигналів;
- для попередньої обробки перед іншими алгоритмами (наприклад, розпізнаванням образів).

Ефективність медіанного фільтра для видалення шумів на зображенні можна оцінити за прикладом (рис. 14.3).

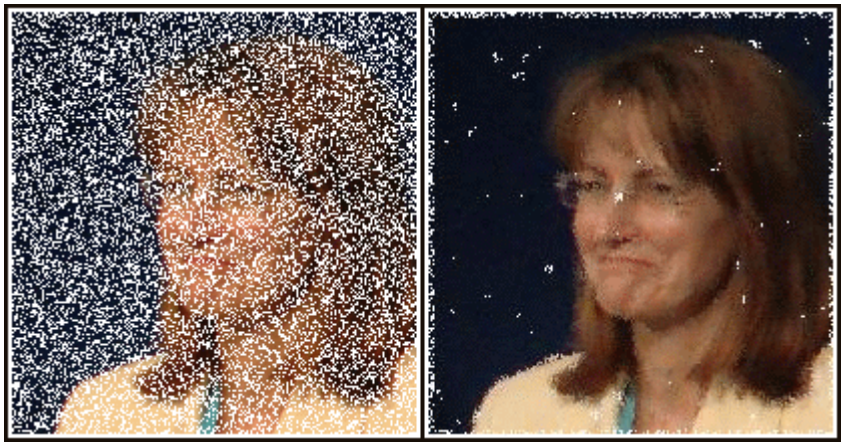


Рис. 14.3. Приклад використання медіанного фільтра

Початкове зображення на рис. 14.3 містить велику кількість шуму типу «сіль» — дрібні світлі плями. Після застосування медіанного фільтра практично всі гуми зникли. Щоб остаточно прибрати залишки шуму можна застосувати медіанний фільтр до вже відфільтрованого зображення ще раз.

14.2. Середньозважений фільтр

Алгоритми розмивання зображень — це методи обробки зображень, що зменшують деталізацію шляхом усереднення значень пікселів, найпоширеніші з яких ґрунтуються на усередненні сусідніх пікселів або на згортці з певним ядром.

Середньозважений фільтр — це метод обробки сигналів та зображень для зменшення шуму, що полягає у заміні значення кожного елемента (пікселя) зваженою сумою (середнім) його сусідів, де різні сусіди можуть мати різну «вагу» (важливість), хоча часто використовується просто середнє арифметичне, що є окремим випадком. Це дозволяє згладити різкі зміни та прибрати дрібні спотворення, але може призвести до розмиття деталей.

Середньозважені фільтри використовуються для розмиття зображень або для підсилення чіткості зображення. Вони відносяться до лінійних фільтрів.

В англомовній літературі є чіткий розподіл між простим середньозваженим фільтром для розмивання зображень, який називається Box або Mean Blur та фільтрами, які використовують ваги — Gaussian Blur. Ми будемо називати середньозваженим фільтром саме Box Blur, а Gaussian Blur — фільтром Гаусса .

Середньозважений фільтр має ядро з одиничними ваговими коефіцієнтами для кожного пікселя (рис. 14.4).

1	1	1
1	1	1
1	1	1

Рис. 14.4. Ядро Box Blur 3x3 пікселя

Середньозважений фільтр заміняє значення кожного пікселя зображення на середнє значення всіх його сусідів, включно з ним самим. Середнє значення вираховується як середнє арифметичне — додаються значення всіх пікселів, що потрапили в ядро фільтра і потім результат ділиться на загальну кількість пікселів у ядрі. Узагальнена формула згортки для розрахунку значення пікселя за ядром Box Blur з непарним розміром сторони N буде наступна:

$$g(x,y) = \frac{1}{N^2} \cdot \sum_{i=-N/2}^{N/2} \sum_{j=-N/2}^{N/2} f(x+i, y+j). \qquad (14.2)$$

Візьмемо той самий приклад, що був для медіанного фільтра і застосуємо середньозважений фільтр (рис. 14.5).

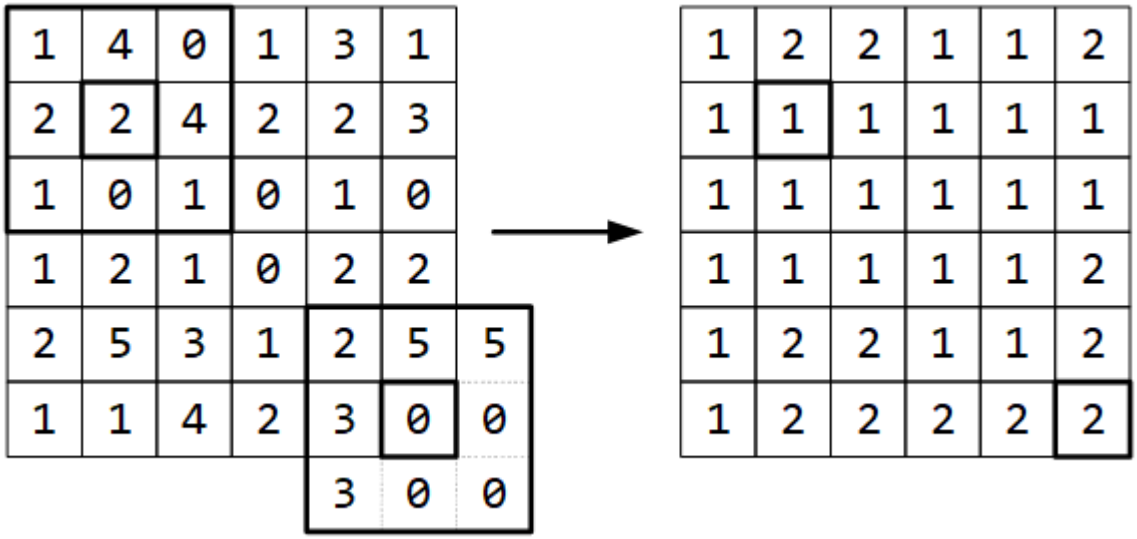


Рис. 14.5. Процес середньозваженої фільтрації двовимірного зображення

Щодо обробки граничних пікселів зображення, то один з методів ми вже розглядали під час використання медіанного фільтра. Як і в прикладі вище, тоді ми використовували «padding» — повторення значень граничних пікселів. Існують й інші методи: «mirroring» — віддзеркалення пікселів, або заповнення додаткових пікселів нулями. Різні методи на прикладі ядра розміром 3x3 подані на рис. 14.6.

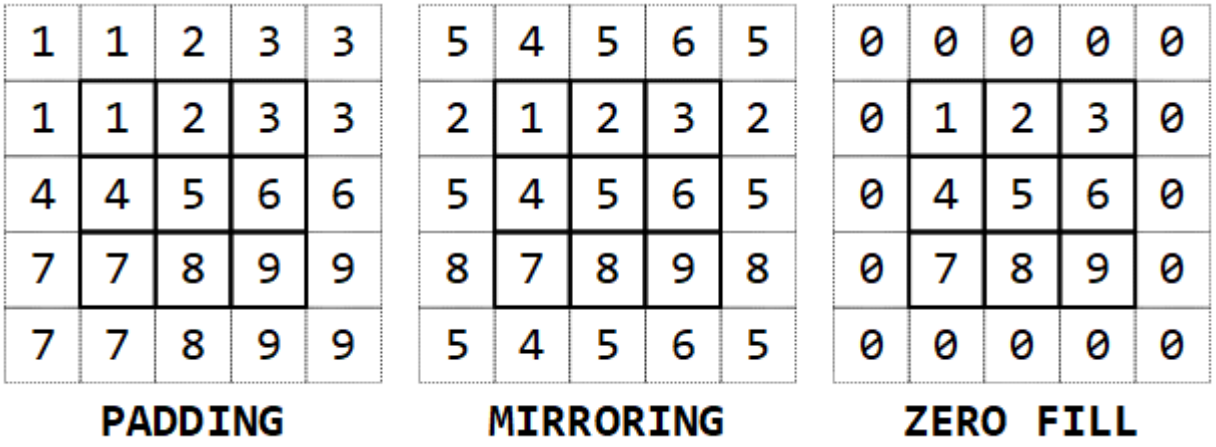


Рис. 14.6. Різні методи обробки крайніх пікселів

Нижче подано приклад зображення до якого застосований середньозважений фільтр Box Blur. Результат відрізняється від розфокусованого зображення, оскільки точка світла з розфокусованого об'єктива має вигляд як світловий диск, а Box Blur дає квадратик (рис. 14.7).

