МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ БУДІВНИЦТВА І АРХІТЕКТУРИ

А.А. Лященко

В.В. Демченко

Є.В. Бородавка

В.В. Смирнов

ГЕОМЕТРИЧНЕ МОДЕЛЮВАННЯ І КОМП'ЮТЕРНА ГРАФІКА: ВИКОРИСТАННЯ БІБЛІОТЕКИ OPENGL

Рекомендовано науково-методичною радою
Київського національного університету будівництва і архітектури
як навчальний посібник для студентів спеціальності
6.080402 "Інформаційні технології проектування"

Київ 2008

УДК 004.92
ББК 32.973
Л19

Рецензенти: О.С. Городецький, доктор технічних наук, професор, заступник директора ДНДІАСБ з наукової роботи

В.О. Анпілогова, канд. техн. наук, професор, професор кафедри нарисної геометрії, інженерної та машинної графіки КНУБА

Затверджено на засіданні науково-методичної ради Київського національного університету будівництва і архітектури, протокол № 1 від 23 вересня 2008 року.

Лященко А.А., Демченко В.В., Бородавка Є.В., Смирнов В.В.

Л19 Геометричне моделювання і комп'ютерна графіка: використання бібліотеки OpenGL: Навчальний посібник. — К.: КНУБА, 2008. — 76 с.

Навчальний посібник містить необхідні теоретичні і довідкові матеріали про графічну бібліотеку OpenGL, а також методичні рекомендації щодо її практичного використання для розробки прикладних програм. Розглянуто приклади роботи з бібліотекою в середовищах програмування Delphi та C++ Builder, основні прийоми оптимізації застосувань.

Призначений для студентів спеціальності 6.080402 "Інформаційні технології проектування".

УДК 004.92
ББК 32.973

© Лященко А.А., Демченко В.В.,
Бородавка Є.В., Смирнов В.В., 2008
© КНУБА, 2008

Зміст

Вступ

Дисципліні «Геометричне моделювання і комп'ютерна графіка» належить особлива роль в навчальному плані підготовки фахівців спеціальності «Інформаційні технології проектування». Адже об'єктами професійної діяльності випускників спеціальності є розробка, модернізація та експлуатація систем автоматизованого проектування (САПР), характерною рисою яких є широке використання методів і засобів інтерактивної комп'ютерної графіки [1].

В робочій навчальній програмі дисципліни передбачене вивчення ключових тем геометричного моделювання і комп'ютерної графіки з використанням сучасних інтегрованих середовищ програмування і стандартної графічної бібліотеки *OpenGL* — одного з найпопулярніших прикладних програмних інтерфейсів (API – Application Programming Interface) для розробки застосувань реалістичної двовимірної та тривимірної графіки.

Стандарт *OpenGL* (Open Graphics Library – відкрита графічна бібліотека) був розроблений і затверджений у 1992 році провідними фірмами в сфері індустрії програмного забезпечення як ефективний апаратно-незалежний інтерфейс, придатний для реалізації на різних платформах. Основою стандарту стала бібліотека *IRIS GL*, розроблена фірмою Silicon Graphics Inc.

На сьогоднішній день графічна система OpenGL підтримується більшістю виробників апаратних і програмних платформ. Ця система доступна тим, хто працює в середовищі Windows, користувачам комп'ютерів Apple. Вільно розповсюджувані коди системи Mesa (пакет API на базі OpenGL) можна компілювати в більшості операційних систем, у тому числі в Linux.

Причиною широкого розповсюдження і розвитку OpenGL як графічного стандарту є такі його ключові характеристики:

- *Стабільність*. Доповнення і зміни в стандарті реалізуються таким чином, щоб зберегти сумісність з розробленим раніше програмним забезпеченням.
- *Надійність і простота перенесення на інші апаратно-програмні платформи*. Застосування, що використовують OpenGL, гарантують однаковий візуальний результат в різних операційних системах та на різному обладнанні. Крім того, ці застосування можуть виконуватися як на персональних комп'ютерах, так і на робочих станціях чи суперкомп'ютерах.

- *Легкість використання*. Стандарт OpenGL має продуману структуру та інтуїтивно зрозумілий інтерфейс, що дозволяє з меншими витратами створювати ефективні застосування, які містять менше рядків коду, ніж створені з використанням інших графічних бібліотек. Необхідні функції для забезпечення сумісності з різним обладнанням реалізовані на рівні бібліотеки і значно спрощують розробку застосувань.

Реалізація бібліотеки OpenGL в операційній системі Windows містить більше 400 різних команд для опису об'єктів і операцій в процесі створення інтерактивних графічних застосувань [2], що, природно, ускладнює її вивчення за експлуатаційною документацією. Метою авторів посібника є викладення основ використання OpenGL як необхідної передумови для наступного поглиблення теоретичних знань і практичних навичок з створення графічних застосувань.

