МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ БУДІВНИЦТВА І АРХІТЕКТУРИ

Є.В. Бородавка

О.О. Терентьєв

КОМП'ЮТЕРНА ГРАФІКА

Навчальний посібник для студентів вищих технічних навчальних закладів, які навчаються за спеціальністю 126 – Інформаційні системи і технології

КИЇВ 2023

УДК 004.92

ББК 32.973

Рецензенти: С.Д. Бушуєв, доктор технічних наук, професор, завідувач катедри управління проєктами Київського національного університету будівництва і архітектури

А.О. Білощицький, доктор технічних наук, професор, проректор з науки та інновацій, Astana IT University

С.Д. Бушуєв, доктор технічних наук, професор, завідувач катедри управління проєктами Київського національного університету будівництва і архітектури

Рекомендовано Вченою радою Київського національного університету будівництва і архітектури як навчальний посібник для студентів вищих технічних навчальних закладів, які навчаються за спеціальністю 126 – Інформаційні системи і технології.

Затверджено на засіданні Вченої ради Київського національного університету будівництва і архітектури, протокол №___ від ________ 2023 р.

Бородавка Є.В.

М 80 Комп'ютерна графіка: навчальний посібник / Є.В. Бородавка, О.О. Терентьєв. Київ: КНУБА, 2023. 132 с.

В посібнику розглянуто основні поняття двовимірної та тривимірної графіки, що використовуються в системах комп'ютерного проєктування та моделювання. Навчальний посібник містить базові теоретичні та довідкові матеріали про графічну бібліотеку OpenGL, а також методичні рекомендації щодо її практичного застосування. Розглянуто приклади роботи з бібліотекою OpenGL в середовищі програмування Microsoft Visual Studio.

УДК 004.92

ББК 32.973

© Бородавка Є.В., 2023

© Терентьєв О.О., 2023

ISBN 966-627-061-7

© КНУБА, 2023

ЗМІСТ

ВСТУП 6

1. БАЗОВІ ПОНЯТТЯ ГЕОМЕТРИЧНОГО МОДЕЛЮВАННЯ 7

1.1. Різновиди обробки геометричної інформації 7
1.2. Концептуальна модель систем геометричного моделювання 8
1.3. Координатні системи в комп'ютерній графіці 9
1.4. Однорідні координати 14
1.4.1. Визначення однорідних координат 15
1.4.2. Геометрична інтерпретація однорідних координат 16
1.4.3. Точка в нескінченності 17
1.4.4. Різниця між точками та векторами 17
Запитання для самоконтролю 18

2. АФІННІ ПЕРЕТВОРЕННЯ КООРДИНАТ 19

2.1. Визначення та класифікація афінних перетворень 19
2.2. Афінні перетворення на площині 19
2.2.1. Переміщення 20
2.2.2. Масштабування 20
2.2.3. Поворот 21
2.2.4. Відображення 21
2.3. Матрична форма афінних перетворень в однорідних координатах 22
2.4. Афінні перетворення в просторі 25
2.5. Ейлерові кути 28
2.6. Кватерніони 29
Запитання для самоконтролю 30

3. МОДЕЛІ ПОДАННЯ КОЛЬОРУ 31

3.1. Моделі RGB та CMY 31
3.2. Моделі HSV/HSB та HLS/HSI 32
Запитання для самоконтролю 35

4. ФРАКТАЛИ 36

4.1. Поняття фракталу 36
4.2. Геометричні фрактали 36
4.3. Алгебраїчні фрактали 38
4.4. Система ітерованих функцій 39
Запитання для самоконтролю 42

5. ПРОЄКТИВНІ ПЕРЕТВОРЕННЯ 43

5.1. Види проєктування 43
5.2. Паралельні проєкції 44
5.2.1. Ортографічна проєкція 45
5.2.2. Аксонометрична проєкція 45

5.2.3. Косокутна проєкція _____ 46
5.3. Перспективні проєкції _____ 47
Запитання для самоконтролю _____ 49
6. МОДЕЛІ ВІДОБРАЖЕННЯ ПОЛІГОНІВ _____ 50
6.1. Плоске зафарбовування _____ 50
6.2. Інтерполяційне зафарбовування _____ 51
6.3. Зафарбовування за методом Гуро _____ 51
6.4. Зафарбовування за методом Фонга _____ 52
Запитання для самоконтролю _____ 53
7. ВИКОРИСТАННЯ GDI ДЛЯ ПОБУДОВИ ЗОБРАЖЕНЬ У WINDOWS _____ 54
7.1. Основні поняття GDI _____ 54
7.2. Функції побудови примітивів _____ 56
7.3. Створення та використання пер і пензлів _____ 58
Запитання для самоконтролю _____ 60
8. ОСНОВИ OPENGL _____ 61
8.1. Основні можливості _____ 61
8.2. Функціональна модель графічних застосунків на основі OpenGL _____ 62
8.3. Інтерфейс OpenGL _____ 63
8.4. Архітектура OpenGL _____ 64
8.5. Синтаксис команд _____ 65
8.6. Шаблон застосунку Visual Studio з використанням OpenGL _____ 66
Запитання для самоконтролю _____ 70
9. ФОРМУВАННЯ ЗОБРАЖЕНЬ ГЕОМЕТРИЧНИХ ОБ'ЄКТІВ _____ 71
9.1. Процес оновлення зображення _____ 71
9.2. Вершини і примітиви _____ 72
9.2.1. Положення вершини в просторі _____ 72
9.2.2. Колір вершини _____ 72
9.2.3. Нормаль _____ 73
9.3. Операторні дужки glBegin/glEnd _____ 73
9.4. Штрихування багатокутників _____ 76
9.5. Примітиви бібліотек GLU і GLUT _____ 78
9.6. Tess-об'єкти _____ 81
9.7. Криві та поверхні _____ 82
9.7.1. Криві та поверхні Без'є _____ 82
9.7.2. NURBS-криві та поверхні _____ 83
9.8. Дисплейні списки _____ 86
9.9. Масиви вершин _____ 87
Запитання для самоконтролю _____ 88
10. ПЕРЕТВОРЕННЯ ОБ'ЄКТІВ _____ 89
10.1. Робота з матрицями _____ 89

10.2. Модельно-видові перетворення 91
10.3. Проєкції 92
10.4. Робоча область 93
Запитання для самоконтролю 94

11. МАТЕРІАЛИ ТА ОСВІТЛЕННЯ 95

11.1. Модель освітлення 95
11.2. Специфікація матеріалів 96
11.3. Опис джерел світла 97
11.4. Створення ефекту туману 99
Запитання для самоконтролю 100

12. НАКЛАДАННЯ ТЕКСТУРИ 101

12.1. Підготовка текстури 101
12.2. Накладання текстури на об'єкти 104
12.3. Текстурні координати 105
Запитання для самоконтролю 107

13. ОПЕРАЦІЇ З ПІКСЕЛЯМИ 108

13.1. Прозорість 108
13.2. Буфер-накопичувач 110
13.3. Буфер маски 111
13.4. Керування растеризацією 112
Запитання для самоконтролю 113

14. ПРИЙОМИ РОБОТИ З OPENGL 114

14.1. Усунення ступінчастості 114
14.2. Побудова тіней 115
14.3. Дзеркальні відображення 119
Запитання для самоконтролю 121

15. ОПТИМІЗАЦІЯ ПРОГРАМ 122

15.1. Поради з підвищення надійності програм 122
15.2. Прийоми підвищення продуктивності застосунків 123
Запитання для самоконтролю 124

СПИСОК ЛІТЕРАТУРИ 125

АЛФАВІТНИЙ ПОКАЖЧИК 126

Вступ

В посібнику розглянуті базові поняття комп'ютерної графіки. Посібник структурно поділений на дві частини: в першій розглядається теоретичний матеріал, а в другій — практичне застосування розглянутих теоретичних підходів і алгоритмів.

У першому розділі розглядаються базові поняття геометричного моделювання та роль і місце комп'ютерної графіки. Розглядаються базові координатні системи, що використовуються в комп'ютерній графіці, а також математичний апарат однорідних координат та його роль в комп'ютерній графіці.

Другий розділ присвячений вивченню афінних перетворень. Розглядаються чотири базові афінні перетворення на площині та в просторі, а також їх матричні форми запису в евклідових та однорідних координатах. Додатково розглядаються Ейлерові кути та кватерніони, як альтернативні способи побудови матриці повороту в просторі.

У третьому розділі розглядаються моделі подання кольору, що найбільш часто використовуються в комп'ютерній графіці.

Четвертий розділ присвячений розгляду фракталів. Більш детально розглядаються геометричні та алгебраїчні фрактали, а також механізм системи ітерованих функцій.

В п'ятому розділі розглядаються проєктивні перетворення тривимірного простору на двовимірний. Подається класифікація типів перетворень та їх візуальні відмінності.

У шостому розділі розглядаються моделі відображення полігонів. Розглянуті методи зафарбовування Гуро та Фонга.

Сьомий розділ відкриває практичну частину посібника. В ньому розглядається програмування інтерактивної комп'ютерної графіки в середовищі Windows. Розглядаються можливості GDI для побудови та відображення двовимірних об'єктів.

Розділи з восьмого по п'ятнадцятий присвячені програмуванню комп'ютерної графіки за допомогою відкритої графічної бібліотеки OpenGL.

Присвячуємо цю роботу пам'яті наших наставників:

Демченка Віктора Вікторовича (26.03.1955–17.10.2015)

Міхайленка Віктора Мефодійовича (17.11.1937–04.02.2022).

1. БАЗОВІ ПОНЯТТЯ ГЕОМЕТРИЧНОГО МОДЕЛЮВАННЯ

В цьому розділі ми розглянемо базові поняття, що необхідні для розуміння основ комп'ютерної графіки: координатні системи та способи перетворення координат.

1.1. РІЗНОВИДИ ОБРОБКИ ГЕОМЕТРИЧНОЇ 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 (рис. 1.1).

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.

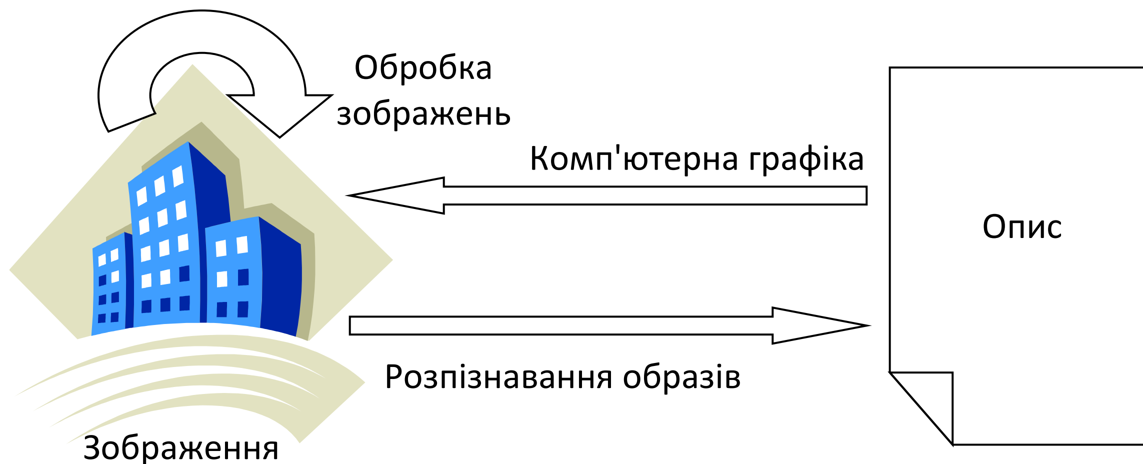


Рис. 1.1. Схема, що ілюструє взаємозв'язок комп'ютерної графіки, обробки зображень і розпізнавання образів

В цьому посібнику ми розглянемо основні задачі комп'ютерної графіки.

1.2. Концептуальна модель систем геометричного моделювання

Будь-яка комп'ютерна система комп'ютерної графіки застосовує модель перетворення геометричної інформації (рис. 1.2).

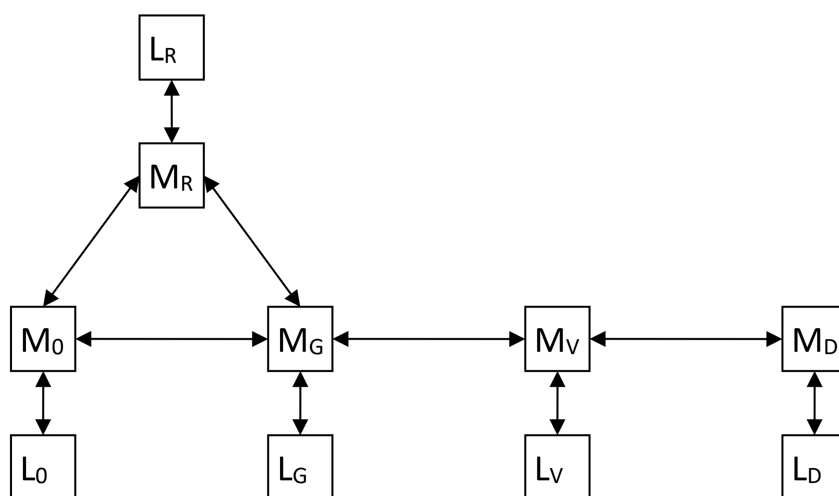


Рис 1.2. Концептуальна модель систем геометричного моделювання

Позначення на схемі:

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

M_0 — інфологічна, цифрова модель об'єкта (ЦМО). Відображає структуру, склад та зв'язки об'єкта проєктування 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 скінченними елементами для проведення розрахунків за відповідним методом. Таким чином модель M_0 одержить від геометричної моделі на цьому кроці розрахункову схему.

Під час відображення 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 під час інтерактивного введення 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 в комп'ютерній графіці

В 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 незалежні від пристрою і мають одиниці виміру, що притаманні тим світам, для опису яких вони призначені (міліметри — для об'єктів проєктування, 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 *піксель* в області виведення 1280×1024, 1280×720, 1600×900, 1920×1200 тощо.

Піксель — це найменша частка зображення на екрані дисплею, яка характеризується положенням і кольором (рис. 1.3). Координати інших пристроїв можуть мати метричну характеристику або крок дискретної сітки робочого поля з характеристикою *dpi* (точок на дюйм).

Рис. 1.3. Піксельна решітка на екрані LCD монітора

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:

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

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

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

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:

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

Розрахунки коефіцієнту та координат центральних точок світового вікна і області індикації необхідно виконувати лише один раз під час зміни 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. 1.4).

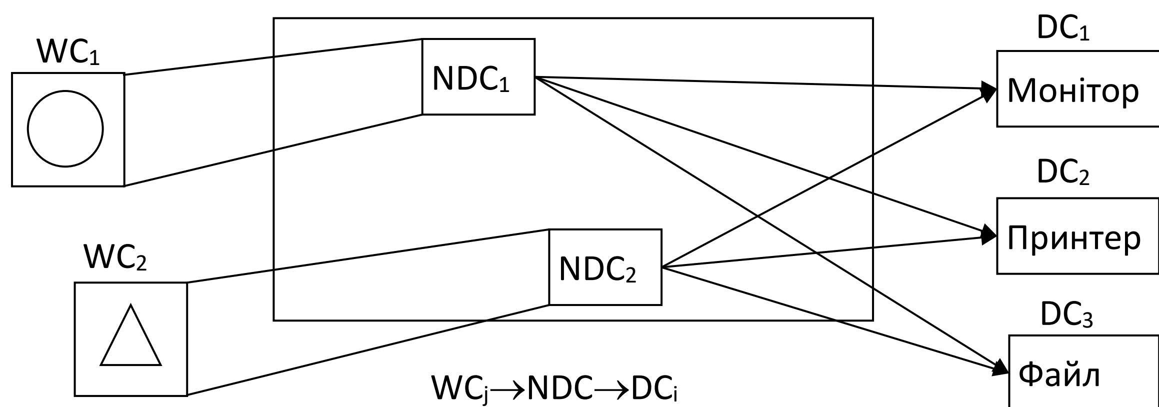


Рис. 1.4. Схема перетворення світових координат у приладові

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

Окрім розглянутих світової та приладової систем координат в 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 комп'ютерній 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. 1.5).

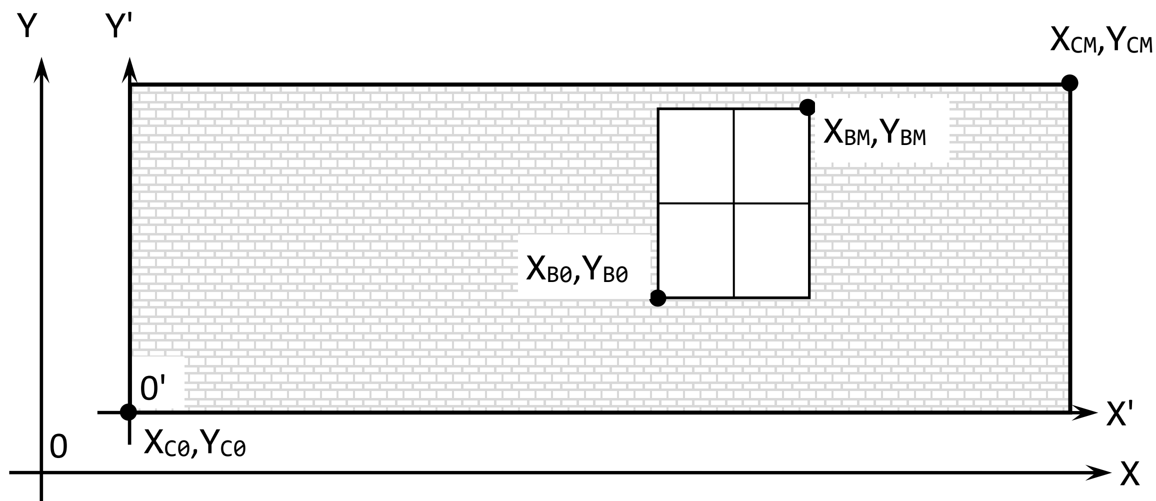


Рис. 1.5. Фасад стіни з вікном

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_{C0},Y_{C0},X_{CM},Y_{CM}), а для вікна – (X_{B0},Y_{B0},X_{BM},Y_{BM}). Наведені координати описують обидва об'єкти (стіну і вікно) в системі координат з початком XOY 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'O'Y' з початком в точці (X_{C0},Y_{C0}), 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'_{CM},Y'_{CM}), а вікно – (X'_{B0},Y'_{B0},X'_{BM},Y'_{BM}). В наведеному 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 (проєкції) 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. Якщо зменшувати 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). Коли $w=1$ ці координати описують точку із скінченними координатами (x,y), а якщо $w=0$ — 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 — степінь 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:

$$Ax + By + Cw = 0. \tag{1.4}$$

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:

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

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 заданого 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 правильними 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 однозначне: 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 операцій утворюють так званий *проєктивний 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 задана точка на площині XOY з однорідними координатами (x,y,w). 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. 1.6).

