МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ БУДІВНИЦТВА І АРХІТЕКТУРИ

В.В. ДЕМЧЕНКО

Є.В. БОРОДАВКА

ГЕОМЕТРИЧНЕ МОДЕЛЮВАННЯ І КОМП'ЮТЕРНА ГРАФІКА

РЕКОМЕНДОВАНО МІНІСТЕРСТВОМ ОСВІТИ І НАУКИ УКРАЇНИ ЯК НАВЧАЛЬНИЙ ПОСІБНИК ДЛЯ СТУДЕНТІВ ВИЩИХ НАВЧАЛЬНИХ ЗАКЛАДІВ

Київ 2010

УДК 004.92

ББК 32.973

Д19

Рецензенти: К.О. Сазонов, доктор технічних наук, професор, завідувач кафедри дизайну інтер'єру та меблів Київського національного університету технології та дизайну
А.А. Лященко, доктор технічних наук, професор
В.Б. Задоров, кандидат технічних наук, професор, завідувач кафедри інформаційних технологій Київського національного університету будівництва і архітектури

Рекомендовано Міністерством освіти і науки України як навчальний посібник для студентів вищих навчальних закладів (лист Міністерства освіти і науки України № 14/18.2 – 1485 від 12. 07. 2010).

Демченко В.В., Бородавка Є.В.

Д19 Геометричне моделювання і комп'ютерна графіка. – К.: КНУБА, 2010. – 288 с.

ISBN 966-627-061-7

В посібнику розглянуто основні алгоритми двовимірної та тривимірної графіки, що використовуються в системах комп'ютерного проектування та моделювання. Навчальний посібник містить необхідні теоретичні і довідкові матеріали про графічну бібліотеку OpenGL, а також методичні рекомендації щодо її практичного застосування. Розглянуто приклади роботи з бібліотекою в середовищах програмування Delphi та C++ Builder, основні прийоми оптимізації застосунків.

Призначений для студентів, що навчаються за напрямом підготовки 6.050101 «Комп'ютерні науки».

УДК 004.92

ББК 32.973

ISBN 966-627-061-7

ЗМІСТ

Вступ

В даному посібнику розглянуті основні поняття і алгоритми геометричного моделювання та комп'ютерної графіки. Посібник логічно поділений на дві частини: в першій розглядається теоретичний матеріал, а в другій – практичне застосування розглянутих теоретичних підходів і алгоритмів за допомогою графічної бібліотеки OpenGL.

1. Основні поняття геометричного моделювання

1.1. Різновиди обробки графічної інформації

Геометричне моделювання (**ГМ**) – займається вивченням та обробкою об'єктів, які мають геометричні властивості або їх характеристики можуть мати геометричну інтерпретацію.

Обробка комп'ютерної графічної інформації має багато різновидів та практичних застосувань. Зазвичай цю область обробки інформації прийнято розділяти на три напрямки.

1. **Комп'ютерна графіка** – займається відтворення зображень у тих випадках, коли вхідною є інформація неграфічного характеру. Складність програм, а також обчислювальні витрати, що необхідні для отримання відповідних візуальних зображень, істотно залежать від характеру конкретної задачі. Прикладами подібної візуалізації можуть слугувати: побудова графіків функцій чи експериментальних даних, виведення інформації на екран та синтез сцен у комп'ютерних іграх.
2. **Обробка зображень** – пов'язана з вирішенням таких задач, у яких і вхідні і вихідні дані є зображеннями. Один з прикладів – системи передачі зображень. Розробники подібних систем вирішують проблеми усунення шуму та стиснення даних. Знімки, отримані з перетримкою чи недотримкою, а також розмиті знімки, можуть бути покращені за допомогою методів підвищення контрасту.
3. **Розпізнавання зображень** – застосування методів, які дозволяють отримати деякий опис зображення, поданого на вхід системи, або віднести зображення до певного класу. Розпізнавання образів, це задача, певним чином, зворотна до задачі комп'ютерної графіки. Процедура розпізнавання застосовується до деякого зображення і забезпечує перетворення його у деякий абстрактний опис: набір чисел, ланцюг символів чи граф. Наступна обробка подібного опису дозволяє віднести вхідне зображення до одного з декількох класів.

До одного з класів задач, що є предметом спільного інтересу цих трьох напрямків, відноситься отримання внутрішнього подання зображень у комп'ютерній техніці (структури даних, збереження та пошук, стиснення).

Початок розвитку комп'ютерної графіки відноситься до 1963 року (поява перших графічних пристроїв). У 60-70 ті роки вартість дисплею

оцінювалась у 10-25 тисяч доларів. В цей час були розроблені основні алгоритми комп'ютерної графіки.

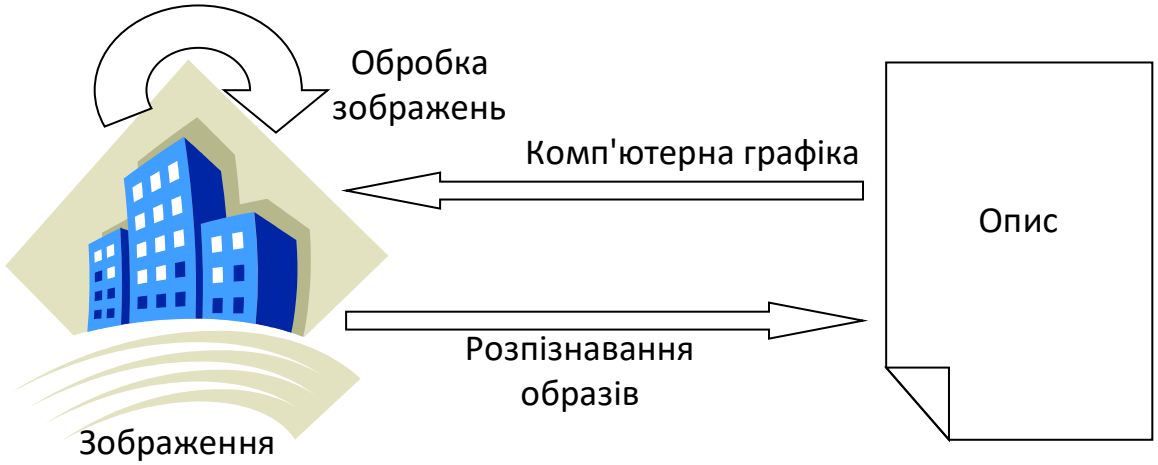


Рис. 1.1. Схема, що ілюструє взаємозв'язок комп'ютерної графіки, обробки зображень і розпізнавання образів

1.2. Концептуальна модель систем геометричного моделювання

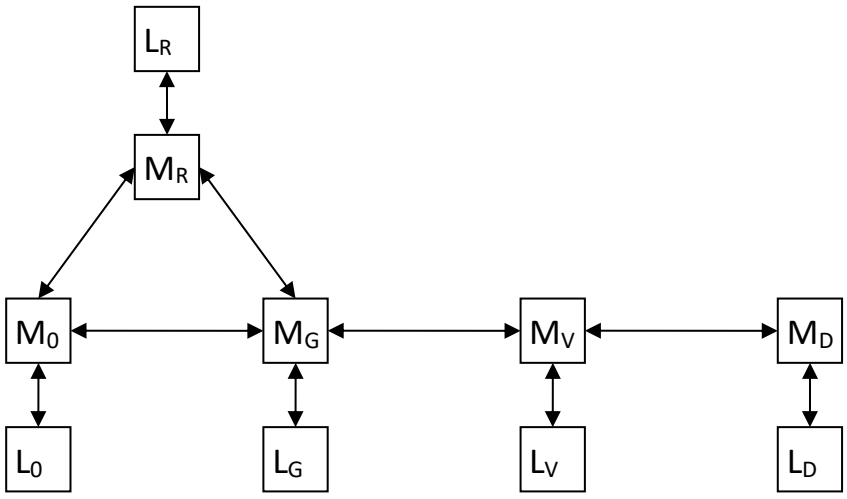


Рисунок 1.2. Концептуальна модель систем геометричного моделювання

$L_D=UL_i$ – мова діалогу складається з об'єднання всіх мов.

M_0 – інфологічна, цифрова модель об'єкта (ЦМО). Відображає структуру, склад та зв'язки об'єкта проектування на відповідному рівні декомпозиції об'єкта і задач.

M_G – геометрична модель об'єкта.

M_R – графічна модель. Складається з графічних елементів, включає сукупність типових елементів відображення не розгорнутих до примітивів.

Примітиви – найменші елементарні об'єкти. До *геометричних* примітивів відносяться: точка, лінія, площина, призми, сфери. *Графічні* примітиви орієнтовані на відображення геометричних примітивів: відрізки, ломані лінії, коло, еліпс, прямокутник, тощо. Кожен з цих примітивів характеризується параметрами та атрибутами візуалізації (колір, товщина, тип лінії).

M_V – лінійна графічна модель в форматах віртуального графічного пристрою у вигляді сукупності примітивів графічного виведення. Примітиви структуровані у відповідності до типових елементів таким чином, що кожному екземпляру типових елементів відповідає один графічний сегмент (графічний блок, як сукупність примітивів).

M_D – це модель формату конкретного графічного пристрою, модель зображення в командах і координатах конкретного графічного пристрою.

ЦМО створюють не тільки з метою відображення або документування об'єкта, а й для проведення розрахунків, наприклад об'єкт у вигляді якоїсь оболонки буде перетворений в геометричну модель поверхні, над геометричною моделлю буде проведене власне перетворення у вигляді відображення поверхні кінцевими елементами для проведення розрахунків за методом скінченних елементів. Відповідно модель M_0 одержить від геометричної моделі на цьому кроці розрахункову схему.

При відображенні об'єкта або результату моделювання деякі елементи можуть безпосередньо передаватись на рівень моделі M_R, але окремі елементи повинні пройти відповідні геометричні перетворення для створення відповідної графічної моделі відображення. Наприклад, у випадку зі сферою спочатку потрібно провести тріангуляцію, виконати перспективне зображення такої моделі, розфарбувати трикутники в залежності від освітлення.

Всі перетворення між моделями є однорідними за структурою. Кожен структурний елемент M_0 має свій унікальний ідентифікатор. Відповідні геометричні і графічні елементи теж повинні мати такі ж ідентифікатори, це дає змогу реалізувати зворотній зв'язок при інтерактивному введенні інформації від графічної моделі до моделі об'єкта. Будь-які структурні зміни в графічній моделі повинні бути перенесені на відповідні зміни у моделі M_0.

Вказані моделі можуть створюватись і фіксуватись у окремих форматах, передаватись у інші системи на відповідних рівнях і підлягати документуванню.

Ці моделі можуть створюватись як тимчасові масиви, які існують лише у процесі роботи програми, але концептуально вони існують у будь-якій системі, як і відповідні перетворення.

1.3. Координатні системи геометричного моделювання

В геометричному моделюванні та комп'ютерній графіці об'єкти моделювання та графічного введення-виведення описуються в різноманітних системах координат.

Система координат – це сукупність правил, які ставлять у відповідність кожному об'єкту (*точці*) визначений набір чисел (*координат*).

Координати – це значення деяких характеристик об'єктів, які однозначно визначають положення об'єкта в просторі. Число координат, яке необхідне для визначення певної точки в системі координат, називається *розмірністю простору*.

В геометричному моделюванні та комп'ютерній графіці застосовуються класичні системи координат: афінна, декартова, полярна, циліндрична, сферична, система однорідних координат тощо.

Окрім класичних, в комп'ютерній графіці визначаються ще дві основні системи координат: *світова* та *приладова*, які відображають процеси перетворення координат під час введення та виведення даних за допомогою графічних пристроїв.

Світова система координат – тривимірна або двовимірна прямокутна декартова система координат, в якій описуються об'єкти певного світу (реальні або абстрактні об'єкти та процеси), що моделюються на комп'ютерах. В геометричному моделюванні всі об'єкти мають *світові координати* (*World Coordinates*).

Світові координати (**WC**) – координати того світу, геометрична модель об'єктів або явищ якого створюється.

Світові системи координат приладово-незалежні і мають одиниці виміру, що притаманні тим світам, для опису яких вони призначені (міліметри – для об'єктів проектування, ампери, вольти – для електротехнічних процесів тощо). Значення світових координат належать множині дійсних чисел.

Приладова система координат – двовимірна прямокутна декартова система координат, в якій формується і виводиться зображення об'єктів на екран дисплею (плотер) або в яку перетворюються зображення з графічних пристроїв введення даних в комп'ютер (сканерів, планшетів

тощо). Відповідно координати в цій системі називаються *приладовими* (*Device Coordinates*).

Приладові координати (**DC**) – координати конкретного пристрою, на який виводиться графічне зображення.

Приладові системи координат визначаються множиною цілих чисел в одиницях растру пристроїв введення-виведення. Для графічних дисплеїв це *піксел* в області виведення 1280×1024, 1280×720, 1600×900, 1920×1200 тощо.

Піксел – це найменша частка зображення на екрані дисплею, яка характеризується положенням і кольором. Координати інших пристроїв можуть мати метричну характеристику, або крок дискретної сітки робочого поля з характеристикою *dpi* (точок на дюйм).

Моделювання об'єктів та формування зображення на комп'ютерах виконується в світовій системі координат, а виведення зображення на екран супроводжується перетворенням координат із світової в приладову систему координат. Це перетворення називається операцією *кадрування*.

Кадрування – операція відображення вікна світової системи координат на область індикації. Ця операція здійснюється шляхом застосування до всіх точок моделі (зображення) відповідного перетворення кадрування. Під час відображення здійснюються одночасні перетворення вікна та сформованого в ньому зображення.

Область світової системи координат, що підлягає виведенню на екран дисплею, визначається прямокутником, який називається *вікном в світових* координатах (*Window WC*) і задається відповідно своїми мінімаксними координатами. Частина екрану, в яку будуть відображатись об'єкти вікна, є областю виведення або індикації (*Viewport*) і задається відповідними приладовими координатами.

Вікно та область індикації мають прямокутну форму, а тому формули, які здійснюють перетворення кадрування, мають простий вигляд. Порядок виконання кадрування наступний.

1. Вираховуємо розміри світового вікна в напрямку осей X та Y:

$$\Delta X^{WC} = \left|X_{MAX}^{WC} - X_0^{WC}\right|;\ \Delta Y^{WC} = \left|Y_{MAX}^{WC} - Y_0^{WC}\right|.$$

2. Визначаємо центральну точку світового вікна:

$$X_C^{WC} = X_0^{WC} + \frac{\Delta X^{WC}}{2}; Y_C^{WC} = Y_0^{WC} + \frac{\Delta Y^{WC}}{2}.$$

3. Вираховуємо розміри області індикації в напрямку осей X та Y:

$$\Delta X^{DC} = \left|X_{MAX}^{DC} - X_0^{DC}\right|;\ \Delta Y^{DC} = \left|Y_{MAX}^{DC} - Y_0^{DC}\right|.$$

4. Визначаємо центральну точку області індикації:

$$X_C^{DC} = X_0^{DC} + \frac{\Delta X^{DC}}{2};\ Y_C^{DC} = Y_0^{DC} + \frac{\Delta Y^{DC}}{2}.$$

5. Перетворення кадрування складаються з двох елементарних перетворень: масштабування (приладової нормалізації) та переміщення до суміщення центрів вікна та області індикації. Коефіцієнти масштабування по осях X та Y є дійсними числами та визначають кількість дискретів приладової системи координат на одну одиницю вікна світової системи координат:

$$k_X = \frac{\Delta X^{DC}}{\Delta X^{WC}};\ k_Y = \frac{\Delta Y^{DC}}{\Delta Y^{WC}}. \quad (1.1)$$

6. Тоді формули перетворення кадрування для кожної точки зображення будуть наступними:

$$\begin{aligned} X_i^{DC} &= \left(X_i^{WC} - X_C^{WC}\right) \cdot k_X + X_C^{DC}; \\ Y_i^{DC} &= \left(Y_i^{WC} - Y_C^{WC}\right) \cdot k_Y + Y_C^{DC}. \end{aligned} \quad (1.2)$$

7. Формули (1.1) і (1.2) описують операцію кадрування з різними перетвореннями масштабування в напрямку X та Y, що є прийнятним для виведення графічних залежностей (графіків), але не прийнятне для виведення креслень, оскільки необхідно забезпечити рівний масштаб для приладової нормалізації світових координат. Для цього необхідно визначити єдиний коефіцієнт масштабування:

$$k = \min(k_X, k_Y).$$

8. Також враховуючи, що сучасні дисплеї мають систему координат, де початок знаходиться у лівому верхньому куті, а вісь Y спрямована зверху вниз, отримаємо фінальні формули для перетворення кадрування:

$$\begin{aligned} X_i^{DC} &= \left(X_i^{WC} - X_C^{WC}\right) \cdot k + X_C^{DC}; \\ Y_i^{DC} &= Y_C^{DC} - \left(Y_i^{WC} - Y_C^{WC}\right) \cdot k. \end{aligned} \quad (1.3)$$

Розрахунки коефіцієнту та координат центральних точок світового вікна і області індикації необхідно виконувати лише один раз при зміні розмірів цих вікон. А формули (1.3) необхідно застосовувати до кожної точки зображення в світових координатах, щоб перетворити її координати в приладові.

В залежності від процесу обробки об'єкта також розрізняють абсолютні, відносні, користувацькі та нормовані координати.

Абсолютні координати – визначають положення об'єкта відносно початку системи координат.

Відносні координати – задають положення однієї точки відносно іншої.

Користувацькі координати – це безрозмірні координати, які задав користувач (практично співпадають із світовими).

Нормовані координати (**NDC**) – це координати абстрактного екрану, які безрозмірні та змінюються від 0 до 1. Використовуються для того, щоб створити спільну картину для різних світових координат з подальшим перетворенням в інші приладові координати. Вони є проміжними координатами в процесі формування комплексної картинки з різних світів на різних приладах (рис. 1.3).

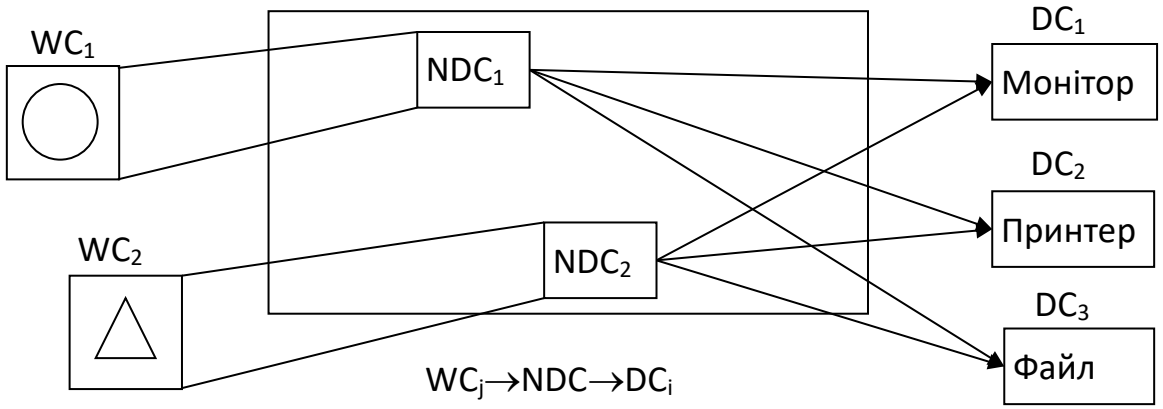


Рис. 1.3. Схема перетворення світових координат у приладові

Застосування NDC дозволяє для N світів і M приладів виконувати N+M перетворень (замість N×M), оскільки потрібно виконати N перетворень $WC_j \rightarrow NDC$ та M перетворень $NDC \rightarrow DC_i$.

Окрім розглянутих світової та приладової систем координат в геометричному моделюванні та комп'ютерній графіці варто звернути увагу також на іншу пару систем координат – *глобальну* та *локальну*.

Глобальна система координат – це фактично світова система координат для об'єкта, що розглядається.

Локальна система координат – це система координат якогось елемента декомпозиції об'єкта на складові.

В геометричному моделюванні та графіці існують дві симетричні задачі: *декомпозиції* та *композиції*.

Декомпозиція – це розділення складного об'єкта на складові. Схема декомпозиції ієрархічна, а атомарні елементи кожного рівня відповідають рівню предметної області задач.

Під час декомпозиції виокремлюють типові елементи або геометричні об'єкти, кожен з таких елементів має локальну систему координат і одну особливу точку, що носить назву *базової*. Відносно базової точки проводиться включення об'єкта в об'єкти більш високого рівня.

Приклад. Задана двовимірна декартова система координат, в якій побудований фасад стіни з вікном (рис. 1.4).

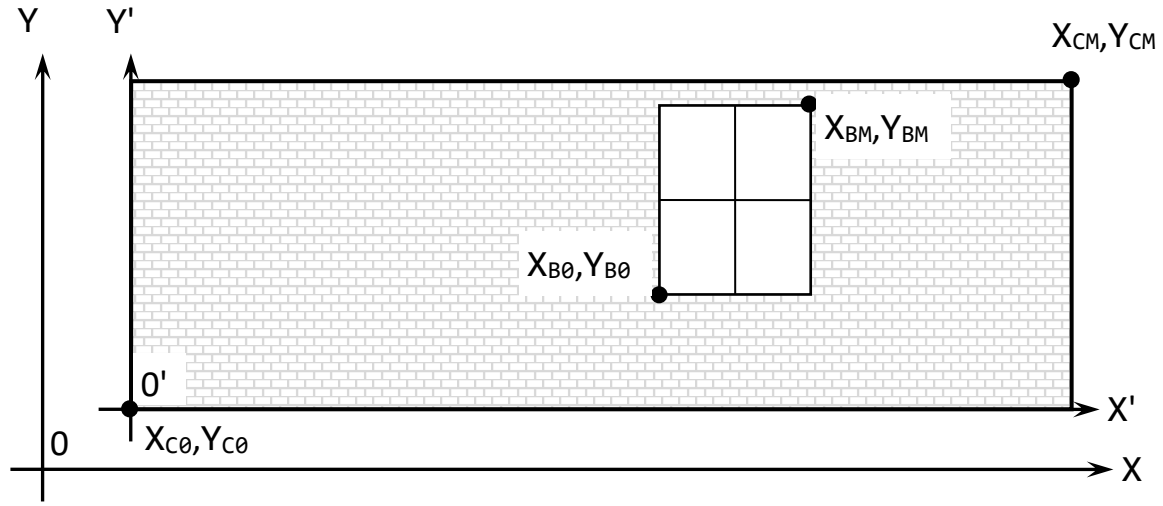


Рис. 1.4. Фасад стіни з вікном

І стіна і вікно мають геометричну форму прямокутник. Для його опису достатньо двох пар координат, що описують початок і кінець головної діагоналі. Для стіни це координати ($X_{С0},Y_{С0},X_{СМ},Y_{СМ}$), а для вікна – ($X_{В0},Y_{В0},X_{ВМ},Y_{ВМ}$). Наведені координати описують обидва об'єкти (стіну і вікно) в системі координат з початком X0Y яка є *глобальною*. Але вікно не може «існувати» без стіни, тому доцільно було б описати вікно в системі координат стіни, як батьківського об'єкта. Тому задамо нову систему координат X'0'Y' з початком в точці ($X_{С0},Y_{С0}$), що співпадає з нижньою лівою точкою стіни. Така система координат є *локальною* для об'єктів стіна і вікно. Стіна в локальній системі матиме координати ($0,0,X'_{СМ},Y'_{СМ}$), а вікно – ($X'_{В0},Y'_{В0},X'_{ВМ},Y'_{ВМ}$). В даному прикладі локальна система координат зміщена відносно глобальної на деякі значення по осям X та Y. Тому перетворення координат із глобальної системи в локальну і навпаки є досить простим. В загальному випадку локальна система координат може довільно розташовуватися відносно глобальної і перехід між ними описується загальним *афінним* перетворенням.

1.4. Однорідні координати

Однорідні координати – це математичний механізм, пов'язаний з визначенням положення точок у просторі. Звичний апарат декартових координат, не підходить для вирішення деяких важливих задач в силу наступних міркувань:

- в декартових координатах неможливо описати нескінченно віддалену точку. А багато які математичні та геометричні концепції значно спрощуються, якщо в них використовується поняття нескінченності. Наприклад, «нескінченно віддалене джерело світла»;
- з точки зору алгебраїчних операцій, декартові координати не дозволяють розрізнити точки і вектори в просторі;
- неможливо використовувати уніфікований механізм роботи з матрицями для опису перетворень точок. За допомогою матриць 3×3 можна описати обертання і масштабування, однак описати зсув не можна;
- аналогічно, декартові координати не дозволяють використовувати матричну форму запису для завдання перспективного перетворення (проекції) точок.

Для вирішення цих проблем використовуються однорідні координати.

1.4.1. Визначення однорідних координат

Існують різні способи визначення однорідних координат. Ми будемо виходити із задачі уніфікованого подання координат точок у просторі, що включає нескінченно віддалені точки.

Нехай задані дійсні числа, a і w. Розглянемо їх відношення a/w. Зафіксуємо значення a, і будемо варіювати значення w. При зменшенні w, значення a/w збільшуватиметься. Зауважимо, що коли w прямує до нуля, то a/w прямує до нескінченності. Таким чином, щоб включити в розгляд поняття нескінченності, для подання значення v використовується пара чисел (a,w), таких, що v=a/w. Якщо $w\neq0$, значення v точно дорівнює a/w. В протилежному випадку v=a/0, тобто рівне нескінченності.

Таким чином, координати двовимірної точки v=(x,y) можна подати через координати (wx,wy,w). При w=1 ці координати описують точку із скінченними координатами (x,y), а при w=0 – точку, нескінченно віддалену у напрямку (x,y). Як було сказано вище, звичайним поданням через декартові координати (x,y) це зробити неможливо.

Розглянемо двовимірну площину, деяку точку (x,y) на ній і задану функцію f(x,y). Якщо замінити x та y на x/w і y/w, то вираз f(x,y)=0 заміниться на f(x/w,y/w)=0. Якщо f(x,y) – многочлен, то його множення на w^n (n – ступінь многочлена) прибере всі знаменники.

Наприклад, нехай задана пряма:

$$Ax + By + C = 0.$$

Заміна x та y на x/w і y/w дає:

$$A\left(\frac{x}{w}\right) + B\left(\frac{y}{w}\right) + C = 0.$$

Помноживши на w, отримуємо:

$$Ax + By + Cw = 0. \qquad (1.4)$$

Інший приклад. Нехай заданий многочлен другого порядку:

$$Ax^2 + Bxy + Cy^2 + 2Dx + 2Ey + F = 0.$$

Після заміни x та y на x/w і y/w відповідно, та множення на w^2, отримуємо:

$$Ax^2 + Bxy + Cy^2 + 2Dxw + 2Eyw + Fw^2 = 0. \qquad (1.5)$$

Якщо уважно подивитися на многочлени (1.4) і (1.5), можна помітити, що степені всіх членів рівні. У випадку многочлена першого порядку, це степінь 1, тоді як для многочлена другого порядку, всі члени (тобто x^2, xy, y^2, xw, yw і w^2) мають степінь 2. Отже, для даного многочлена n-го порядку, після введення координати w всі члени будуть мати степінь n. Такі многочлени називаються *однорідними*, а координати (x,y,w) називаються однорідними координатами (*Homogenous Coordinates*).

Наведені міркування залишаються вірними і у випадку тривимірного простору. Координати (x,y,z) замінюються на (x/w,y/w,z/w) і після множення на w у відповідній степені n дають однорідний многочлен.

Однорідні координати вимагають три компоненти для подання точки на площині (і чотири компоненти для точки в просторі). Які ж однорідні координати відповідають точці з координатами (x,y)? Легко побачити, що це буде (x,y,1), тобто w приймається рівною 1.

У загальному випадку, це перетворення не однозначне. Однорідні координати точки (x,y) рівні (xw,yw,w) для будь-якого ненульового w. Аналогічно в тривимірному просторі: точці (x,y,z) відповідають координати (xw,yw,zw,w). У той же час, перетворення з однорідних координат в евклідові однозначно: точці (x,y,w) відповідає точка (x/w,y/w).

Наведемо більш формальне визначення: *однорідними координатами* точки $P=(x_1,...,x_n)$, $P \in R^n$ називаються координати $P_{hom}=(wx_1,wx_2,...,wx_n,w)$, $P_{hom} \in R^{n+1}$, причому хоча б один елемент повинен бути відмінний від нуля.

Насправді, безліч векторів P_{hom} за певних додаткових операціях утворюють так званий *проективний простір*, що має важливе значення з точки зору комп'ютерної графіки. Ми на цьому зупинятися не будемо. Важливо запам'ятати наступне: *перетворення з однорідних координат в евклідові однозначне, а перетворення з евклідових координат в однорідні – ні*.

1.4.2. Геометрична інтерпретація однорідних координат

Можна подати просту геометричну інтерпретацію однорідних координат на площині. Нехай дані однорідні координати (x,y,w) точки на площині XOY. Поставимо їй у відповідність точку в тривимірному евклідовому просторі з координатами x, y і w по осях X, Y та W відповідно. Пряма, що з'єднує цю точку з початком координат, перетинає площину w=1 в точці (x/w,y/w,1) (рис. 1.5).

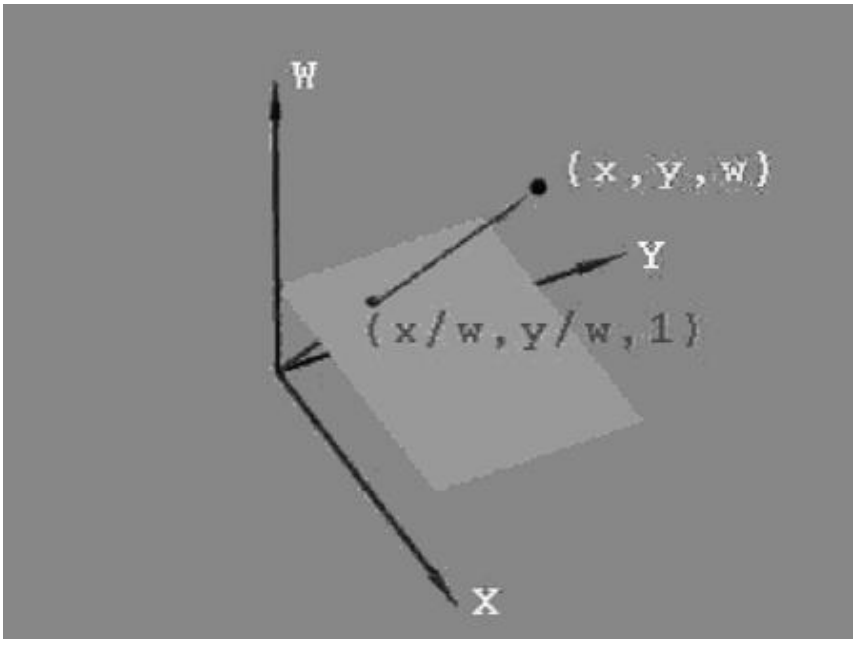


Рис. 1.5. Геометрична інтерпретація однорідних координат

Таким чином, перетворення з однорідних координат в евклідові еквівалентно проекції точки на площину w=1 вздовж лінії, що з'єднує точку з початком координат.

З рисунка 1.5 також видно, що якщо перетворення з однорідних координат в евклідові однозначне, то зворотне перетворення – ні, тому

що всі точки на лінії, що з'єднує точку (x,y,w) і початок координат будуть проектуватися в точку (x/w,y/w).

1.4.3. Точка в нескінченності

Як було сказано раніше, за допомогою однорідних координат можна легко описувати нескінченність. Розглянемо точку з однорідними координатами (x,y,w). Їй відповідає точка з евклідовими координатами (x/w,y/w). Зафіксуємо x та y і спрямуємо w до нуля. Точка (x/w,y/w) буде віддалятися все далі й далі в нескінченність в напрямку (x,y). Коли w стане нулем, (x/w,y/w) йде в нескінченність. Отже, однорідні координати (x,y,0) – ідеальна точка (*Ideal Point*) або, по-іншому, точка в нескінченності (*Point at Infinity*) за напрямом (x,y). Аналогічно для тривимірного простору: точка (x,y,z,0) – точка в нескінченності за напрямом (x,y,z).

Наприклад, в OpenGL для визначення положення джерела світла використовуються однорідні координати. За їх допомогою можна визначити як точкове джерело світла (w=1), так і паралельне джерело світла (w=0).

1.4.4. Різниця між точками та векторами

Нехай є система координат (O,[i],[j],[k]). Щоб подати заданий вектор V, необхідно знайти три числа (v_1,v_2,v_3), причому такі, що виконується співвідношення:

$$V = v_1 \cdot \bar{\imath} + v_2 \cdot \bar{\jmath} + v_3 \cdot \bar{k}.$$

Це означає, що вектор V задає напрямок відносно векторів базису [i],[j],[k].

З іншого боку, щоб подати точку P, можна розглядати її місце розташування як зміщення на певний вектор (p_1, p_2, p_3) відносно початку координат. Отже, положення точки P можна записати наступним чином:

$$P = O + p_1 \cdot \bar{\imath} + p_2 \cdot \bar{\jmath} + p_3 \cdot \bar{k}.$$

Таким чином, для опису положення точки трьох параметрів недостатньо.

Використовуючи однорідні координати, ці вирази можна записати як v=(v_1,v_2,v_3,0) і P=(p_1,p_2,p_3,1). В даному випадку 1 або 0 показують, чи бере початок координат участь в обчисленнях. Дійсно, це узгоджується з уявленням про те, що вектор – це точка, нескінченно віддалена у певному напрямку (тобто з w=0 в однорідних координатах).

Зауважимо, що координатні операції з векторами зберігають однорідну форму запису координат:

- різниця двох точок (x,y,z,1) та (d,e,f,1) дорівнює (x-d,y-e,z-f,0), тобто як і очікувалося, є вектором;
- сума точки (x,y,z,1) і вектора (d,e,f,0) дорівнює іншій точці (x+d,y+e,z+f,1);
- два вектора можна складати, в результаті виходить вектор (d,e,f,0)+(m,n,r,0)=(d+m,e+n,f+r,0);
- має сенс масштабування вектора 3(d,e,f,0)=(3d,3e,3f,0);
- має сенс створення будь-якої лінійної комбінації векторів.

Запитання для самоконтролю

1. Чим займається геометричне моделювання?
2. Які різновиди обробки інформації ви знаєте?
3. З яких елементів складається концептуальна модель систем геометричного моделювання?
4. Назвіть основні типи координатних систем, що використовуються в комп'ютерній графіці.
5. Для чого призначена операція кадрування?
6. Що таке нормовані координати і для чого вони використовуються?
7. Для чого використовуються однорідні координати? Який їх зв'язок з евклідовими?

2. АФІННІ ПЕРЕТВОРЕННЯ КООРДИНАТ

2.1. ВИЗНАЧЕННЯ ТА КЛАСИФІКАЦІЯ АФІННИХ ПЕРЕТВОРЕНЬ

Афінне перетворення – (лат. *affinis*, «пов'язаний з») відображення f: $R^n \to R^n$, яке можна записати у вигляді:

$$f(x) = M \cdot x + v,$$

де M – оборотна (не вироджена) матриця та $v \in R^n$.

Інакше кажучи, відображення називається афінним, якщо його можна отримати наступним способом:

- обрати «новий» базис простору з «новим» початком координат v;
- координатам x кожної точки простору поставити у відповідність нові координати $f(x)$, які мають те саме положення в просторі відносно «нової» системи координат, яке координати x мали в «старій».

Розрізняють наступні типи афінних перетворень:

- *власне* – афінне перетворення під час якого система координат зберігає знак;
- *невласне* – афінне перетворення під час якого система координат змінює знак;
- *еквіафінне* – афінне перетворення, що зберігає площу;
- *центроафінне* — афінне перетворення, що зберігає початок координат.

Властивості афінних перетворень:

- під час афінного перетворення пряма переходить в пряму;
- якщо розмірність простору $n \geq 2$, то будь-яке перетворення простору (тобто бієкція простору на себе), яке переводить прямі в прямі, є афінним;
- окремим випадком афінних перетворень є ізометрії та перетворення подібності;
- афінні перетворення утворюють групу відносно композиції.

2.2. АФІННІ ПЕРЕТВОРЕННЯ НА ПЛОЩИНІ

Припустимо, що на площині задана прямолінійна система координат. Тоді кожній точці M ставиться у відповідність впорядкована пара чисел (x,y) – її координат. Задаючи на площині ще одну систему координат, ми ставимо у відповідність тій же точці M іншу пару чисел – (x',y'). Перехід

від однієї прямолінійної координатної системи на площині до іншої описується наступними співвідношеннями:

$$x' = \alpha x + \beta y + \lambda;\ y' = \gamma x + \delta y + \mu, \qquad (2.1)$$

де $\alpha,\beta,\gamma,\delta,\lambda,\mu$ – довільні числа, що пов'язані нерівністю:

$$\begin{vmatrix} \alpha & \beta \\ \gamma & \delta \end{vmatrix} \neq 0.$$

Формули (2.1) можна розглядати двояко: або зберігається точка і змінюється координатна система – в такому випадку довільна точка M залишиться тою ж, змінюються лише її координати (x,y) $\rightarrow$ (x',y'); або змінюється точка і зберігається координатна система – в цьому випадку формули задають відображення, що переводить довільну точку M(x,y) в точку M'(x',y'), координати якої визначені в тій же системі координат. Ми в подальшому розглядати перший варіант інтерпретації наведених формул.

Формули (2.1) описують афінне перетворення на площині довільного типу. Будь-яке афінне перетворення може бути описане як *суперпозиція* чотирьох часткових випадків афінних перетворень. Тепер розглянемо ці чотири часткових випадки афінних перетворень на площині.

2.2.1. Переміщення

Переміщення – найпростіше афінне перетворення, що описує зсув нової системи координат відносно старої (рис. 2.1).

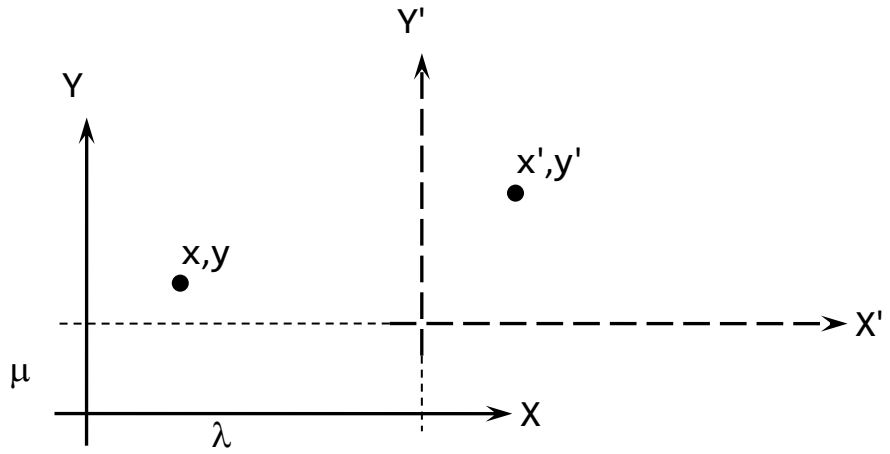


Рис. 2.1. Переміщення системи координат

Перетворення координат точки в старій системі координат в координати точки в новій системі координат відбувається за наступними формулами:

$$x' = x + \lambda;\ y' = y + \mu. \qquad (2.2)$$

2.2.2. Масштабування

Масштабування – це афінне перетворення під час якого відбувається розтягнення або стиснення вздовж координатних осей. Описується наступними формулами:

$$x' = \alpha x;\ y' = \delta y, \tag{2.3}$$

де $\alpha>0$, $\delta>0$.

Розтягнення вздовж координатних осей відбувається за умови, що $\alpha>1$, $\delta>1$, а стиснення за умови – $\alpha<1$, $\delta<1$.

Формули (2.3) можна об'єднати і записати в матричній формі:

$$[x' \quad y'] = [x \quad y] \cdot \begin{bmatrix} \alpha & 0 \\ 0 & \delta \end{bmatrix}.$$

2.2.3. Поворот

Поворот – це афінне перетворення під час якого система координат здійснює оберт навколо нерухомої осі Z, що напрямлена до спостерігача (рис. 2.2).

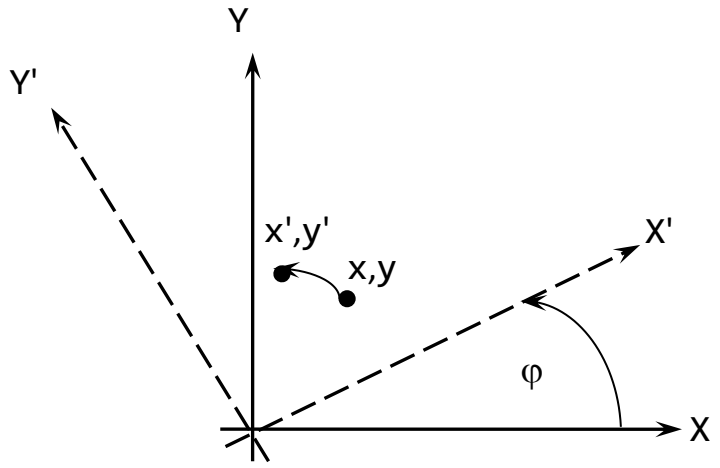


Рис. 2.2. Поворот системи координат

Поворот навколо початкової точки на кут φ описується наступними формулами:

$$x' = x\cos\varphi - y\sin\varphi\ ;\ y' = x\sin\varphi + y\cos\varphi. \tag{2.4}$$

Формули (2.4) можна об'єднати і записати в матричній формі:

$$[x' \quad y'] = [x \quad y] \cdot \begin{bmatrix} \cos\varphi & \sin\varphi \\ -\sin\varphi & \cos\varphi \end{bmatrix}.$$

2.2.4. Відображення

Відображення – невласне афінне перетворення під час якого одна з координат змінює свій знак на протилежний. На площині може бути відображення відносно осі X або відносно осі Y (рис. 2.3).

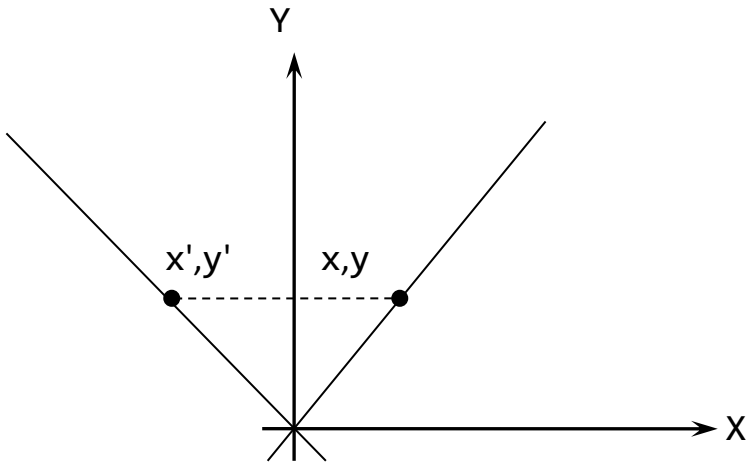


Рис. 2.3. Відображення відносно осі Y

Відображення відносно осі Y задається наступними формулами:

$$x' = -x;\ y' = y. \qquad (2.5)$$

Відповідно для відображення відносно осі X необхідно змінити знак координати y. Матрична форма запису для відображення дуже проста, тому тут не наводиться.

2.3. Матрична форма афінних перетворень в однорідних координатах

Як вже зазначалося, будь-яке афінне перетворення можна подати у вигляді суперпозиції чотирьох найпростіших перетворень, що наведені вище. В комп'ютерній графіці використовується матрична форма запису цих перетворень. Тоді суперпозиція знаходиться як результат послідовного перемноження матриць, що описують відповідні афінні перетворення.

В розглянутих вище трьох найпростіших перетвореннях окрім формул також розглянута їх матрична форма подання. Але для найпростішого афінного перетворення *переміщення* записати матричну форму неможливо. Для усунення цього недоліку використовуються однорідні координати: замість матриць 2×2 використовуються матриці 3×3 і вектори (x,y,1).

Матриця переміщення (*Translation*) [T] тоді буде мати наступний вигляд:

$$[T] = \begin{bmatrix} 1 & 0 & \lambda \\ 0 & 1 & \mu \\ 0 & 0 & 1 \end{bmatrix}.$$

Відповідно, переміщення точки (x,y,1) на вектор (λ,μ) розраховується наступним чином:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & \lambda \\ 0 & 1 & \mu \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} x + \lambda \\ y + \mu \\ 1 \end{bmatrix},$$

що повністю відповідає формулам (2.2).

Матриця масштабування (*Scaling*) [S] буде мати наступний вигляд:

$$[S] = \begin{bmatrix} \alpha & 0 & 0 \\ 0 & \delta & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

Відповідно розтягнення (стиснення) точки (x,y,1) на коефіцієнти (α,δ) розраховується наступним чином:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} \alpha & 0 & 0 \\ 0 & \delta & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} \alpha x \\ \delta y \\ 1 \end{bmatrix},$$

що повністю відповідає формулам (2.3).

Матриця повороту (*Rotation*) [R] буде мати наступний вигляд:

$$[R] = \begin{bmatrix} \cos\varphi & -\sin\varphi & 0 \\ \sin\varphi & \cos\varphi & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

Відповідно поворот точки (x,y,1) навколо початку системи координат на кут φ розраховується наступним чином:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\varphi & -\sin\varphi & 0 \\ \sin\varphi & \cos\varphi & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} x\cos\varphi - y\sin\varphi \\ x\sin\varphi + y\cos\varphi \\ 1 \end{bmatrix},$$

що повністю відповідає формулам (2.4).

Матриця відображення (*Reflection*) [M] відносно осей X та Y відповідно буде мати наступний вигляд:

$$[M_X] = \begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix};\ [M_Y] = \begin{bmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

Відповідно відображення точки (x,y,1) відносно осі Y розраховується наступним чином:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} -x \\ y \\ 1 \end{bmatrix},$$

що повністю відповідає формулам (2.5).

Об'єднавши елементи всіх матриць в одну, можна записати *матрицю афінного перетворення*:

$$\begin{bmatrix} \alpha & \beta & \lambda \\ \gamma & \delta & \mu \\ 0 & 0 & 1 \end{bmatrix}.$$

Тоді нові координати точки (x,y,1) після довільного афінного перетворення розраховуються наступним чином:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} \alpha & \beta & \lambda \\ \gamma & \delta & \mu \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} \alpha x + \beta y + \lambda \\ \gamma x + \delta y + \mu \\ 1 \end{bmatrix},$$

що повністю відповідає загальним формулам (2.1).

Елементи матриці довільного афінного перетворення не несуть в собі явно вираженого геометричного змісту. Тому, щоб реалізувати те чи інше перетворення, необхідно розбити його на елементарні складові, побудувати їх матриці і перемножити їх в потрібному порядку.

В цьому підрозділі всі матриці подавалися в транспонованому вигляді і множилися на вертикальний вектор значень. Але при складному афінному перетворенні простіше їх перемножувати в нормальному вигляді, а тоді використовувати множення горизонтального вектора на результуючу матрицю. Порядок наступний – спочатку знаходимо результуючу матрицю шляхом їх послідовного перемноження між собою, а потім вектор множимо на матрицю. Надалі ми будемо використовувати нормальні матриці і горизонтальні вектори.

Приклад. Побудувати матрицю повороту навколо точки A(a,b) на кут φ.

1. Спочатку переміщуємо точку A в початок системи координат:

$$[T_{-A}] = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -a & -b & 1 \end{bmatrix}.$$

2. Робимо поворот на кут φ:

$$[R_\varphi] = \begin{bmatrix} \cos\varphi & \sin\varphi & 0 \\ -\sin\varphi & \cos\varphi & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

3. Тепер повертаємо точку A в початкове положення:

$$[T_A] = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ a & b & 1 \end{bmatrix}.$$

4. Множимо матриці в тому ж порядку, як вони записані: $[T_{-A}][R_\varphi][T_A]$. В результаті отримаємо матрицю:

$$\begin{bmatrix} \cos\varphi & \sin\varphi & 0 \\ -\sin\varphi & \cos\varphi & 0 \\ a - a\cos\varphi + b\sin\varphi & b - a\sin\varphi - b\cos\varphi & 1 \end{bmatrix}.$$

Дуже важливо пам'ятати про порядок перемноження матриць. Адже переміщення, а потім поворот дають один результат, тоді як поворот, а потім переміщення – зовсім інший.

2.4. Афінні перетворення в просторі

Афінні перетворення в просторі аналогічні до перетворень на площині, лише додається координата z. Відповідно всі чотири типи афінних перетворень, що розглянуті для площини справедливі і для простору. Вони описуються матрицями в однорідних координатах розмірністю 4×4. Коротко розглянемо ці матриці.

Матриця переміщення (*Translation*) [T] буде мати наступний вигляд:

$$[T] = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ \lambda & \mu & \nu & 1 \end{bmatrix}.$$

Матриця масштабування (*Scaling*) [S] буде мати наступний вигляд:

$$[S] = \begin{bmatrix} \alpha & 0 & 0 & 0 \\ 0 & \beta & 0 & 0 \\ 0 & 0 & \gamma & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Матриця повороту (*Rotation*) [R] навколо осей Z, X та Y відповідно буде мати наступний вигляд:

$$[R_Z] = \begin{bmatrix} \cos\chi & \sin\chi & 0 & 0 \\ -\sin\chi & \cos\chi & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}; [R_X] = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\varphi & \sin\varphi & 0 \\ 0 & -\sin\varphi & \cos\varphi & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}; [R_Y] = \begin{bmatrix} \cos\psi & 0 & -\sin\psi & 0 \\ 0 & 1 & 0 & 0 \\ \sin\psi & 0 & \cos\psi & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Матриця відображення (*Reflection*) [M] відносно площин XOY, YOZ та ZOX відповідно буде мати наступний вигляд:

$$[M_Z] = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}; \; [M_X] = \begin{bmatrix} -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}; \; [M_Y] = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Для того, щоб виконати складне афінне перетворення в просторі потрібно, як і в плоскому випадку, розбити перетворення на складові із простих перетворень. Побудувати для кожного з них матриці, а потім перемножити їх в правильному порядку.

Приклад. Побудувати матрицю повороту на кут φ навколо прямої L, що проходить через точку A(a,b,c) та має направляючий вектор (l,m,n) (рис. 2.4). Можна вважати, що направляючий вектор прямої є одиничним: $l^2+m^2+n^2=1$.

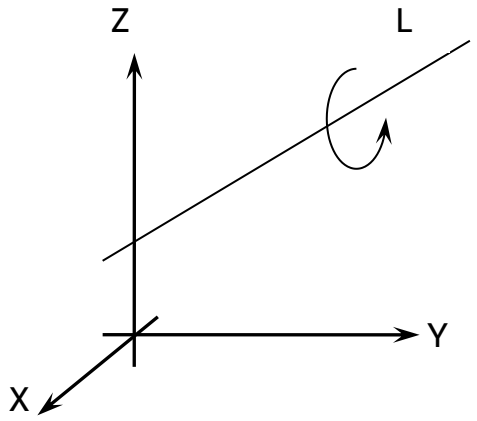


Рис. 2.4. Умова задачі для прикладу

Вирішення даної задачі розбивається на кілька кроків.

1. Будуємо матрицю переміщення на вектор –A(-a,-b,-c):

$$[T] = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -a & -b & -c & 1 \end{bmatrix}.$$

В результаті цього переміщення ми добиваємося того, що пряма L проходить через початок координат.

2. Суміщаємо вісь аплікат (Z) з прямою L двома поворотами навколо осі абсцис (X) та осі ординат (Y).

Перший поворот – навколо осі абсцис на кут ψ. Цей кут необхідно вирахувати. Щоб знайти цей кут, розглянемо ортогональну проекцію L' заданої прямої L на площину X=0 (рис. 2.5).

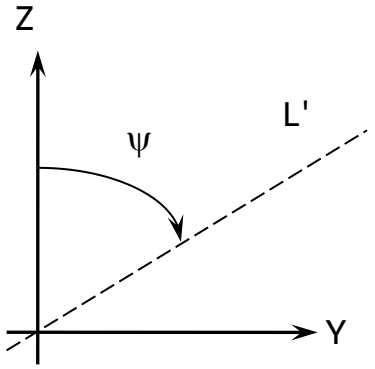


Рис. 2.5. Проекція прямої L на площину X=0

Направляючий вектор прямої L' визначається просто – він дорівнює (0,m,n). Звідси знаходимо:

$$\cos\psi = \frac{n}{d}; \sin\psi = \frac{m}{d}; d = \sqrt{m^2 + n^2}.$$

Тоді матриця повороту навколо осі абсцис матиме наступний вигляд:

$$[R_X] = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \frac{n}{d} & \frac{m}{d} & 0 \\ 0 & -\frac{m}{d} & \frac{n}{d} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Застосувавши перетворення, що описує ця матриця до вектора (l,m,n) отримаємо направляючий вектор прямої L в новій системі координат:

$$[l \quad m \quad n \quad 1] \cdot [R_X] = [l \quad 0 \quad d \quad 1].$$

Другий поворот – навколо осі ординат на кут θ, що визначається співвідношеннями:

$$\cos\theta = l;\ \sin\theta = -d.$$

Тоді відповідна матриця повороту буде мати вигляд:

$$[R_Y] = \begin{bmatrix} l & 0 & d & 0 \\ 0 & 1 & 0 & 0 \\ -d & 0 & l & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

3. Поворот навколо прямої L на заданий кут φ. Оскільки тепер пряма L співпадає з віссю аплікат, то відповідна матриця має наступний вигляд:

$$[R_Z] = \begin{bmatrix} \cos\varphi & \sin\varphi & 0 & 0 \\ -\sin\varphi & \cos\varphi & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

4. Тепер виконуємо зворотні дії в порядку вкладеності. Тобто спочатку виконуємо поворот навколо осі ординат на кут $-\theta$.
5. Потім виконуємо поворот навколо осі абсцис на кут $-\psi$.
6. І останнім робимо переміщення на вектор A(a,b,c).

Перемноживши знайдені матриці в порядку їх побудови, отримаємо:

$$[T] \cdot [R_X] \cdot [R_Y] \cdot [R_Z] \cdot [R_Y]^{-1} \cdot [R_X]^{-1} \cdot [T]^{-1}.$$

Запишемо кінцевий результат, вважаючи для простоти, що вісь обертання L проходить через центральну точку:

$$\begin{bmatrix} l^2 + \cos\varphi\,(1 - l^2) & l(1-\cos\varphi)m + n\sin\varphi & l(1-\cos\varphi)n - m\sin\varphi & 0 \\ l(1-\cos\varphi)m - n\sin\varphi & m^2 + \cos\varphi\,(1 - m^2) & m(1-\cos\varphi)n + l\sin\varphi & 0 \\ l(1-\cos\varphi)n + m\sin\varphi & m(1-\cos\varphi)n - l\sin\varphi & n^2 + \cos\varphi\,(1 - n^2) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Як бачимо з прикладу, знаходження елементів матриці довільного афінного перетворення в просторі доволі трудомістка задача. Особливо, якщо виконується поворот навколо довільної прямої – тоді доводиться вираховувати як мінімум два кути і здійснювати повороти відносно всіх осей координат в певній послідовності. Але задачу повороту навколо довільної прямої можна вирішувати й іншими способами: за допомогою *Ейлерових кутів* або за допомогою *кватерніонів*.

2.5. Ейлерові кути

Ейлерові кути — три кути, за допомогою яких математично описується поворот однієї системи координат відносно іншої в тривимірному просторі.

Здебільшого використовуються для математичного опису обертання абсолютно твердого тіла, під час якого одна система координат – система спостерігача, а інша жорстко зв'язується з тілом.

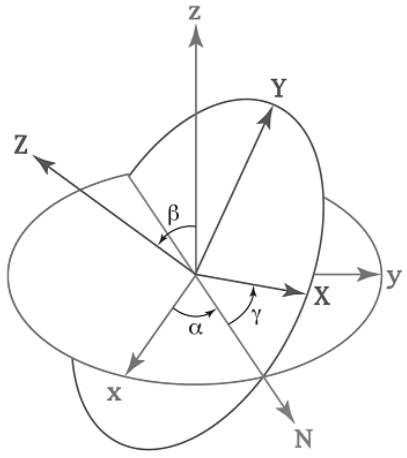


Рис. 2.6. Ейлерові кути

На рис. 2.6. нерухома система координат позначена малими літерами xyz, а рухома система координат – великими XYZ.

Лінія вузлів N – це лінія перетину площин xy та XY.

Назви і значення Ейлерових кутів наступні:

- *кут нутації* β – кут між осями Z та z.
- *кут прецесії* α – кут між віссю x та лінією вузлів N.
- *кут власного обертання* γ – кут між лінією вузлів N та віссю X.

Замість позначень α, β, γ вживаються також ψ, θ, φ.

Кути прецесії та власного обертання змінюються в межах $[0,2\pi]$. Кут нутації в межах $(0,\pi)$. Як бачимо при значенні кута нутації 0 або π Ейлерові кути не визначаються, оскільки здійснюється плоский поворот в площині XOY

Матриця повороту однієї системи координат відносно іншої виражається через добуток матриць послідовних поворотів навколо осей Z, X, Y на відповідні кути Ейлера:

$$[R] = [R_Z(\psi)] \cdot [R_X(\theta)] \cdot [R_Y(\varphi)].$$

Якщо виконати перемноження, то результуюча матриця матиме наступний вигляд:

$$[R] = \begin{bmatrix} \cos\psi\cos\varphi - \sin\psi\sin\varphi\cos\theta & -\sin\psi\cos\varphi\cos\theta - \cos\psi\sin\varphi & \sin\psi\sin\theta & 0 \\ \sin\psi\cos\varphi + \cos\psi\cos\theta\sin\varphi & \cos\psi\cos\varphi\cos\theta - \sin\psi\sin\varphi & -\cos\psi\sin\theta & 0 \\ \sin\theta\sin\varphi & \cos\varphi\sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

При застосуванні Ейлерових кутів достатньо їх вирахувати і скласти по ним матрицю, що наведена вище. Це буде матриця переходу від однієї системи координат до іншої.

2.6. Кватерніони

Для вирішення задачі повороту навколо заданого вектора найкраще підходять *кватерніони*.

Кватерніон – гіперкомплексне число, яке реалізується в 4-вимірному просторі. Вперше описане В. Р. Гамільтоном у 1843 році.

Кватерніон має вигляд q=a+bi+cj+dk, де a,b,c,d – дійсні числа; i,j,k – уявні одиниці. Кватерніон можна також подати у вигляді пари скаляра та 3-вимірного вектора: q=(s,v), s=a, v=(a,b,c).

Якщо в 3-вимірному просторі заданий вектор v=(x,y,z) та кут повороту навколо нього α, то кватерніон можна записати у вигляді:

$$q = \left(\cos\frac{\alpha}{2}, \bar{v}\sin\frac{\alpha}{2}\right). \quad (2.6)$$

Розклавши формулу (2.6) отримаємо наступний вираз:

$$q = \cos\frac{\alpha}{2} + (x \cdot i + y \cdot j + z \cdot k)\sin\frac{\alpha}{2}.$$

Точка в 4-вимірному просторі описується як кватерніон з нульовим скаляром: `p=0+xi+yj+zk`. Тоді поворот будь-якої точки p навколо вектора v=(x,y,z), на кут α можна описати формулою:

$$p' = qpq^{-1},$$

де p' – точка після повороту;

q – кватерніон, що побудований за формулою (2.6);

q^{-1} – спряжений кватерніон: q^{-1}=`a-bi-cj-dk`.

Множення кватерніонів виражається через скалярний та векторний добутки 3-вимірних векторів:

$$q_1 q_2 = (s_1, \overline{v_1})(s_2, \overline{v_2}) = (s_1 s_2 - \overline{v_1} \cdot \overline{v_2},\ s_1\overline{v_2} + s_2\overline{v_1} + \overline{v_1} \times \overline{v_2})$$

Кватерніону (2.6) відповідає наступна матриця повороту:

$$R_{\bar{v}}(\alpha) = \begin{bmatrix} x^2 & xy & xz \\ xy & y^2 & yz \\ xz & yz & z^2 \end{bmatrix} \cdot (1 - \cos\alpha) + \begin{bmatrix} \cos\alpha & z\sin\alpha & -y\sin\alpha \\ -z\sin\alpha & \cos\alpha & x\sin\alpha \\ y\sin\alpha & -x\sin\alpha & \cos\alpha \end{bmatrix}. \quad (2.7)$$

Фактично вираз (2.7) аналогічний матриці, що ми отримали в результаті вирішення задачі з розділу 2.4. Але використання кватерніонів для опису поворотів навколо довільного вектору набагато зручніше і простіше, ніж використання матриць афінних перетворень.

Запитання для самоконтролю

1. Що таке афінне перетворення координат?
2. Які прості афінні перетворення ви знаєте?
3. Що таке суперпозиція афінних перетворень?
4. Що таке Ейлерові кути і для чого вони призначені?
5. Для чого використовуються кватерніони? Коли їх зручно застосовувати?