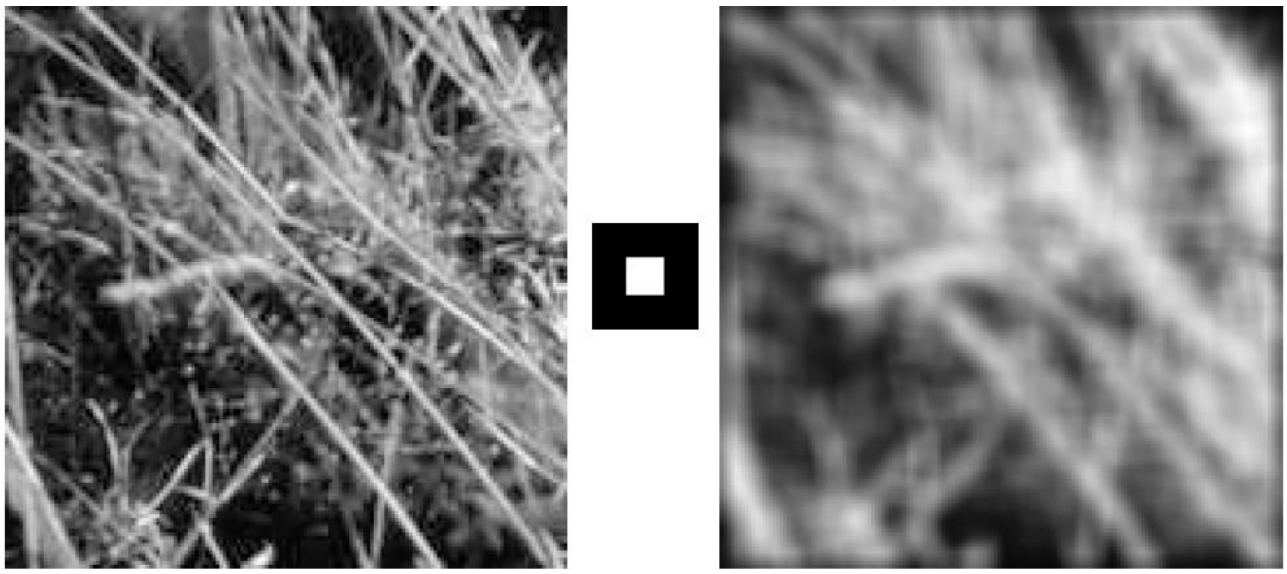


Рис. 14.7. Приклад застосування Box Blur

Для кольорових зображень фільтрація застосовується окремо для кожного кольору. Для пришвидшення роботи алгоритму фільтрації, розмивання роблять одновимірним фільтром спочатку в горизонтальній площині, а потім у вертикальній. Щоб зменшити кількість арифметичних операцій, суму пікселів не перераховують для кожного `i`-го пікселя. Достатньо від поточної суми відняти крайній лівий сусідній піксель і додати новий правий сусідній піксель (рис. 14.8).

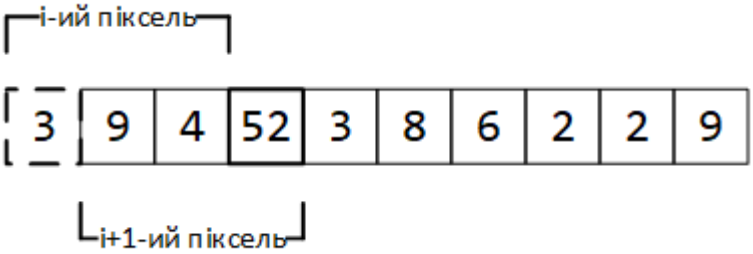


Рис. 14.8. Швидкий метод розрахунку одновимірного Box Blur

В прикладі на рис. 14.8 спочатку рахується сума трьох перших пікселів і зберігається. Щоб отримати розмите значення `i`-го пікселя, потрібно просто поділити суму на 3. Для отримання наступної суми необхідно від попередньої відняти піксель зі значенням 3 і додати піксель зі значенням 52. Таким чином ми отримаємо нову суму. На маленькому фільтрі розміром 3 переваги не буде, але на більших розмірах це буде відчутне пришвидшення.

14.3. Фільтр Гаусса

Фільтр Гаусса або Гауссове згладжування — спосіб розфокусування зображення за допомогою фільтра, який використовує математичну функцію, названу на честь вченого Карла Фрідріха Гаусса. Ефект, створюваний цим способом, називають Гаусовим розмиттям або згладжуванням. Цей ефект широко використовується в графічних програмах, як правило, для зменшення зашумленості зображення та зниження деталізації.

Функція Гаусса може бути одновимірною та двовимірною. Для розмиття зображень зазвичай використовують двовимірну функцію, що задається формулою:

$$G(x,y) = \frac{1}{2 \cdot \pi \cdot \sigma^2} \cdot e^{-\frac{(x^2+y^2)}{2 \cdot \sigma^2}}, \qquad (14.3)$$

де x — це відстань від початку координат в осі абсцис, y — це відстань від початку координат в осі ординат, а σ — стандартне відхилення розподілу Гаусса. Коли метод застосовується у двох вимірах, отримується поверхня, контури якої — концентричні кола розподілу Гаусса з центральної точки (рис. 14.9). Значення з цього розподілу використовуються для створення матриці згортки. Для кожного нового значення пікселя визначається середнє зважене в околі пікселя. Значення поточного оригінального пікселя має більшу вагу (найвище значення розподілу Гаусса), а сусідні пікселі отримують все меншу вагу в залежності від того, наскільки далеко вони знаходяться від поточного оригінального пікселя. Це надає ефект розмитості, який зберігає кордони та краї краще, ніж інші, аналогічні фільтри розмиття.

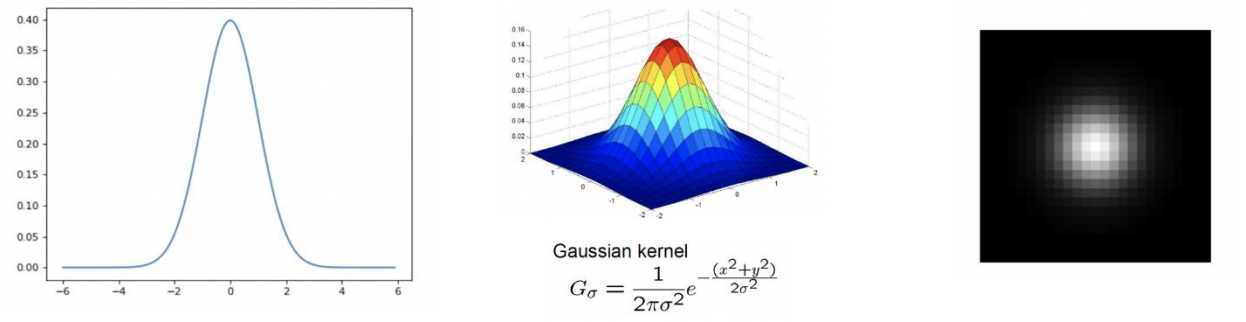


Рис. 14.9. Функція Гаусса (1 і 2 вимірна)

Значення стандартного відхилення σ — впливає на значення вагових коефіцієнтів ядра фільтра Гаусса. Чим менше це значення тим меншу вагу мають пікселі, що віддалені від центру (рис. 14.10).

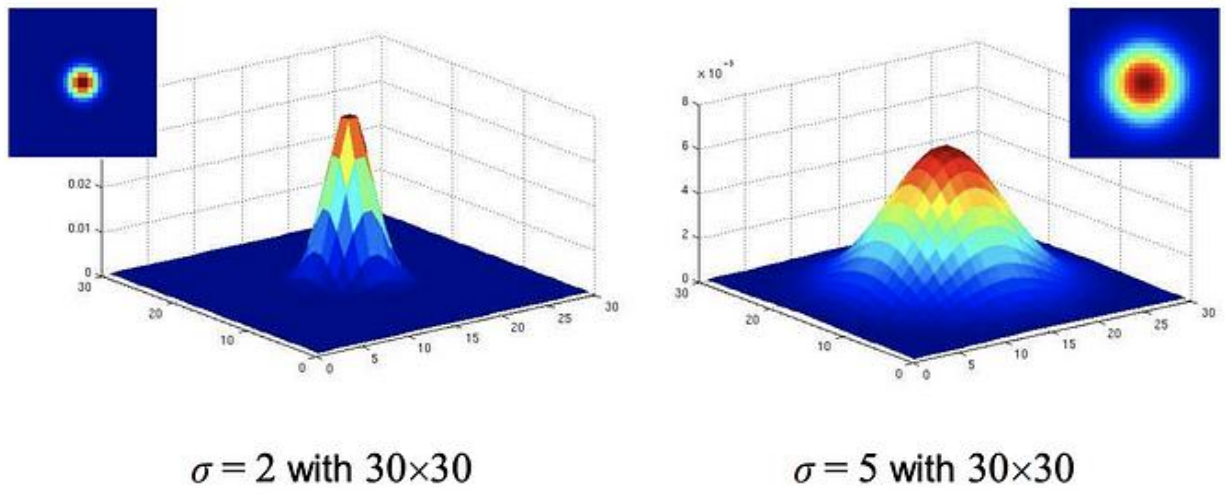


Рис. 14.10. Ядро фільтра Гаусса розміром 30x30 з σ=2 і σ=5

З формули (14.3) очевидно, що значення ядра фільтра Гаусса будуть дійсними числами, що є не дуже добре для швидкості розрахунків. Тому часто використовують попередньо розраховані ядра фільтра Гаусса зі стандартним відхиленням σ≈1 (рис. 14.11).