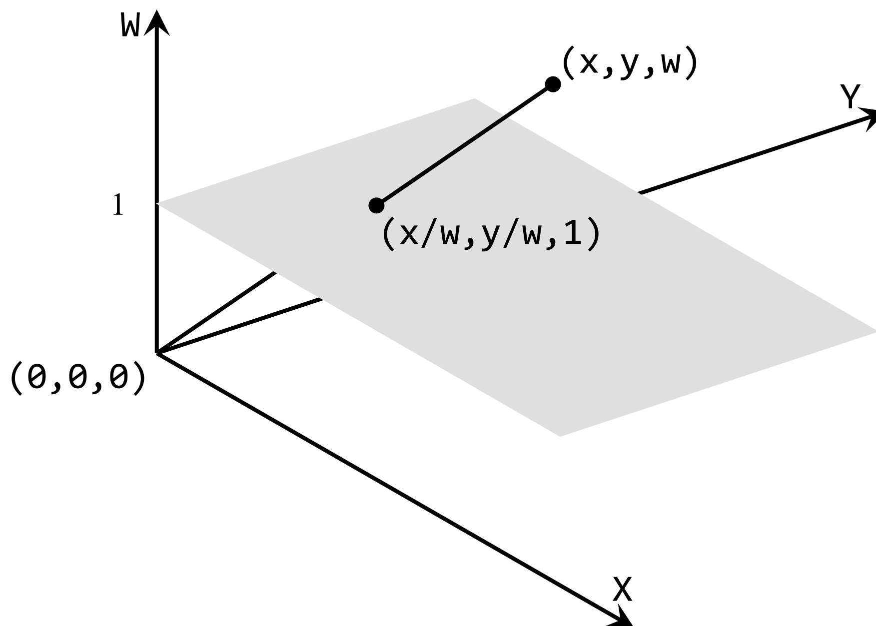


Рис. 1.6. Геометрична інтерпретація однорідних координат

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

З рисунка 1.6 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 проєктуватися 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) піде 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 цьому 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

В цьому розділі ми розглянемо визначення афінних перетворень та їх основні типи. Будуть наведені формули афінних перетворень та їх матричні подання.

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 біекція 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 координати. 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 М 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 М(x,y) в точку М'(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).

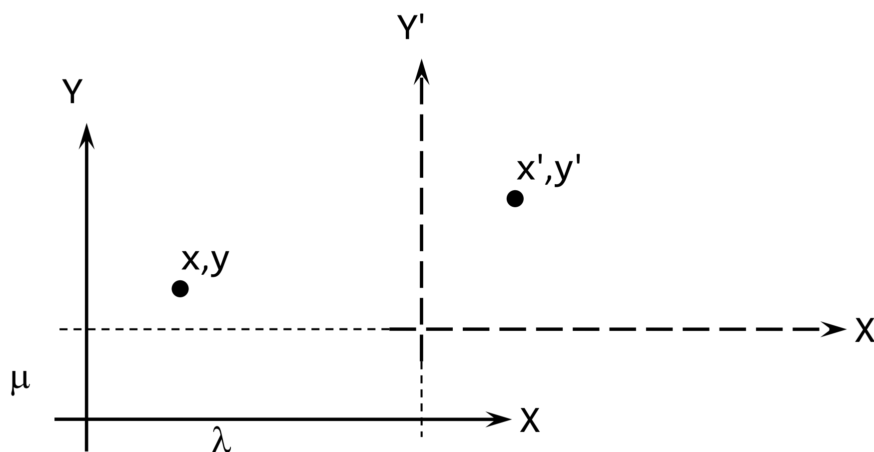


Рис. 2.1. Переміщення системи координат

Перетворення координат точки в новій системі координат в координати точки в початковій 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:

$$x' = \alpha x;\ y' = \delta y, \qquad (2.3)$$

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).

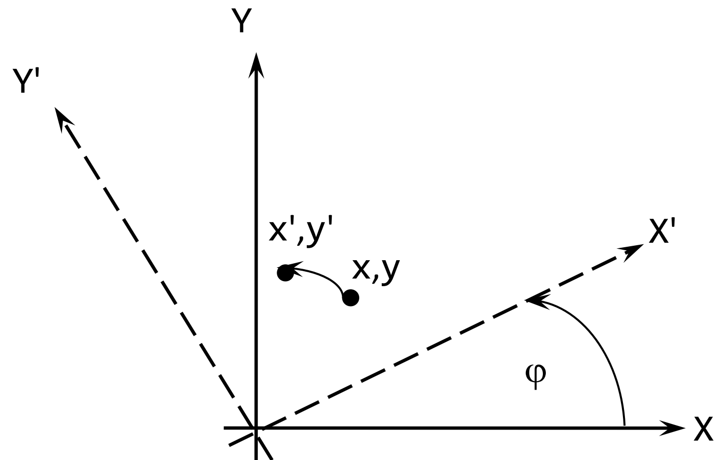


Рис. 2.2. Поворот системи координат

Поворот навколо початкової точки на кут φ описується наступними формулами:

$$x' = x\cos\varphi - y\sin\varphi\,;\; y' = x\sin\varphi + y\cos\varphi. \qquad (2.4)$$

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).

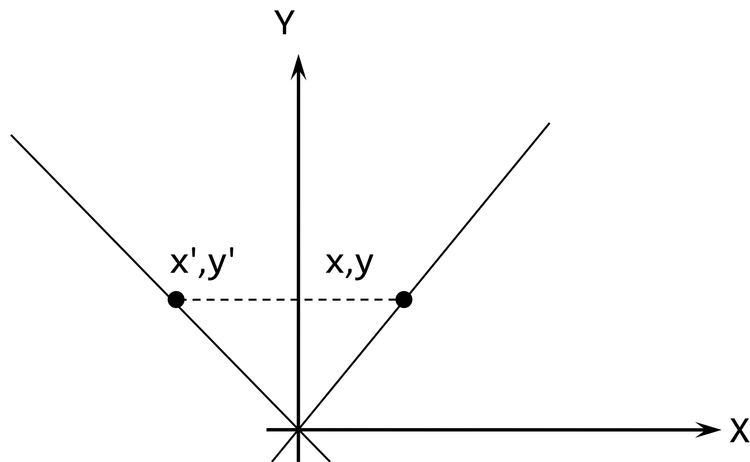


Рис. 2.3. Відображення відносно осі Y

Відображення відносно осі Y задається наступними формулами:

$$x' = -x;\ y' = y. \tag{2.5}$$

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 для складного афінного перетворення 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: $l^2+m^2+n^2=1$.

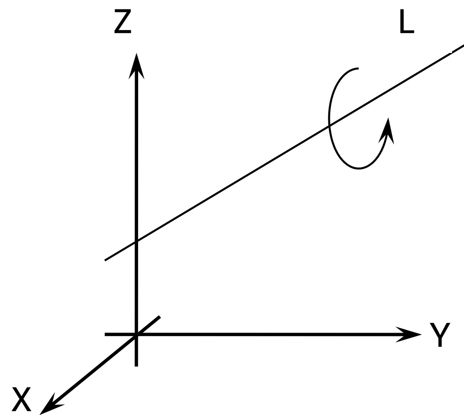


Рис. 2.4. Умова задачі для прикладу

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 проєкцію L' заданої прямої L на площину X=0 (рис. 2.5).

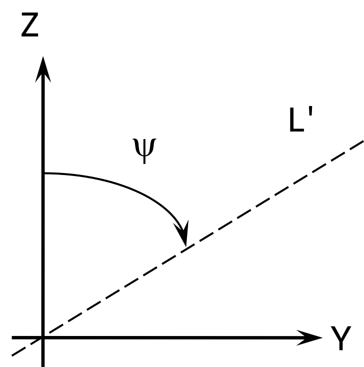


Рис. 2.5. Проєкція прямої L на площину X=0

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 *Ейлеревих 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 це система спостерігача, а інша жорстко зв'язується з тілом.

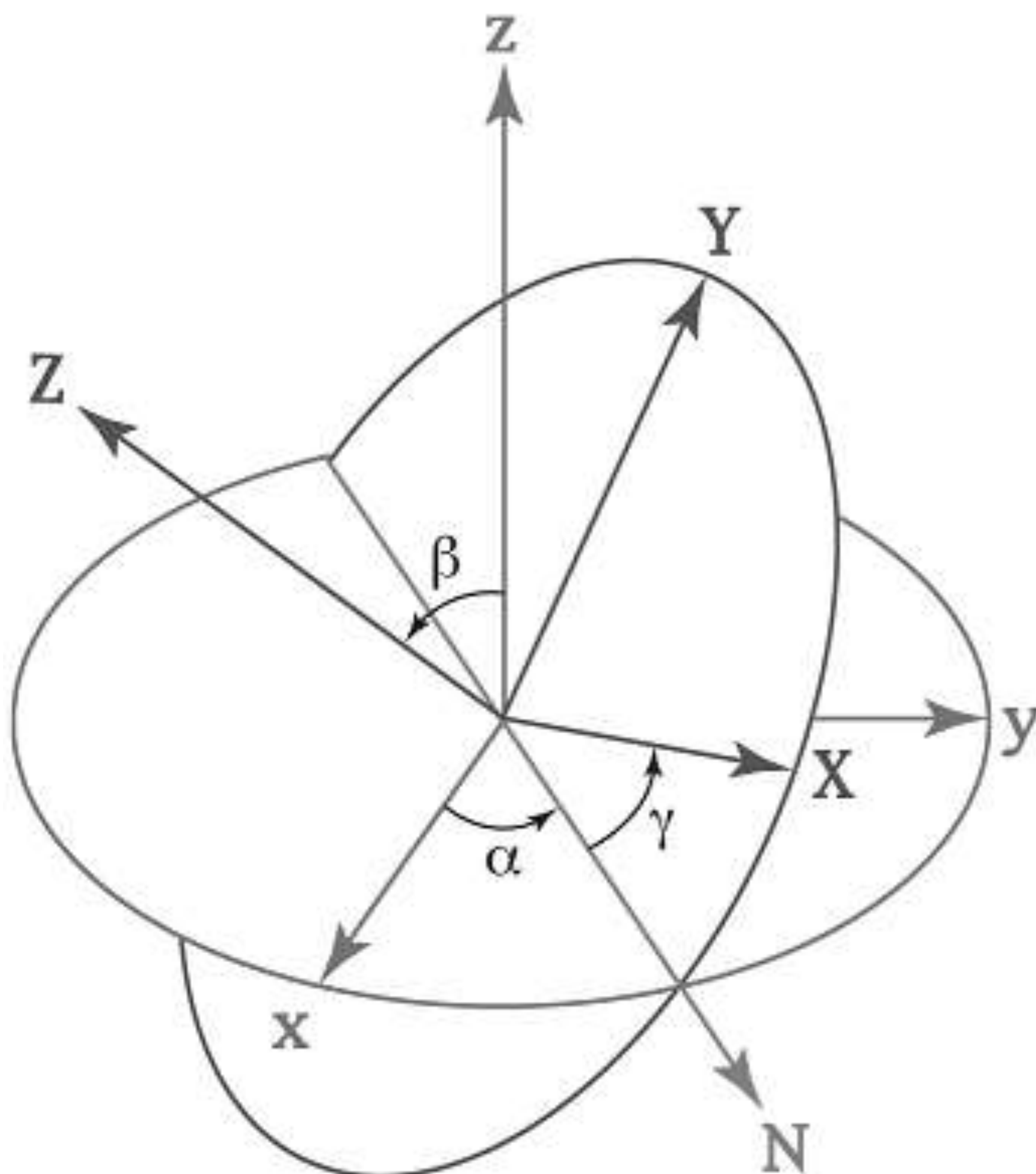


Рис. 2.6. Ейлерові кути

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 Ейлеревих 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:

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

Для застосування Ейлеревих кутів достатньо їх вирахувати і скласти з них 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 тривимірного вектора: q=(s,v), s=a, v=(a,b,c).

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

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

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 тривимірних 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:

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

Фактично вираз (2.7) аналогічний матриці, що ми отримали в результаті вирішення задачі з підрозділу 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. Моделі подання кольору

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

3.1. 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. 3.1). Головна діагональ куба, що характеризується рівним внеском трьох базових кольорів, подає сірі кольори: від чорного (0, 0, 0) до білого (1, 1, 1).

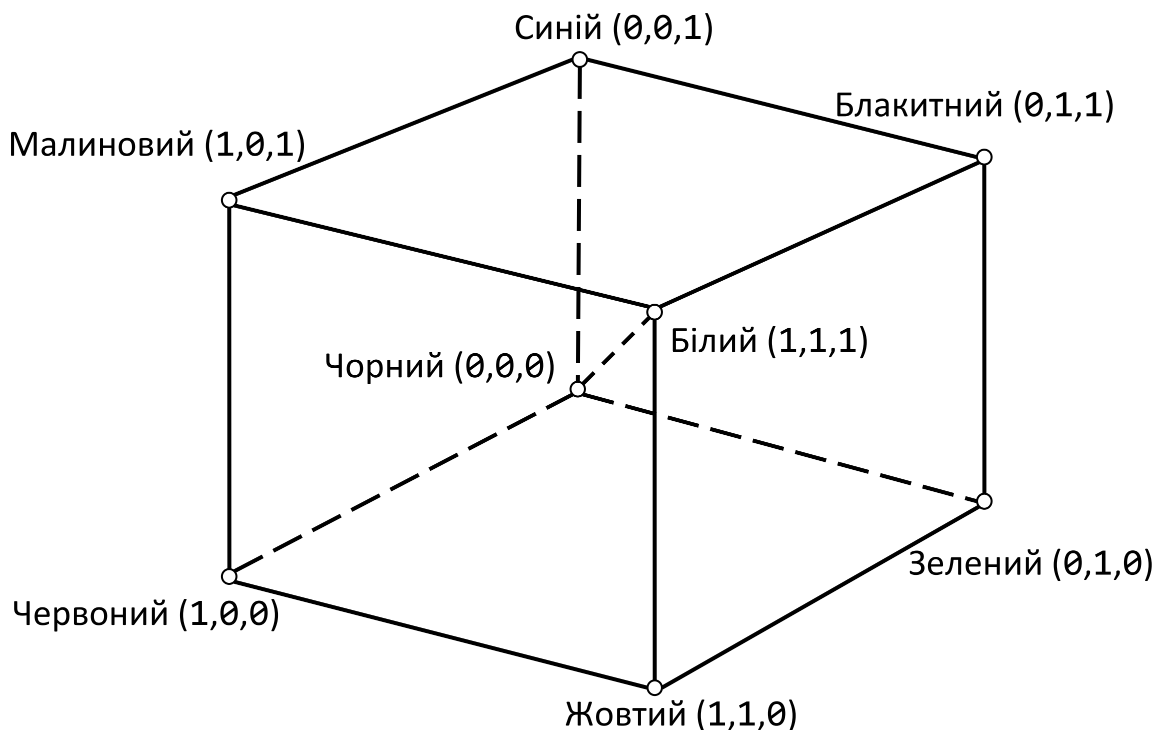


Рис. 3.1. Система RGB

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:

$$\begin{pmatrix} C \\ M \\ Y \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} - \begin{pmatrix} R \\ G \\ B \end{pmatrix}. \quad (3.1)$$

Зворотні перетворення виконуються за формулою:

$$\begin{pmatrix} R \\ G \\ B \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} - \begin{pmatrix} C \\ M \\ Y \end{pmatrix}. \quad (3.2)$$

З цілого ряду причин (велика витрата дорогого кольорового чорнила, висока якість паперу, одержувана під час друку на струменевих принтерах, небажані візуальні ефекти, що виникають за рахунок того, що під час виведення 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*, *blackK*). Для переходу від моделі CMY до моделі CMYK використовують наступні співвідношення:

$$\begin{aligned} K &= \min(C, M, Y); \\ C &= C - K; \\ M &= M - K; \\ Y &= Y - K. \end{aligned} \quad (3.3)$$

Співвідношення (3.1)-(3.3) правильні 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.

3.2. 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* та *яскравості*. У цій моделі використовується циліндрична система координат, а множина всіх допустимих кольорів є шестигранним конусом, що стоїть на вершині (рис. 3.2).

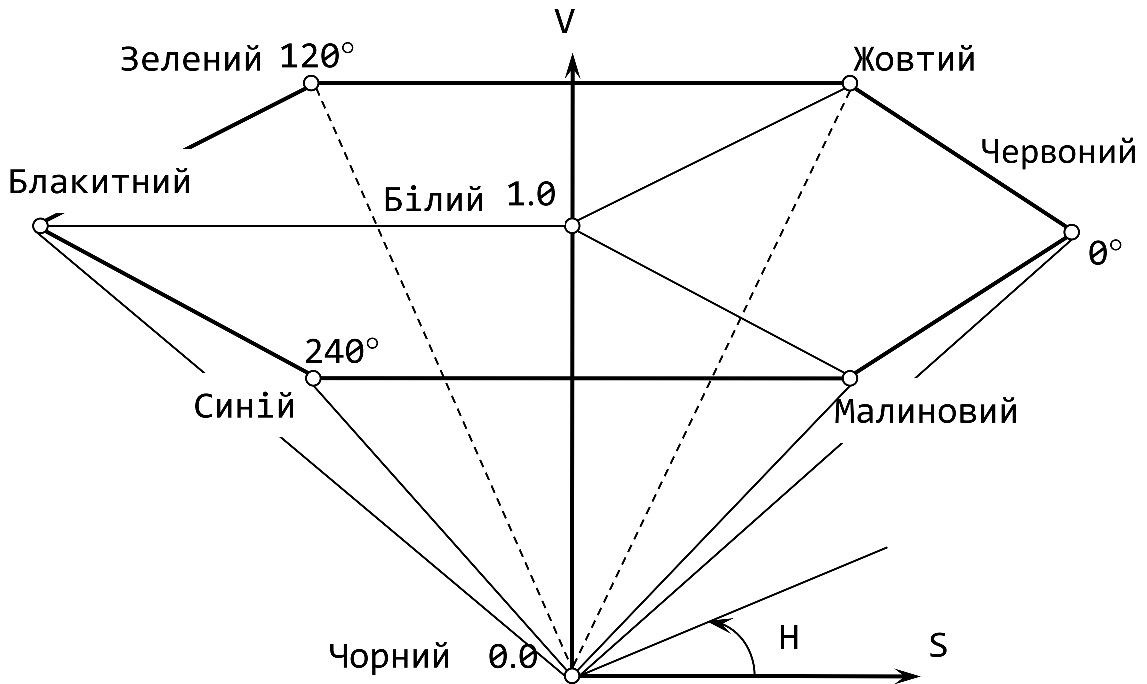


Рис. 3.2. Модель HSV/HSB

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:

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

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

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

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 (рис. 3.3), причому вершина нижнього конуса співпадає з початком координат. Тон як і раніше задається кутом, що відкладається від вертикальної осі з червоним кольором.

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 зсунутий вгору, щоб сформувати верхній конус з площини V=1.