При підготовці цього посібника, крім експлуатаційної документації та наукової і довідкової літератури, використано навчально-методичний посібник Ю.М. Баяковського та інших «Графическая библиотека OpenGL» [3], який пройшов апробацію на факультеті обчислювальної математики і кібернетики Московського державного університету і містить опис базових можливостей OpenGL та практичні рекомендації щодо прийомів ефективної роботи з бібліотекою.

Наведені в [3] приклади програм суттєво змінені. З урахуванням особливостей організації комп'ютерного забезпечення навчального процесу в університеті, програмні реалізації прикладів подані для двох інструментальних середовищ програмування (C++ Builder та Delphi). Відмінності в синтаксисі мов програмування цих середовищ розробки лише ілюструють окремі практичні аспекти використання можливостей програмного інтерфейсу стандарту OpenGL, основна ж увага в посібнику приділяється вивченню методології побудови графічних програм та основних прийомів роботи з бібліотекою.

Посібник складається з восьми розділів та трьох додатків.

Основні розділи посібника містять систематизований опис структури і абстрактних моделей функціонування та використання бібліотеки, синтаксису і семантики основних команд, методичні вказівки щодо прийомів підвищення ефективності графічних програм. У всіх розділах наведено запитання для контролю знань.

В додатках наведено опис розширень базової бібліотеки та приклади демонстраційних програм.

Розділ 1. Основи OpenGL

1.1 Основні можливості

Основне призначення OpenGL — відображення двовимірних та тривимірних об'єктів у статичних та динамічних сценах. Об'єкти представляються у вигляді сукупності вершин (для геометричних фігур) або пікселів (для растрових зображень). OpenGL спочатку перетворює вихідні 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:

- *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.
- *Функції опису джерел світла* служать для опису положення і параметрів джерел світла, розташованих у тривимірній сцені.
- *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* виводу на екран. Як атрибути OpenGL використовує колір, характеристики матеріалу, текстури, параметри освітлення.
- *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.

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

1.2 Функціональна модель графічних застосувань на основі OpenGL

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

Узагальнена функціональна модель прикладних графічних застосувань на основі OpenGL може бути представлена в вигляді ієрархії обробних систем [4] $S_k=(L_k, I_k)$, де L_k — мова, I_k – інтерпретатор мови L_k в мову L_{k+1} наступної обробної системи S_{k+1}.

У першому наближенні (на початкових стадіях етапу проектування застосування) функціональна модель визначається трійкою $\{(L_i, I_i), (L_m, I_m), (L_o, I_o)\}$. 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_o (інтерфейс прикладного програмування для генерації команд і списків команд OpenGL). Інтерпретатор I_o (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 застосувань (системи автоматизації проектування, дизайну, геометричного моделювання явищ і процесів тощо) є досить трудомісткою.

Природньо, що в об'єктно-орієнтованих середовищах програмування у розробників виникає спокуса створення класів-оболонок для інкапсу-

ляції 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_o 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 і управління параметрами сцен.

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

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

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

Крім того, функції, специфічні для конкретної віконної підсистеми, звичайно входять у її прикладний програмний інтерфейс. Так, функції, що підтримують виконання OpenGL, є в складі *Win32 API* і *X Window*. На рис.

1.1 схематично представлена організація системи бібліотек у версії, що працює під управлінням системи Windows. Аналогічна організація використовується й в інших версіях OpenGL.

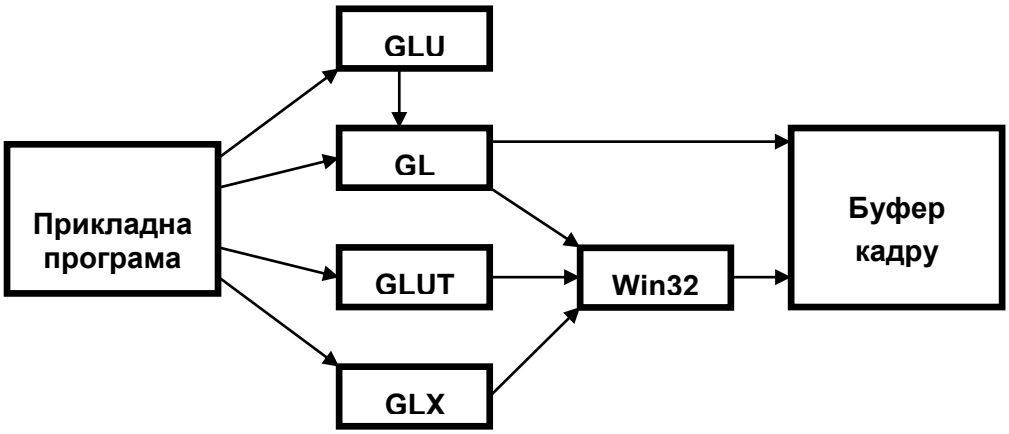


Рис. 1.1. Організація бібліотеки OpenGL

1.4 Архітектура OpenGL

Функції OpenGL реалізовані з використанням моделі клієнт-сервер. Застосування (прикладна програма) виступає в ролі клієнта — воно 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. 1.2).