3. ПРОЕКТИВНІ ПЕРЕТВОРЕННЯ

При візуалізації двовимірних зображень, достатньо задати вікно видимості в світових координатах та область індикації в приладових, куди буде виведено це зображення. В цьому випадку достатньо виконати операцію кадрування. У випадку тривимірного зображення задається не вікно, а об'єм видимості. Після цього необхідно виконати проектування об'єму видимості на область індикації, яку ще називають *картинна площина*. Модель процесу візуалізації наведена на рис. 3.1.

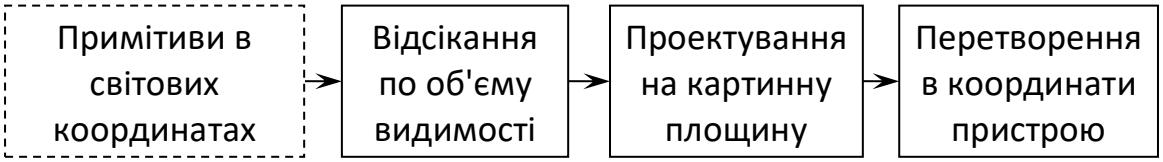


Рис. 3.1. Модель процесу візуалізації

Проектування – відображення точок, заданих в системі координат розмірністю N, в точки в системі з меншою розмірністю. Під час відображення тривимірних зображень на дисплей три виміри відображаються в два.

3.1. ВИДИ ПРОЕКТУВАННЯ

Проектування виконується за допомогою прямолінійних проекторів (променів проектування), що йдуть з центру проекції через кожну точку об'єкта до перетину з картинної поверхнею (поверхнею проекції). Далі розглядаються тільки плоскі проекції, для яких поверхня проекції – площина в тривимірному просторі.

За розташуванням центру проектування відносно площини проекції розрізняють *центральне* та *паралельне* проектування.

Центральне проектування – проектування, під час якого всі прямі виходять з однієї точки – центра власного пучка (рис. 3.2).

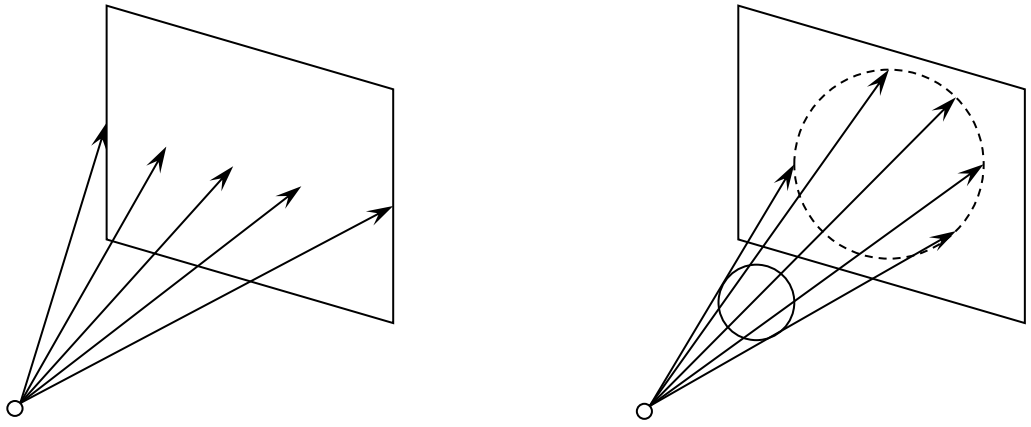


Рис. 3.2. Центральне проектування

Паралельне проектування – проектування, під час якого центр пучка вважається таким, що лежить в нескінченності (рис. 3.3).

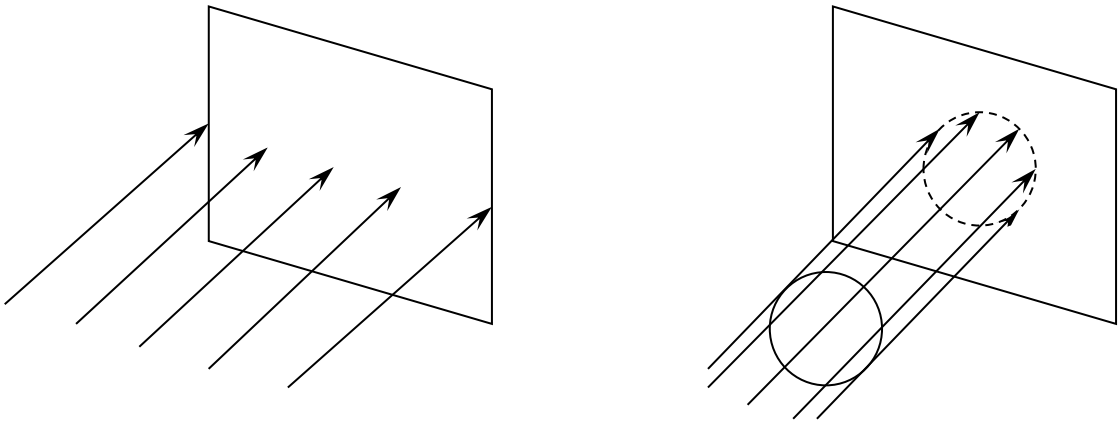


Рис. 3.3. Паралельне проектування

Кожен з цих двох основних класів розбивається на декілька підкласів в залежності від взаємного розташування картинної площини та координатних осей.

3.2. Паралельні проекції

В паралельних проекціях виокремлюють три класи: ортографічна, аксонометрична та косокутня. Дві останні з них ще додатково поділяються на підкласи (рис. 3.4).

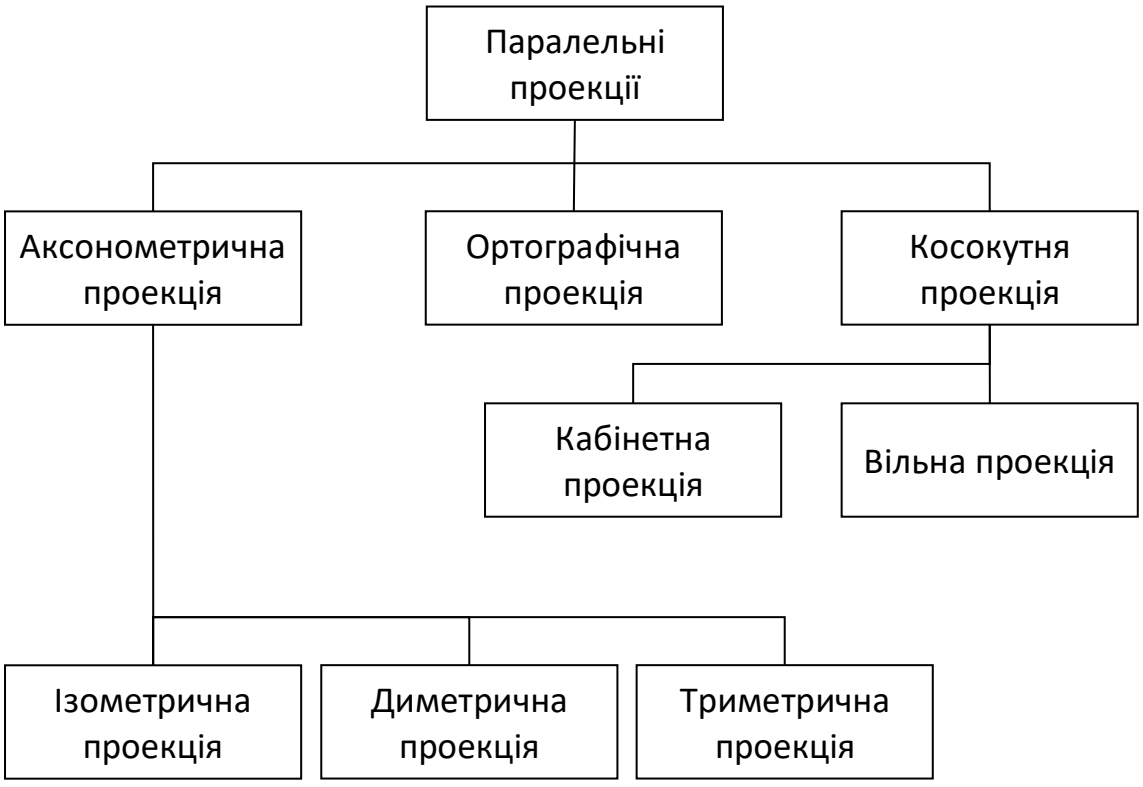


Рис. 3.4. Класифікація паралельних проекцій

Тепер розглянемо кожен клас паралельної проекції більш детально.

3.2.1. Ортографічна проекція

Під час *ортографічної* проекції картинна площина співпадає з однією з координатних площин або паралельна їй (рис. 3.5).

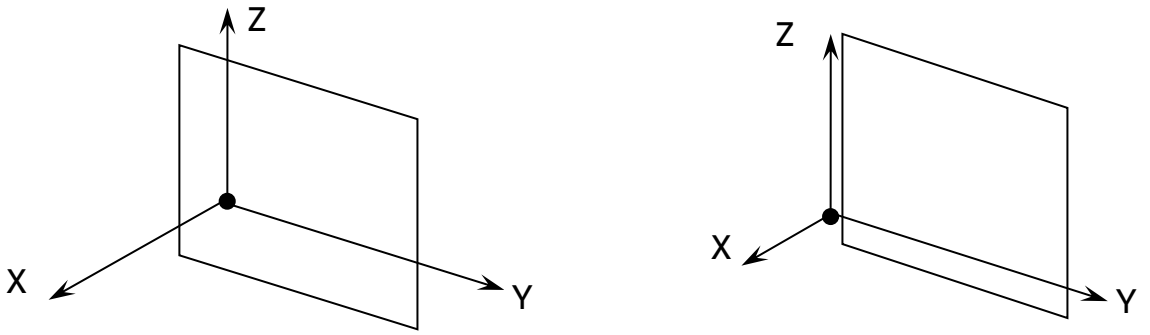


Рис. 3.5. Ортографічна проекція на площину Z0Y

Матриця проектування вздовж осі 0X на площину Z0Y має наступний вигляд:

$$[P_X] = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

У випадку, якщо площина проектування паралельна координатній площині, необхідно помножити матрицю $[P_x]$ на матрицю переміщення. В результаті отримуємо:

$$[P_X] \cdot \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ p & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ p & 0 & 0 & 1 \end{bmatrix}.$$

Аналогічно записуються матриці проектування вздовж двох інших осей:

$$[P_Y] = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & q & 0 & 1 \end{bmatrix}; \; [P_Z] = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & r & 1 \end{bmatrix}.$$

Всі три матриці проектування для ортографічної проекції – *вироджені*, тобто їх визначник дорівнює 0.

3.2.2. Аксонометрична проекція

Під час *аксонометричної* проекції прямі проектування перпендикулярні до картинної площини.

У відповідності до взаємного розташування площини проектування і координатних осей розрізняють три види проекцій:

- *триметрію* – нормальний вектор картинної площини утворює з ортами координатних осей попарно різні кути;
- *диметрію* – два кути між нормаллю картинної площини і координатними осями рівні;
- *ізометрію* – всі три кути між нормаллю картинної площини та координатними осями рівні.

Кожен із трьох видів аксонометричних проекцій отримується за допомогою комбінації поворотів, після якої виконується паралельне проектування. Якщо виконати поворот на кут ψ відносно осі ординат, потім поворот на кут φ навколо осі абсцис і наступне проектування вздовж осі аплікат отримаємо матрицю:

$$[M] = \begin{bmatrix} \cos\psi & \sin\varphi\sin\psi & 0 & 0 \\ 0 & \cos\psi & 0 & 0 \\ \sin\psi & -\sin\varphi\cos\psi & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Покажемо, як при цьому перетворюються орти координатних осей Z,Y,X:

$$\begin{aligned} &(1 \quad 0 \quad 0 \quad 1)\cdot[M] = (\cos\psi \quad \sin\varphi\sin\psi \quad 0 \quad 1); \\ &(0 \quad 1 \quad 0 \quad 1)\cdot[M] = (0 \quad \cos\psi \quad 0 \quad 1); \\ &(0 \quad 0 \quad 1 \quad 1)\cdot[M] = (\sin\psi \quad -\sin\varphi\cos\psi \quad 0 \quad 1). \end{aligned}$$

Диметрія характеризується тим, що довжини двох проекцій співпадають:

$$\cos^2\psi + \sin^2\varphi\cdot\sin^2\psi = \cos^2\varphi.$$

Звідси випливає, що:

$$\sin^2\psi = \tan^2\varphi.$$

У випадку ізометрії отримуємо:

$$\begin{aligned} \cos^2\psi + \sin^2\varphi\cdot\sin^2\psi &= \cos^2\varphi; \\ \sin^2\psi + \sin^2\varphi\cdot\cos^2\psi &= \cos^2\varphi. \end{aligned} \qquad (3.1)$$

З формул (3.1) випливає що:

$$\sin^2\varphi = \frac{1}{3};\ \sin^2\psi = \frac{1}{2}.$$

Для триметрії довжини проекцій попарно різні.

3.2.3. Косокутна проекція

Проекції, для отримання яких використовується пучок прямих, не перпендикулярних площині екрана, називають *косокутними*. При косокутному проектуванні орта осі Z на площину XOY (рис. 3.6) отримуємо:

$$(0 \quad 0 \quad 1 \quad 1) \rightarrow (\alpha \quad \beta \quad 0 \quad 1).$$

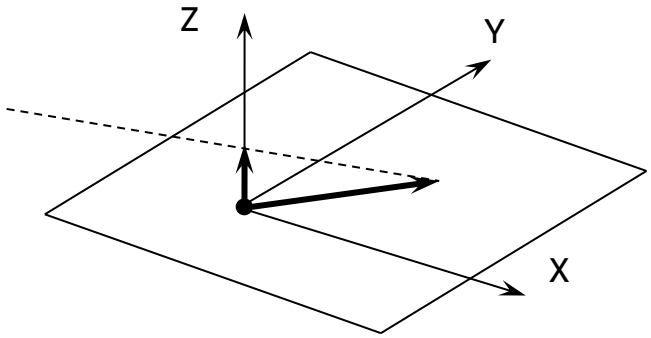


Рис. 3.6. Косокутна проекція

Матриця відповідного перетворення має наступний вигляд:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ \alpha & \beta & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Розрізняють два види косокутних проекцій:

- *вільна* – кут нахилу прямих проектування до площини екрана рівний половині прямого:

$$\alpha = \beta = \cos\frac{\pi}{4};$$

- *кабінетна* – частинний випадок вільної, коли масштаб по третій вісі вдвічі менший:

$$\alpha = \beta = \frac{1}{2}\cos\frac{\pi}{4}.$$

3.3. Перспективні проекції

Найбільш реалістично тривимірні об'єкти виглядають в центральній проекції із-за перспективних деформацій сцени. Центральні проекції паралельних прямих, які не паралельні площині проектування будуть сходитися в точці *сходу*. В залежності від кількості точок сходу, тобто від числа координатних осей, що перетинають площину проекції, розрізняються наступні перспективні проекції: одноточкова, двоточкова і триточкова (рис. 3.7).

Рис. 3.7. Перспективні проекції

Розглянемо спочатку одноточкову проекцію. Припустимо, що центр проектування лежить на осі Z в точці C(0,0,c) і площина проектування співпадає з координатною площиною XOY (рис. 3.8).

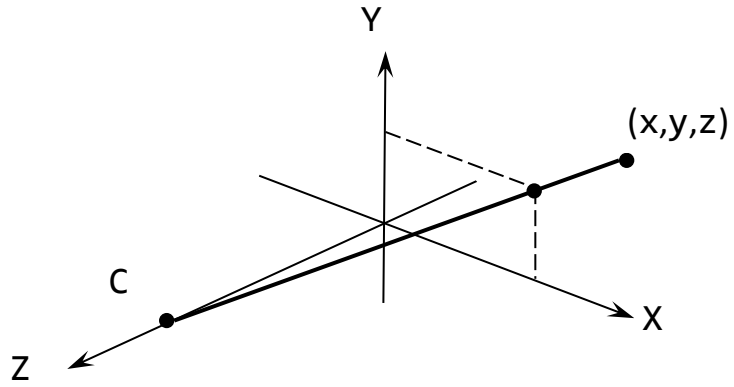


Рис. 3.8. Перспективна проекція

Візьмемо в просторі довільну точку M(x,y,z), проведемо через неї і точку C пряму та запишемо відповідні параметричні рівняння:

$$X' = xt;\ Y' = yt;\ Z' = c + (z - c)t$$

Знайдемо координати точки перетину прямої з площиною XOY. З умови Z'=0 отримаємо:

$$t = \frac{1}{1 - \frac{z}{c}}.$$

Тоді координати точки перетину будуть:

$$X' = \frac{1}{1 - \frac{z}{c}}x;\ Y' = \frac{1}{1 - \frac{z}{c}}y.$$

Запишемо отримані результати у вигляді матриці перспективного перетворення з точкою сходу на осі Z без проектування на площину XOY:

$$[Q] = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & -1/c \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Матриця перспективного перетворення з трьома точками сходу має наступний вигляд:

$$[Q] = \begin{bmatrix} 1 & 0 & 0 & -1/a \\ 0 & 1 & 0 & -1/b \\ 0 & 0 & 1 & -1/c \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Приклади зображення куба в перспективних проекціях з різною кількістю точок сходу наведені на рис. 3.9.

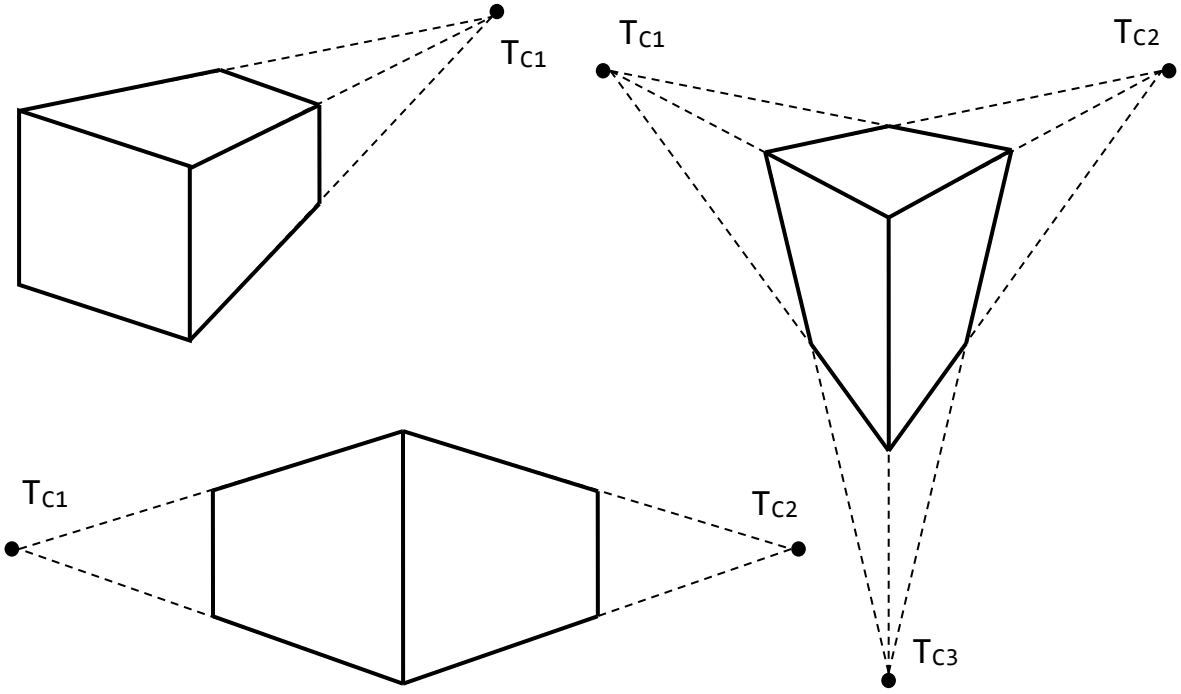


Рис. 3.9. Зображення кубу в перспективних проекціях

Запитання для самоконтролю

1. Що таке проективне перетворення?
2. Які два основні класи проективних перетворень розрізняють?
3. Які основні підкласи паралельної проекції?
4. Які основні підкласи перспективної проекції?

4. Фрактали

4.1. Поняття фракталу

Поняття фрактал і фрактальна геометрія, які з'явилися в кінці 70-х, середині 80-х років 20-го століття, надійно увійшли в користування математиків і програмістів. Слово фрактал утворене від латинського *fractus* і в перекладі означає «той, що складається з фрагментів». Воно було запропоноване *Бенуа Мандельбротом* в 1975 році для визначення нерегулярних, але самоподібних структур, якими він займався. Народження фрактальної геометрії прийнято пов'язувати з виходом в 1977 році книги Мандельброта «*The Fractal Geometry of Nature*». В його роботах використані наукові результати інших вчених, які працювали в період 1875-1925 років в тій же області (Пуанкаре, Фату, Жюліа, Кантор, Хаусдорф). Але тільки в наш час з'явилась можливість об'єднати їх роботи в єдину систему.

Роль фракталів в комп'ютерній графіці сьогодні достатньо велика. Вони приходять на допомогу, наприклад, коли потрібно, за допомогою декількох коефіцієнтів, задати лінії та поверхні дуже складної форми. З точки зору комп'ютерної графіки, фрактальна геометрія незамінна при генерації штучних хмарок, гір, поверхні моря. Фактично знайдений спосіб легкого подання складних неевклідових об'єктів, образи яких дуже схожі на природні.

Однією з основних властивостей фракталів є самоподібність. В найпростішому випадку деяка частина фракталу містить інформацію про увесь фрактал.

Визначення фракталу, яке сформулював Мандельброт, звучить так: «Фракталом називається структура, яка складається з частин, які в декому сенсі подібні цілому».

Виокремлюють наступні групи фракталів: геометричні, алгебраїчні, стохастичні, рукотворні, природні, детерміновані, недетерміновані.

Стохастичні фрактали, отримуються в тому випадку, якщо в ітераційному процесі випадковим чином змінювати які-небудь його параметри. При цьому виходять об'єкти дуже схожі на природні – несиметричні дерева, порізані берегові лінії і т. д. Двовимірні стохастичні фрактали використовуються для моделювання рельєфу місцевості і поверхні моря.

Геометричні та алгебраїчні фрактали розглянемо більш детально.

4.2. Геометричні фрактали

Фрактали цього класу найбільш наглядні. В двовимірному випадку їх отримують за допомогою деякої ламаної (або поверхні в тримірному випадку), яка називається *генератором*. За один крок алгоритму кожен з відрізків, які складають ламану, замінюється на ламану-генератор у відповідному масштабі. В результаті безкінечного повторювання цієї процедури, отримується фрактал геометричної форми.

Розглянемо один з таких фрактальних об'єктів – *тріадну криву Коха*. Побудова кривої починається з відрізка одиничної довжини – це нульове покоління кривої Коха. Далі кожна ланка (в нульовому поколінні один відрізок) замінюється на утворюючий елемент, який позначений через n=1 (рис. 4.1).

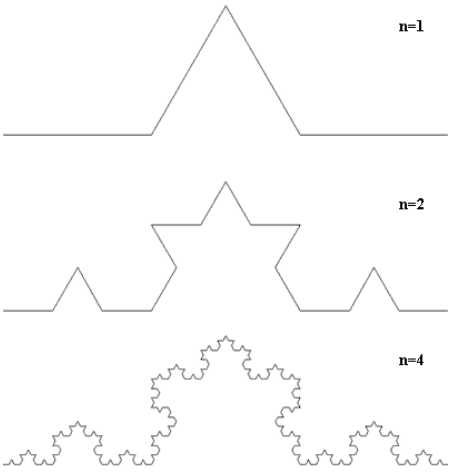


Рис. 4.1. Побудова тріадної кривої Коха

В результаті такої заміни отримується наступне покоління кривої Коха. В першому поколінні – це крива з чотирьох прямолінійних ланок, кожна довжиною по 1/3. Для отримання третього покоління виконуються ті ж дії – кожна ланка замінюється на зменшений утворюючий елемент. Отже, для отримання кожного наступного покоління необхідно замінити кожну ланку поточного зменшеним утворюючим елементом. Крива n-го покоління при будь-якому скінченному n називається *предфракталом*. На рис. 4.1 показані перше, друге і четверте покоління кривої. При n, що прямує до нескінченності крива Кох стає фрактальним об'єктом.

Для отримання іншого фрактальоного об'єкта треба змінити правила побудови. Нехай утворюючим елементом будуть два рівних відрізка, які з'єднані під прямим кутом. В нульовому поколінні замінимо одиничний відрізок на утворюючий елемент так, щоб кут був зверху. Можна сказати, що під час такої заміни виконується зміщення середини ланки.

Для побудови наступних поколінь виконується правило: найперша зліва ланка замінюється на утворюючий елемент так, щоб середина ланки зміщувалась ліворуч від напрямку руху, а при заміні наступних ланок, направлення зміщення середин відрізків повинні чергуватися. На рис. 4.2 показані декілька перших поколінь та 11-е покоління кривої, яка побудована за принципом, що описаний вище. Гранична фрактальна крива (при n прямуючому до нескінченності) називається *драконом Хартера-Хейтуея*.

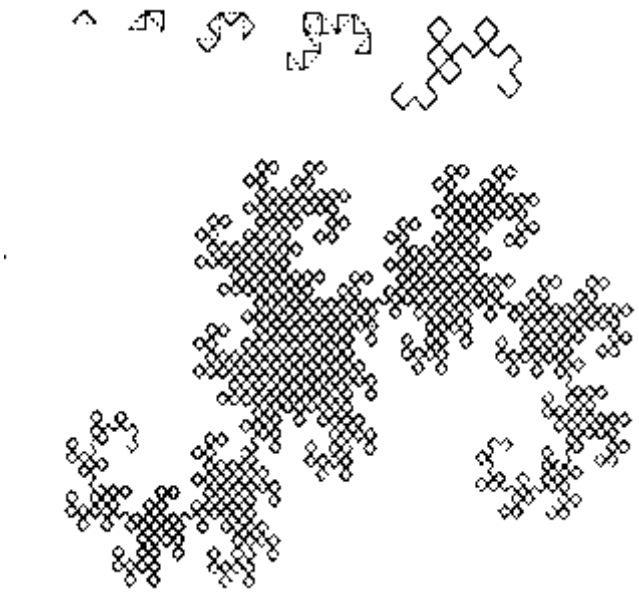


Рис. 4.2. Побудова дракона Хартера-Хейтуея

В комп'ютерній графіці використання графічних фракталів необхідно для отримання зображень дерев, кущів, берегової лінії. Двовимірні геометричні фрактали використовуються для створення об'ємних структур (рисунків на поверхні об'єкту).

4.3. Алгебраїчні фрактали

Це найкрупніша група фракталів. Отримують їх за допомогою нелінійних процесів в n-вимірних просторах. Найбільш вивчені двовимірні процеси. Інтерпретуючи нелінійний ітераційний процес, як дискретну динамічну систему, можна користуватися термінологією теорії цих систем: фазовий портрет, сталий процес, атрактор і т. д.

Відомо, що нелінійні динамічні системи володіють декількома стійкими станами. Той стан, в якому виявилася динамічна система після деякого числа ітерацій, залежить від її початкового стану. Тому кожен стійкий стан (або як говорять – атрактор) володіє деякою областю станів, з яких система обов'язково попаде в задані кінцеві стани. Таким чином

фазовий простір системи розбивається на області тяжіння атракторів. Якщо фазовим є двовимірний простір, то, забарвлюючи області тяжіння різними кольорами, можна отримати колірний фазовий портрет цієї системи (ітераційного процесу). Змінюючи алгоритм вибору кольору, можна отримати складні фрактальні картини з химерними багатоколірними візерунками. Несподіванкою для математиків стала можливість за допомогою примітивних алгоритмів породжувати дуже складні нетривіальні структури.

Як приклад розглянемо *множину Мандельброта* (рис. 4.3). Алгоритм її побудови досить простий і заснований на простому ітеративному виразі:

$$Z_{i+1} = Z_i^2 + C,$$

де Z_i та C – комплексні змінні.

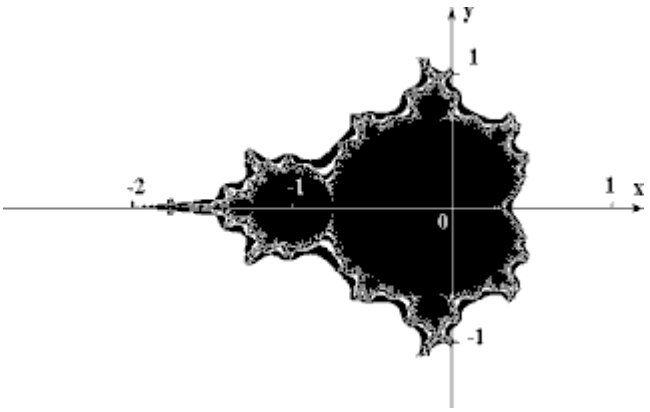


Рис. 4.3. Множина Мандельброта

Ітерації виконуються для кожної стартової точки C прямокутної або квадратної області – підмножини комплексної площини. Ітераційний процес продовжується до тих пір, поки Z_i не вийде за межі кола радіусу 2, центр якого лежить в точці (0,0) (це означає, що атрактор динамічної системи знаходиться в нескінченності), або після великого числа ітерацій (наприклад 200-500) Z_i зійдеться до якої-небудь точки кола. Залежно від кількості ітерацій, під час яких Z_i залишалася всередині кола, можна встановити колір точки C (якщо Z_i залишається всередині кола протягом достатньо великої кількості ітерацій, ітераційний процес припиняється і ця точка растру забарвлюється в чорний колір).

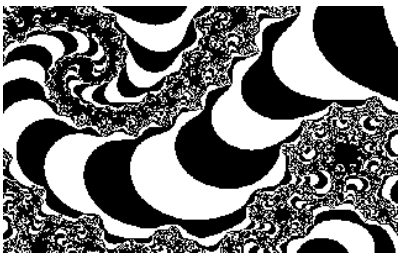


Рис. 4.4. Ділянка межі множини Мандельброта збільшена в 200 разів

Алгоритм, що описаний вище, дає наближення до так званої множини Мандельброта. Множині Мандельброта належать точки, які протягом нескінченного числа ітерацій не вирушають в нескінченність (точки мають чорний колір). Точки, що належать межі множини (саме там виникають складні структури) вирушають в нескінченність за скінченне число ітерацій, а точки, що лежать за межами множини, вирушають в нескінченність через декілька ітерацій (білий фон).

4.4. Система ітерованих функцій

Метод «Системи ітерованих функцій» (*Iterated Functions System* – IFS) з'явився в середині 80-х років як простий засіб отримання фрактальних структур.

IFS є системою функцій з деякого фіксованого класу функцій, що відображають одну багатовимірну нескінченність на іншу. Найбільш проста IFS складається з афінних перетворень площини:

$$X' = A \cdot X + B \cdot Y + C;\ Y' = D \cdot X + E \cdot Y + F.$$

У 1988 році відомі американські фахівці в теорії динамічних систем і ергодичної теорії *Барнслі* та *Слоан* запропонували деякі ідеї, засновані на міркуваннях теорії динамічних систем, для стиснення і зберігання графічної інформації. Вони назвали свій метод методом фрактального стиснення інформації. Походження назви пов'язане з тим, що геометричні образи, які виникають в цьому методі, зазвичай мають фрактальну природу в сенсі Мандельброта.

На підставі цих ідей Барнслі та Слоан створили алгоритм, який, за їх твердженням, дозволить стискувати інформацію 500-1000 разів. Коротко метод можна описати таким чином. Зображення кодується декількома простими перетвореннями (у нашому випадку афінними), тобто коефіцієнтами цих перетворень (у нашому випадку A, B, C, D, E, F).

Наприклад, закодувавши якесь зображення двома афінними перетвореннями, ми однозначно визначаємо його за допомогою 12-ти коефіцієнтів. Якщо тепер задатися якою-небудь початковою точкою (наприклад X=0, Y=0) і запустити ітераційний процес, то ми після першої ітерації отримаємо дві точки, після другої – чотири, після третьої – вісім і так далі. Через декілька десятків ітерацій сукупність отриманих точок описуватиме закодоване зображення. Але проблема полягає в тому, що дуже важко знайти коефіцієнти IFS, які б кодували довільне зображення.

Для побудови IFS застосовують окрім афінних й інші класи простих геометричних перетворень, які задаються невеликим числом параметрів, наприклад, проективні:

$$X' = \frac{(A_1 \cdot X + B_1 \cdot Y + C_1)}{(D_1 \cdot X + E_1 \cdot Y + F_1)};\ Y' = \frac{(A_2 \cdot X + B_2 \cdot Y + C_2)}{(D_2 \cdot X + E_2 \cdot Y + F_2)},$$

або квадратичні перетворення на площині:

$$X' = A_1 \cdot X^2 + B_1 \cdot X \cdot Y + C_1 \cdot Y^2 + D_1 \cdot X + E_1 \cdot Y + F_1;$$
$$Y' = A_2 \cdot X^2 + B_2 \cdot X \cdot Y + C_2 \cdot Y^2 + D_2 \cdot X + E_2 \cdot Y + F_2.$$

Як приклад використання IFS для побудови фрактальних структур, розглянемо криву Коха (рис. 4.1) і дракон Хартера-Хейтуея (рис. 4.2). Виокремимо в цих структурах подібні частини і, для кожної з них обчислимо коефіцієнти афінного перетворення. В афінний колаж буде включено стільки афінних перетворень, скільки існує частин подібних цілому зображенню.

Побудуємо IFS для дракона Хартера-Хейтуея. Для цього розташуємо перше покоління цього фрактала на сітці координат дисплея 640×350 (рис. 4.5). Позначимо точки ломаної, що вийшла A, B, C. По правилах побудови в цього фрактала дві частини подібні цілому – це ламані ADB і BEC. Знаючи координати кінців цих відрізків, можна обчислити коефіцієнти двох афінних перетворень, що переводять ламану ABC у ADB і BEC:

$$X' = -0.5 \cdot X - 0.5 \cdot Y + 490;\ Y' = 0.5 \cdot X - 0.5 \cdot Y + 120;$$
$$X' = 0.5 \cdot X - 0.5 \cdot Y + 340;\quad Y' = 0.5 \cdot X + 0.5 \cdot Y - 110.$$

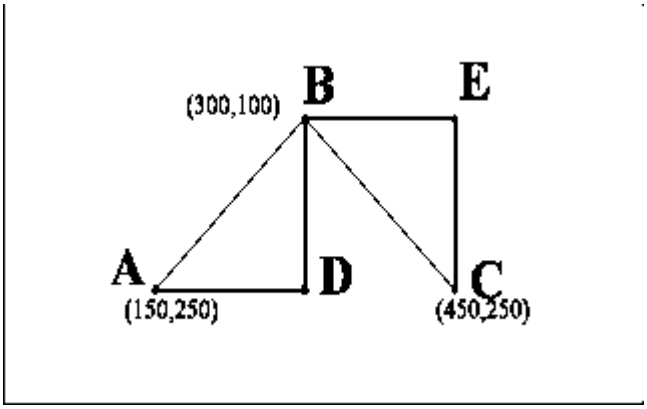


Рис. 4.5. Заготовка для побудови IFS дракона Хартера-Хейтуея

Задавшись початковою стартовою точкою (наприклад X=0, Y=0) і ітераційно діючи на неї цією IFS, після десятої ітерації на екрані отримаємо фрактальну структуру (рис. 4.6), яка є драконом Хартера-Хейтуея. Його кодом є набір коефіцієнтів двох афінних перетворень.

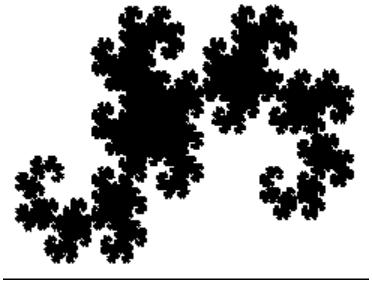


Рис. 4.6. Дракон Хартера-Хейтуея, побудований з допомогою IFS у прямокутнику 640×350

Аналогічно можна побудувати IFS для кривої Коха. Неважко побачити, що ця крива має чотири частини, подібні до цілої кривої (рис. 4.2). Для знаходження IFS знову розташуємо перше покоління цього фрактала на сітці координат дисплея 640×350 (рис. 4.7).

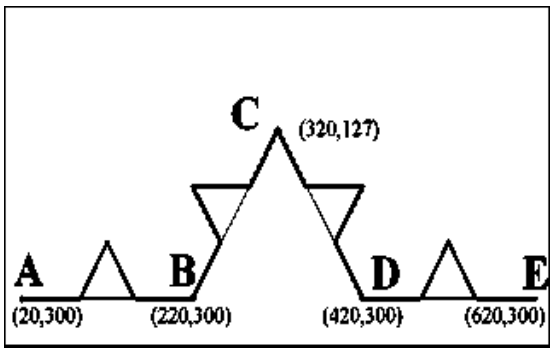


Рис. 4.7. Заготовка для побудови IFS кривої Коха

Для її побудови потрібний набір афінних перетворень, що складається з чотирьох перетворень:

$$X' = 0.333 \cdot X + 13.333; \qquad Y' = 0.333 \cdot Y + 200$$
$$X' = 0.333 \cdot X + 413.333; \qquad Y' = 0.333 \cdot Y + 200$$
$$X' = 0.167 \cdot X - 0.289 \cdot Y + 130; \quad Y' = -0.289 \cdot X + 0.167 \cdot Y + 256$$
$$X' = 0.167 \cdot X - 0.289 \cdot Y + 403; \quad Y' = 0.289 \cdot X + 0.167 \cdot Y + 71$$

Результат застосування цього афінного колажу після десятої ітерації можна побачити на рис. 4.8.

Рис. 4.8. Крива Коха, побудована за допомогою IFS у прямокутнику 640×350

Використання IFS для стискування звичайних зображень (наприклад фотографій) засновано на виявленні локальної самоподоби, на відміну від фракталів, де спостерігається глобальна самоподоба і знаходження IFS не дуже складне (ми самі тільки що в цьому переконалися). По алгоритму Барнслі відбувається виокремлення в зображенні пар областей, менша з яких подібна більшій, та збереження декількох коефіцієнтів, що кодують перетворення яке переводить велику область в меншу. Потрібно, щоб безліч «менших» областей покривало все зображення. При цьому в файл, що кодує зображення, будуть записані не лише коефіцієнти, які характеризують знайдені перетворення, але і місце розташування та лінійні розміри «великих» областей, які разом з коефіцієнтами описуватимуть локальну самоподобу кодованого зображення. Відновлювальний алгоритм, в цьому випадку, повинен застосовувати кожне перетворення не до всієї множини точок, що вийшли на попередньому кроці алгоритму, а до деякої їх підмножини, що належить області, відповідній застосовуваному перетворенню.

Запитання для самоконтролю

1. Що таке фрактали і яка їх основна властивість?
2. Які типи фракталів ви знаєте?
3. Назвіть та опишіть найвідоміші фрактальні криві?
4. Що таке IFS і для чого вони використовуються?

5. 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.

5.1. 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:

$$x(y_1w_2 - w_1y_2) + y(w_1x_2 - x_1w_2) + w(x_1y_2 - y_1x_2) = 0.$$

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

5.2. 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. 5.1).

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.

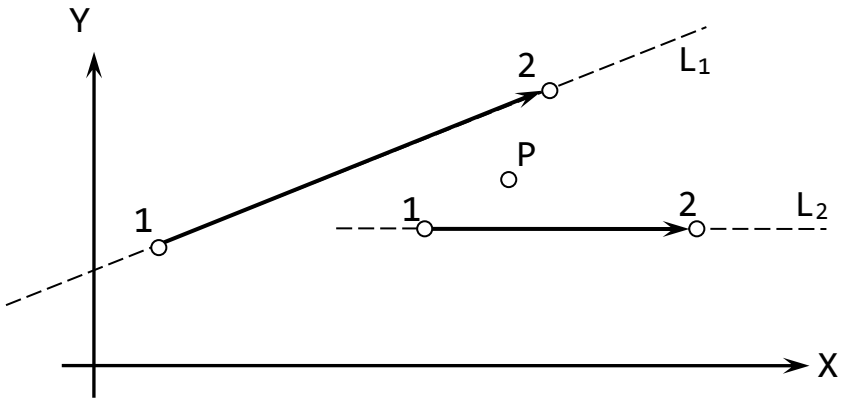


Рисунок 5.1. Взаємне розташування відрізків прямих і точки

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_1 та 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:

$$X(y_1w_2 - w_1y_2) + Y(w_1x_2 - x_1w_2) + W(x_1y_2 - y_1x_2) < 0. \qquad (5.1)$$

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.

5.3. 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, те ж саме відноситься до S_3 та S_4.

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

$$\begin{aligned} x(y_1w_2 - w_1y_2) + y(w_1x_2 - x_1w_2) + w(x_1y_2 - y_1x_2) &= 0; \\ x(y_3w_4 - w_3y_4) + y(w_3x_4 - x_3w_4) + w(x_3y_4 - y_3x_4) &= 0. \end{aligned} \qquad (5.2)$$

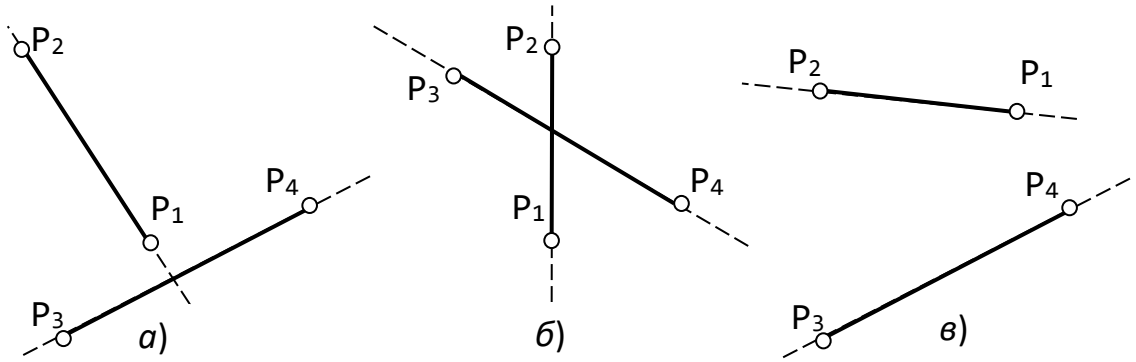


Рис. 5.2. Ілюстрація до співвідношень знаків величин, які визначаються рівняннями: *а*) $S_1S_2>0$ і $S_3S_4<0$; *б*) $S_1S_2<0$ і $S_3S_4<0$; *в*) $S_1S_2>0$ і $S_3S_4>0$

Умову перетину можна записати у компактному вигляді:

$$S_1 = \det(P_1, P_3, P_4)\,;\ S_2 = \det(P_2, P_3, P_4)\,;\ S_1S_2 < 0;$$
$$S_3 = \det(P_3, P_1, P_2)\,;\ S_4 = \det(P_4, P_1, P_2)\,;\ S_3S_4 < 0.$$

Аналогічно рівняння (5.2) 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.

5.4. 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.

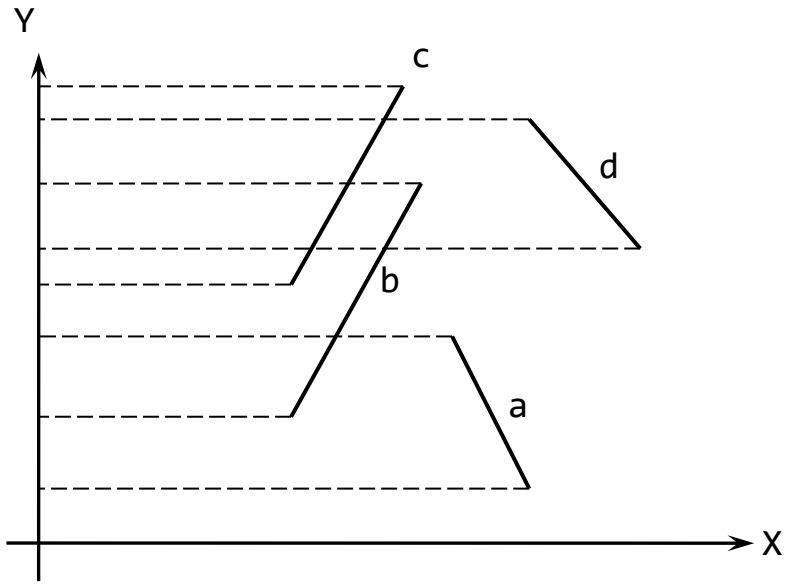


Рис. 5.3. Ілюстрація до затінення відрізків прямих

На рис 5.3. 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:

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

5.5. 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. 5.4).

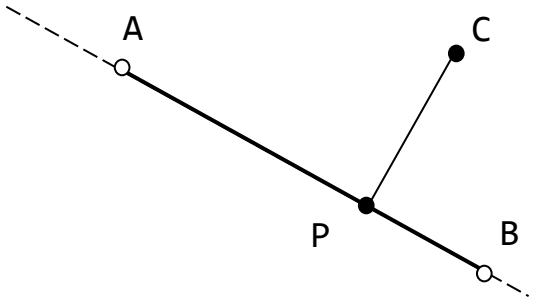


Рис. 5.4. Ілюстрація до задачі знаходження відстані від точки до прямої

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 детермінанту 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?

6. Задачі з прямокутниками

6.1. 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 проекції на вісь j=1…d.

Нехай задані два інтервали R'=[x'$_1$, x'$_2$] і R''=[x''$_1$, x''$_2$]. Умова R'∩R''≠∅ еквівалентна одній з наступних взаємовиключних умов:

$$x_1' \le x_1'' \le x_2'; \; x_1'' \le x_1' \le x_2''. \qquad (6.1)$$

Чотири можливі ситуації взаємного розташування кінців інтервалів, що відповідають умові R'∩R''≠∅ фактично охоплюється співвідношеннями (6.1). Тому для перевірки перетину R' і R'' достатньо перевірити факт потрапляння або лівого кінця R' всередину R'', або лівого кінця R'' всередину R'.

6.2. 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. 6.1.

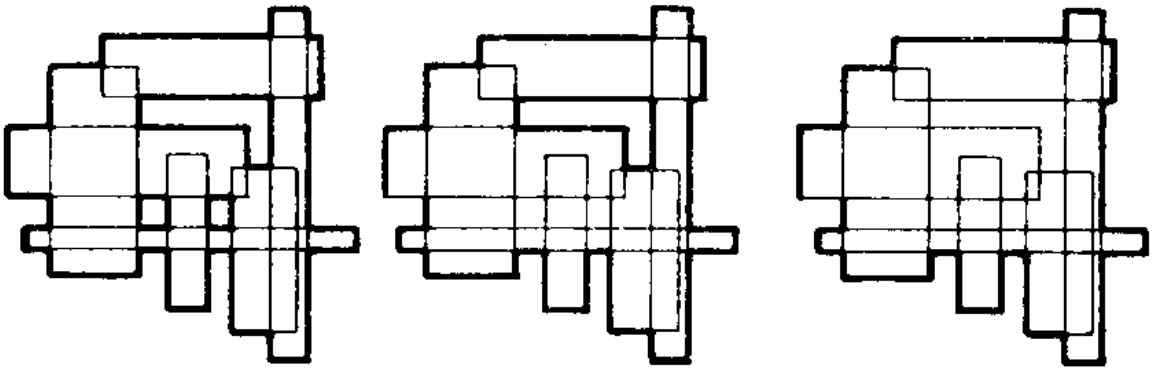


Рис. 6.1. Приклади: контуру, нетривіального контуру і зовнішнього контуру для об'єднання прямокутників

Для подальшого викладу зручно вважати кожне ребро таким, що складається з двох дуг, орієнтованих в протилежних напрямах і лежачих по різні сторони від заданого ребра (рис. 6.2).

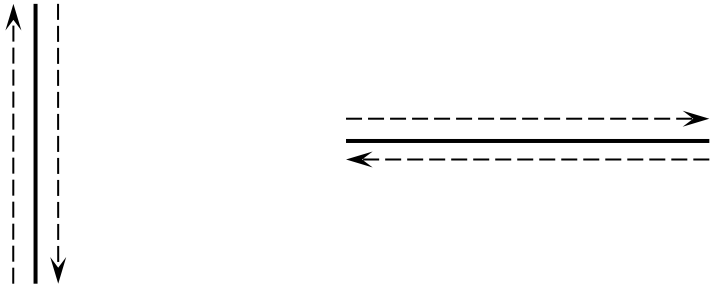


Рис. 6.2. Погодження відносно напрямів об'єднуючих дуг для ребра

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. 6.1 проілюстровано на рис. 6.3.

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*.

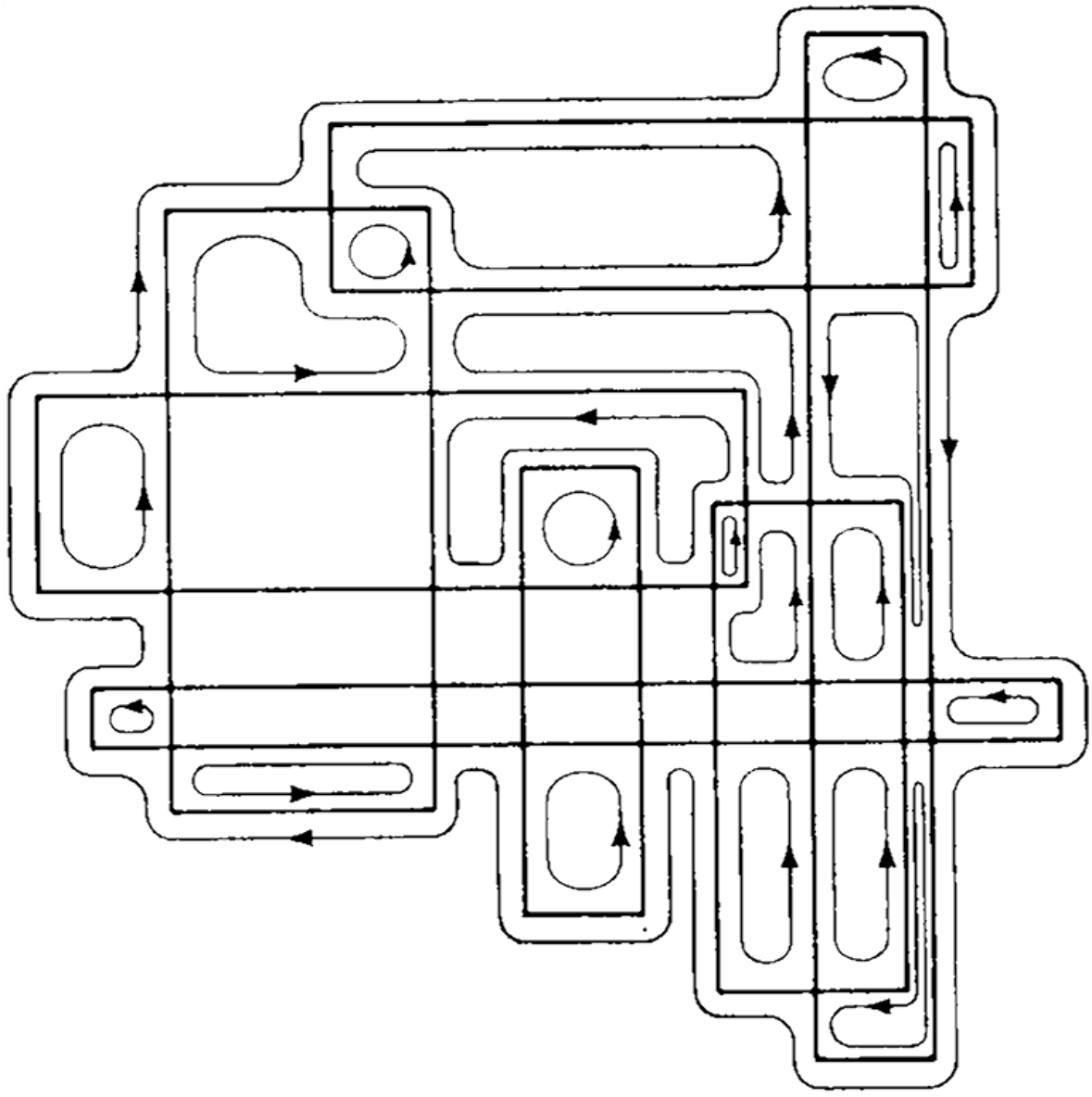


Рис. 6.3. Нетривіальні ланцюги для об'єднання прямокутників з рис. 6.1

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. 6.4) 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.

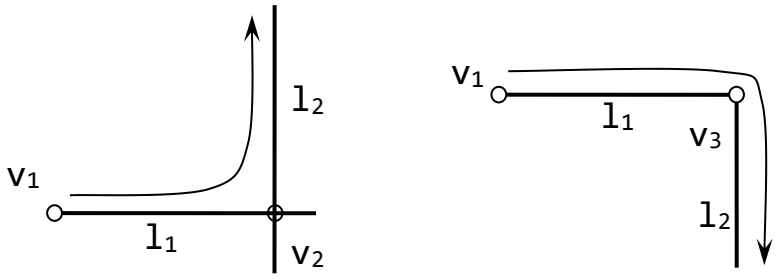


Рис. 6.4. Ситуації, що виникають в алгоритмі на етапі руху

Вищеописаний 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. 6.5). 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.

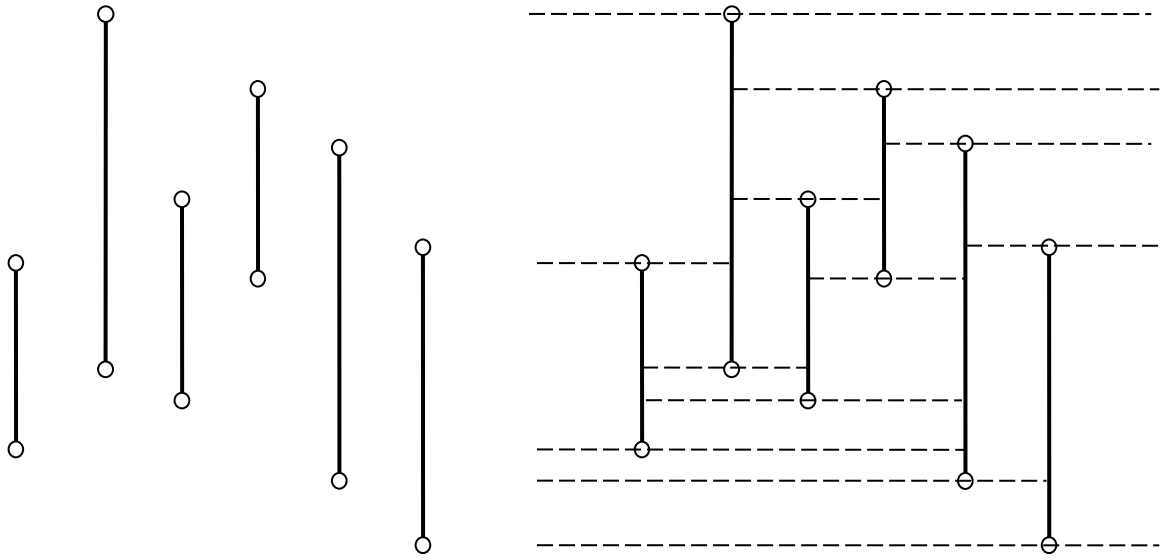


Рис. 6.5. Множина вертикальних відрізків V та результуюча ГКС

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. 6.4), 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 $x_2 \le x_3$, 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?

7. ЗАДАЧІ З БАГАТОКУТНИКАМИ

7.1. 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:

$$S(P) = \frac{1}{2}\sum_{i=0}^{n-1}\left(v_{i,x} \cdot v_{i+1,y} - v_{i,y} \cdot v_{i+1,x}\right). \qquad (7.1)$$

Формула (7.1) підраховує площу багатокутника зі знаком, що залежить від орієнтації його вершин. У випадку, коли вершини впорядковані в напрямку проти годинникової стрілки $S(P)<0$.

7.2. 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 (5.1) 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. При вирішенні 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. 7.1). 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 – точку дотику необхідно враховувати два рази.

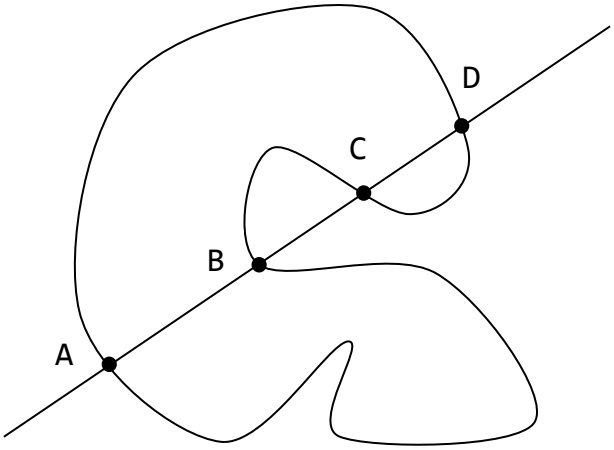


Рис. 7.1. Приклад використання принципу перевірки на парність для визначення точок, які лежать всередині замкненої кривої

7.3. Розрізання довільного відрізка прямої довільним 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:

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

$$S_i = X_i(y_1w_2 - w_1y_2) + Y_i(w_1x_2 - x_1w_2) + W_i(x_1y_2 - y_1x_2).$$

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 знака, 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:

$$U_1 = \det(P_1, A_j, A_{j+1});$$
$$U_1 = x_1(Y_jW_{j+1} - W_jY_{j+1}) + y_1(W_jX_{j+1} - X_jW_{j+1}) + w_1(X_jY_{j+1} - Y_jX_{J+1}); \quad (7.2)$$

$$U_2 = \det(P_2, A_j, A_{j+1});$$
$$U_2 = x_2(Y_jW_{j+1} - W_jY_{j+1}) + y_2(W_jX_{j+1} - X_jW_{j+1}) + w_2(X_jY_{j+1} - Y_jX_{J+1}); \quad (7.3)$$

$$U_3 = \det(P_1, A_k, A_{k+1});$$
$$U_3 = x_1(Y_kW_{k+1} - W_kY_{k+1}) + y_1(W_kX_{k+1} - X_kW_{k+1}) + w_1(X_kY_{k+1} - Y_kX_{k+1}); \quad (7.4)$$

$$U_4 = \det(P_2, A_k, A_{k+1});$$
$$U_4 = x_2(Y_kW_{k+1} - W_kY_{k+1}) + y_2(W_kX_{k+1} - X_kW_{k+1}) + w_2(X_kY_{k+1} - Y_kX_{k+1}). \quad (7.5)$$

Якщо орієнтація сторін багатокутника впорядкована за годинниковою стрілкою, величина U_i має від'ємний знак у рівняннях (7.2)-(7.5), 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 7.2 наведені 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.

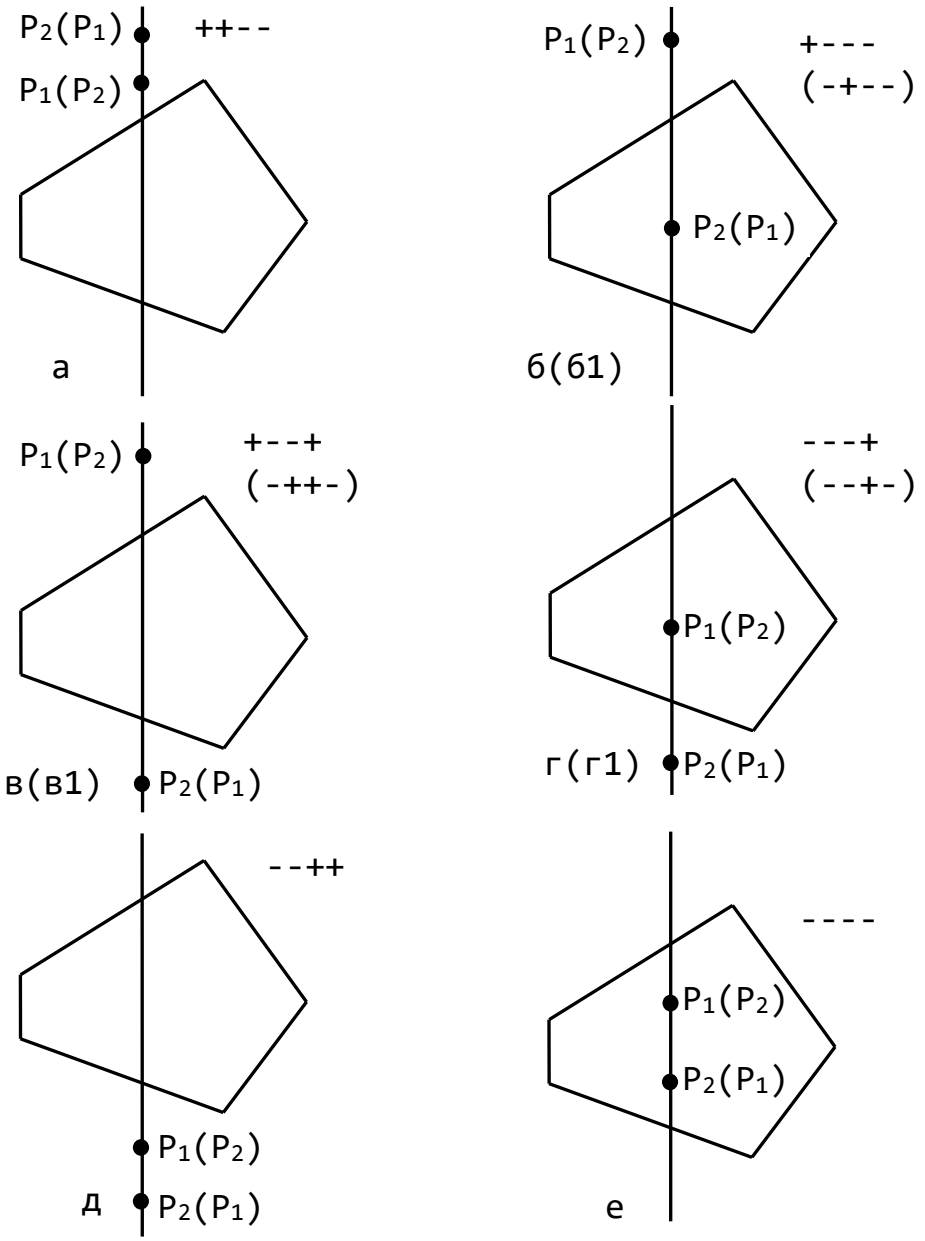


Рис. 7.2. Шість варіантів розрізання відрізка прямої багатокутником

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?