1/16

1	2	1
2	4	2
1	2	1

1/273

1	4	7	4	1
4	16	26	16	4
7	26	41	26	7
4	16	26	16	4
1	4	7	4	1

1/1003

0	0	1	2	1	0	0
0	3	13	22	13	3	0
1	13	59	97	59	13	1
2	22	97	159	97	22	2
1	13	59	97	59	13	1
0	3	13	22	13	3	0
0	0	1	2	1	0	0

Рис. 14.11. Ядра фільтра Гаусса з розмірами 3x3, 5x5 і 7x7 для σ≈1

Двовимірну функцію Гаусса можна розкласти на добуток двох одновимірних:

$$G(x,y) = \left(\frac{1}{\sigma \cdot \sqrt{2 \cdot \pi}} \cdot e^{-\frac{x^2}{2 \cdot \sigma^2}}\right) \cdot \left(\frac{1}{\sigma \cdot \sqrt{2 \cdot \pi}} \cdot e^{-\frac{y^2}{2 \cdot \sigma^2}}\right). \qquad (14.4)$$

Таким чином, реалізувати Гауссове розмиття можна аналогічно до середньозваженого — спочатку застосувати до зображення горизонтальний фільтр, а потім вертикальний.

Порівняння середньозваженого та Гауссового розмиття (рис. 14.12).

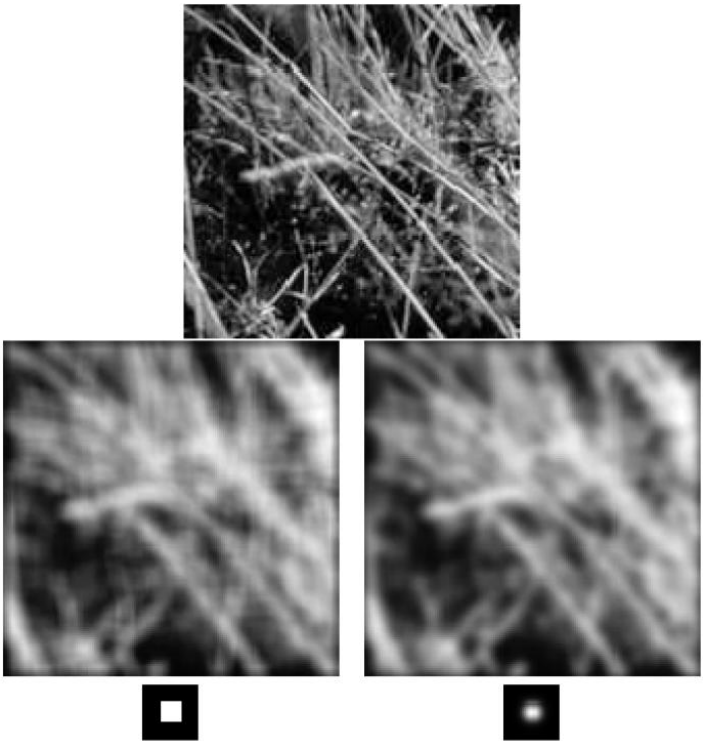


Рис. 14.12. Оригінальне зображення, середньозважене і Гауссове розмиття

Хоча Гауссове розмиття вимагає більше розрахунків, візуальний ефект від його застосування має кращий вигляд, ніж застосування середньозваженого розмиття. Це досягається більш плавними переходами між сусідніми пікселями з різною інтенсивністю (рис .14.13).

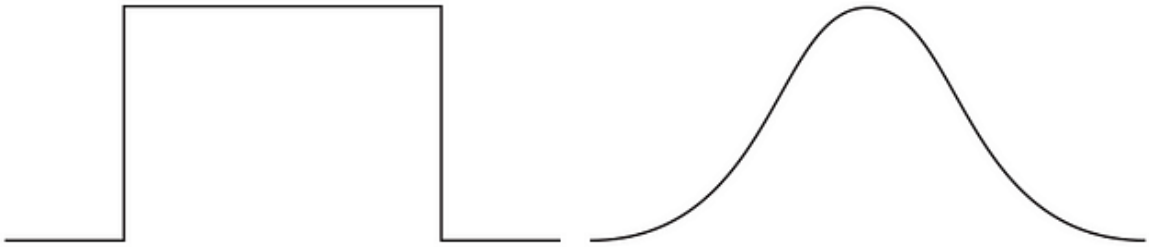


Рис. 14.13. Двовимірний графік зміни інтенсивності пікселів для Box Blur і Gaussian Blur

Проте для усунення шумів типу «сіль і перець» медіанний фільтр залишається найкращим варіантом (рис. 14.14).

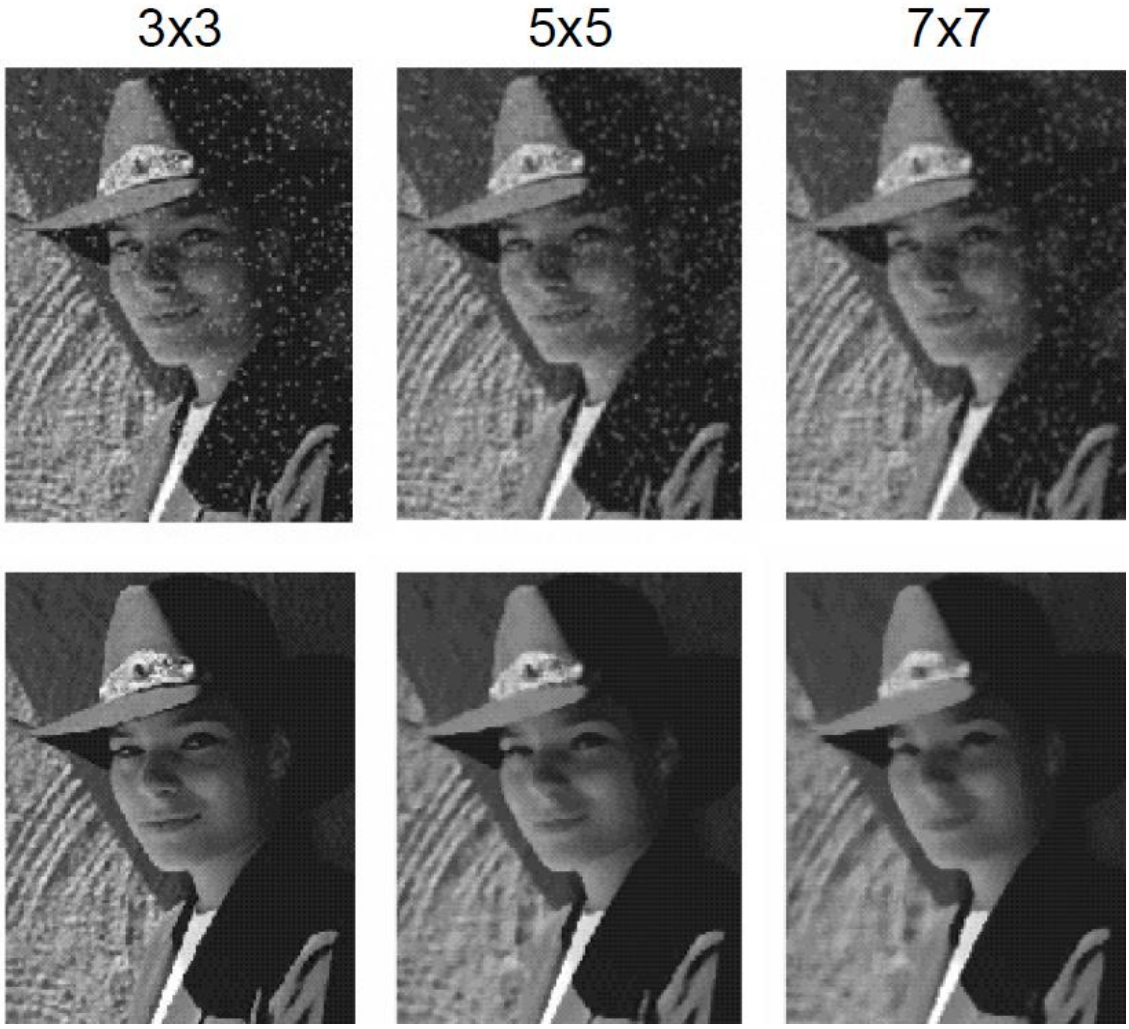


Рис. 14.1.4. Порівняння Гауссового розмиття і медіанного фільтра

Гауссове згладжування зазвичай використовується з визначенням країв. Більшість алгоритмів виявлення країв чутливі до шуму; двовимірний фільтр Лапласа, побудований на основі дискретизації оператора Лапласа, дуже чутливий до шумного середовища.