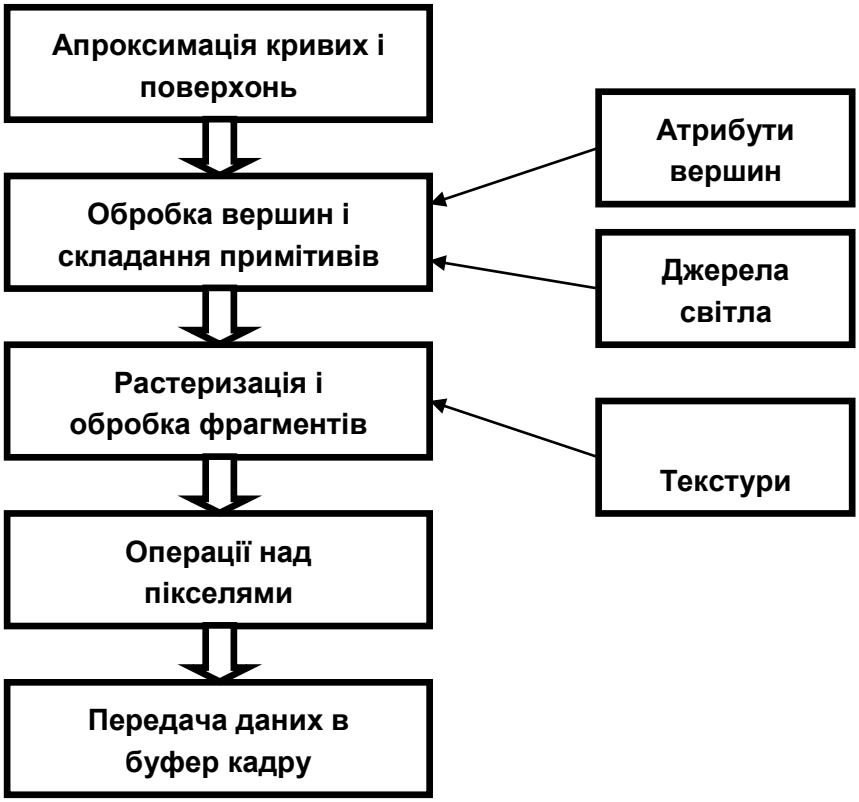


Рис. 1.2. Схема функціонування конвеєра OpenGL

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

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

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

Крім того, OpenGL можна розглядати як скінченний автомат [7], стан якого визначається множиною значень спеціальних змінних і значеннями поточної нормалі, кольору, координат текстури й інших атрибутів і ознак. Уся ця інформація буде використана при надходженні в графічну систему координат вершини для побудови фігури, до якої вона належить. Зміна

станів відбувається за допомогою команд, що оформлюються як виклики функцій.

1.5 Синтаксис команд

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

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

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

Наприклад, у секції *interface* заголовного файлу знаходиться наступне випереджальне оголошення функції очищення екрану:

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

В секції *imlementation* модуля опис має наступний вигляд:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

1.6 Приклад простого застосування

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

В проекті Delphi список *uses* доповнений посиланням на модуль OpenGL.pas — це програміст повинен зробити самостійно.

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

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

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

Розділ *private* опису класу форми містить два рядки:

В проекті Delphi	В проекті C++ Builder	
`glDC: HDC;`	`HDC glDC;`	`// посилання на контекст пристрою`
`hrc: HGLRC;`	`HGLRC hrc;`	`// посилання на контекст відображення`

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

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

Обробник події *OnCreate* форми містить наступні рядки:

В проекті Delphi	В проекті C++ Builder

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

Перший рядок — отримання контексту пристрою з головної форми проекту. Наступний рядок — звернення до описаної у цьому ж модулі процедури користувача, яка задає формат пікселя:

В проекті Delphi	В проекті C++ Builder

`procedure SetDCPixelFormat(hdc:HDC);`
`var pfd : TPixelFormatDescriptor;`
`nPixelFormat : Integer;`
`begin`
`FillChar (pfd, SizeOf(pfd), 0);`
`nPixelFormat:=`
`ChoosePixelFormat(hdc, @pfd);`
`SetPixelFormat(hdc, nPixelFormat,`
`@pfd);`
`end;`
	`void SetDCPixelFormat(HDC hdc)`
	`{ PIXELFORMATDESCRIPTOR pfd, *ppfd;`
	`int nPixelFormat;`
	`ppfd = &pfd;`
	`ppfd->nSize=`
	`sizeof(PIXELFORMATDESCRIPTOR);`
	`ChoosePixelFormat(hdc, ppfd)`
	`SetPixelFormat(hdc, nPixelFormat,`
	`ppfd)`
	`}`