8. 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.

8.1. 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. 8.1).

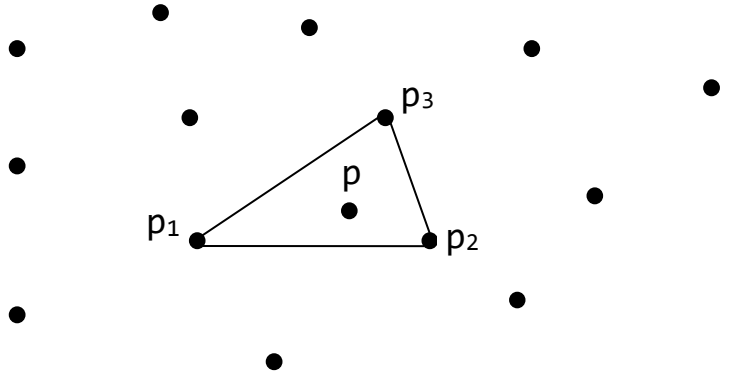


Рис. 8.1. Ілюстрація до визначення не крайньої точки

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 по напряму з позитивним напрямом осі X системи координат. Протягом руху від вершини до вершини полярний кут променя монотонно збільшується. Саме це малося на увазі, коли говорилося про те, що вершини багатокутника P «впорядковані» (рис. 8.2).

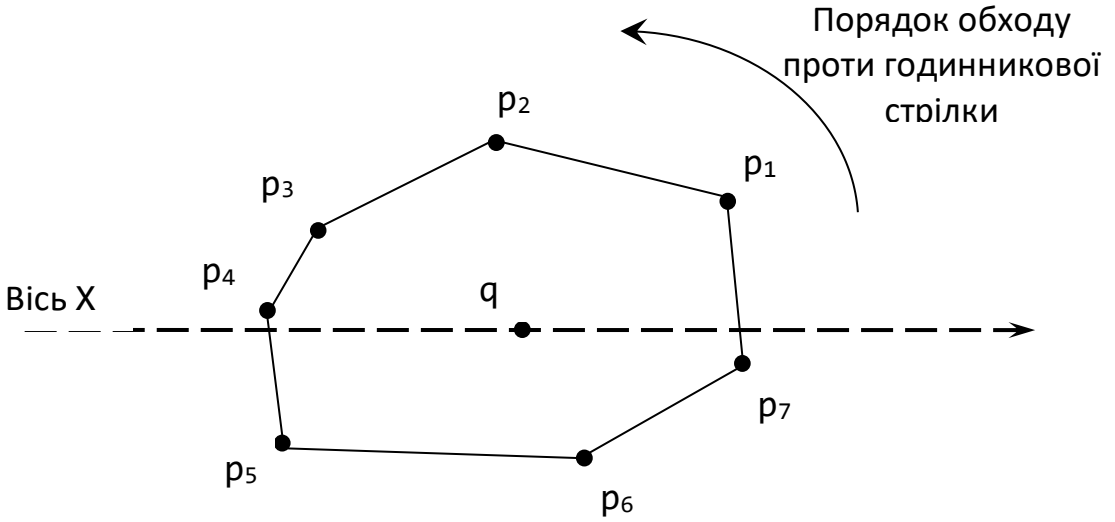


Рис. 8.2. Вершини багатокутника P впорядковані відносно точки q

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 тривіально обчислений за O(Nk) арифметичних операцій. Грехем запропонував інший метод знаходження внутрішньої точки, відмітивши, що цілком достатньо узяти центроїд будь-яких трьох не колінеарних точок.

8.2. Метод 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 розділі ми досліджуємо наш алгоритм з точки зору наявності в ньому непотрібних обчислень.

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, 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.

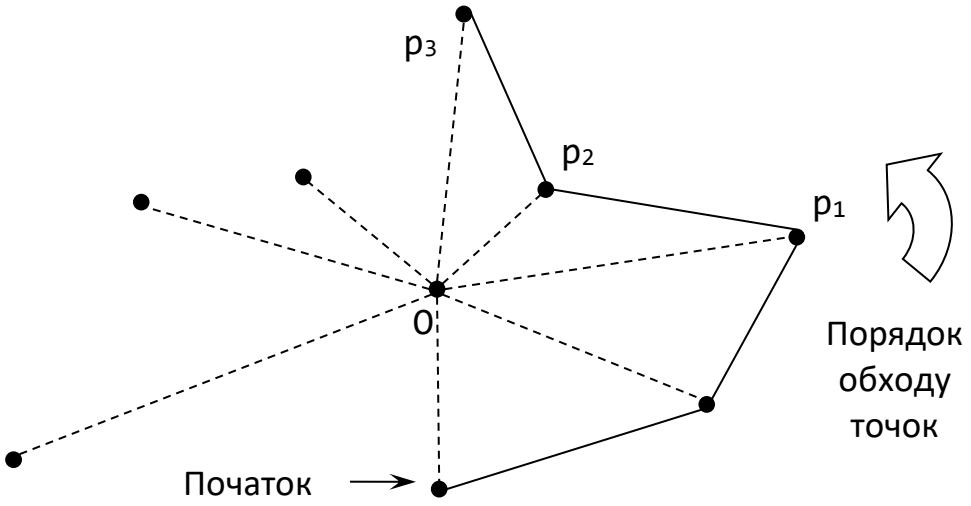


Рис. 8.3. Ілюстрація до алгоритму Грехема

Подавши впорядковані точки у вигляді двічі зв'язаного кільцевого списку отримуємо ситуацію, що показана на рис. 8.3. 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.

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».

Це можна легко визначити, скориставшись формулою (5.1). 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 ($0p_1p_3$). 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 методом обходу Грехема. Опукла оболонка N точок на площині може бути знайдена за час O(N log N) при пам'яті O(N) з використанням лише арифметичних операцій і порівнянь.

В методі Грехема використовуються полярні координати, що може бути не зручно в деяких системах. Ендрю запропонував метод, що дозволяє запобігти цьому. Коротко розглянемо цей метод.

Нехай на площині задана множина з N точок. Визначимо спочатку його ліву і праву крайні точки l і r (рис. 8.4) 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.

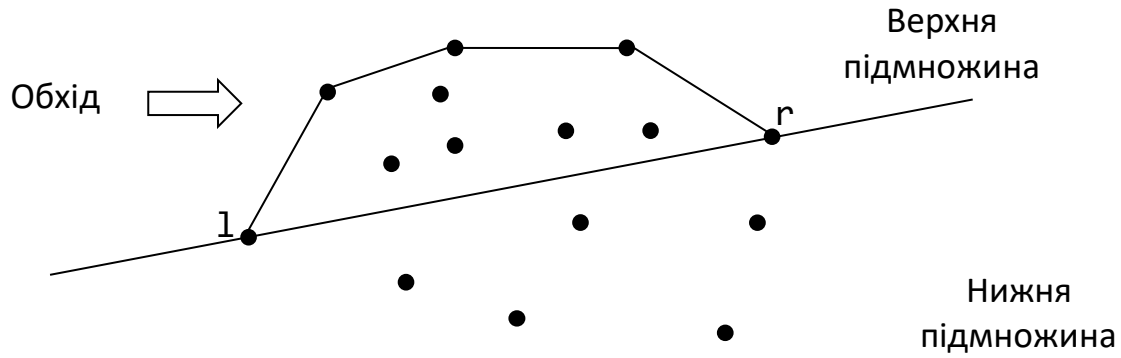


Рис. 8.4. Ілюстрація до алгоритму Ендрю

Розглянемо побудову верхньої оболонки. Точки впорядковуються відповідно до зростання абсциси, і до отриманої послідовності застосовується метод обходу Грехема. При такому підході 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.

8.3. Метод 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, є чи ні деяка точка крайньою. Проте якщо дано дві точки, то безпосередньо можна перевірити, є чи ні відрізок, що їх сполучає, ребром опуклої оболонки.

Теорема. Відрізок l, що визначений двома точками, є ребром опуклої оболонки тоді і лише тоді, коли всі інші точки заданої множини лежать на l або з однієї сторони від нього (рис. 8.5).

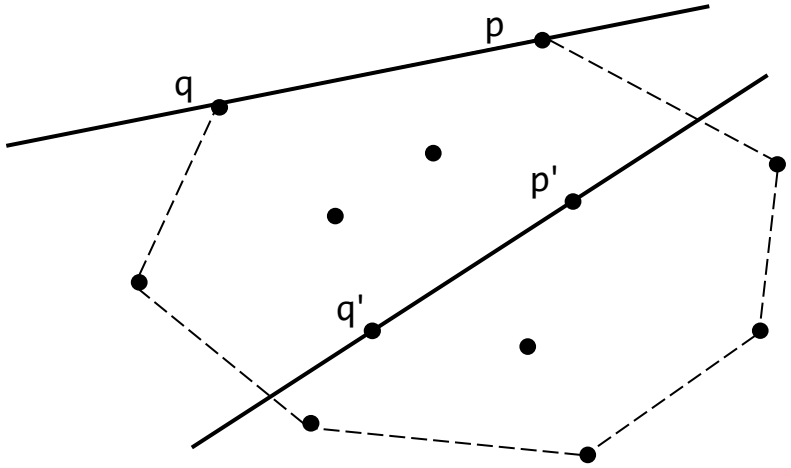


Рис. 8.5. Ребро опуклої оболонки

Ребро випуклої 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 випуклої 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 показано, як використати цей факт, щоб зменшити необхідний час до $O(N^2)$.

Алгоритм Джарвіса обходить кругом опуклу оболонку (звідси і відповідна назва – обхід методом Джарвіса), породжує в потрібному порядку послідовність крайніх точок, по одній точці на кожному кроці (рис. 8.6). Таким чином будується частина опуклої оболонки (ламана лінія) від найменшої в лексикографічному порядку точки (p_1 на рис. 8.6) 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.

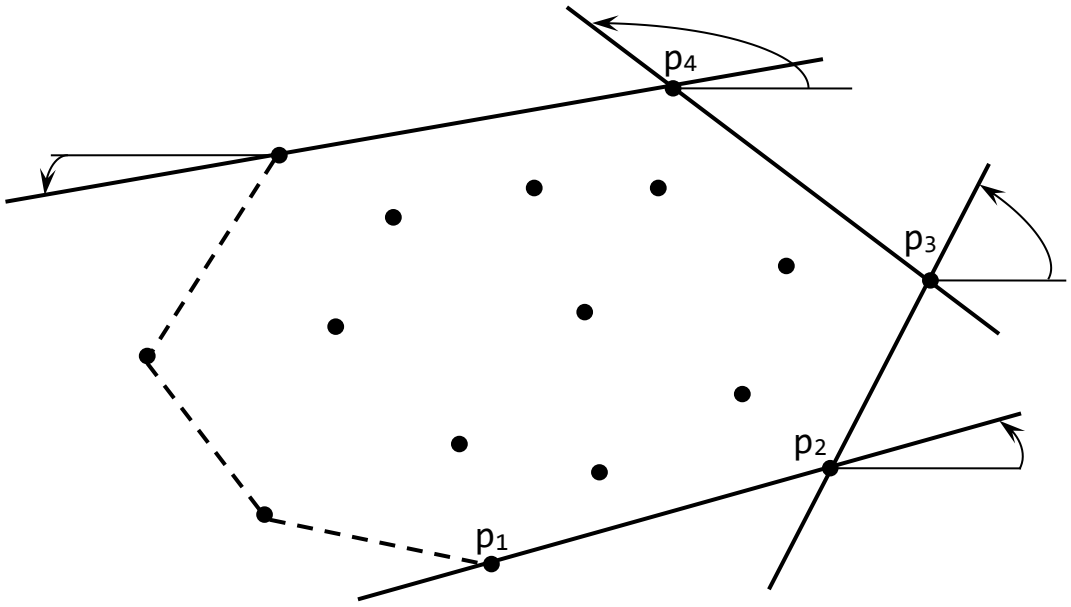


Рис. 8.6. Ілюстрація до алгоритму Джарвіса

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, той 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 метод 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.

8.4. 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. 8.7), так само, як і у варіанті алгоритму Грехема, запропонованому Ендрю.

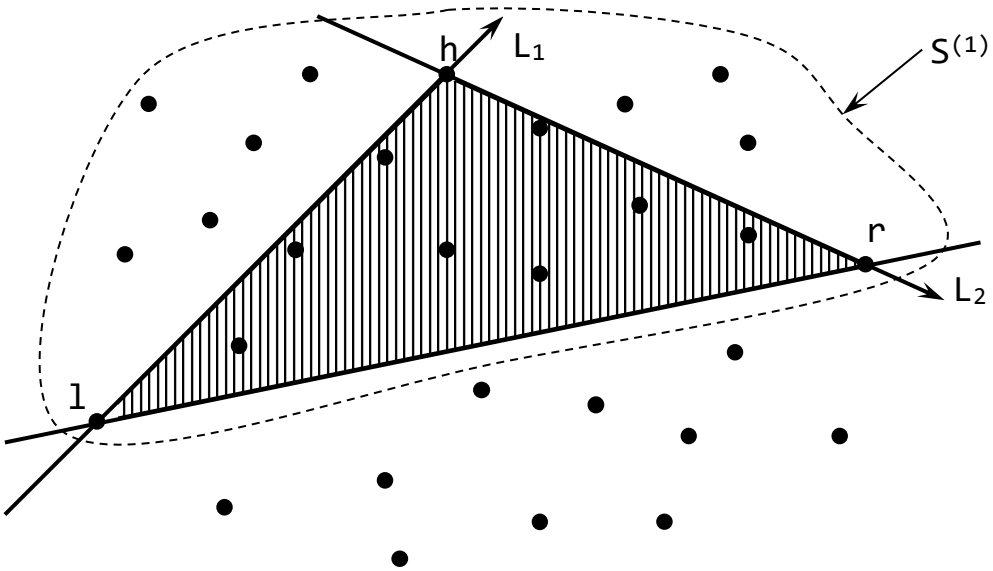


Рис. 8.7. Ілюстрація до швидкого методу побудови опуклої оболонки

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. 8.7). 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 випуклої 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. Даний метод використовує процедури обчислення площі трикутника та визначення положення точки відносно прямої. Кожна з цих процедур вимагає декількох операцій складання і множення.

Даний алгоритм в середньому випадку має час роботи $O(N \log N)$, а в найгіршому – $O(N^2)$.

8.5. Алгоритм апроксимації опуклої оболонки

Замість вибору алгоритму побудови опуклої оболонки на підставі його складності в середньому, можна піти по альтернативній дорозі, розробляючи алгоритми, що будують апроксимації для реальної опуклої оболонки, розмінюючи тим самим точність на простоту і 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 сповна прийнятно в прикладній статистиці, де результати спостережень не є точними, а відомі з деякою цілком певною 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.

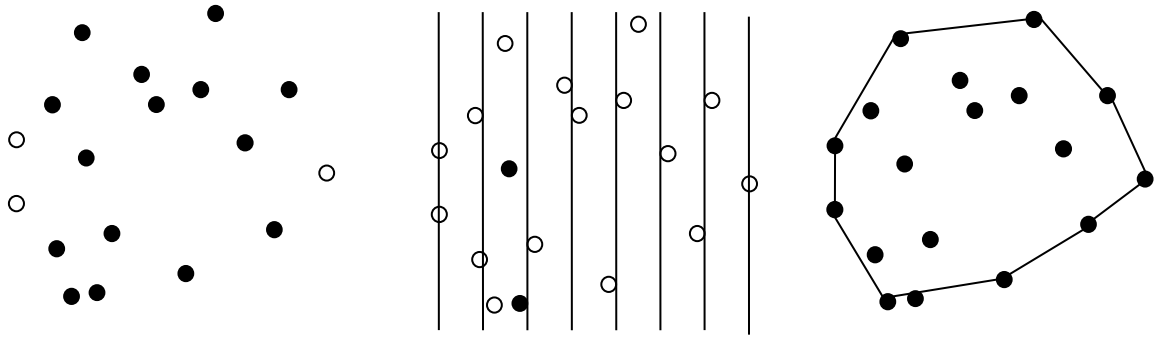


Рис. 8.8. Ілюстрація до алгоритму апроксимації випуклої оболонки: (а) – задана множина точок на площині, в якій виокремлюють найбільш ліві та найбільш праві точки; (б) – розбиття множини точок на частини в залежності від належності їх до однієї з k смуг та визначення в кожній смузі точок з екстремальною ординатою; (в) – наближена опукла оболонка.

На першому кроці алгоритму шукаються мінімальне і максимальне значення координати X точок множини (рис. 8.8, 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 шукаються дві точки, що мають мінімальне і максимальне значення координати Y (рис. 8.8, б). Крім того, вибираються точки з екстремальними значеннями координати X. Якщо при цьому 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 2k+4 точок. Нарешті, будується опукла оболонка множини S^*, яка є апроксимацією оболонки заданої множини (рис. 8.8, в). Відзначимо, що оболонка яка вийшла, насправді є лише апроксимацією: у прикладі на рис. 8.8 (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 даному 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.

9. 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.

9.1. ЖАДІБНА ТРІАНГУЛЯЦІЯ

«Жадібний» метод – це такий метод, під час якого ніколи не відміняється 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, використовуючи 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 при досягненні кількості ребер в множині RT 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.

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

9.2. 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. 9.1).

Рис. 9.1. Тріангуляція Делоне

Тріангуляція Делоне добре збалансована – її трикутники спрямовуються 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. 9.2).

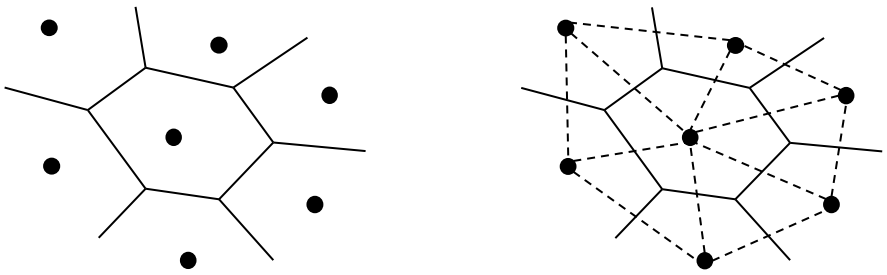


Рис. 9.2. Діаграма Вороного

9.2.1. 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.

9.2.2. 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. 9.3).

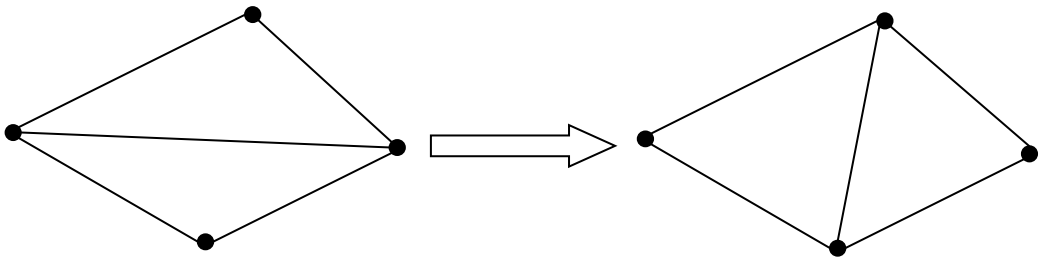


Рис. 9.3. Фліп ребра суміжних трикутників

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 дане твердження неістинне, то можна побудувати декілька різних тріангуляцій Делоне;
- ребра знаходяться по правилу лівої орієнтації (всі точки знаходяться праворуч, пуста 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 примикають 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 для можна порівняти з надуванням плоскої бульбашки на ребрі (рис. 9.4).

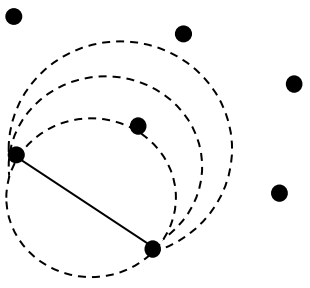


Рис. 9.4. Знаходження спряженої вершини для ребра

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.

9.3. 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.

9.3.1. 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. 9.5.

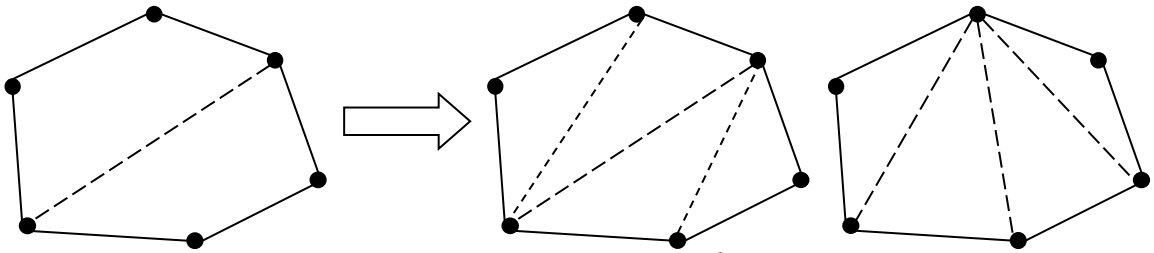


Рис. 9.5. Тріангуляція опуклих багатокутників

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

9.3.2. 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.

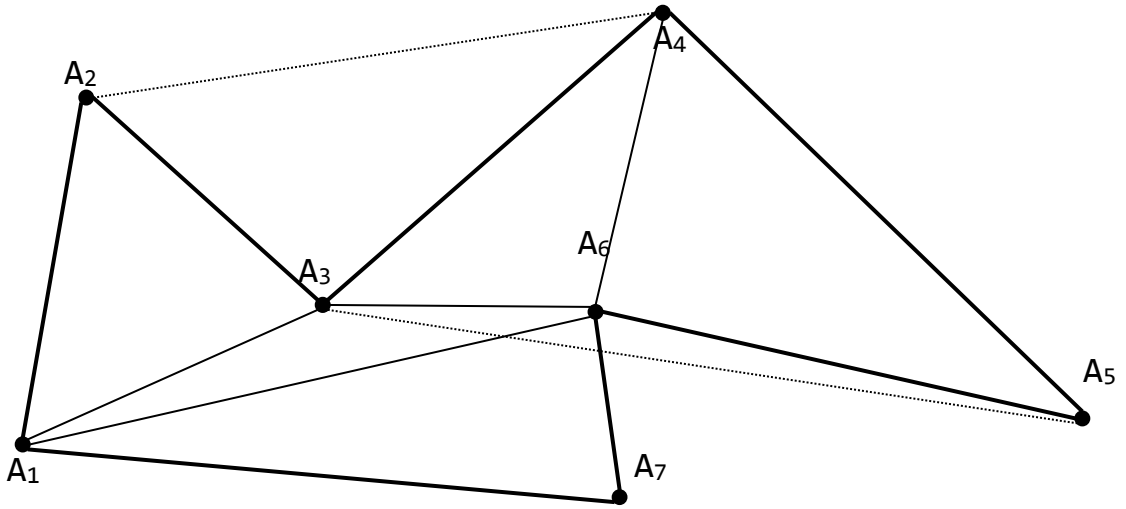


Рис. 9.6. Тріангуляція неопуклого багатокутника

Одним з методів тріангуляції неопуклого багатокутника є алгоритм послідовної побудови трикутників на трьох сусідніх вершинах з перевіркою отриманого результату (рис. 9.6).

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+2} та A_iA_{i+1} лівий поворот. Їх векторний добуток повинен бути додатнім. Для цього знаходимо знак визначника, що побудований з координат цих точок.
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 даному посібнику 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?

10. Моделювання кривих

Моделювання – 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.

10.1. Інтерполяція

Задача. Задано $f(x_i)=y_i$, $x_i<x_{i+1}$, $i\in[1,n]$ (рис. 10.1). Побудувати функцію $f(x)$, що задовольняє наведеній умові. При цьому 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.

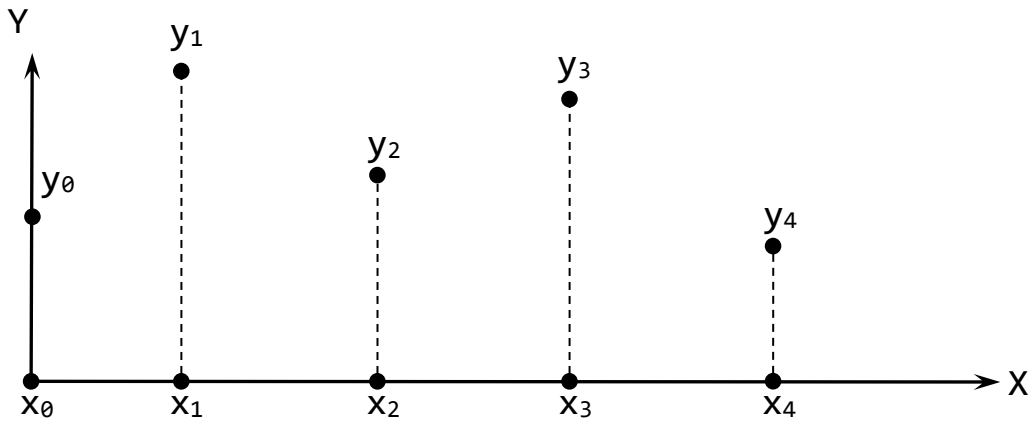


Рис. 10.1. Постановка задачі для інтерполяції

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.

10.1.1. Інтерполяційний многочлен Лагранжа

Інтерполяційний многочлен Лагранжа – многочлен мінімального степеня, що приймає дані значення у даному 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) степеня не більшого від 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 степеня n;
- $l_j(x_i)=1$;
- $l_j(x_i)=0$ при $i\neq j$.

Звідси випливає, що L(x), як лінійна комбінація $l_j(x)$, може мати степінь не більший від n, та $L(x_j)=y_j$.

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 при додаванні нової точки.

10.1.2. 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).

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 при додаванні нової точки, оскільки попередні результати від цього не змінюються. Всі інші недоліки многочлена Лагранжа можна віднести і до многочлена Ньютона.

10.1.3. 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. 10.2).

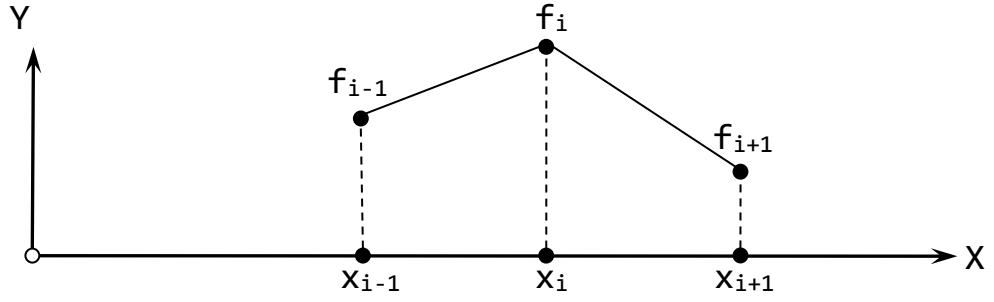


Рис. 10.2. Лінійна інтерполяція

З геометричної точки зору це означає заміну функції f прямою,яка 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 називається частинно-лінійна інтерполяція. Така крива є неперервною, тобто відноситься до класу C^0. Основним недоліком лінійної інтерполяції є низька гладкість отриманої кривої.

10.1.4. Сплайни

Сплайн – частинний поліном степеня K з неперервною похідною степеня K-1 у точках з'єднання сегментів (рис. 10.3).

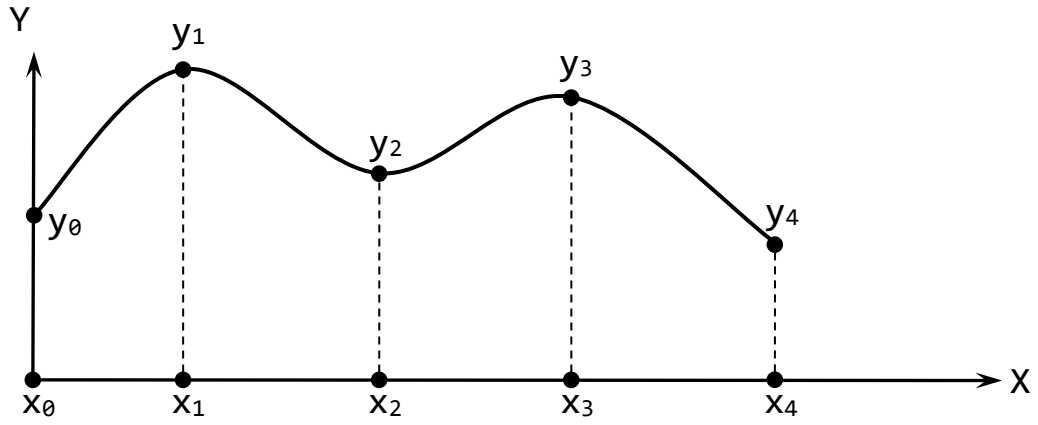


Рис. 10.3. Сплайн

Термін сплайн походить від англійського слова *spline* – що означає гнучку смужку сталі, яку застосовували креслярі для проведення плавних кривих при побудові 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.

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

$$S_i(x) = a_i + b_i(x - x_i) + \frac{c_i}{2}(x - x_i)^2 + \frac{d_i}{6}(x - x_i)^3. \qquad (10.1)$$

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:

$$S_i(x_{i-1}) = S_{i-1}(x_{i-1});\ S_i'(x_{i-1}) = S_{i-1}'(x_{i-1});\ S_i''(x_{i-1}) = S_{i-1}''(x_{i-1}). \qquad (10.2)$$

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

$$S_i(x_{i-1}) = f(x_{i-1}). \qquad (10.3)$$

Об'єднавши рівняння (10.2) та (10.3) в систему та позначивши через h_i різницю $(x_i - x_{i-1})$, отримаємо наступні формули для знаходження коефіцієнтів сплайну:

$$a_i = f(x_i);$$

$$h_i c_{i-1} + 2(h_i + h_{i+1})c_i + h_{i+1}c_{i+1} = 6\left(\frac{f_{i+1} - f_i}{h_{i+1}} - \frac{f_i - f_{i-1}}{h_i}\right);$$

$$d_i = \frac{c_i - c_{i-1}}{h_i};$$
$$b_i = -\frac{1}{2}h_i c_i - \frac{1}{6}h_i^2 d_i + \frac{f_i - f_{i-1}}{h_i}.$$

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 трьохдіагональної 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:

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

Знайшовши за методом прогонки коефіцієнти c_i легко знаходимо інші коефіцієнти та підставивши їх у рівняння (10.1) отримуємо значення функції в шуканій точці.

10.2. Апроксимація

В попередньому пункті 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.

10.2.1. Криві Без'є

При вирішення задачі апроксимації типовим є використання кривих Без'є. 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, тобто переміщуючи опорні точки курсором на екрані).

Криві Без'є були запроваджені в 1962 році *П'єром Без'є* 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. 10.4). 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.

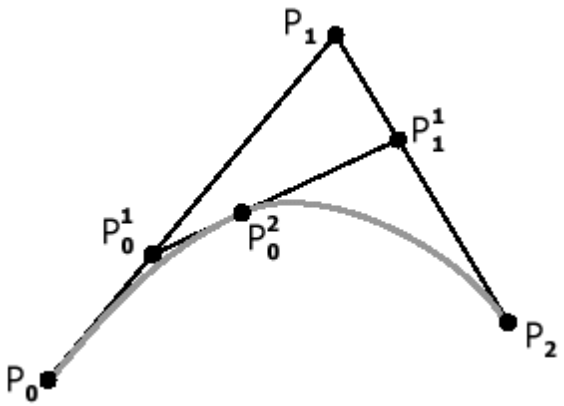


Рис. 10.4. Крива Без'є побудована по трьом опорним точкам

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.

$$\begin{aligned} P_0^1 &= (1-t)P_0 + tP_1; \\ P_1^1 &= (1-t)P_1 + tP_2; \\ P_0^2 &= (1-t)P_0^1 + tP_1^1 = (1-t)^2 P_0 + 2t(1-t)P_1 + t^2 P_2. \end{aligned} \qquad (10.4)$$

Таким чином, із формул (10.4) випливає, що ми отримали криву другого порядку.

Тепер аналогічним чином побудуємо криву Без'є за чотирма опорними точками (рис. 10.5).

$P_0^1 = (1-t)P_0 + tP_1;$
$P_1^1 = (1-t)P_1 + tP_2;$
$P_2^1 = (1-t)P_2 + tP_3;$
$P_0^2 = (1-t)P_0^1 + tP_1^1 = (1-t)^2P_0 + 2t(1-t)P_1 + t^2P_2;$
$P_1^2 = (1-t)P_1^1 + tP_2^1 = (1-t)^2P_1 + 2t(1-t)P_2 + t^2P_3;$
$P_0^3 = (1-t)P_0^2 + tP_1^2 = (1-t)^3P_0 + 3t(1-t)^2P_1 + 3t^2(1-t)P_2 + t^3P_3.$

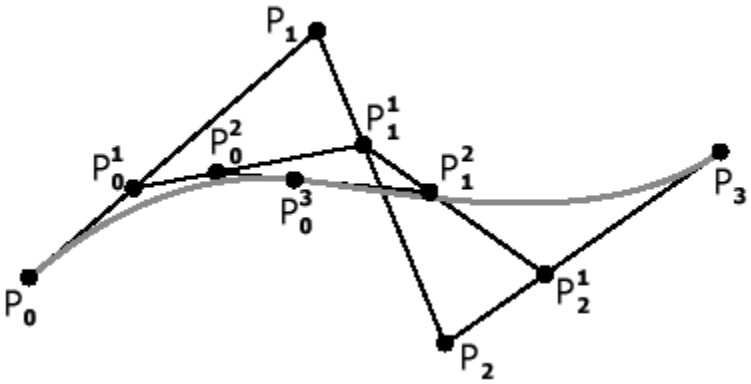


Рис. 10.5. Крива Без'є побудована по чотирьом опорним точкам

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

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

Загальне аналітичне подання кривої Без'є 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 Без'є/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 Без'є:

$$B_{P_0P_1\ldots P_n}(t) = (1-t)B_{P_0P_1\ldots P_{n-1}}(t) + tB_{P_1P_2\ldots P_n}(t).$$

Криві Без'є мають наступні властивості:

1. Інваріантність відносно афінних перетворень.
2. Інваріантність відносно лінійних замін параметризації t=(x-a)/(b-a)*b.
3. Крива Без'є належить опуклій оболонці опорних точок (виходить з геометричного способу побудови). *Висновок:* Якщо всі опорні точки знаходяться на одній прямій, то крива Без'є вироджується у відрізок, що з'єднує ці точки.
4. Крива Без'є 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.

10.2.2. 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 Без'є, 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 Без'є залежить від всіх визначаючих 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 Без'є 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:

$$N_i^1(t) = \begin{cases} 1, & x_i \le t \le x_{i+1} \\ 0, & \text{інакше} \end{cases},$$
$$N_i^k(t) = \frac{(t - x_i) N_i^{k-1}(t)}{x_{i+k-1} - x_i} + \frac{(x_{i+k} - t) N_{i+1}^{k-1}(t)}{x_{i+k} - x_{i+1}}. \qquad (10.5)$$

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 $N_i^k \ge 0$;
- окрім k=1, всі базисні функції мають рівно один максимум;
- максимальний порядок кривої дорівнює кількості вершин визначаючого багатокутника;

- крива володіє властивістю зменшення варіації. Крива перетинає будь-яку пряму не частіше, ніж її визначаючий багатокутник;
- загальна форма кривої повторює форму визначаючого 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 типа 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 функції (рис. 10.6).

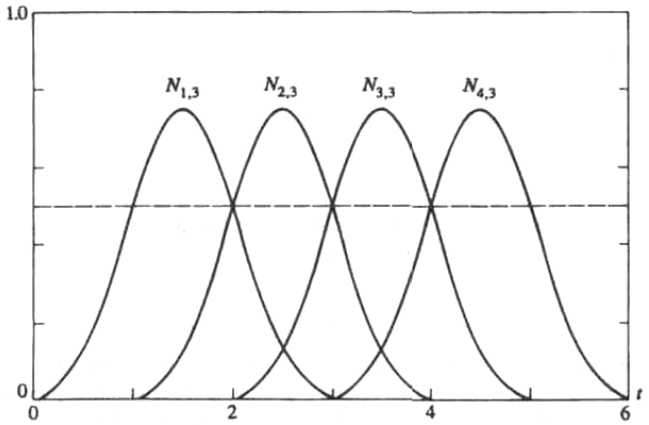


Рис. 10. 6. Базисні функції періодичного рівномірного B-сплайну, n+1=4, k=3, [X]=[0 1 2 3 4 5 6]

У відкритого рівномірного вузлового вектора кількість однакових вузлових значень в 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 же, як і криві Без'є. 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 Без'є. 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 Без'є – B-сплайн. Відповідні базисні функції зображені на рис. 10.7.

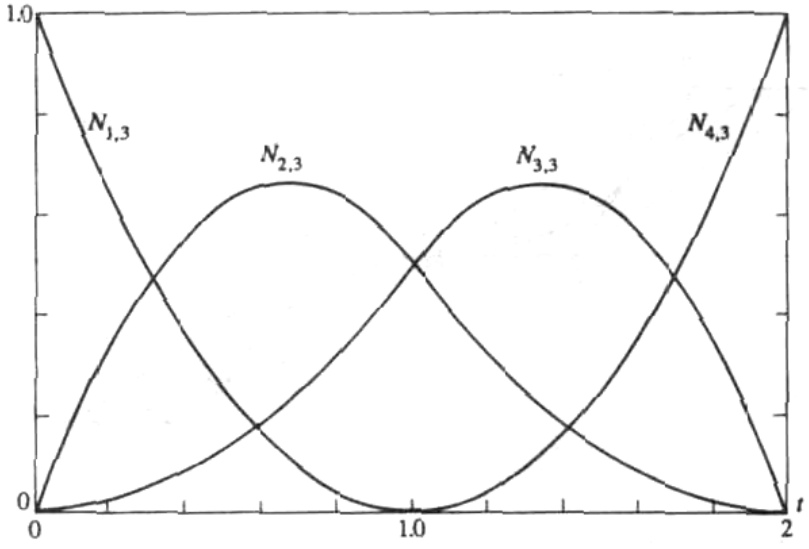


Рис. 10.7. Базисні функції відкритого рівномірного B-сплайну, n+1=4, k=3, [X]=[0 0 0 1 2 2 2]

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. 10.8 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, при суміщених вузлових значеннях в однієї з функцій з'являється злам. На рисунку видно, що положення зламу залежить від розташування суміщеного значення у вузловому векторі.

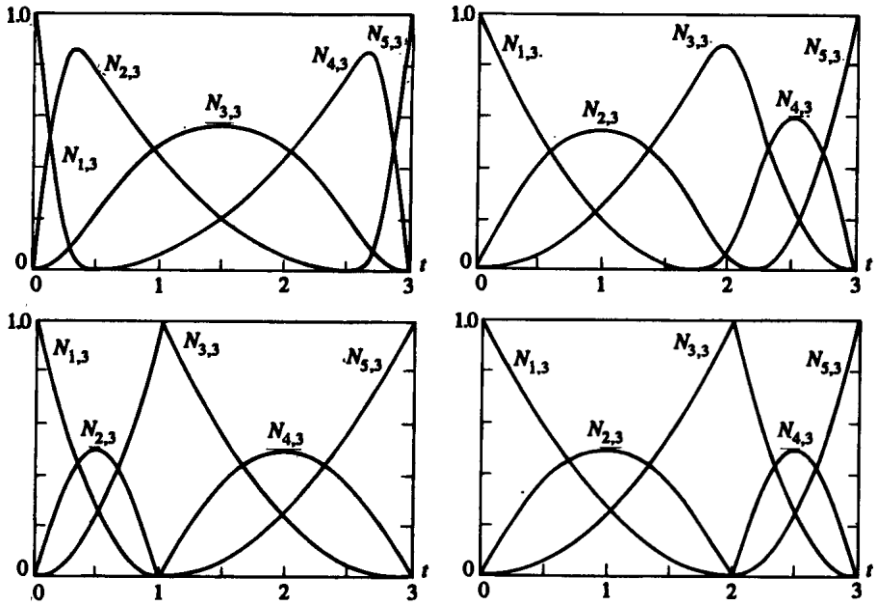


Рис. 10. 8. Функції нерівномірного базису для n+1=5, k=3,

[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 Без'є.

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 Без'є. 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. 10.9 зображені три відкритих B-сплайни різного порядку, задані одним набором з чотирьох вершин. Крива четвертого порядку – це крива Без'є – 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 визначаючим 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.

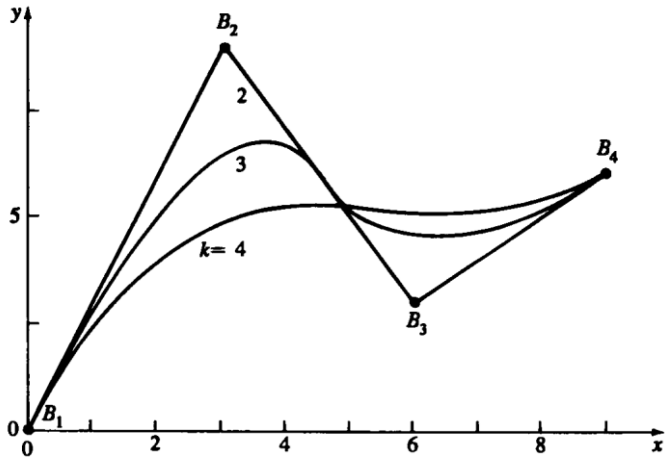


Рис. 10.9. Залежність форми відкритого B-сплайну від його порядку

Щоб сегменти кривої «притягувалися» до вершин визначаючого багатокутника необхідно повторювати необхідну вершину декілька разів при розрахунку B-сплайну. Чим більше разів повторюється вершина, тим ближче до неї «притягується» крива, і при кратності $k-1$ вона співпадає з вершиною визначаючого багатокутника.

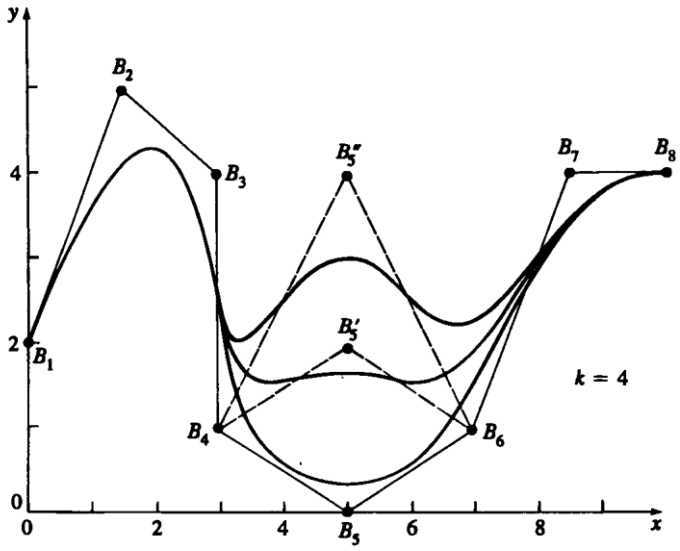


Рис. 10.10. Локальна корекція B-сплайну

На рис. 10.10 показані три B-сплайни четвертого порядку. Кожен визначаючий 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.

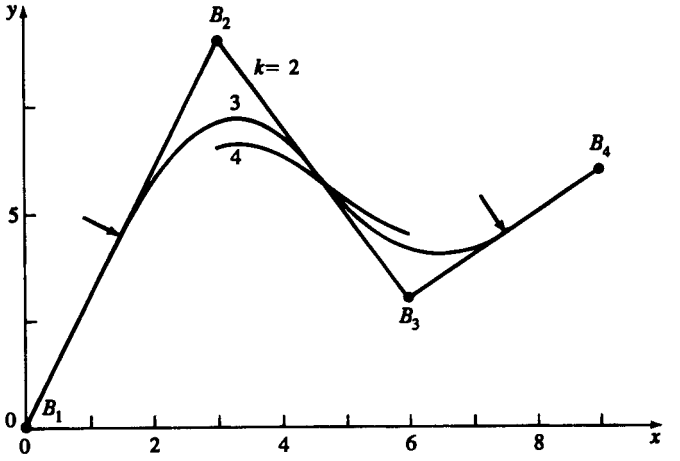


Рис. 10.11. Залежність форми періодичного B-сплайну від його порядку

Тепер розглянемо періодичні B-сплайни. На рис. 10.11 показані три періодичних B-сплайни різного порядку. Всі криві визначені тими самими вершинами, що і для відкритих B-сплайнів (див. рис. 10.9). Для k=2 B-сплайн знову збігається з визначаючим багатокутником. Відзначимо, однак, що у періодичного B-сплайну при 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. 10.9) показує, що криву можна визначити на повному діапазоні параметра, задаючи кратні вузлові значення на кінцях векторів. При цьому крива розтягується до кінців багатокутника.

В розглянутому випадку крива четвертого порядку знову складається з єдиного кубічного сегмента; крива третього порядку – з двох параболічних сегментів, з'єднаних в середині другого ребра з неперервністю C^1; крива 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 при розрахунку періодичного B-сплайну дає той самий ефект, що і для відкритих B-сплайнів.

Тепер розглянемо нерівномірні B-сплайни. На рис. 10.12 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. 10.7. Нижня крива третього порядку побудована з нерівномірним вузловим вектором [0 0 0 1 1 3 3 3]. Її базисні функції – на рис. 10.8.

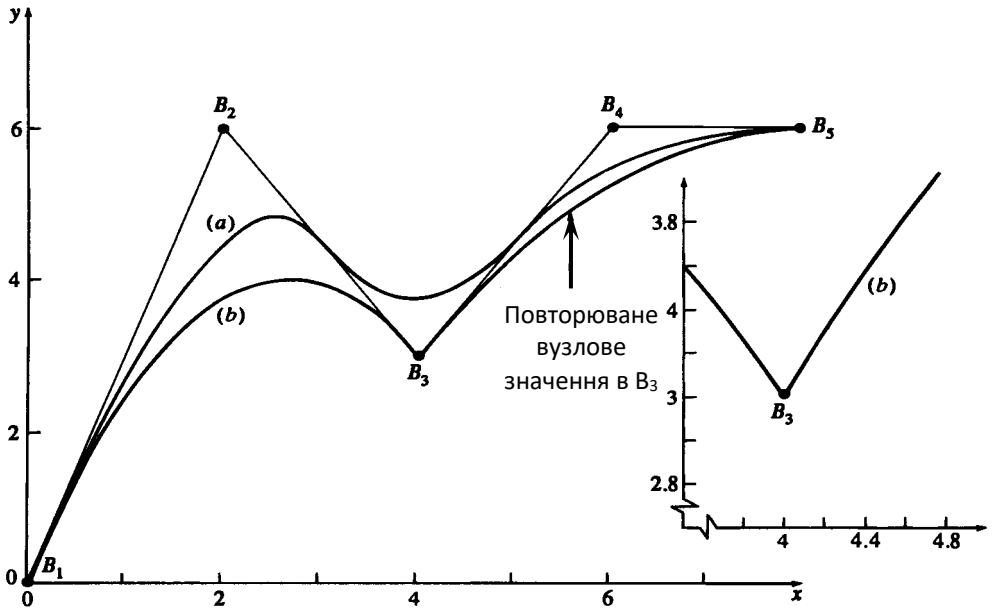


Рис. 10.12.Нерівномірні B-сплайни, k=3

З рис. 10.12 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. 10.12) 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 при невеликій зміні відносної відстані між вершинами.

10.2.3. РАЦІОНАЛЬНІ B-СЛАЙНИ

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 Без'є, 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, використовувані в комп'ютерній графіці та проектуванні.

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

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

$$P(t) = \sum_{i=1}^{n+1} B_i^h N_i^k(t), \qquad (10.6)$$

де B_i^h – вершини багатокутника для нераціонально 4D B-сплайна в чотирьохмірному просторі, $N_i^k(t)$ – функція базису нераціонального B-сплайну з рівняння (10.5).

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

$$P(t) = \frac{\sum_{i=1}^{n+1} B_i h_i N_i^k(t)}{\sum_{i=1}^{n+1} h_i N_i^k(t)} = \sum_{i=1}^{n+1} B_i R_i^k(t), \quad (10.7)$$

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

$$R_i^k(t) = \frac{h_i N_i^k(t)}{\sum_{i=1}^{n+1} h_i N_i^k(t)}. \quad (10.8)$$

Тут $h_i \geq 0$ для всіх i.

Як видно з рівнянь (10.6)-(10.8), 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:

$$\sum_{i=1}^{n+1} R_i^k(t) \equiv 1; \quad (10.9)$$

- 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 визначаючого багатокутника;
- раціональний B-сплайн володіє властивістю зменшення варіації;
- загальна форма раціонального B-сплайну повторює контури визначаючого багатокутника;
- будь-яке проективне перетворення раціонального B-сплайну виконується відповідним перетворенням вершин визначаючого багатокутника, тобто крива інваріанта відносно проектного 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 (10.7) і (10.8) надають додаткові можливості вигину кривої; h=1 називається афінним простором і відповідає фізичному простору. При h=1 крива раціонального B-сплайну співпадає з кривою нераціонального.

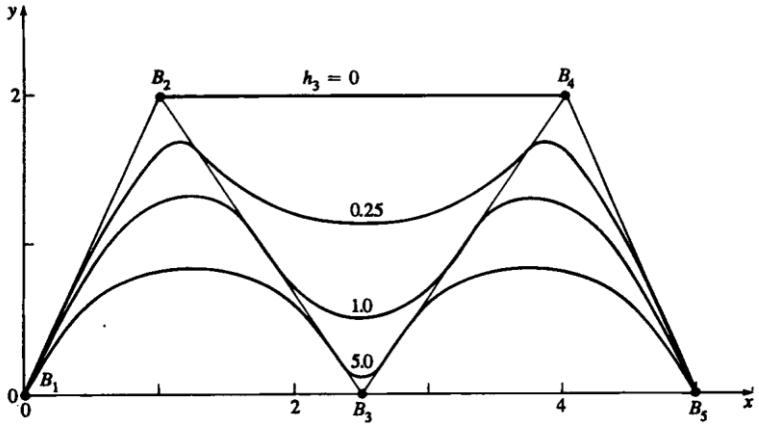


Рис. 10.13. Раціональні B-сплайни для n+1=5, k=3 з відкритим вузловим вектором [X]=[0 0 0 1 2 3 3 3], [H]=[1 1 h_3 1 1]

Раціональний B-сплайн для h_3=1 (рис. 10.13) 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 визначаючого багатокутника B_2 і B_4 з'єднані прямою. При збільшенні h_3 також зростає R_3^3 і, внаслідок рівняння (10.9) R_2^3 та R_4^3зменшуються. На рис. 10.13 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 при 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 Без'є і при якому порядку?
8. Що таке раціональний B-сплайн? Які його переваги?

11. Моделювання поверхонь

В комп'ютерній графіці 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 замітаємими. Третя група методів заснована на побудові поверхонь за заданою множиною точок. Методи цієї групи розглянемо в цьому розділі більш детально.

11.1. 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. 11.1).

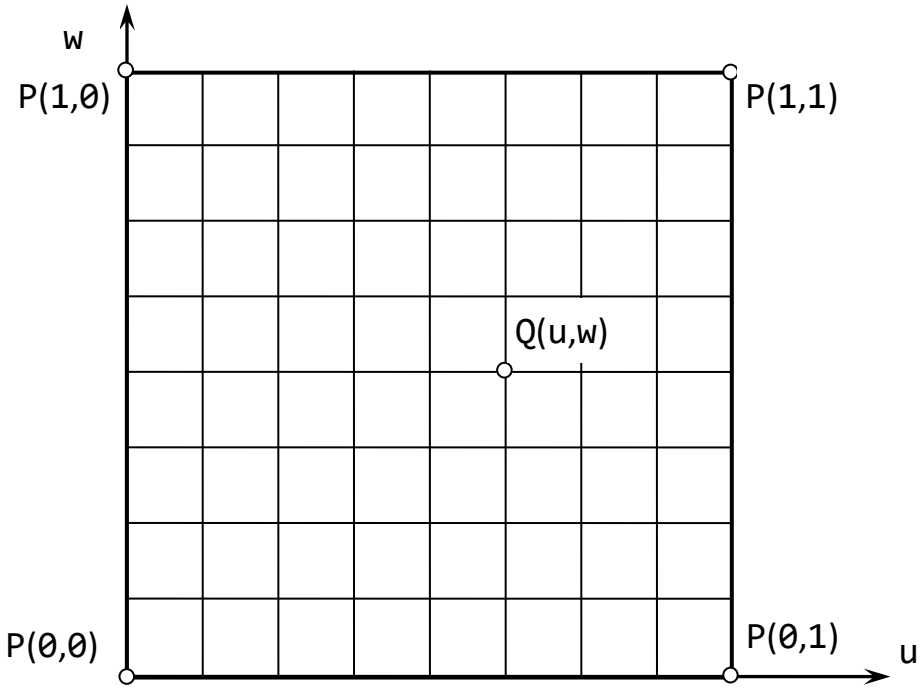


Рис. 11.1. Білінійна інтерполяція в параметричному просторі

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:

$$Q(u,w) = \begin{bmatrix} 1-u & u \end{bmatrix} \begin{bmatrix} P(0,0) & P(0,1) \\ P(1,0) & P(1,1) \end{bmatrix} \begin{bmatrix} 1-w \\ w \end{bmatrix}. \qquad (11.1)$$

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

Рівняння (11.1) 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. 11.2). На рисунку визначаючі точки лежать на кінцях протилежних діагоналей одиничного куба. В результаті ми отримали гіперболічний параболоїд.

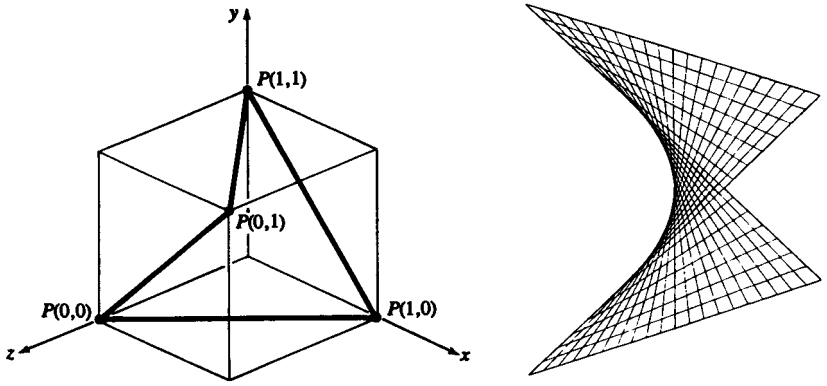


Рис. 11.2. Білінійна поверхня з визначаючими точками на кінцях протилежних діагоналей одиничного куба

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

11.2. Поверхні Без'є

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

$$Q(u,w) = \sum_{i=0}^{n} \sum_{j=0}^{m} B_{i,j} J_i^n(u) K_j^m(w), \qquad (11.2)$$

де $J_i^n(u)$ і $K_j^m(w)$ – базисні функції Бернштейна в параметричних напрямках u та w. Як і для кривих Без'є, 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 (11.2) є вершинами визначаючої полігональної сітки (рис. 11.3). Індекси n і m на одиницю менші кількості вершин багатогранника у напрямках u та w відповідно. Для чотирибічних частин поверхні визначаюча полігональна сітка повинна бути топологічно прямокутною, тобто повинна мати однакову кількість вершин у кожному «ряді».

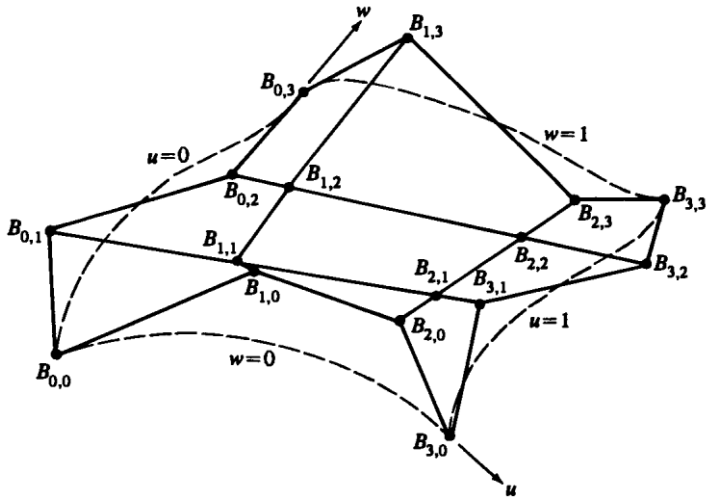


Рис. 11.3. Поверхня Без'є та вершини характеристичного багатогранника

Як і для кривих Без'є 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 визначаючого багатогранника у цьому напрямку;
- гладкість поверхні у кожному параметричному напрямку на дві одиниці менше кількості вершин визначаючого багатогранника у цьому напрямку;
- у загальному вигляді поверхня відтворює форму визначаючої полігональної сітки;
- співпадають тільки кутові точки визначаючої полігональної сітки та поверхні;
- поверхня міститься усередині опуклої оболонки визначаючої сітки;
- поверхня інваріантна по відношенню до афінного перетворення.

Розглянемо визначаючу полігональні сітку для бікубічної поверхні Без'є розміром 4×4 (рис. 11.4).

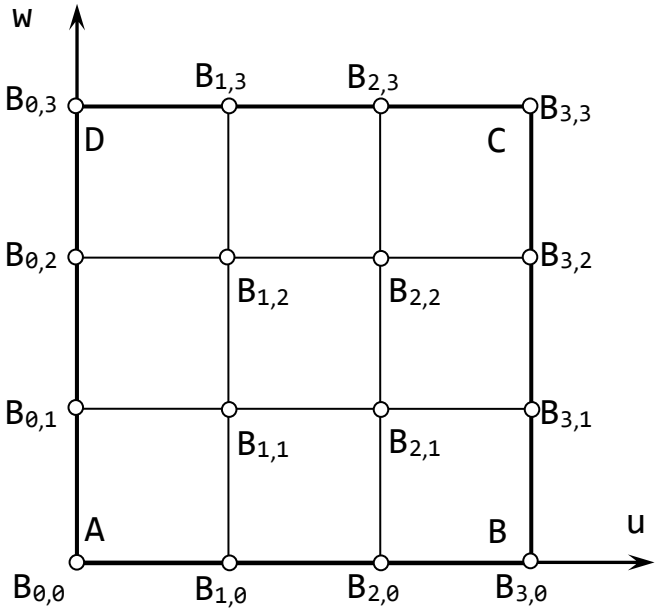


Рис. 11.4. Схема визначаючої полігональної сітки 4×4 для поверхні Без'є

Кожна з граничних кривих поверхні Без'є є кривою Без'є. 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 Без'є 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.

11.3. B-сплайн поверхні

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

$$Q(u,w) = \sum_{i=1}^{n+1} \sum_{j=1}^{m+1} B_{i,j} N_i^k(u) M_j^l(w), \qquad (11.3)$$

де $N_i^k(u)$ та $M_j^l(w)$ – 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 (11.3) $B_{i,j}$ – вершини визначаючої полігональної сітки. Для чотирикутних частин поверхні визначаюча 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 визначаючого багатогранника у цьому напрямку;
- гладкість поверхні у кожному параметричному напрямку на дві одиниці менше кількості вершин визначаючого 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 визначаючої 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 Без'є;

- при тріангуляції визначаюча 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.