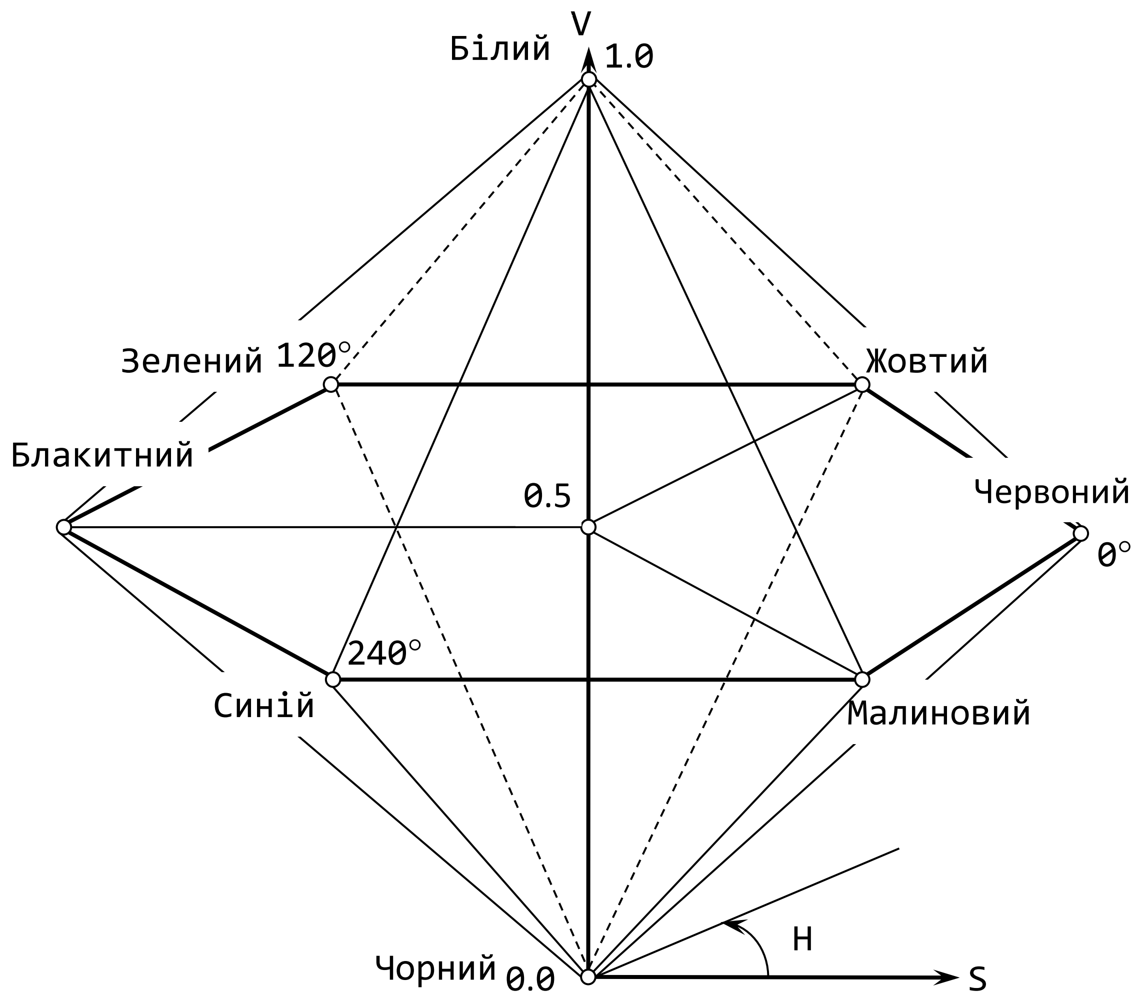


Рис. 3.3. Модель HLS/HSI

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

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

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

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

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

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

$$P = 2S - Q;$$

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

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

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

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?

4. Фрактали

В цьому розділі ми розглянемо поняття фрактальної геометрії та найбільш відомі фрактальні зображення.

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, Гаусдорф). 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 хмар, гір, поверхні моря. Фактично знайдений спосіб легкого подання складних неевклідових об'єктів, обриси 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 деякому 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. В такому випадку 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 нескінченного 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).

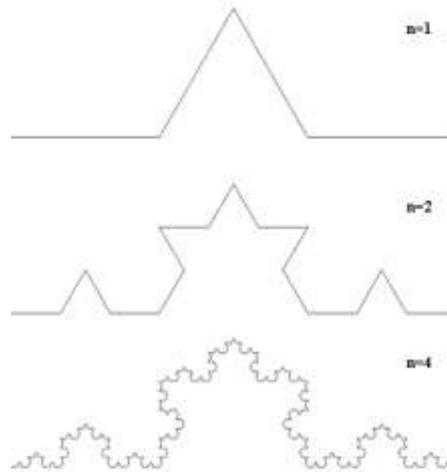


Рис. 4.1. Побудова тріадної кривої Коха

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 для будь-якого скінченного n називається *предфракталом*. На рис. 4.1 показані перше, друге і четверте покоління кривої. Для n, що прямує до нескінченності, крива Кох стає фрактальним об'єктом.

Для отримання іншого фрактального об'єкта треба змінити правила побудови. Нехай утворюючим елементом будуть два рівних відрізки, 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 під час заміни 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 (для n, що прямує до нескінченності) називається *драконом Хартера-Хейтуея*.

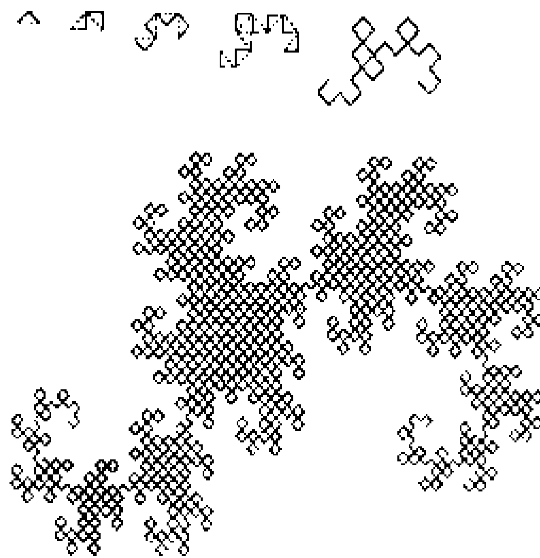


Рис. 4.2. Побудова дракона Хартера-Хейтуея

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

4.3. Алгебраїчні фрактали

Це найбільша 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 потрапить 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.

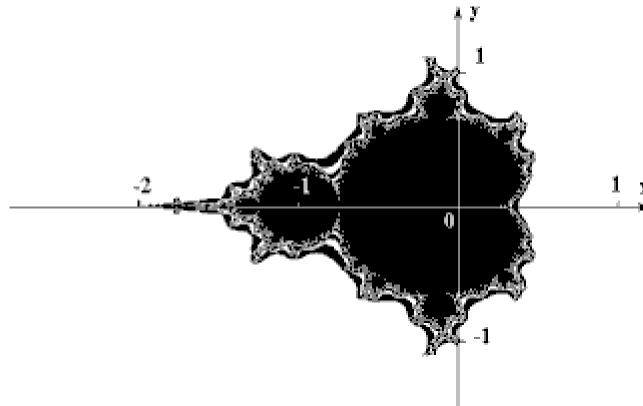


Рис. 4.3. Множина Мандельброта

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 прямують в нескінченність (точки мають чорний колір). Точки, що належать межі множини (саме там виникають складні структури) прямують в нескінченність за скінченне число ітерацій, а точки, що лежать за межами множини, прямують 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, проєктивні:

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

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 ламаної, 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:

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

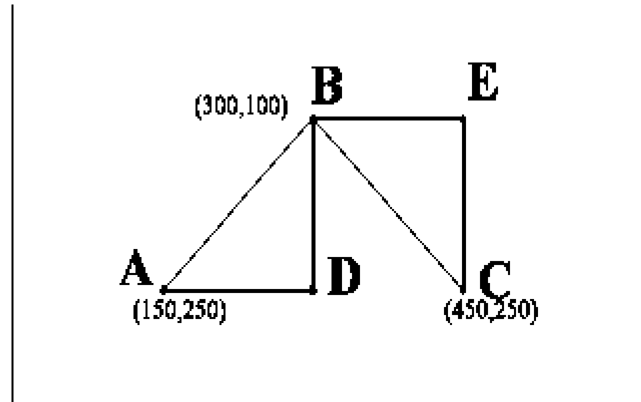


Рис. 4.5. Заготовка для побудови IFS дракона Хартера-Хейтуея

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.

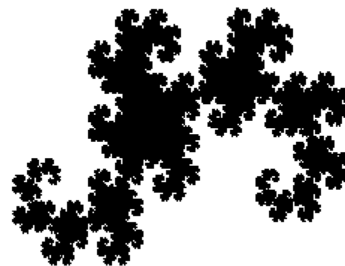


Рис. 4.6. Дракон Хартера-Хейтуея, побудований з допомогою IFS у прямокутнику 640×350

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).

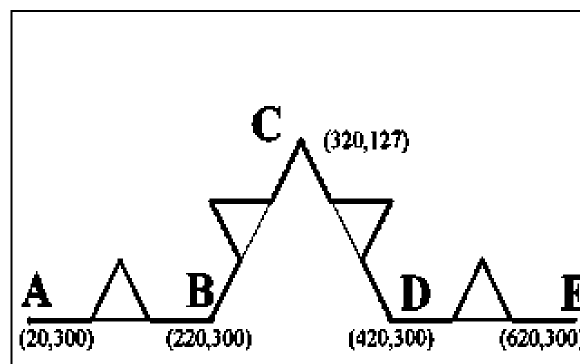


Рис. 4.7. Заготовка для побудови IFS кривої Коха

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

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

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

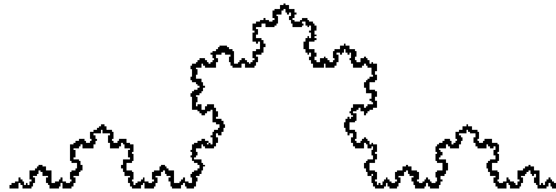


Рис. 4.8. Крива Коха, побудована за допомогою IFS у прямокутнику 640×350

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). За алгоритмом 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. Водночас у 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. ПРОЄКТИВНІ ПЕРЕТВОРЕННЯ

Під час 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 проєктування об'єму видимості на область індикації, яку ще називають *картинна площина*. Модель процесу візуалізації наведена на рис. 5.1.

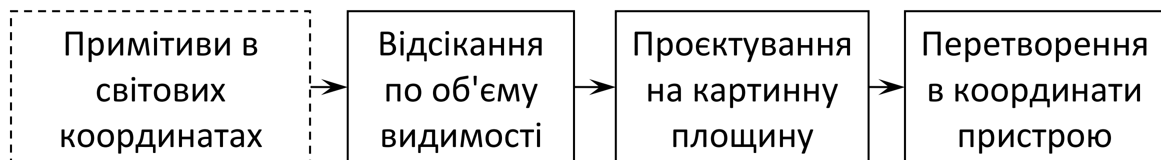


Рис. 5.1. Модель процесу візуалізації

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

5.1. ВИДИ ПРОЄКТУВАННЯ

Проєктування виконується за допомогою прямолінійних проєкторів (променів проєктування), що йдуть з центру проєкції через кожну точку об'єкта до перетину з картинної поверхнею (поверхнею проєкції). Далі розглядаються тільки плоскі проєкції, для яких поверхня проєкції — площина в тривимірному просторі.

За розташуванням центру проєктування відносно площини проєкції розрізняють *центральне* та *паралельне* проєктування.

Центральне проєктування — проєктування, під час якого всі прямі виходять з однієї точки — центра власного пучка (рис. 5.2).

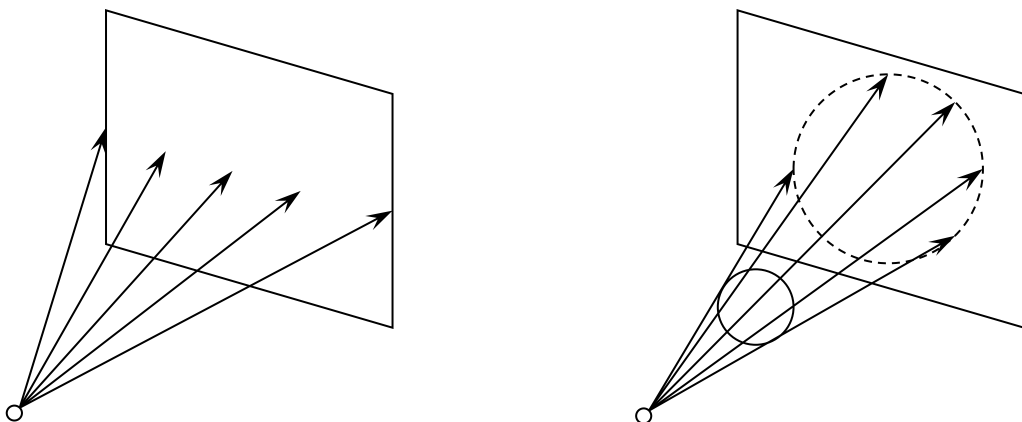


Рис. 5.2. Центральне проєктування

Паралельне проєктування — проєктування, під час якого центр пучка вважається таким, що лежить в нескінченності (рис. 5.3).

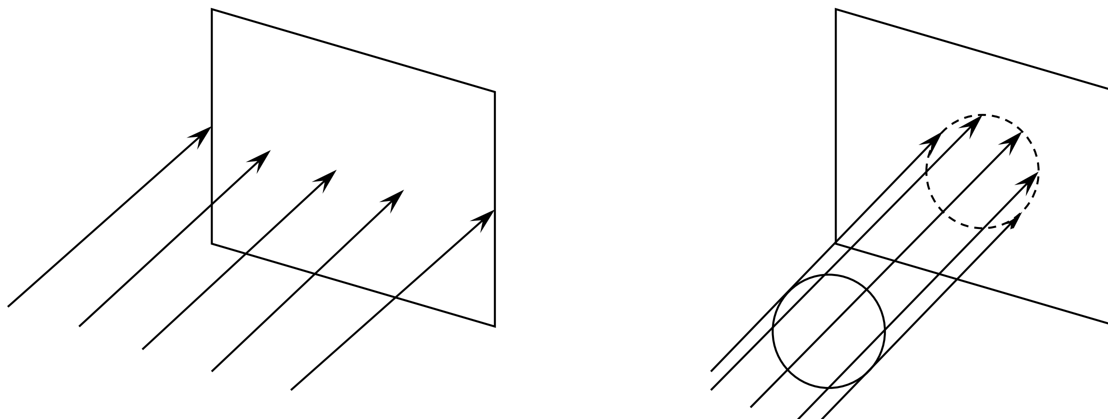


Рис. 5.3. Паралельне проєктування

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

5.2. Паралельні проєкції

В паралельних проєкціях виокремлюють три класи: ортографічна, аксонометрична та косокутна. Дві останні з них ще додатково поділяються на підкласи (рис. 5.4).

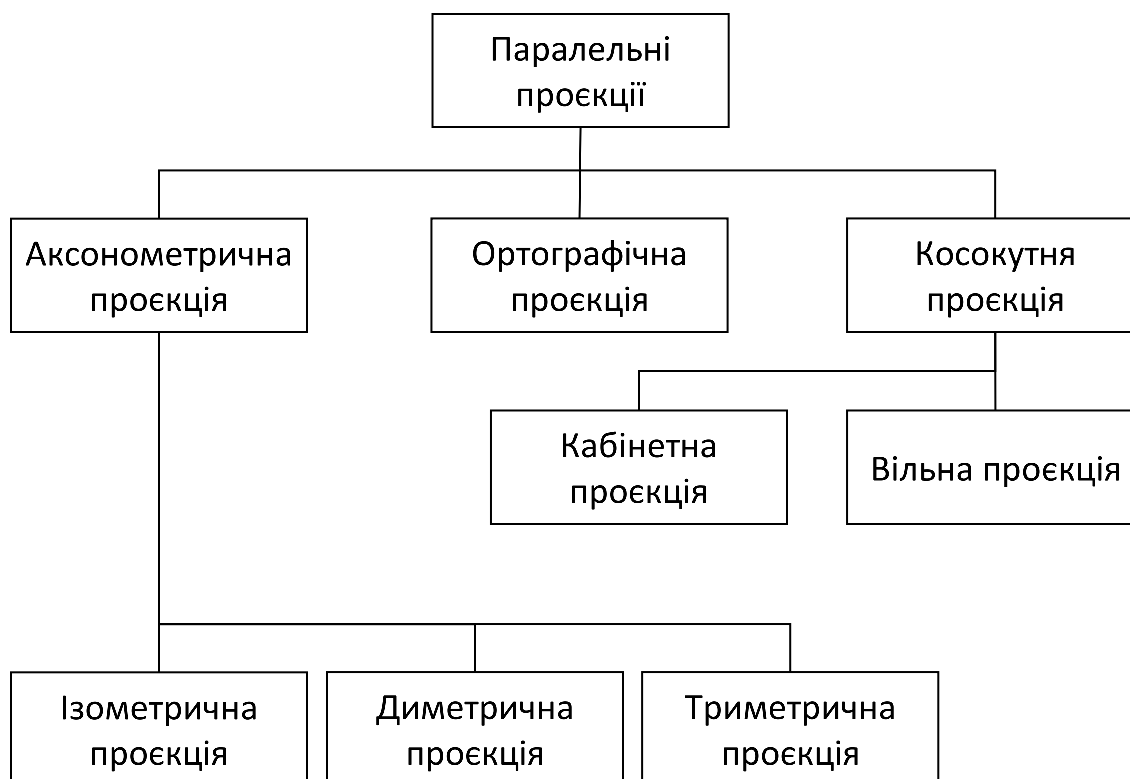


Рис. 5.4. Класифікація паралельних проєкцій

Тепер розглянемо кожен клас паралельних проєкцій більш детально.

5.2.1. Ортографічна проєкція

Під час *ортографічної* проєкції картинна площина співпадає з однією з координатних площин або паралельна їй (рис. 5.5).

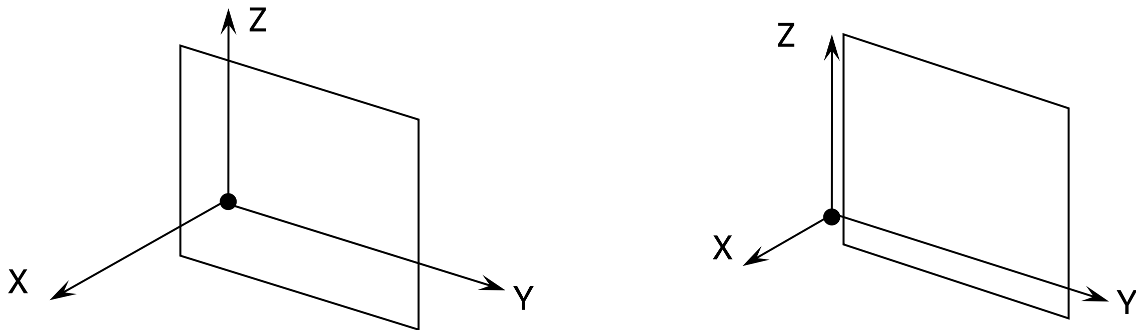


Рис. 5.5. Ортографічна проєкція на площину ZOY

Матриця проєктування вздовж осі 0X на площину ZOY 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 проєктування 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 проєктування 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 проєктування для ортографічної проєкції — *вироджені*, тобто їх визначник дорівнює 0.

5.2.2. Аксонометрична проєкція

Під час *аксонометричної* проєкції прямі проєктування перпендикулярні до картинної площини.

У відповідності до взаємного розташування площини проєктування і координатних осей розрізняють три види проєкцій:

- *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 проєкцій отримується за допомогою комбінації поворотів, після якої виконується паралельне проєктування. Якщо виконати поворот на кут ψ відносно осі ординат, потім поворот на кут φ навколо осі абсцис і наступне проєктування 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 проєкцій 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:

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

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

З попередніх формул випливає що:

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

Для триметрії довжини проєкцій попарно різні.

5.2.3. Косокутна проєкція

Проєкції, для отримання яких використовується пучок прямих, не перпендикулярних площині екрана, називають *косокутними*. Під час косокутного проєктування орта осі Z на площину XOY (рис. 5.6) отримуємо:

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

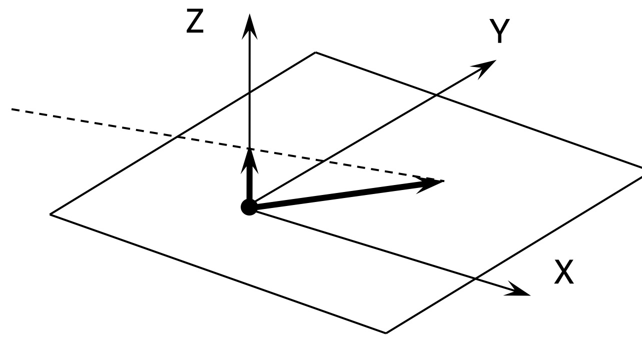


Рис. 5.6. Косокутна проєкція

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 проєкцій:

- *вільна* — кут нахилу прямих проєктування до площини екрана рівний половині прямого:

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

- *кабінетна* — частковий випадок вільної, коли масштаб по третій вісі вдвічі менший:

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

5.3. Перспективні проєкції

Найбільш реалістично тривимірні об'єкти виглядають в центральній проєкції через перспективні деформації сцени. Центральні проєкції паралельних прямих, які не паралельні площині проєктування будуть сходитися в точці *сходу*. В залежності від кількості точок сходу, тобто від числа координатних осей, що перетинають площину проєкції, розрізняються наступні перспективні проєкції: одноточкова, двоточкова і триточкова (рис. 5.7).