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

Після заповнення полів структури *TPixelFormatDesriptor*, ми визначаємо свої власні вимоги до графічної системи, на якій буде виконуватись програма, а OpenGL підбирає найбільш відповідний формат і встановлює його у якості формату пікселя для подальшої роботи. Але OpenGL не до-

зволить встановити нереальний для конкретного робочого місця формат. Те, як ми задамо значення прапорців у полі структури *dwFlags*, може суттєво позначитись на роботі програми і тому випадково задавати ці значення небажано. Більш детально ознайомитись зі структурою *TPixelFormatDesriptor* можна за допомогою файлу довідки та файлу *windows.pas* або в заголовному файлі *wingdi.h*.

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

Обробник події *OnPaint* наведений нижче:

В проекті Delphi	**В проекті C++ Builder**

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

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

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

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

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

Повний текст модуля головної форми в проекті Delphi:

```
unit ufGL;
interface
uses
  Windows, Messages, Forms, Classes, Controls, ExtCtrls, ComCtrls,
  StdCtrls, Dialogs, SysUtils, OpenGL;
type
  TfrmGL = class(TForm)
    procedure FormCreate(Sender: TObject);
    procedure FormPaint(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
  private
    glDC: HDC;
    hrc: HGLRC;
  end;
var
  frmGL: TfrmGL;
implementation
{$R *.DFM}
procedure TfrmGL.FormPaint(Sender: TObject);
var  ps : TPaintStruct;
begin
  BeginPaint(glDC, ps);
  glClearColor (0.5, 0.5, 0.75, 1.0);
  glClear (GL_COLOR_BUFFER_BIT);
  glColor3f(1.0,0.0,0.0);
  glBegin(GL_QUADS);
    glVertex2f(0.5, 0.5);
    glVertex2f(-0.5,0.5);
    glVertex2f(-0.5, -0.5);
    glVertex2f(0.5, -0.5);
  glEnd;
  SwapBuffers(glDC);
  EndPaint(glDC, ps);
end;
procedure SetDCPixelFormat (hdc : HDC);
var
  pfd : TPixelFormatDescriptor;
  nPixelFormat : Integer;
begin
  FillChar (pfd, SizeOf (pfd), 0);
  nPixelFormat := ChoosePixelFormat (hdc, @pfd);
  SetPixelFormat (hdc, nPixelFormat, @pfd);
end;
procedure TfrmGL.FormCreate(Sender: TObject);
begin
  glDC := GetDC(Handle);
```