11.4. 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 характеристичного багатограника, а $N_i^k(u)$ та $M_j^l(w)$ – нераціональні базисні функції B-сплайну в біпараметричних напрямках u та w.

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

$$Q(u,w)=\frac{\sum_{i=1}^{n+1}\sum_{j=1}^{m+1}h_{i,j}B_{i,j}N_i^k(u)M_j^l(w)}{\sum_{i=1}^{n+1}\sum_{j=1}^{m+1}h_{i,j}N_i^k(u)M_j^l(w)}=\sum_{i=1}^{n+1}\sum_{j=1}^{m+1}B_{i,j}S_{i,j}(u,w), \qquad (11.4)$$

де $B_{i,j}$ – є 3D-точками визначаючої 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:

$$S_{i,j}(u,w)=\frac{h_{i,j}N_i^k(u)M_j^l(w)}{\sum_{i1=1}^{n+1}\sum_{j1=1}^{m+1}h_{i1,j1}N_{i1}^k(u)M_{j1}^l(w)}. \qquad (11.5)$$

Зручно прийняти для $h_{i,j}\geq 0$ 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 характеристичного 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 проективного перетворення, тобто будь-яке проективне перетворення може бути застосоване до поверхні шляхом його застосування до визначаючої полігональної сітки. Ця умова більш строга, ніж для нераціональної B-сплайн поверхні;
- поверхня лежить всередині опуклої оболонки визначаючої полігональної сітки, що утворюється об'єднанням всіх опуклих оболонок k, l сусідніх вершин полігональної сітки;
- при тріангуляції визначаюча полігональна сітка створює плоску апроксимацію поверхні;
- якщо кількість вершин визначаючої полігональної сітки рівна порядку в кожному параметричному напрямку і дублювання внутрішніх вузлових величин нема, то раціональна B-сплайн поверхня є раціональною поверхнею Без'є.

З формул (11.4) і (11.5) 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 B-сплайн поверхонь і самі поверхні перетворюються в їх нераціональні еквіваленти.

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

Приклад бікубічної (k=l=4) раціональної B-сплайн поверхні та її визначаючої полігональної сітки для $h_{1,3}=h_{2,3}=0,1,5$ наведений на рис. 11.5.

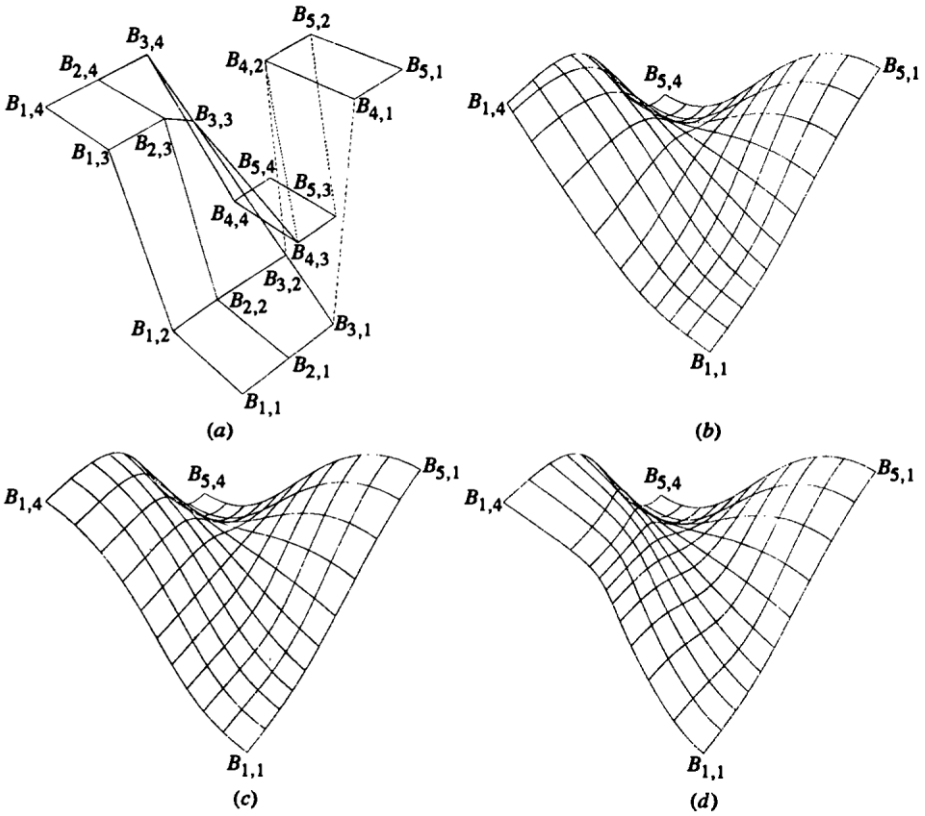


Рис. 11.5. Раціональні B-сплайн поверхні з n+1=5, m+1=4, k=l=4. (a) – характеристичний багатогранник; (b) – $h_{1,3}=h_{2,3}=0$; (c) – $h_{1,3}=h_{2,3}=1$; (d) – $h_{1,3}=h_{2,3}=5$

Запитання для самоконтролю

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

12. 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). При такому підході 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. 12.1). Нижче розглянуті можливі способи подання полігональних моделей.

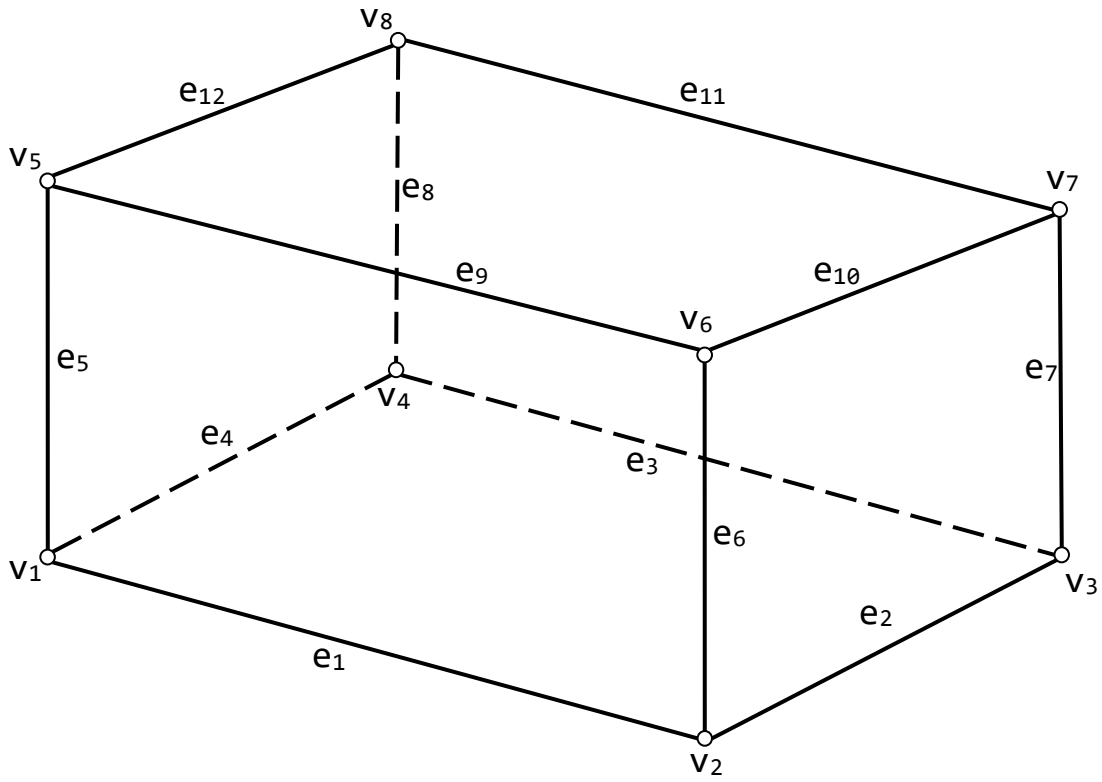


Рис. 12.1. Полігональна модель паралелепіпеда

12.1. 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.

12.2. 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.

12.3. 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.

12.4. 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. 12.2).

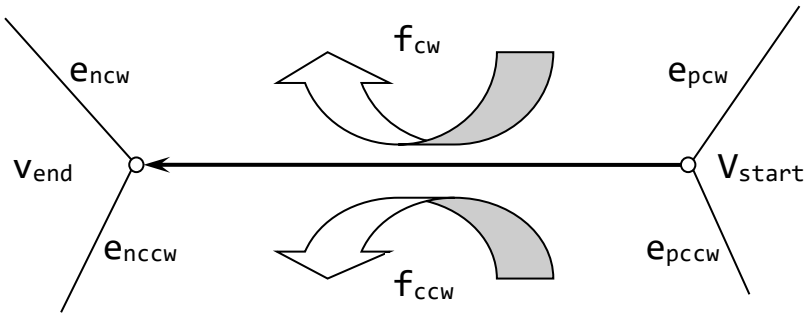


Рис. 12.2. «Крилате» подання

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 даному випадку ncw означає наступне ребро, що знаходиться в тій грані, де дане 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 даній 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 дане 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?

13. 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 розбиття геометричного простру 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, довільно переміщувана в цьому просторі (типовий запит в тривимірному просторі – це куля або брус). Регіональний пошук полягає у

вичитуванні (завдання звіту) або в підрахунку числа (завдання підрахунку) всіх точок всередині регіону (області) запиту.

13.1. Підрахунок кількості точок

Задача. Дані 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 заданих a, b, c і d (рис. 13.1)?

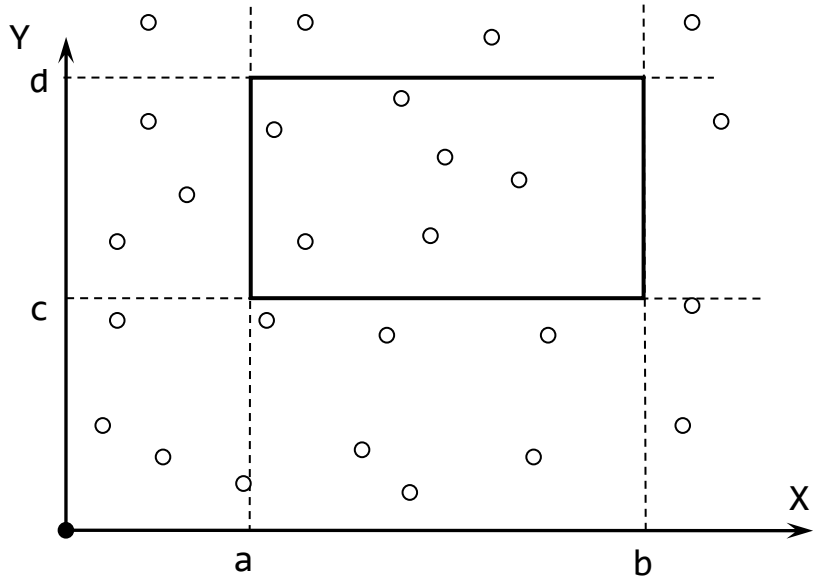


Рис. 13.1. Регіональний запит: скільки точок всередині прямокутника?

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 $x \le x(p)$ і $y \le y(p)$, тобто числа точок в лівому нижньому квадранті, який визначається вершиною p (рис. 13.2).

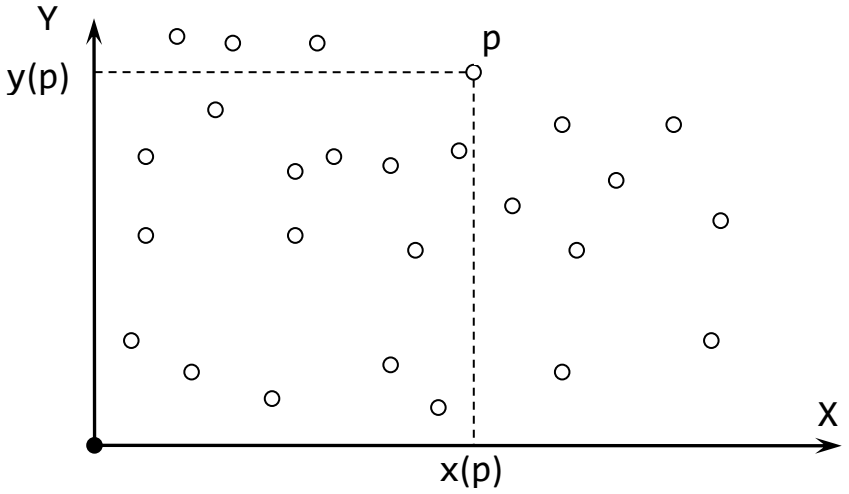


Рис. 13.2. Скільки точок «південно-західніше» p?

Поняття, з яким ми зустрілися тут – це *векторне домінування*. Говорять, що точка (вектор) v домінує над w, тоді і тільки тоді, коли для всіх індексів i вірна умова $v_i \ge w_i$. 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. 13.3.

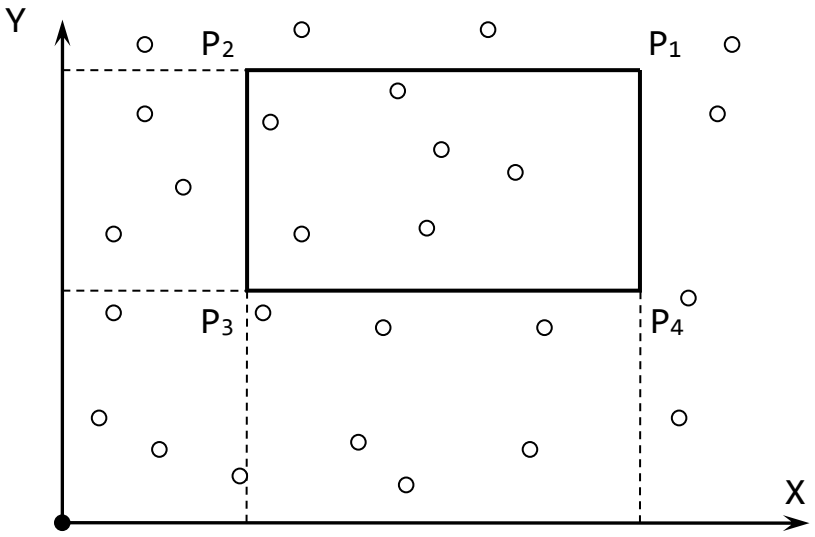


Рис. 13.3. Регіональний пошук у вигляді 4-х запитів про домінування

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.

13.2. Локалізація точки

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

Ми вже коротко розглядали задачі локалізації точки відносно прямокутника (див. розділ 6.1) та багатокутника (див. розділ 7.2). 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. 13.4).

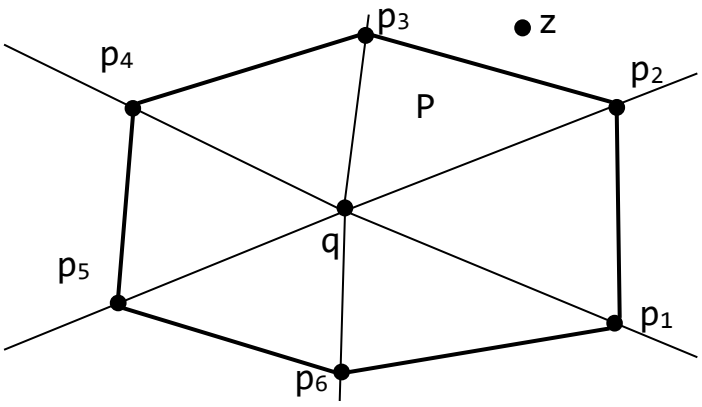


Рис. 13.4. Розбиття на клини для задачі про належність опуклому багатокутнику

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) при затраті 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. 13.5).

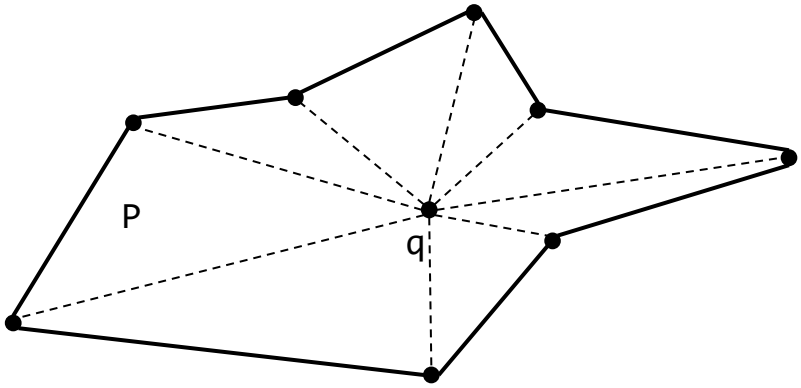


Рис. 13.5. Зірчастий багатокутник

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) при затраті 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»:

ВИПУКЛІСТЬ $\subset$ ЗІРКОВІСТЬ $\subset$ 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 9.3.

Запитання для самоконтролю

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

14. 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.

14.1. 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 при дихотомії, тобто принцип отримання приблизно рівної кількості елементів (точок) відносно кожної сторони розрізу.

Прямокутником будемо вважати таку область на площині, що визначається декартовим добутком $[x_1,x_2]\times[y_1,y_2]$, включаючи граничні випадки, 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.

Даний метод проілюстровано на прикладі (рис. 14.1) 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»).

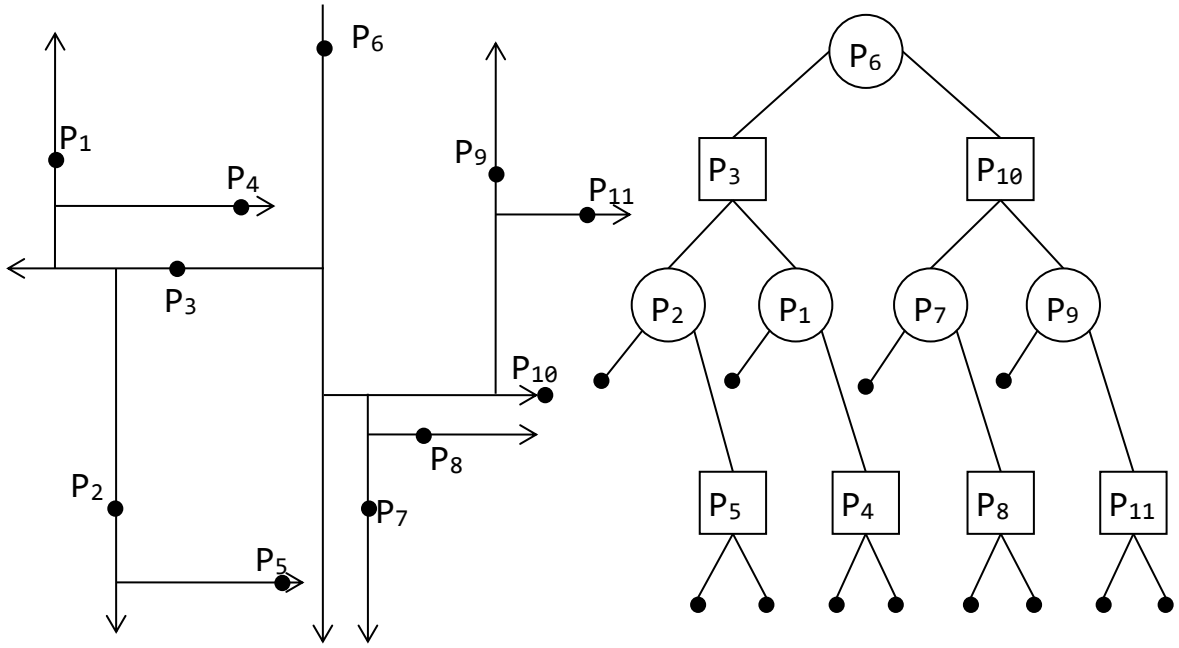


Рис. 14.1. Ілюстрація методу пошуку за допомогою двовимірного двійкового дерева

Пошук починається завжди з кореневого вузла і продовжується вглиб дерева. Ілюстрація регіонального пошуку наведена на рис. 14.2.

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).

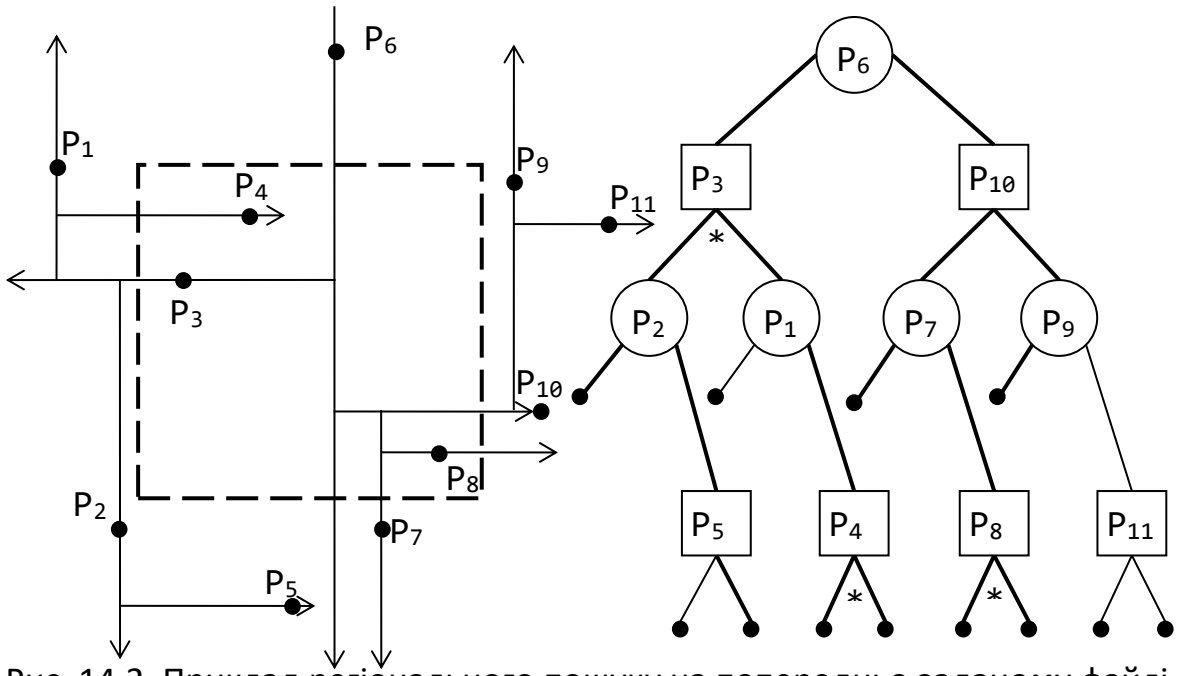


Рис. 14.2. Приклад регіонального пошуку на попередньо заданому файлі

Перетин 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. 14.3). Єдиний тип перетину, який є завжди продуктивним – це тип 4; всі інші можуть бути непродуктивними.

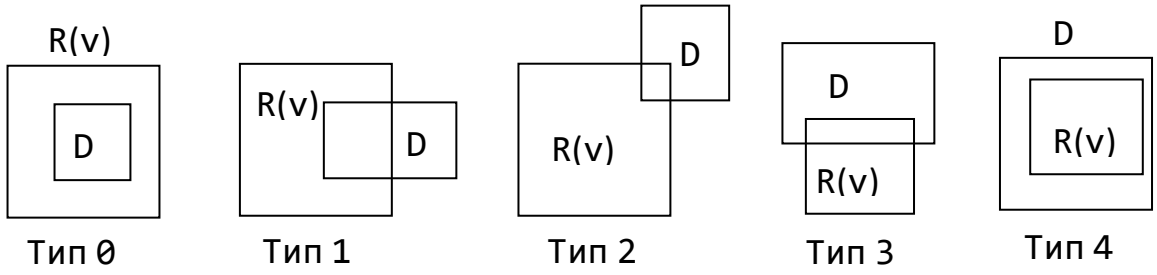


Рис. 14.3. Приклад різних типів перетинів

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.

14.2. 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 14.4 наведено квадро-дерево та зв'язана з ним квадратична декомпозиція простору.

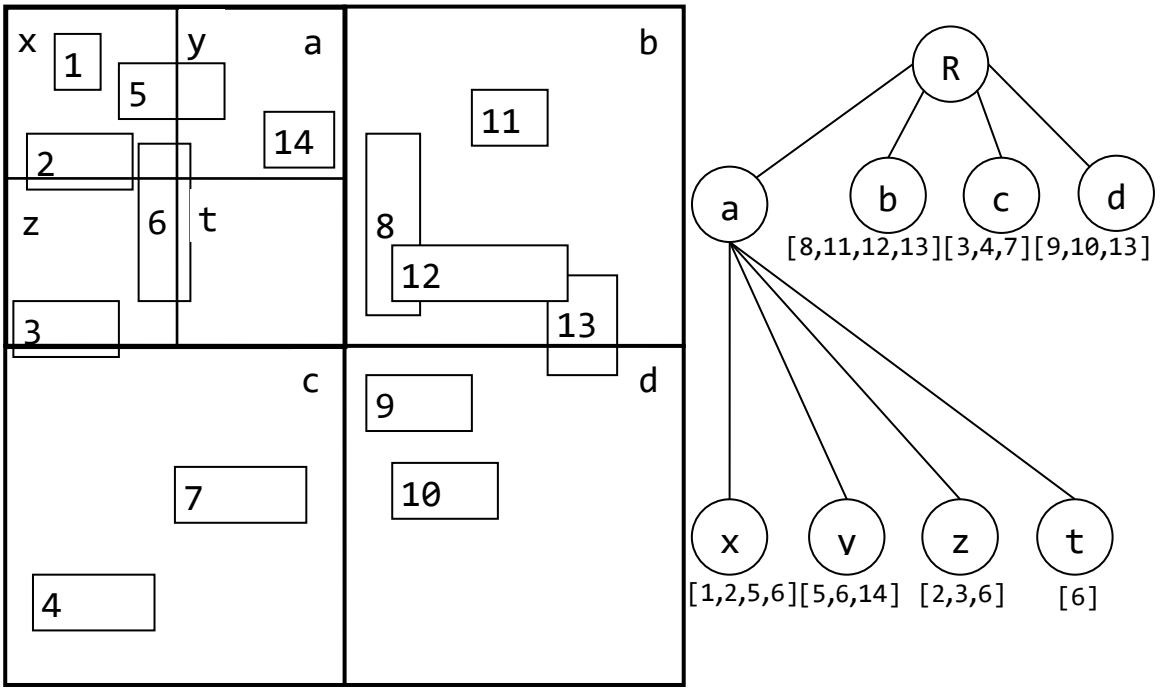


Рис. 14.4. Квадро-дерево

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 14.5 ілюструє обробку точкового запиту.

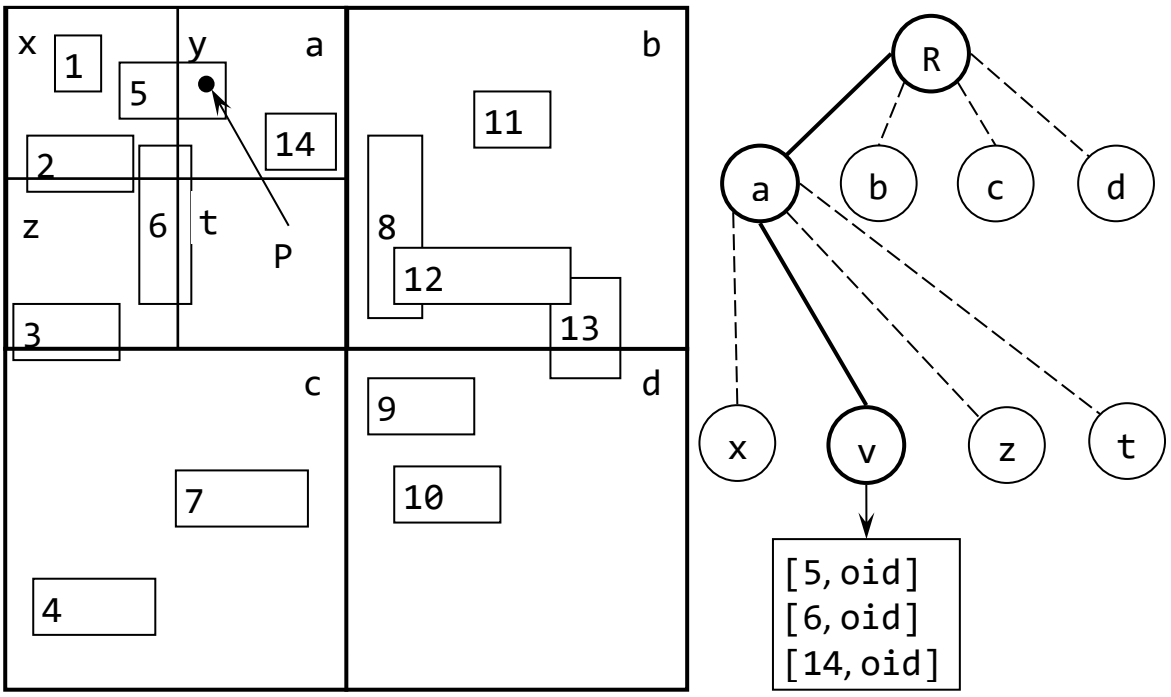


Рис. 14.5. Обробка точкового запиту в квадро-деревах

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 14.6 ілюструє вставку об'єктів у квадро-дерево. Вставка об'єкту 16 не призводить до розбиття квадрантів, а вставка об'єкту 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.

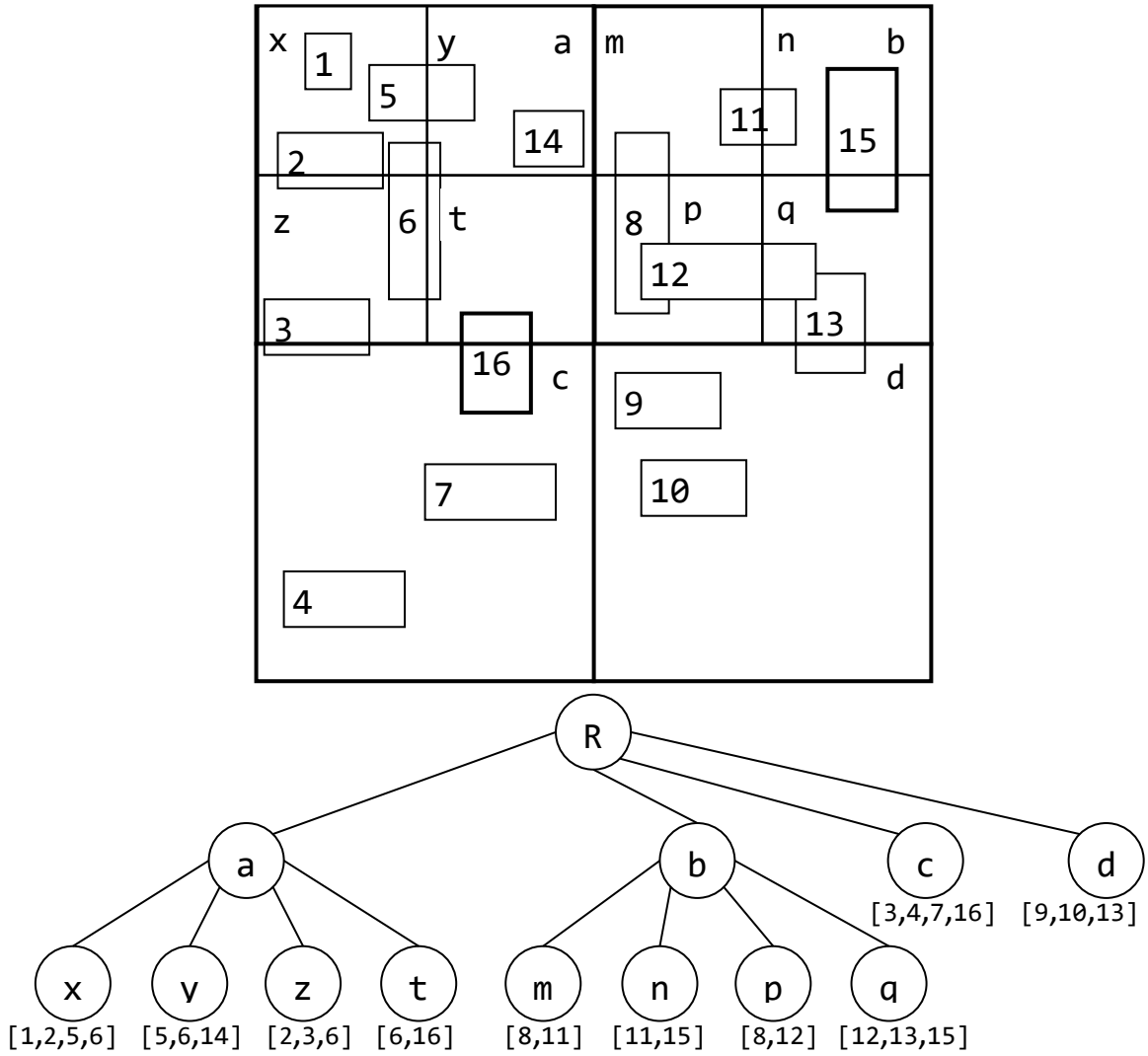


Рис. 14.6. Вставка об'єкта в квадро-дерево

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 при застосуванні 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.

14.2.1. 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. 14.7). Існує можливість сортування комірок за лексикографічним порядком.

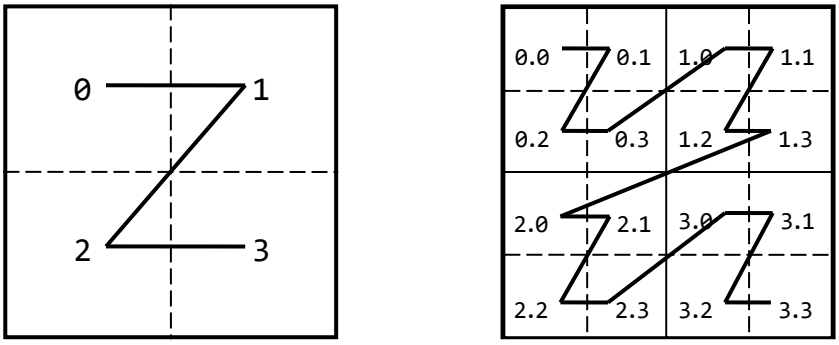


Рис. 14.7. Крива z-порядку

Крива Хілберта (Hilbert) – має форму П. На відміну від кривої z-порядку, крива Хілберта складається з частин однорідної довжини, це виключає переходи при скануванні від однієї комірки до іншої, якщо вони знаходяться одна від одної на значній відстані (рис 14.8).

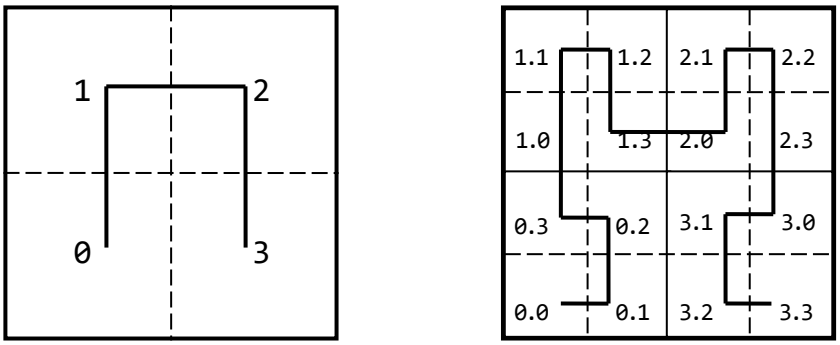


Рис. 14.8. Крива Хілберта

На рисунках 14.4-14.6 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.

14.2.2. 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 при видаленні чи вставці об'єктів у колекцію. Але подібна схема має проблему надмірності.

14.3. 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'єкту, oid – ідентифікатор об'єкту;
- корінь має не менш ніж два виходи;
- усі листові сторінки знаходяться на одному рівні.

На рисунку 14.9 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.

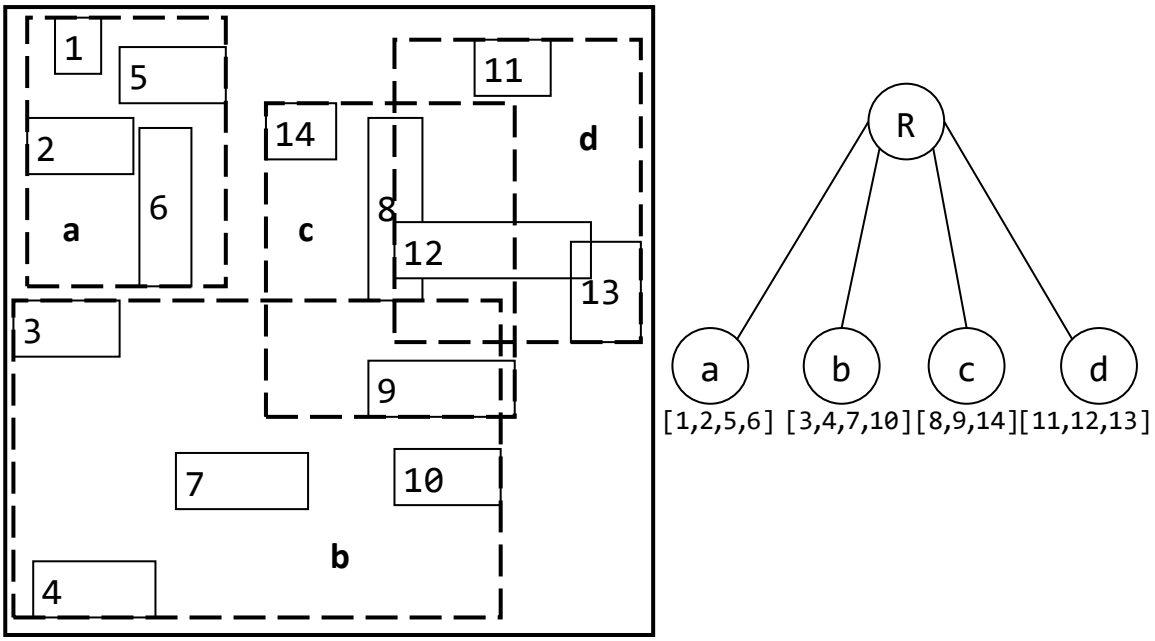


Рис. 14.9. R-дерево

Наведені вище 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 $\lfloor \log M(N) \rfloor - 1$.

14.3.1. Пошук в R-деревах

Нижче наведений 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. 14.10).

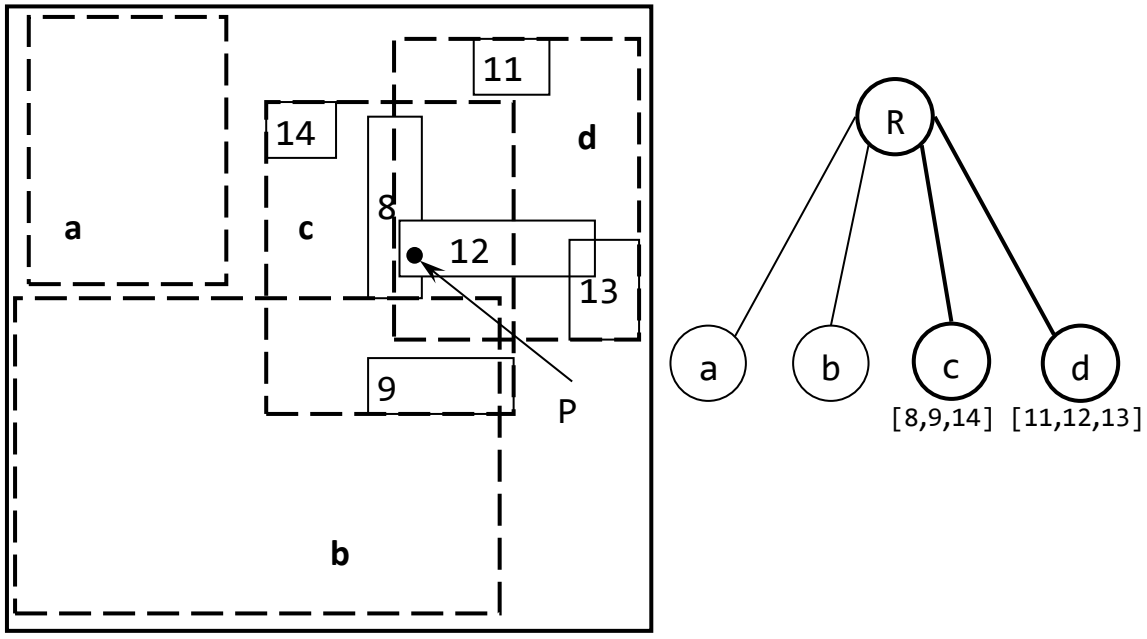


Рис. 14.10. Ілюстрація обробки точкового запиту в R-дереві

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

- в даному вузлі ні один з прямокутників не містить у собі точку P – пошук завершується. Дана 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 14.10 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 усе 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.

14.3.2. 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, то розглядати піддерева вузла до досягнення листової сторінки. Із декількох піддерев вибирається 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, при якому 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 14.11.

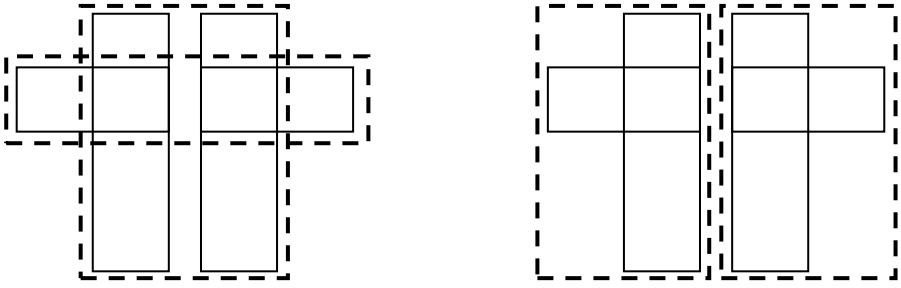


Рис. 14. 11. Розбиття з мінімальними областю та перекриттям

Метод вирішення «в лоб», при використанні 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.

Даний 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 при виконанні 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 простру є найбільшим, та розподілення елементів, що залишилися у групу, mbb якої потребує найменшого розширення області при вставці даного елементу. Даний алгоритм є більш простим та швидшим. Але, як було доведено в декількох експериментах, результат роботи даного алгоритму погіршує надмірний перетин граничних прямокутників груп.

14.3.3. 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 наведені два варіанта 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 даного методу зображені на рисунку 14.12.

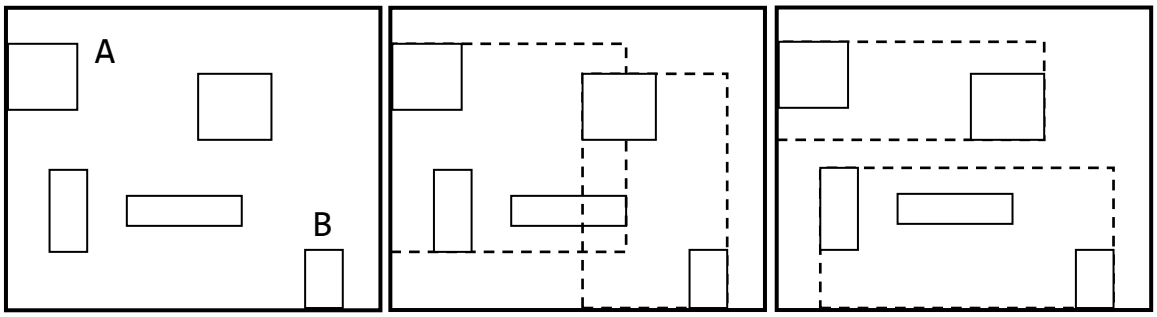


Рис. 14.12. Стратегії розбиття R-дерева та R*-дерева

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. 14.13).

На рисунку 14.13, що ілюструє стратегію реорганізації R*-дерев при заповненні 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.

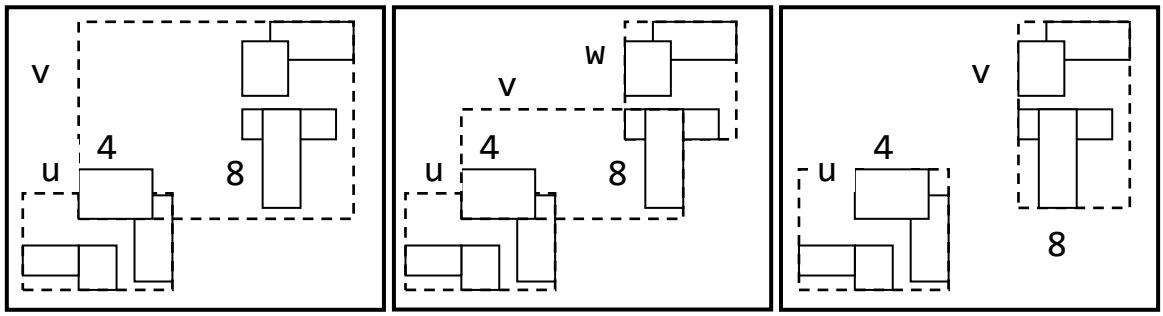


Рис. 14.13. Стратегія повторної вставки об'єкта: вставка об'єкта 8 (переповнення вузла v); розбиття R-дерева та реорганізація у R*-дереві

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.

14.3.4. 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.

14.4. Z-ВПОРЯДКОВАНІ ДЕРЕВА

На відміну від інших типів дерев, даний 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 попадає об'єкт o, та будуємо B+ дерево. Декомпозиція припиняється по досягненні максимальної глибини d.

Рисунок 14.14 ілюструє декомпозицію та апроксимацію об'єкта. Апроксимація об'єкта – список квадрантів {023, 03, 103, 12, 201, 210, 211, 300, 301, 302}.

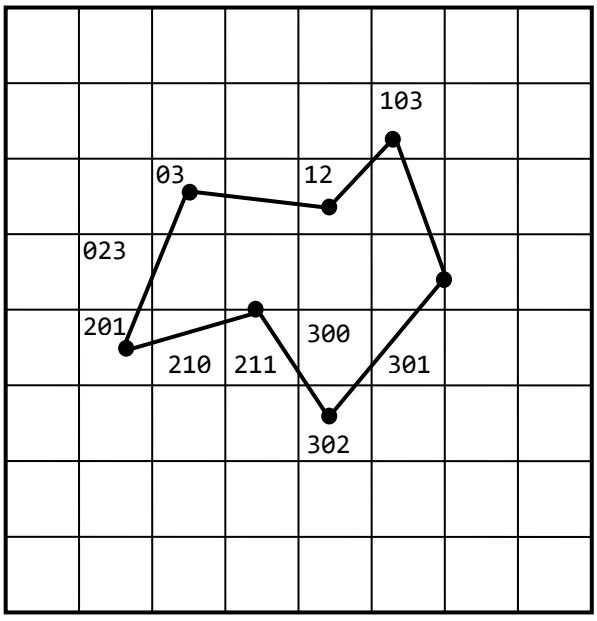
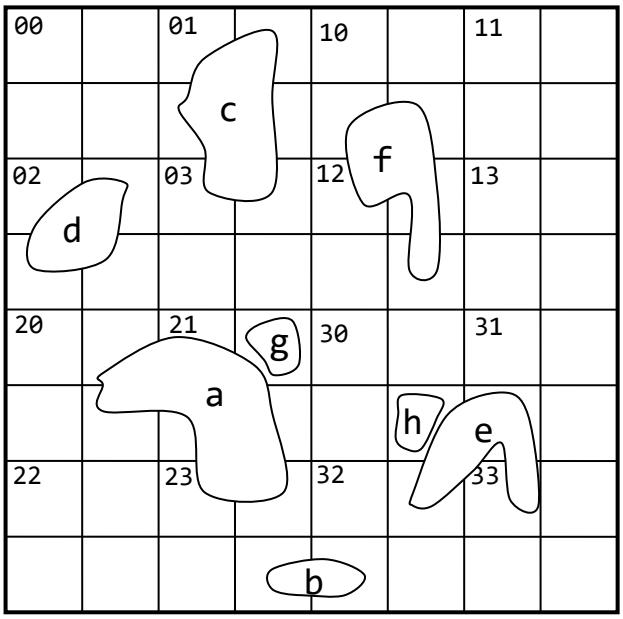


Рис. 14.14. Z-впорядкування та декомпозиція об'єкта

Колекція з восьми об'єктів разом з їх Z-впорядкованою декомпозицією наведена на рисунку 14.15. 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. 14.15. Набір об'єктів із Z-впорядкованою декомпозицією

На рисунку 14.16 наведено B+ дерево, яке отримано від Z-впорядкованого набору даних (рис. 14.15). 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 при Z-впорядкуванні. 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.

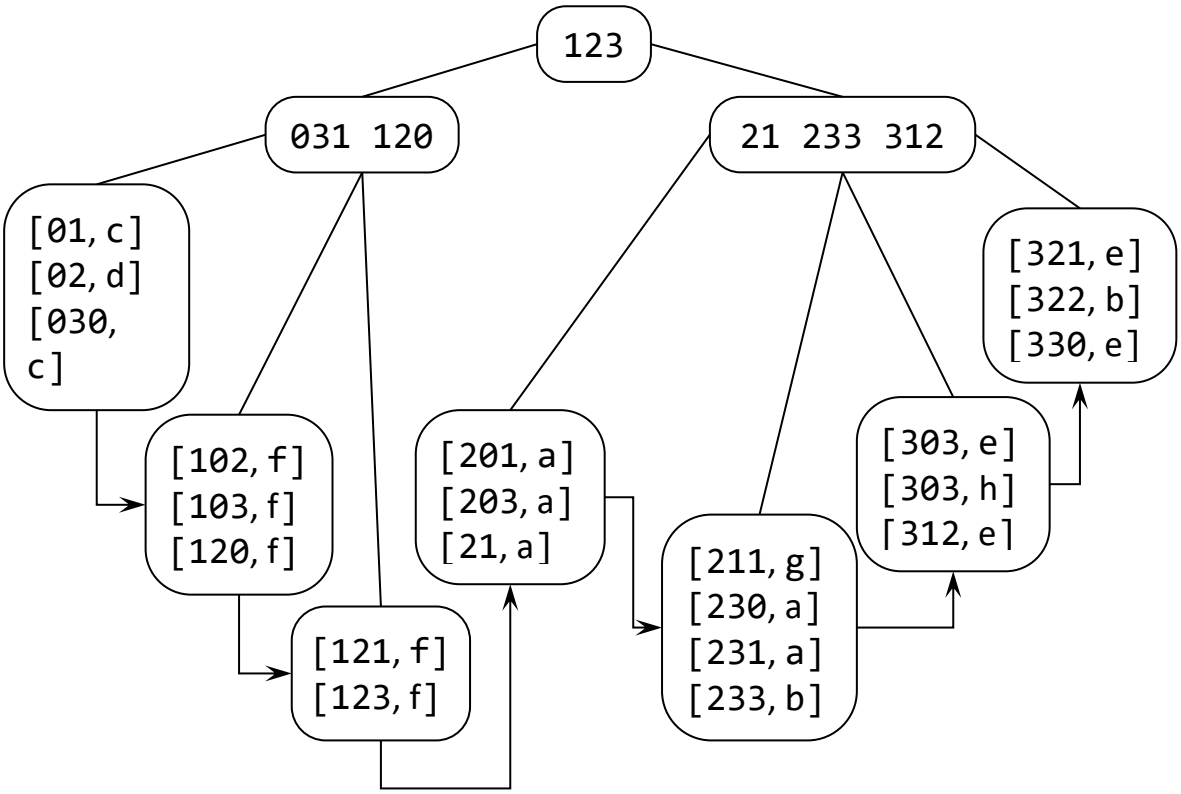


Рис. 14.16. B+ дерево побудоване на Z-впорядкованому наборі даних

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? Яки їх особливості та відмінності від R-дерев?
8. Які особливості Z-впорядкованих дерев?

15. Видалення невидимих ребер та граней

Однією з найважливіших задач тривимірної графіки є визначення, які частини об'єктів (ребра, грані), що знаходяться у тривимірному просторі, будуть видимі при заданому способі проектування, а які будуть закриті від спостерігача іншими об'єктами. В якості можливих видів проектування традиційно розглядаються паралельне і центральне (див. розділ 3).

Саме проектування відбувається на так звану картинну площину (екран): крізь кожну точку кожного об'єкта проводиться проекційний промінь (проектор) до картинної площини. Усі проектори створюють пучок або паралельних променів (при паралельному проектуванні), або променів, що виходять з однієї точки (центральне проектування). Перетин проектора з картинною площиною дає проекцію точки. Видимими будуть тільки ті точки, що розміщені вздовж напрямку проектування найближче до картинної площини. Усі три точки P_1, P_2 та P_3 (рис. 16.1) знаходяться на одному і тому ж проекторі, тобто проектуються в одну і ту ж точку картинної площини.

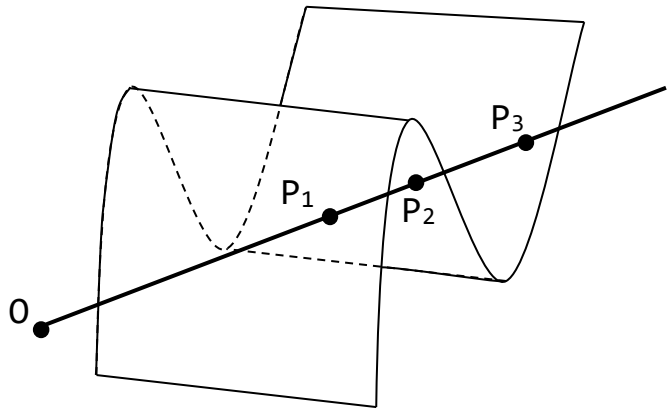


Рис. 15.1. Точки об'єкта на одному проекторі

Але оскільки точка P_1 лежить ближче до картинної площини, ніж точки P_2 і P_3 та закриває їх при проектуванні, 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, включаючи і методи, 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,Y). 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 проектування здійснюється на площину XOY. Проектування при цьому 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 закрашені 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 що працюють з проекціями 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 піксела 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 проекціями на картинну площину, виділяючи на ній області, відповідні проекціям 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 в 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 піксела тієї грані, яка є найближчою до нього уздовж напряму проектування. Зміна допуску 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 вона присутня 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 даний 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 даний піксел відповідає точці грані P, то швидше за все сусідні пікселі також відповідають точкам тієї ж грані;
- когерентність в просторі об'єктів – якщо даний 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 – тимчасова: грані, видимі в даному кадрі, швидше за все будуть видимі і в наступному; аналогічно грані, невидимі в даному кадрі, швидше за все будуть невидимі і в наступному.

Акуратне використання когерентності дозволяє помітно скоротити кількість виникаючих перевірок і помітно підвищити швидкодію алгоритму.

15.1. Відсікання нелицьових граней

Розглянемо багатогранник, для кожної грані якого заданий одиничний вектор зовнішньої нормалі (рис. 15.2). Неважко відмітити, що коли вектор нормалі грані n складає з вектором l, який задає напрям проектування, 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*.

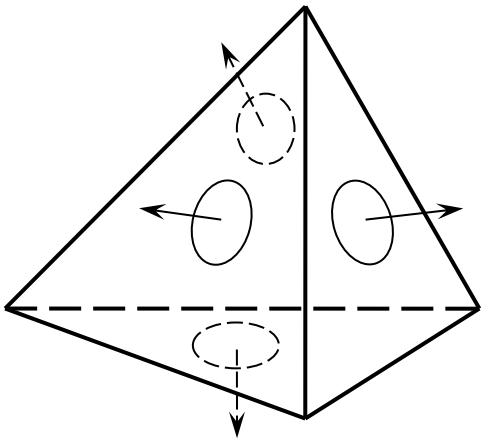


Рис. 15.2. Лицьові та нелицьові грані

При паралельному проектуванні умову на кут можна записати у вигляді нерівності $(n,l)<0$, оскільки напрям проектування від грані не залежить.

При центральному проектуванні з центром в точці c вектор проектування 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 дану грань, лежить центр проектування.

Оскільки при центральному проектуванні промінь проектування залежить від грані (і не залежить від вибору точки на грані), то лицьова грань може стати нелицьовою, а нелицьова лицьовою навіть при паралельному зсуві. При паралельному проектуванні зсув не змінює кутів і те, чи є грань лицьовою чи ні, залежить лише від кута між нормаллю до грані і напрямом проектування.

Відмітимо, що якщо по аналогії з визначенням належності точки багатокутнику, пропустити через довільну точку картинної площини промінь проектування 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 даних 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. 15.3).

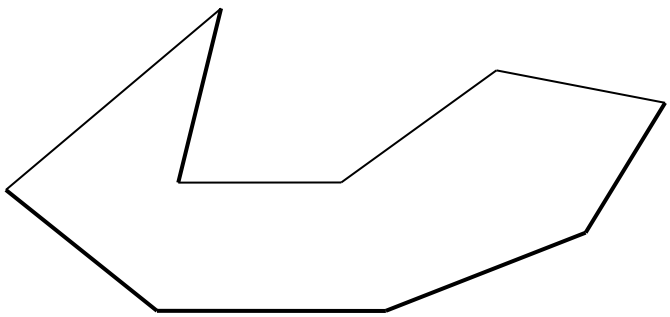


Рис. 15.3. Перекриття лицьової грані іншими лицьовими гранями

Ребра між нелицьовими гранями також завжди не видно. Проте ребро між лицьовою і нелицьовою гранями цілком може бути видимим.

15.2. 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. 15.4):

- 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 даною гранню не закриваються).

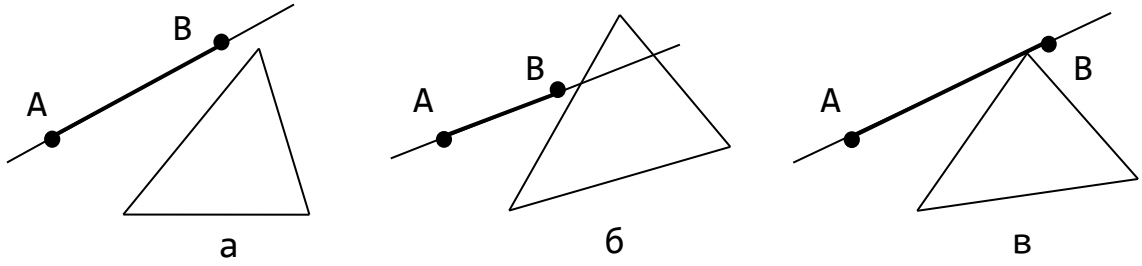


Рис. 15.4. Варіанти співвідношення грані і ребра

Розглянемо, як здійснюються ці перевірки.

Нехай задано 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`. Дану 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 проекція грані задається набором проекцій 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 проекції 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 при:

$$F_i F_{i+1} \le 0.$$

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 параметру 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 проекцією грані, окрім, можливо, однієї точки. Це може мати місце, коли:
   - пряма AB не перетинає ребра проекції (рис. 15.4, 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. 15.4, б);
   - пряма AB проходить через одну вершину, не зачіпаючи внутрішності трикутника (рис. 15.4, в).

Вочевидь, що в цьому випадку відповідна грань ніяк не може закривати собою ребро AB.

2. Проекція ребра повністю міститься всередині проекції грані (рис. 15.5, 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 проекції грані в двох точках і або $t_1<0<t_2<1$, або $0<t_1<1<t_2$ (рис. 15.5, 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. Проекція другої частини лежить поза проекцією грані і тому цією гранню не закривається.
4. Пряма AB перетинає ребра проекції грані в двох точках, причому $0<t1<t2<1$ (рис. 15.5, г). Якщо ребро AB лежить далі від картинної

площини, ніж відповідна грань, то воно розбивається на три частини, середня з яких відкидається.

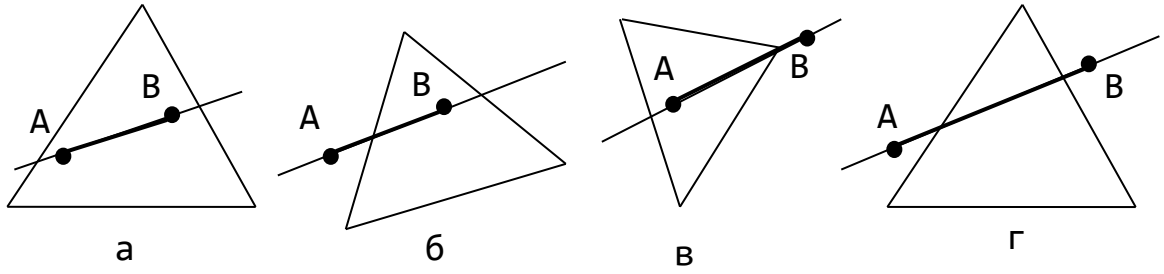


Рис. 15.5. Варіанти перетину ребра і грані

Для визначення того, що лежить ближче до картинної площини – відрізок AB (проекція якого лежить в проекції 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 відрізку 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 даного 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 проекції мають з даною клітинкою непустий перетин. Для перевірки довільного ребра на перетин з гранями відберемо спочатку всі ті клітинки, які проекція даного 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 проектуванні і визначенні клітинок, в які потрапляють проекції, то витрати на складання всіх списків також складають $O(N)$.