Рис. 5.7. Перспективні проєкції

Розглянемо спочатку одноточкову проєкцію. Припустимо, що центр проєктування лежить на осі Z в точці C(0,0,c) і площина проєктування співпадає з координатною площиною XOY (рис. 5.8).

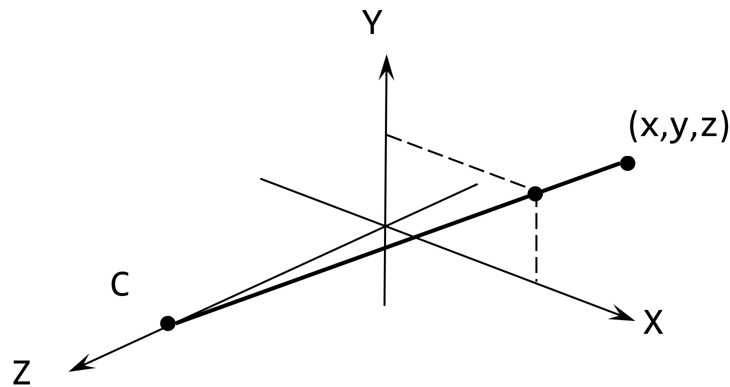


Рис. 5.8. Перспективна проєкція

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 проєктування 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 проєкціях з різною кількістю точок сходу наведені на рис. 5.9.

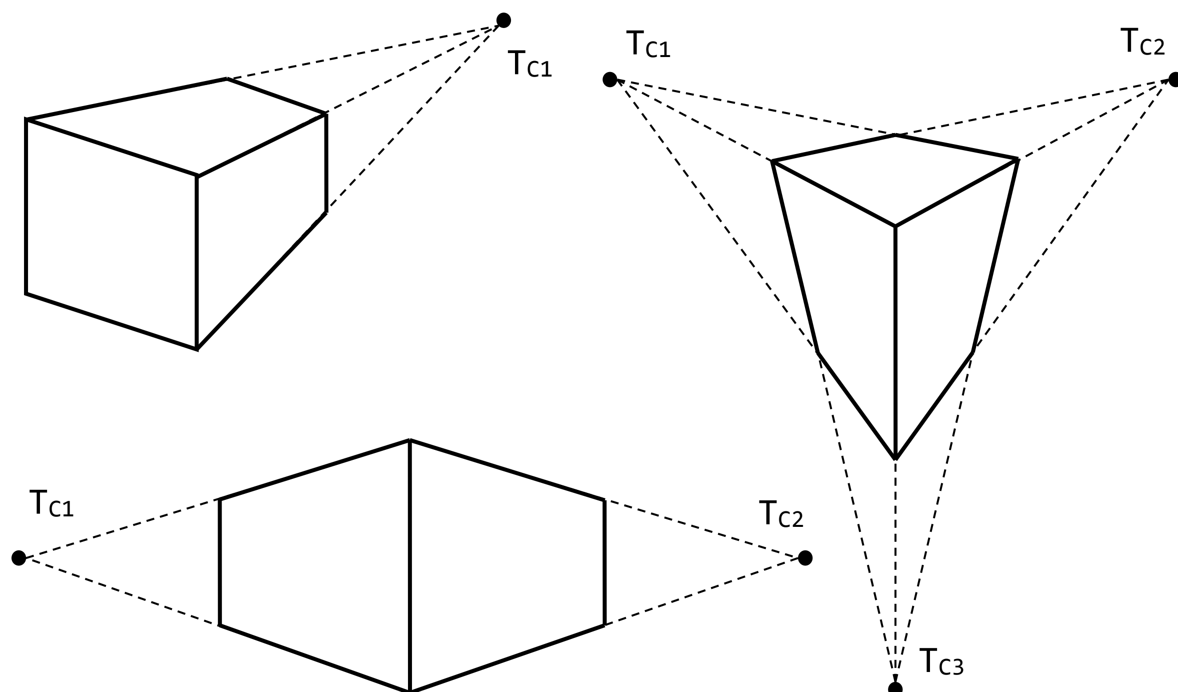


Рис. 5.9. Зображення кубу в перспективних проєкціях

ЗАПИТАННЯ ДЛЯ САМОКОНТРОЛЮ

1. Що таке проєктивне перетворення?
2. Які два основні класи проєктивних перетворень розрізняють?
3. Які основні підкласи паралельної проєкції?
4. Які основні підкласи перспективної проєкції?

6. Моделі відображення полігонів

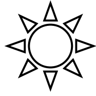
Маючи в своєму розпорядженні засоби обчислення векторів нормалі, можна для заданого розташування джерел світла і спостерігача застосувати розглянуті моделі до всіх точок поверхонь об'єктів сцени. Але, на жаль, використання рівнянь обчислення нормалі, стосовно сферичної поверхні, приводить до неприйнятно 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).

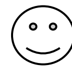
Розглянемо полігональну мережу. Кожен багатокутник в такій мережі — плоский, і обчислити компоненти вектора нормалі до нього досить легко. Нижче ми розглянемо чотири методи зафарбовування багатокутників: плоске, інтерполяційне, зафарбовування за методом Гуро (*Gouraud*), і зафарбовування за методом Фонга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

6.1. Плоске зафарбовування

Під час переміщення від однієї точки на поверхні до іншої в загальному випадку можуть змінюватися три вектори — 1, 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 1 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 в цьому випадку потрібно 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. 6.1). 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).*





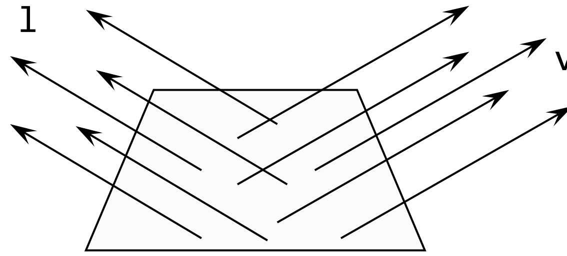


Рис. 6.1. Віддалене джерело світла та спостерігач

У OpenGL режим плоского зафарбовування задається аргументом `GL_FLAT` під час виклику функції `glShadeModel()`.

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

6.2. 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. Під час обчислення 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 під час використання 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.

6.3. 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. 6.2).

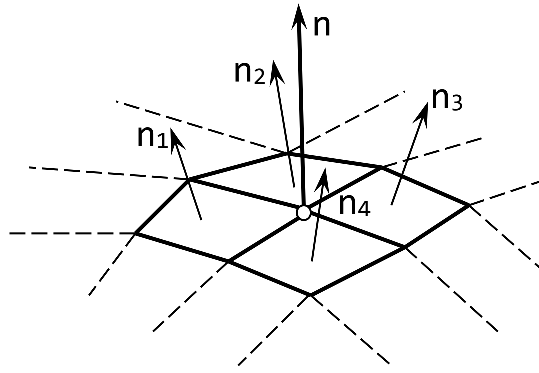


Рис. 6.2. Полігональна мережа з векторами нормалей

Кожен багатокутник має свій вектор нормалі. *Метод зафарбовування Гуро* полягає в тому, що з вершинами зв'язуються нормалі, які отримують в результаті усереднювання нормалей багатокутників, що перетинаються в цій вершині. Для прикладу, що поданий на рисунку 6.2, 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.

6.4. Зафарбовування за методом Фонга

Але навіть використання методу Гуро у ряді випадків не дозволяє 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. Водночас 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. 6.3) .

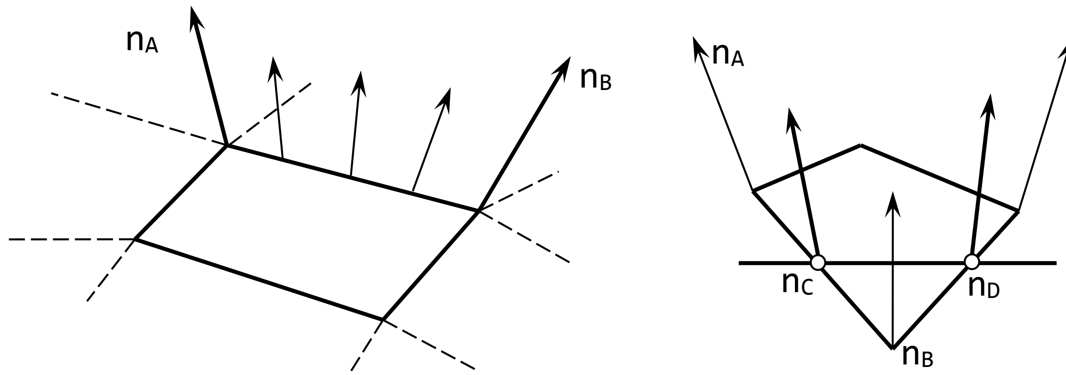


Рис. 6.3. Інтерполяція векторів нормалей вздовж ребра

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?

7. Використання GDI для побудови зображень у Windows

У середовищі Windows доступ до графічного інструментарію здійснюється за допомогою графічного інтерфейсу, відомого як *GDI* (*Graphic Device Interface*). В традиційному програмуванні для середовища Windows, програмісти працюють безпосередньо з функціями та інструментами GDI. Такий підхід зберігається зокрема в середовищі Microsoft Visual C++. Деякі інші середовища програмування інкапсулюють стандартні функції GDI в класах. Наприклад Embarcadero C++ та Delphi мають клас *TCanvas*, який інкапсулює та спрощує використання GDI-функцій, засобів та методів. В цьому розділі ми розглянемо стандартний підхід до використання GDI на прикладах у Microsoft Visual C++.

7.1. Основні поняття GDI

Фактично GDI є бібліотекою, яка поставляється разом з операційною системою Windows і яка надає розробникам *API* (*Application Programing Interface*) з доступом до функцій бібліотеки. Повний перелік функцій та констант бібліотеки можна знайти в заголовному файлі *wingdi.h*.

Функції GDI здійснюють виведення зображення не на конкретний пристрій напряму (монітор, принтер, вікно), а на контекст пристрою *DC* (device context). Контекст пристрою можна розглядати як віртуальний пристрій, на який здійснюється виведення зображення, а перетворення його на реальне зображення конкретного пристрою забезпечує операційна система Windows. Контекст пристрою це фактично структура даних, що містить набір характеристик цього пристрою: бітові карти зображення (*bitmap*) або растр, перо (*pen*), пензель (*brush*), шрифт (*font*). Все, що потрібно від програміста, це отримати посилання (*Handle*) на контекст необхідного пристрою і працювати з ним незалежно від того, яким насправді є фізичний пристрій.

Побудова будь-яких зображень за допомогою GDI виконується на етапі перемалювання об'єкта на який здійснюється виведення. Зазвичай виведення здійснюється на вікно, тому потрібно перехоплювати обробник події перемалювання вікна і туди додавати свій код. В середовищі Microsoft Visual C++ після створення проєкту автоматично створюється функція *WndProc*, яка перехоплює повідомлення вікну. За замовчуванням там вже створений розділ `WM_PAINT` для випадку перемалювання вікна. В ньому вже доданий код для отримання контексту присторою:

```
case WM_PAINT:
{
  PAINTSTRUCT ps;
  HDC hdc = BeginPaint(hWnd, &ps);
    // TODO: Add any drawing code that uses hdc here...
  EndPaint(hWnd, &ps);
}
```

Перша команда BeginPaint отримує змінну типу HWND, тобто посилання на ідентифікатор вікна в системі та створює і повертає посилання на контекст пристрою, що зв'язаний з цим вікном. Також ця команда заповнює структуру PAINTSTRUCT, де окрім посилання на контекст пристрою міститься інформація про розмір області малювання та деякі системні зарезервовані поля. Після отримання посилання на контекст пристрою, ми можемо будувати зображення на ньому за допомогою функцій GDI. Після завершення всіх побудов викликається функція EndPaint яка звільняє контекст пристрою і звільняє пам'ять виділену для нього.

Також можна використати іншу пару команд, які описані у заголовному файлі *winuser.h*:

```
HDC GetDC (HWND hWnd)
int ReleaseDC (HWND hWnd, HDC hdc).
```

Фактично вони роблять те саме, що і пара попередніх функцій.

Побудова зображень за допомогою GDI здійснюється шляхом зміни кольору окремого пікселя. Для цього використовується функція

```
SetPixel (HDC hdc, int x, int y, unsigned long color)
```

де hdc — це посилання на контекст пристрою; x, y — координати пікселя; color — колір пікселя (червоний колір займає 0-й байт, зелений колір — 1-й байт, а синій — 2-й байт). Для простішого задання кольору можна використовувати макрос RGB(r,g,b) який перетворює три окремі компоненти кольору (кожен по 1 байту) на чотирибайтний код. Третій, старший байт в коді, відведений під збереження альфа-компоненти (ступеня прозорості) кольору.

Для отримання значення кольору пікселя використовується зворотна функція

```
unsigned long GetPixel (HDC hdc, int x, int y)
```

яка повертає чотирибайтний код кольору пікселя за вказаними координатами.

В загальному випадку, все зображення можна побудувати шляхом задання кольору кожному пікселю, що воно використовує. Але такий підхід буде занадто важким для реалізації. Тому в GDI використовуються графічні об'єкти які полегшують побудову зображень. Основні з них такі:

- перо (*pen*) — задає параметри побудови контурів об'єктів. Параметрами пера є товщина (*width*), колір (color), тип лінії (*style*) — суцільна (*solid*), точкова (*dot*), штрихова (*dash*), штрихпунктирна (*dashdot*);
- пензель (*brush*) — задає параметри зафарбовування об'єктів. Перед використанням пензель повинен бути створений як об'єкт. Основними параметрами пензля є колір (*color*), тип (*style*) — суцільне зафарбовування (*solid*), без зафарбовування (*null* або *hollow*), зафарбовування паттернами (*pattern*) або штрихуванням лініями (*hatched*). В свою чергу штрихування може бути горизонтальним (*horizontal*), вертикальним (*vertical*), діагональним

(*fdiagonal* та *bdiagonal*), хрестоподібним (*cross*) та діагональним хрестоподібним (*diagcross*);

- растр (*bitmap*) — набір байтів, що містить значення кольорів та інформацію про координати для відображення на пристрої пікселів, які в сукупності становлять зображення. Растрові зображення можуть створюватися як всередині програми так і завантажуватися із зовнішніх файлів;
- шрифт (*font*) — це набір пікселів для матричних шрифтів або набір кривих, що описують контури букв для векторних. Перед використанням шрифти можуть створюватися програмно або використовуватися заздалегідь підготовлені та встановлені в системі шрифти.

Окрім вказаних графічних об'єктів, бібліотека GDI містить функції для побудови векторних примітивів за їх координатами.

7.2. Функції побудови примітивів

Бібліотека GDI має вбудовані функції для побудови більшості простих двовимірних об'єктів. Основні з них ми розглянемо в цьому параграфі.

Побудова об'єктів здійснюється з поточними налаштуваннями пера та пензля. Якщо ніякого налаштування не виконано, то за замовчанням використовується перо одиничної товщини чорного кольору, а пензель працює в режимі без зафарбовування. Всі функції в якості першого параметру використовують посилання на контекст пристрою (`hdc`).

Для побудови відрізка необхідно спочатку перемістити поточну позицію курсора в точку початку, а потім викликати функцію побудови відрізка з поточної точки до вказаної:

```
BOOL MoveToEx (HDC hdc, int x, int y, LPPOINT lppt)
BOOL LineTo (HDC hdc, int x, int y);
```

де `x`, `y` — координати пікселя; `lppt` — покажчк на структуру `POINT` куди записується попередня позиція (може бути `nullptr`).

Для побудови прямокутника використовується функція

```
BOOL Rectangle (HDC hdc, int left, int top, int right, int bottom)
```

де `left` — координата X лівої сторони прямокутника; `top` — координата Y верхньої сторони прямокутника; `right` — координата X правої сторони прямокутника; `bottom` — координата Y нижньої сторони прямокутника.

Для побудови еліпса, що вписаний в прямокутник, використовується функція

```
BOOL Ellipse (HDC hdc, int left, int top, int right, int bottom)
```

де `left` — координата X лівої сторони прямокутника; `top` — координата Y верхньої сторони прямокутника; `right` — координата X правої сторони прямокутника; `bottom` — координата Y нижньої сторони прямокутника.

Для побудови прямокутника зі округленими кутами використовується функція

```
BOOL RoundRect (HDC hdc, int left, int top, int right, int bottom, int
width, int height)
```

де `left` — координата X лівої сторони прямокутника; `top` — координата Y верхньої сторони прямокутника; `right` — координата X правої сторони прямокутника; `bottom` — координата Y нижньої сторони прямокутника; `width` та `height` — відповідно ширина та висота еліпса для скруглення кутів.

Для побудови еліптичної дуги, що вписана в прямокутник, використовується функція

```
BOOL Arc (HDC hdc, int x1, int y1, int x2, int y2, int x3, int y3, int x4,
int y4)
```

де (`x1`, `y1`) — координата верхнього лівого кута прямокутника; (`x2`, `y2`) — координата нижнього правого кута прямокутника; ; (`x3`, `y3`) та (`x4`, `y4`) — координати точок перетину еліпса та променів, що виходять з центра прямокутника (рис. 7.1).

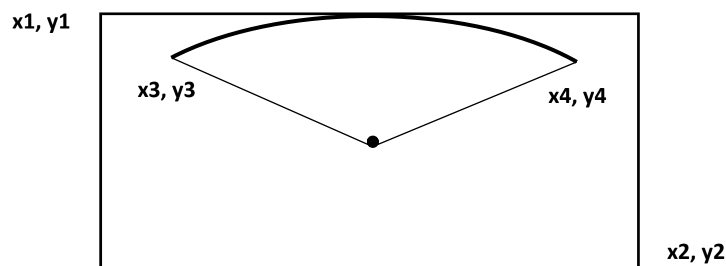


Рис. 7.1. Визначення координат точок дуги

Для побудови сектора еліпса, що вписаний в прямокутник, використовується функція

```
BOOL Pie (HDC hdc, int left, int top, int right, int bottom, int xr1, int
yr1, int xr2, int yr2)
```

де `left` — координата X лівої сторони прямокутника; `top` — координата Y верхньої сторони прямокутника; `right` — координата X правої сторони прямокутника; `bottom` — координата Y нижньої сторони прямокутника; (`xr1`, `yr1`) та (`xr2`, `yr2`) — координати точок перетину еліпса та променів, що виходять з центра прямокутника.

Для побудови сегмента еліпса, що вписаний в прямокутник, використовується функція

```
BOOL Chord (HDC hdc, int x1, int y1, int x2, int y2, int x3, int y3, int
x4, int y4).
```

Значення параметрів аналогічні до функції `Arc`.

Щоб побудувати ламану лінію використовується функція

```
BOOL Polyline (HDC hdc, CONST POINT *apt, int cpt)
```

де apt — покажчик на масив, що містить координати точок; cpt — кількість точок в масиві.

Для побудови багатокутника використовується функція

```
BOOL Polygon (HDC hdc, CONST POINT *apt, int cpt)
```

що має аналогічні параметри як і функція Polyline.

Функція

```
BOOL PolyBezier (HDC hdc, CONST POINT *apt, int cpt)
```

використовується для побудови кривої Без'є за заданими точками. Параметри команди аналогічні до параметрів двох попередніх функцій.