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

procedure TfrmGL.FormDestroy(Sender: TObject);
begin
  wglMakeCurrent (0, 0);
  wglDeleteContext(hrc);
  ReleaseDC(Handle, glDC);
  DeleteDC(glDC);
end;
end.
```

Проект в C++ Builder пропонується студенту зробити самостійно на основі приведених вище фрагментів коду.

Контрольні питання

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?

Розділ 2. Формування зображень геометричних об’єктів

2.1 Процес оновлення зображення

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

Звичайно застосування 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 застосування про те, що вміст вікна необхідно перемалювати (наприклад, якщо вікно було перекрито іншим). Створюване зображення може бути як статичним, так і анімованим, тобто залежати від деяких параметрів, що змінюються з часом. У цьому випадку краще викликати функцію оновлення самостійно.

Як правило типова функція оновлення зображення забезпечує:

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

Очищення буферів забезпечується командою:

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

Ця команда очищує задані буфери з поточним значенням (див. розділ 6). 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 і повинні належати відрізку [0,1]. Четвертий параметр задає так

звану альфа-компоненту (див. п. 6.1). Як правило, вона дорівнює 1. За замовчуванням колір — чорний (0,0,0,1).

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

Установка положення спостерігача і перетворення тривимірних об'єктів (поворот, зсув і т.д.) контролюються за допомогою завдання матриць перетворень. Перетворення об'єктів і налагодження положення віртуальної камери описані в розділі 3.

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

2.2 Вершини і примітиви

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

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

2.2.1 Положення вершини в просторі

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

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

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

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

2.2.2 Колір вершини

Для завдання поточного кольору вершини використовуються команди:

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

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

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

Для керування режимом інтерполяції використовується команда

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

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

2.2.3 Нормаль

Визначити нормаль у вершині можна командами

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

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

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

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

Відзначимо, що команди

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

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

2.3 Операторні дужки glBegin/glEnd

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

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

Параметр *mode* визначає тип примітива, який задається всередині і може приймати наступні значення (рис. 2.1):

GL_POINTS	кожна вершина задає координати деякої 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 тім, 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

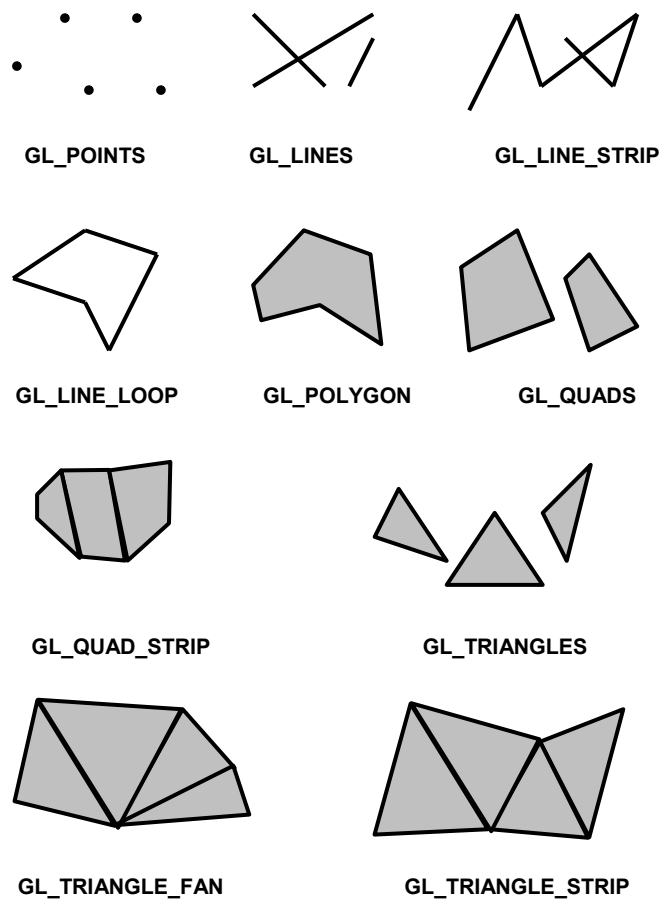


Рис. 2.1. Примітиви *OpenGL*

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

В проекті Delphi	**В проекті C++ Builder**

const
  BlueColor : **array** [0..2] **of** GLfloat = (0,0,1);
…
glBegin(GL_TRIANGLES);
glColor3f(1.0, 0.0, 0.0);//*червоний*
glVertex3f(0.0, 0.0, 0.0);
glColor3ub(0,255,0); // *зелений*
glVertex3f(1.0, 0.0, 0.0);
glColor3fv(@BlueColor); //*синій*
glVertex3f(1.0, 1.0, 0.0);
glEnd;
	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 самих багатокутників, можна визначити і метод їх відображення на екрані. Однак спочатку треба визначити поняття лицьових і зворотних граней.

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

glFrontFace (*mode*: GLenum)

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

Для зміни методу відображення багатокутника використовується команда

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

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

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

Параметр *mode* може дорівнювати:

GL_POINT	відображення тільки вершин багатокутників
GL_LINE	багатокутники будуть представлятися набором відрізків
GL_FILL	багатокутники будуть зафарбовуватися поточним кольором з урахуванням освітлення (цей режим встановлений за замовчуванням)

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

glCullFace (*mode*: GLenum)

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

Крім розглянутих стандартних примітивів, у бібліотеках GLU і GLUT описані і більш складні фігури, такі як сфера, циліндр, диск (в GLU) і сфе-

ра, 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 5).

Наприклад, щоб намалювати сферу чи циліндр, введемо змінну спеціального типу:

```
quadObj : GLUquadricObj;
```

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

```
quadObj := gluNewQuadric;
```

а потім викличемо відповідну команду:

```
gluSphere (quadObj, radius: GLdouble, slices: GLint, stacks: GLint)
gluCylinder (quadObj, baseRadius: GLdouble, topRadius: GLdouble, height:
                  GLdouble, slices: GLint, stacks: GLint)
```

де параметр *slices* задає число розбивок навколо вісі *Z*, а *stacks* — уздовж вісі *Z*.

Більш докладну інформацію про ці та інші команди побудови примітивів можна знайти в *дод. В*.

2.4 Дисплейні списки

Якщо ми кілька разів звертаємося до однієї і тієї ж групи команд, то їх можна об’єднати в так званий дисплейний список (*display list*), і викликати його за необхідності. Для того, щоб створити новий дисплейний список, треба помістити всі команди, що повинні в нього ввійти, між наступними операторними дужками:

```
glNewList (list: GLuint, mode: GLenum)
glEndList
```

Для ідентифікації списків використовуються цілі позитивні числа, що задаються при створенні списку значенням параметра *list*, а параметр *mode* визначає режим обробки команд, що входять у список:

GL_COMPILE	команди записуються в список без виконання
GL_COMPILE_AND_EXECUTE	команди спочатку виконуються, а потім записуються в список

Створений список можна викликати командою

```
glCallList (list: GLuint)
```

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