15.3. Метод трасування променів

Задача видалення невидимих граней є помітно складнішою, ніж задача видалення невидимих ліній, хоча б за загальним обсягом виникаючої інформації. Якщо практично всі методи, що слугують 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*), при якому для кожного піксела картинної площини визначається найближча до нього грань, для чого через цей піксел пропускається промінь, знаходяться всі точки його перетину з гранями і серед них вибирається найближча. Даний 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 піксела.

Ще одним безперечним плюсом метода є велика кількість методів оптимізації, що дозволяють працювати з сотнями тисяч граней і що забезпечують часові витрати порядку O(C log N), де C – загальна кількість пікселів на екрані і N – загальна кількість об'єктів в сцені. Більш того, існують методи, що забезпечують практичну незалежність часових витрат від кількості об'єктів.

15.4. Метод Z-буфера

Одним з найпростіших алгоритмів видалення невидимих граней і поверхонь є, метод Z-буфера (буфера глибини), де для кожного піксела, як і в методі трасування променів, знаходиться грань, найближча до нього уздовж напряму проектування, проте тут цикли по пікселам і по об'єктам міняються місцями:

```
for all objects
for all covered pixels
```

```
compare z
```

Поставимо у відповідність кожному пікселу (x,y) картинної площини окрім кольору c(x,y), що зберігається у відеопам'яті, його відстань до картинної площини уздовж напряму проектування z(x,y) (його глибину).

Для виводу на картинну площину довільної грані вона переводиться в растрове подання на картинній площині і потім для кожного піксела цієї грані знаходиться його глибина. У випадку, якщо ця глибина менша значення глибини, що зберігається в Z-буфері, піксел малюється і його глибина заноситься в Z-буфер.

Досить ефективним є поєднання растрової розгортки грані з виводом в Z-буфер. При цьому для обчислення глибини пікселів можуть застосовуватися інкрементальні методи, що вимагають всього декількох додавань на піксел.

Грань малюється послідовно рядок за рядком; для знаходження необхідних значень використовується лінійна інтерполяція (рис. 15.6).

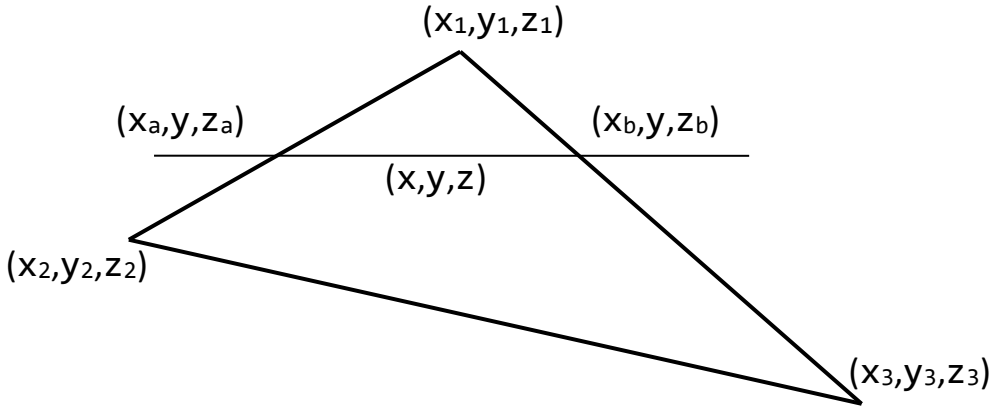


Рис. 15.6. Послідовний вивід грані методом Z-буфера

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 піксела 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 найбільшій 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, даний піксел накривається десятьма різними лицьовими гранями, то для кожного відповідного піксела кожній з цих десяти граней необхідно провести розрахунок кольору. При використанні 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 тимчасову когерентність.

Назвемо грань прихованою (невидимою) по відношенню до Z-буфера, якщо для будь-якого піксела картинної площини, що накривається цією гранню, глибина відповідного піксела грані не менше значення в Z-

буфері. Ясно, що виводити приховані грані не має 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 піксела, 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 підошві. 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 даному 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.

15.5. 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 при їх виведенні в цьому порядку виходило коректне зображення сцени. Для цього необхідно, щоб для будь-яких двох граней P і Q та з них, яка при виведенні 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 15.7, а). Тоді необхідно провести додаткове розбиття граней так, щоб множину граней, що вийшла після розбиття, вже можна було впорядкувати.

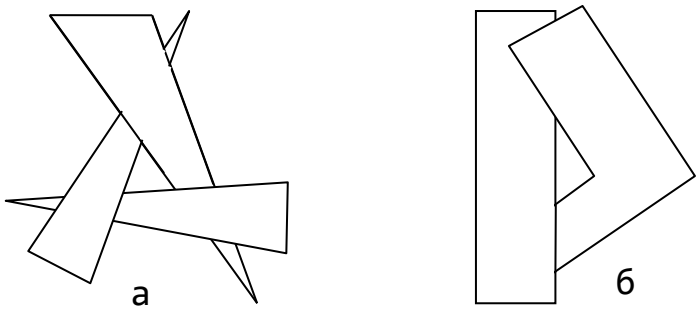


Рис. 15.7. Випадки неможливості впорядкування граней

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. 15.7, б).

15.5.1. 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. 15.8) – так, грань B буде помилково виведена раніше, ніж грань A; тому після сортування бажано перевірити порядок, в якому грані виводитимуться.

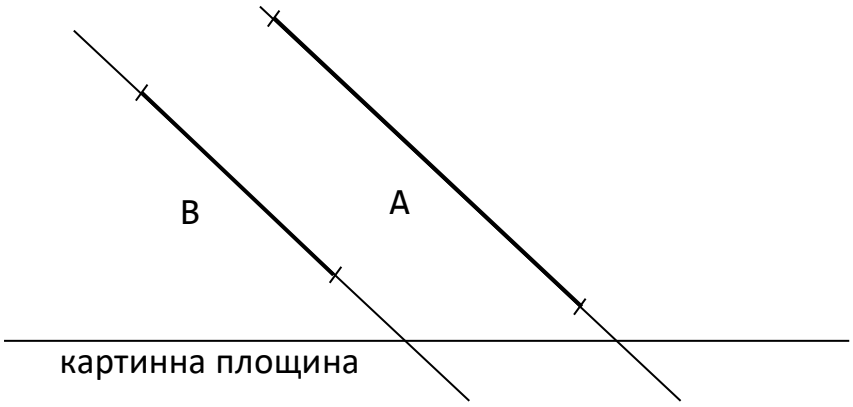


Рис. 15.8. Помилкове впорядкування граней

Пропонується наступний алгоритм цієї перевірки. Для простоти вважатимемо, що розглядається паралельне проектування по осі OZ.

Перед виведенням чергової грані P слід переконатися, що ніяка інша грань Q, яка стоїть в списку пізніше, ніж P, і проекція якої на вісь OZ перетинається з проекцією 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 проекції цих граней на вісь OX?
2. Чи перетинаються проекції цих граней на вісь OY?

Якщо хоч б на одне із двох питань одержана негативна відповідь, то проекції 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 мал. 15.9, де для кожної вершини вказана її глибина.

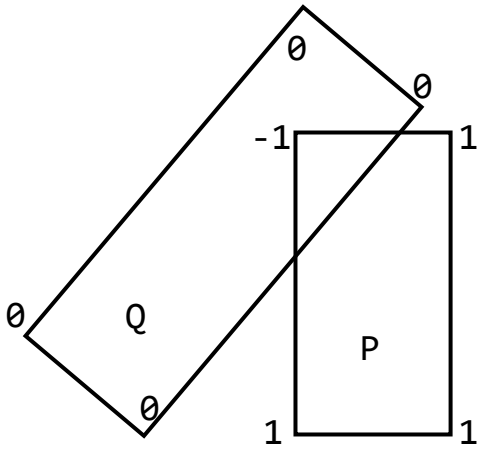


Рис. 15.9. Надмірне розбиття грані

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*). При виведенні 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.

15.5.2. 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 (кластера), 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 проектування).

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. 15.10).

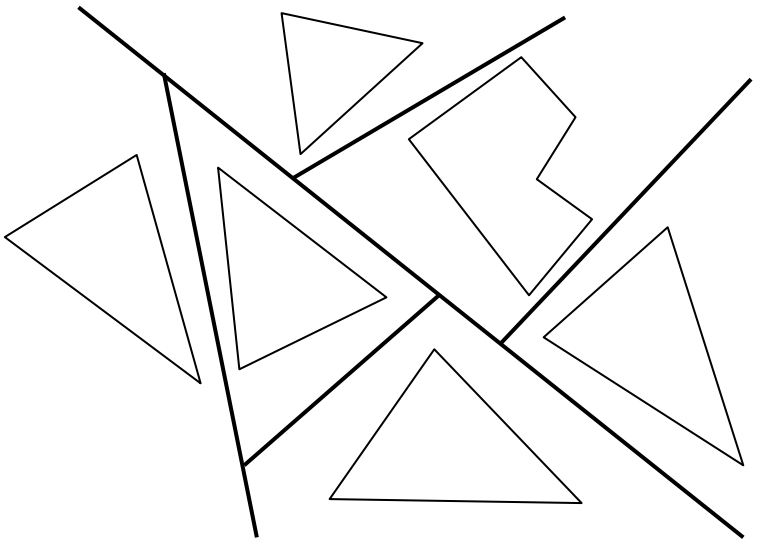


Рис. 15.10. Двійкове розбиття простору

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 при розбитті поміщаються у відповідні піддерева.

Для прикладу розглянемо сцену, що зображена на рис. 15.11.

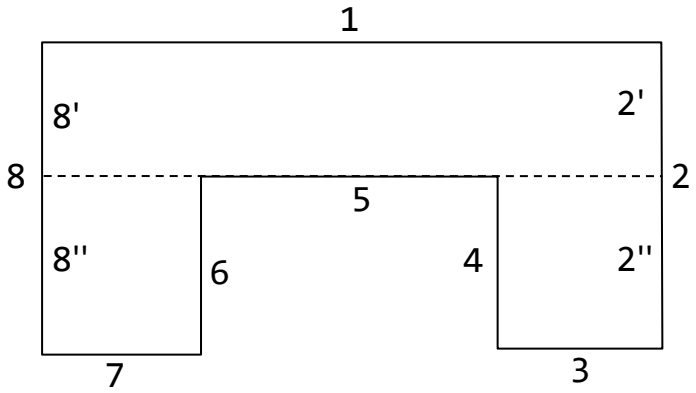


Рис. 15.11. Сцена для приклада двійкового розбиття простору

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 (мал. 15.12).

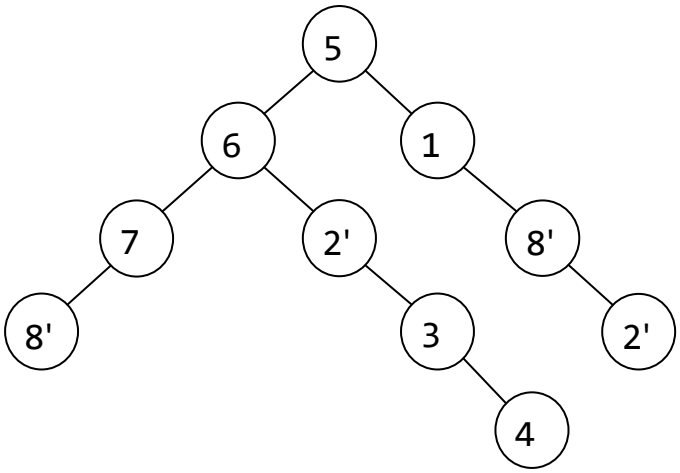


Рис. 15.12. BSP-дерево

Таким чином, процес побудови BSP-дерев полягає у виборі площини (грані), що розбиває, розбитті множини всіх граней на дві части (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 лежачть 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.

Нажаль, ці критерії, як правило, є взаємовиключаючими. 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 проектування (положення центру проектування, напряму проектування та ін.), що робить його досить зручним для побудови серій зображень однієї і тієї ж сцени з різних точок спостереження. Ця обставина привела 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 при роботі 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 даний 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.

16. Моделі подання кольору

Існують різні системи подання кольорів. Найбільш поширені та застосовні з них розглянемо в цьому розділі.

16.1. Моделі RGB та CMY

Найбільш простою є модель RGB, що застосовується в цілому ряді відеопристроїв. Це адитивна модель кольору: для отримання шуканого кольору базові кольори в ній складаються. Колірним простором є одиничний куб (рис. 16.1). Головна діагональ куба, що характеризується рівним внеском трьох базових кольорів, подає сірі кольори: від чорного (0, 0, 0) до білого – (1, 1, 1).

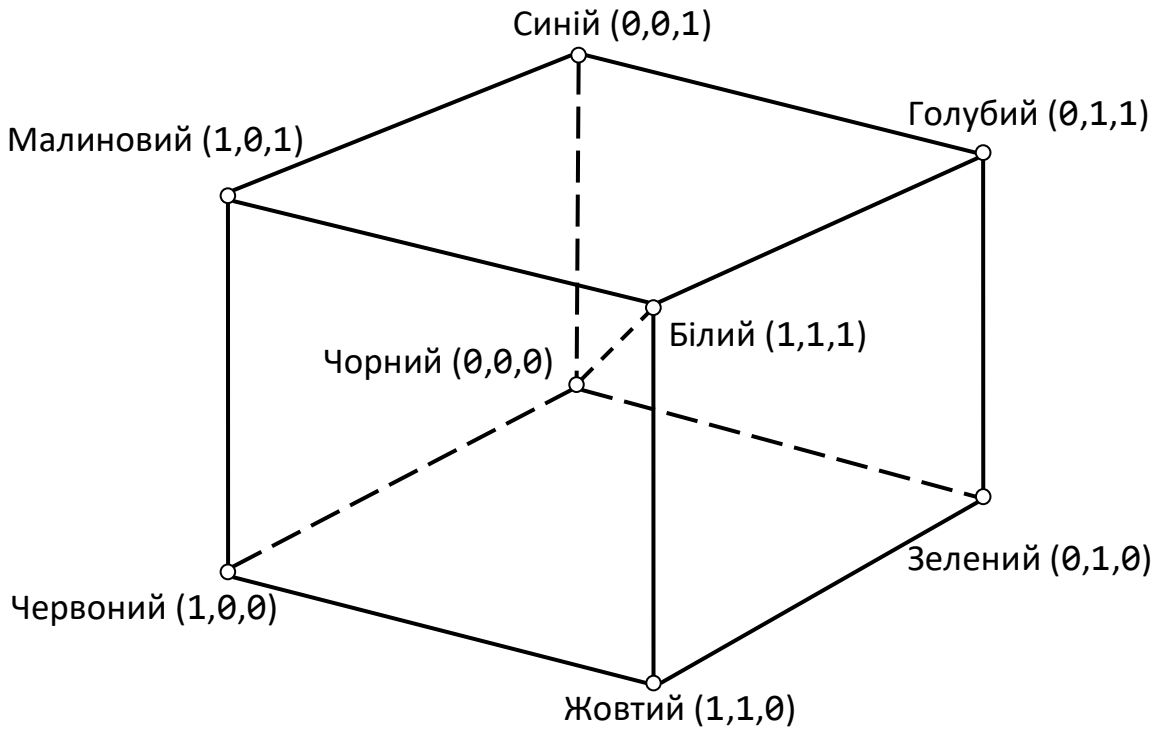


Рис. 16.1. Система RGB

Для друку частіше використовуються моделі CMY (*Cyan*, *Magenta*, *Yellow*) і CMYK (*Cyan*, *Magenta*, *Yellow*, *blacK*). Ці моделі на відміну від RGB є субтрактивними (точніше сказати, мультиплікативними) – для того, щоб одержати необхідний колір, базові кольори віднімаються з білого кольору.

Розглянемо, як це відбувається. Коли на поверхню паперу наноситься блакитний (*cyan*) колір, то червоний колір, падаючий на папір, повністю поглинається. Таким чином, блакитний фарбник немовби віднімає червоний колір з падаючого білого (що є сумою червоного, зеленого і синього кольорів). Аналогічно малиновий фарбник (*magenta*) поглинає

зелений, а жовтий фарбник (*yellow*) – синій колір. Поверхня, покрита блакитним і жовтим фарбниками, поглинає червоний і синій, залишаючи тільки зелену компоненту. Блакитний, жовтий і малиновий фарбники поглинають червоний, зелений і синій кольори, залишаючи в результаті чорний. Ці співвідношення можна подати у вигляді наступної формули:

$$\begin{pmatrix} C \\ M \\ Y \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} - \begin{pmatrix} R \\ G \\ B \end{pmatrix}. \quad (16.1)$$

Зворотні перетворення виконуються за формулою:

$$\begin{pmatrix} R \\ G \\ B \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} - \begin{pmatrix} C \\ M \\ Y \end{pmatrix}. \quad (16.2)$$

З цілого ряду причин (велика витрата дорогого кольорового чорнила, висока якість паперу, одержувана при друці на струменевих принтерах, небажані візуальні ефекти, що виникають за рахунок того, що при виводі точки трьох базових кольорів виходять з невеликими відхиленнями) використання трьох фарбників для отримання чорного кольору виявляється незручним. Тому його просто додають до трьох базових. Так виходить модель CMYK (*Cyan*, *Magenta*, *Yellow*, *blacK*). Для переходу від моделі CMY до моделі CMYK використовують наступні співвідношення:

$$\begin{aligned} K &= \min(C, M, Y)\,; \\ C &= C - K; \\ M &= M - K; \\ Y &= Y - K. \end{aligned} \quad (16.3)$$

Співвідношення (16.1)-(16.3) вірні лише у тому випадку, коли спектральні криві віддзеркалення для базових кольорів не перетинаються. Проте насправді, між відповідними спектральними кривими перетин існує, тому для точної передачі кольорів і відтінків зображення ці співвідношення мало застосовні. У телебаченні часто використовується модель YIQ. Перехід з системи RGB в YIQ здійснюється за наступними формулами:

$$\begin{pmatrix} Y \\ I \\ Q \end{pmatrix} = \begin{pmatrix} 0.30 & 0.59 & 0.11 \\ 0.60 & -0.28 & -0.32 \\ 0.21 & -0.52 & 0.31 \end{pmatrix} - \begin{pmatrix} R \\ G \\ B \end{pmatrix}.$$

Моделі RGB, CMY і CMYK орієнтовані на роботу з апаратурою для передачі кольору і для завдання кольору людиною незручні. Для цього використовуються інші моделі, що будуть розглянуті далі.

16.2. Моделі HSV/HSB та HLS/HSI

Модель HSV (*Hue*, *Saturation*, *Value*), іноді також називається HSB (*Hue*, *Saturation*, *Brightness*), більше орієнтована на роботу з людиною і дозволяє задавати кольори, спираючись на інтуїтивні поняття *тону*, *насиченості* і *яскравості*. У цій моделі використовується циліндрична система координат, а множина всіх допустимих кольорів є шестигранним конусом, поставленим на вершину (рис. 16.2).

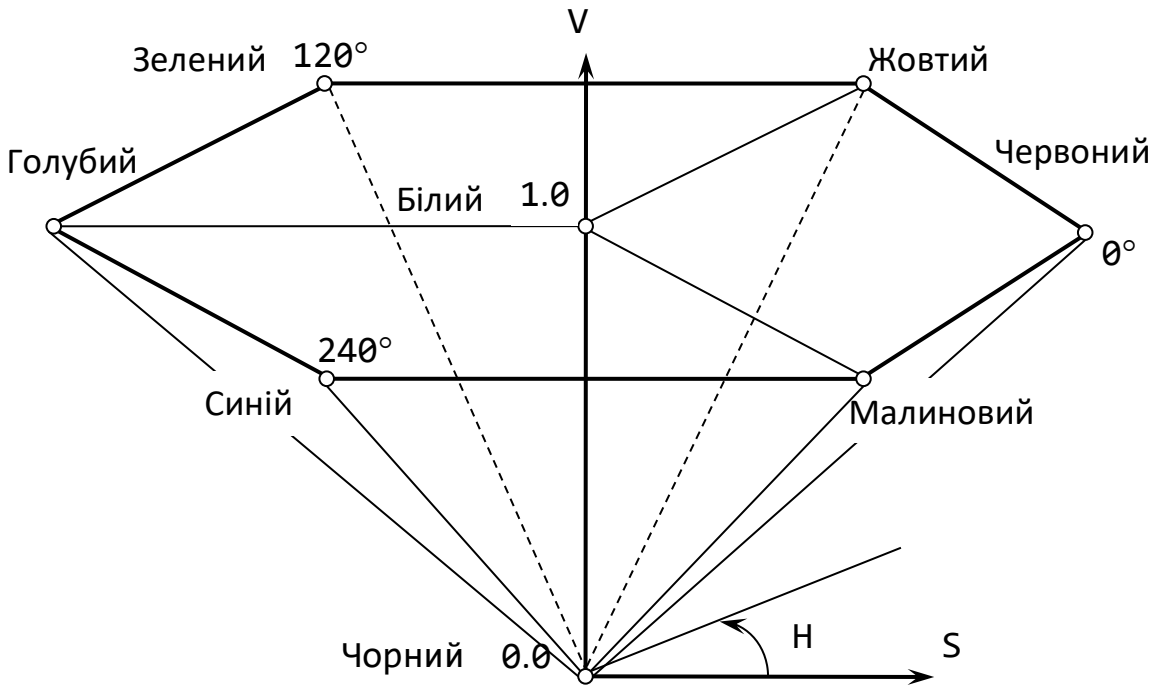


Рис. 16.2. Модель HSV/HSB

H – кольоровий тон, змінюється в межах від 0° до 360°. S і V/B – насиченість і значення/яскравість змінюються в діапазоні від 0 до 1.

Перехід між моделлю RGB та HSV/HSB розраховується за наступними формулами:

$$H = \begin{cases} 0, & \text{якщо } \max(R,G,B) = \min(R,G,B), \\ 60 \times \dfrac{G-B}{\max(R,G,B) - \min(R,G,B)} + 0, & \text{якщо } \max(R,G,B) = R \text{ і } G \geq B, \\ 60 \times \dfrac{G-B}{\max(R,G,B) - \min(R,G,B)} + 360, & \text{якщо } \max(R,G,B) = R \text{ і } G < B, \\ 60 \times \dfrac{B-R}{\max(R,G,B) - \min(R,G,B)} + 120, & \text{якщо } \max(R,G,B) = G, \\ 60 \times \dfrac{R-G}{\max(R,G,B) - \min(R,G,B)} + 240, & \text{якщо } \max(R,G,B) = B. \end{cases}$$

$$S = \begin{cases} 0, & \text{якщо } \max(R,G,B) = 0, \\ 1 - \dfrac{\min(R,G,B)}{\max(R,G,B)}, & \text{інакше.} \end{cases}$$

$$V = \max(R,G,B).$$

Зворотній перехід від моделі HSV/HSB до моделі RGB виконується за наступними формулами:

$$H_i = \left\lfloor \frac{H}{60} \right\rfloor \bmod 6;$$
$$f = \frac{H}{60} - \left\lfloor \frac{H}{60} \right\rfloor;$$
$$p = V(1 - S);$$
$$q = V(1 - fS);$$
$$t = V(1 - (1 - f)S);$$
$$[R \quad G \quad B] = \begin{cases} [V \quad t \quad p], & \text{якщо } H_i = 0, \\ [q \quad V \quad p], & \text{якщо } H_i = 1, \\ [p \quad V \quad t], & \text{якщо } H_i = 2, \\ [p \quad q \quad V], & \text{якщо } H_i = 3, \\ [t \quad P \quad V], & \text{якщо } H_i = 4, \\ [V \quad p \quad q], & \text{якщо } H_i = 5. \end{cases}$$

В комп'ютерній графіці прийнято S і V подавати цілими числами в діапазоні 0-255, замість дійсних чисел від 0 до 1.

Ще одним прикладом моделі, побудованої на інтуїтивних поняттях тону, насиченості і яскравості, є модель HLS (*Hue*, *Lightness*, *Saturation*) або ще її називають HSI (*Hue*, *Saturation*, *Intensity*). Тут також використовується циліндрична система координат, проте множина всіх кольорів подана двома шестигранними конусами, що поставлені один на одного (основа до основи, рис.16.3), причому вершина нижнього конуса співпадає з початком координат. Тон як і раніше задається кутом, що відкладається від вертикальної осі з червоним кольором (кут 0°).

Порядок кольорів на периметрі спільної основи конусів такий же, як і в моделі HSV. Модель HLS можна розглядати як модифікацію моделі HSV, де білий колір зрушений вгору, щоб сформувати верхній конус з площини V=1.

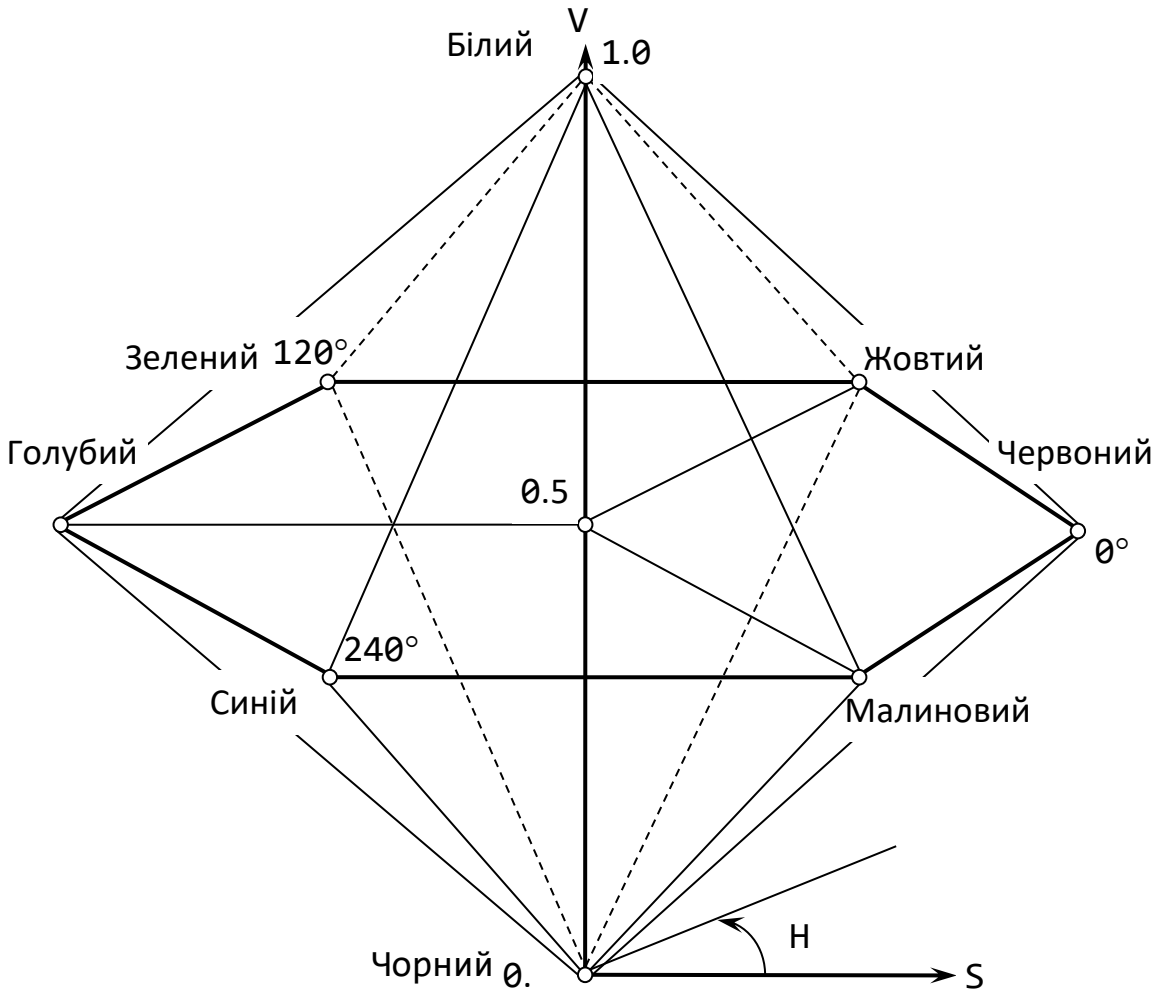


Рис. 16.3. Модель HLS/HSI

Перехід між моделлю RGB та HLS/HSI розраховується за наступними формулами:

$$H = \begin{cases} \text{не визначено}, & \text{якщо } \max(R,G,B) = \min(R,G,B), \\ 60 \times \dfrac{G-B}{\max - \min} + 0, & \text{якщо } \max(R,G,B) = R \text{ і } G \geq B, \\ 60 \times \dfrac{G-B}{\max - \min} + 360, & \text{якщо } \max(R,G,B) = R \text{ і } G < B, \\ 60 \times \dfrac{B-R}{\max - \min} + 120, & \text{якщо } \max(R,G,B) = G, \\ 60 \times \dfrac{R-G}{\max - \min} + 240, & \text{якщо } \max(R,G,B) = B. \end{cases}$$

$$S = \begin{cases} 0, & \text{якщо L} = 0 \text{ або } \max = \min, \\ \dfrac{\max - \min}{\max - \min} = \dfrac{\max - \min}{2\text{L}}, & \text{якщо } 0 < \text{L} \leq \dfrac{1}{2}, \\ \dfrac{\max - \min}{2 - (\max + \min)} = \dfrac{\max - \min}{2 - 2\text{L}}, & \text{якщо } \dfrac{1}{2} < \text{L} < 1, \\ 1, & \text{якщо L} = 1. \end{cases}$$

$$L = \frac{1}{2}(\max(R,G,B) + \min(R,G,B)).$$

Зворотній перехід від моделі HLS/HSI до моделі RGB виконується за наступними формулами:

$$Q = \begin{cases} S(1+L), & \text{якщо } S \le 0.5, \\ S(1+L)+L, & \text{інакше.} \end{cases}$$

$$P = 2S - Q;$$

$$A = \frac{Q-P}{60};$$

$$[T_R \quad T_G \quad T_B] = [H+120 \quad H \quad H-120];$$

$$[R \quad G \quad B] = \begin{cases} [S \quad S \quad S], & \text{якщо } L = 0, \\ P + A[T_R \quad T_G \quad T_B], & \text{якщо } 0 \le H < 60, \\ [Q \quad Q \quad Q], & \text{якщо } 60 \le H < 180, \\ P + A[(240 - T_R) \quad (240 - T_G) \quad (240 - T_B)], & \text{якщо } 180 \le H < 240, \\ [P \quad P \quad P], & \text{якщо } 240 \le H \le 360. \end{cases}$$

Окрім розглянутих вище, існують ще такі моделі кольору: XYZ, RYB, LAB, PMS, LMS, модель Манселла, NCS, RAL, YUV та інші. Всі вони можуть бути приведені до моделі RGB і використовуються в різних галузях науки і техніки, де їх використання зручніше за використання моделей, що розглянуті вище.

Запитання для самоконтролю

1. Яким графічним способом можна подати модель RGB?
2. Яка модель є оберненою до моделі RGB?
3. Якими параметрами задається модель кольору HSV/HSB?
4. Яка фігура відповідає моделі подання кольору HSV/HSB?
5. В чому відмінність моделей HLS/HSI та HSV/HSB?

17. Моделі рендерінгу полігонів

Маючи в своєму розпорядженні засоби обчислення векторів нормалі, можна при заданому розташуванні джерел світла і спостерігача застосувати розглянуті моделі до всіх точок поверхонь об'єктів сцени. Але, на жаль, використання рівнянь обчислення нормалі, стосовно сферичної поверхні, приводить до неприйнятно великих обчислювальних витрат. Використання полігональної моделі для виконання тонування зображення сцени, що складається з множини криволінійних об'єктів, істотно зменшує об'єм обчислень. Саме така модель, що припускає апроксимацію криволінійних поверхонь множиною маленьких плоских багатокутників, і використовується в більшості графічних систем, у тому числі і в *OpenGL* (детально про OpenGL читайте в другій половині посібника).

Розглянемо полігональну мережу. Кожен багатокутник в такій мережі – плоский, і обчислити компоненти вектора нормалі до нього досить легко. Нижче ми розглянемо чотири методи зафарбовування багатокутників: плоске, інтерполяційне, зафарбовування за методом Гуро (*Gouraud*), і зафарбовування за методом Фонга (*Phong*). Існують й інші методи, але ми обмежимося розглядом вказаних.

17.1. Плоске зафарбовування

При переміщенні від однієї точки на поверхні до іншої в загальному випадку можуть змінюватися три вектори – l, n і v. Проте, якщо поверхня плоска, вектор n залишається постійним для всіх точок цієї поверхні. Якщо спостерігач розташований достатньо далеко від цієї поверхні, то зміною вектора v при переході від точки до точки на поверхні плоского багатокутника невеликого розміру також можна знехтувати і вважати його постійним. І, нарешті, якщо в сцені використовується віддалене джерело світла, то вектор l також вважається постійним для всіх точок поверхні, обмеженої зафарбовуваним багатокутником. В даному випадку термін «віддалений» має прямий сенс, тобто вважається, що джерело нескінченно далеко віддалено від освітлюваної поверхні. Для реалізації алгоритму зафарбовування потрібно в цьому випадку замість розташування джерела задавати напрям на джерело. Але термін «віддалений» можна розглядати і у відносному сенсі, порівнюючи розміри зафарбовуваного багатокутника з відстанню до спостерігача або до джерела (рис. 17.1). Саме така інтерпретація цього терміну використовується в більшості графічних систем.

Якщо три вказані вектори постійні для всіх точок багатокутника, то всі необхідні обчислення для його зафарбовування можна виконати тільки один раз і застосувати результати до всіх точок цього багатокутника. Цей метод одержав назву плоского (*flat*) або *рівномірного зафарбовування (constant shading).*

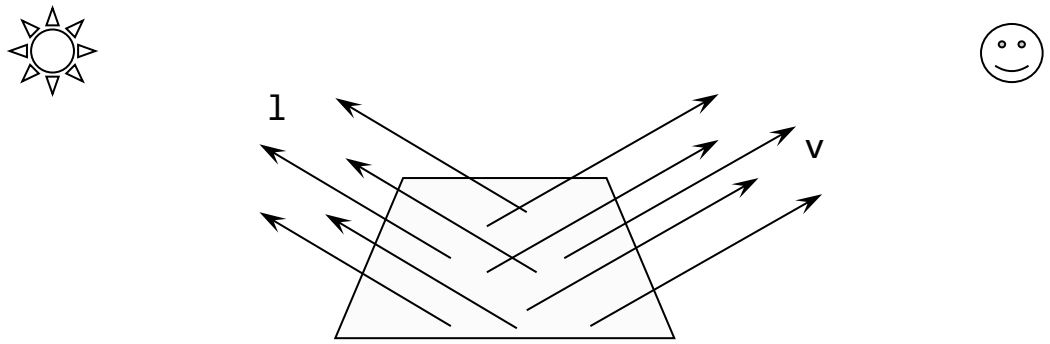


Рис. 17.1. Віддалене джерело світла та спостерігач

У OpenGL режим плоского зафарбовування задається аргументом GL_FLAT при виклику функції glShadeModel():

```
glShadeModel(GL_FLAT)
```

Плоске зафарбовування вимагає найменшої кількості розрахунків і тому працює дуже швидко Але якість отриманого зображення далека від ідеальної.

17.2. Інтерполяційне зафарбовування

Якщо встановити режим згладжування зафарбовування передавши як аргумент функції glShadeModel() значення GL_SMOOTH, то OpenGL інтерполюватиме колір уздовж примітиву, що відображається, наприклад прямої. Припустимо що в програмі встановлені режими згладжування зафарбовування і розрахунку освітлення, а також що з кожною вершиною асоційований вектор нормалі відповідного багатокутника. При обчисленні освітлення кожної вершини визначається її колір, який залежить від властивостей матеріалу і векторів v і l, обчислених раніше для цієї вершини. Зверніть увагу на те, що при використанні віддалених джерел світла і відсутності дзеркальної складової алгоритм інтерполяційного зафарбовування сформує однаковий колір для всієї внутрішньої області багатокутника.

Цей метод зафарбовування дасть вже кращу якість зображення, ніж плоске зафарбовування, але оскільки кожен багатокутник має свою нормаль і він повністю зафарбований одним тоном, то на місцях стиків багатокутників будуть видні різкі перепади кольорових відтінків.

17.3. Зафарбовування за методом Гуро

Повертаючись до полігональної мережі, відзначимо, що ідея використання в обчисленнях нормалей, що асоціюються з вершинами такої мережі, повинна викликати у будь-якого математика заперечення, оскільки з математичної точки зору вона абсолютно некоректна. Оскільки вершина є точкою перетину, як мінімум, двох по-різному орієнтованих багатокутників, то в ній відбувається розрив неперервності функції вектора нормалі. Хоча така ситуація і може серйозно ускладнити математику алгоритму, *Гуро* дійшов висновку, що нормаль в точці вершини може бути визначена у такий спосіб, який дозволить згладити зафарбовування. Розглянемо одну з вершин всередині мережі, в якій перетинаються чотири багатокутники (рис. 17.2).

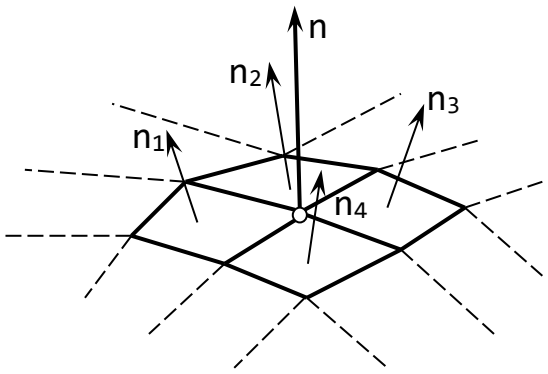


Рис. 17.2. Полігональна мережа з векторами нормалей

Кожен багатокутник має свій вектор нормалі. *Метод зафарбовування Гуро* полягає в тому, що з вершинами зв'язуються нормалі, які виходять в результаті усереднювання нормалей багатокутників, що перетинаються в цій вершині. Для прикладу, що поданий на рисунку 17.3, з виділеною вершиною асоціюється нормаль, обчислена відповідно до співвідношення:

$$n = \frac{n_1 + n_2 + n_3 + n_4}{|n_1 + n_2 + n_3 + n_4|}.$$

Метод Гуро досить просто реалізується в програмі, що використовує OpenGL. Від програміста потрібно тільки правильно обчислити нормалі, що асоціюються з вершинами полігональної мережі. У літературі часто змішуються метод зафарбовування Гуро та інтерполяційний метод зафарбовування. Іноді це викликає певні проблеми. Як визначити, які нормалі слід усереднювати для обчислення нормалі, що асоціюється з певною вершиною? Якщо структура даних в програмі лінійна, тобто всі вершини перераховані в звичайному лінійному списку. Ми не маємо в своєму розпорядженні інформації про те, які саме багатокутники

перетинаються в певній вершині. Сам собою напрошується висновок, що потрібна така структура даних, яка відображала б зв'язки між багатокутниками в мережі. Проглядаючи таку структуру даних, можна визначити ті вершини, в яких слід усереднювати вектори нормалей. Такого роду структура даних повинна зв'язувати, як мінімум, багатокутники, вершини і властивості матеріалів.

17.4. Зафарбовування за методом Фонга

Але навіть при використанні методу Гуро у ряді випадків не вдається уникнути появи на зображенні *смуг Маха*. Смуги Маха – це ефект, що виникає на кордоні між об'єктами, що зафарбовані однаковим кольором але з різною яскравістю. При цьому око сприймає не плавний перехід, а ілюзорні смуги: в яскравішій частині ми бачимо більш світлу смугу, а в темнішій – більш темну. *Фонг* запропонував інтерполювати не колір точок від вершини до вершини, а напрям нормалей послідовних точок на ребрах кожного багатокутника. Розглянемо окремий багатокутник полігональної мережі (рис.17.3) .

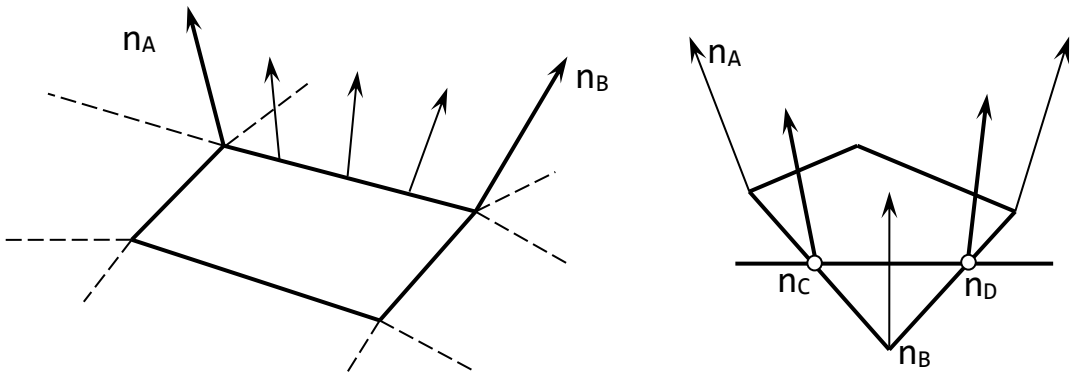


Рис. 17.3. Інтерполяція векторів нормалей вздовж ребра

З кожною вершиною можна зв'язати нормаль, яка формується усереднюванням нормалей до граней що перетинаються в цій вершині. Далі за допомогою білінійної інтерполяції можна інтерполювати нормалі по всій внутрішній області багатокутника. Вектори нормалей у вершинах A і B використовуються для інтерполяції уздовж ребра, що зв'язує ці дві вершини:

$$N(\alpha) = (1-\alpha)n_A + \alpha n_B, \alpha \in [0 \dots 1].$$

Аналогічну процедуру можна виконати і для інших ребер багатокутника. Потім можна обчислити нормаль в будь-якій точці внутрішньої області цього багатокутника, знаючи розподіл нормалей на його ребрах:

$$N(\alpha, \beta) = (1-\beta)n_C + \beta n_D, \beta \in [0 \dots 1].$$

Одержавши вектор нормалі в певній точці, можна виконати всі необхідні обчислення, що визначаються використовуваною моделлю віддзеркалення.

Метод Фонга дозволяє одержати гладке тонування зображення, але за це доводиться платити неабияку ціну – об'єм обчислень різко зростає. Існує безліч варіантів апаратної реалізації методу Гуро, які дозволяють одержувати зображення прийнятної якості, практично не збільшуючи час зафарбовування, чого не скажеш про метод Фонга. В результаті в даний час метод Фонга використовується тільки в тих системах, де не потрібно формувати зображення в реальному масштабі часу.

Запитання для самоконтролю

1. Які параметри впливають на тонування об'єкта?
2. Які переваги плоского зафарбовування?
3. Яка відмінність інтерполяційного зафарбовування від плоского?
4. Що запропонував Гуро для зафарбовування багатокутників?
5. В чому суть зафарбовування за методом Фонга?

18. Основи OpenGL

18.1. Основні можливості

Основне призначення *OpenGL* – відображення двовимірних та тривимірних об'єктів у статичних та динамічних сценах. Об'єкти подаються у вигляді сукупності вершин (для геометричних фігур) або пікселів (для растрових зображень). OpenGL спочатку перетворює вхідні дані (примітиви та зображення) в піксельне подання, асоціюючи з кожним сформованим пікселем необхідні для його відображення і подальшої роботи дані, а потім розташовує результат перетворення в буфері кадру.

Реалізація OpenGL для Windows містить більше 400 функцій (368 базових функцій основної бібліотеки *opengl32.dll* та 52 функції бібліотеки утиліт *glu32.dll*).

Базові функції забезпечують побудову зображень графічних примітивів (точки, лінії, багатокутники, растрові зображення), перетворення координат, обмеження області видимості, управління кольором, освітленням, текстурою, туманом.

Функції бібліотеки утиліт є розширенням базового набору функцій і призначені для формування зображень сфер, дисків, конічних циліндрів, управління текстурою і перетвореннями координат, тріангуляції багатокутників, побудови кривих та поверхонь на нерегулярній сітці контрольних точок з використанням форм Без'є та раціональних B-сплайнів.

Всі базові функції можна розділити на п'ять категорій.

1. *Функції опису примітивів* визначають об'єкти нижнього рівня ієрархії (примітиви), які здатна відображати графічна система. У OpenGL примітивами є точки, лінії, багатокутники і т.д.
2. *Функції опису джерел світла* слугують для опису положення і параметрів джерел світла, розташованих у тривимірній сцені.
3. *Функції завдання атрибутів*. За допомогою завдання атрибутів програміст визначає, *як* будуть виглядати на екрані відображувані об'єкти. Іншими словами, якщо за допомогою примітивів визначається, *що* з'явиться на екрані, то атрибути визначають *спосіб* виведення на екран. В якості атрибутів OpenGL використовує колір, характеристики матеріалу, текстури, параметри освітлення.
4. *Функції візуалізації* дозволяють задати положення спостерігача у віртуальному просторі, параметри об'єктива камери. Знаючи ці

параметри, система зможе не тільки правильно побудувати зображення, але і відсікти об'єкти, які не потрапили в поле зору.

5. Набір *функцій геометричних перетворень* дозволяє програмісту виконувати різні перетворення об'єктів – поворот, зсув, масштабування.

18.2. Функціональна модель графічних застосунків на основі OpenGL

Характерними рисами сучасних технологій програмування є використання об'єктно-орієнтованих візуальних засобів розробки програмного забезпечення та інтерфейсу користувача універсального призначення (*Microsoft Visual Studio*, *Delphi* тощо). Водночас, засоби графічного програмування (*OpenGL*, *Windows GDI*, *Direct3D*) реалізовані у вигляді бібліотек функцій і процедур, що пояснюється жорсткими вимогами до швидкості побудови зображень.

Узагальнена функціональна модель прикладних графічних застосунків на основі OpenGL може бути подана у вигляді ієрархії обробних систем:

$$S_k = (L_k, I_k),$$

де L_k – мова, I_k – інтерпретатор мови L_k в мову L_{k+1} наступної обробної системи S_{k+1}.

У першому наближенні (на початкових стадіях етапу проектування застосунку) функціональна модель визначається трійкою {(L_i, I_i), (L_m, I_m), (L_0, I_0)}. На вершині функціональної моделі знаходяться вхідна мова графічної системи L_i (команди і дані команд) та інтерпретатор мови L_i у внутрішню мову L_m (моделі даних та геометричні операції над ними). Інтерпретатор моделі I_m забезпечує опис зображень двовимірних чи тривимірних об'єктів в статичних і динамічних сценах на мові L_0 (інтерфейс прикладного програмування для генерації команд і списків команд OpenGL). Інтерпретатор I_0 (сервер OpenGL) забезпечує формування пікселів зображення в буфері кадру, виведення вмісту буферу на екран та повернення результатів запитів. Необхідний рівень деталізації функціональної моделі досягається об'єктною чи процедурною декомпозицією обробних систем (L_i, I_i) та (L_m, I_m), залежно від можливостей середовища програмування. Рівень розвитку засобів візуального програмування об'єктів інтерфейсу з користувачем суттєво впливає на алфавіт і синтаксис конструкцій мови L_i та трудомісткість розробки інтерпретатора I_i.

З урахуванням великої кількості команд OpenGL, розробка інтерпретатора I_m для складних графічних застосунків (системи

автоматизації проектування, дизайну, геометричного моделювання явищ і процесів тощо) є досить трудомісткою.

Природно, що в об'єктно-орієнтованих середовищах програмування у розробників виникає спокуса створення класів-оболонок для інкапсуляції команд OpenGL в методах та властивостях, що еквівалентно введенню в функціональну модель додаткової обробної системи (L_v, I_v) на передостанньому рівні ієрархії. Однак, просте «загортання» команд мови L_0 в оболонку класів об'єктів мови L_v викличе лише додаткові витрати часу на формування та виведення зображень. Спрощення розробки I_m без втрати ефективності функціонування програми може бути досягнуто, якщо елементами мови L_v будуть методи формування зображень складних об'єктів моделі та управління параметрами сцен.

Функції координатної ідентифікації елементів зображень та управління візуалізацією об'єктів і сцен (повороти, панорамування, масштабування) можуть бути інкапсульовані безпосередньо в методах екранних форм чи фреймів (візуальних компонентів Delphi), що найбільш повно відповідає концепціям об'єктно-орієнтованого програмування і полегшує створення застосунків для одночасної роботи з зображеннями кількох сцен.

18.3. Інтерфейс OpenGL

OpenGL складається з набору бібліотек. Усі базові функції зберігаються в основній бібліотеці, для позначення якої надалі будемо використовувати абревіатуру GL. Крім основної, OpenGL містить кілька додаткових бібліотек.

Перша з них — бібліотека утиліт *GLU* (*GL Utility*). Усі функції цієї бібліотеки визначені через базові функції OpenGL. До складу GLU увійшла реалізація більш складних функцій, таких як набір популярних геометричних примітивів (куб, куля, циліндр, диск), функції побудови сплайнів, реалізація додаткових операцій над матрицями і т.п.

OpenGL не містить у собі ніяких спеціальних команд для роботи з вікнами чи отримання інформації від користувача. Тому були створені спеціальні бібліотеки для забезпечення функцій, що часто використовуються при взаємодії з користувачем і для відображення інформації за допомогою віконної підсистеми. Найбільш популярною є бібліотека *GLUT* (GL Utility Toolkit). Формально GLUT не входить у OpenGL, але фактично включається майже в усі його дистрибутиви і має реалізації для різних платформ. GLUT надає лише мінімально необхідний набір функцій для створення OpenGL-застосування. Функціонально аналогічна бібліотека GLX менш популярна.

Крім того, функції, специфічні для конкретної віконної підсистеми, звичайно входять у її прикладний програмний інтерфейс. Так, функції, що підтримують виконання OpenGL, є в складі *Win32 API* та *X Window*. На рис. 18.1 схематично подана організація системи бібліотек у версії, що працює під управлінням системи Windows. Аналогічна організація використовується й в інших версіях OpenGL.

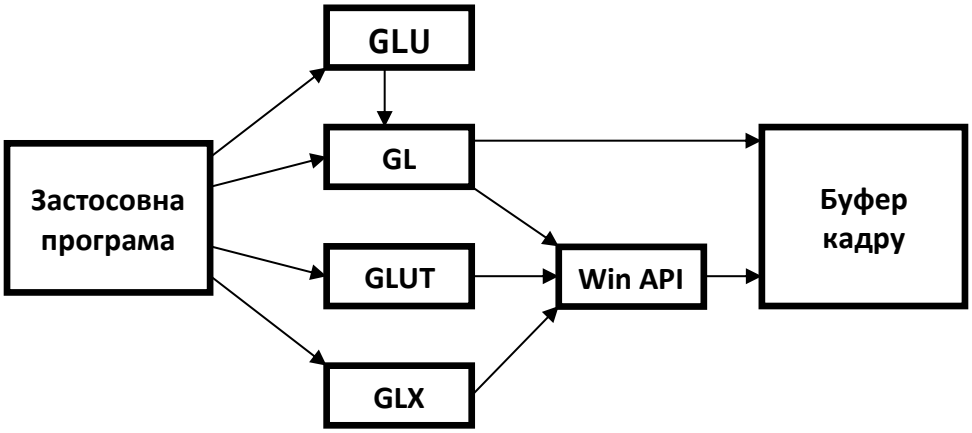


Рис. 18.1. Організація бібліотеки OpenGL

18.4. Архітектура OpenGL

Функції OpenGL реалізовані з використанням моделі клієнт-сервер. Застосунок (прикладна програма) виступає в ролі клієнта – він виробляє команди, а сервер OpenGL інтерпретує і виконує їх. Сам сервер може знаходитися як на тому ж комп'ютері, що й клієнт (наприклад, у вигляді бібліотеки динамічного завантаження – *DLL*), так і на іншому (для цього може бути використаний спеціальний протокол передачі даних між машинами).

OpenGL обробляє і формує в буфері кадру зображення графічних примітивів з урахуванням деякого числа обраних режимів. Окремий примітив – це точка, відрізок, багатокутник і т.д. Кожен режим може бути змінений незалежно від інших. Визначення примітивів, вибір режимів та інші операції описуються за допомогою команд у формі викликів функцій прикладної бібліотеки.

Примітиви визначаються набором з однієї чи декількох *вершин* (*vertex*). *Вершина* визначає точку, кінець відрізка чи кут багатокутника. З кожною вершиною асоціюються деякі дані (координати, колір, нормаль, текстурні координати і т. д.), які називаються *атрибутами*. У переважній більшості випадків кожна вершина обробляється незалежно від інших.

Архітектура OpenGL реалізує схему конвеєра, що складається з кількох послідовних етапів обробки графічних даних (рис. 18.2).

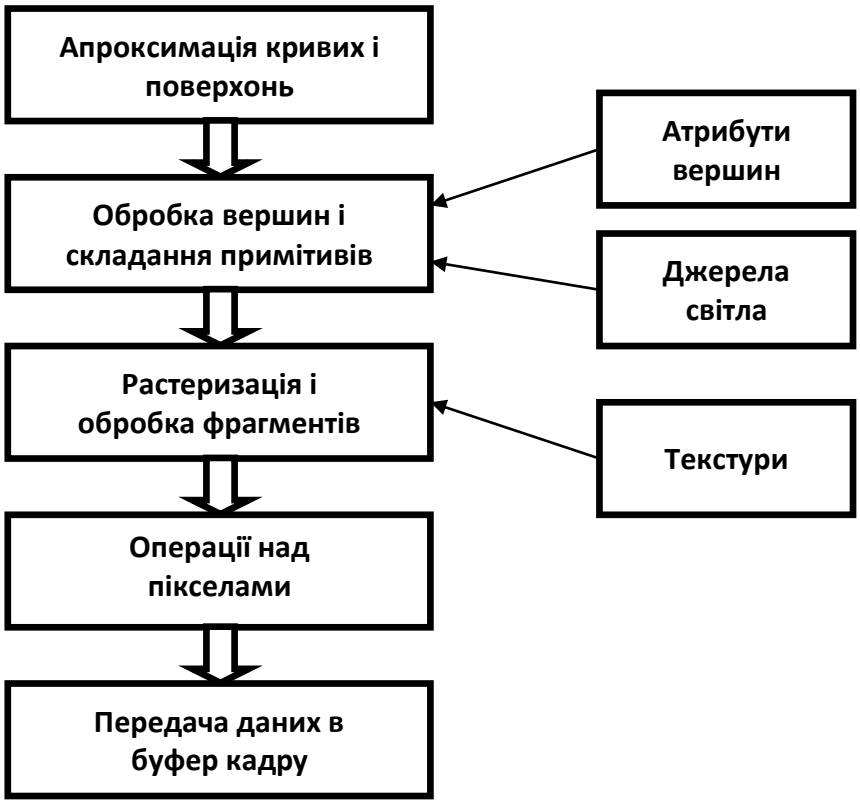


Рис. 18.2. Схема функціонування конвеєра OpenGL

Команди OpenGL завжди оброблюються в порядку їх надходження, хоча можуть відбуватися затримки перед тим, як проявиться ефект від їхнього виконання. У більшості випадків OpenGL реалізує безпосередній інтерфейс, тобто визначення об'єкта викликає його візуалізацію в буфері кадру.

З точки зору розробників, OpenGL – це набір команд, які керують використанням графічної апаратури. Якщо апаратура складається тільки з адресованого буфера кадру, то тоді функції OpenGL повинні бути реалізовані повністю за рахунок ресурсів центрального процесора. Як правило графічна апаратура забезпечує різноманітні рівні прискорення: від апаратної реалізації виводу ліній і багатокутників до витончених графічних процесорів з підтримкою різних операцій над геометричними даними.

OpenGL є прошарком між апаратним рівнем та рівнем користувача, що дозволяє надавати єдиний інтерфейс на різних платформах, використовуючи можливості апаратної підтримки.

Крім того, OpenGL можна розглядати як скінченний автомат, стан якого визначається множиною значень спеціальних змінних і значеннями поточної нормалі, кольору, координат текстури та інших атрибутів і

ознак. Уся ця інформація буде використана при надходженні в графічну систему координат вершини для побудови фігури, до якої вона належить. Зміна станів відбувається за допомогою команд, що оформлюються як виклики функцій.

18.5. Синтаксис команд

Бібліотеки *opengl32.dll* та *glu32.dll* написані на мові *C++*. Тому для використання цих бібліотек в проектах C++ Builder достатньо підключити їх заголовні файли (*gl.h* та *glu.h* відповідно):

```
#include <gl.h>
#include <glu.h>
```

Разом з Delphi версій 3 та вище надається заголовний файл, який дозволяє підключати бібліотеку OpenGL до проектів Delphi. Цей файл містить лише описи прототипів функцій бібліотеки, а самі функції розташовані у відповідних файлах DLL.

Наприклад, у секції `interface` заголовного файлу знаходиться наступне попереднє оголошення функції очищення екрану:

```
procedure glClearColor (red,green,blue,alpha : GLclampf);
stdcall;
```

В секції `imlementation` модуля опис має наступний вигляд:

```
procedure glClearColor; external opengl32;
```

Службове слово `stdcall` вказане для усіх процедур та функцій цього модуля, означає стандартний виклик функції чи процедури та визначає конкретні правила обміну даними між застосунками (прикладною програмою) та бібліотекою: як передаються параметри (через регістри чи стек), в якому порядку перелічуються параметри, і хто, прикладна програма чи бібліотека, очищує області пам'яті після їх застосування.

Службове слово `external` застосовується для функцій, що підключаються з бібліотек. Після нього вказується ім'я бібліотеки, що підключається. Тут `opengl32` – константа, що визначається у іншому модулі *windows.pas*:

```
opengl32 = 'opengl32.dll';
```

Константа, що відповідає іншій бібліотеці, знаходиться у модулі opengl.pas:

```
const glu32 = 'glu32.dll';
```

Змістовна частина модуля opengl.pas, що відповідає його ініціалізації, вміщує один рядок:

```
Set8087CW($133F);
```

Ця процедура призначена для включення/виключення виняткових ситуацій при проведенні операцій з плаваючою точкою. Для OpenGL обробку виняткової ситуації рекомендується відключати.

Усі команди (процедури і функції) бібліотеки OpenGL починаються з префікса gl, усі константи – з префікса GL_. Команди і константи бібліотек GLU і GLUT аналогічно мають префікси glu (GLU_) і glut (GLUT_).

Крім того, в імена команд входять суфікси, що несуть інформацію про число і тип переданих параметрів. В OpenGL повне ім'я команди має такий вид:

```
type glCommand_name[1 2 3 4][b i f d ub us ui][v](type1 arg1,…,typeN argN)
```

gl	ім'я бібліотеки, у який описана ця функція: для базових функцій OpenGL це gl, для функцій із бібліотек GL, GLU, GLUT, GLAUX – glu, glut, aux відповідно
Command_name	ім'я команди (процедури чи функції)
[1 2 3 4]	число аргументів команди
[b s i f d ub us ui]	тип аргументу: символ b – GLbyte (аналог *char* у C\C++), символ i – GLint (аналог *int*), символ f – GLfloat (аналог *float*) і т.д. Повний список типів і їх опис можна подивитися у файлі opengl.pas
[v]	наявність цього символу показує, що у якості параметрів функції використовується покажчик на масив значень

Символи в квадратних дужках у деяких назвах не використовуються. Наприклад, описана в бібліотеці OpenGL команда glVertex2i, використовує як параметри два цілих числа, а команда glColor3fv використовує як параметр покажчик на масив із 3-х дійсних чисел.

18.6. Приклад простого застосунку

Розглянемо мінімальну програму, яка використовує OpenGL. Програма за допомогою команд OpenGL малює в центрі вікна червоний квадрат.

В проекті Delphi список uses доповнений посиланням на модуль opengl.pas – це програміст повинен зробити самостійно.

```
uses
```

```
Windows, Messages, Forms, Classes, Controls, ExtCtrls,
ComCtrls, StdCtrls, Dialogs, SysUtils, OpenGL;
```

В проекті C++ Builder потрібно підключити заголовний файл OpenGL:

```
#include <gl.h>
```

Розділ `private` опису класу форми містить два рядки:

```
glDC: HDC; // посилання на контекст пристрою
hrc: HGLRC; // посилання на контекст відображення
```

Контекст відображення для OpenGL виконує ту ж роль, що і контекст пристрою для GDI.

Контекст пристрою є структурою, яка визначає комплект графічних об'єктів та зв'язаних з ними атрибутів і графічні режими, що впливають на виведення зображення. Графічний об'єкт вміщує у собі олівець для зображення ліній, пензлик для зафарбовування та заповнення областей, растр для копіювання чи прокрутки частин екрану, палітру для визначення комплекту доступних кольорів, області для відсікання та інших операцій, маршрут для операцій малювання.