Використання фільтра Гаусса перед виявленням країв має на меті зменшити рівень шуму в зображенні, що покращує результат наступного алгоритму визначення країв. Цей підхід зазвичай називають фільтрацією Лапласа-Гауса або фільтрацією LoG. Про це більш детально описано в наступному розділі.

Запитання для самоконтролю

1. На які два типи діляться фільтри?
2. Що таке згортка?
3. Що таке ядро згортки?
4. Який принцип роботи медіанного фільтра?
5. Який принцип роботи середньозваженого фільтра?
6. Як розраховуються коефіцієнти для ядра фільтра Гаусса?
7. Яка принципова різниця між середньозваженим Гауссовим розмиттям?
8. Який тип фільтра найкраще підходить для видалення шуму «сіль і перець»?
9. Який спосіб реалізації середньозваженого фільтра найшвидший?
10. Чи можна замінити двовимірний фільтр Гаусса на одновимірні аналоги?

15. Збільшення чіткості та виявлення контурів

В попередньому розділі ми розглянули поняття фільтрації, згортки та ядра згортки. Також детальніше розглянули деякі фільтри для розмиття зображень. В цьому розділі ми продовжимо розгляд деяких ядер згорток, що застосовуються для обробки зображень, а саме збільшення чіткості зображення та виявлення контурів на зображенні.

15.1. Збільшення чіткості зображення

Застосування розмиття призводить до втрати деяких деталей зображення. Зазвичай ми намагаємося видалити з зображення небажані артефакти. Але бувають випадки, коли необхідно зробити зворотній процес, а саме додати більше чіткості зображенню, щоб розгледіти деталі. Наприклад це актуально для відновлення розмитих зображень особливо тих, де присутня текстова і числова інформація.

Формальний опис алгоритму підвищення чіткості зображення наступний.

1. Здійснити розмиття зображення одним із фільтрів (Box Blur або Gaussian Blur).
2. Відняти результат розмиття від оригінального зображення. Таким чином ми отримаємо високочастотні деталі зображення.
3. Додати високочастотні деталі до оригінального зображення.

Математично цей процес описується формулою:

$$\texttt{image} + (\texttt{image} - \texttt{blurred}) = 2 \cdot \texttt{image} - \texttt{blurred}. \qquad (15.1)$$

Якщо розглядати монохромне зображення, то кожен піксель описаний своєю інтенсивністю. Нам потрібно накласти простий лінійний фільтр, що вдвічі підсилить інтенсивність зображення і відняти фільтр Гаусса (рис. 15.1).

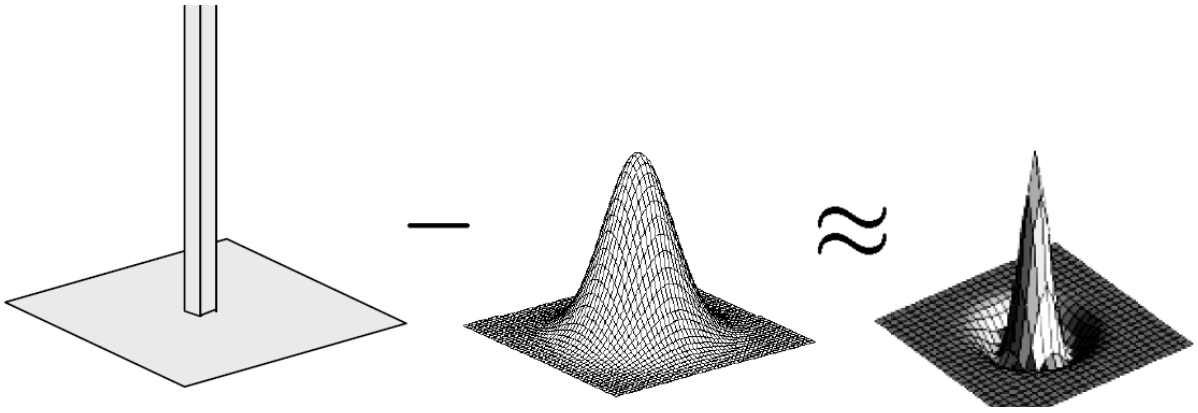


Рис. 15.1. Ілюстрація до збільшення чіткості зображення

Власне з точки зору зміни зображення, нам необхідно підсилити інтенсивність поточного пікселя і водночас зменшити інтенсивність його сусідів. Застосувавши формулу (15.1) ми отримаємо ядра різних розмірів для фільтрів підвищення чіткості зображення (рис. 15.2).

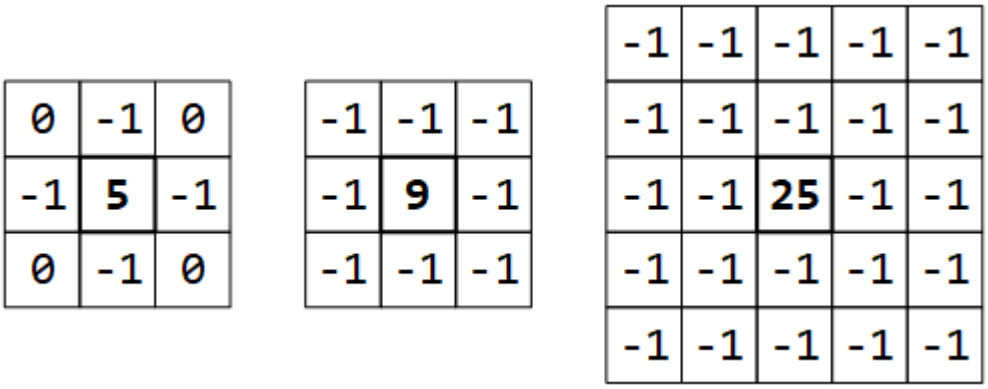


Рис. 15.2. Ядра фільтрів підвищення чіткості зображення

Як бачимо з рисунку, ядра для підвищення чіткості зазвичай мають коефіцієнти -1 для сусідніх пікселів і додатній коефіцієнт для центрального пікселя. Значення коефіцієнта для центрального пікселя підбирається таким чином, щоб сума всіх коефіцієнтів ядра була 1.

15.2. Оператори для виявлення контурів

Одна із задач обробки зображень це виявлення контурів об'єктів, що є на зображенні. Основним підходом є визначення градієнтів зображення в горизонтальному, вертикальному і діагональному напрямках.

Градієнт (∇) — міра зростання або спадання в просторі якоїсь фізичної величини на одиницю довжини. У випадку з зображеннями вимірюють величину зміни значення інтенсивності зображення в заданому напрямку. Фактично це похідні інтенсивності зображення за координатними осями.

Методи, що використовують градієнт, називають також операторами. Їх існує декілька і ми розглянемо деякі з них.

Оскільки всі методи об'єднує розрахунок градієнта, то їх позначення за напрямами буде однакове для всіх методів. Через I(x,y) позначимо інтенсивність пікселя зображення з відповідними координатами. Тоді G_x(x,y) та G_y(x,y) наближені похідні в напрямках X та Y відповідно. Тоді градієнт можна визначити як:

$$\nabla I(x,y) = G(x,y) = \sqrt{G_x^2 + G_y^2}. \quad (15.2)$$

Напрямок градієнта також можна визначити наступним чином:

$$\theta(x,y) = \arctan\left(\frac{G_y(x,y)}{G_x(x,y)}\right) - \frac{3\pi}{4}. \quad (15.3)$$

Зауважте, що кут 0° відповідає вертикальному спрямуванню, такому, що напрямок максимального контрасту від чорного до білого на зображенні лежить зліва направо.

15.2.1. Оператор Робертса

Оператор Робертса (Roberts cross) використовують в обробці зображень та комп'ютерному баченні для виявляння контурів. Він був одним із перших методів виявлення контурів, первинно запропонованим Лоуренсом Робертсом 1963 року. Як у різницевого оператора, ідея оператора Робертса полягає в наближенні градієнта зображення шляхом дискретного диференціювання, якого досягають обчисленням суми квадратів різниць між діагонально сусідніми пікселями [26].

Згідно з Робертсом, метод виявлення контурів повинен мати наступні властивості: видавані контури повинні бути чітко визначеними, тло повинне вносити якомога менше шуму, а яскравість контурів повинна якомога ближче відповідати людському сприйняттю. З огляду на ці критерії та на основі переважної на той час психофізичної теорії Робертс запропонував такі рівняння:

$$y_{i,j} = \sqrt{x_{i,j}}$$
$$z_{i,j} = \sqrt{\left(y_{i,j} - y_{i+1,j+1}\right)^2 + \left(y_{i+1,j} - y_{i,j+1}\right)^2}, \quad (15.4)$$

де x — первинне значення яскравості на зображенні, z — обчислювана похідна, а i,j подають розташування на зображенні.