```
glCallLists (n: GLsizei, type: GLenum, lists: pointer)
```

яка викликає *n* списків з ідентифікаторами з масиву *lists*, тип елементів якого вказується в параметрі *type*. Це можуть бути типи GL_BYTE, GL_UNSIGNED_BYTE, GL_SHORT, GL_INT, GL_UNSIGNED_INT і деякі інші. Для видалення списків використовується команда

glDeleteLists (*list*: GLint, *range*: GLsizei)

яка видаляє списки з ідентифікаторами *ID* з діапазону *list* ≤ *ID* ≤ *list+range-1*.

Приклад:

```
glNewList(1, GL_COMPILE);
  glBegin(GL_TRIANGLES);
    glVertex2f(0.0, 0.0);
    glVertex2f(1.0, 0.0);
    glVertex2f(0.0, 1.0);
  glEnd;
glEndList;
…
glCallList(1);
```

Дисплейні списки зберігаються в пам'яті сервера в оптимальному, скомпільованому виді, 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.

2.5 Масиви вершин

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

glVertexPointer (*size*: GLint, *type*: GLenum, *stride*: GLsizei, *ptr*: pointer)

яка визначає спосіб відображення і координати вершин. Параметр *size* задає кількість координат вершини (2, 3 або 4), а *type* визначає тип даних (GL_SHORT, GL_INT, GL_FLOAT або GL_DOUBLE). Іноді зручно зберігати в одному масиві інші атрибути вершини, тоді параметр *stride* задає зсув від координат однієї вершини до координат наступної; якщо *stride* дорівнює нулю, це значить, що координати розташовані послідовно. У параметрі *ptr* вказується адреса, за якою знаходяться дані. Аналогічно командами

glNormalPointer (*type*: GLenum, *stride*: GLsizei, *ptr*: pointer)
glColorPointer (*size*: GLint, *type*: GLenum, *stride*: GLsizei, *ptr*: pointer)

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

glEnableClientState (*array*: GLenum)

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

glDisableClientState (*array*: GLenum)

з відповідним значенням параметра *array.*

Для відображення вмісту масивів використовується команда

glArrayElement (*index*: GLint)

яка передає OpenGL атрибути вершини, використовуючи елементи масиву з номером *index*. Це аналогічно послідовному застосуванню команд виду `glColor(…)`, `glNormal(…)`, `glVertex(…)` з відповідними параметрами. Однак частіше застосовується команда

glDrawArrays (*mode*: GLenum, *first*: GLint, *count*: GLsizei)

яка малює *count* примітивів, визначених параметром *mode*, використовуючи елементи з масивів з індексами від *first* до *first+count-1*. Це еквівалентно виклику послідовності команд `glArrayElement` з відповідними індексами.

У випадку, якщо одна вершина входить у кілька примітивів, то замість дублювання її координат у масиві зручно використовувати її індекс. Для цього треба викликати команду

glDrawElements (*mode*: GLenum, *count*: GLsizei, *type*: GLenum, *indices*: pointer)

де *indices* — це масив номерів вершин, які треба використовувати для побудови примітивів, *type* визначає тип елементів цього масиву: `GL_UNSIGNED_BYTE`, `GL_UNSIGNED_SHORT`, `GL_UNSIGNED_INT`, а *count* задає їх кількість.

Слід зауважити, що використання масивів вершин дозволяє оптимізувати передачу даних на сервер OpenGL, і, як наслідок, підвищити швидкість малювання тривимірної сцени. Такий метод визначення примітивів є одним з найшвидших і добре підходить для візуалізації великих обсягів даних.

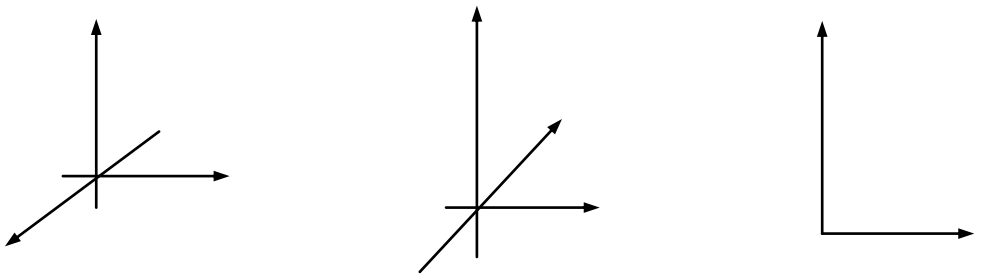
Контрольні питання

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?

Розділ 3. Перетворення об’єктів