Обробник події `OnCreate` форми в Delpi містить наступні рядки:

```
glDC:=GetDC(Handle);
SetDCPixelFormat(glDC);
hrc:=wglCreateContext(glDC);
wglMakeCurrent(glDC, hrc);
```

В конструктор форми в C++ Builder потрібно додати наступні рядки:

```
glDC=GetDC(Handle);
SetDCPixelFormat(glDC);
hrc=wglCreateContext(glDC);
wglMakeCurrent(glDC, hrc);
```

Перший рядок – отримання контексту пристрою з головної форми проекту. Наступний рядок – звернення до описаної у цьому ж модулі функції (процедури) користувача, яка задає формат піксела. Для Delphi ця процедура має наступний вигляд:

```
procedure SetDCPixelFormat(hdc:HDC);
var pfd : TPixelFormatDescriptor;
    nPF : Integer;
begin
  FillChar (pfd, SizeOf(pfd), 0);
  nPF:=ChoosePixelFormat(hdc, @pfd);
```

```
  SetPixelFormat(hdc, nPF, @pfd);
end;
```

В C++ Builder функція установки формату піксела наступна:

```
void SetDCPixelFormat(HDC hdc)
{ PIXELFORMATDESCRIPTOR pfd, *ppfd;
  int nPF;
  ppfd = &pfd;
  ppfd->nSize= sizeof(PIXELFORMATDESCRIPTOR);
  nPF=ChoosePixelFormat(hdc, ppfd);
  SetPixelFormat(hdc, nPF, ppfd);
}
```

До отримання контексту відображення, сервер OpenGL повинен спочатку отримати детальні характеристики використовуваного обладнання. Ці характеристики зберігаються у спеціальній структурі, тип якої TPixelFormatDesriptor (опис формату піксела). Формат піксела визначає конфігурацію буферу кольору і допоміжних буферів.

Після заповнення полів структури TPixelFormatDesriptor, ми визначаємо свої власні вимоги до графічної системи, на якій буде виконуватись програма, а OpenGL підбирає найбільш відповідний формат і встановлює його у якості формату піксела для подальшої роботи. Але OpenGL не дозволить встановити нереальний для конкретного робочого місця формат. Те, як ми задамо значення прапорців у полі структури dwFlags, може суттєво позначитись на роботі програми і тому випадково задавати ці значення небажано. Більш детально ознайомитись зі структурою TPixelFormatDesriptor можна за допомогою файлу довідки та файлу windows.pas або в заголовному файлі *wingdi.h*.

У наступному рядку обробника OnCreate задається змінна типу HGLRC, тобто створюється контекст відображення. Аргументом функції wglCreateContext є посилання на контекст пристрою, на якому буде виконуватись виведення. У цьому прикладі пристроєм виводу слугує вікно форми. Для отримання контексту OpenGL необхідна змінна типу HDC. Останній рядок обробника робить контекст відображення поточним.

Обробник події OnPaint для Delphi наведений нижче:

```
var ps : TPaintStruct;
BeginPaint(glDC, ps);
```

```
glClearColor (0.5, 0.5, 0.75, 1.0);
glClear (GL_COLOR_BUFFER_BIT);
glColor3f(1.0,0.0,0.0);
glBegin(GL_QUADS);
  glVertex2f(0.5, 0.5);
  glVertex2f(-0.5,0.5);
  glVertex2f(-0.5, -0.5);
  glVertex2f(0.5, -0.5);
glEnd;
SwapBuffers(glDC);
EndPaint(glDC, ps);
```

В проекті C++ Builder код буде практично аналогічним:

```
PAINTSTRUCT ps;
BeginPaint(glDC, &ps);
glClearColor (0.5, 0.5, 0.75, 1.0);
glClear (GL_COLOR_BUFFER_BIT);
glColor3f(1.0,0.0,0.0);
glBegin(GL_QUADS);
  glVertex2f(0.5, 0.5);
  glVertex2f(-0.5,0.5);
  glVertex2f(-0.5, -0.5);
  glVertex2f(0.5, -0.5);
glEnd();
SwapBuffers(glDC);
EndPaint(glDC, &ps);
```

Все виведення знаходиться між командами `BeginPaint` та `EndPaint`. Першою командою задається колір фону, далі йде рядок, який викликає очищення екрану і зафарбування його заданим кольором. Потім задаємо колір квадрата (у прикладі – червоний), далі визначаємо його координати (більш детально операції роботи з графічними примітивами будуть розглянуті у наступних розділах). Коли все зображення побудоване, виводимо його на екран за допомогою команди `SwapBuffers`.

Обробник події `OnDestroy` містить команди, для звільнення контекстів відображення та пристрою. Їх синтаксис однаковий як в проектах Delphi так і в проектах C++ Builder:

```
wglMakeCurrent (0, 0);
wglDeleteContext(hrc);
ReleaseDC(Handle, glDC);
DeleteDC(glDC);
```

Спочатку звільнюємо контекст відображення, потім видаляємо його для звільнення пам'яті. Останні два рядки звільнюють та видаляють контекст пристрою.

Повний текст модуля головної форми в проекті Delphi:

```
unit ufGL;
interface
uses
  Windows, Messages, Forms, Classes, Controls, ExtCtrls,
ComCtrls,
  StdCtrls, Dialogs, SysUtils, OpenGL;
type
  TfrmGL = class(TForm)
    procedure FormCreate(Sender: TObject);
    procedure FormPaint(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
  private
    glDC: HDC;
    hrc: HGLRC;
  end;
var
  frmGL: TfrmGL;
implementation
{$R *.DFM}
procedure TfrmGL.FormPaint(Sender: TObject);
var ps : TPaintStruct;
begin
  BeginPaint(glDC, ps);
  glClearColor (0.5, 0.5, 0.75, 1.0);
  glClear (GL_COLOR_BUFFER_BIT);
  glColor3f(1.0,0.0,0.0);
  glBegin(GL_QUADS);
    glVertex2f(0.5, 0.5);
```

```
    glVertex2f(-0.5,0.5);
    glVertex2f(-0.5, -0.5);
    glVertex2f(0.5, -0.5);
  glEnd;
  SwapBuffers(glDC);
  EndPaint(glDC, ps);
end;
procedure SetDCPixelFormat (hdc : HDC);
var
  pfd : TPixelFormatDescriptor;
  nPixelFormat : Integer;
begin
  FillChar (pfd, SizeOf (pfd), 0);
  nPixelFormat := ChoosePixelFormat (hdc, @pfd);
  SetPixelFormat (hdc, nPixelFormat, @pfd);
end;
procedure TfrmGL.FormCreate(Sender: TObject);
begin
  glDC := GetDC(Handle);
  SetDCPixelFormat(glDC);
  hrc := wglCreateContext(glDC);
  wglMakeCurrent(glDC, hrc);
end;
procedure TfrmGL.FormDestroy(Sender: TObject);
begin
  wglMakeCurrent (0, 0);
  wglDeleteContext(hrc);
  ReleaseDC(Handle, glDC);
  DeleteDC(glDC);
end;
end.
```

Проект в C++ Builder пропонується зробити самостійно на основі приведених вище фрагментів коду. Насправді проекти на Delphi та C++ Builder мало чим відрізняються, оскільки набір команд OpenGL, їх параметрів та типів даних, що використовуються, абсолютно однакові.

Запитання для самоконтролю

1. У чому, на Вашу думку, полягає необхідність створення стандартної графічної бібліотеки?
2. Коротко опишіть архітектуру бібліотеки OpenGL і організацію конвеєра графічних перетворень.
3. Назвіть категорії базових команд (функцій) бібліотеки.
4. Навіщо потрібні різні варіанти команд OpenGL, що відрізняються тільки типами параметрів?
5. Чому організацію OpenGL часто порівнюють із скінченним автоматом?

19. Формування зображень геометричних об'єктів

19.1. Процес оновлення зображення

Як правило, задачею програми, що використовує OpenGL, є обробка тривимірної сцени та її відображення в буфері кадру. Сцена складається з набору тривимірних об'єктів, джерел світла і віртуальної камери, яка визначає поточне положення спостерігача.

Зазвичай застосунок OpenGL у нескінченному циклі викликає функцію оновлення зображення у вікні. У цій функції і зосереджені виклики основних команд OpenGL. Якщо використовується бібліотека GLUT, то це буде функція зі зворотним викликом, зареєстрована за допомогою виклику `glutDisplayFunc`. GLUT викликає цю функцію, коли операційна система інформує застосунок про те, що вміст вікна необхідно перемалювати (наприклад, якщо вікно було перекрито іншим). Створюване зображення може бути як статичним, так і анімованим, тобто залежати від деяких параметрів, що змінюються з часом. У цьому випадку краще викликати функцію оновлення самостійно.

Як правило типова функція оновлення зображення забезпечує:

- очищення буферів OpenGL;
- установка положення спостерігача;
- перетворення і малювання геометричних об'єктів.

Очищення буферів забезпечується командою:

```
glClear (mask : bitfield).
```

Ця команда очищує задані буфери з поточним значенням (див. розділ 23). Параметр `mask` є комбінацією наступних значень:

GL_COLOR_BUFFER_BIT	буфер кольору
GL_DEPTH_BUFFER_BIT	буфер глибини
GL_ACCUM_BUFFER_BIT	буфер-накопичувач
GL_STENCIL_BUFFER_BIT	буфер трафарету

Типова програма викликає команду

```
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)
```

для очищення буферів кольору і глибини.

Команда `glClearColor` встановлює колір, яким буде заповнений буфер кадру. Перші три параметри команди задають R, G і B компоненти

кольору та повинні належати відрізку [0,1]. Четвертий параметр задає так звану альфа-компоненту (див. розділ 23.1). Як правило, вона дорівнює 1. За замовчуванням колір – чорний (0,0,0,1).

glClearColor (*r*: clampf, *g*: clampf, *b*: clampf, *a*: clampf).

Установка положення спостерігача і перетворення тривимірних об'єктів (поворот, зсув і т.д.) контролюються за допомогою завдання матриць перетворень. Перетворення об'єктів і налагодження положення віртуальної камери описані в розділі 20.

Зараз зосередимося на тому, як передати в OpenGL опис об'єктів, що знаходяться в сцені. Кожен об'єкт є набором примітивів OpenGL.

19.2. Вершини і примітиви

Вершина є атомарним графічним примітивом OpenGL і визначає точку, кінець відрізка, кут багатокутника і т.д. Всі інші примітиви формуються за допомогою завдання вершин, що входять у даний примітив. Наприклад, відрізок визначається двома вершинами, що є кінцями відрізка.

З кожною вершиною асоціюються її *атрибути*, основними з яких є положення вершини в просторі, її колір і вектор нормалі.

19.2.1. Положення вершини в просторі

Положення вершини визначається завданням її координат у дво-, три-, або чотиривимірному просторі (однорідні координати). Це реалізується за допомогою декількох варіантів команди glVertex:

glVertex [2 3 4] [s i f d] (coords: GLtype)

glVertex [2 3 4] [s i f d]v (coords: ^GLtype).

Кожна команда задає чотири координати вершини: x, y, z, w. Команда glVertex2* одержує значення x і y. Координата z у такому випадку встановлюється за замовчуванням рівною 0, координата w – рівною 1. Команда glVertex3* одержує координати x, y, z і заносить у координату w значення 1. Команда glVertex4* дозволяє задати всі чотири координати.

Для асоціації з вершинами кольорів, нормалей і текстурних координат використовуються поточні значення відповідних даних, що відповідає організації OpenGL як скінченного автомата. Ці значення можуть бути змінені в будь-який момент за допомогою виклику відповідних команд.

19.2.2. Колір вершини

Для завдання поточного кольору вершини використовуються команди:

```
glColor [3 4] [b s i f]  (components: GLtype)
glColor [3 4] [b s i f]v (components: ^GLtype).
```

Перші три параметри задають R, G і B компоненти кольору, а останній параметр визначає коефіцієнт непрозорості (так звану альфа-компоненту). Якщо в назві команди зазначений тип f (*float*), то значення всіх параметрів повинні належати відрізку [0,1], при цьому за замовчуванням значення альфа-компоненти встановлюється рівним 1, що відповідає повній непрозорості. Тип ub (*unsigned byte*) вказує, що значення повинні лежати у відрізку [0,255].

Вершинам можна призначати різні кольори, і, якщо включений відповідний режим, то буде проводитися лінійна інтерполяція кольорів по поверхні примітива.

Для керування режимом інтерполяції використовується команда

```
glShadeModel (mode: GLenum)
```

виклик якої з параметром GL_SMOOTH вмикає інтерполяцію (ввімкнена за замовчуванням), а з параметром GL_FLAT – вимикає.

19.2.3. Нормаль

Визначити нормаль у вершині можна командами

```
glNormal3 [b s i f d] (coords: GLtype)
glNormal3[b s i f d]v (coords: ^GLtype).
```

Для правильного розрахунку освітлення необхідно, щоб вектор нормалі мав одиничну довжину. Командою glEnable(GL_NORMALIZE) можна ввімкнути спеціальний режим, за якого нормалі, що задаються, будуть нормуватися автоматично.

Режим автоматичної нормалізації повинен бути ввімкненим, якщо застосунок (прикладна програма) використовує модельні перетворення розтягування/стиснення, тому що в цьому випадку довжина нормалей змінюється при множенні на модельно-видову матрицю.

Однак застосування цього режиму зменшує швидкість роботи механізму візуалізації OpenGL, тому що нормалізація векторів має помітну обчислювальну складність (здобуття квадратного кореня і т. п.). Тому краще відразу задавати одиничні нормалі.

Відзначимо, що команди

```
glEnable (mode: GLenum)
glDisable (mode: GLenum)
```

виконують ввімкнення і вимкнення того чи іншого режиму роботи конвеєра OpenGL. Ці команди застосовуються досить часто, і їхні можливі параметри будуть розглядатися в кожному конкретному випадку.

19.3. Операторні дужки glBegin/glEnd

Ми розглянули завдання атрибутів однієї вершини. Однак, щоб задати атрибути графічного примітива, лише координат вершин недостатньо. Ці вершини треба об'єднати в одне ціле, визначивши необхідні властивості. Для цього в OpenGL використовуються так звані операторні дужки, що є викликами спеціальних команд OpenGL. Визначення примітива чи послідовності примітивів відбувається між викликами команд

```
glBegin (mode: GLenum)
glEnd
```

Параметр mode визначає тип примітива, який задається всередині і може приймати наступні значення (рис. 19.1):

GL_POINTS	кожна вершина задає координати точки
GL_LINES	кожна окрема пара вершин визначає відрізок; якщо задане непарне число вершин, то остання вершина ігнорується
GL_LINE_STRIP	кожна наступна вершина задає відрізок разом з попередньою
GL_LINE_LOOP	відмінність від попереднього примітива тільки в тому, що останній відрізок визначається останньою і першою вершиною, утворюючи замкнуту ламану
GL_TRIANGLES	кожні окремі три вершини визначають трикутник; якщо задане не кратне трьом число вершин, то останні вершини ігноруються
GL_TRIANGLE_STRIP	кожна наступна вершина задає трикутник разом із двома попередніми
GL_TRIANGLE_FAN	трикутники задаються першою вершиною і кожною наступною парою вершин (пари не перетинаються)
GL_QUADS	кожна окрема четвірка вершин визначає чотирикутник; якщо задане не кратне

	чотирьом число вершин, то останні вершини ігноруються
GL_QUAD_STRIP	чотирикутник з номером n визначається вершинами з номерами 2n-1, 2n, 2n+2, 2n+1
GL_POLYGON	послідовно задаються вершини *опуклого* багатокутника

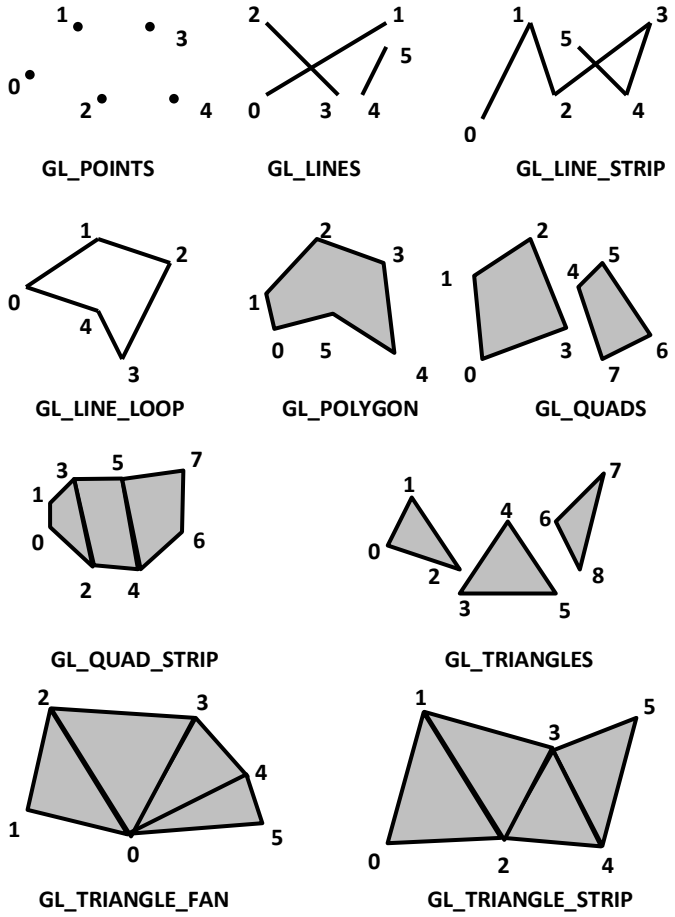


Рис. 19.1. Примітиви OpenGL

Наприклад, щоб намалювати трикутник в проекті Delpi з різними кольорами у вершинах, досить написати:

```
const
  BlueColor : array [0..2] of GLfloat = (0,0,1);
…
glBegin(GL_TRIANGLES);
  glColor3f(1.0, 0.0, 0.0);//червоний
  glVertex3f(0.0, 0.0, 0.0);
  glColor3ub(0,255,0);    //зелений
```

```
    glVertex3f(1.0, 0.0, 0.0);
    glColor3fv(@BlueColor); //синій
    glVertex3f(1.0, 1.0, 0.0);
  glEnd;
```

В проекті C++ Builder код буде відрізнятися лише рядком з оголошенням масиву кольору і відповідно у виклику його з команди glColor3fv зникне символ @.

Як правило, різні типи примітивів мають різну швидкість візуалізації на різних платформах. Для збільшення продуктивності краще використовувати примітиви, які передають на сервер меншу кількість інформації (GL_TRIANGLE_STRIP, GL_QUAD_STRIP, GL_TRIAGLE_FAN).

Крім завдання самих багатокутників, можна визначити і метод їх відображення на екрані. Однак спочатку треба визначити поняття лицьових і зворотних граней.

Під *гранню* розуміється одна зі сторін багатокутника, і за замовчуванням лицьовою вважається та сторона, вершини якої обходяться проти годинникової стрілки. Напрямок обходу вершин лицьових граней можна змінити викликом команди

glFrontFace (*mode*: GLenum)

зі значенням параметра mode рівним GL_CW (clockwise), а повернути значення за замовчуванням можна константою GL_CCW (counter-clockwise).

Для зміни методу відображення багатокутника використовується команда

glPolygonMode (*face*: GLenum, *mode*: GLenum).

Параметр mode визначає, як будуть відображатися багатокутники, а параметр face встановлює тип багатокутників, до яких буде застосовуватися ця команда і може приймати наступні значення:

GL_FRONT	для лицевих граней
GL_BACK	для зворотних граней
GL_FRONT_AND_BACK	для всіх граней

Параметр mode може дорівнювати:

GL_POINT	відображення тільки вершин багатокутників
GL_LINE	багатокутники будуть подаватися набором відрізків
GL_FILL	багатокутники будуть зафарбовуватися поточним

	кольором з урахуванням освітлення (цей режим встановлений за замовчуванням)

Також можна вказувати, який тип граней відображати на екрані. Для цього спочатку треба встановити відповідний режим викликом команди glEnable(GL_CULL_FACE), а потім вибрати тип відображення граней за допомогою команди:

```
glCullFace (mode: GLenum)
```

Виклик з параметром GL_FRONT приводить до видалення з зображення всіх лицьових граней, а з параметром GL_BACK – зворотних (установка за замовчуванням).

19.4. Штрихування багатокутників

По замовчуванню багатокутники малюються з шаблоном суцільного замальовування. Вони також можуть бути залиті шаблоном штрихування розміром 32×32 біта, який вирівняний відносно сторін вікна, яке задається за допомогою команди:

```
glPolygonStipple (const GLUbyte @mask).
```

Ця команда визначає поточний шаблон штрихування для заповнення багатокутників. Параметр mask є покажчиком на растрове зображення розміром 32×32, яке інтерпретується як маска з 0 та 1. Там, де у масці з'являється одиниця, відображається відповідний піксел в багатокутнику.

В доповнення до визначення поточного шаблону штрихування необхідно включити режим штрихування glEnable (GL_POLYGON_STIPPLE).

Для того щоб відключити режим штрихування багатокутників необхідно застосувати команду glDisable() з тим самим параметром.

Приклад.

```
const
 grating : array [0..127] of GLUbyte =
 ($18, $00, $18, $00, $18, $00, $18, $00,
……);
…
glEnable(GL_POLYGON_STIPPLE);
glPolygonStipple(@grating);
glColor3f (0.0, 0.0, 0.0);
glBegin(GL_QUADS);
```

```
    glVertex2f (-0.5, 0.5);
    glVertex2f (-0.5, -0.5);
    glVertex2f (0.5, -0.5);
    glVertex2f (0.5, 0.5);
  glEnd;
  glDisable(GL_POLYGON_STIPPLE);
```

Штрихування можна застосовувати до будь-яких фігур, що побудовані з використанням багатокутників.

19.5. Примітиви бібліотек GLU і GLUT

Крім розглянутих стандартних примітивів, у бібліотеках GLU і GLUT описані і більш складні фігури, такі як сфера, циліндр, диск (в GLU) і сфера, куб, конус, тор, тетраедр, додекаедр, ікосаедр, октаедр і чайник (в GLUT). Автоматичне накладання текстури передбачене тільки для фігур з бібліотеки GLU (створення текстур у OpenGL буде розглядатися в розділі 22).

Наприклад, щоб побудувати примітив з бібліотеки GLU, потрібно ввести змінну спеціального типу:

```
quadObj : GLUquadricObj;
```

Далі у програмі викличемо команду, яка створює quadric-об'єкт:

```
quadObj := gluNewQuadric;
```

а потім викличемо одну з команд gluSphere, gluCylinder, gluDisk, gluPartialDisk.

Режим відображення об'єкту задається командою gluQuadricDrawStyle, першим аргументом якої вказується ім'я об'єкту, а другим службове слово, яке встановлює стиль відображення об'єкту (POINT, LINE, FILL, SILHOUETTE).

Команда gluQuadricOrientation задає напрямок нормалей до поверхні об'єкта (OUTSIDE, INSIDE).

Команда gluQuadricNormals визначає чи будуються нормалі для кожної вершини, для сегменту чи не будуються зовсім (SMOOTH, FLAT, NONE).

По закінченні роботи програми, до звільнення контексту відображення, необхідно звільнити пам'ять, яку використовують quadric-об'єкти:

```
gluDeleteQuadric (quadObj);
```

Розглянемо команди побудови quadric-об'єктів.

gluSphere (quadObj, *radius*: GLdouble, *slices*: GLint, *stacks*: GLint)

Ця функція будує сферу з центром в початку системи координат і радіусом radius. При цьому число розбиття сфери навколо осі Z задається параметром slices, а уздовж осі Z – параметром stacks.

gluCylinder (quadObj, *baseRadius*: GLdouble, *topRadius*: GLdouble, *height*: GLdouble, *slices*: GLint, *stacks*: GLint)

Дана функція будує циліндр без основ (тобто кільце), поздовжня вісь рівнобіжна осі Z, задня основа має радіус baseRadius, і розташована в площині Z=0, а передня основа має радіус topRadius і розташована в площині Z=height. Якщо задати один з радіусів рівним нулю, то буде побудований конус. Параметри slices і stacks мають аналогічну семантику, що й у попередній команді.

gluDisk(*qobj*: ^GLUquadricObj, *innerRadius*: GLdouble, *outerRadius*: GLdouble, *slices*: GLint, *loops*: GLint)

Функція будує плоский диск (коло) з центром в початку системи координат і радіусом outerRadius. При цьому якщо значення innerRadius відмінно від нуля, то в центрі диска буде знаходитися отвір радіусом innerRadius. Параметр slices задає число розбивок диска навколо осі Z, а параметр loops – число концентричних кілець, перпендикулярних осі Z.

gluPartialDisk(*qobj*: ^GLUquadricObj, *innerRadius*: GLdouble, *outerRadius*: GLdouble, *slices*: GLint, *loops*: GLint, *startAngle*: GLdouble, *sweepAngle*: GLdouble)

Відмінність цієї команди від попередньої полягає в тому, що вона будує сектор кола, початковий і кінцевий кути якого відраховуються проти годинникової стрілки від позитивного напрямку осі Y і задаються параметрами startAngle і sweepAngle. Кути вимірюються в градусах.

Команди, що проводять побудову примітивів з бібліотеки GLUT, реалізовані через стандартні примітиви і команди OpenGL і GLU. Для побудови потрібного примітива досить зробити виклик відповідної команди.

glutSolidSphere(*radius*: GLdouble, *slices*: GLint, *stacks*: GLint)

glutWireSphere(*radius*: GLdouble, *slices*: GLint, *stacks*: GLint)

Команда glutSolidSphere будує сферу, а glutWireSphere – каркас сфери радіусом radius. Інші параметри ті ж, що й у попередніх командах.

glutSolidCube(*size*: GLdouble)

glutWireCube(*size*: GLdouble)

Команди будують куб чи каркас куба з центром в початку системи координат і довжиною ребра size.

glutSolidCone(*base*: GLdouble, *height*: GLdouble, *slices*: GLint, *stacks*: GLint)

glutWireCone(base: GLdouble, height:GLdouble, *slices*: GLint, *stacks*: GLint)

Ці команди будують конус чи його каркас висотою height і радіусом основи base, розташований уздовж осі Z. Основа знаходиться в площині Z=0.

glutSolidTorus(*innerRadius*: GLdouble, *outerRadius*: GLdouble, *nsides*: GLint, *rings*: GLint)

glutWireTorus(*innerRadius*: GLdouble, *outerRadius*: GLdouble, *nsides*: GLint, *rings*: GLint)

Ці команди будують тор чи його каркас у площині Z=0. Внутрішній і зовнішній радіуси задаються параметрами innerRadius, outerRadius. Параметр nsides задає число сторін у кільцях, що складають ортогональний перетин тора, а rings – число радіальних розбивок тора.

glutSolidTetrahedron

glutWireTetrahedron

Ці команди будують тетраедр (правильну трикутну піраміду) чи його каркас, при цьому радіус описаної сфери довкола нього дорівнює 1.

glutSolidOctahedron

glutWireOctahedron

Ці команди будують октаедр чи його каркас, радіус описаної довкола нього сфери дорівнює 1.

glutSolidDodecahedron

glutWireDodecahedron

Ці команди будують додекаедр чи його каркас, радіус описаної довкола нього сфери дорівнює квадратному кореню з трьох.

glutSolidIcosahedron

glutWireIcosahedron

Ці команди будують ікосаедр чи його каркас, радіус описаної довкола нього сфери дорівнює 1.

Для коректної побудови перерахованих примітивів необхідно видаляти невидимі лінії і поверхні, для чого треба увімкнути відповідний режим командою glEnable(GL_DEPTH_TEST) .

19.6. Tess-об'єкти

Мозаїчні (*tesselated*) об'єкти – є одним з останніх нововведень бібліотеки GLU, вони призначені для спрощення побудови неопуклих багатокутників. Робота з ними у Delphi, пов'язана з деякими незручностями через помилку у заголовному файлі.

У програмі вводиться спеціальний тип для зберігання координат однієї точки:

```
type TVector = array [0..2] of GLdouble;
```

Необхідно перед розділом implementation переписати прототип однієї з команд:

```
procedure gluTessBeginPolygon (tess: GLUtesselator;
polygon_data: ^TVector); stdcall; external GLU32;
```

Необхідно ініціалізувати змінну спеціального типу, введеного для роботи з мозаїчними об'єктами:

```
var tobj: gluTesselator;
…
tobj := gluNewTess;
```

За допомогою команди gluTessCallBack задаються адреси процедур, які викликаються на різних етапах малювання tess-об'єкта. Якщо при малюванні не планується вводити додаткові операції, то передаються наступні адреси процедур для проекту Delphi:

```
gluTessCallback(tobj, GLU_TESS_BEGIN, @glBegin);
gluTessCallback(tobj, GLU_TESS_VERTEX, @glVertex3dv);
gluTessCallback(tobj, GLU_TESS_END, @glEnd);
```

В проекті C++ Builder ці команди виглядають наступним чином:

```
gluTessCallback(tobj,GLU_TESS_BEGIN,(void(CALLBACK*)(void
))glBegin);
gluTessCallback(tobj,GLU_TESS_VERTEX,(void(CALLBACK*)(voi
d))glVertex3dv);
gluTessCallback(tobj,GLU_TESS_END,(void(CALLBACK*)(void))
glEnd);
```

Побудова багатокутника відбувається наступним чином:

```
gluTessBeginPolygon(tobj, nil);
```

```
    gluTessBeginContour (tobj);
    for j:=0 to 7 do gluTessVertex(tobj,@poly[j],@poly[j]);
    gluTessEndContour (tobj);
  gluTessEndPolygon (tobj);
```

Де poly це:

```
const
 poly: array [0..7] of TVector = (
 (50.0 ,200.0 ,0.0),(50.0 ,50.0 ,0.0),
 (200.0 ,50.0 ,0.0),(200.0 ,90.0 ,0.0),
 (100.0 ,90.0 ,0.0),(100.0 ,160.0 ,0.0),
 (200.0 ,160.0 ,0.0),(200.0 ,200.0 ,0.0));
```

Слід зауважити що для правильної побудови багатокутника його точки необхідно задавати у напрямку проти годинникової стрілки.

Якщо в полігоні tess-об'єкта задати декілька контурів, то перший контур буде вважатися зовнішнім, а всі наступні – його отворами. Але треба уважно слідкувати за координатами отворів. Якщо вони будуть задані не вірно, і отвір перетинатиме зовнішній контур, то tess-об'єкт не буде виведений на екран. Приклад програми, що використовує tess-об'єкт наведено в *додатку 4*.

19.7. Криві та поверхні

19.7.1. Криві та поверхні Без'є

Криві Без'є в бібліотеці OpenGL задаються за допомогою одновимірного обчислювача:

```
glMap1 [f d](target: GLenum, u1: GLfloat, u2: GLfloat,
stride: GLuint, order: GLuint, points: ^GLfloat).
```

Параметр target визначає, які значення будуть розраховані і може приймати наступні значення:

GL_MAP1_VERTEX_3	координати вершин (x,y,z)
GL_MAP1_VERTEX_4	координати вершин (x,y,z,w)
GL_MAP1_INDEX	індекс кольору
GL_MAP1_COLOR_4	компоненти кольору (R,G,B,A)
GL_MAP1_NORMAL	координати нормалі
GL_MAP1_TEXYURE_COORD_1	координати текстури s
GL_MAP1_TEXYURE_COORD_2	координати текстури (s,t)

GL_MAP1_TEXYURE_COORD_3	координати текстури (s,t,r)
GL_MAP1_TEXYURE_COORD_4	координати текстури (s,t,r,q)

u1 та u2 – кінцеві точки інтервалу кривої, що обчислюється (зазвичай від 0 до 1).

stride – задає кількість чисел, що містяться у одній порції даних.

order – кількість опорних (визначаючих) точок.

points – покажчик на масив опорних точок.

Для побудови кривої використовується команда:

```
glEvalCoord1 [f d](u: GLfloat)
```

***Приклад*.**

```
glBegin(GL_POINTS); //or GL_LINE_STRIP
  for i:=0 to 30 do glEvalCoord1f (i/30);
glEnd;
```

Альтернативою може слугувати використання команд:

```
glMapGrid1 [f d](n: GLint, u1: GLint, u2: GLint)
```

Визначає одновимірну сітку від u1 до u2 з кроком n.

```
glEvalMesh1 (mode: GLenum, p1: GLint, p2: GLint)
```

Параметр mode може приймати значення GL_POINT чи GL_LINE.

***Приклад*.**

```
glMapGrid1f(30, 0, 1);
glEvalMesh1(GL_POINT,0,30);
```

Поверхні Без'є формуються у OpleGL приблизно за тією ж методикою, що і криві, тільки роль функції ініціалізації відіграє не glMap1*, а glMap2*, а для зчитування результатів необхідно звертатися до функції glEvalCoord2*.

```
glMap2 [f d](type: GLenum, u_min: GLfloat, u_max: GLfloat, u_stride: GLint, u_order: GLint, v_min: GLfloat, v_max: GLfloat, v_stride: GLint, v_order: GLint, point_array: ^GLfloat)
```

Перший параметр type – константа, яка визначає тип величин, що розраховуються.

u_min, u_max, v_min, v_max – задають визначаючу полігональну сітку.

u_stride, v_stride – кількість значень параметра між сегментами, кількість чисел у порції даних.

u_order та v_order – задають порядок полігона.

Приклад.

```
glMap2f(GL_MAP_VERTEX_3,0.0,1.0,3,4,0.0,1.0,12,4,@points)
```

Значення v_stride для другого параметру дорівнює 12, тому що для того, щоб дістатися необхідних даних потрібно перескочити через 3×4=12 чисел у форматі float.

```
for j:=0 to 100 do begin
  glBegin(GL_LINE_STRIP); //or GL_QUAD_STRIP
    for i:=0 to 100 do glEvalCoord2f(i/100,j/100);
  glEnd;
  glBegin(GL_LINE_STRIP); //or GL_QUAD_STRIP
    for i:=0 to 100 do glEvalCoord2f (j/100,i/100);
  glEnd;
end;
```

Якщо включене обчислення вершин (GL_MAP2_VERTEX_3 чи GL_MAP2_VERTEX_4), то нормаль до поверхні розраховується аналітично. Ця нормаль зв'язується зі згенерованою вершиною, якщо включена автоматична генерація векторів нормалі командою glEnable(GL_AUTO_NORMAL).

Для роботи на рівномірній сітці параметрів необхідно використовувати функції glMapGrid2* та glEvalMesh2.

19.7.2. NURBS-КРИВІ ТА ПОВЕРХНІ

NURBS один з класів B-сплайнів – раціональні B-сплайни, що задаються не нерівномірній сітці (*Non-Uniform Rational B-spline*, *NURBS*) є стандартним для комп'ютерної графіки способом визначення параметричних кривих та поверхонь.

Бібліотека GLU надає набір команд, що дозволяють використовувати даний клас поверхонь.

Для роботи з NURBS-об'єктами у бібліотеці GLU містяться змінні спеціального типу:

```
theNurb : gluNurbsObj;
```

Для використання NURBS-об'єкта необхідно спочатку його створити:

```
theNurb := gluNewNurbsRenderer;
```

Після завершення роботи з ним, необхідно видалити об'єкт:

```
gluDeleteNurbsRenderer(theNurb: gluNurbsObj);
```

Для роботи з NURBS-поверхнями є п'ять основних функцій:

```
gluNewNurbsRenderer();
gluNurbsProperty();
gluBeginNurbsSurface();
gluNurbsSurface();
gluEndNurbsSurface();
```

Дві перші функції формують новий об'єкт та задають спосіб його відображення. Наступні три функції використовуються для формування поверхні. При роботі з NURBS-кривими замість них необхідно використовувати gluBeginNurbsCurve, gluNurbsCurve, gluEndNurbsCurve.

Розглянемо основні команди більш детально. Команда, яка будує криву, має наступні параметри:

```
gluNurbsCurve (theNurb: gluNurbsObj, kcount: GLint,
cknots: ^GLfloat, stride: GLint, points: ^GLfloat, order:
GLint, type: GLenum)
```

Перший параметр ім'я NURBS-об'єкта, для якого будується крива.

kcount – кількість параметричних вузлів, повинна дорівнювати кількості опорних точок плюс порядок кривої.

cknots – покажчик на масив, що зберігає координати вузлів.

stride – зміщення, тобто кількість дійсних чисел, що містяться в одній порції даних.

points – покажчик на масив, що зберігає координати опорних точок.

order – порядок кривої плюс одиниця.

type – визначає тип кривої. Якщо команда викликається всередині пари gluBeginNurbsCurve/gluEndNurbsCurve то можливі значення аналогічні до значень одновимірного обчислювача glMap1, якщо ж команда викликається всередині пари gluBeginTrim/gluEndTrim то єдині допустимі значення – GLU_MAP1_TRIM_2 і GLU_MAP1_TRIM_3.

Пара gluBeginTrim/gluEndTrim визначає завдання області вирізки в існуючому NURBS-об'єкті. Звичайно вирізка задається замкнутою кривою у вигляді команди gluNurbsCurve, або частинами лінійної кривої, що задається командою gluPwlCurve.

***Приклад*.**

```
gluBeginCurve(theNurb);
gluNurbsCurve(theNurb,8,@ck,3,@pts,4,GL_MAP1_VERTEX_3);
```

```
gluEndCurve(theNurb);
```

Після команди gluBeginSurface один або декілька викликів команди gluNurbsSurface визначають атрибути поверхні. Один з таких викликів повинен обов'язково задати тип поверхні GL_MAP2_VERTEX_3 або GL_MAP2_VERTEX_4 для генерації вершин. Команда gluEndSurface використовується для закінчення опису поверхні. Вирізання NURBS-поверхонь також підтримується між gluBeginSurface та gluEndSurface. Синтаксис команди для побудови поверхні наступний:

gluNurbsSurface(*theNurb*: gluNurbsObj, *knot_count*: GLint, *uknot*: ^GLfloat, *vknot_count*: GLint, *vknot*: ^GLfloat, *u_stride*: GLint, *v_stride*: GLint, *ctlarray*: ^GLfloat, *uorder*: GLint, *vorder*: GLint, *type*: GLenum)

Ця команда описує вершини (або поверхневі нормалі чи текстурні координати) NURBS-поверхні theNurb. Деякі значення повинні бути визначені для обох параметричних напрямків u та v. Це послідовності вузлів (uknot та vknot), кількість вузлів (uknot_count та vknot_count) і порядок поліномів (uonder та vonder) для NURBS-поверхонь. Зверніть увагу, що кількість контрольних точок не визначена. Замість цього задається кількість контрольних точок по кожному параметру як кількість вузлів мінус порядок (knot-order). Тоді кількість контрольних точок для поверхні дорівнює кількості контрольних точок в кожному параметричному напрямку, помноженим один на одного. Параметр ctlarray вказує на масив контрольних точок.

Останній параметр – type, визначає один з двохвимірних типів обчислювача. Звичайно використовується тип GL_MAP2_VERTEX3 для нераціональних контрольних точок чи GL_MAP2_VERTEX4 для раціональних контрольних точок. Також можна використовувати інші типи, наприклад, GL_MAP2_TEXTURE_COORD_* або GL_MAP2_NORMAL для обчислення та призначення текстурних координат чи нормалей до поверхні. Наприклад, для створення освітленої (з нормалями до поверхні) та текстурованої NURBS-поверхні потрібно викликати наступну послідовність команд:

```
gluBeginSurface(theNurb);
  gluNurbsSurface(theNurb,..., GL_MAP2_TEXTURE_COORD_2);
  gluNurbsSurface(theNurb,..., GL_MAP2_NORMAL);
  gluNurbsSurface(theNurb,..., GL_MAP2_VERTEX_3);
gluEndSurface(theNurb);
```

Параметри u_stride та v_stride представляють собою кількість значень з плаваючою точкою між контрольними точками в кожному параметричному напрямку. Тип обчислювача та його порядок впливають на значення параметрів u_stride та v_stride. Значення u_stride повинно дорівнювати vorder*3 або vorder*4, в залежності від того який режим координат заданий – нераціональний (3 координати) чи раціональний (4 координати).

19.8. Дисплейні списки

Якщо ми кілька разів звертаємося до однієї і тієї ж групи команд, то їх можна об'єднати в так званий дисплейний список (display list), і викликати його за необхідності. Для того, щоб створити новий дисплейний список, треба помістити всі команди, що повинні в нього ввійти, між наступними операторними дужками:

```
glNewList (list: GLuint, mode: GLenum)
glEndList
```

Для ідентифікації списків використовуються цілі позитивні числа, що задаються при створенні списку значенням параметра list, а параметр mode визначає режим обробки команд, що входять у список:

GL_COMPILE	команди записуються в список без виконання
GL_COMPILE_AND_EXECUTE	команди спочатку виконуються, а потім записуються в список

Створений список можна викликати командою

```
glCallList (list: GLuint)
```

вказавши в параметрі list його ідентифікатор. Викликати відразу кілька списків можна командою

```
glCallLists (n: GLsizei, type: GLenum, lists: pointer)
```

яка викликає n списків з ідентифікаторами з масиву lists, тип елементів якого вказується в параметрі type. Це можуть бути типи GL_BYTE, GL_UNSIGNED_BYTE, GL_SHORT, GL_INT, GL_UNSIGNED_INT і деякі інші. Для видалення списків використовується команда

```
glDeleteLists (list: GLint, range: GLsizei)
```

яка видаляє списки з ідентифікаторами ID з діапазону list≤ID≤list+range-1.

Приклад.

```
glNewList(1, GL_COMPILE);
```

```
  glBegin(GL_TRIANGLES);
    glVertex2f(0.0, 0.0);
    glVertex2f(1.0, 0.0);
    glVertex2f(0.0, 1.0);
  glEnd;
glEndList;
…
glCallList(1);
```

Дисплейні списки зберігаються в пам'яті сервера в оптимальному, скомпільованому вигляді, що дозволяє малювати їх примітиви максимально швидко. У той же час, надто великі обсяги даних списків займають багато пам'яті, що теж призводить до падіння продуктивності.

19.9. Масиви вершин

Якщо вершин багато, то щоб не викликати для кожної з них команду glVertex*, зручно поєднувати вершини в масиви командою

glVertexPointer (*size*: GLint, *type*: GLenum, *stride*: GLsizei, *ptr*: pointer)

яка визначає спосіб відображення і координати вершин. Параметр size задає кількість координат вершини (2, 3 або 4), а type визначає тип даних (GL_SHORT, GL_INT, GL_FLOAT або GL_DOUBLE). Іноді зручно зберігати в одному масиві інші атрибути вершини, тоді параметр stride задає зсув від координат однієї вершини до координат наступної; якщо stride дорівнює нулю, це значить, що координати розташовані послідовно. У параметрі ptr вказується адреса, за якою знаходяться дані. Аналогічно командами

glNormalPointer (*type*: GLenum, *stride*: GLsizei, *ptr*: pointer)

glColorPointer (*size*: GLint, *type*: GLenum, *stride*: GLsizei, *ptr*: pointer)

можна визначити масив нормалей, кольорів і деяких інших атрибутів вершини. Для того, щоб ці масиви надалі можна було використовувати, потрібно викликати команду

glEnableClientState (*array*: GLenum)

з параметрами GL_VERTEX_ARRAY, GL_NORMAL_ARRAY, GL_COLOR_ARRAY відповідно. Після закінчення роботи з масивом бажано викликати команду

glDisableClientState (array: GLenum)

з відповідним значенням параметра array.

Для відображення вмісту масивів використовується команда

glArrayElement (*index*: GLint)

яка передає OpenGL атрибути вершини, використовуючи елементи масиву з номером index. Це аналогічно послідовному застосуванню команд виду glColor, glNormal, glVertex з відповідними параметрами. Однак частіше застосовується команда

glDrawArrays (*mode*: GLenum, *first*: GLint, *count*: GLsizei)

яка малює count примітивів, визначених параметром mode, використовуючи елементи з масивів з індексами від first до first+count-1. Це еквівалентно виклику послідовності команд glArrayElement з відповідними індексами.

У випадку, якщо одна вершина входить у кілька примітивів, то замість дублювання її координат у масиві зручно використовувати її індекс. Для цього треба викликати команду

glDrawElements (*mode*: GLenum, *count*: GLsizei, *type*: GLenum, *indices*: pointer)

де indices – це масив номерів вершин, які треба використовувати для побудови примітивів, type визначає тип елементів цього масиву: GL_UNSIGNED_BYTE, GL_UNSIGNED_SHORT, GL_UNSIGNED_INT, а count задає їх кількість.

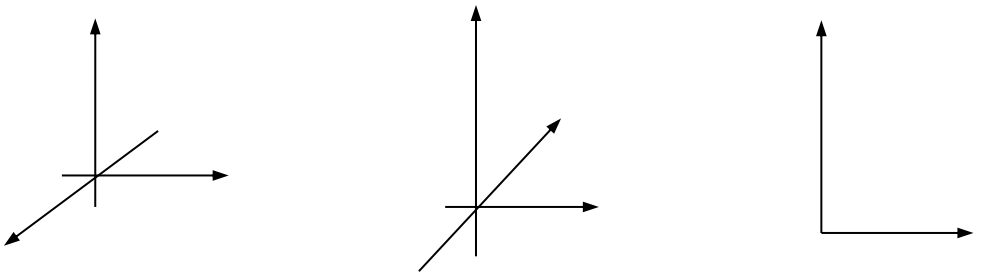
Слід зауважити, що використання масивів вершин дозволяє оптимізувати передачу даних на сервер OpenGL, і, як наслідок, підвищити швидкість малювання тривимірної сцени. Такий метод визначення примітивів є одним з найшвидших і добре підходить для візуалізації великих обсягів даних.

Запитання для самоконтролю

1. Що таке примітив?
2. Що в OpenGL є атомарним примітивом?
3. Для чого в OpenGL використовуються команди glEnable/glDisable?
4. Що таке дисплейні списки?

20. Перетворення об'єктів

В OpenGL використовуються як основні три системи координат: *лівобічна*, *правобічна* і *віконна* (рис. 20.1). Перші дві системи є тривимірними і відрізняються одна від одної напрямком осі Z: у правобічній вона спрямована на спостерігача, в лівобічній – у глибину екрана. Вісь X спрямована вправо щодо спостерігача, вісь Y – вгору.



а) правобічна **б) лівобічна** **в) віконна**

Рис. 20.1. Системи координат в OpenGL

OpenGL дозволяє шляхом маніпуляцій з матрицями моделювати як праву, так і ліву систему координат. Але на даному етапі краще піти простим шляхом і запам'ятати: основною системою координат OpenGL є *правобічна* система.

20.1. Робота з матрицями

Для завдання різних перетворень об'єктів в OpenGL використовуються операції над матрицями, при цьому розрізняються три типи матриць: модельно-видова, матриця проекцій і матриця текстури. Усі вони мають розмір 4x4. Видова матриця визначає перетворення об'єкта у світових координатах, такі як паралельний перенос, зміна масштабу і поворот. Матриця проекцій визначає, як будуть проектуватися тривимірні об'єкти на площину екрану (у віконні координати), а матриця текстури визначає накладання текстури на об'єкт.

Множення координат на матриці відбувається в момент виклику відповідної команди OpenGL, яка визначає координату (як правило, це команда `glVertex`).

Для того щоб вибрати, яку матрицю треба змінити, використовується команда:

```
glMatrixMode (mode: GLenum)
```

виклик якої зі значенням параметра `mode` рівним `GL_MODELVIEW`, `GL_PROJECTION`, або `GL_TEXTURE` включає режим роботи з модельно-видовою матрицею, матрицею проекцій чи матрицею текстури

відповідно. Для виклику команд, що задають матриці того чи іншого типу, необхідно спочатку встановити відповідний режим.

Для визначення елементів матриці поточного типу викликається команда

glLoadMatrix[f d] (*m*: ^GLtype)

де m вказує на масив з 16 елементів типу float чи double відповідно до назви команди, при чому спочатку в ньому повинний бути записаний перший стовпець матриці, потім другий, третій і четвертий. Ще раз звернемо увагу: у масиві m матриця записана по стовпцях.

Команда

glLoadIdentity

заміняє поточну матрицю на одиничну.

Часто буває необхідно зберегти вміст поточної матриці для подальшого використання, для чого застосовуються команди:

glPushMatrix

glPopMatrix.

Вони записують і відновлюють поточну матрицю зі стека, причому для кожного типу матриць використовується свій стек. Для модельно-видових матриць максимальна глибина стеку 32, для інших – 2.

Для множення поточної матриці на іншу матрицю використовується команда

glMultMatrix[f d] (*m*: ^GLtype)

де параметр m повинний задавати матрицю розміром 4×4. Якщо позначити поточну матрицю за M, передану матрицю за T, то в результаті виконання команди glMultMatrix поточною стає матриця M*T. Однак звичайно для зміни матриці того чи іншого типу зручно використовувати спеціальні команди, що за значеннями своїх параметрів створюють потрібну матрицю і множать її на поточну.

Послідовність перетворень координат для відображення об'єктів сцени у вікно застосунку зображена на рис. 20.2. Запам'ятайте: усі перетворення об'єктів і камери в OpenGL відбуваються за допомогою множення векторів координат на матриці. Причому множення відбувається на *поточну матрицю* в момент визначення координати командою glVertex і деякими іншими.

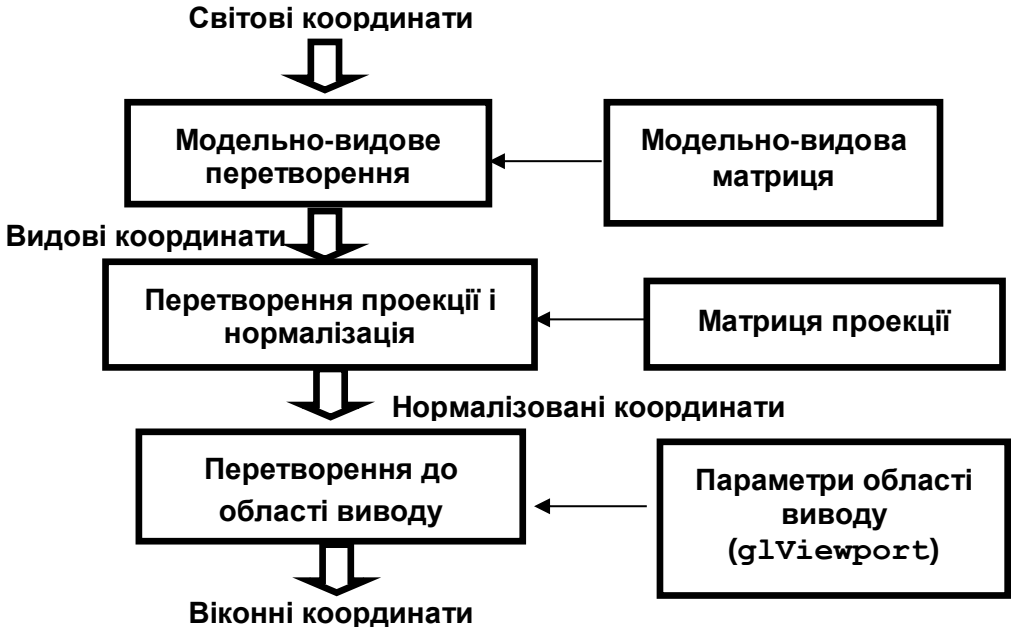


Рис. 20.2. Перетворення координат в OpenGL

20.2. Модельно-видові перетворення

До модельно-видових перетворень відносяться *переміщення*, *поворот* і *масштабування* вздовж координатних осей. Для проведення цих операцій досить помножити на відповідну матрицю кожну вершину об'єкта і одержати змінені координати цієї вершини:

$$(\mathrm{x}', \mathrm{y}', \mathrm{z}', 1)^{\mathrm{T}} = \mathrm{M} \cdot (\mathrm{x}, \mathrm{y}, \mathrm{z}, 1)^{\mathrm{T}},$$

де M – матриця модельно-видового перетворення. Перспективне перетворення і проекція виконуються аналогічно. Сама матриця може бути створена за допомогою наступних команд:

glTranslate[f d](*x*: GLtype, *y*: GLtype, *z*: GLtype)

glRotate[f d](*angle*: GLtype, *x*: GLtype, *y*: GLtype, z: GLtype)

glScale[f d](*x*: GLtype, *y*: GLtype, *z*: GLtype)

glTranslate – переміщує об'єкт, додаючи до координат його вершин значення своїх параметрів.

glRotate – повертає об'єкт проти годинникової стрілки на кут angle (задається у градусах) навколо вектора (x,y,z).

glScale – забезпечує масштабування об'єкта (розтягнення або стиснення) уздовж вектора (x,y,z), перемножуючи відповідні координати його вершин на значення своїх параметрів.

Усі ці перетворення змінюють поточну матрицю, а тому застосовуються до примітивів, що визначаються пізніше. У випадку, якщо треба, наприклад, повернути один об'єкт сцени, а інший залишити нерухомим, зручно спочатку зберегти поточну видову матрицю в стеку командою glPushMatrix, потім викликати glRotate з відповідними параметрами, описати примітиви, з яких складається цей об'єкт, а потім відновити поточну матрицю командою glPopMatrix.

Крім зміни положення самого об'єкта, часто буває необхідно змінити положення спостерігача, що також приводить до зміни модельно-видової матриці. Це можна зробити за допомогою команди

gluLookAt (*eyex*: GLdouble, *eyey*: GLdouble, *eyez*: GLdouble, *centerx*: GLdouble, *centery*: GLdouble, *centerz*: GLdouble, *upx*: GLdouble, *upy*: GLdouble, *upz*: GLdouble),

де точка (eyex, eyey, eyez) визначає точку спостереження, (centerx, centery, centerz) задає центр сцени, що буде проектуватися в центр області виводу, а вектор (upx, upy, upz) задає позитивний напрямок осі y, визначаючи поворот камери. Якщо, наприклад, камеру не треба повертати, то задається значення (0,1,0), а зі значенням (0,-1,0) сцена буде перевернута.

Ця команда робить переміщення і поворот об'єктів сцени, але в такому вигляді задавати параметри буває зручніше. Слід зазначити, що викликати команду gluLookAt має сенс перед визначенням перетворень об'єктів, коли модельно-видова матриця дорівнює одиничній.

Запам'ятайте: в загальному випадку матричні перетворення в OpenGL потрібно записувати в зворотному порядку. Наприклад, якщо ви хочете спочатку повернути об'єкт, а потім пересунути його, спочатку викличте команду glTranslate, а тільки потім – glRotate. Ну а після цього визначайте сам об'єкт.

20.3. Проекції

В OpenGL існують стандартні команди для завдання ортографічної (паралельної) і перспективної проекцій. Перший тип проекції (рис. 20.3) може бути заданий командами

glOrtho (*left*: GLdouble, *right*: GLdouble, *bottom*: GLdouble, *top*: GLdouble, *znear*: GLdouble, *zfar*: GLdouble)

gluOrtho2D (*left*: GLdouble, *right*: GLdouble, *bottom*: GLdouble, *top*: GLdouble)

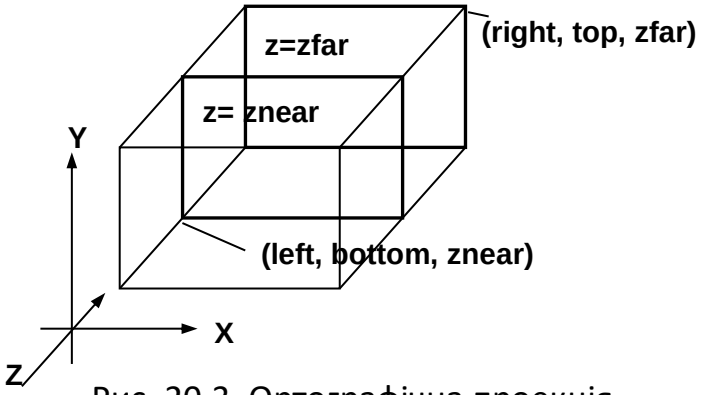


Рис. 20.3. Ортографічна проекція

Перша команда створює матрицю проекції в зрізаний об'єм видимості (паралелепіпед видимості) у лівобічній системі координат. Параметри команди задають точки (`left`, `bottom`, `znear`) і (`right`, `top`, `zfar`), які відповідають лівому нижньому і правому верхньому кутам вікна виводу. Параметри `znear` і `zfar` задають відстань до ближньої і дальньої площин відсікання по віддаленні від точки (`0`,`0`,`0`) і можуть бути негативними.

В другій команді, на відміну від першої, значення `znear` і `zfar` встановлюються рівними `-1` і `1` відповідно. Це зручно, якщо OpenGL використовується для малювання двовимірних об'єктів. У цьому випадку положення вершин можна задавати, використовуючи команди `glVertex2*`.

Перспективна проекція (рис. 20.4) визначається командою

```
gluPerspective (angley: GLdouble, aspect: GLdouble,
znear: GLdouble, zfar: GLdouble)
```

яка задає зрізаний конус видимості в лівобічній системі координат.

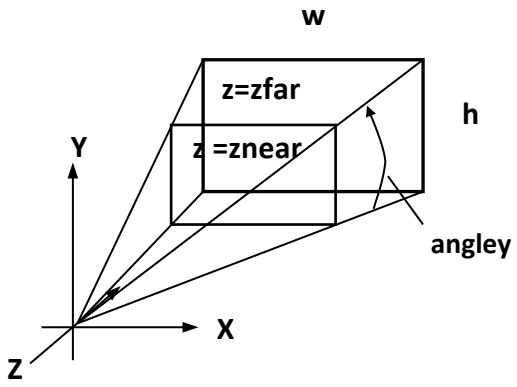


Рис. 20.4. Перспективна проекція

Параметр `angley` визначає кут видимості в градусах по осі Y і повинен знаходитися в діапазоні від `0` до `180`. Кут видимості уздовж осі X задається параметром `aspect`, який звичайно задається як відношення

сторін області виводу (як правило, розмірів вікна). Параметри zfar і znear задають відстань від спостерігача до площин відсікання по глибині і повинні бути позитивними. Чим більше відношення zfar/znear, тим гірше в буфері глибини будуть розрізнятися розташовані поруч поверхні, тому що за замовчуванням в нього буде записуватися «стиснута» глибина в діапазоні від 0 до 1.

Перш ніж задавати матриці проекцій, не забудьте увімкнути режим роботи з потрібною матрицею командою glMatrixMode (GL_PROJECTION) і скинути поточну викликом glLoadIdentity. Наприклад:

```
// ортографічна проекція
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
glOrtho(0, w, 0, h, -1.0, 1.0);
```

20.4. Робоча область

Після застосування матриці проекцій на вхід наступного перетворення подаються так звані відсічені (*clipped*) координати. Потім знаходяться нормалізовані координати вершин за формулою:

$$(x_n, y_n, z_n)^T = \left(\frac{x_c}{w_c}, \frac{y_c}{w_c}, \frac{z_c}{w_c}\right)^T$$

Робоча область є прямокутником у віконній системі координат, розміри якого задаються командою:

glViewPort (*x*: GLint, *y*: GLint, *width*: GLint, *height*: GLint)

Значення всіх параметрів задаються в пікселях та визначають ширину і висоту робочої області з координатами лівого нижнього кута (x,y) у віконній системі координат. Розміри віконної системи координат визначаються поточними розмірами вікна застосунку, точка (0,0) знаходиться в лівому нижньому куті вікна.

Використовуючи параметри команди glViewPort, OpenGL обчислює віконні координати центра робочої області (O_x, O_y) за формулами:

$$O_x = x + \frac{width}{2};\ O_y = y + \frac{height}{2}.$$

Нехай p_x=width, p_y=height, тоді можна знайти віконні координати кожної вершини:

$$(x_w, y_w, z_w)^T = \left(\left[\frac{p_x}{2} \cdot x_n + O_x\right], \left[\frac{p_y}{2} \cdot y_n + O_y\right], \left[\frac{f-n}{2} \cdot z_n + \frac{n+f}{2}\right]\right)^T$$

При цьому цілі позитивні величини `n` і `f` задають мінімальну і максимальну глибину точки у вікні, за замовчуванням рівну `0` та `1` відповідно. Глибина кожної точки записується в спеціальний буфер глибини (*z-буфер*), що використовується для видалення невидимих ліній і поверхонь. Встановити значення `n` і `f` можна викликом функції

```
glDepthRange (n: GLclampd, f: GLclampd)
```

Запитання для самоконтролю

1. Які системи координат використовуються в OpenGL?
2. Перелічіть види матричних перетворень у OpenGL.
3. Що таке матричний стек?
4. Перелічіть способи зміни положення спостерігача в OpenGL.
5. Що означає поняття «робоча область OpenGL»?

21. Матеріали та освітлення

Для створення реалістичних зображень необхідно визначити як властивості самого об'єкта, так і властивості середовища, в якому він знаходиться. Перша група властивостей містить у собі параметри матеріалу, з якого зроблено об'єкт, способи накладення текстури на його поверхню, ступінь прозорості об'єкта. До другої групи можна віднести кількість і властивості джерел світла, рівень прозорості середовища, а також модель освітлення. Усі ці властивості можна задавати відповідними команди OpenGL.