Окрім розглянутих функцій, GDI надає ще досить багато інших засобів для побудови зображень. Всі інші функції можна знайти в заголовному файлі *wingdi.h*.

7.3. Створення та використання пер і пензлів

Перо та пензель це об'єкти GDI, які можна створювати і використовувати. Для опису цих об'єктів існують відповідні структури даних: HPEN — для опису пера та HBRUSH — для опису пензля.

Щоб створити перо потрібно викликати функцію

```
HPEN CreatePen (int iStyle, int cWidth, COLORREF color)
```

де iStyle — це стиль пера; cWidth — товщина пера; color — колір пера. Стиль пера може приймати одне із наступних значень:

PS_SOLID	Суцільна лінія
PS_DASH	Штрихова лінія. Цей стиль застосовний тільки якщо товщина пера 0 або 1
PS_DOT	Лінія з точок. Цей стиль застосовний тільки якщо товщина пера 0 або 1
PS_DASHDOT	Штрихпунктирна лінія. Цей стиль застосовний тільки якщо товщина пера 0 або 1
PS_DASHDOTDOT	Штрихпунктирна лінія з двома точками. Цей стиль застосовний тільки якщо товщина пера 0 або 1
PS_NULL	Невидима лінія

Після створення пера, його необхідно зробити поточним. Для цього використовується функція

```
HGDIOBJ SelectObject (HDC hdc, HGDIOBJ h);
```

яка в якості другого параметра використовує посилання на будь-який об'єкт GDI. Після використання пера, його необхідно видалити командою

```
BOOL DeleteObject (HGDIOBJ ho).
```

Таким чином, ми можемо заздалегідь створити набір необхідних пер і керувати їх вибором під час побудови зображення. А вже по завершенню побудови зображення всі пера видалити.

Створення пензля відбувається аналогічним чином. Різниця лише в тому, що стиль пензля визначається функцією, яка його створює. Тому в GDI є декілька функцій для створення пензлів різних стилів.

Для створення пензля із суцільним зафарбовуванням використовується функція

```
HBRUSH CreateSolidBrush (COLORREF color)
```

де color — колір пензля. Такий пензель зафарбовує замкнені області суцільним заданим кольором.

Для створення пензля із заданим шаблоном штрихування використовується функція

```
HBRUSH CreateHatchBrush (int iHatch, COLORREF color)
```

де iHatch — це стиль штрихування; color — колір пензля. Змінна iHatch може набувати наступних значень:

HS_BDIAGONAL	Штрихування під кутом 45° зліва на право
HS_CROSS	Горизонтальне і вертикальне штрихування
HS_DIAGCROSS	Штрихування під кутом 45° зліва на право і навпаки
HS_FDIAGONAL	Штрихування під кутом 45° справа на ліво
HS_HORIZONTAL	Горизонтальне штрихування
HS_VERTICAL	Вертикальне штрихування

Для створення пензля із заданим паттерном зафарбовування використовується функція

```
HBRUSH CreatePatternBrush (HBITMAP hbm)
```

де hbm — посилання на бітову карту (*bitmap*), яка може бути створена безпосередньо в програмі або завантажена із зовнішнього файлу. Це може бути формат незалежний від пристрою (*device independent bitmap*, *DIP*) або залежний від пристрою формат.

Для DIP формату також є окрема функція для створення пензля

```
HBRUSH CreateDIPPatternBrushPt (const VOID *lpPackedDIB, UINT iUsage).
```

Використання пензлів аналогічне до використання пер — за допомогою функцій SelectObject та DeleteObject.

Розглянемо приклад, в якому спочатку створюється точкове перо зеленого кольору та пензель із суцільним зафарбовуванням. Потім вони робляться поточними і виконується побудова прямокутника. Далі створюється штрихпунктирне перо жовтого кольору та пензель із хрестоподібним штриховим зафарбовуванням синього кольору. Далі вони робляться поточними і виконується побудова еліпса. І насамкінець відбувається видалення об'єктів перо та пензель.

Приклад.

```
HDC hdc;
HBRUSH hBrush;
HPEN hPen;
PAINTSTRUCT ps;
…
case WM_PAINT:
{
  hdc = BeginPaint(hWnd, &ps);
    hPen = CreatePen(PS_DOT,1,RGB(0,255,0));
    hBrush = CreateSolidBrush(RGB(200,200,200));
    SelectObject(hdc, hPen);
    SelectObject(hdc, hBrush);
    Rectangle(hdc, 10,10,200,200);
    hPen = CreatePen(PS_DASHDOT,1,RGB(255,255,0));
    hBrush = CreateHatchBrush(HS_CROSS, RGB(0,0,255));
    SelectObject(hdc, hPen);
    SelectObject(hdc, hBrush);
    Ellipse(hdc, 10,10,200,200);
    DeleteObject(hPen);
    DeleteObject(hBrush);
  EndPaint(hWnd, &ps);
}
```

Запитання для самоконтролю

1. Що таке GDI? В якій операційній системі він доступний?
2. Які графічні об'єкти використовуються в GDI?
3. Які графічні примітиви можна побудувати за допомогою GDI?
4. Які типи пензлів можна створити в GDI?

8. Основи OpenGL

Перша версія бібліотеки була опублікована в 1992 році. Останньою версією на момент написання цього посібника є 4.6, що вийшла 31 липня 2017 року. Всі версії зворотно сумісні, але специфікації значно відрізняються і набір функції останньої версії абсолютно не співпадає з набором функцій перших версій. В цьому посібнику ми будемо розглядати набір функцій для версій 1.x — вони доступні на всіх версіях Windows без додаткових налаштувань та є найпростішими для початкового ознайомлення з бібліотекою.

Приклади написання коду будуть наведені для середовища Microsoft Visual Studio 2022 Community Edition (безкоштовна версія).

8.1. 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.

8.2. 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, 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 проєктування 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 проєктування, 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), що найбільш повно відповідає концепціям об'єктно-орієнтованого програмування і полегшує створення застосунків для одночасної роботи з зображеннями кількох сцен.

8.3. 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 під час 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-застосунку. 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. 8.1 схематично подана організація системи бібліотек у версії, що працює під управлінням системи Windows. Аналогічна організація використовується й в інших версіях OpenGL.

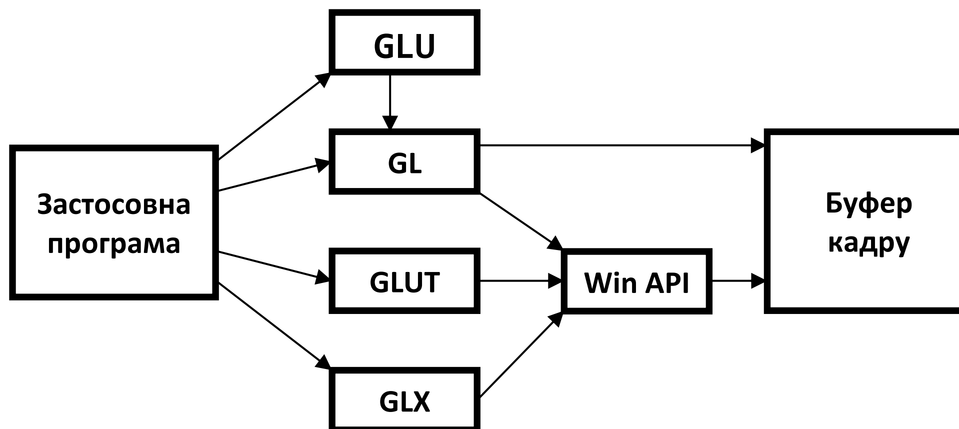


Рис. 8.1. Організація бібліотеки OpenGL

8.4. 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. 8.2).

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.

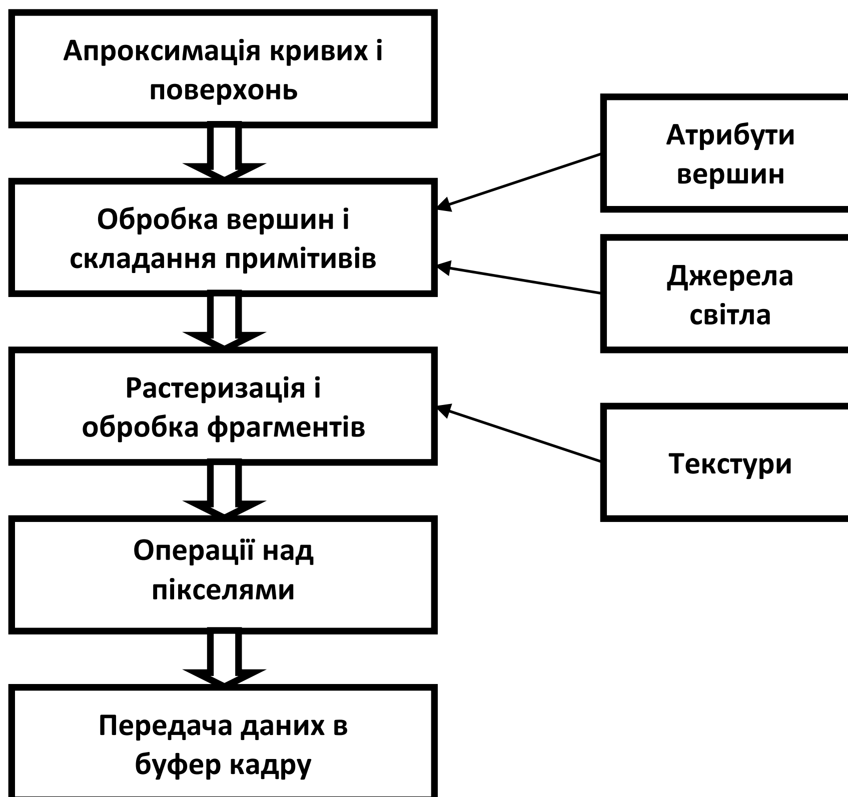


Рис. 8.2. Схема функціонування конвеєра OpenGL

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 в момент надходження в графічну систему координат вершини для побудови фігури, до якої вона належить. Зміна станів відбувається за допомогою команд, що оформлюються як виклики функцій.

8.5. Синтаксис команд

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

Усі команди бібліотеки OpenGL починаються з префіксу gl, усі константи — з префіксу 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 вигляд:

```
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
[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.
[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.

8.6. Шаблон застосунку Visual Studio з використанням OpenGL

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

Перш за все необхідно налаштувати проєкт для можливості використання в ньому бібліотеки OpenGL. Для цього створюємо Windows Desktop Application і заходимо в налаштування проєкту. Відкриваємо розділ Linker->Input та додаємо файли opengl32.dll і glu32.dll в пункт Additional Dependencies (рис. 8.3).

Не зважаючи на те, що зазвичай зараз використовується 64 розрядна платформа для операційних систем та застосунків, файли бібліотеки OpenGL мають закінчення 32 для зворотної сумісності. Тобто і для 32 розрядних і для 64 розрядних застосунків назви файлів бібліотек однакові, просто знаходяться в різних системних директоріях.

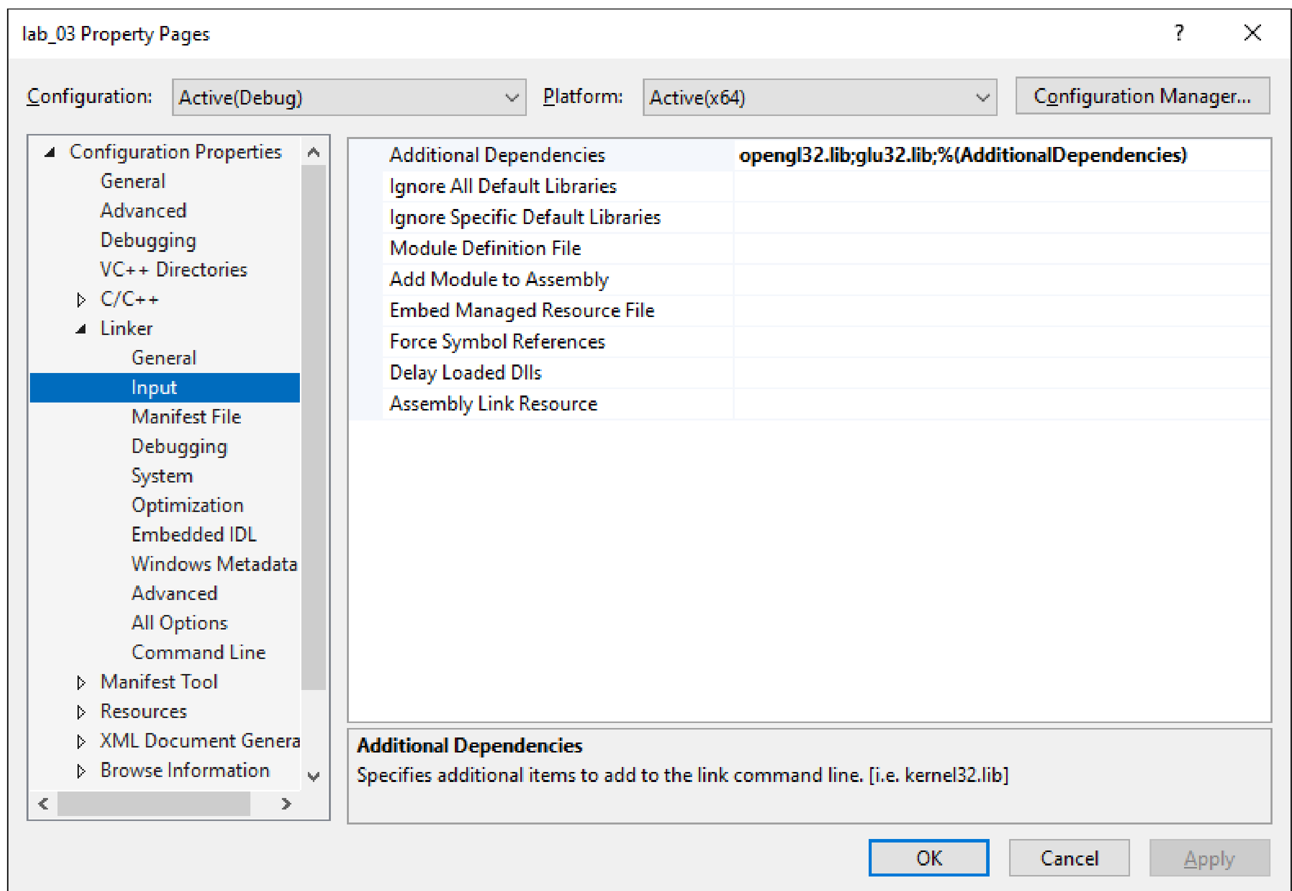


Рис. 8.3. Налаштування проєкту для використання OpenGL

Тепер нам необхідно додати новий модуль до проєкту. Назвемо його draw і в заголовному файлі (*draw.h*) опишемо прототип функції для встановлення формату пікселя. Ця функція обов'язково має бути описана окремо (не в якості методу класу).

```
//----------------------------------------------------------------
#include <windef.h>
//----------------------------------------------------------------
void set_dc_pixel_format(HDC hdc);
//----------------------------------------------------------------
```

Далі напишемо реалізацію цієї функції в файлі *draw.cpp*.

```
void set_dc_pixel_format(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);
}
```

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

Після заповнення полів структури PIXELFORMATDESCRIPTOR, 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 пікселя 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 PIXELFORMATDESCRIPTOR можна за допомогою файлу довідки або в заголовному файлі *wingdi.h*.

Наступним кроком необхідно включити заголовні файли OpenGL та наш новостворений файл draw.h до файлу framework.h, який згенерований автоматично під час створення проєкту.

```
#include <GL/gl.h>
#include <GL/glu.h>
#include "draw.h"
```

Таким чином головний модуль проєкту отримає доступ до функцій, що задекларовані в цих заголовних файлах.

Тепер переходимо в головний *cpp* файл проєкту, який має таку ж назву, як і сам проєкт (як ви назвете проєкт — не важливо). Всі подальші кроки ми будемо виконувати саме в цьому файлі.

В розділі глобальних змінних додаємо нові змінні.

```
HDC   GLDC;     // посилання на контекст пристрою
HGLRC GLRC;     // посилання на контекст відображення
UINT Width;     // ширина вікна
UINT Height;    // висота вікна
```

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 містить 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.

Наступним кроком потрібно додати ініціалізацію OpenGL в функцію InitInstance. В кінці функції, перед викликом ShowWindow необхідно додати наступні рядки.

```
GLDC = GetDC(hWnd);  // Отримання контексту пристрою
set_dc_pixel_format(GLDC);   // Установка формату пікселя
GLRC = wglCreateContext(GLDC);   // Отримання контексту відображення
wglMakeCurrent(GLDC, GLRC);  // Встановлення OpenGL поточним контекстом
glClearColor(1.0, 0.95, 0.85, 1.0);  //  Колір фону
```

Таким чином ми активізували контекст відображення OpenGL та зв'язали його з вікном проєкту.

Наступним кроком необхідно реалізувати правильне оброблення таких подій для вікна: зміна розміру, малювання та знищення вікна. Для цього перейдемо до автоматично створеної функції обробки подій вікна WinProc. Тут ми бачимо оператор switch, який здійснює вибір подій для вікна. Спочатку додамо подію WM_SIZE — зміна розмірів вікна.

```
case WM_SIZE:
        {
            Width = LOWORD(lParam);
            Height = HIWORD(lParam);
            glViewport(0, 0, Width, Height); // Область виводу
            glMatrixMode(GL_PROJECTION); // Матриця проєкцій
            glLoadIdentity(); // Записати в матрицю 1 по головній діагоналі
            gluOrtho2D(-1.01, 1.01, -1.01, 1.01);   // Розміри світу
            InvalidateRect(hWnd, nullptr, false);   // Перемалювати вікно
        }
        break;
```

Під час зміни розмірів вікна виконується написаний нами код. Команда glViewport задає область виведення у вікні (ми використовуємо все вікно повністю). Команда glMatrixMode встановлює поточну матрицю для редагування (ми встановили матрицю проєкцій) і записали в неї одиниці по головній діагоналі командою glLoadIdentity, тобто фактично встановили ортографічну проєкцію, в якій картинна площина співпадає з координатною площиною XOY. За замовчуванням світ OpenGL є прямокутником з координатами головної діагоналі (-1,-1) – (1,1). Але ми його трішки розширимо використавши команду gluOrtho2D — таким чином світ матиме координати (-1.01,-1.01) – (1.01, 1.01). Команда InvalidateRect викликає вбудовану функцію перемалювання вікна.

Наступною розглянемо подію знищення вікна — WM_DESTROY. Тут ми повинні знищити всі створені об'єкти та звільнити виділену пам'ять.

```
wglMakeCurrent(nullptr, nullptr); // Відключення поточного контексту
wglDeleteContext(GLRC); // Очищення поточного контексту відображення
ReleaseDC(hWnd, GLDC);  // Видалення контексту відображення OpenGL
DeleteDC(GLDC); // Видалення контексту пристрою
```

Вказаний фрагмент коду потрібно розмістити до команди `PostQuitMessage`.

І, нарешті, остання подія, яку ми повинні описати це подія перемалювання — `WM_PAINT`. Саме тут відразу після системного коментаря `//TODO` ми повинні розмістити код для побудови зображення за допомогою команд OpenGL.