Результати цієї операції висвітлюватимуть зміни яскравості в діагональному напрямку. Одним із найпривабливіших аспектів цієї операції є її простота; ядро невелике й містить лише цілі числа. Проте зі швидкістю сучасних комп'ютерів ця перевага незначна, а оператор Робертса сильно страждає на чутливість до шуму.

Для виявлення контурів за допомогою оператора Робертса, спершу потрібно згорнути первинне зображення з наступними двома ядрами (15.3).

+1	0
0	-1

0	+1
-1	0

Рис. 15.3. Ядра для оператора Робертса

Згортка за першим ядром дає нам горизонтальні похідні $G_x(x,y)$, а згортка за другим — вертикальні похідні $G_y(x,y)$. Далі за формулою (15.2) визначаємо загальний градієнт і будуємо зображення, де інтенсивністю пікселя є значення його градієнта (15.4). На рис. 15.4 значення градієнтів подано з множником 5, щоб підсилити контрастність зображення.

Рис. 15.4. Оригінальне зображення і результат оператора Робертса

15.2.2. Оператор Собеля

Оператор Собеля (Sobel operator), який іноді називають оператором Собеля–Фельдмана (Sobel–Feldman operator) та фільтром Собеля, використовують в обробці зображень та комп'ютерному баченні, зокрема в алгоритмах виявляння контурів, де він створює зображення, яке виділяє контури. Його названо на честь Ірвіна Собеля та Гері Фельдмана, колег зі Стенфордської лабораторії штучного інтелекту (SAIL). Собель та Фельдман запропонували ідею «ізотропного оператора градієнта зображення 3×3» на виступі в SAIL у 1968 році.

На відміну від оператора Робертся, цей оператор використовує два ядра 3×3 (15.5), які згортають з первинним зображенням, щоб обчислювати наближення похідних — однієї для горизонтальних змін, та однієї для вертикальних [27].

+1	0	-1
+2	0	-2
+1	0	-1

+1	+2	+1
0	0	0
-1	-2	-1

Рис. 15.5. Ядра для оператора Собеля

Згортка за першим ядром дає нам горизонтальні похідні $G_x(x,y)$, а згортка за другим — вертикальні похідні $G_y(x,y)$. Далі за формулою (15.2) визначаємо загальний градієнт і будуємо зображення, де інтенсивністю пікселя є значення його градієнта (15.6).

Рис. 15.6. Оригінальне зображення і результат оператора Собеля

Оператор Собеля–Фельдмана пропонує доволі неточне наближення градієнта зображення, але все ще має достатню якість для практичного використання в багатьох застосуваннях. Точніше, він використовує значення яскравості лише в області 3×3 навколо кожної точки зображення, щоби наблизити відповідний градієнт зображення, й використовує лише цілі значення для коефіцієнтів, які зважують яскравості зображення, щоб отримати наближення градієнта.

Щоб покращити точність оператора Собеля, були запропоновані деякі його альтернативні модифікації: оператори Шарра, Каялі, Гаста, Круна.

15.2.3. Оператор Прюітт

Оператор Прюітт (Prewitt operator) використовує підхід, що є подібним до оператора Собеля, але значення середніх коефіцієнтів ядра згортки відрізняються (рис. 15.7). Оператор Прюітт розробила Джудіт Прюітт у 1970 році.

+1	0	-1
+1	0	-1
+1	0	-1

+1	+1	+1
0	0	0
-1	-1	-1

Рис. 15.7. Ядра для оператора Прюітт

Оператор Прюітт простіший з точки зору розрахунків, ніж оператор Собеля, але і результат наближення значень похідних в кожній точці зображення дає грубіший (рис. 15.8).

Рис. 15.8. Оригінальне зображення і результат оператора Прюітт

15.2.4. Оператор Лапласа

Відмінною рисою оператора Лапласа від всіх попередніх методів є те, що він використовує суму других похідних, а не перших. Таким чином це дозволяє порахувати суму вертикальних і горизонтальних похідних за один прохід згортки ядром фільтру Лапласа. Існують декілька варіантів ядра фільтра Лапласа, в залежності від того чи враховуються діагональні сусідні пікселі під час визначення градієнта (рис. 15.9).

0	1	0
1	-4	1
0	1	0

1	1	1
1	-8	1
1	1	1

0	-1	0
-1	4	-1
0	-1	0

-1	-1	-1
-1	8	-1
-1	-1	-1

Рис. 15.9. Ядра оператора Лапласа

Сума всіх коефіцієнтів ядра оператора Лапласа завжди дорівнює 0. Щоб зменшити залежність від шумів, зазвичай перед застосуванням оператора Лапласа застосовуються фільтр згладжування Гаусса.

Порівнянню застосування оператора Лапласа та оператора Собеля для одного і того ж зображення можна побачити на рис. 15.10. Цей приклад наочно демонструє, що оператор Лапласа потребує лише одного проходу згортки, щоб отримати сумарний градієнт зображення, тоді як оператор Собеля потребує два проходи — для розрахунку горизонтальної та вертикальної похідних.

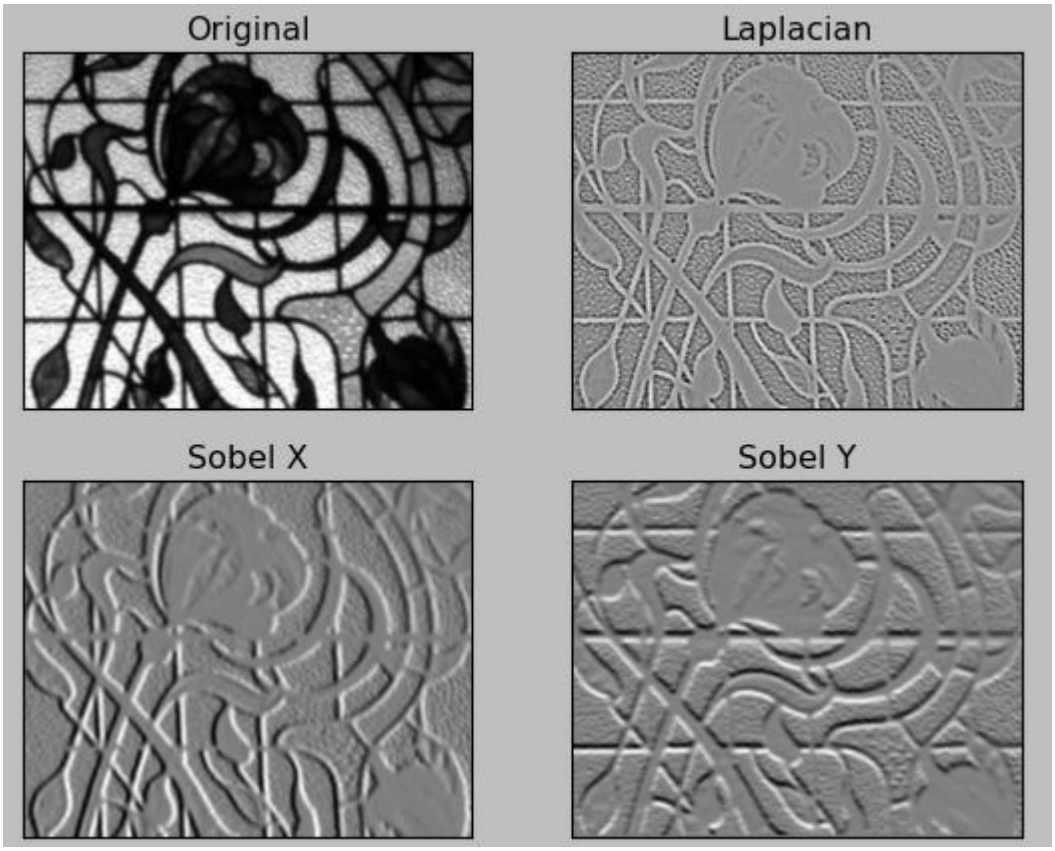


Рис. 15.10. Порівняння операторів Лапласа і Собеля

Недоліком оператора Лапласа є те, що результат не містить інформації про напрямок градієнта в кожному пікселі зображення, на відміну від попередніх операторів, що використовують перші похідні за напрямками.

15.3. Алгоритм Кенні для виявлення контурів

Алгоритм Кенні (Canny edge detector) — це оператор виявляння контурів, який використовує багатоетапний алгоритм для виявляння широкого спектру контурів на зображеннях. Його розробив Джон Кенні 1986 року [28].