21.1. Модель освітлення

В OpenGL використовується модель освітлення, згідно з якою колір точки визначається декількома факторами: властивостями матеріалу і текстури, величиною нормалі в цій точці, а також положенням джерела світла і спостерігача. Для конкретного розрахунку освітленості в точці треба використовувати одиничні нормалі, однак команди типу `glScale`, можуть змінювати довжину нормалей. Щоб це враховувати, використовуйте вже згадуваний в розділі 19.2.3. режим нормалізації нормалей, що включається викликом команди `glEnable(GL_NORMALIZE)`.

Для завдання глобальних параметрів освітлення використовуються команди

`glLightModel[i f]` (*`pname`*: GLenum, *`param`*: GLenum)

`glLightModel[i f]v` (*`pname`*: GLenum, *`params`*: ^GLtype)

Аргумент `pname` визначає, який параметр моделі освітлення буде настроюватися і може приймати наступні значення:

`GL_LIGHT_MODEL_LOCAL_VIEWER`	параметр `param` повинен бути булевим і задає положення спостерігача. Якщо він дорівнює `GL_FALSE` (за замовчуванням), то напрямок огляду вважається паралельним осі `-z`, в незалежності від положення у видових координатах. Якщо ж він дорівнює `GL_TRUE`, то спостерігач знаходиться в початку видової системи координат. Це може поліпшити якість освітлення, але ускладнює його розрахунок.
`GL_LIGHT_MODEL_TWO_SIDE`	параметр `param` повинний бути булевим і керує режимом розрахунку освітленості, як для лицьових, так і для зворотних граней. Якщо він

	дорівнює GL_FALSE (за замовчуванням), то освітленість розраховується тільки для лицьових граней, якщо GL_TRUE – то розрахунок проводиться і для зворотних граней.
GL_LIGHT_MODEL_AMBIENT	параметр params повинний містити чотири цілих чи дійсних числа, що визначають колір фонового освітлення навіть у випадку відсутності визначених джерел світла. Значення за умовчуванням: (0.2, 0.2, 0.2, 1.0).

21.2. Специфікація матеріалів

Для завдання параметрів поточного матеріалу використовуються команди

glMaterial[i f] (*face*: GLenum, *pname*: GLenum, *param*: GLtype)

glMaterial[i f]v (*face*: GLenum, *pname*: GLenum, *params*: ^GLtype)

За допомогою цих команд можна визначити розсіяний, дифузний і дзеркальний кольори матеріалу, а також ступінь дзеркального відображення й інтенсивність випромінювання світла, якщо об'єкт повинен світитися. Який саме параметр буде визначатися значенням param, залежить від значення pname:

GL_AMBIENT	параметр params повинний містити чотири цілих чи дійсних значення кольорів RGBA (за замовчуванням (0.2, 0.2, 0.2, 1.0)), які визначають розсіяний колір матеріалу (колір матеріалу в тіні)
GL_DIFFUSE	параметр params повинний містити чотири цілих чи дійсних значення кольорів RGBA (за замовчуванням (0.8, 0.8, 0.8, 1.0)), які визначають дифузний колір матеріалу
GL_SPECULAR	параметр params повинний містити чотири цілих чи дійсних значення кольорів RGBA (за замовчуванням (0.0, 0.0, 0.0, 1.0)), які визначають дзеркальний колір матеріалу
GL_SHININESS	параметр params повинний містити одне ціле чи дійсне значення в діапазоні від 0 (за замовчуванням) до 128, яке визначає ступінь

	дзеркального відображення матеріалу
GL_EMISSION	параметр params повинний містити чотири цілих чи дійсних значення кольорів RGBA (за замовчуванням (0.0, 0.0, 0.0, 1.0)), які визначають інтенсивність випромінюваного світла матеріалу
GL_AMBIENT_ AND_DIFFUSE	еквівалентно двом викликам команди glMaterial зі значенням pname GL_AMBIENT і GL_DIFFUSE і однаковими значеннями params

З цього випливає, що виклик команди glMaterial[i f] можливий тільки для установки ступеня дзеркального відображення матеріалу (*shininess*). Команда glMaterial[i f]v використовується для завдання інших параметрів.

Параметр face визначає тип граней, для яких задається цей матеріал і може приймати значення GL_FRONT, GL_BACK або GL_FRONT_AND_BACK.

Якщо в сцені матеріали об'єктів розрізняються лише одним параметром, бажано спочатку встановити потрібний режим, викликавши glEnable з параметром GL_COLOR_MATERIAL, а потім використовувати команду

```
glColorMaterial (face: GLenum, pname: GLenum)
```

де параметр face має аналогічне значення, а параметр pname може приймати всі перераховані значення. Після цього значення обраного за допомогою pname властивості матеріалу для конкретного об'єкта (чи вершини) встановлюються викликом команди glColor, що дозволяє уникнути викликів більш ресурсномісткої команди glMaterial і підвищує ефективність програми. Приклад визначення властивостей матеріалу:

```
const
  mat_dif : array [0..2] of GLFloat = (0.8,0.8,0.8);
  mat_amb : array [0..2] of GLFloat = (0.2, 0.2, 0.2);
  mat_spec :array [0..2] of GLFloat = (0.6, 0.6, 0.6);
  shininess = 0.7 * 128;
  …
  glMaterialfv(GL_FRONT_AND_BACK,GL_AMBIENT, @mat_amb);
  glMaterialfv(GL_FRONT_AND_BACK,GL_DIFFUSE, @mat_dif);
  glMaterialfv(GL_FRONT_AND_BACK,GL_SPECULAR, @mat_spec);
  glMaterialf(GL_FRONT,GL_SHININESS, @shininess);
```

21.3. Опис джерел світла

Визначення властивостей матеріалу об'єкта має сенс, тільки якщо в сцені є джерела світла. Інакше всі об'єкти будуть чорними (точніше будуть мати колір, який дорівнює розсіяному кольору матеріалу). Додати в сцену джерело світла можна за допомогою команд:

glLight[i f] (*light*: GLenum, *pname*: GLenum, *param*: GLfloat)

glLight[i f]v (*light*: GLenum, *pname*: GLenum, *params*: ^GLfloat)

Параметр light однозначно визначає джерело світла. Він вибирається з набору спеціальних символічних імен виду GL_LIGHTi, де i повинно лежати в діапазоні від 0 до константи GL_MAX_LIGHT, яка звичайно не перевищує восьми.

Параметри pname і params мають сенс, аналогічний команді glMaterial. Розглянемо значення параметра pname:

GL_SPOT_EXPONENT	параметр param повинний містити ціле чи дійсне число від 0 (розсіяне світло за замовчуванням) до 128, що задає розподіл інтенсивності світла. Цей параметр описує рівень сфокусованості джерела світла
GL_SPOT_CUTOFF	параметр param повинний містити ціле чи дійсне число між 0 і 90 чи дорівнювати 180 (розсіяне світло за замовчуванням), яке визначає максимальний кут розсіювання світла. Значенням цього параметра є половина кута у вершині конусоподібного світлового потоку, створюваного джерелом
GL_AMBIENT	параметр params повинний містити чотири цілих чи дійсних значення кольорів RGBA (за замовчуванням (0, 0, 0, 1)), які визначають колір фонового освітлення
GL_DIFFUSE	параметр params повинний містити чотири цілих чи дійсних значення кольорів RGBA (за замовчуванням (1, 1,

	1, 1) для GL_LIGHT0 і (0, 0, 0, 1) для інших), які визначають колір дифузного освітлення
GL_SPECULAR	параметр params повинний містити чотири цілих чи дійсних значення кольорів RGBA (за замовчуванням (1, 1, 1, 1) для GL_LIGHT0 і (0, 0, 0, 1) для інших), які визначають колір дзеркального відображення
GL_POSITION	параметр params повинний містити чотири цілих чи дійсних числа, що визначають положення джерела світла. Якщо значення компоненти w дорівнює 0.0, то джерело вважається нескінченно віддаленим і при розрахунку освітленості враховується тільки напрямок на точку (x,y,z), у противному випадку вважається, що джерело розташоване в точці (x,y,z,w). У першому випадку ослаблення світла при віддаленні від джерела не відбувається, тобто джерело вважається нескінченно віддаленим. Значення за замовчуванням: (0, 0, 1, 0)
GL_SPOT_DIRECTION	параметр params повинний містити чотири цілих чи дійсних числа, які визначають напрямок світла. Значення за замовчуванням: (0, 0, -1, 1). Ця характеристика джерела має сенс, якщо значення GL_SPOT_CUTOFF відмінне від 180 (яке, до речі, задано за замовчуванням)
GL_CONSTANT_ATTENUATION GL_LINEAR_ATTENUATION GL_QUADRATIC_ATTENUATION	параметр params задає значення одного з трьох коефіцієнтів, що означають ослаблення інтенсивності світла при віддаленні від джерела. Допускаються тільки ненегативні значення. Якщо джерело не є спрямованим (див. GL_POSITION), то

	ослаблення обернено пропорційно сумі: $\mathrm{att}_{\mathrm{constant}} + \mathrm{att}_{\mathrm{linear}} \cdot \mathrm{d} + \mathrm{att}_{\mathrm{quadratic}} \cdot \mathrm{d}^2$ де d – відстань між джерелом світла і освітлюваною ним вершиною, $\mathrm{att}_{\mathrm{constant}}$, att_{linear} і $\mathrm{att}_{\mathrm{quadratic}}$ дорівнюють параметрам, заданим за допомогою констант GL_CONSTANT_ATTENUATION, GL_LINEAR_ATTENUATION і GL_QUADRATIC_ATTENUATION відповідно. За замовчуванням ці параметри задаються трійкою (1, 0, 0), і фактично ослаблення не відбувається

При зміні положення джерела світла варто враховувати наступний факт: у OpenGL джерела світла є об'єктами, багато в чому такими ж, як багатокутники і точки. На них поширюється основне правило обробки координат у OpenGL – параметри, які описують положення у просторі, перетворюються поточною модельно-видовою матрицею в момент формування об'єкта, тобто в момент виклику відповідних команд OpenGL. Таким чином, формуючи джерело світла одночасно з об'єктом сцени чи камерою, його можна прив'язати до цього об'єкта. Чи, навпаки, сформувати стаціонарне джерело світла, що буде залишатися на місці, поки інші об'єкти пересуваються.

Загальні правила наступні:

1. Якщо положення джерела світла задається командою glLight перед визначенням положення віртуальної камери (наприклад, командою glLookAt), то буде вважатися, що координати (0,0,0) джерела знаходяться в точці спостереження і, отже, положення джерела світла визначається щодо положення спостерігача.
2. Якщо положення встановлюється між визначенням положення камери і перетвореннями модельно-видової матриці об'єкта, то воно фіксується, тобто в цьому випадку положення джерела світла задається у світових координатах.

Для використання освітлення спочатку треба встановити відповідний режим викликом команди glEnable(GL_LIGHTING), а потім увімкнути потрібне джерело командою glEnable(GL_LIGHTi).

Ще раз звернемо увагу на те, що при вимкненому освітленні колір вершини дорівнює поточному кольору, що задається командами `glColor`. При увімкненому освітленні колір вершини обчислюється виходячи з інформації про матеріал, нормалі та джерела світла.

При вимиканні освітлення візуалізація відбувається швидше, однак у такому випадку застосунок повинен сам розраховувати кольори вершин.

Текст програми, що демонструє основні принципи визначення матеріалів і джерел світла, наведений в *додатку 1*.

21.4. Створення ефекту туману

На завершення розділу розглянемо одну цікаву і часто використовувану можливість OpenGL – створення ефекту туману. Легке затуманення сцени створює реалістичний ефект, а часто може сховати деякі артефакти, які з'являються, коли в сцені присутні віддалені об'єкти.

Туман у OpenGL реалізується шляхом зміни кольору об'єктів у сцені в залежності від їхньої глибини, тобто відстані до точки спостереження. Зміна кольору відбувається або для вершин примітивів, або для кожного пікселя на етапі растеризації в залежності від реалізації OpenGL.

Для включення ефекту затуманення необхідно викликати команду `glEnable(GL_FOG)`.

Метод обчислення інтенсивності туману у вершині можна визначити за допомогою команд

```
glFog[if] (pname: enum, param: T)
glFog[if]v (pname: enum, params: T)
```

Аргумент `pname` може приймати наступні значення:

<table>
<tr><td rowspan="5">GL_FOG_MODE</td><td colspan="2">аргумент <code>param</code> визначає формулу, по якій буде обчислюватися інтенсивність туману в точці. Може приймати значення:</td></tr>
<tr><td>GL_EXP</td><td>інтенсивність обчислюється по формулі: $f = e^{-d \cdot z}$</td></tr>
<tr><td>GL_EXP2</td><td>інтенсивність обчислюється по формулі: $f = e^{-(d \cdot z)^2}$</td></tr>
<tr><td>GL_LINEAR</td><td>інтенсивність обчислюється по формулі: $f = \frac{e-z}{e-s}$</td></tr>
<tr><td colspan="2">де z – відстань від вершини, у якій обчислюється інтенсивність туману, до точки спостереження</td></tr>
</table>

GL_FOG_DENSITY	аргумент param визначає коефіцієнт d
GL_FOG_START	аргумент param визначає коефіцієнт s
GL_FOG_END	аргумент param визначає коефіцієнт e
GL_FOG_COLOR	у цьому випадку params – вказівник на масив з 4-х компонент кольору туману

Запитання для самоконтролю

1. Поясніть різницю між локальними і нескінченно віддаленими джерелами світла.
2. Для чого служить команда glColorMaterial?
3. Які основні параметри джерела світла?
4. Які існують моделі освітлення?
5. Яким чином в OpenGL створюється ефект туману?

22. НАКЛАДАННЯ ТЕКСТУРИ

Під *текстурою* будемо розуміти деяке растрове зображення, яке треба певним чином накласти на об'єкт, наприклад, для додання ілюзії рельєфності поверхні.

Накладення текстури на поверхню об'єктів сцени підвищує її реалістичність, однак при цьому треба враховувати, що цей процес вимагає значних обчислювальних витрат, особливо якщо OpenGL не підтримується апаратно.

Для роботи з текстурою потрібно виконати наступну послідовність дій:

- вибрати зображення і перетворити його до потрібного формату;
- передати зображення в OpenGL;
- визначити, як текстура буде наноситися на об'єкт і як вона буде з ним взаємодіяти;
- зв'язати текстуру з об'єктом.

22.1. ПІДГОТОВКА ТЕКСТУРИ

Для використання текстури необхідно спочатку завантажити в пам'ять потрібне зображення і передати його OpenGL.

Зчитування графічних даних з файлу та їх перетворення можна проводити вручну. Можна також скористатися функцією, що входить до складу бібліотеки GLAUX (для її використання треба додатково підключити *glaux.lib*), яка сама проводить необхідні операції. Це функція

```
AUX_RGBImageRec* auxDIBImageLoad (const char *file)
```

де `file` – назва файлу з розширенням *.bmp чи *.dib. Функція повертає покажчик на область пам'яті, де зберігаються перетворені дані. Як зрозуміло із синтаксису, ця функція використовується в мові C++. При програмуванні в Delphi можна використати універсальну процедуру, реалізація якої приводиться нижче.

```
procedure PrepareImage(BmpFileName: AnsiString);
type
  PPixelArray = ^TPixelArray;
  TPixelArray = array [0..0] of Byte;
var
  Bitmap : TBitmap;
  Data : PPixelArray;
  BMInfo : TBitmapInfo;
```

```
    I, ImageSize : Integer;
    Temp : Byte;
    MemDC : HDC;
  begin
    Bitmap := TBitmap.Create;
    Bitmap.LoadFromFile(BmpFileName);
    with BMinfo.bmiHeader do begin
      FillChar (BMInfo, SizeOf(BMInfo), 0);
      biSize := sizeof (TBitmapInfoHeader);
      biBitCount := 24;
      biWidth := Bitmap.Width;
      biHeight := Bitmap.Height;
      ImageSize := biWidth * biHeight;
      biPlanes := 1;
      biCompression := BI_RGB;
      MemDC := CreateCompatibleDC (0);
      GetMem (Data, ImageSize * 3);
      try
        GetDIBits (MemDC, Bitmap.Handle, 0, biHeight, Data,
                   BMInfo, DIB_RGB_COLORS);
        For I := 0 to ImageSize - 1 do begin
            Temp := Data [I * 3];
            Data [I * 3] := Data [I * 3 + 2];
            Data [I * 3 + 2] := Temp;
        end;
        glTexImage2d(GL_TEXTURE_2D, 0, 3, biWidth,
              biHeight, 0, GL_RGB, GL_UNSIGNED_BYTE, Data);
      finally
        FreeMem (Data);
        DeleteDC (MemDC);
        Bitmap.Free;
      end;
    end;
  end;
```

При створенні образа текстури в пам'яті варто враховувати наступні вимоги.

По-перше, розміри текстури, як по горизонталі, так і по вертикалі повинні бути ступенями двійки. Ця вимога накладається для компактного розміщення текстури в текстурній пам'яті і сприяє її ефективному використанню. Працювати тільки з такими текстурними файлами звичайно незручно, тому після завантаження їх треба перетворити. Зміну розмірів текстури можна провести за допомогою команди

gluScaleImage (*format*: GLenum, *widthin*: GLint, *heightin*: GLint, *typein*: GLenum, *datain*: pointer, *widthout*: GLint, *heightout*: GLint, *typeout*: GLenum, *dataout*: pointer)

У якості значення параметра format звичайно використовується значення GL_RGB чи GL_RGBA, яке визначає формат збереження інформації. Параметри widthin, heightin, widhtout, heightout визначають розміри вхідного і вихідного зображень, а за допомогою typein і typeout задається тип елементів масивів, розташованих за адресами datain і dataout. Зазвичай, це може бути тип GL_UNSIGNED_BYTE, GL_SHORT, GL_INT і т.д. Результат своєї роботи функція заносить в область пам'яті, на яку вказує параметр dataout.

По-друге, треба передбачити випадок, коли об'єкт після растеризації виявляється за розмірами значно меншим нанесеної на нього текстури. Чим менший об'єкт, тим меншою повинна бути текстура, що наноситься на нього, і тому вводиться поняття *рівнів деталізації текстури* (*mipmapping*). Кожен рівень деталізації задає деяке зображення, що є, як правило, зменшеною в два рази копією оригіналу. Такий підхід дозволяє поліпшити якість нанесення текстури на об'єкт. Наприклад, для зображення розміром $2^m \times 2^n$ можна побудувати max(m,n)+1 зменшених зображень, що відповідають різним рівням деталізації.

Ці два етапи створення образу текстури у внутрішній пам'яті OpenGL можна зробити за допомогою команди

gluBuild2DMipmaps (*target*: GLenum, *components*: GLint, *width*: GLint, *height*: GLint, *format*: GLenum, *type*: GLenum, *data*: pointer)

де параметр target повинен дорівнювати GL_TEXTURE_2D. Параметр components визначає кількість колірних компонентів текстури і може приймати наступні основні значення:

GL_LUMINANCE	один компонент – яскравість (текстура буде монохромною)
GL_RGB	червоний, синій, зелений
GL_RGBA	усі компоненти

Параметри `width`, `height`, `data` визначають розміри і розташування текстури відповідно, а `format` і `type` мають аналогічне значення, що й у команди `gluScaleImage`.

Після виконання цієї команди текстура копіюється у внутрішню пам'ять OpenGL, і тому пам'ять, що зайнята початковим зображенням, можна звільнити.

У OpenGL допускається використання одномірних текстур, тобто розміру 1×N, однак, це завжди треба вказувати, задаючи як значення `target` константу GL_TEXTURE_1D. Корисність одномірних текстур сумнівна, тому не будемо зупинятися на них докладно.

При використанні в сцені декількох текстур, у OpenGL застосовується підхід, що нагадує створення списків зображень (так звані *текстурні об'єкти*). Спочатку за допомогою команди

```
glGenTextures (n: GLsizei, textures: ^GLuint)
```

треба створити n ідентифікаторів текстур, які будуть записані в масив `textures`. Перед початком визначення властивостей чергової текстури варто зробити її поточною («прив'язати» текстуру), викликавши команду

```
glBindTexture (target: GLenum, texture: GLuint)
```

де `target` може приймати значення GL_TEXTURE_1D або GL_TEXTURE_2D, а параметр `texture` повинен дорівнювати ідентифікатору тієї текстури, до якої будуть відноситися наступні команди. Для того, щоб у процесі малювання зробити поточною текстуру з деяким ідентифікатором, досить знову викликати команду `glBindTexture` з відповідним значенням `target` і `texture`. Таким чином, команда `glBindTexture` вмикає режим створення текстури з ідентифікатором `texture`, якщо така текстура ще не створена, або режим її використання, тобто робить цю текстуру поточною.

Оскільки не кожна апаратура може оперувати текстурами великого розміру, доцільно обмежити розміри текстури до 256×256 чи 512×512 пікселів. Зазначимо, що використання невеликих текстур підвищує ефективність програми.

22.2. Накладання текстури на об'єкти

При накладанні текстури, як вже згадувалося, треба враховувати випадок, коли розміри текстури відрізняються від віконних розмірів об'єкта, на який вона накладається. При цьому можливе як розтягування, так і стиснення зображення, і те, як будуть проводитися ці перетворення, може серйозно вплинути на якість побудованого

зображення. Для визначення положення точки на текстурі використовується параметрична система координат (s,t), причому значення s і t знаходяться у відрізку [0,1] (рис. 22.1).

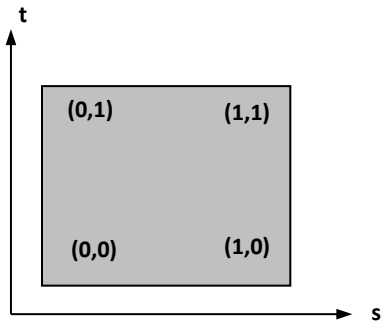


Рис. 22.1. Текстурні координати

Для зміни різних параметрів текстури застосовуються команди:

glTexParameter[i f] (*target*: GLenum, *pname*: GLenum, *param*: GLenum)

glTexParameter[i f]v (*target*: GLenum, *pname*: GLenum, *params*: ^ GLenum)

Параметр target може приймати значення GL_TEXTURE_1D або GL_TEXTURE_2D, pname визначає, яка властивість буде змінюватися, а за допомогою param чи params встановлюється нове значення. Можливі значення pname:

GL_TEXTURE_MIN_FILTER	параметр param визначає функцію, яка буде використовуватися для стиснення текстури. При значенні GL_NEAREST буде використовуватися один (найближчий), а при значенні GL_LINEAR (значення за замовчуванням) чотири найближчих елементи текстури
GL_TEXTURE_MAG_FILTER	параметр param визначає функцію, яка буде використовуватися для збільшення (розтягання) текстури. При значенні GL_NEAREST буде використовуватися один (найближчий), а при значенні GL_LINEAR (значення за замовчуванням) чотири найближчих елементи текстури
GL_TEXTURE_WRAP_S	параметр param встановлює значення координати s, якщо воно не входить у

	відрізок [0,1]. При значенні GL_REPEAT (значення за замовчуванням) ціла частина s відкидається, і в результаті зображення розмножується по поверхні. При значенні GL_CLAMP використовуються крайові значення: 0 чи 1, що зручно використовувати, якщо на об'єкт накладається одне зображення
GL_TEXTURE_WRAP_T	аналогічно попередньому значенню, тільки для координати t

Використання режиму GL_NEAREST підвищує швидкість накладення текстури, однак при цьому знижується якість, тому що на відміну від GL_LINEAR інтерполяція не виконується.

Для того щоб визначити, як текстура буде взаємодіяти з матеріалом, з якого зроблений об'єкт, використовуються команди

glTexEnv[i f] (target: GLenum, pname: GLenum, param: GLtype)

glTexEnv[i f]v (target: GLenum, pname: GLenum, params: ^GLtype)

Параметр target повинен дорівнювати GL_TEXTURE_ENV, а для pname розглянемо тільки одне значення GL_TEXTURE_ENV_MODE, яке найчастіше застосовується.

Найчастіше використовуються такі значення параметра param:

GL_MODULATE	кінцевий колір знаходиться як добуток кольору точки на поверхні і кольору відповідної їй точки на текстурі
GL_REPLACE	як кінцевий колір використовується колір точки на текстурі

22.3. Текстурні координати

Перед нанесенням текстури на об'єкт необхідно встановити відповідність між точками на поверхні об'єкту і на самій текстурі. Задавати цю відповідність можна двома методами: окремо для кожної вершини чи для усіх вершин, задавши параметри спеціальної функції відображення.

Перший метод реалізується за допомогою команд

glTexCoord[1 2 3 4][s i f d] (*coord*: GLtype)

```
glTexCoord[1 2 3 4][s i f d]v (coords: ^GLtype)
```

Найчастіше використовуються команди виду glTexCoord2, які задають поточні координати текстури. Поняття поточних координат текстури аналогічно поняттям поточного кольору і поточної нормалі, і є атрибутом вершини. Однак навіть для куба знаходження відповідних координат текстури є досить трудомістким заняттям, тому в бібліотеці GLU крім команд, що проводять побудову таких примітивів, як сфера, циліндр і диск, передбачене також накладення на них текстур. Для цього досить викликати команду

```
gluQuadricTexture (quadObject: ^GLUquadricObj, textureCoords: GLboolean)
```

з параметром textureCoords, який дорівнює GL_TRUE, і тоді поточна текстура буде автоматично накладатися на примітив.

Другий метод реалізується за допомогою команд

```
glTexGen[i f d] (coord: GLenum, pname: GLenum, param: GLtype)
glTexGen[i f d]v (coord: GLenum, pname: GLenum, params: ^GLtype)
```

Параметр coord визначає, для якої координати задається формула, і може приймати значення GL_S, GL_T, а параметр pname може дорівнювати одному з наступних значень:

GL_TEXTURE_GEN_MODE	визначає функцію для накладення текстури. У цьому випадку аргумент param приймає значення:	
	GL_OBJECT_LINEAR	значення відповідної текстурної координати визначається відстанню до площини, що задається за допомогою значення pname GL_OBJECT_PLANE (див. нижче). Формула має наступний вигляд: $g = x \cdot x_p + y \cdot y_p + z \cdot z_p + w \cdot w_p$ де g – відповідна текстурна координата (s чи t), x, y, z, w – координати відповідної точки, x_p, y_p, z_p, w_p – коефіцієнти рівняння площини. У формулі використовуються координати об'єкта

	GL_EYE_ LINEAR	аналогічно попередньому значенню, тільки у формулі використовуються видові координати. Тобто координати текстури об'єкта в цьому випадку залежать від положення цього об'єкта
	GL_SPHERE_ MAP	дозволяє емулювати віддзеркалювання від поверхні об'єкта. Текстура начебто «обертається» навколо об'єкта. Для даного методу використовуються видові координати і необхідне завдання нормалей
GL_OBJECT_ PLANE	дозволяє задати площину, відстань до якої буде використовуватися при генерації координат, якщо встановлений режим GL_OBJECT_LINEAR. У цьому випадку параметр params є вказівником на масив з чотирьох коефіцієнтів рівняння площини	
GL_EYE_PLANE	аналогічно попередньому значенню. Дозволяє задати площину для режиму GL_EYE_LINEAR	

Для встановлення автоматичного режиму завдання текстурних координат необхідно викликати команду glEnable з параметром GL_TEXTURE_GEN_S або GL_TEXTURE_GEN_T.

Для прикладу розглянемо, як можна задати дзеркальну текстуру. При такому накладенні текстури зображення буде начебто відбиватися від поверхні об'єкта, викликаючи цікавий оптичний ефект. Для цього спочатку треба створити два цілочисельних масиви коефіцієнтів s_coeffs і t_coeffs зі значеннями (1,0,0,1) і (0,1,0,1) відповідно, а потім викликати команди:

```
glEnable (GL_TEXTURE_GEN_S);
glTexGeni (GL_S, GL_TEXTURE_GEN_MODE, GL_EYE_LINEAR);
glTexGendv (GL_S, GL_EYE_PLANE, s_coeffs);
```

і такі ж команди для координати t з відповідними змінами.

Програма, що використовує накладення текстури, наведена в *додатку 2*.

Запитання для самоконтролю

1. Для чого використовуються рівні деталізації текстури (mipmapping)?
2. Як встановити відповідність між координатами об'єкта та текстури?
3. Які обмеження накладаються на файли текстури?
4. Що таке текстурні координати?

23. Операції з пікселями

Після проведення всіх операцій по перетворенню координат вершин, обчислення кольору і т.п., OpenGL переходить до етапу *растеризації***Помилка! Закладку не визначено.**, на якому відбувається растеризація всіх примітивів, накладення текстури і створення ефекту туману**Помилка! Закладку не визначено.**. Для кожного примітиву результатом цього процесу є займана ним у буфері кадру область, кожному пікселю цієї області приписується колір і значення глибини. OpenGL використовує цю інформацію, щоб записати оновлені дані в буфер кадру. Для цього OpenGL має не тільки окремий конвеєр обробки пікселів, але і кілька додаткових буферів різного призначення. Це дозволяє програмісту гнучко контролювати процес візуалізації на найнижчому рівні.

Як вже зазначалося в розділі 19, графічна бібліотека OpenGL підтримує роботу з наступними буферами:

- кілька буферів кольору**Помилка! Закладку не визначено.**;
- буфер глибини;
- буфер-накопичувач (акумулятор);
- буфер маски (трафарету).

Група буферів кольору включає буфер кадру, але таких буферів може бути декілька. При використанні подвійної буферизації говорять про робочий (*front*) і фоновий (*back*) буфери. Як правило, у фоновому буфері програма створює зображення, що потім разом копіюється в робочий буфер. На екрані може з'явитися інформація тільки буферів кольору.

Буфер глибини призначений для видалення невидимих поверхонь, і пряма робота з ним потрібна вкрай рідко. Буфер-накопичувач можна застосовувати для різних операцій, більш докладно робота з ним описана в розділі 23.2. Буфер маски використовується для формування піксельних масок (трафаретів), які служать для вирізання із загального масиву тих пікселів, які варто вивести на екран.

23.1. Прозорість

Різноманітні прозорі об'єкти – скло, прозорий посуд і т.д., часто зустрічаються в реальності, тому важливо вміти створювати такі об'єкти в інтерактивній графіці. OpenGL надає програмісту механізм роботи з напівпрозорими об'єктами, який буде коротко описаний в цьому розділі.

Прозорість реалізується за допомогою спеціального режиму змішування кольорів (*blending*) . Алгоритм змішування комбінує кольори так званих вхідних пікселів (тобто «кандидатів», які будуть вміщуватися в буфер кадру) з кольорами відповідних пікселів, що вже зберігаються в буфері. Для змішування використовується четвертий компонент кольору – альфа-компонент, тому цей режим називають ще альфа-змішуванням. Програма може керувати інтенсивністю альфа-компоненти так само, як і інтенсивністю основних кольорів, тобто задавати значення інтенсивності для кожного пікселя чи кожної вершини примітиву.

Режим включається за допомогою команди `glEnable(GL_BLEND)`.

Визначити параметри змішування можна за допомогою команди:

`glBlendFunc`(*`src`*: GLenum, *`dst`*: GLenum)

Параметр `src` визначає, як одержати коефіцієнт k_1 вихідного кольору пікселя, а `dst` задає спосіб одержання коефіцієнту k_2 для кольору в буфері кадру. Для одержання результуючого кольору використовується наступна формула:

$$\mathrm{res} = \mathrm{c_{src}} \cdot \mathrm{k_1} + \mathrm{c_{dst}} \cdot \mathrm{k_2},$$

де c_{src} – колір вихідного пікселя, c_{dst} – колір пікселя в буфері кадру (`res`, k_1, k_1, c_{src}, c_{dst} – чотирикомпонентні RGBA-вектори).

Приведемо найбільш часто використовувані значення аргументів `src` і `dst`:

GL_SRC_ALPHA	$\mathrm{k} = (\mathrm{A_s}, \mathrm{A_s}, \mathrm{A_s}, \mathrm{A_s})$
GL_SRC_ONE_MINUS_ALPHA	$\mathrm{k} = (1,1,1,1) - (\mathrm{A_s}, \mathrm{A_s}, \mathrm{A_s}, \mathrm{A_s})$
GL_DST_COLOR	$\mathrm{k} = (\mathrm{R_d}, \mathrm{G_b}, \mathrm{B_b})$
GL_ONE_MINUS_DST_COLOR	$\mathrm{k} = (1,1,1,1) - (\mathrm{R_b}, \mathrm{G_b}, \mathrm{B_b}, \mathrm{A_b})$
GL_DST_ALPHA	$\mathrm{k} = (\mathrm{A_d}, \mathrm{A_d}, \mathrm{A_d}, \mathrm{A_d})$
GL_DST_ONE_MINUS_ALPHA	$\mathrm{k} = (1,1,1,1) - (\mathrm{A_d}, \mathrm{A_d}, \mathrm{A_d}, \mathrm{A_d})$
GL_SRC_COLOR	$\mathrm{k} = (\mathrm{R_s}, \mathrm{G_s}, \mathrm{B_s})$
GL_ONE_MINUS_SRC_COLOR	$\mathrm{k} = (1,1,1,1) - (\mathrm{R_s}, \mathrm{G_s}, \mathrm{B_s}, \mathrm{A_s})$

Наприклад, ми хочемо реалізувати виведення прозорих об'єктів. Коефіцієнт прозорості задається альфа-компонентою кольору. Нехай `1` – непрозорий об'єкт, а `0` – абсолютно прозорий, тобто невидимий. Для реалізації служить наступний код:

```
glEnable(GL_BLEND);
glBlendFunc(GL_SRC_ALPHA,GL_SRC_ONE_MINUS_ALPHA);
```

Наприклад, напівпрозорий трикутник можна задати в такий спосіб:

```
glColor3f(1.0, 0.0, 0.0, 0.5);
glBegin(GL_TRIANGLES);
  glVertex3f(0.0, 0.0, 0.0);
  glVertex3f(1.0, 0.0, 0.0);
  glVertex3f(1.0, 1.0, 0.0);
glEnd;
```

Якщо в сцені є кілька прозорих об'єктів, що можуть перекривати один одного, коректне виведення можна гарантувати тільки у випадку виконання наступних умов:

- усі прозорі об'єкти виводяться після непрозорих;
- при виведенні об'єкти з прозорістю повинні бути упорядковані по зменшенню глибини, тобто виводитися, починаючи з найбільш віддалених від спостерігача.

В OpenGL команди обробляються в порядку їхнього надходження, тому для реалізації перерахованих вимог досить розставити у відповідному порядку виклики команд glVertex, але і це в загальному випадку нетривіально.

23.2. Буфер-накопичувач

Буфер-накопичувач (*accumulation buffer*) – це один з додаткових буферів OpenGL. У ньому можна зберігати зображення для візуалізації, застосовуючи спеціальні операції до кожного пікселя. Буфер-накопичувач широко використовується для створення різних спецефектів.

Зображення береться з буфера, вибраного для зчитування командою

```
glReadBuffer(buf: GLenum)
```

Аргумент buf визначає буфер для зчитування. Значення buf, які дорівнюють GL_BACK, GL_FRONT, визначають відповідні буфери кольору для читання. GL_BACK задає як джерело пікселів фоновий буфер; GL_FRONT – поточний вміст вікна виводу. Команда має значення, якщо використовується подвійна буферизація. В іншому випадку використовується тільки один буфер, що відповідає вікну виведення.

Буфер-накопичувач є додатковим буфером кольору. Він не використовується для виведення зображень, але вони додаються в нього після виведення в один з буферів кольору. Застосовуючи описані нижче операції, можна потроху «накопичувати» зображення в буфері.

Потім отримане зображення переноситься з буфера-накопичувача в один з буферів кольору, обраний для запису командою

```
glDrawBuffer(buf: GLenum)
```

Значення buf аналогічне значенню відповідного аргументу в команді glReadBuffer.

Всі операції з буфером-накопичувачем контролюються командою

```
glAccum(op: GLenum, value: GLfloat)
```

Аргумент op задає операцію над пікселями і може приймати наступні значення:

GL_LOAD	піксель береться з буфера, обраного для читання, його значення збільшується на value і заноситься в буфер-накопичувач
GL_ACCUM	аналогічно попередньому, але отримане після множення значення складається з уже наявним у буфері
GL_MULT	ця операція множить значення кожного пікселя в буфері-накопичувачі на value
GL_ADD	аналогічно попередньому, тільки замість множення використовується додавання
GL_RETURN	зображення переноситься з буфера-накопичувача в буфер, обраний для запису. Перед цим значення кожного пікселя множиться на value

Слід зазначити, що для використання буфера-накопичувача немає необхідності викликати будь-яку команду glEnable. Досить ініціалізувати тільки сам буфер.

23.3. Буфер маски

При виводі пікселів у буфер кадру виникає необхідність виводити не всі пікселі, а тільки деяку їх підмножину, тобто накласти *трафарет* (*маску*) на зображення. Для цього OpenGL надає так званий буфер маски (*stencil buffer*). Крім накладення маски, цей буфер надає ще кілька цікавих можливостей.

Перш ніж помістити піксел в буфер кадру, механізм візуалізації OpenGL дозволяє виконати порівняння (тест) між заданим значенням в буфері маски. Якщо тест проходить, піксел малюється в буфері кадру.

Механізм порівняння дуже гнучкий і контролюється наступними командами:

glStencilFunc (*func*: GLenum, *ref*: GLint, *mask*: GLuint)

glStencilOp (*sfail*: GLenum, *dpfail*: GLenum, *dppass*: GLenum)

Аргумент ref команди glStencilFunc задає значення для порівняння. Він повинний приймати значення від 0 до 2^s-1, де s – число біт на точку в буфері маски.

За допомогою аргументу func задається функція порівняння. Він може приймати наступні значення:

GL_NEVER	тест ніколи не проходить, тобто завжди повертає false
GL_ALWAYS	тест проходить завжди, тобто завжди повертає true
GL_LESS	тест проходить у випадку, якщо ref менше значення в буфері маски
GL_LEQUAL	тест проходить у випадку, якщо ref менше або дорівнює значенню в буфері маски
GL_EQUAL	тест проходить у випадку, якщо ref дорівнює значенню в буфері маски
GL_GEQUAL	тест проходить у випадку, якщо ref більше або дорівнює значенню в буфері маски
GL_GREATE	тест проходить у випадку, якщо ref більше значення в буфері маски
GL_NOTEQUAL	тест проходить у випадку, якщо ref не дорівнює значенню в буфері маски

Аргумент *mask* задає маску для значень. Тобто у підсумку для цього тесту одержуємо наступну формулу:

$$[(\text{ref AND mask}) \text{ or } (\text{svalue AND mask})].$$

Команда glStencilOp призначена для визначення дій над *пікселями* буфера маски у випадку позитивного чи негативного результату тесту.

Аргумент sfail задає дію у випадку негативного результату тесту, і може приймати наступні значення:

GL_KEEP	зберігає значення в буфері маски
GL_ZERO	обнуляє значення в буфері маски

GL_REPLACE	значення в буфері маски замінює на значення *ref*
GL_INCR	збільшує значення в буфері маски
GL_DECR	зменшує значення в буфері маски
GL_INVERT	побітно інвертує значення в буфері маски

Аргументи dpfail визначають дії у випадку негативного результату тесту на глибину в z-буфері, а dppass задає дію у випадку позитивного результату цього тесту. Аргументи приймають ті ж значення, що й аргумент sfail. За замовчуванням всі три параметри встановлені на GL_KEEP.

Для включення маскування треба виконати команду glEnable(GL_STENCIL_TEST).

Буфер маски використовується при створенні таких спецефектів, як тінь, віддзеркалення, плавні переходи однієї картинки в іншу тощо.

23.4. Керування растеризацією

Спосіб виконання растеризації примітивів можна частково регулювати командою

```
glHint (target: GLenum, mode: GLenum)
```

де target – вид контрольованих дій, приймає одне з наступних значень:

GL_FOG_HINT	точність обчислень при накладанні туману. Обчислення можуть виконуватися для кожного пікселя (найбільша точність) чи тільки у вершинах. Якщо реалізація OpenGL не підтримує попіксельного обчислення, то виконується тільки обчислення по вершинах
GL_LINE_SMOOTH_HINT	керування якістю прямих. При значенні mode, яке дорівнює GL_NICEST, зменшується ступінчастість прямих за рахунок більшого числа пікселів в прямих
GL_PERSPECTIVE_ CORRECTION_HINT	точність інтерполяції координат при обчисленні кольорів і накладення текстури. Якщо реалізація OpenGL не підтримує режим GL_NICEST, то здійснюється лінійна інтерполяція

	координат
GL_POINT_ SMOOTH_HINT	керування якістю точок. При значенні параметра mode, який дорівнює GL_NICEST точки малюються як кола
GL_POLYGON_ SMOOTH_HINT	керування якістю виводу сторін багатокутника

Параметр mode інтерпретується в такий спосіб:

GL_FASTEST	використовується найбільш швидкий алгоритм
GL_NICEST	використовується алгоритм, що забезпечує кращу якість
GL_DONT_CARE	вибір алгоритму залежить від реалізації

Слід зауважити, що командою glHint програміст може тільки визначити свої побажання щодо того чи іншого аспекту растеризації примітивів. В конкретній реалізації OpenGL ці побажання можуть бути і проігноровані.

Зверніть увагу на те, що glHint не можна викликати між операторними дужками glBegin/glEnd.

Запитання для самоконтролю

1. Скільки буферів існує в OpenGL? Які їх назви і призначення?
2. Як створюється ефект прозорості в OpenGL?
3. Який буфер використовується для спеціальних ефектів?
4. Сформулюйте принципи використання маски зображення.

24. Прийоми роботи з OpenGL

У цьому розділі ми розглянемо, як за допомогою OpenGL створювати деякі цікаві візуальні ефекти, безпосередня підтримка яких відсутня у стандарті бібліотеки.

24.1. Усунення ступінчастості

Почнемо з задачі усунення ступінчастості (*antialiasing*). Ефект ступінчастості (*aliasing*) виникає в результаті похибок растеризації примітивів у буфері кадру через скінченну (і, як правило, невелику) роздільну здатність. Є кілька підходів до вирішення даної проблеми. Наприклад, можна застосовувати фільтрацію отриманого зображення. Також цей ефект можна усувати на етапі растеризації, згладжуючи образ кожного примітива. Тут ми розглянемо прийом, що дозволяє усувати подібні артефакти для всієї сцени в цілому.

Для кожного кадру необхідно намалювати сцену кілька разів, на кожному проході трохи зсуваючи камеру відносно початкового положення. Положення камер, наприклад, можуть утворювати коло. Якщо зсув камери відносно малий, то похибки дискретизації проявляться по-різному, і, за рахунок їх усереднення, ми одержимо згладжене зображення.

Найпростіше зсувати положення спостерігача, але до цього потрібно обчислити величину зсуву так, щоб приведене до координат екрана значення не перевищувало, скажімо, половини розміру пікселя.

Всі отримані зображення зберігаємо в буфері-накопичувачі з коефіцієнтом 1/n, де n – число проходів для кожного кадру. Чим більше таких проходів тим нижче продуктивність, але краще результат.

```
for i:=0 to samples_count do
begin
// зазвичай samples_count лежить у межах від 5 до 10
  ShiftCamera(i); // зміщуємо камеру
  RenderScene();
  if i = 0 then
   // на першій ітерації завантажуємо зображення
   glAccum(GL_LOAD,1/samples_count);
  else
   // додаємо до вже існуючого
   glAccum(GL_ACCUM,1/samples_count);
```

```
end;
// Записуємо те, що вийшло, назад у вихідний буфер
glAccum(GL_RETURN,1.0);
```

Слід зазначити, що усунення ступінчастості відразу для всієї сцени, як правило, зв'язане із серйозним падінням продуктивності візуалізації, тому що вся сцена малюється кілька разів. Сучасні прискорювачі зазвичай апаратно реалізують інші методи, засновані на так називаному ресемплінзі зображень.

24.2. Побудова тіней

В OpenGL немає вбудованої підтримки побудови тіней на рівні базових команд. У значній мірі це пояснюється тим, що існує безліч алгоритмів їхньої побудови, які можуть бути реалізовані через функції OpenGL. Присутність тіней суттєво збільшує реалістичність тривимірного зображення, тому розглянемо один з прийомів їх побудови.

Більшість алгоритмів побудови тіней використовують модифіковані принципи перспективної проекції. Ми розглянемо один з найпростіших методів. З його допомогою можна одержувати тіні, що відкидаються тривимірним об'єктом на площину.

Загальний підхід такий: для всіх точок об'єкта знаходиться їхня проекція паралельно вектору, що з'єднує дану точку і точку, в якій знаходиться джерело світла, на деяку задану площину. Таким чином одержуємо новий об'єкт, що цілком належить заданій площині. Цей об'єкт і є тінню заданого.

Розглянемо математичні основи даного методу. Нехай:

P – точка в тривимірному просторі, що відкидає тінь.

L – положення джерела світла, що освітлює дану точку.

S=a(L-P)-P – точка, у яку відкидає тінь точка P, де a – параметр.

Припустимо, що тінь падає на площину z=0. У цьому випадку $a = \frac{z_p}{(z_1 - z_p)}$.

Таким чином:

$$x_s = \frac{(x_p \cdot z_1 - x_1 \cdot z_p)}{(z_1 - z_p)};\ y_s = \frac{(y_p \cdot z_1 - y_1 \cdot z_p)}{(z_1 - z_p)};\ z_s = 0.$$

Введемо однорідні координати:

$$x_s = \frac{x_s'}{w_s'};\ y_s = \frac{y_s'}{w_s'};\ z_s = 0;\ z_s' = z_1 - z_p.$$

Тоді координати S можуть бути отримані з використанням множення матриць у такий спосіб:

$$[x'_s\ y'_s\ 0\ w'_s] = [x_s\ y_s\ z_s\ 1]\begin{bmatrix} z_1 & 0 & 0 & 0 \\ 0 & z_1 & 0 & 0 \\ -x_1 & -y_1 & 0 & -1 \\ 0 & 0 & 0 & z_1 \end{bmatrix}.$$

Для того, щоб алгоритм міг розраховувати тінь, що падає на довільну площину, розглянемо довільну точку на лінії між S і P, подану в однорідних координатах:

$$a \cdot P + b \cdot L,$$

де a і b – скалярні параметри.

Наступна матриця задає площину через координати її нормалі:

$$G = \begin{bmatrix} x_n \\ y_n \\ z_n \\ d \end{bmatrix}$$

Точка, у якій промінь, проведений від джерела світла через дану точку P, перетинає площину G, визначається параметрами a і b, які задовольняють наступне рівняння:

$$(a \cdot P + b \cdot L) \cdot G = 0.$$

Звідси одержуємо:

$$a \cdot (P \cdot G) + b \cdot (L \cdot G) = 0.$$

Цьому рівнянню задовольняють:

$$a = L \cdot G;\ b = -(P \cdot G).$$

Отже, координати шуканої точки:

$$S = (L \cdot G) \cdot P - (P \cdot G) \cdot L.$$

Користуючись асоціативністю матричного добутку, одержимо:

$$S = P \cdot [(L \cdot G) \cdot I - G \cdot L].$$

де I – одинична матриця.

Матриця (LG)I-GL використовується для побудови тіні на довільній площині.

Розглянемо деякі аспекти практичної реалізації даного методу з використанням OpenGL.

Припустимо, що матриця *floorShadow* була раніше отримана нами з формули (LG)I-GL. Наступний код використовує її для побудови тіні на заданій площині:

```
// Робимо тіні напівпрозорими з використанням
// змішування кольорів (blending)
glEnable(GL_BLEND);
glBlendFunc(GL_SRC_ALPHA,GL_ONE_MINUS_SRC_ALPHA);
glDisable(GL_LIGHTING);
glColor4f(0.0, 0.0, 0.0, 0.5);
glPushMatrix();
   // Проектуємо тінь
   glMultMatrixf(@floorShadow);
   // Формуємо зображення сцени в проекції
   RenderGeometry();
glPopMatrix();
glEnable(GL_LIGHTING);
glDisable(GL_BLEND);
// Формуємо зображення сцени в звичайному режимі
RenderGeometry();
```

Матриця *floorShadow* може бути отримана із розглянутих рівнянь за допомогою наступної процедури:

```
procedure ShadowMatrix(plane, lightpos: array [0..3] of
GLfloat; var matrix: array [0..3, 0..3] of GLfloat)
{ параметри:    plane – коефіцієнти рівняння площини
                lightpos – координати джерела світла
          повертаємо: matrix – результуюча матриця }
var
  dot: GLFloat;
begin
  dot := plane[0]*lightpos[0] + plane[1]*lightpos[1] +
         plane[2]*lightpos[2] + plane[3]*lightpos[3];
  matrix[0,0] := dot - lightpos[0]*plane[0];
  matrix[1,0] := 0.0 - lightpos[0]*plane[1];
  matrix[2,0] := 0.0 - lightpos[0]*plane[2];
  matrix[3,0] := 0.0 - lightpos[0]*plane[3];
  matrix[0,1] := 0.0 - lightpos[1]*plane[0];
```

```
    matrix[1,1] := dot - lightpos[1]*plane[1];
    matrix[2,1] := 0.0 - lightpos[1]*plane[2];
    matrix[3,1] := 0.0 - lightpos[1]*plane[3];
    matrix[0,2] := 0.0 - lightpos[2]*plane[0];
    matrix[1,2] := 0.0 - lightpos[2]*plane[1];
    matrix[2,2] := dot - lightpos[2]*plane[2];
    matrix[3,2] := 0.0 - lightpos[2]*plane[3];
    matrix[0,3] := 0.0 - lightpos[3]*plane[0];
    matrix[1,3] := 0.0 - lightpos[3]*plane[1];
    matrix[2,3] := 0.0 - lightpos[3]*plane[2];
    matrix[3,3] := dot - lightpos[3]*plane[3];
end;
```

Але тіні, побудовані таким чином, мають ряд недоліків:

- описаний алгоритм припускає, що площини нескінченні, і не відсікає тіні по границі. Наприклад, якщо деякий об'єкт відкидає тінь на стіл, вона не буде відтинатися по границі, і, тим більше, «загортатися» на бічну поверхню столу;
- у деяких місцях тіні може спостерігатися ефект «подвійного змішування» (*reblending*), тобто темні плями в тих ділянках, де спроектовані трикутники перекривають один одного;
- зі збільшенням числа поверхонь складність алгоритму різко збільшується, тому що для кожної поверхні потрібно заново будувати всю сцену, навіть якщо проблема відсікання тіней по границі буде вирішена;
- тіні звичайно мають розмиті границі, а в наведеному алгоритмі вони завжди мають різкі краї. Частково уникнути цього дозволяє розрахунок тіней з декількох джерел світла, розташованих поруч і наступне змішування результатів.

Для усунення перших двох недоліків можна використати буфер маски.

Отже, задача – відсікти виведення геометрії (тіні, у даному випадку) по границі деякої довільної області й уникнути «подвійного змішування». Загальний алгоритм вирішення задачі з використанням буфера маски такий:

1. Очищуємо буфер маски значенням 0.

```
// Очищуємо буфер маски
glClearStencil(0);
```

```
// вмикаємо тест
glEnable(GL_STENCIL_TEST);
```

2. Відображаємо задану область відсікання, встановлюючи значення в буфері маски в 1.

```
// умова завжди виконана і значення в буфері буде
// дорівнювати 1
glStencilFunc (GL_ALWAYS, 0, $FFFFFFF);
// у будь-якому випадку заміняємо значення в буфері маски
glStencilOp (GL_REPLACE, GL_REPLACE, GL_REPLACE);
// виводимо геометрію, по якій потім буде відсічена тінь
RenderPlane();
```

3. Малюємо тіні в тих областях, де в буфері маски встановлені значення 1. Якщо тест проходить, встановлюємо в ці області значення 2.

```
// умова виконана і тест дає істину тільки якщо
// значення в буфері маски дорівнює 1
glStencilFunc (GL_EQUAL, 0, $FFFFFFF);
// значення в буфері дорівнює 2, якщо тінь вже виведена
glStencilOp (GL_KEEP, GL_KEEP, GL_INCR);
// виводимо тіні
RenderShadow();
```

Однак, навіть при застосуванні маскування залишаються невирішеними деякі проблеми пов'язані з роботою *z*-буфера. Зокрема, окремі ділянки тіней можуть стати невидимими. Для вирішення цієї проблеми можна спробувати трохи підняти тіні над площиною за допомогою модифікації рівняння, яке описує площину. Опис інших методів виходить за рамки даного посібника. Приклад програми, що виконує побудову тіні наведено в *додатку 3*.

24.3. Дзеркальні відображення

У цьому розділі розглянемо алгоритм побудови відображень від плоских об'єктів. Такі відображення додають більше реалістичності побудованому зображенню і їх відносно легко отримати.

Алгоритм використовує інтуїтивне представлення повної сцени з дзеркалом як складеної з двох: «дійсної» і «віртуальної» – яка знаходиться за дзеркалом. Отже, процес формування відображень складається з двох частин: візуалізації звичайної сцени та побудови і візуалізації віртуальної.

Для кожного об'єкта «дійсної» сцени будується його відбитий двійник, який спостерігач і побачить у дзеркалі (рис. 24.1).

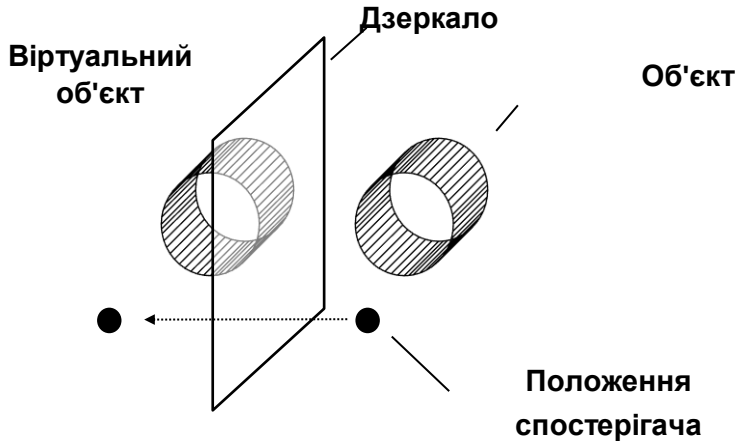


Рис. 24.1. Схема дзеркального відображення

Для ілюстрації розглянемо кімнату з дзеркалом на стіні. Кімната й об'єкти, що знаходяться в ній, виглядають у дзеркалі так, ніби дзеркало було вікном, а за ним була б ще одна така кімната з тими же об'єктами, але симетрично відображеними щодо площини, проведеної через поверхню дзеркала.

Спрощений варіант алгоритму створення плоского відображення складається з наступних кроків:

1. Формуємо сцену як звичайно, але без об'єктів-дзеркал.
2. З використанням буферу маски обмежуємо подальше виведення проекції дзеркала на екран.
3. Формуємо сцену, відображену відносно площини дзеркала. При цьому буфер маски дозволить обмежити виведення формою проекції об'єкта-дзеркала.

Ця послідовність дій дозволить одержати переконливий ефект відображення. Розглянемо етапи більш докладно.

Спочатку необхідно намалювати сцену як звичайно. Не будемо зупинятися на цьому етапі докладно. Зауважимо тільки, що, очищаючи буфери OpenGL безпосередньо перед малюванням, потрібно не забути очистити буфер маски:

```
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT |
GL_STENCIL_BUFFER_BIT);
```

Під час візуалізації сцени краще не малювати об'єкти, що потім стануть дзеркальними.

На другому етапі необхідно обмежити подальше виведення проекції дзеркального об'єкта на екран.

Для цього налагоджуємо буфер маски і малюємо дзеркало

```
glEnable(GL_STENCIL_TEST);
// умова завжди виконана і значення в буфері буде
// дорівнювати 1
glStencilFunc(GL_ALWAYS, 1, 0);
glStencilOp(GL_KEEP, GL_KEEP, GL_REPLACE);
RenderMirrorObject();
```

У результаті ми одержали:

- у буфері кадру – коректно зображена сцена, за винятком області дзеркала;
- в області дзеркала (там, де ми хочемо бачити відображення) значення буфера маски дорівнює 1.

На третьому етапі потрібно намалювати сцену, відображену відносно площини дзеркального об'єкта.

Спочатку налаштовуємо матрицю відображення. Матриця відображення повинна дзеркально відбивати всю геометрію відносно площини, у якій лежить об'єкт-дзеркало. Її можна одержати, наприклад, за допомогою такої функції (спробуйте одержати цю матрицю самостійно як вправу):

```
procedure ReflectionMatrix(plane_point, plane_normal: array [0..2] of GLfloat; var rf_matrix: array [0..3] of GLfloat)
var
  pv: GLfloat;
begin
  p:=plane_point[0]*plane_normal[0]+plane_point[1]*
     plane_normal[1]+plane_point[2]*plane_normal[2];
  rf_matrix[0,0]=1-2*plane_normal[0]* plane_normal[0];
  rf_matrix[1,0]=0-2*plane_normal[0]*plane_normal[1];
  rf_matrix[2,0]=0-2*plane_normal[0]*plane_normal[2];
  rf_matrix[3,0]=2*p*plane_normal[0];
  rf_matrix[0,1]=0-2*plane_normal[0]*plane_normal[1];
  rf_matrix[1,1]=1-2*plane_normal[1]*plane_normal[1];
  rf_matrix[2,1]=0-2*plane_normal[1]*plane_normal[2];
  rf_matrix[3,1]=2*p*plane_normal[1];
```

```
    rf_matrix[0,2]=0-2*plane_normal[0]*plane_normal[2];
    rf_matrix[1,2]=0-2*plane_normal[1]*plane_normal[2];
    rf_matrix[2,2]=1-2*plane_normal[2]*plane_normal[2];
    rf_matrix[3,2]=2*p*plane_normal[2];
    rf_matrix[0,3] = 0;
    rf_matrix[1,3] = 0;
    rf_matrix[2,3] = 0;
    rf_matrix[3,3] = 1;
end;
```

Налаштовуємо буфер маски на малювання тільки в областях, де значення буферу дорівнює 1:

```
// умова виконана і тест дає істину тільки якщо
// значення в буфері маски дорівнює 1
glStencilFunc (GL_EQUAL, 0, $FFFFFFF);
// нічого не змінюємо в буфері
glStencilOp(GL_KEEP, GL_KEEP, GL_KEEP);
```

і малюємо сцену ще раз (без дзеркальних об'єктів)

```
glPushMatrix();
glMultMatrixf(@reflection_matrix);
RenderScene();
GlPopMatrix();
```

Нарешті, відключаємо маскування

```
glDisable(GL_STENCIL_TEST);
```

В разі потреби, після цього можна ще раз вивести дзеркальний об'єкт, наприклад, з альфа-змішуванням – для створення ефекту помутніння дзеркала і т.д.

Існує кілька модифікацій цього алгоритму, які відрізняються послідовністю дій, обмеженнями на геометрію і т.д.

Запитання для самоконтролю

1. В чому полягає ефект ступінчастості? Алгоритм його усунення?
2. Принципи побудови тіней.
3. Які складові враховуються при побудові матриці тіні?
4. Опишіть алгоритм побудови дзеркальних зображень.

25. Оптимізація програм

Для мінімізації часу формування та виведення зображень необхідно притримуватися багатьох класичних рекомендацій по програмуванню на OpenGL: інтенсивно використовувати заздалегідь підготовлені списки команд, векторну форму завдання параметрів функцій, перехоплення повідомлень Windows об'єктам інтерфейсу з користувачем тощо. Загалом, тема оптимізації графічних програм є захоплюючою і практично невичерпною, а тому не може бути з достатньою детальністю розглянута в рамках даного посібника. Наведемо лише окремі поради і прийоми з підвищення надійності і продуктивності графічних застосувань на основі використання бібліотеки OpenGL.