```
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);
```

Першою командою йде очищення екрану і зафарбування його фоновим кольором. Потім задаємо колір квадрата (у прикладі — червоний), далі визначаємо його. Коли все зображення побудоване, виводимо його на екран за допомогою команди `SwapBuffers`.

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?

9. Формування зображень геометричних об'єктів

В цьому розділі будуть розглянуті базові принципи формування зображень за допомогою OpenGL. Ми розглянемо як побудувати зображення на площині та як налаштовати параметри відображення об'єктів.

9.1. 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;
- встановлення положення спостерігача;
- перетворення і малювання геометричних об'єктів.

Очищення буферів забезпечується командою:

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

Ця команда очищує вказані буфери з поточним значенням (див. розділ 12). 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. підрозділ 12.1). Як правило, вона дорівнює 1. За замовчуванням колір — чорний (0,0,0,1).

```
glClearColor (clampf r, clampf g, clampf b, clampf a).
```

Встановлення 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 наступному розділі.

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

9.2. 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 цей 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.

9.2.1. 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] (GLtype coords)
glVertex [2 3 4] [s i f d]v (GLtype* coords).
```

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.

9.2.2. Колір вершини

Для завдання поточного кольору вершини використовуються команди:

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

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], водночас, 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 (GLenum mode)
```

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

9.2.3. Нормаль

Визначити нормаль у вершині можна командами:

```
glNormal3 [b s i f d] (Gltype coords)
glNormal3[b s i f d]v (GLtype* coords).
```

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

Режим автоматичної нормалізації повинен бути увімкненим, якщо застосунок (прикладна програма) використовує модельні перетворення розтягування/стиснення, тому що в цьому випадку довжина нормалей змінюється під час множення 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 (GLenum mode)
glDisable (GLenum mode)
```

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

9.3. 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 (GLenum mode)
glEnd (void).
```

Параметр mode визначає тип примітива, який задається всередині і може приймати наступні значення (рис. 9.1):

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

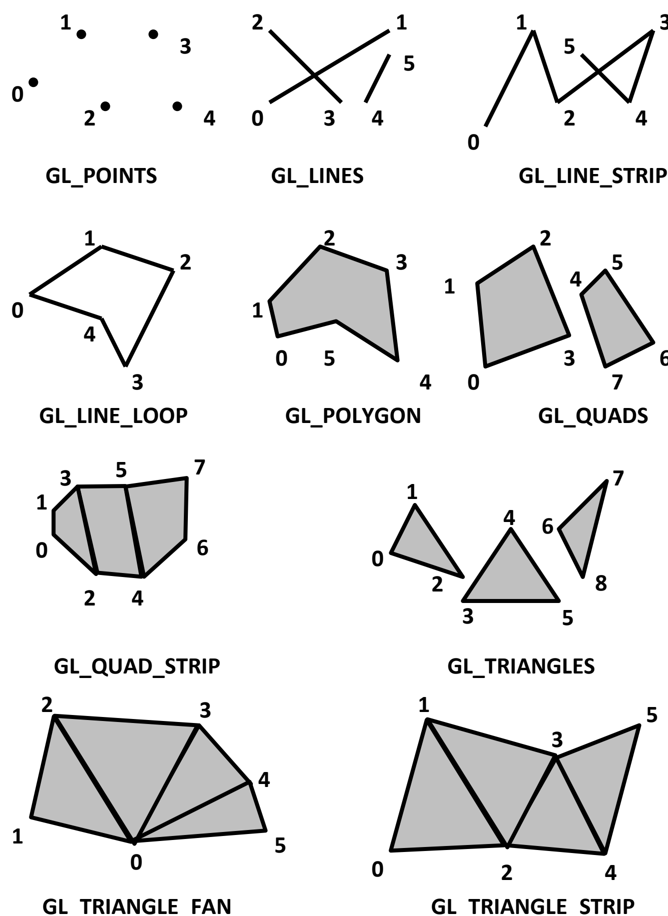


Рис. 9.1. Примітиви OpenGL

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

```
GLfloat BlueColor[3] = {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();
```

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 власне 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 (GLenum mode)
```

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 (GLenum face, GLenum mode).
```

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 потрібно встановити відповідний режим викликом команди glEnable(GL_CULL_FACE), а потім вибрати тип відображення граней за допомогою команди:

```
glCullFace (GLenum mode)
```

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

9.4. Штрихування багатокутників

За замовчуванням 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, відображується відповідний піксель в багатокутнику (Рис. 9.2).

	1	2	3	4
1	00	FF	FF	80
2	01	40	40	48
3	02	20	20	28
4	04	10	10	16
5	08	08	08	08
6	08	04	04	04
7	08	02	02	02
8	08	01	01	01
9	08	00	80	81
10	04	00	40	41
11	02	00	20	21
12	01	00	10	11
13	00	80	08	09
14	00	40	04	05
15	00	20	02	03
16	00	10	01	01
17	04	09	00	81
18	04	0A	00	41
19	04	04	00	21
20	04	0B	00	11
21	04	10	80	09
22	04	20	40	05
23	04	40	20	03
24	04	80	10	01
25	05	00	08	02
26	06	00	04	04
27	04	00	02	08
28	08	00	01	F0
29	1F	FF	00	00
30	20	00	00	00
31	40	00	00	00
32	80	00	00	00

7 6 5 4 3 2 1 0 7 6 5 4 3 2 1 0 7 6 5 4 3 2 1 0 7 6 5 4 3 2 1 0

Рис. 9.2. Ілюстрація шаблону штрихування

Фактично ми отримуємо двовимірну бітову маску, яку в програмі описуємо одновимірним масивом зі 128 байт — кожні 4 байти описують рядок з 32 пікселів. Щоб правильно розставити значення байтів необхідно знати правила перетворення двійкових чисел в десяткові, а бажано і в шістнадцяткові. На ілюстрації (Рис. 8.2) використані шістнадцяткові числа для подання значення кожного байту. В масив байти вносяться в такому порядку: зліва на право, знизу до гори.

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

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

Приклад.

```
GLubyte grating[128] = (0x80, 0x00, 0x00, 0x00, // 32-й рядок
                        0x40, 0x00, 0x00, 0x00, // 31-й рядок
…);
…
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. Проте варто пам'ятати — штрихування відбувається на етапі растеризації в площині вікна, тобто ніяк не пов'язане з реальною геометрією об'єкта. Це означає, що, наприклад, у випадку повороту прямокутника на 45° шаблон штрихування не буде повернутий, а залишиться в площині вікна незмінним. Для штрихування в площині об'єкта необхідно використовувати текстури (див. розділ 12).

9.5. 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 12).

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

```
GLUquadricObj* quadObj;
```

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'єкта задається командою gluQuadricDrawStyle, першим аргументом якої вказується ім'я об'єкта, а другим службове слово, яке встановлює стиль відображення об'єкта (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 (GLUquadricObj* quadObj, Gldouble radius, Glint slices, Glint stacks).
```

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

```
gluCylinder (GLUquadricObj* quadObj, GLdouble baseRadius, GLdouble topRadius, GLdouble height, Glint slices, Glint stacks).
```

Ця 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(GLUquadricObj* quadObj, GLdouble innerRadius, GLdouble outerRadius, Glint slices, Glint loops).
```

Функція будує плоский диск (коло) з центром в початку системи координат і радіусом outerRadius. Водночас якщо значення innerRadius відмінне 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(GLUquadricObj* quadObj, GLdouble innerRadius, GLdouble outerRadius, Glint slices, Glint loops, GLdouble startAngle, GLdouble sweepAngle).
```

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(GLdouble radius, Glint slices, Glint stacks)
glutWireSphere(GLdouble radius, Glint slices, Glint stacks).
```

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

```
glutSolidCube(GLdouble size)
glutWireCube(GLdouble size).
```

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

```
glutSolidCone(GLdouble base, GLdouble height, Glint slices, Glint stacks)
glutWireCone( GLdouble base, GLdouble height, Glint slices, Glint stacks).
```

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

```
glutSolidTorus(GLdouble innerRadius, GLdouble outerRadius, Glint nsides, Glint rings)
glutWireTorus( GLdouble innerRadius, GLdouble outerRadius, Glint nsides, Glint rings).
```

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 з радіусом описаної сфери довкола нього, що 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)` .

9.6. TESS-ОБ'ЄКТИ

Мозаїчні (*tesselated*) об'єкти — є одним з останніх нововведень бібліотеки GLU, вони призначені для спрощення побудови неопуклих багатокутників.

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

```
GLUtesselator* gluTess;
…
gluTess = 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:

```
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);
```

Побудова багатокутника відбувається наступним чином:

```
gluTessBeginPolygon(gluTess, nullptr);
  gluTessBeginContour(gluTess);
    for(int i = 0; i < Max; i++) gluTessVertex(gluTess, Obj[i], Obj[i]);
  gluTessEndContour(gluTess);
gluTessEndPolygon(gluTess);
```

Де `Obj` це:

```
TVector Obj[10] ={ {-2.5,-4.5,-1.0}, {4.0,-4.5,-1.0}, {2.5,-2.5,-1.0},
 {-1.0,-2.5,-1.0}, {3.5,2.5,-1.0}, {2.5,4.5,-1.0}, {-4.0,4.5,-1.0},
 {-2.5,2.5,-1.0}, {1.0,2.5,-1.0}, {-4.0,-2.5,-1.0} };
```

В свою чергу `TVector` це тип даних користувача, який визначає масив із трьох значень типу `GLdouble`, що відповідають координатам точки (x, y, z):

```
typedef GLdouble TVector[3];
```

Слід зауважити що для правильної побудови багатокутника його точки необхідно задавати в порядку 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 неправильно, і отвір перетинатиме зовнішній контур, то tess-об'єкт не буде виведений на екран.

9.7. Криві та поверхні

Бібліотека OpenGL містить команди, що дозволяють будувати криві та поверхні декількох типів: Без'є та B-сплайни.

9.7.1. Криві та поверхні Без'є

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

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

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, 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](GLfloat u).
```

***Приклад*.**

```
glBegin(GL_POINTS); // or GL_LINE_STRIP
  for(int i:=0; i<31; i++) glEvalCoord1f (i/30);
glEnd();
```

Альтернативою може слугувати використання команд:

```
glMapGrid1 [f d](Glint n, Glint u1, Glint u2).
glEvalMesh1 (GLenum mode, Glint p1, Glint p2)
```

Перша визначає одновимірну сітку від u1 до u2 з кроком n, а друга задає режим візуалізації для згенерованої сітки. 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](GLenum type, GLfloat u_min, GLfloat u_max, Glint u_stride,
Glint u_order, GLfloat v_min, GLfloat v_max, Glint v_stride, Glint v_order,
GLfloat* point_array).
```

`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(int j:=0; j<101; j++) begin
  glBegin(GL_LINE_STRIP); // or GL_QUAD_STRIP
    for(int i:=0; i<101; j++) glEvalCoord2f(i/100,j/100);
  glEnd();
  glBegin(GL_LINE_STRIP); // or GL_QUAD_STRIP
    for(int i:=0; i<101; j++) glEvalCoord2f(i/100,j/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`.

9.7.2. 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 цей клас поверхонь. Для роботи з NURBS-об'єктами у бібліотеці GLU містяться змінні спеціального типу:

```
GLUnurbsObj* theNurb;
```

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

```
theNurb = gluNewNurbsRenderer();
```

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

```
gluDeleteNurbsRenderer(gluNurbsObj* theNurb);
```

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. Під час роботи з NURBS-кривими замість них необхідно використовувати gluBeginNurbsCurve, gluNurbsCurve, gluEndNurbsCurve.

Розглянемо основні команди більш детально. Команда, яка будує криву, має наступні параметри:

```
gluNurbsCurve (GLUnurbsObj* theNurb, Glint kcount, GLfloat* cknots, Glint stride, GLfloat* points, Glint order, GLenum type).
```

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(gluNurbsObj* theNurb, Glint knot_count, GLfloat* uknot, Glint vknot_count, GLfloat* vknot, Glint u_stride, Glint v_stride, GLfloat* ctlarray, Glint uorder, Glint vorder, GLenum type).
```

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 двовимірних типів обчислювача. Зазвичай 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 це 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).

9.8. 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 (GLuint list, GLenum mode)
glEndList().
```

Для ідентифікації списків використовуються цілі позитивні числа, що задаються під час створення 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 (GLuint list)
```

вказавши в параметрі list його ідентифікатор. Викликати відразу кілька списків можна командою

```
glCallLists (GLsizei n, GLenum type, const GLvoid* lists)
```

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 (Glint list, GLsizei range)
```

що 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.

9.9. Масиви вершин

Якщо вершин багато, то щоб не викликати для кожної з них команду glVertex*, зручно поєднувати вершини в масиви командою

glVertexPointer (Glint *size*, GLenum *type*, GLsizei *stride*, **const** GLvoid* *ptr*)

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 (Glenum *type*, GLsizei *stride*, **const** GLvoid* *ptr*)

glColorPointer (Glint *size*, GLenum *type*, GLsizei *stride*, **const** GLvoid* *ptr*)

можна визначити масив нормалей, кольорів і деяких інших атрибутів вершини. Для того, щоб ці масиви надалі можна було використовувати, потрібно викликати команду

glEnableClientState (GLenum *array*)

з параметрами GL_VERTEX_ARRAY, GL_NORMAL_ARRAY, GL_COLOR_ARRAY відповідно. Після закінчення роботи з масивом бажано викликати команду

glDisableClientState (GLenum array)

з відповідним значенням параметра array.

Для відображення вмісту масивів використовується команда

glArrayElement (Glint *index*)

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 (GLenum mode, Glint first, GLsizei count)
```

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 (GLenum mode, GLsizei count, GLenum type, const GLvoid* indices)
```

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?

10. Перетворення об'єктів

В OpenGL використовуються як основні три системи координат: *лівобічна*, *правобічна* і *віконна* (рис. 10.1). 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 праворуч щодо спостерігача, вісь Y — вгору.

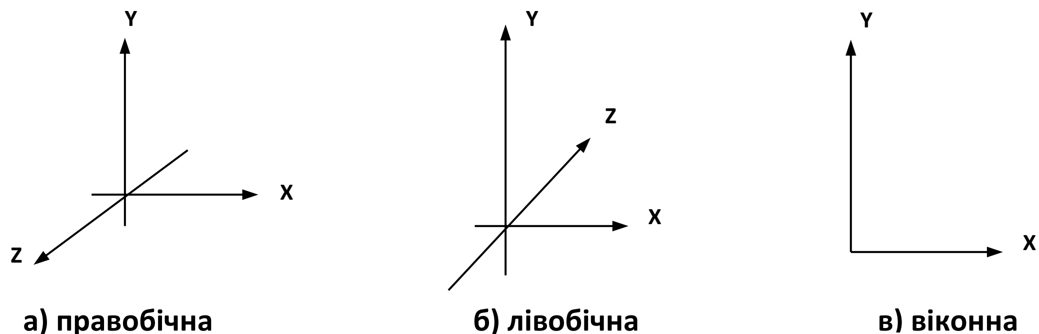


Рис. 10.1. Системи координат в OpenGL

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.

10.1. Робота з матрицями

Для завдання різних перетворень об'єктів в OpenGL використовуються операції над матрицями. Розрізняють три типи матриць: модельно-видова, матриця проєкцій 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 проєкцій визначає, як будуть проєктуватися 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 (GLenum mode)
```

виклик якої зі значенням параметра `mode` рівним `GL_MODELVIEW`, `GL_PROJECTION`, або `GL_TEXTURE` включає режим роботи з модельно-видовою матрицею, матрицею проєкцій або 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] (GLtype* m)
```

де `m` вказує на масив з 16 елементів типу `float` чи `double` відповідно до назви команди, в якому спочатку повинен 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 стеку, 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] (GLtype* m)
```

де параметр `m` повинен 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. 10.2.

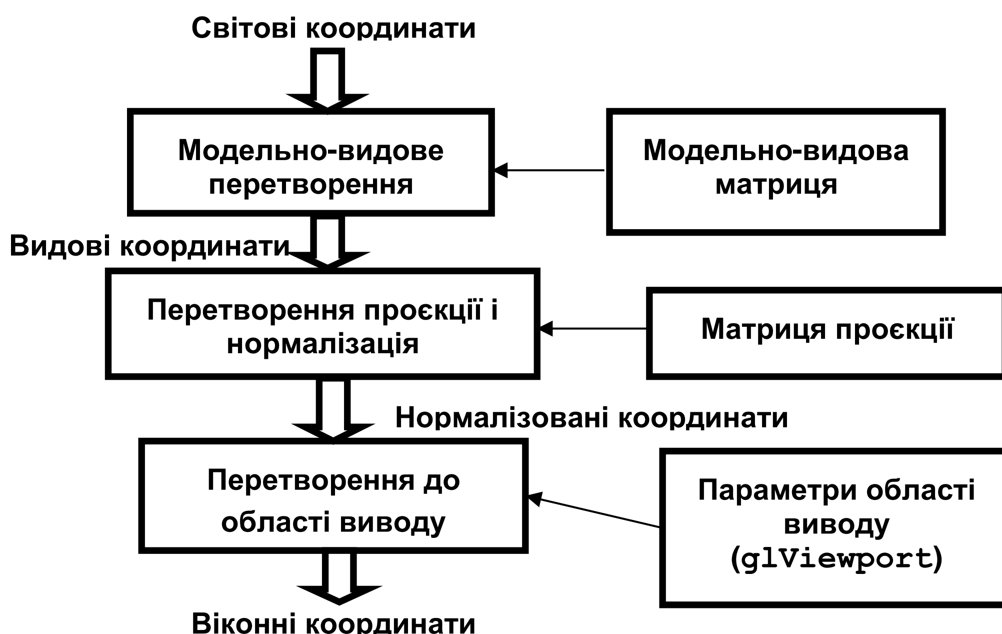


Рис. 10.2. Перетворення координат в OpenGL

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.