Процес алгоритму виявляння контурів Кенні складається з п'яти етапів.

1. Застосувати фільтр Гаусса, щоб згладити зображення задля усунення шуму.
2. Знайти градієнти яскравості зображення.
3. Застосувати порогування величини градієнта або відрізне пригнічування нижньої межі, щоб позбутися паразитної реакції на виявляння контурів.
4. Застосувати подвійний поріг для визначення потенційних контурів.
5. Простежити контури за допомогою гістерезису: завершити виявляння контурів, пригнітивши всі інші контури, слабкі та не пов'язані з сильними.

15.3.1. Застосування фільтра Гаусса

Оскільки на всі результати виявляння контурів легко впливає шум на зображенні, важливо відфільтрувати цей шум, щоби запобігти спричиненому ним помилковому виявлянню. Щоби згладити зображення, із ним згортають ядро

фільтра Гаусса. Цей крок дещо згладить зображення, щоби зменшити вплив явного шуму на виявляч контурів. Ми вже ознайомилися з фільтром Гаусса в підрозділі 14.3. На цьому кроці алгоритму Кенні доцільно застосувати фільтр Гаусса розміром 5x5. Його цілочисельна версія для σ≈1 подана на рис. 15.11.

$$\frac{1}{159}\begin{bmatrix} 2 & 4 & 5 & 4 & 2 \\ 4 & 9 & 12 & 9 & 4 \\ 5 & 12 & 15 & 12 & 5 \\ 4 & 9 & 12 & 9 & 4 \\ 2 & 4 & 5 & 4 & 2 \end{bmatrix}$$

Рис. 15.11. Ядро фільтра Гаусса 5x5 для σ≈1

Важливо розуміти, що вибір розміру гауссового ядра впливатиме на продуктивність виявляча. Що більший розмір, то менша чутливість виявляча до шуму. Крім того, зі збільшенням розміру ядра гауссового фільтра трохи збільшуватиметься похибка розташування у виявлянні контурів. Розмір 5×5 добрий для більшості випадків, але це також різнитиметься залежно від конкретних ситуацій.

Застосувавши фільтр Гаусса, ми отримаємо наступний результат (рис. 15.12).

Рис. 15.12. Оригінальне зображення (ліворуч) та розмите фільтром Гаусса

15.3.2. Знаходження градієнта яскравості зображення

На цьому етапі визначається інтенсивність та напрямок країв, за допомогою обчислення градієнта зображення за допомогою операторів виявлення контурів.

Контури відповідають зміні інтенсивності пікселів. Найпростіший спосіб виявити це — застосувати фільтри, які виділяють цю зміну інтенсивності в обох напрямках: горизонтальному (x) та вертикальному (y). Можна застосовувати будь-який з розглянутих раніше операторів, що використовують першу похідну за кожним напрямком. Наприклад, оператор Собеля (рис. 15.5). За результатами розрахунків, ми отримаємо значення градієнтів для кожного пікселя, а також їхні напрямки. Кут напрямку контуру округлюється до одного з чотирьох кутів, що подають вертикаль, горизонталь та дві діагоналі (0°, 45°, 90° та 135°). Зображення з градієнтами є результатом цього етапу (рис. 15.13), а кути напряму контуру будуть використані на наступному.

Рис. 15.13. Зображення після фільтра Гаусса (ліворуч) та градієнти (праворуч)

Результат майже такий, як очікувалося, але ми бачимо, що деякі ребра товсті, а інші – тонкі. Етап порогування величини допоможе нам зменшити товсті контури.

Більше того, рівень інтенсивності градієнта знаходиться в діапазоні від 0 до 255, що не є однорідним. Краї кінцевого результату повинні мати однакову інтенсивність (тобто білий піксель = 255).

15.3.3. Порогування величини градієнта

Мінімумне відрізне пригнічування величин градієнта, або порогування нижньої межі, — це одна з методик витончування контурів.

Відрізне пригнічування нижньої межі застосовують для пошуку місць із найрізкішою зміною значення яскравості. Алгоритм для кожного пікселя в градієнтному зображенні:

- порівняти вираженість контуру в поточному пікселі з вираженістю контуру в пікселях у додатному та від'ємному напрямках градієнта;
- якщо вираженість контуру в поточному пікселі найбільша порівняно з іншими пікселями в масці з тим же напрямком (наприклад, піксель, що вказує у напрямку y, порівнюватиметься з пікселями над і під ним на вертикальній осі), то значення буде збережено, інакше значення буде пригнічено.

Розглянемо як це працює на декількох прикладах.

Червоний прямокутник у верхньому лівому куті зображення (рис. 15.14) це піксель градієнтної інтенсивності, що обробляється. Відповідний напрямок градієнта показаний помаранчевою стрілкою з кутом -π радіан.

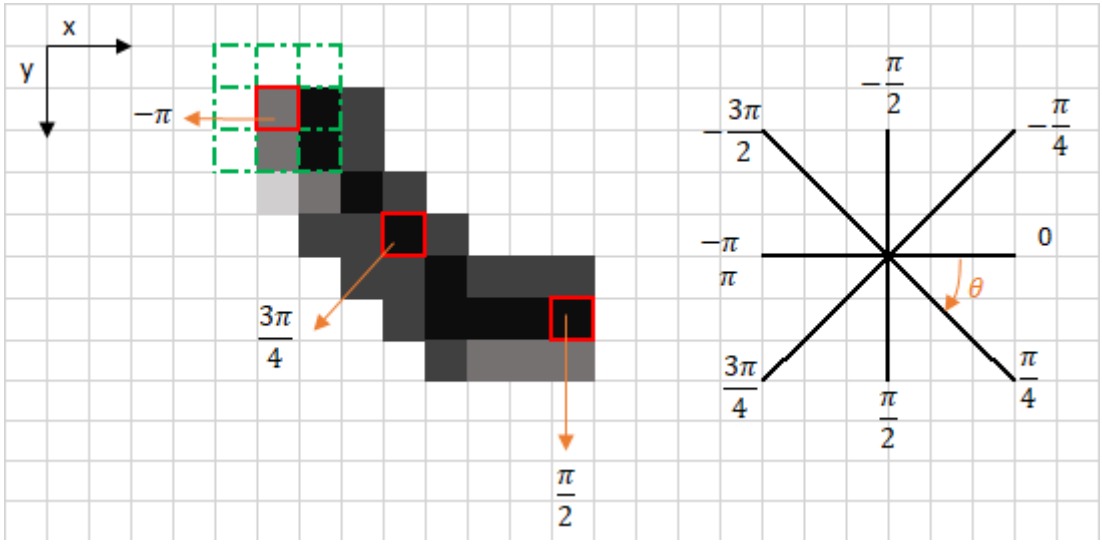


Рис. 15.14. Фрагмент контуру з напрямками градієнтів

Розглянемо верхній лівий піксель, що позначений червоним квадратом, та його сусідів (рис. 15.15).

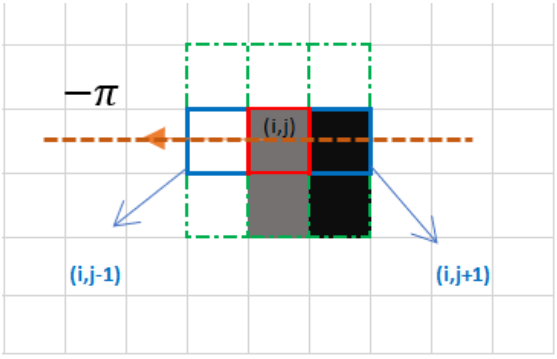


Рис. 15.15. Піксель контуру та його сусіди

Напрямок градієнта – це помаранчева пунктирна лінія (горизонтально зліва направо). Мета алгоритму – перевірити, чи є пікселі в одному напрямку більш чи менш інтенсивними, ніж ті, що обробляються. У прикладі піксель (`i,j`) обробляється, а пікселі в одному з ним напрямку позначені синім (`i,j-1`) та (`i,j+1`). Якщо один із цих двох пікселів інтенсивніший за той, що обробляється, то зберігається лише інтенсивніший. Піксель (`i,j-1`) здається інтенсивнішим, оскільки він білий (значення 255). Отже, значення інтенсивності поточного пікселя (`i,j`) встановлюється в `0`. Якщо в напрямку градієнта немає пікселів з інтенсивнішими значеннями, то зберігається значення поточного пікселя.

Тепер розглянемо інший приклад — піксель з кутом градієнта 3π/4 (рис. 15.16).