25.1. Поради з підвищення надійності програм

В літературі та Інтернет-джерелах можна знайти багато порад з надійного програмування графічних застосунків. Наведемо лише окремі з них:

- завжди перевіряйте програми на наявність помилок. Для виявлення помилок викликайте команду `glGetError` відразу після рендерінгу сцени;
- враховуйте реакцію бібліотеки на помилки в залежності від версії OpenGL. Наприклад, OpenGl 1.1 ігнорує матричні операції, що викликаються між командами `glBegin` та `glEnd`, але наступні версії можуть реагувати іншим чином. Семантика помилок OpenGL також може змінюватись між синтаксично сумісними версіями;
- якщо необхідно згорнути усі геометричні побудови у окремій площині, то треба використовувати проекційну матрицю. При використанні матриці видового перетворення необхідний кінцевий результат не гарантується;
- не бажано робити багато послідовних змін окремої матриці. Наприклад, не виконуйте анімацію обертанням шляхом неперервного виклику команди `glRotate` із зростаючим кутом повороту. Краще використовувати команду `glLoadIdentity` для ініціалізації даної матриці у кожному кадрі, а потім викликати команду `glRotate` з необхідним кутом для цього кадру;
- не варто очікувати повідомлення про помилки під час створення списку зображень. Команди у межах списку зображень генерують помилки тільки під час використання списку;

- розміщуйте найближчу площину видимого об'єму якомога далі від точки спостереження для того, щоб оптимізувати роботу буферу глибини;
- використовуйте команду `glFlush` для примусового викликання усіх попередніх команд OpenGL;
- використовуйте весь діапазон буферу-накопичувача. Наприклад, при накопичуванні чотирьох зображень масштабуйте кожне зображення `1/4` від об'єму зображень, що накопичуються;
- якщо необхідно точна двохвимірна растеризація, то треба ретельно визначити ортогональну проекцію і вершини примітивів, які необхідно растеризувати. Ортогональна проекція повинна бути визначена з цілочисельними координатами: `gluOrtho2D(0, width, 0, height)`, де `width` та `height` – розміри області перегляду. Для растеризації дані проекційної матриці, вершини багатокутників та позиції піксельних образів повинні бути задані цілочисельними координатами;
- уникайте використання від'ємних значень координати вершини `w` та координати текстури `q`. OpenGL не може проводити вірне відсікання, а також може робити помилки при інтерполяції та тонуванні примітивів, що задані такими координатами;
- не прогнозуйте точність виконання операцій, виходячи з типів даних для параметрів команд OpenGL. Наприклад, якщо Ви використовуєте команду `glRotated`, то необов'язково, що конвеєр, який обробляє геометричні об'єкти, збереже точність, очікувану для числа з плаваючою точкою подвійної точності на протязі всієї операції. Можливо, що параметри команди `glRotated`, будуть перетворені у інші типи даних перед обробкою.

25.2. Прийоми підвищення продуктивності застосунків

Для підвищення продуктивності прикладних програм використовуйте наступні прийоми:

- використовуйте команду `glColorMaterial` тільки тоді, коли властивість матеріалу швидко змінюється (наприклад, у кожній вершині). Використовуйте команду `glMaterial` для рідких змін чи при швидкій зміні більше, ніж однієї властивості матеріалу;
- завжди використовуйте команду `glLoadIdentity` для ініціалізації матриці замість завантаження власної копії одиничної матриці;

- використовуйте спеціальні виклики матриць, такі як glRotate, glTranslate, glScale, замість створення власних матриць обертання, переміщення чи масштабування та виклику команди glMultMatrix;
- використовуйте функції запитів, коли прикладна програма вимагає тільки декілька значень параметрів стану для своїх обчислень. Якщо програмі потрібно декілька значень з однієї групи атрибутів, то використовуйте команди glPushAttrib та glPopAttrib, для збереження та відновлення значень;
- використовуйте списки зображень для інкапсуляції викликів рендерінга жорстко заданих об'єктів, які будуть багаторазово відображатися;
- використовуйте текстурні об'єкти для інкапсуляції текстурних даних. Розміщуйте у текстурному об'єкті усі виклики команди glTexImage (включаючи *minmap*), потрібні для повного визначення текстури, а також зв'язані виклики команди glTexParametr (які визначають властивості структури);
- використовуйте обчислювачі Без'є для випадків простого розбиття поверхні, з метою мінімізації трафіку в клієнт-серверних середовищах;
- встановлюйте нормалі одиничної довжини, якщо це можливо, та уникайте додаткових витрат режиму GL_NORMALIZE. Уникайте використання команди glScale при застосуванні освітлення, тому що для нормальної роботи у цьому випадку необхідне включення режиму GL_NORMALIZE;
- встановіть режим glShadeModel у стан GL_FLAT, якщо не потрібен режим згладжування (*smooth shading*);
- використовуйте, якщо можливо, один виклик команди glClear для одного кадру;
- використовуйте один виклик glBegin(GL_TRIANGLES) для відображення множини незалежних трикутників замість використання множини викликів glBegin(GL_TRIANGLES) чи glBegin(GL_POLYGON). Аналогічно застосовуйте виклики команд glBegin(GL_QUADS) та glBegin(GL_LINES);
- застосування масивів вершин зменшує додаткові витрати на виклик функцій;

- використовуйте векторні форми команд, для передачі попередньо обчислених даних, та скалярні форми команд, для того щоб передати значення, що будуть обчислені відразу після виклику;
- якщо не має необхідності, уникайте використання різних режимів для передньої (*front*) та задньої (*back*) сторін многокутників.

Запитання для самоконтролю

1. Перелічіть відомі Вам поради з підвищення надійності графічних застосунків.
2. Які Вам відомі прийоми підвищення продуктивності застосунків?
3. Як підвищити ефективність виконання афінних перетворень в OpenGL?

Список літератури

1. *Баяковский Ю.М.* Графическая библиотека OpenGL: учебно-методическое пособие / Ю.М. Баярковский, А.В. Игнатенко, А.И. Фролов. – М.: Факультет вычислительной математики и кибернетики МГУ им. Ломоносова, 2003. – 132 с.
2. *Гилой В.* Интерактивная машинная графика / В. Гилой. – М.: Мир, 1981. – 384 с.
3. *Демченко В.В.* Функціональна модель графічних застосувань на основі OpenGL / В.В. Демченко, В.О. Анпілогова // Сучасні проблеми геометричного моделювання: Зб. праць міжнар. наук.-практ. конф. – Харків, 2001. – С. 194-196.
4. *Краснов М.В.* OpenGL. Графика в проектах Delphi / М.В. Краснов. – СПб.: БХВ – Санкт-Петербург, 2000. – 352 с.
5. *Михайленко В.Е.* Геометрическое моделирование и машинная графика в САПР: учебник / В.Е. Михайленко, В.Н. Кислоокий, А.А. Лященко и др. – К.: Выща шк., 1991. – 374 с.
6. *Препарата Ф*. Вычислительная геометрия : Введение / Ф. Препарата, М. Шеймос. – М.: Мир, 1989. – 480 с.
7. *Роджерс Д*. Математические основы машинной графики / Д. Роджерс, Дж. Адамс. – М.: Мир, 2001. – 556 с.
8. *Скворцов А.В*. Триангуляция Делоне и ее применение / А.В. Скворцов – Томск: Издательство Томского университета, 2002. – 128 с.
9. *Тихомиров Ю.* Программирование трехмерной графики / Ю. Тихомиров. – СПб.: БХВ – Санкт-Петербург, 1999. – 256 с.
10. *Шикин Е.В.* Компьютерная графика. Полигональные модели / Е.В. Шикин, А.В. Боресков. – М.: ДИАЛОГ-МИФИ, 2001. – 464 с.

Додаток 1. Модель освітлення OpenGL

Програма призначена для демонстрації моделі освітлення OpenGL на прикладі спрощеної астрономічної моделі «Червоний малюк», що складається з зірки та планети, що обертається навколо неї (зірка та планета створені за допомогою quadric-об'єктів). У програмі задаються характеристики джерела світла та характеристики матеріалів. Текст програми наведений нижче.

```
unit Main;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics,
Controls, Forms, Dialogs, OpenGL, Menus, ExtCtrls;
type
  TfrmGL = class(TForm)
    Timer1: TTimer;
    procedure FormCreate(Sender: TObject);
    procedure FormPaint(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
    procedure FormKeyDown(Sender: TObject; var Key: Word;
      Shift: TShiftState);
    procedure FormResize(Sender: TObject);
    procedure Timer1Timer(Sender: TObject);
  private
    DC : HDC;
    hrc: HGLRC;
    quadObj : GLUquadricObj;
  end;
var
  frmGL: TfrmGL;
  year : Integer = 0;
  day : Integer = 0;
implementation
{$R *.DFM}
procedure yearAdd;
begin
    year := (year + 5);
```

```
    If year > 360 then year := 0;
end;
{Перемалювання вікна}
procedure TfrmGL.FormPaint(Sender: TObject);
const
  sColor: array [0..3] of GLfloat = (1, 0.75, 0, 1);
  pColor: array [0..3] of GLfloat = (0.4, 0, 0.2, 1);
  mColor: array [0..3] of GLfloat = (0.7, 0.4, 0.5, 1);
  black : array [0..3] of GLfloat = (0, 0, 0, 1);
begin
// очищення буферу кольору та глибини
  glClear(GL_COLOR_BUFFER_BIT OR GL_DEPTH_BUFFER_BIT);
  glPushMatrix;
// малюємо сонце
    glPushMatrix;
// повертаємо сонце прямо
      glRotatef (90.0, 1.0, 0.0, 0.0);
// задаємо випромінювання світла
     glMaterialfv(GL_FRONT_AND_BACK,GL_EMISSION,@sColor);
      gluSphere (quadObj, 1.0, 15, 10);
// вимикаємо випромінювання
      glMaterialfv(GL_FRONT_AND_BACK,GL_EMISSION,@black);
    glPopMatrix;
// малюємо червону планету
    glRotatef (year, 0.0, 1.0, 0.0);
    glTranslatef (2.0, 0.0, 0.0);
    glRotatef (90.0, 1.0, 0.0, 0.0); // повертаємо прямо
// встановлення характеристик відбиття світла для планети
    glMaterialfv(GL_FRONT,GL_AMBIENT_AND_DIFFUSE,
@pColor);
    gluSphere (quadObj, 0.2, 10, 10);
   glMaterialfv(GL_FRONT,GL_AMBIENT_AND_DIFFUSE,@mColor);
  glPopMatrix;
  SwapBuffers(DC);
end;
{Встановдення формату пікселя}
```

```
procedure SetDCPixelFormat (hdc : HDC);
var
  pfd : TPixelFormatDescriptor;
  nPixelFormat : Integer;
begin
  FillChar (pfd, SizeOf (pfd), 0);
  pfd.dwFlags := PFD_DRAW_TO_WINDOW or PFD_SUPPORT_OPENGL
or PFD_DOUBLEBUFFER;
  nPixelFormat := ChoosePixelFormat (hdc, @pfd);
  SetPixelFormat (hdc, nPixelFormat, @pfd);
end;
{Створення форми}
procedure TfrmGL.FormCreate(Sender: TObject);
const
  lcol : Array [0..3] of GLFloat = (0.97, 0.956, 0, 1);
  lpos : Array [0..3] of GLFloat = (0, 0, 0, 1.5);
  spec : array [0..3] of GLfloat = (0.5, 0.5, 0.5, 0.5);
begin
  DC := GetDC (Handle);
  SetDCPixelFormat(DC);
  hrc := wglCreateContext(DC);
  wglMakeCurrent(DC, hrc);
  quadObj := gluNewQuadric; //створюємо quadric-об'єкт
  gluQuadricDrawStyle (quadObj, GLU_Fill);
// Задаємо характеристики світла
  glLightfv(GL_LIGHT0, GL_POSITION, @lpos);
  glLightfv(GL_LIGHT0, GL_DIFFUSE, @lcol);
  glMaterialf(GL_FRONT, GL_SHININESS, 60);
  glMaterialfv(GL_FRONT, GL_SPECULAR, @spec);
//встановлюємо режим згладжування
  glShadeModel (GL_SMOOTH);
  glEnable(GL_LIGHTING); // вмикаємо режим освітлення
  glEnable(GL_LIGHT0); //вмикаємо джерело освітлення
// встановлюємо режим тестування глибини
  glEnable(GL_DEPTH_TEST);
  Timer1.Enabled:=true;
```

```
end;
{Закінчення роботи програми}
procedure TfrmGL.FormDestroy(Sender: TObject);
begin
  gluDeleteQuadric (quadObj); //видаляємо quadric-об'єкт
//звільнюємо контексти пристрою та відображення
  wglMakeCurrent(0, 0);
  wglDeleteContext(hrc);
  ReleaseDC (Handle, DC);
  DeleteDC (DC);
end;
procedure TfrmGL.FormKeyDown(Sender: TObject; var Key:
Word;
  Shift: TShiftState);
begin
  If Key = VK_ESCAPE then Close;
end;
procedure TfrmGL.FormResize(Sender: TObject);
begin
  glViewport(0, 0, ClientWidth, ClientHeight);
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity;
  gluPerspective(60,ClientWidth/ClientHeight,1,20);
  glMatrixMode(GL_MODELVIEW);
  glLoadIdentity;
  glTranslatef (0.0, 0.0, -5.0);
  InvalidateRect(Handle, nil, False);
end;
procedure TfrmGL.Timer1Timer(Sender: TObject);
begin
  yearAdd;
  InvalidateRect(Handle, nil, False);
end;
end.
```

Додаток 2. Накладення текстури

Результатом виконання цієї програми є побудова куба, на грані якого нанесена текстура. Текстура задається растровим зображенням (рис. Д2.1) розміром 384×256 пікселів. Дане зображення розрізається на окремі зображення розміром 64×64 пікселів, які й наносяться на грані куба.

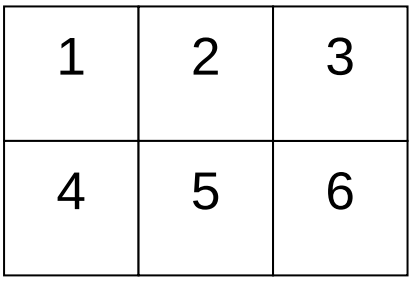


Рис. Д2.1. Растрове зображення текстури

```
unit Main;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics,
Controls, Forms, Dialogs,OpenGl, ExtCtrls;
type
  TForm1 = class(TForm)
    Timer1: TTimer;
    procedure FormCreate(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
    procedure FormResize(Sender: TObject);
    procedure FormPaint(Sender: TObject);
    procedure FormKeyDown(Sender: TObject; var Key: Word;
      Shift: TShiftState);
    procedure Timer1Timer(Sender: TObject);
  private
    { Private declarations }
    DC : HDC;
    hrc: HGLRC;
  public
    { Public declarations }
  end;
type
```

```
  TSquare = class
  private
    { Private declarations }
  public
    { Public declarations }
  procedure PrepareImage(bmap: string);
  procedure initlist(CB: GLint);
  procedure SiccorsBitMap(bm: string; src_x1, src_y1,
src_x2, src_y2: integer);
end;
var
  Form1: TForm1;
  Square1: TSquare;
implementation
{$R *.DFM}
{Встановлення формату пікселя}
procedure SetDCPixelFormat (hdc : HDC);
var
  pfd : TPixelFormatDescriptor;
  nPixelFormat : Integer;
begin
  FillChar (pfd, SizeOf (pfd), 0);
  pfd.dwFlags := PFD_DRAW_TO_WINDOW or PFD_SUPPORT_OPENGL
or PFD_DOUBLEBUFFER;
  nPixelFormat := ChoosePixelFormat (hdc, @pfd);
  SetPixelFormat (hdc, nPixelFormat, @pfd);
end;
procedure TForm1.FormCreate(Sender: TObject);
begin
  DC := GetDC (Handle);
  SetDCPixelFormat(DC);
  hrc := wglCreateContext(DC);
  wglMakeCurrent(DC, hrc);
  glShadeModel (GL_FLAT);
  Square1:=TSquare.Create;
  Square1.initlist(1);
```

```
  glEnable(GL_DEPTH_TEST);
  Timer1.Enabled:=true;
end;
procedure TForm1.FormDestroy(Sender: TObject);
begin
// видалення списку
  glDeleteLists(1,1);
// звільнюємо контексти пристрою та відображення
  wglMakeCurrent(0, 0);
  wglDeleteContext(hrc);
  ReleaseDC (Handle, DC);
  DeleteDC (DC);
end;
procedure TForm1.FormResize(Sender: TObject);
begin
  glViewport(0, 0, ClientWidth, ClientHeight);
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity;
  gluPerspective(60,ClientWidth/ClientHeight,1,20);
  glMatrixMode(GL_MODELVIEW);
  glLoadIdentity;
  glTranslatef (0.0, 0.0, -5.0);
  InvalidateRect(Handle, nil, False);
end;
procedure TForm1.FormPaint(Sender: TObject);
begin
// очищення буферу кольору та глибини
  glClear(GL_COLOR_BUFFER_BIT or GL_DEPTH_BUFFER_BIT);
  glRotatef (2, 0.0, 1.0, 0.0);
  glCallList(1);//виклик створеного списку багатокутника
  SwapBuffers(DC);
end;
procedure TForm1.FormKeyDown(Sender: TObject; var Key:
Word;
  Shift: TShiftState);
begin
```

```
 If Key = VK_ESCAPE then Close;
end;
// Підготовка текстури
procedure TSquare.PrepareImage(bmap: string);
type
  PPixelArray = ^TPixelArray;
  TPixelArray = array [0..0] of Byte;
var
  Bitmap : TBitmap;
  Data : PPixelArray;
  BMInfo : TBitmapInfo;
  I, ImageSize : Integer;
  Temp : Byte;
  MemDC : HDC;
begin
  Bitmap := TBitmap.Create;
  Bitmap.LoadFromFile (bmap);
  with BMinfo.bmiHeader do begin
    FillChar (BMInfo, SizeOf(BMInfo), 0);
    biSize := sizeof (TBitmapInfoHeader);
    biBitCount := 24;
    biWidth := Bitmap.Width;
    biHeight := Bitmap.Height;
    ImageSize := biWidth * biHeight;
    biPlanes := 1;
    biCompression := BI_RGB;
    MemDC := CreateCompatibleDC (0);
    GetMem (Data, ImageSize * 3);
    try
      GetDIBits (MemDC, Bitmap.Handle, 0, biHeight, Data,
                 BMInfo, DIB_RGB_COLORS);
      For I := 0 to ImageSize - 1 do begin
          Temp := Data [I * 3];
          Data [I * 3] := Data [I * 3 + 2];
          Data [I * 3 + 2] := Temp;
      end;
```

```
      glTexImage2d(GL_TEXTURE_2D, 0, 3, biWidth,
            biHeight, 0, GL_RGB, GL_UNSIGNED_BYTE, Data);
    finally
      FreeMem (Data);
      DeleteDC (MemDC);
      Bitmap.Free;
    end;
  end;
end;
//процедура розрізання вихідного зображення
procedure TSquare.SiccorsBitmap(bm: string; src_x1,
src_y1, src_x2, src_y2: integer);
var LoadBitmap,CopyBitmap,SaveBitmap: TBitmap;
    DestRect, SrcRect: TRect;
begin
  LoadBitmap:=TBitmap.Create;
  CopyBitmap:=TBitmap.Create;
  SaveBitmap:=TBitmap.Create;
  CopyBitmap.Height:=64;
  CopyBitmap.Width:=64;
  LoadBitmap.LoadFromFile(bm);
  CopyBitmap.Canvas.CopyMode:=cmSrcCopy;
  DestRect:=Rect(0,0,64,64);
  SrcRect:=Rect(src_x1,src_y1,src_x2,src_y2);
  Copybitmap.Canvas.BrushCopy(DestRect,      LoadBitmap,
SrcRect, clBlack);
  CopyBitmap.Canvas.CopyRect(DestRect, LoadBitmap.Canvas,
SrcRect);
  SaveBitmap.Assign(CopyBitmap);
  SaveBitmap.SaveToFile('xxx.bmp');
  CopyBitmap.free;
  SaveBitmap.free;
  LoadBitmap.free;
end;
procedure TSquare.initlist(CB: GLint);
begin
//встановлення параметрів текстури
```

```
  glTexParameteri(GL_TEXTURE_2D,   GL_TEXTURE_MIN_FILTER,
GL_linear);
  glTexParameteri(GL_TEXTURE_2D,   GL_TEXTURE_MAG_FILTER,
GL_linear);
  glTexEnvi(GL_Texture_ENV,GL_Texture_Env_Mode,Gl_Decal);
  glRenderMode(GL_RENDER);
  glEnable(GL_TEXTURE_2D);
//Створення та ініціалізація елементу списка
  glNewList(CB, GL_COMPILE);
    glColor3f (0.4, 0.5, 0.2);
    glPolygonMode (GL_Front,Gl_Fill);
    glPushMatrix;
      glRotatef(-45.0,1.0, 0.0, 0.0);
      glRotatef(225,0.0, 0.0, 1.0);
      SiccorsBitmap('numbers.bmp',0,0,128,128);
      prepareImage('xxx.bmp');
      glBegin(GL_QUADS);
        glNormal3f(-1.0, 0.0, 0.0);
        glTexCoord2d(1,0);glVertex3f(-0.5,-0.5,-0.5); //1
        glTexCoord2d(1,1);glVertex3f(-0.5,-0.5,0.5); //2
        glTexCoord2d(0,1);glVertex3f(-0.5,0.5,0.5); //3
        glTexCoord2d(0,0);glVertex3f(-0.5,0.5,-0.5); //4
      glEnd;
      SiccorsBitmap('numbers.bmp',128,0,256,128);
      prepareImage('xxx.bmp');
      glBegin(GL_QUADS);
        glNormal3f(0.0, 1.0, 0.0);
        glTexCoord2d(1,0);glVertex3f(-0.5,0.5,-0.5); //4
        glTexCoord2d(1,1);glVertex3f(-0.5,0.5,0.5); //3
        glTexCoord2d(0,1);glVertex3f(0.5,0.5,0.5); //5
        glTexCoord2d(0,0);glVertex3f(0.5,0.5,-0.5); //6
      glEnd;
      SiccorsBitmap('numbers.bmp',256,0,384,128);
      prepareImage('xxx.bmp');
      glBegin(GL_QUADS);
        glNormal3f(1.0, 0.0, 0.0);
```

```
        glTexCoord2d(1,1);glVertex3f(0.5,0.5,-0.5); //6
        glTexCoord2d(1,0);glVertex3f(0.5,0.5,0.5); //5
        glTexCoord2d(0,0);glVertex3f(0.5,-0.5,0.5); //7
        glTexCoord2d(0,1);glVertex3f(0.5,-0.5,-0.5); //8
      glEnd;
      SiccorsBitmap('numbers.bmp',0,128,128,256);
      prepareImage('xxx.bmp');
      glBegin(GL_QUADS);
        glNormal3f(0.0, -1.0, 0.0);
        glTexCoord2d(1,0);glVertex3f(0.5,-0.5,-0.5); //8
        glTexCoord2d(1,1);glVertex3f(0.5,-0.5,0.5); //7
        glTexCoord2d(0,1);glVertex3f(-0.5,-0.5,0.5); //2
        glTexCoord2d(0,0);glVertex3f(-0.5,-0.5,-0.5); //1
      glEnd;
      SiccorsBitmap('numbers.bmp',256,128,384,256);
      prepareImage('xxx.bmp');
      glBegin(GL_QUADS);
        glNormal3f(0.0, 0.0, 1.0);
        glTexCoord2d(0,1);glVertex3f(0.5,-0.5,0.5); //7
        glTexCoord2d(0,0);glVertex3f(0.5,0.5,0.5); //5
        glTexCoord2d(1,0);glVertex3f(-0.5,0.5,0.5); //3
        glTexCoord2d(1,1);glVertex3f(-0.5,-0.5,0.5); //2
      glEnd;
      SiccorsBitmap('numbers.bmp',128,128,256,256);
      prepareImage('xxx.bmp');
      glBegin(GL_QUADS);
        glNormal3f(0.0, 0.0, -1.0);
        glTexCoord2d(0,1);glVertex3f(0.5,0.5,-0.5);
        glTexCoord2d(0,0);glVertex3f(0.5,-0.5,-0.5);
        glTexCoord2d(1,0);glVertex3f(-0.5,-0.5,-0.5);
        glTexCoord2d(1,1);glVertex3f(-0.5,0.5,-0.5);
      glEnd;
    glPopMatrix;
  glEndList;
end;
procedure TForm1.Timer1Timer(Sender: TObject);
```

```
begin
  InvalidateRect(Handle, nil, False);
end;
end.
```

Додаток З. Демонстрація ефекту тіні

Дана програма демонструє малювання тіні для простого об'єкта. У програмі описана користувацька процедура для малювання тіні з урахуванням того, що усі грані куба паралельні координатним площинам. Тінь малюється у вигляді 6 окремих сірих багатокутників, для кожної грані об'єкта. Також у програмі використовуються окремі функції бібліотеки GLUT.

```
unit Main;

interface

uses

  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs, ExtCtrls, ComCtrls, StdCtrls, Menus, Buttons, OpenGL;

const

// колір туману

  fogColor: Array [0..3] of GLfloat = (0.5,0.5,0.5,1.0);

// колір площини

  glfSquareAmbient: Array [0..3] of GLfloat = (0.247, 0.199, 0.0745, 1.0);

  glfSquareDiffuse: Array [0..3] of GLfloat = (0.751, 0.606, 0.22648, 1.0);

  glfSquareSpecular: Array [0..3] of GLfloat = (0.6282, 0.556, 0.366, 1.0);

// джерело світла

  glfLightAmbient: Array [0..3] of GLfloat = (0.25, 0.25, 0.25, 1.0);

  glfLightDiffuse: Array [0..3] of GLfloat = (1.0, 1.0, 1.0, 1.0);

  glfLightSpecular: Array [0..3] of GLfloat = (1.0, 1.0, 1.0, 1.0);

  glfLightPosition: Array [0..3] of GLfloat = (0.0, 0.0, 20.0, 1.0);

  glfLightModelAmbient: Array [0..3] of GLfloat = (0.25, 0.25, 0.25, 1.0);

// позиція першого джерела світла

  LightPosition: Array [0..3] of GLfloat = (0.0, 0.0, 15.0, 1.0);

// позиція другого джерела світла

  glfLight1Position: Array [0..3] of GLfloat=(15.0, 15.0, -5.0, 1.0);
```

```
type
  TfrmGL = class(TForm)
    procedure Init;
    procedure SetProjection(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure SetDCPixelFormat;
    procedure FormDestroy(Sender: TObject);
    procedure FormKeyDown(Sender: TObject; var Key: Word;
                          Shift: TShiftState);
  public
    DC : HDC;
    hrc : HGLRC;
    cubeX, cubeY, cubeZ : GLfloat;
    cubeL, cubeH, cubeW : GLfloat;
    AddX, AddY, AddZ : GLfloat; // початкові зсуви
    SquareLength : GLfloat;     // площина
    procedure Square;
    procedure Shadow;
  protected
    procedure WMPaint(var Msg:TWMPaint);message WM_PAINT;
  end;
var
  frmGL: TfrmGL;
implementation
uses DGLUT;
{$R *.DFM}
{Реакція на події клавіатури}
procedure TfrmGL.FormKeyDown(Sender: TObject; var Key:
Word; Shift: TShiftState);
begin
  If Key = VK_ESCAPE then Close;
  If Key = VK_LEFT then begin
     cubeX := cubeX + 0.1;
     InvalidateRect(Handle, nil, False);
  end;
  If Key = VK_RIGHT then begin
```

```
      cubeX := cubeX - 0.1;
      InvalidateRect(Handle, nil, False);
  end;
  If Key = VK_UP then begin
      cubeZ := cubeZ + 0.1;
      InvalidateRect(Handle, nil, False);
  end;
  If Key = VK_DOWN then begin
      cubeZ := cubeZ - 0.1;
      InvalidateRect(Handle, nil, False);
  end;
end;
{Малювання тіні}
procedure TfrmGL.Shadow;
// розрахунок точки тіні для однієї точки об'єкта
  procedure OneShadow (x, y, z, h : GLfloat; var x1, y1 :
GLfloat);
  begin
    x1 := x*LightPosition[2]/(LightPosition[2]-(z+h));
    If LightPosition[0]<x
      then begin If x1>0 then x1:=LightPosition[0]+x1 end
     else begin If x1>0 then x1:=LightPosition[0]-x1 end;
    y1 := y*LightPosition[2]/(LightPosition[2]-(z+h));
    If LightPosition[1]<y
      then begin If y1>0 then y1:=LightPosition[1]+y1 end
     else begin If y1>0 then y1:=LightPosition[1]-y1 end;
    If x1<0 then x1 := 0 else
       If x1 > SquareLength then x1 := SquareLength;
    If y1 < 0 then y1 := 0 else
       If y1 > SquareLength then y1 := SquareLength;
  end;
var
  x1, y1, x2, y2, x3, y3, x4, y4 : GLfloat;
  wrkx1, wrky1, wrkx2, wrky2, wrkx3, wrky3, wrkx4, wrky4:
GLfloat;
begin
```

```
OneShadow(cubeX+cubeL,cubeY+cubeH,cubeZ,cubeW,x1,y1);
OneShadow(cubeX,cubeY+cubeH,cubeZ,cubeW,x2,y2);
OneShadow(cubeX,cubeY,cubeZ,cubeW,x3,y3);
OneShadow(cubeX+cubeL,cubeY,cubeZ,cubeW,x4,y4);
If cubeZ+cubeW >= 0 then
begin
  glBegin(GL_QUADS);
    glVertex3f(x1,y1,-0.99); glVertex3f(x2,y2,-0.99);
    glVertex3f(x3,y3,-0.99); glVertex3f(x4,y4,-0.99);
  glEnd;
end;
If cubeZ >= 0 then
begin
   wrkx1 := x1; wrky1 := y1; wrkx2 := x2; wrky2 := y2;
   wrkx3 := x3; wrky3 := y3; wrkx4 := x4; wrky4 := y4;
   OneShadow(cubeX+cubeL,cubeY+cubeH,cubeZ,0,x1,y1);
   OneShadow(cubeX,cubeY+cubeH,cubeZ,0,x2,y2);
   OneShadow(cubeX,cubeY,cubeZ,0,x3,y3);
   OneShadow(cubeX+cubeL,cubeY,cubeZ,0,x4,y4);
   glBegin(GL_QUADS);
     glVertex3f(x1,y1,-0.99); glVertex3f(x2,y2,-0.99);
     glVertex3f(x3,y3,-0.99); glVertex3f(x4,y4,-0.99);
     glVertex3f(wrkx2, wrky2, -0.99);
     glVertex3f(x2,y2,-0.99); glVertex3f(x3,y3,-0.99);
     glVertex3f(wrkx3, wrky3, -0.99);
     glVertex3f(wrkx1, wrky1, -0.99);
     glVertex3f(wrkx4, wrky4, -0.99);
     glVertex3f(x4,y4,-0.99); glVertex3f(x1,y1,-0.99);
     glVertex3f(wrkx1, wrky1, -0.99);
     glVertex3f(x1,y1,-0.99); glVertex3f(x2,y2,-0.99);
     glVertex3f(wrkx2, wrky2, -0.99);
     glVertex3f(wrkx3, wrky3, -0.99);
     glVertex3f(x3,y3,-0.99); glVertex3f(x4,y4,-0.99);
     glVertex3f (wrkx4, wrky4, -0.99);
   glEnd;
end;
```

```
end;
{Малювання площини}
procedure TfrmGL.Square;
begin
  glPushAttrib (GL_ALL_ATTRIB_BITS );
   glMaterialfv(GL_FRONT,GL_AMBIENT,@glfSquareAmbient);
   glMaterialfv(GL_FRONT,GL_DIFFUSE,@glfSquareDiffuse);
   glMaterialfv(GL_FRONT,GL_SPECULAR,@glfSquareSpecular);
   glMaterialf(GL_FRONT, GL_SHININESS, 90.2);
   glBegin(GL_QUADS);
      glNormal3f(squarelength/2.0,squarelength/2.0,-1.0);
      glVertex3f(squarelength, squarelength, -1.0);
      glVertex3f(0.0, squarelength, -1.0);
      glVertex3f(0.0, 0.0, -1.0);
      glVertex3f(squarelength, 0.0, -1.0);
    glEnd;
  glPopAttrib;
end;
{Ініціалізація}
procedure TfrmGL.Init;
begin
  glEnable (GL_FOG); glEnable(GL_NORMALIZE);
  glEnable(GL_DEPTH_TEST);
  glLightfv(GL_LIGHT0, GL_AMBIENT, @glfLightambient);
  glLightfv(GL_LIGHT0, GL_DIFFUSE, @glfLightdiffuse);
  glLightfv(GL_LIGHT0, GL_SPECULAR, @glfLightspecular);
  glLightfv(GL_LIGHT0, GL_POSITION, @glfLightposition);
  glLightModelfv(GL_LIGHT_MODEL_AMBIENT,
@glfLightmodelambient);
  // друге джерело світла
  glLightfv(GL_LIGHT1, GL_AMBIENT, @glfLightambient);
  glLightfv(GL_LIGHT1, GL_DIFFUSE, @glfLightdiffuse);
  glLightfv(GL_LIGHT1, GL_SPECULAR, @glfLightspecular);
  glLightfv(GL_LIGHT1, GL_POSITION, @glfLight1position);
  glEnable(GL_LIGHTING);
  glEnable(GL_LIGHT0);
```

```
  glEnable(GL_LIGHT1);
end;
{Зміна розмірів вікна}
procedure TfrmGL.SetProjection(Sender: TObject);
begin
  glViewport(0, 0, ClientWidth, ClientHeight);
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity;
  glFrustum (-0.5, 0.5, -0.5, 0.5, 1.0, 100.0);
  glMatrixMode(GL_MODELVIEW);
  glLoadIdentity;
  glTranslatef(0.0, 0.0, -32.0);
  glRotatef(120.0, 1.0, 0.0, 0.0);
  glRotatef(180.0, 0.0, 1.0, 0.0);
  glRotatef(40.0, 0.0, 0.0, 1.0);
  InvalidateRect(Handle, nil, False);
end;
{Початок роботи програми}
procedure TfrmGL.FormCreate(Sender: TObject);
begin
  DC := GetDC(Handle);
  SetDCPixelFormat;
  hrc := wglCreateContext(DC);
  wglMakeCurrent(DC, hrc);
  Init;
// параметри туману
  glFogi(GL_FOG_MODE, GL_EXP);
  glFogfv(GL_FOG_COLOR, @fogColor);
  glFogf(GL_FOG_DENSITY, 0.015);
  SquareLength := 50.0;
  AddX := 0; AddY := 0; AddZ := 0;
  cubeX := 1.0; cubeY := 2.0; cubeZ := 3.0;
  cubeL := 1.0; cubeH := 2.0; cubeW := 3.0;
end;
{Аналог події OnPaint}
procedure TfrmGL.WMPaint(var Msg: TWMPaint);
```

```
var
  ps : TPaintStruct;
const
  CubeColor : Array [0..3] of GLfloat = (1.0, 0.0, 0.0,
0.0);
begin
  BeginPaint(Handle, ps);
    glClear(GL_COLOR_BUFFER_BIT or GL_DEPTH_BUFFER_BIT);
    glPushMatrix;
      glTranslatef(AddX, AddY, AddZ);
      glEnable (GL_LIGHT1);
        Square;
      glDisable (GL_LIGHT1);
      glPushAttrib (GL_ALL_ATTRIB_BITS );
glMaterialfv(GL_FRONT,GL_AMBIENT_AND_DIFFUSE,@CubeColor);
        glPushMatrix;
          glTranslatef (cubeX, cubeY, cubeZ);
          glScalef (cubeL, cubeH, cubeW);
          glutSolidCube (1.0);
        glPopMatrix;
      glPopAttrib;
      Shadow;
    glPopMatrix;
    SwapBuffers(DC);
  EndPaint(Handle, ps);
end;
{Кінець роботи програми}
procedure TfrmGL.FormDestroy(Sender: TObject);
begin
  wglMakeCurrent(0, 0);
  wglDeleteContext(hrc);
  ReleaseDC(Handle, DC);
  DeleteDC (DC);
end;
{Формат пікселів}
procedure TfrmGL.SetDCPixelFormat;
```

```
var
  nPixelFormat: Integer;
  pfd: TPixelFormatDescriptor;
begin
  FillChar(pfd, SizeOf(pfd), 0);
  with pfd do begin
    nSize     := sizeof(pfd);
    nVersion  := 1;
    dwFlags   := PFD_DRAW_TO_WINDOW or PFD_SUPPORT_OPENGL
or PFD_DOUBLEBUFFER;
    iPixelType:= PFD_TYPE_RGBA;
    cColorBits:= 24;
    cDepthBits:= 32;
    iLayerType:= PFD_MAIN_PLANE;
  end;
  nPixelFormat := ChoosePixelFormat(DC, @pfd);
  SetPixelFormat(DC, nPixelFormat, @pfd);
end;
end.
```

Додаток 4. Демонстрація використання Tess-об'єктів, текстури та джерела світла зі змінюваним положенням

Наведена нижче програма демонструє принципи використання tess-об'єктів в OpenGL, накладення текстури та джерела світла зі змінним положенням. На відміну від попередніх прикладів даний виконаний в середовищі C++ Builder. Окрім основних описаних можливостей він також містить реалізацію засобів керування зображенням за допомогою миші: масштабування зображення, обертання в горизонтальній і вертикальній площинах. Програма складається з модуля головного вікна і додаткового модуля, в якому реалізовані списки зображення.

uDraw.h

```
//---------------------------------------------------------
#ifndef uDrawH
#define uDrawH
//---------------------------------------------------------
#include <Classes.hpp>
//---------------------------------------------------------
const GLint TessList=1;
const GLint SquareList=2;
const GLfloat H = 8;
typedef GLdouble TVector[3];
//---------------------------------------------------------
void SetDCPixelFormat(HDC hdc);
void DrawAxes(bool Local);
void ColorToRGB(int Color,GLfloat *R,GLfloat *G,GLfloat *B);
void InitLists(void);
void Calc3PointsNormal(TVector p1, TVector p2, TVector p3,
TVector Normal);
//---------------------------------------------------------
#endif
```

uDraw.cpp

```
//---------------------------------------------------------
#include <vcl.h>
#include <gl.h>
#include <glu.h>
#include <math.h>
#pragma hdrstop
```

```
#include "uDraw.h"
//---------------------------------------------------------------
#pragma package(smart_init)
//============= Встановити формат пікселя ================//
void SetDCPixelFormat(HDC hdc)
{ PIXELFORMATDESCRIPTOR pfd, *ppfd; int iPF;
  ppfd = &pfd;
  ppfd->nSize    = sizeof(PIXELFORMATDESCRIPTOR);
  ppfd->dwFlags = PFD_DRAW_TO_WINDOW | PFD_SUPPORT_OPENGL |
PFD_DOUBLEBUFFER;
  iPF = ChoosePixelFormat(hdc, ppfd);
  SetPixelFormat(hdc, iPF, ppfd);
}
//================= Малювання осей ==================//
void DrawAxes(bool Local)
{ GLfloat color[3]; // масив для збереження складових кольору
  try
  { glPushMatrix();     // Запамятали поточну матрицю
     // запамятали поточний колір
     glGetFloatv(GL_CURRENT_COLOR, color);
     // Встановили масштабування по всім осям
     glScalef(0.75, 0.75, 0.75);
     glBegin(GL_LINES);
      if(Local) glColor3f(0, 0, 0); else glColor3f (1, 0, 0);
          glVertex3f(0, 0, 0);
          glVertex3f(3, 0, 0);
      if(Local) glColor3f(0, 0, 0); else glColor3f (0, 1, 0);
          glVertex3f(0, 0, 0);
          glVertex3f(0, 3, 0);
      if(Local) glColor3f(0, 0, 0); else glColor3f (0, 0, 1);
          glVertex3f(0, 0, 0);
          glVertex3f(0, 0, 3);
     glEnd();
     // буква X
     if(Local) glColor3f(0, 0, 0); else glColor3f(1, 0, 0);
     glBegin(GL_LINES);
```

```
          glVertex3f(3.1, -0.05, 0.1);
          glVertex3f(3.1, 0.05, -0.1);
          glVertex3f(3.1, -0.05, -0.1);
          glVertex3f(3.1, 0.05, 0.1);
      glEnd();
      // буква Y
      if(Local) glColor3f(0, 0, 0); else glColor3f(0, 1, 0);
      glBegin(GL_LINES);
          glVertex3f(0.0, 3.1, 0.0);
          glVertex3f(0.0, 3.1, -0.1);
          glVertex3f(0.0, 3.1, 0.0);
          glVertex3f(0.1, 3.1, 0.1);
          glVertex3f(0.0, 3.1, 0.0);
          glVertex3f(-0.1, 3.1, 0.1);
      glEnd();
      // буква Z
      if(Local) glColor3f(0, 0, 0); else glColor3f(0, 0, 1);
      glBegin(GL_LINES);
          glVertex3f(0.1, -0.1, 3.1);
          glVertex3f(-0.1, -0.1, 3.1);
          glVertex3f(0.1, 0.1, 3.1);
          glVertex3f(-0.1, 0.1, 3.1);
          glVertex3f(-0.1, -0.1, 3.1);
          glVertex3f(0.1, 0.1, 3.1);
      glEnd();
   }
   __finally
   { // Відновлюємо значення поточного кольору
      glColor3f(color[0], color[1], color[2]);
      glPopMatrix(); // відновлюємо значення матриць
   }
}
//========== Розклад типу TColor на RGB =================//
void ColorToRGB(int Color,GLfloat *R,GLfloat *G,GLfloat *B)
{ *R = ((GLfloat) (Color & 0xFF))/255;
  *G = ((GLfloat)((Color & 0xFF00) >> 8))/255;
```

```
  *B = ((GLfloat)((Color & 0xFF0000) >> 16))/255;
}
//---------------------------------------------------------
const int Max = 10;
TVector External[Max] =
{{-3.0,-5.0,0.0},
 {5.0,-5.0,0.0},
 {3.0,-2.0,0.0},
 {0.0,-2.0,0.0},
 {5.0,2.0,0.0},
 {3.0,5.0,0.0},
 {-5.0,5.0,0.0},
 {-3.0,2.0,0.0},
 {0.0,2.0,0.0},
 {-5.0,-2.0,0.0}
};
TVector Internal[Max] =
{{-2.5,-4.5,0.0},
 {4.0,-4.5,0.0},
 {2.5,-2.5,0.0},
 {-1.0,-2.5,0.0},
 {3.5,2.5,0.0},
 {2.5,4.5,0.0},
 {-4.0,4.5,0.0},
 {-2.5,2.5,0.0},
 {1.0,2.5,0.0},
 {-4.0,-2.5,0.0}
};
//---------------------------------------------------------
GLuint texture[1];  // Месце для накладення 1-ї текстури
typedef struct RGBImageRec
{ GLint sizeX, sizeY;
  GLubyte *data;
} TRGBImageRec;
//---------------------------------------------------------
TRGBImageRec *ImageLoad(AnsiString TexFileName)
```

```
{ TRGBImageRec *rgbImageRec = NULL;
  Graphics::TBitmap *bitmap = NULL;
  int dx = 0, dy = 0;
  try
  { rgbImageRec = new TRGBImageRec();
     bitmap = new Graphics::TBitmap();
     bitmap->LoadFromFile(TexFileName);
     dx = rgbImageRec->sizeX = bitmap->Width;
     dy = rgbImageRec->sizeY = bitmap->Height;
     rgbImageRec->data=(GLubyte*)malloc(bitmap->Width*bitmap-
>Height*3*sizeof(GLubyte));
     for(int i = 0; i < dy; i++)
     { for(int j = 0; j < dx; j++)
       {*(rgbImageRec->data+(i*dy+j)*3)=
(GLubyte)GetRValue(bitmap->Canvas->Pixels[j][dy-i-1]);
          *(rgbImageRec->data+(i*dy+j)*3+1)=
(GLubyte)GetGValue(bitmap->Canvas->Pixels[j][dy-i-1]);
          *(rgbImageRec->data+(i*dy+j)*3+2)=
(GLubyte)GetBValue(bitmap->Canvas->Pixels[j][dy-i-1]);
       }
     }
  }
  __finally
  { delete bitmap;
  }
  return rgbImageRec;
}
//---------------------------------------------------------
void InitLists(void)
{ TRGBImageRec *tex = ImageLoad("..\\Bmp\\KNUBA.bmp");
  glGenTextures(1, texture);
  glBindTexture(GL_TEXTURE_2D, texture[0]);
  glTexParameteri(GL_TEXTURE_2D,       GL_TEXTURE_MIN_FILTER,
GL_LINEAR);
  glTexParameteri(GL_TEXTURE_2D,       GL_TEXTURE_MAG_FILTER,
GL_LINEAR);
  //glTexEnvi(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_DECAL);
  glTexEnvi(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_BLEND);
```

```
  glRenderMode(GL_RENDER);
  //glTexParameteri(GL_TEXTURE_2D,         GL_TEXTURE_WRAP_S,
GL_CLAMP );
  //glTexParameteri(GL_TEXTURE_2D,         GL_TEXTURE_WRAP_T,
GL_CLAMP );
  glTexImage2D(GL_TEXTURE_2D, 0, 3, tex->sizeX, tex->sizeY,
0, GL_RGB, GL_UNSIGNED_BYTE, tex->data);
  GLUtesselator *gluTess = gluNewTess();
  glNewList(TessList, GL_COMPILE);
     glEnable(GL_TEXTURE_GEN_S);
     glEnable(GL_TEXTURE_GEN_T);
     gluTessCallback(gluTess,                GLU_TESS_BEGIN,
(void(CALLBACK*)(void))glBegin);
     gluTessCallback(gluTess,               GLU_TESS_VERTEX,
(void(CALLBACK*)(void))glVertex3dv);
     gluTessCallback(gluTess,                 GLU_TESS_END,
(void(CALLBACK*)(void))glEnd);
     gluTessBeginPolygon(gluTess, NULL);
       gluTessBeginContour(gluTess);
          for(int i=0; i<Max; i++) gluTessVertex(gluTess,
External[i], External[i]);
       gluTessEndContour(gluTess);
       gluTessBeginContour(gluTess);
          for(int i=0; i<Max; i++) gluTessVertex(gluTess,
Internal[i], Internal[i]);
       gluTessEndContour(gluTess);
     gluTessEndPolygon(gluTess);
     glDisable(GL_TEXTURE_GEN_S);
     glDisable(GL_TEXTURE_GEN_T);
  glEndList();
  gluDeleteTess(gluTess);
  TVector normal = {0.0, 0.0, 0.0};
  TVector p1, p2, p3;
  glNewList(SquareList, GL_COMPILE);
     glBegin(GL_QUADS);
     for(int i=0; i<Max; i++)
     { // задаємо три точки площини
       p1[0] = External[i][0];
```

```
      p1[1] = External[i][1];
      p1[2] = External[i][2]-H/2;
      if(i == Max-1)
      { p2[0] = External[0][0];
        p2[1] = External[0][1];
        p2[2] = External[0][2]-H/2;
        p3[0] = External[0][0];
        p3[1] = External[0][1];
        p3[2] = External[0][2]+H/2;
      }
      else
      { p2[0] = External[i+1][0];
        p2[1] = External[i+1][1];
        p2[2] = External[i+1][2]-H/2;
        p3[0] = External[i+1][0];
        p3[1] = External[i+1][1];
        p3[2] = External[i+1][2]+H/2;
      }
      // рахуємо одиничну нормаль по 3-м точкам
      Calc3PointsNormal(p1, p2, p3, normal);
      // задаємо нормаль до площини для всіх її точок
      glNormal3f(normal[0], normal[1], normal[2]);
      glTexCoord2d (0.0, 0.0);
      glVertex3f(External[i][0],            External[i][1],
External[i][2]-H/2);
      if(i == Max-1)
      { glTexCoord2d (1.0, 0.0);
        glVertex3f(External[0][0],          External[0][1],
External[0][2]-H/2);
        glTexCoord2d (1.0, 2.0);
        glVertex3f(External[0][0],          External[0][1],
External[0][2]+H/2);
      }
      else
      { glTexCoord2d (1.0, 0.0);
        glVertex3f(External[i+1][0],      External[i+1][1],
External[i+1][2]-H/2);
```

```
          glTexCoord2d (1.0, 2.0);
          glVertex3f(External[i+1][0],      External[i+1][1],
External[i+1][2]+H/2);
       }
       glTexCoord2d (0.0, 2.0);
       glVertex3f(External[i][0],            External[i][1],
External[i][2]+H/2);
       // задаємо три точки площини
       p3[0] = Internal[i][0];
       p3[1] = Internal[i][1];
       p3[2] = Internal[i][2]-H/2;
       if(i == Max-1)
       { p2[0] = Internal[0][0];
          p2[1] = Internal[0][1];
          p2[2] = Internal[0][2]-H/2;
          p1[0] = Internal[0][0];
          p1[1] = Internal[0][1];
          p1[2] = Internal[0][2]+H/2;
       }
       else
       { p2[0] = Internal[i+1][0];
          p2[1] = Internal[i+1][1];
          p2[2] = Internal[i+1][2]-H/2;
          p1[0] = Internal[i+1][0];
          p1[1] = Internal[i+1][1];
          p1[2] = Internal[i+1][2]+H/2;
       }
       // рахуємо одиничну нормаль по 3-м точкам
       Calc3PointsNormal(p1, p2, p3, normal);
       // задаємо нормаль до площини для всіх її точок
       glTexCoord2d (0.0, 0.0);
       glNormal3f(normal[0], normal[1], normal[2]);
       glVertex3f(Internal[i][0],            Internal[i][1],
Internal[i][2]-H/2);
       if(i == Max-1)
       { glTexCoord2d (2.0, 0.0);
```

```
          glVertex3f(Internal[0][0],          Internal[0][1],
Internal[0][2]-H/2);
          glTexCoord2d (2.0, 2.0);
          glVertex3f(Internal[0][0],          Internal[0][1],
Internal[0][2]+H/2);
       }
       else
       { glTexCoord2d (2.0, 0.0);
          glVertex3f(Internal[i+1][0],      Internal[i+1][1],
Internal[i+1][2]-H/2);
          glTexCoord2d (2.0, 2.0);
          glVertex3f(Internal[i+1][0],      Internal[i+1][1],
Internal[i+1][2]+H/2);
       }
       glTexCoord2d (0.0, 2.0);
       glVertex3f(Internal[i][0],            Internal[i][1],
Internal[i][2]+H/2);
     }
     glEnd();
  glEndList();
  delete tex;
}
//====== Розрахунок одиничної нормалі до площини =========//
void Calc3PointsNormal(TVector p1, TVector p2, TVector p3,
TVector Normal)
{ GLfloat wrki,vx1,vy1,vz1,vx2,vy2,vz2,nx,ny,nz;
  vx1 = p1[0] - p2[0];
  vy1 = p1[1] - p2[1];
  vz1 = p1[2] - p2[2];
  vx2 = p2[0] - p3[0];
  vy2 = p2[1] - p3[1];
  vz2 = p2[2] - p3[2];
  nx = vy1 * vz2 - vz1 * vy2;
  ny = vz1 * vx2 - vx1 * vz2;
  nz = vx1 * vy2 - vy1 * vx2;
  wrki = sqrt (nx * nx + ny * ny + nz * nz);
  if (wrki <= 1E-8) wrki = 1;
  Normal[0] = nx / wrki;
```

```
  Normal[1] = ny / wrki;
  Normal[2] = nz / wrki;
}
```

ufFormGL.h

```
//---------------------------------------------------------------
#ifndef ufFormGLH
#define ufFormGLH
//---------------------------------------------------------------
#include <Classes.hpp>
#include <Controls.hpp>
#include <StdCtrls.hpp>
#include <Forms.hpp>
//---------------------------------------------------------------
class TFormGL : public TForm
{
__published:   // IDE-managed Components
     void __fastcall FormResize(TObject *Sender);
     void __fastcall FormPaint(TObject *Sender);
     void __fastcall FormKeyDown(TObject *Sender, WORD &Key,
TShiftState Shift);
     void __fastcall FormMouseMove(TObject *Sender,
TShiftState Shift, int X, int Y);
     void __fastcall FormMouseWheel(TObject *Sender,
TShiftState Shift, int WheelDelta,
          TPoint &MousePos, bool &Handled);
private:  // User declarations
     HDC   FGLDC;    // посилання на контекст пристрою
     HGLRC FHRC;     // посилання на контекст відображення
     int FMouseX;
     int FMouseY;
     GLfloat FTrans[3];
     TColor  FColor; // колір
     GLfloat FR;     // складова R (червона) кольору
     GLfloat FG;     // складова G (зелена) кольору
     GLfloat FB;     // складова B (синя) кольору
```

```
    GLfloat FEye[3];  // положення ока спостерігача (камери)
    GLfloat FCenter[3]; // точка, куди дивиться спостерігач
    GLfloat FUp[3]; // вектор повороту сцени
    GLfloat FPAngle;    // кут повороту навколо осі OZ
    GLfloat FQAngle;    // кут повороту навколо горизонталі
    GLfloat FDistance;  // дистанція до центру сцени
    GLfloat FZnear;
    GLfloat FZfar;
    DWORD uTimerId;
public:        // User declarations
    __fastcall  TFormGL(TComponent* Owner);
    __fastcall ~TFormGL(void);
};
//---------------------------------------------------------------
extern PACKAGE TFormGL *FormGL;
//---------------------------------------------------------------
#endif
```

ufFormGL.cpp

```
//---------------------------------------------------------------
#include <vcl.h>
#include <gl.h>
#include <glu.h>
#include <math.h>
#include <mmsystem.h>
#pragma hdrstop
#include "ufFormGL.h"
#include "uDraw.h"
//---------------------------------------------------------------
#pragma package(smart_init)
#pragma resource "*.dfm"
TFormGL *FormGL;
const GLfloat P[4] = {H, 0.0, H/2+2, 1.0};
const GLfloat D[4] = {0.0, 0.0, H, 1.0};
GLint Spin = H;
//---------------------------------------------------------------
```

```
void TimeCallBack(UINT uTimerID, UINT uMessage, DWORD dwUser,
DWORD dw1, DWORD dw2)
{ //Spin -= 1;
  Spin += 1;
  InvalidateRect(FormGL->Handle, NULL, false);
}
//---------------------------------------------------------
__fastcall TFormGL::TFormGL(TComponent* Owner)
                      :TForm(Owner)
{ //----------------для 3D-----------------------------------
  FTrans[0] = 0.0;
  FTrans[1] = 0.0;
  FTrans[2] = 0.0;
  FMouseX = 0;
  FMouseY = 0;
  FEye[0] = 5.0;
  FEye[1] = 5.0;
  FEye[2] = 5.0;
  FCenter[0] = 0.0;
  FCenter[1] = 0.0;
  FCenter[2] = 0.0;
  FUp[0] = 0.0;
  FUp[1] = 0.0;
  FUp[2] = 1.0;
  FPAngle = 45.0;
  FQAngle = 45.0;
  FDistance = 17.0;
  FZnear = 1.0;
  FZfar  = 30.0;
  //---------------------------------------------------------
  FGLDC = GetDC(Handle);
  SetDCPixelFormat(FGLDC);
  FHRC = wglCreateContext(FGLDC);
  wglMakeCurrent(FGLDC, FHRC);
  FColor = clBlack; //clCream; //clSkyBlue; //clCream;
  ColorToRGB(FColor, &FR, &FG, &FB);
```

```
  glClearColor(FR, FG, FB, 1.0);
  glEnable(GL_DEPTH_TEST);
  glEnable(GL_LIGHT0);
  glEnable(GL_LIGHTING);
  glEnable(GL_COLOR_MATERIAL);
  glEnable(GL_TEXTURE_2D);// Дозволити накладення текстури
  glColorMaterial(GL_FRONT_AND_BACK, GL_AMBIENT_AND_DIFFUSE);
  InitLists();
  uTimerId=timeSetEvent(20,0,TimeCallBack,0,TIME_PERIODIC);
}
//---------------------------------------------------------
__fastcall TFormGL::~TFormGL(void)
{ timeKillEvent(uTimerId);
  glDeleteLists(TessList, 2);
  wglMakeCurrent(NULL, NULL);
  wglDeleteContext(FHRC);
  ReleaseDC(Handle, FGLDC);
  DeleteDC(FGLDC);
}
//---------------------------------------------------------
void __fastcall TFormGL::FormResize(TObject *Sender)
{ glViewport(0, 0, ClientWidth, ClientHeight);
//----------------для 3D-----------------------------------
  glMatrixMode(GL_PROJECTION);   // Матриця проекцій
  glLoadIdentity();
  gluPerspective(70.0,ClientWidth/ClientHeight,FZnear,FZfar);
  glTranslatef(FTrans[0], FTrans[1], FTrans[2]);
  //---- Розраховуємо координати положення спостерігача ---//
  FEye[0]=FDistance*sin(FPAngle*M_PI/180)*
cos(FQAngle*M_PI/180)+FCenter[0];
  FEye[1]=FDistance*sin(FPAngle*M_PI/180)*
sin(FQAngle*M_PI/180)+FCenter[1];
  FEye[2]=FDistance*cos(FPAngle*M_PI/180)+FCenter[2];
  // Перевіряємо, чи не перекрутилися ми на 180 градусів
  int k = (int)FPAngle/360;
```

```
  if(FPAngle>=0+360*k  &&  FPAngle<=180+360*k)  FUp[2]  =  1.0;
else FUp[2] = -1.0;
  // Встановлюємо позицію спостерігача і сцени
  gluLookAt(FEye[0],     FEye[1],     FEye[2],     FCenter[0],
FCenter[1], FCenter[2], FUp[0], FUp[1], FUp[2]);
  glMatrixMode(GL_MODELVIEW);   // Матриця моделі
  glLoadIdentity();
//---------------------------------------------------------
  InvalidateRect(Handle, NULL, false); // Перемалювати вікно
}
//---------------------------------------------------------
void __fastcall TFormGL::FormPaint(TObject *Sender)
{ glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
  glPushMatrix();
     glPushMatrix();
       glRotatef(Spin, 0.0, 0.0, 1.0);
       glLightfv(GL_LIGHT0, GL_POSITION, P);
     glPopMatrix();
     glColor3f(0.3, 0.7, 0.5);
     glPushMatrix();
       glTranslatef(0.0, 0.0, -H/2);
       glNormal3f(0.0, 0.0, -1.0);
       glCallList(TessList);
     glPopMatrix();
     glPushMatrix();
       glNormal3f(0.0, 0.0, 1.0);
       glTranslatef(0.0, 0.0, H/2);
       glCallList(TessList);
     glPopMatrix();
     glCallList(SquareList);
  glPopMatrix();
  DrawAxes(false); // малюємо осі вихідної системи координат
  SwapBuffers(FGLDC);
}
//---------------------------------------------------------
void __fastcall TFormGL::FormKeyDown(TObject *Sender, WORD
&Key, TShiftState Shift)
```

```
{ if (Key == VK_ESCAPE) Close();
  if (Key == VK_RIGHT)  FTrans[0] += 0.05;
  if (Key == VK_LEFT)   FTrans[0] -= 0.05;
  if (Key == VK_UP)     FTrans[1] += 0.05;
  if (Key == VK_DOWN)   FTrans[1] -= 0.05;
  FormResize(this);
}
//---------------------------------------------------------
void __fastcall TFormGL::FormMouseMove(TObject *Sender,
TShiftState Shift, int X, int Y)
{ // ------------ для повороту в 3D ------------------//
  if(Shift.Contains(ssLeft))
  { FDistance=sqrt((FEye[0]-FCenter[0])*(FEye[0]-FCenter[0])+
(FEye[1]-FCenter[1])*(FEye[1]-FCenter[1])+(FEye[2]-
FCenter[2])*(FEye[2]-FCenter[2]));
     if(FMouseX != X)
     { if(X>FMouseX) FQAngle -= 2; else FQAngle += 2;
       FMouseX = X;
     }
     if(FMouseY != Y)
     { if(Y>FMouseY) FPAngle -= 2; else FPAngle += 2;
       FMouseY = Y;
     }
     FormResize(this);
  }
}
//---------------------------------------------------------
void __fastcall TFormGL::FormMouseWheel(TObject *Sender,
TShiftState Shift, int WheelDelta, TPoint &MousePos, bool
&Handled)
{ //----- Приблизити або віддалити зображення для 3D ------//
  if(!WheelDelta) return;
  if(WheelDelta > 0)
  { FDistance -= 1;
     if(FDistance <= FZnear) FDistance = FZnear + 1;
     FMouseX = MousePos.x;
     FMouseY = MousePos.y;
```

```
    }
    if(WheelDelta < 0)
    {  FDistance += 1;
       FMouseX = MousePos.x;
       FMouseY = MousePos.y;
       if(FDistance > FZfar) FDistance = FZfar - 1;
    }
    FormResize(this);
}
```

Алфавітний покажчик

E

З

І

К

Л

М

Н

О

П

Р

С

Т

Ф

Ц

Навчальне видання

Демченко Віктор Вікторович

Бородавка Євгеній Володимирович

ГЕОМЕТРИЧНЕ МОДЕЛЮВАННЯ І КОМП'ЮТЕРНА ГРАФІКА

Навчальний посібник

Редагування та коректура

Комп'ютерна верстка

Оформлення обкладинки *Є.В. Бородавки*

Підписано до друку 29.09.2010. Формат 60x84 1/16.
Ум. друк. арк. 11,0. Облік.-вид. арк. 11,0.
Тираж 200 прим. Вид. № 3/І-10. Зам. № 16/1-03.

КНУБА, Повітрофлотський проспект, 31, Київ, Україна, 03680

E-mail: red-isdat@knuba.edu.ua

Віддруковано в редакційно-видавничому відділі
Київського національного університету будівництва і архітектури

Свідоцтво про внесення до Державного реєстру суб'єктів
Видавничої справи ДК №808 від 13.02.2002 р.