10.2. 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:

$$(x', y', z', 1)^T = M \cdot (x, y, z, 1)^T$$

де M — матриця модельно-видового перетворення. Перспективне перетворення і проєкція виконуються аналогічно. Сама матриця може бути створена за допомогою наступних команд:

```
glTranslate[f d] (GLtype x, GLtype y, GLtype z)
glRotate[f d] (GLtype angle, GLtype x, GLtype y, GLtype z)
glScale[f d] (GLtype x, GLtype y, GLtype z)
```

`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 (GLdouble eyex, GLdouble eyey, GLdouble eyez, GLdouble centerx,
GLdouble centery, GLdouble centerz, GLdouble upx, GLdouble upy, GLdouble
upz),
```

де точка (eyex, eyey, eyez) визначає точку спостереження, (`centerx`, `centery`, `centerz`) задає центр сцени, що буде проєктуватися 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`. І вже тоді визначайте сам об'єкт.

10.3. Проєкції

В OpenGL існують стандартні команди для завдання ортографічної (паралельної) і перспективної проєкцій. Перший тип проєкції (рис. 10.3) може бути заданий командами

```
glOrtho (GLdouble left, GLdouble right, GLdouble bottom, GLdouble top, GLdouble znear, GLdouble zfar)
gluOrtho2D (GLdouble left, GLdouble right, GLdouble bottom, GLdouble top)
```

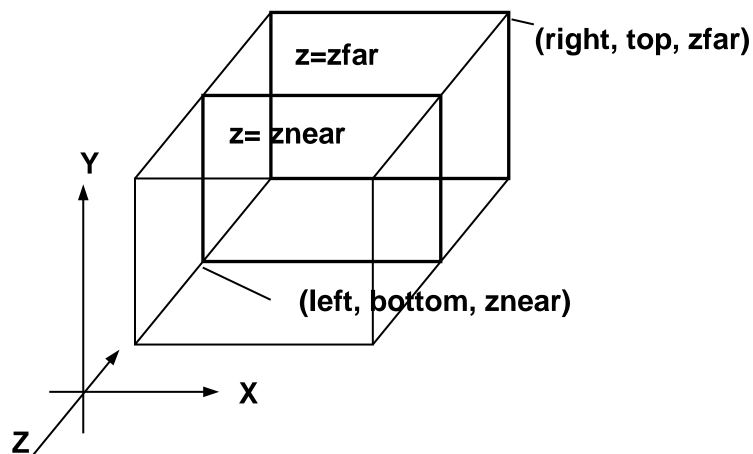


Рис. 10.3. Ортографічна проєкція

Перша команда створює матрицю проєкції 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 проєкція (рис. 10.4) визначається командою

```
gluPerspective (GLdouble angley, GLdouble aspect, GLdouble znear, GLdouble zfar)
```

яка задає зрізаний конус видимості в лівобічній системі координат.

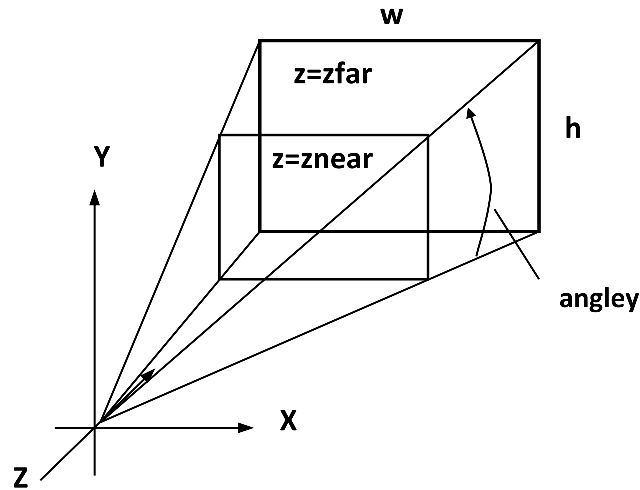


Рис. 10.4. Перспективна проєкція

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 проєкцій, не забудьте увімкнути режим роботи з потрібною матрицею командою glMatrixMode (GL_PROJECTION) і скинути поточну викликом glLoadIdentity. ***Приклад.***

```
// ортографічна проєкція
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
glOrtho(0, w, 0, h, -1.0, 1.0);
```

10.4. Робоча область

Після застосування матриці проєкцій 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 (Glint x, Glint y, Glint width, Glint height).
```

Значення всіх параметрів задаються в пікселях та визначають ширину і висоту робочої області з координатами лівого верхнього кута (x,y) у віконній системі координат. Розміри віконної системи координат визначаються поточними

розмірами вікна застосунку, точка (0,0) знаходиться в лівому верхньому 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:

$$(x_w, y_w, z_w)^T = \left(\left[\frac{p_x}{2} \cdot x_n + O_x\right], \left[\frac{p_y}{2} \cdot y_n + O_y\right], \left[\frac{f-n}{2} \cdot z_n + \frac{n+f}{2}\right]\right)^T$$

Де 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 (GLclampd n, GLclampd f)
```

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»?

11. 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 та 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.

11.1. 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 9.2.3. режим нормалізації нормалей, що включається викликом команди `glEnable(GL_NORMALIZE)`.

Для задання глобальних параметрів освітлення використовуються команди

```
glLightModel[i f] (GLenum pname, GLenum param)
glLightModel[i f]v (GLenum pname, GLtype* params).
```

Аргумент `pname` визначає, який параметр моделі освітлення буде налаштовуватися 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).

11.2. СПЕЦИФІКАЦІЯ МАТЕРІАЛІВ

Для задання параметрів поточного матеріалу використовуються команди:

```
glMaterial[i f] (GLenum face, GLenum pname, GLtype param)
glMaterial[i f]v (GLenum face, GLenum pname, GLtype* params).
```

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 задання 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 (GLenum face, GLenum pname)
```

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:

```
GLfloat mat_dif[3] = {0.8,0.8,0.8};
GLfloat mat_amb[3] = {0.2, 0.2, 0.2};
GLfloat mat_spec[3] = {0.6, 0.6, 0.6};
GLfloat 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);
```

11.3. 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] (GLenum light, GLenum pname, GLfloat param)
glLight[i f]v (GLenum light, GLenum pname, GLfloat params).
```

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 під час розрахунку освітленості враховується тільки напрямок на точку (x,y,z), у протилежному випадку вважається, що джерело розташоване в точці (x,y,z,w). У першому випадку ослаблення світла з віддаленням 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 під час віддалення від джерела. Допускаються тільки позитивні значення. Якщо джерело не є спрямованим (див. GL_POSITION), то ослаблення обернено пропорційно сумі: $att_{constant} + att_{linear} \cdot d + att_{quadratic} \cdot d^2$ де d — відстань між джерелом світла і освітлюваною ним вершиною, $att_{constant}$, att_{linear} і $att_{quadratic}$ дорівнюють параметрам, заданим за допомогою вказаних констант. За замовчуванням ці параметри задаються трійкою (1, 0, 0), і фактично ослаблення не відбувається.

Під час зміни 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 за вимкненого освітлення колір вершини дорівнює поточному кольору, що задається командами `glColor`. За увімкненого освітлення колір вершини обчислюється з урахуванням інформації про матеріал, нормалі та джерела світла.

З вимкненим освітленням візуалізація відбувається швидше, однак у такому випадку застосунок повинен сам розраховувати кольори вершин.

11.4. 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.

Для включення ефекту затуманення необхідно викликати команду `glEnable(GL_FOG)`.

Метод обчислення інтенсивності туману у вершині можна визначити за допомогою команд:

```
glFog[if] (GLenum pname, GLfloat param)
glFog[if]v (GLenum pname, const GLfloat* param)
```

Аргумент pname може приймати наступні значення:

<table>
<tr><td rowspan="5">GL_FOG_MODE</td><td colspan="2">аргумент param 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>
<tr><td>GL_FOG_DENSITY</td><td colspan="2">аргумент param визначає коефіцієнт d</td></tr>
<tr><td>GL_FOG_START</td><td colspan="2">аргумент param визначає коефіцієнт s</td></tr>
<tr><td>GL_FOG_END</td><td colspan="2">аргумент param визначає коефіцієнт e</td></tr>
<tr><td>GL_FOG_COLOR</td><td colspan="2">у цьому випадку params — покажчик на масив з 4-х компонентів кольору туману</td></tr>
</table>

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?

12. НАКЛАДАННЯ ТЕКСТУРИ

Під *текстурою* будемо розуміти деяке растрове зображення, яке треба певним чином накласти на об'єкт, наприклад, для додання ілюзії рельєфності поверхні.

Накладання текстури на поверхню об'єктів сцени підвищує її реалістичність, однак водночас потрібно 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.

12.1. 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 за допомогою стандартних функцій мови С++.

Під час створення 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 висувається для компактного розміщення текстури в текстурній пам'яті і сприяє її ефективному використанню. Працювати тільки з такими текстурними файлами зазвичай незручно, тому після завантаження їх потрібно перетворити. Зміну розмірів текстури можна провести за допомогою команди

```
gluScaleImage (GLenum format, Glint widthin, Glint heightin, GLenum typein, pointer datain, Glint widthout, Glint heightout, GLenum typeout, pointer dataout).
```

У якості значення параметра `format` зазвичай 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 записує в область пам'яті, на яку вказує параметр `dataout`.

По-друге, треба передбачити випадок, коли об'єкт після растеризації виявиться за розмірами значно меншим накладеної на нього текстури. Чим менший об'єкт, тим меншою повинна бути текстура, що накладається 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 накладання 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 команд

glTexImage2D (GLenum *target*, GLint *level*, GLint *internalformat*, GLsizei *width*, GLsizei *height*, GLint *border*, GLenum *format*, GLenum *type*, **const** GLvoid **pixels*)

gluBuild2DMipmaps (GLenum *target*, Glint *components*, Glint *width*, Glint *height*, GLenum *format*, GLenum *type*, **const** GLvoid **pixels*)

де параметр target повинен дорівнювати GL_TEXTURE_2D. Параметри width, height, pixels визначають розміри і розташування текстури відповідно, а format і type мають аналогічне значення, що й у команди gluScaleImage.

Перша функція будує один із заданих рівнів деталізації текстури, який вказується параметром level, а параметр internalformat задає кількість кольорів текстури (1,2,3 або 4). Параметр border визначає наявність або відсутність рамки — 1 або 0 відповідно.

Друга функція будує всі рівні деталізації одночасно. Параметр components визначає кількість колірних компонентів текстури і може набувати наступних основних значень:

GL_LUMINANCE	один компонент — яскравість (текстура буде монохромною)
GL_RGB	червоний, синій, зелений
GL_RGBA	усі компоненти

Після виконання однієї з цих команд текстура копіюється у внутрішню пам'ять OpenGL, і тому пам'ять, що зайнята початковим зображенням, можна звільнити.

У OpenGL допускається використання одновимірних текстур, тобто розміру 1×N, однак, це завжди треба вказувати, задаючи як значення target константу GL_TEXTURE_1D. Корисність одновимірних текстур сумнівна, тому не будемо зупинятися на них докладно.

За використання в сцені декількох текстур, у OpenGL застосовується підхід, що нагадує створення списків зображень (так звані *текстурні об'єкти*). Спочатку за допомогою команди

glGenTextures (GLsizei *n*, GLuint* *textures*)

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 (GLenum *target*, GLuint *texture*)

де target може приймати значення GL_TEXTURE_1D або GL_TEXTURE_2D, а параметр texture повинен дорівнювати ідентифікатору тієї текстури, до якої будуть застосовані 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.

Розглянемо функцію, яка завантажує файл *BMP* та робить з нього текстуру.

```
void load_texture(GLuint tex, LPCWSTR filename)
{ HBITMAP hBitmap;
  BITMAP bm;
// BMP 24 bit, підтримувані розміри: 64x64, 128x128, 256x256, 512x512
  BYTE R, G, B;
// завантаження файлу BMP
  hBitmap = (HBITMAP)LoadImage(nullptr, filename, IMAGE_BITMAP, 0, 0,
                             LR_LOADFROMFILE | LR_CREATEDIBSECTION);
  GetObject(hBitmap, sizeof(BITMAP), &bm);
// отримання адреси початку даних зображення в пам'яті
  LPVOID data = bm.bmBits;
// зміна порядку кольорів з BGR в RGB
  UINT idx = 0;
  for(UINT i = 0; i < UINT(bm.bmWidth * bm.bmHeight); i++)
  { idx = i * 3;
    B = ((BYTE*)data)[idx + 0];
    G = ((BYTE*)data)[idx + 1];
    R = ((BYTE*)data)[idx + 2];
    ((BYTE*)data)[idx + 0] = R;
    ((BYTE*)data)[idx + 1] = G;
    ((BYTE*)data)[idx + 2] = B;
  }
// створення текстури із завантаженого зображення
  glGenTextures(1, &tex);
  glBindTexture(GL_TEXTURE_2D, tex);
  glTexImage2D(GL_TEXTURE_2D, 0, 4, bm.bmWidth, bm.bmHeight, 0, GL_RGB,
               GL_UNSIGNED_BYTE, bm.bmBits);
}
```

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.

12.2. Накладання текстури на об'єкти

Під час накладання текстури, як вже згадувалося, треба враховувати випадок, коли розміри текстури відрізняються від віконних розмірів об'єкта, на який вона накладається. Під час накладання 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. 12.1).

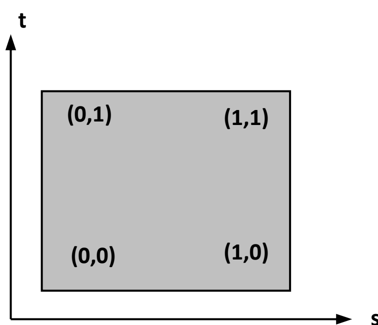


Рис. 12.1. Текстурні координати

Для зміни різних параметрів текстури застосовуються команди:

```
glTexParameter[i f] (GLenum target, GLenum pname, GLenum param)
glTexParameter[i f]v (GLenum target, GLenum pname, GLenum* params).
```

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. За значення GL_NEAREST буде використовуватися один (найближчий), а за значення GL_LINEAR (значення за замовчуванням) чотири найближчих елементи текстури
GL_TEXTURE_MAG_FILTER	параметр param визначає функцію, яка буде використовуватися для збільшення (розтягання) текстури. Значення аналогічні попередньому параметру
GL_TEXTURE_WRAP_S	параметр param встановлює значення координати s, якщо воно не входить у відрізок [0,1]. За значення GL_REPEAT (значення за замовчуванням) ціла частина s відкидається, і в результаті зображення розмножується по поверхні. За значення 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 накладання текстури, однак водночас 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] (GLenum target, GLenum pname, GLtype param)
glTexEnv[i f]v (GLenum target, GLenum pname, GLtype* params).
```

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

12.3. Текстурні координати

Перед накладанням текстури на об'єкт необхідно встановити відповідність між точками на поверхні об'єкта 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] (GLtype coord)
glTexCoord[1 2 3 4][s i f d]v (GLtype* coords).
```

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 (GLUquadricObj* quadObject, GLboolean textureCoords)
```

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] (GLenum coord, GLenum pname, GLtype param)
glTexGen[i f d]v (GLenum coord, GLenum pname, GLtype* params).
```

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 накладання 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 цього методу використовуються видові координати і необхідне задання 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 покажчиком 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 задання 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. За такого накладання 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.

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?

13. Операції з пікселями

Після проведення всіх операцій по перетворенню координат вершин, обчислення кольору і т.п., OpenGL переходить до етапу *растеризації*, на якому відбувається растеризація всіх примітивів, накладення текстури і створення ефекту туману. 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 8, графічна бібліотека OpenGL підтримує роботу з наступними буферами:

- кілька буферів кольору;
- буфер глибини;
- буфер-накопичувач (акумулятор);
- буфер маски (трафарету).

Група буферів кольору включає буфер кадру, але таких буферів може бути декілька. У разі використання 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 буде описана далі. Буфер маски використовується для формування піксельних масок (трафаретів), які служать для вирізання із загального масиву тих пікселів, які варто вивести на екран.

13.1. 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(GLenum src, GLenum dst)
```

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:

$$res = c_{src} \cdot k_1 + c_{dst} \cdot k_2,$$

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).

Розглянемо найбільш часто використовувані значення аргументів `src` і `dst`:

GL_SRC_ALPHA	$k = (A_s, A_s, A_s, A_s)$
GL_SRC_ONE_MINUS_ALPHA	$k = (1,1,1,1) - (A_s, A_s, A_s, A_s)$
GL_DST_COLOR	$k = (R_d, G_b, B_b)$
GL_ONE_MINUS_DST_COLOR	$k = (1,1,1,1) - (R_b, G_b, B_b, A_b)$
GL_DST_ALPHA	$k = (A_d, A_d, A_d, A_d)$
GL_DST_ONE_MINUS_ALPHA	$k = (1,1,1,1) - (A_d, A_d, A_d, A_d)$
GL_SRC_COLOR	$k = (R_s, G_s, B_s)$
GL_ONE_MINUS_SRC_COLOR	$k = (1,1,1,1) - (R_s, G_s, B_s, A_s)$

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:

```
glColor4f(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;
- під час виведення об'єкти з прозорістю повинні бути упорядковані за зменшенням 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.

13.2. 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(GLenum buf).
```

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(GLenum buf).
```

Значення buf аналогічне значенню відповідного аргументу в команді glReadBuffer.

Всі операції з буфером-накопичувачем контролюються командою

```
glAccum(GLenum op, GLfloat value).
```

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.

13.3. Буфер маски

Під час виведення 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 піксель в буфер кадру, механізм візуалізації OpenGL дозволяє виконати порівняння (тест) між заданим значенням і значенням в буфері маски. Якщо тест проходить, піксель малюється в буфері кадру.

Механізм порівняння дуже гнучкий і контролюється наступними командами:

```
glStencilFunc (GLenum func, Glint ref, GLuint mask)
glStencilOp (GLenum sfail, GLenum dpfail, GLenum dppass).
```

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:

$$[(ref\ AND\ mask)\ or\ (svalue\ AND\ mask)].$$

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 потрібно виконати команду glEnable(GL_STENCIL_TEST).

Буфер маски використовується для створення таких спецефектів як: тінь, віддзеркалення, плавні переходи однієї картинки в іншу тощо.

13.4. Керування растеризацією

Спосіб виконання растеризації примітивів можна частково регулювати командою

```
glHint (GLenum target, GLenum mode)
```

де target — вид контрольованих дій, приймає одне з наступних значень:

GL_FOG_HINT	точність обчислень під час накладання туману. Обчислення можуть виконуватися для кожного пікселя (найбільша точність) чи тільки у вершинах. Якщо реалізація OpenGL не підтримує попіксельного

	обчислення, то виконується тільки обчислення у вершинах
GL_LINE_SMOOTH_HINT	керування якістю прямих. Для значення mode, яке дорівнює GL_NICEST, зменшується ступінчастість прямих за рахунок більшого числа пікселів у прямих
GL_PERSPECTIVE_ CORRECTION_HINT	точність інтерполяції координат під час обчислення кольорів і накладання текстури. Якщо реалізація OpenGL не підтримує режим GL_NICEST, то здійснюється лінійна інтерполяція координат
GL_POINT_ SMOOTH_HINT	керування якістю точок. Для значення 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.

14. 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.

14.1. 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 цієї 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 для 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(int i:=0; i<=samples_count;i++)
{
// зазвичай samples_count лежить у межах від 5 до 10
  ShiftCamera(i); // зміщуємо камеру
  RenderScene();
  if(i==0)
   // на першій ітерації завантажуємо зображення
   glAccum(GL_LOAD,1/samples_count);
  else
   // додаємо до вже існуючого
   glAccum(GL_ACCUM,1/samples_count);
}
// 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 повторній дискретизації (*resampling*) зображень.

14.2. 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 проєкції. 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 проєкція паралельно вектору, що з'єднує задану 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 цього методу. Нехай:

P — точка в тривимірному просторі, що відкидає тінь;

L — положення джерела світла, що освітлює дану точку;

`S=a(L-P)-P` — точка, в 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 задану 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:

```
glEnable(GL_BLEND); // Робимо тіні напівпрозорими змішуванням
glBlendFunc(GL_SRC_ALPHA,GL_ONE_MINUS_SRC_ALPHA);
glDisable(GL_LIGHTING);
glColor4f(0.0, 0.0, 0.0, 0.5);
glPushMatrix();
  glMultMatrixf(floorShadow); // Проєктуємо тінь
  RenderGeometry(); // Формуємо зображення сцени в проєкції
glPopMatrix();
glEnable(GL_LIGHTING);
glDisable(GL_BLEND);
RenderGeometry(); // Формуємо зображення сцени в звичайному режимі
```

Матриця *floorShadow* може бути отримана із розглянутих рівнянь за допомогою наступної функції:

```
void shadow_matrix(GLfloat plane[4], // коефіцієнти рівняння площини
                   GLfloat light[4], // координати джерела світла
                   GLfloat matrix[4][4])
{
  GLfloat dot = plane[0] * light[0] + plane[1] * light[1] +
                plane[2] * light[2] + plane[3] * light[3];
  matrix[0][0] = GLfloat(dot - GLdouble(light[0]) * plane[0]);
  matrix[1][0] = GLfloat(0.0 - GLdouble(light[0]) * plane[1]);
  matrix[2][0] = GLfloat(0.0 - GLdouble(light[0]) * plane[2]);
  matrix[3][0] = GLfloat(0.0 - GLdouble(light[0]) * plane[3]);
  matrix[0][1] = GLfloat(0.0 - GLdouble(light[1]) * plane[0]);
  matrix[1][1] = GLfloat(dot - GLdouble(light[1]) * plane[1]);
  matrix[2][1] = GLfloat(0.0 - GLdouble(light[1]) * plane[2]);
  matrix[3][1] = GLfloat(0.0 - GLdouble(light[1]) * plane[3]);
  matrix[0][2] = GLfloat(0.0 - GLdouble(light[2]) * plane[0]);
  matrix[1][2] = GLfloat(0.0 - GLdouble(light[2]) * plane[1]);
  matrix[2][2] = GLfloat(dot - GLdouble(light[2]) * plane[2]);
  matrix[3][2] = GLfloat(0.0 - GLdouble(light[2]) * plane[3]);
  matrix[0][3] = GLfloat(0.0 - GLdouble(light[3]) * plane[0]);
  matrix[1][3] = GLfloat(0.0 - GLdouble(light[3]) * plane[1]);
  matrix[2][3] = GLfloat(0.0 - GLdouble(light[3]) * plane[2]);
  matrix[3][3] = GLfloat(dot - GLdouble(light[3]) * plane[3]);
}
```

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 спроєктовані 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, 0xFFFFFFF);
// значення в буфері дорівнює 2, якщо тінь вже виведена
glStencilOp (GL_KEEP, GL_KEEP, GL_INCR);
// виводимо тіні
RenderShadow();
```

Однак, навіть із застосуванням 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 цього посібника.