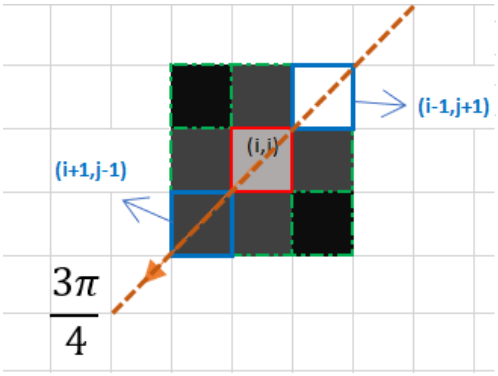


Рис. 15.16. Піксель контуру та його сусіди

Очевидно, що найінтенсивнішим пікселем у цьому напрямку є піксель (`i-1, j+1`). Отже, значення інтенсивності поточного пікселя (`i,j`) встановлюється в `0`.

Після обробки всіх пікселів ми отримаємо наступне зображення (рис. 15.17).

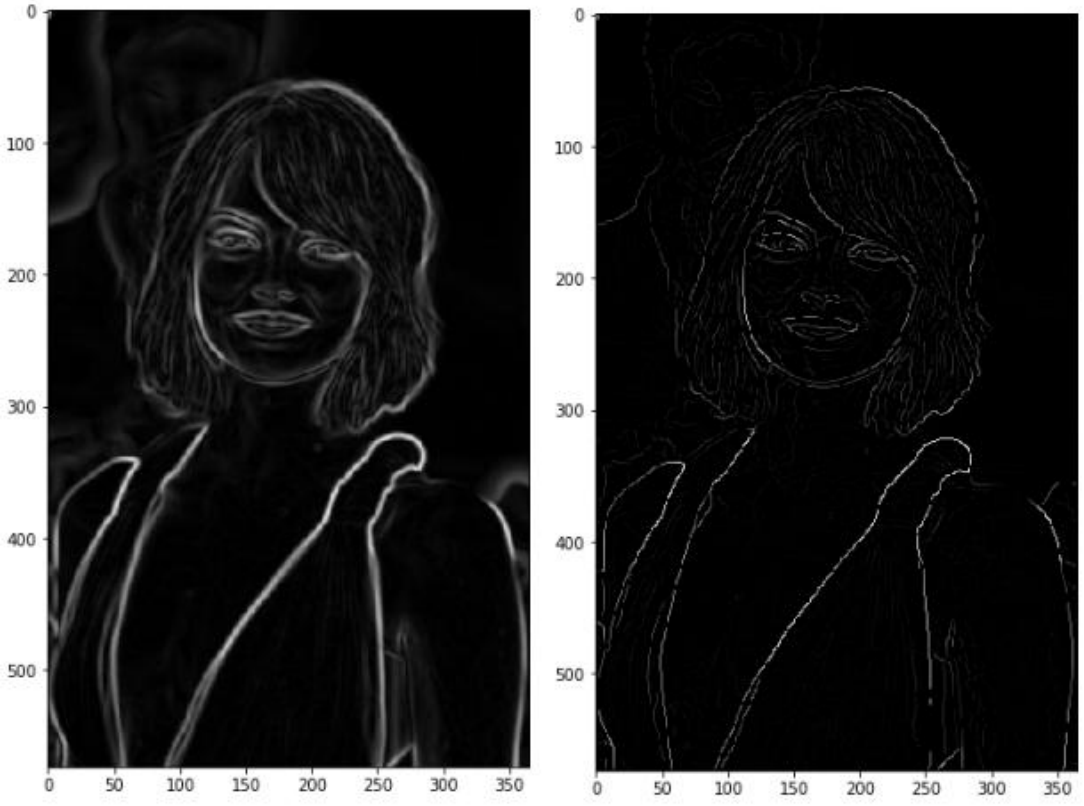


Рис. 15.17. Зображення до та після пригнічування нижньої межі

15.3.4. Подвійний поріг

Після застосування пригнічування нижньої межі решта пікселів контурів забезпечують точніше подання справжніх контурів на зображенні. Проте лишаються деякі контурні пікселі, спричинені шумом та мінливістю кольору. Щоб врахувати ці помилкові реакції, важливо відфільтрувати контурні пікселі зі слабким значенням градієнта, та зберегти з високим. Цього досягають обранням верхнього та нижнього порогових значень. Якщо значення градієнта контурного пікселя вище за верхнє порогове значення, його позначують як *сильний* контурний піксель. Якщо значення градієнта контурного пікселя менше за верхнє порогове значення та більше за нижнє порогове значення, його позначують як *слабкий* контурний піксель. Якщо значення градієнта контурного пікселя менше ніж нижнє порогове значення, його буде пригнічено. Ці два порогові значення визначають емпірично, і їх визначення залежатиме від вмісту заданого вхідного зображення.

Результатом роботи цього етапу є зображення лише з двома типами пікселів: сильними та слабкими (рис. 15.18).

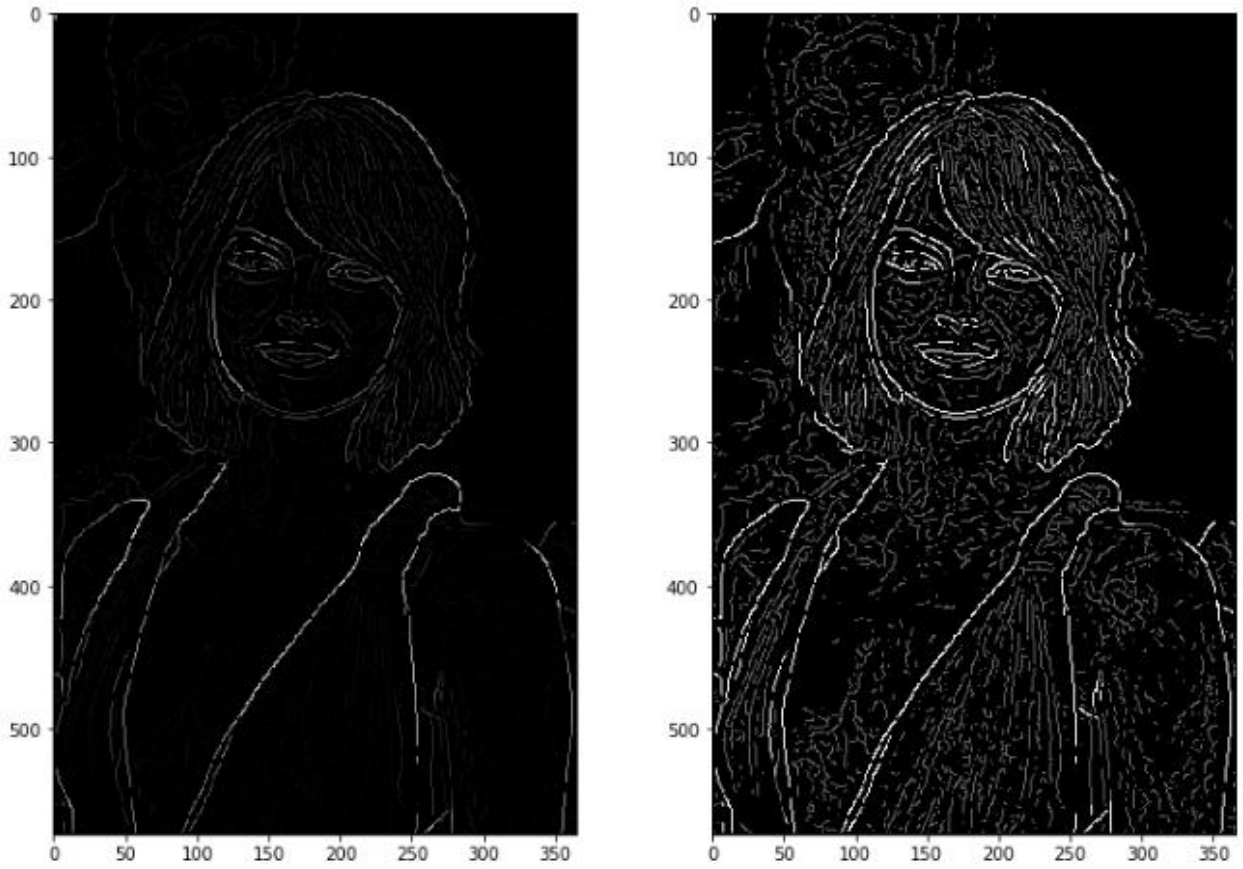


Рис. 15.18. Зображення до та після подвійного порогування (слабкі пікселі подані сірим кольором, а сильні — білим)

15.3.5. Простежування контурів за допомогою гістерезису

Сильні пікселі однозначно будуть присутніми в контурах об'єктів зображення, а от зі слабкими пікселями потрібно додатково розібратися. Вони можуть бути просто елементами шуму на зображенні — тоді їх слід видалити, але також вони можуть бути частинами контурів реальних об'єктів — в такому разі їх необхідно перетворити на сильні пікселі контуру.

Власне алгоритм гістерезису доволі простий (рис. 15.19):

- перевірити вісім сусідніх пікселів для поточного;
- якщо серед сусідів немає сильного пікселя, то інтенсивність пікселя робимо 0;
- якщо серед сусідів є хоча б один сильний піксель, тоді поточний теж стає сильним (інтенсивність 255).