14.3. Дзеркальні відображення

У цьому параграфі розглянемо алгоритм побудови відображень від плоских об'єктів. Такі відображення додають більше реалістичності побудованому зображенню і їх відносно легко отримати.

Алгоритм використовує інтуїтивне подання 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. 14.1).

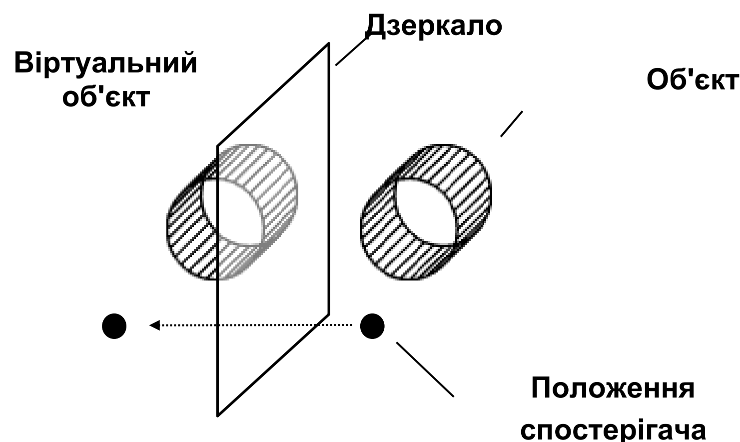


Рис. 14.1. Схема дзеркального відображення

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 ж 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 проєкції дзеркала на екран.
3. Формуємо сцену, відображену відносно площини дзеркала. Водночас буфер маски дозволить обмежити виведення формою проєкції 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 проєкції 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):

```
void Reflection(GLfloat point[3], GLfloat normal[3], GLfloat matrix[4])
{ GLfloat p= point[0]*normal[0] + point[1]*normal[1] + point[2]*normal[2];
  matrix[0][0] = 1 -2 * normal[0] * normal[0];
  matrix[1][0] = 0 -2 * normal[0] * normal[1];
  matrix[2][0] = 0 -2 * normal[0] * normal[2];
  matrix[3][0] = 2 * p * normal[0];
  matrix[0][1] = 0 -2 * normal[0] * normal[1];
  matrix[1][1] = 1 -2 * normal[1] * normal[1];
  matrix[2][1] = 0 -2 * normal[1] * normal[2];
  matrix[3][1] = 2 * p * normal[1];
  matrix[0][2] = 0 -2 * normal[0] * normal[2];
  matrix[1][2] = 0 -2 * normal[1] * normal[2];
  matrix[2][2] = 1 -2 * normal[2] * normal[2];
  matrix[3][2] = 2 * p * normal[2];
  matrix[0][3] = 0; matrix[1][3] = 0;
  matrix[2][3] = 0; matrix[3][3] = 1;
}
```

Налаштовуємо буфер маски на малювання тільки в областях, де значення буферу дорівнює 1:

```
// умова виконана і тест проходить тільки якщо значення в буфері маски 1
glStencilFunc (GL_EQUAL, 1, 0xFFFFFFF);
// 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.

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.

15. Оптимізація програм

Для мінімізації часу формування та виведення зображень необхідно притримуватися багатьох класичних рекомендацій з програмування на OpenGL: інтенсивно використовувати заздалегідь підготовлені списки команд, векторну форму завдання параметрів функцій, перехоплення повідомлень Windows об'єктами 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 цього посібника. Наведемо лише окремі поради і прийоми з підвищення надійності і продуктивності графічних застосунків на основі використання бібліотеки OpenGL.

15.1. 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 візуалізації 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 проєкційну матрицю. У разі використання 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 цієї 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, у разі накопичення чотирьох зображень масштабуйте кожне зображення `1/4` від об'єму зображень, що накопичуються;
- якщо необхідна точна двовимірна растеризація, то треба ретельно визначити ортогональну проєкцію і вершини примітивів, які необхідно растеризувати.

Ортогональна проєкція повинна бути визначена з цілочисельними координатами: gluOrtho2D(0, width, 0, height), де width та height — розміри області перегляду. Для растеризації дані проєкційної 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 правильне відсікання, а також може робити помилки під час 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 протягом всієї операції. Можливо, що параметри команди glRotated, будуть перетворені у інші типи даних перед обробкою.

15.2. 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 рідкісних змін чи у разі швидкої зміни 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 під час застосування 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. Giloi W. Interactive computer graphics: data structures, algorithms, languages. Englewood Cliffs, N.J : Prentice-Hall, 1978. 354 p.
2. Rogers D. F. Mathematical elements for computer graphics. 2nd ed. New York : McGraw, 1989. 611 p.
3. Геометрическое моделирование и машинная графика в САПР : підручник / В. Е. Михайленко та ін. Київ : Вища школа, 1991. 374 с.
4. Краснов М. В. Open GL. Графика в проектах Delphi. Санкт-Петербург; Дюссельдорф : BHV, 2002. 352 с.
5. Тихомиров Ю. Программирование трехмерной графики. СПб. : БХВ – Санкт-Петербург, 1998. 256 с.
6. Шикин Е. В., Боресков А. В. Компьютерная графика. Полигональные модели. Москва : ДИАЛОГ-МИФИ, 2001. 464 с.
7. Геометричне моделювання і комп'ютерна графіка: використання бібліотеки OpenGL : навчальний посібник / А. А. Лященко та ін. Київ : КНУБА, 2009. 90 с.
8. Пічугін М., Канкін І., Воротніков В. Комп'ютерна графіка : навчальний посібник. Київ : Центр навчальної літератури, 2019. 346 с.

Алфавітний покажчик

A

API ... 56

D

DIP ... 61

G

GDI ... 56
GLU ... 66
GLUT ... 66
GLX ... 66

H

Handle ... 56

M

Microsoft Visual C++ ... 56

O

OpenGL ... 52, 53, 64

А

альфа-змішування ... 116

Б

буфер
- глибини ... 115
- кадру ... 115
- кольору ... 115
  - робочий ... 115
  - фоновий ... 115
- маски (трафарету) ... 115, 118, 127
- накопичувач (акумулятор) ... 115, 117
- очистка ... 74

В

вершина ... 67, 75
- атрибут ... 67, 75
- масив ... 91
- нормаль ... 75

Г

геометричне моделювання ... 7
- комп'ютерна графіка ... 7
- обробка зображень ... 7
- розпізнавання зображень ... 7

грань ... 79
- зворотна ... 79
- лицьова ... 79

Д

декомпозиція ... 14
дисплейний список ... 90

Е

Ейлерові кути ... 29
- кут власного обертання ... 30
- кут нутації ... 30
- кут прецесії ... 30
- лінія вузлів ... 30

З

зафарбовування
- за методом Гуро ... 54
- за методом Фонга ... 55
- інтерполяційне ... 53
- плоске ... 52

змішування кольорів (blending) ... 115

К

кадрування ... 11, 45
картинна площина ... 45

кватерніон 30
команди GDI
- Arc 59
- BeginPaint 57
- Chord 59
- CreateDIPPatternBrushPt 61
- CreateHatchBrush 61
- CreatePatternBrush 61
- CreatePen 60
- CreateSolidBrush 61
- DeleteObject 61, 62
- Ellipse 58
- EndPaint 57
- GetDC 57
- GetPixel 57
- LineTo 58
- MoveToEx 58
- Pie 59
- PolyBezier 60
- Polygon 60
- Polyline 60
- Rectangle 58
- ReleaseDC 57
- RoundRect 59
- SelectObject 61, 62
- SetPixel 57

команди GL
- glAccum 118
- glArrayElement 92
- glBegin 77
- glBindTexture 109
- glBlendFunc 116
- glCallList 90
- glCallLists 90
- glClear 74
- glClearColor 75
- glColor 76, 105
- glColorMaterial 102
- glColorPointer 92
- glCullFace 80
- glDeleteLists 91
- glDepthRange 99
- glDisable 77
- glDisableClientState 92
- glDrawArrays 92
- glDrawBuffer 118
- glDrawElements 92
- glEnable 77
- glEnableClientState 92
- glEnd 77
- glEndList 90
- glEvalCoord1 86
- glEvalCoord2 87
- glEvalMesh1 86
- glEvalMesh2 87
- glFog 105
- glFrontFace 79
- glGenTextures 109
- glHint 120
- glLight 103, 105
- glLightModel 100
- glLoadIdentity 95
- glLoadMatrix 95
- glLookAt 105
- glMap1 85
- glMap2 87
- glMapGrid1 86
- glMapGrid2 87
- glMaterial 101
- glMatrixMode 94
- glMultMatrix 95
- glNewList 90
- glNormal 76
- glNormalPointer 91
- glOrtho 97
- glPolygonMode 79
- glPolygonStipple 80
- glPopMatrix 95
- glPushMatrix 95
- glReadBuffer 117
- glRotate 96
- glScale 96
- glShadeModel 76
- glStencilFunc 118
- glStencilOp 119
- glTexCoord 112
- glTexEnv 111
- glTexGen 112
- glTexImage2D 108
- glTexParameter 110
- glTranslate 96
- gluLookAt 96
- glVertex 75
- glVertexPointer 91
- glViewPort 99

команди GLU
- gluBeginNurbsCurve 88
- gluBeginNurbsSurface 88
- gluBeginTrim 89
- gluBuild2DMipmaps 108

gluCylinder 82
gluDeleteNurbsRenderer 88
gluDeleteQuadric 82
gluDisk 83
gluEndNurbsCurve 88
gluEndNurbsSurface 88
gluEndTrim 89
gluNewNurbsRenderer 88
gluNewQuadric 82
gluNewTess 84
gluNurbsCurve 88
gluNurbsProperty 88
gluNurbsSurface 88
gluOrtho2D 97
gluPartialDisk 83
gluPerspective 98
gluPwlCurve 89
gluQuadricDrawStyle 82
gluQuadricNormals 82
gluQuadricOrientation 82
gluQuadricTexture 112
gluScaleImage 107
gluSphere 82
gluTessCallBack 84
команди GLUT
glutDisplayFunc 74
glutSolidCone 83
glutSolidCube 83
glutSolidDodecahedron 84
glutSolidIcosahedron 84
glutSolidOctahedron 84
glutSolidSphere 83
glutSolidTetrahedron 84
glutSolidTorus 83
glutWireCone 83
glutWireCube 83
glutWireDodecahedron 84
glutWireIcosahedron 84
glutWireOctahedron 84
glutWireSphere 83
glutWireTetrahedron 84
glutWireTorus 83
конвеєр OpenGL
режим роботи 77
константи GL
GL_ACCUM 118
GL_ACCUM_BUFFER_BIT 74
GL_ADD 118
GL_ALWAYS 119
GL_AMBIENT 101, 103
GL_AMBIENT_AND_DIFFUSE 102
GL_BACK 79, 80, 117
GL_BLEND 116
GL_BYTE 91
GL_CCW 79
GL_CEQUAL 119
GL_CLAMP 111
GL_COLOR_ARRAY 92
GL_COLOR_BUFFER_BIT 74
GL_COLOR_MATERIAL 102
GL_COMPILE 90
GL_COMPILE_AND_EXECUTE 90
GL_CONSTANT_ATTENUATION 104
GL_CREATE 119
GL_CULL_FACE 80
GL_CW 79
GL_DECR 120
GL_DEPHT_TEST 84
GL_DEPTH_BUFFER_BIT 74
GL_DIFFUSE 101, 103
GL_DONT_CARE 121
GL_DOUBLE 91
GL_DST_ALPHA 116
GL_DST_COLOR 116
GL_DST_ONE_MINUS_ALPHA 116
GL_EMISSION 102
GL_EQUAL 119
GL_EYE_LINEAR 113
GL_EYE_PLANE 113
GL_FALSE 100
GL_FASTEST 121
GL_FILL 80
GL_FLAT 76, 133
GL_FLOAT 91
GL_FOG_COLOR 106
GL_FOG_DENSITY 106
GL_FOG_END 106
GL_FOG_HINT 120
GL_FOG_MODE 105
GL_FOG_START 106
GL_FRONT 79, 80, 117
GL_FRONT_AND_BACK 79
GL_INCR 120
GL_INT 91, 107
GL_INVERT 120
GL_KEEP 120
GL_LEQUAL 119
GL_LESS 119
GL_LIGHT_MODEL_AMBIENT 101
GL_LIGHT_MODEL_LOCAL_VIEWER 100

GL_LIGHT_MODEL_TWO_SIDE............ 100
GL_LIGHTi 103, 105
GL_LIGHTING 105
GL_LINE.. 79
GL_LINE_LOOP...................................... 77
GL_LINE_SMOOTH_HINT..................... 121
GL_LINE_STRIP...................................... 77
GL_LINEAR 111
GL_LINEAR_ATTENUATION 104
GL_LINES...................................... 77, 133
GL_LOAD 118
GL_LUMINANCE.................................. 108
GL_MAP1_COLOR_4.............................. 86
GL_MAP1_INDEX.................................. 86
GL_MAP1_NORMAL 86
GL_MAP1_TEXYURE_COORD_1 86
GL_MAP1_TEXYURE_COORD_2 86
GL_MAP1_TEXYURE_COORD_3 86
GL_MAP1_TEXYURE_COORD_4 86
GL_MAP1_VERTEX_3............................ 85
GL_MAP1_VERTEX_4............................ 85
GL_MAP2_NORMAL 89
GL_MAP2_VERTEX_3....................... 87, 89
GL_MAP2_VERTEX_4....................... 87, 89
GL_MAX_LIGHT................................... 103
GL_MODELVIEW 94
GL_MODULATE................................... 112
GL_MULT 118
GL_NEAREST 111
GL_NEVER 119
GL_NICEST.. 121
GL_NORMAL_ARRAY 92
GL_NORMALIZE 76, 100, 133
GL_NOTEQUAL.................................... 119
GL_OBJECT_LINEAR 113
GL_OBJECT_PLANE 113
GL_ONE_MINUS_DST_COLOR............. 116
GL_ONE_MINUS_SRC_COLOR............. 116
GL_PERSPECTIVE_CORRECTION_HINT . 121
GL_POINT... 79
GL_POINT_SMOOTH_HINT.................. 121
GL_POINTS... 77
GL_POLYGON............................... 78, 133
GL_POLYGON_SMOOTH_HINT............ 121
GL_POLYGON_STIPPLE 81
GL_POSITION 104
GL_PROJECTION............................. 94, 98
GL_QUAD_STRIP 78
GL_QUADRATIC_ATTENUATION.......... 104
GL_QUADS 78, 133
GL_REPEAT 111
GL_REPLACE 112, 120
GL_RETURN 118
GL_RGB...................................... 107, 108
GL_RGBA.................................... 107, 108
GL_S 112
GL_SHININESS..................................... 102
GL_SHORT....................................... 91, 107
GL_SMOOTH.. 76
GL_SPECULAR 101, 103
GL_SPHERE_MAP 113
GL_SPOT_CUTOFF 103
GL_SPOT_DIRECTION 104
GL_SPOT_EXPONENT 103
GL_SRC_ALPHA.................................... 116
GL_SRC_COLOR 116
GL_SRC_ONE_MINUS_ALPHA 116
GL_STENCIL_BUFFER_BIT....................... 74
GL_STENCIL_TEST............................... 120
GL_T 112
GL_TEXTURE 94
GL_TEXTURE_1D.......................... 109, 111
GL_TEXTURE_2D.................. 108, 109, 111
GL_TEXTURE_ENV 112
GL_TEXTURE_ENV_MODE................... 112
GL_TEXTURE_GEN_MODE................... 113
GL_TEXTURE_GEN_S 113
GL_TEXTURE_GEN_T 113
GL_TEXTURE_MAG_FILTER 111
GL_TEXTURE_MIN_FILTER................... 111
GL_TEXTURE_WRAP_S 111
GL_TEXTURE_WRAP_T 111
GL_TRIANGLE_FAN................................ 77
GL_TRIANGLE_STRIP.............................. 77
GL_TRIANGLES.............................. 77, 133
GL_TRUE 100, 112
GL_UNSIGNED_BYTE 91, 92, 107
GL_UNSIGNED_INT.......................... 91, 92
GL_UNSIGNED_SHORT 92
GL_VERTEX_ARRAY 92
GL_ZERO 120
константи GLU
GLU_MAP1_TRIM_2.............................. 89
GLU_MAP1_TRIM_3.............................. 89
контекст пристрою 56
координати 10
абсолютні... 13
відносні.. 13
користувацькі 13
нормовані 13

однорідні 15
приладові 10
світові 10

М

матриця
афінного перетворення 24
вироджена 47
відображення 24, 26, 129
масштабування 24, 26
модельно-видова 94
одинична 95
переміщення 23, 26
повороту 24, 26, 30
проєктування 47
проєкцій 94
текстури 94
тіні 125
модель
кольору
CMY 32
CMYK 32
HLS 35
HSB 33
HSI 35
HSV 33
RGB 32
YIQ 33

О

операторні дужки 77

П

перетворення
афінне 14, 20
відображення 22
власне 20
еквіафінне 20
масштабування 22
невласне 20
переміщення 21
поворот 22
центроафінне 20
модельно-видове
масштабування 96
переміщення 96
поворот 96
піксель 11
подвійна буферизація 115, 117
примітив 9
атомарний *Див.* вершина
відрізок 77
геометричний 9
графічний 9
тип 77
трикутник 77
чотирикутник 78
проєктування 45
паралельне 46
центральне 45
проєкція
паралельна 46
аксонометрична 47
диметрія 48
ізометрія 48
триметрія 48
косокутна 48
вільна 49
кабінетна 49
ортографічна 47, 97
перспективна 49, 97, 98
простір
проєктивний 16
розмірність 10

Р

растеризація 115

С

система координат 10
віконна 94
глобальна 13
лівобічна 94, 97
локальна 14
правобічна 94
приладова 10
світова 10
смуга Маха 54

Т

текстура 107
рівень деталізації 108

текстурні об'єкти 109
типи даних GLU
GLUnurbsObj 88
GLUtesslator 84
точка
ідеальна 17
сходу.. 49
туман... 115

Ф

фрактал 37
IFS... 41
алгебраїчний
атрактор 39
множина Мандельброта 40
геометричний.......................................37
генератор 38
дракон Хартера-Хейтуея..................39
предфрактал 38
тріадна крива Коха 38
система ітерованих функцій 41
стохастичний 37
фрактальна геометрія 37
фрактальне стиснення інформації 41

Ц

ЦМО 8

Навчальне видання

БОРОДАВКА ЄВГЕНІЙ ВОЛОДИМИРОВИЧ

ТЕРЕНЬЄВ ОЛЕКСАНДР ОЛЕКСАНДРОВИЧ

КОМП'ЮТЕРНА ГРАФІКА

НАВЧАЛЬНИЙ ПОСІБНИК