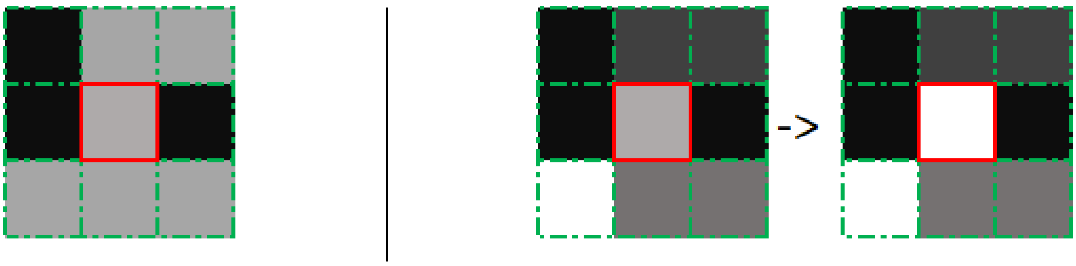


Рис. 15.19. Алгоритм гістерезису

Результат роботи гістерезису показаний на рис. 15.20.

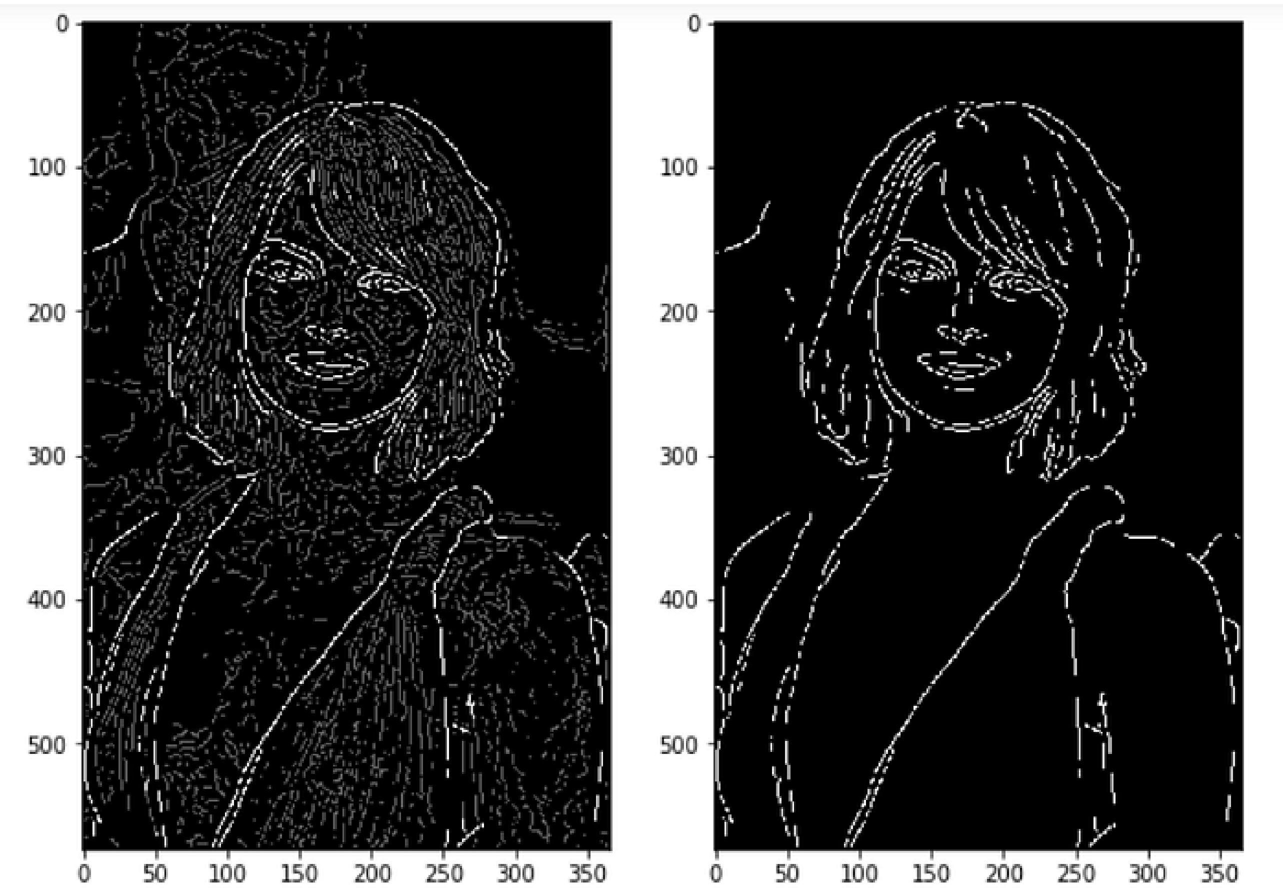


Рис. 15.20. Зображення до та після гістерезису

Отримане зображення складається лише з двох кольорів: білого та чорного. Чорний колір позначає фон зображення, а білий — контури об'єктів.

Запитання для самоконтролю

1. Яким чином можна збільшити чіткість зображення? Для яких задач це може бути актуальним?
2. Яка спільна риса в більшості методів виявлення контурів об'єктів на зображеннях?
3. Який розмір ядра фільтра, що використовується в операторах Робертса, Собеля і Прюітт?
4. Що вирізняю оператор Лапласа на фоні інших операторів виявлення контурів?
5. Скільки етапів має алгоритм Кенні для виявлення контурів?

ВИКОРИСТАНІ ДЖЕРЕЛА

1. Giloi W. Interactive computer graphics: data structures, algorithms, languages. Englewood Cliffs, N.J : Prentice-Hall, 1978. 354 p.
2. Rogers D. F. Mathematical elements for computer graphics. 2nd ed. New York : McGraw, 1989. 611 p.
3. Пічугін М., Канкін І., Воротніков В. Комп'ютерна графіка : навчальний посібник. Київ : Центр навчальної літератури, 2019. 346 с.
4. https://uk.wikipedia.org/wiki/Лінійна_інтерполяція
5. https://uk.wikipedia.org/wiki/Многочлен_Лагранжа
6. https://uk.wikipedia.org/wiki/Многочлен_Ньютона
7. https://uk.wikipedia.org/wiki/Кубічний_сплайн
8. https://gregorygundersen.com/blog/2019/11/15/lagrange-polynomial
9. https://pages.hmc.edu/ruye/MachineLearning/lectures/ch7/node4.html
10. https://atsushisakai.github.io/PythonRobotics/modules/path_planning/cubic_spline/cubic_spline.html
11. https://uk.wikipedia.org/wiki/Крива_Безьє
12. https://www.tutorialspoint.com/computer_graphics/computer_graphics_cyrus_beck_line_clipping_algorithm.htm
13. https://www.geeksforgeeks.org/line-clipping-set-2-cyrus-beck-algorithm/
14. https://www.scribd.com/presentation/390168709/LineClipping-19-sep-18-2
15. https://uk.wikipedia.org/wiki/Алгоритми_побудови_відрізка
16. https://uk.wikipedia.org/wiki/Алгоритм_DDA-лінії
17. https://www.geeksforgeeks.org/computer-graphics/dda-line-generation-algorithm-computer-graphics/
18. https://uk.wikipedia.org/wiki/Алгоритм_Брезенхейма
19. https://www.mathros.net.ua/rasteryzacija-kola-vykorystovujuchy-algorytm-brezenhema.html
20. https://www.geeksforgeeks.org/dsa/bresenhams-line-generation-algorithm/
21. https://www.geeksforgeeks.org/c/bresenhams-circle-drawing-algorithm/
22. https://uk.wikipedia.org/wiki/Медіанна_фільтрація
23. https://uk.wikipedia.org/wiki/Гауссове_згладжування
24. https://uk.wikipedia.org/wiki/Виявляння_контурів
25. https://uk.wikipedia.org/wiki/Оператор_Робертса
26. https://uk.wikipedia.org/wiki/Оператор_Собеля
27. https://uk.wikipedia.org/wiki/Оператор_Прюітт
28. https://uk.wikipedia.org/wiki/Алгоритм_Кенні
29. https://medium.com/data-science/canny-edge-detection-step-by-step-in-python-computer-vision-b49c3a2d8123

Г

Д

З

І

К

Л

М

Н

О

П

Р

С

Т

Ф

Ц

Я

Навчальне видання

Бородавка Євгеній Володимирович

Бугров Анатолій Анатолійович

Волох Богдан Юрійович

ГРАФІЧНІ ІНФОРМАЦІЙНІ ТЕХНОЛОГІЇ ТА ОБЧИСЛЮВАЛЬНА ГЕОМЕТРІЯ

Підручник