В OpenGL використовуються як основні три системи координат: лівобічна, правобічна і віконна (рис. 3.1). Перші дві системи є тривимірними і відрізняються одна від одної напрямком осі *Z*: у правобічній вона спрямована на спостерігача, в лівобічній — у глибину екрана. Вісь *X* спрямована вправо щодо спостерігача, вісь *Y*— вгору.

Лівобічна система використовується для завдання значень параметрам команди `gluPerspective`, `glOrtho`, які будуть розглянуті далі. Правобічна система координат використовується у всіх інших випадках. Відображення тривимірної інформації відбувається в двовимірну *віконну* систему координат.



а) правобічна **б) лівобічна** **в) віконна**

Рис. 3.1. 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.

3.1 Робота з матрицями

Для завдання різних перетворень об’єктів в OpenGL використовуються операції над матрицями, при цьому розрізняються три типи матриць: модельно-видова, матриця проекцій і матриця текстури. Усі вони мають розмір 4x4. Видова матриця визначає перетворення об’єкту у світових координатах, такі як паралельний перенос, зміна масштабу і поворот. Матриця проекцій визначає, як будуть проектуватися тривимірні об’єкти на площину екрану (у віконні координати), а матриця текстури визначає накладання текстури на об’єкт.

Множення координат на матриці відбувається в момент виклику відповідної команди OpenGL, яка визначає координату (як правило, це команда `glVertex`).

Для того щоб вибрати, яку матрицю треба змінити, використовується команда:

glMatrixMode (*mode*: GLenum)

виклик якої зі значенням параметра *mode* рівним GL_MODELVIEW, GL_PROJECTION, або GL_TEXTURE включає режим роботи з модельно-видовою матрицею, матрицею проекцій чи матрицею текстури відповідно. Для виклику команд, що задають матриці того чи іншого типу, необхідно спочатку встановити відповідний режим.

Для визначення елементів матриці поточного типу викликається команда

glLoadMatrix[f d] (*m*: ^GLtype)

де *m* вказує на масив з 16 елементів типу *float* чи *double* відповідно до назви команди, при чому спочатку в ньому повинний бути записаний перший стовпець матриці, потім другий, третій і четвертий. Ще раз звернемо увагу: у масиві *m* матриця записана по стовпцях.

Команда

glLoadIdentity

заміняє поточну матрицю на одиничну.

Часто буває необхідно зберегти вміст поточної матриці для подальшого використання, для чого застосовуються команди

glPushMatrix
glPopMatrix

Вони записують і відновлюють поточну матрицю зі стека, причому для кожного типу матриць використовується свій стек. Для модельно-видових матриць мінімальна глибина стеку дорівнює 32, для інших — 2.

Для множення поточної матриці на іншу матрицю використовується команда

glMultMatrix[f d] (*m*: ^GLtype)

де параметр *m* повинний задавати матрицю розміром 4x4. 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 застосування зображена на рис. 3.2. Запам’ятайте: усі перетворення об’єктів і камери в OpenGL відбуваються за допомогою множення векторів координат на матриці. Причому множення відбувається на

поточну матрицю в момент визначення координати командою `glVertex` і деякими іншими.

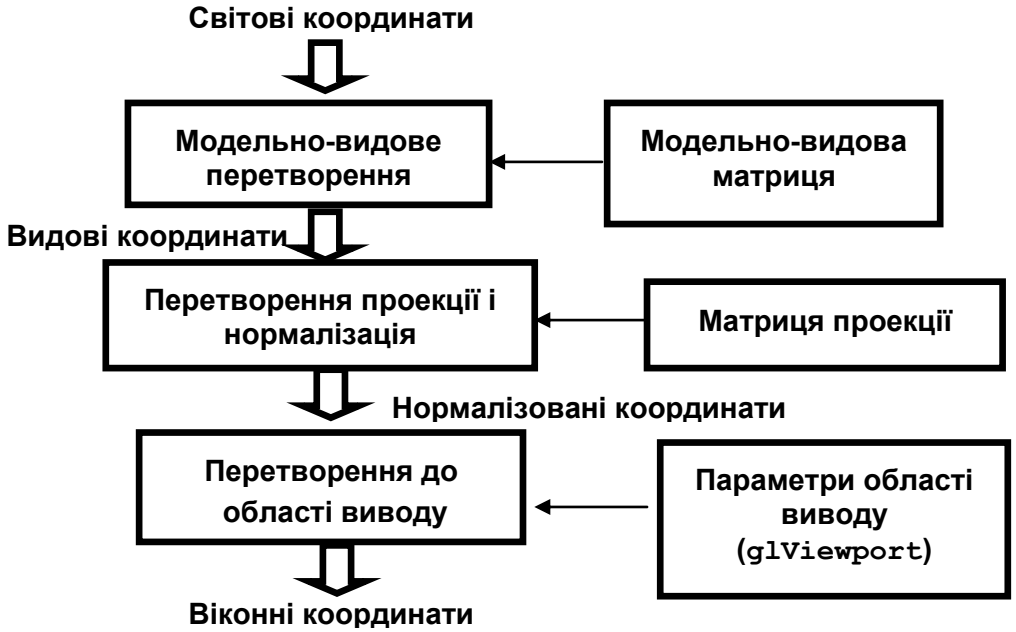


Рис. 3.2. Перетворення координат в OpenGL

3.2 Модельно-видові перетворення

До модельно-видових перетворень відносяться зсув, поворот і зміну масштабу уздовж координатних осей. Для проведення цих операцій досить помножити на відповідну матрицю кожну вершину об'єкту 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 проекція виконуються аналогічно. Сама матриця може бути створена за допомогою наступних команд:

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

`glTranslate` — зсуває об'єкт, додаючи до координат його вершин значення своїх параметрів.

`glRotate` — повертає об'єкт проти годинникової стрілки на кут *angle* (завдається у градусах) навколо вектора (*x*, *y*, *z*).

`glScale` — забезпечує масштабування об'єкта (розтягнення або стискання) уздовж вектора (*x*, *y*, *z*), перемножуючи відповідні координати його вершин на значення своїх параметрів.

Усі ці перетворення змінюють поточну матрицю, а тому застосовуються до примітивів, що визначаються пізніше. У випадку, якщо треба,

наприклад, повернути один об'єкт сцени, а інший залишити нерухомим, зручно спочатку зберегти поточну видову матрицю в стеку командою `glPushMatrix`, потім викликати `glRotate` з відповідними параметрами, описати примітиви, з яких складається цей об'єкт, а потім відновити поточну матрицю командою `glPopMatrix`.

Крім зміни положення самого об'єкта, часто буває необхідно змінити положення спостерігача, що також приводить до зміни модельно-видової матриці. Це можна зробити за допомогою команди

gluLookAt (*eyex*: GLdouble, *eyey*: GLdouble, *eyez*: GLdouble, *centerx*: GLdouble, *centery*: GLdouble, *centerz*: GLdouble, *upx*: GLdouble, *upy*: GLdouble, *upz*: GLdouble)

де точка (*eyex, eyey, eyez*) визначає точку спостереження, (*centerx, centery, centerz*) задає центр сцени, що буде проектуватися в центр області виводу, а вектор (*upx, upy, upz*) задає позитивний напрямок осі y, визначаючи поворот камери. Якщо, наприклад, камеру не треба повертати, то задається значення (0, 1, 0), а зі значенням (0,-1, 0) сцена буде перевернута.

Ця команда робить перенос і поворот об'єктів сцени, але в такому виді 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: у загальному випадку матричні перетворення в OpenGL потрібно записувати в зворотному порядку. Наприклад, якщо ви хочете спочатку повернути об'єкт, а потім пересунути його, спочатку викличте команду `glTranslate`, а тільки потім — `glRotate`. Ну а після цього визначайте сам об'єкт.

3.3 Проекції

В OpenGL існують стандартні команди для завдання ортографічної (паралельної) і перспективної проекцій. Перший тип проекції (рис. 3.3) може бути заданий командами

glOrtho (*left*: GLdouble, *right*: GLdouble, *bottom*: GLdouble, *top*: GLdouble, *near*: GLdouble, *far*: GLdouble)

gluOrtho2D (*left*: GLdouble, *right*: GLdouble, *bottom*: GLdouble, *top*: GLdouble)

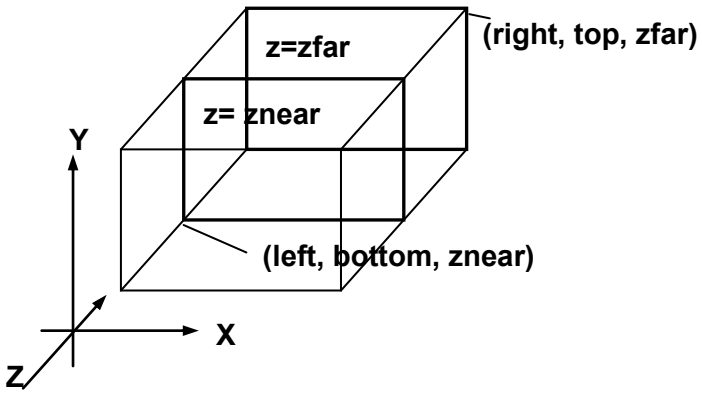


Рис. 3.3. Ортографічна проекція

Перша команда створює матрицю проекції в зрізаний об'єм видимості (паралелепіпед видимості) у лівобічній системі координат. Параметри команди задають точки (*left, bottom, znear*) і (*right, top, zfar*), які відповідають лівому нижньому і правому верхньому кутам вікна виводу. Параметри *near* і *far* 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 *near* і *far* встановлюються рівними –1 і 1 відповідно. Це зручно, якщо OpenGL використовується для малювання двовимірних об'єктів. У цьому випадку положення вершин можна задавати, використовуючи команди `glVertex2*`.

Перспективна проекція (рис. 3.4) визначається командою

gluPerspective (*angley*: GLdouble, *aspect*: GLdouble, *znear*: GLdouble, *zfar*: GLdouble)

яка задає зрізаний конус видимості в лівобічній системі координат.

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.

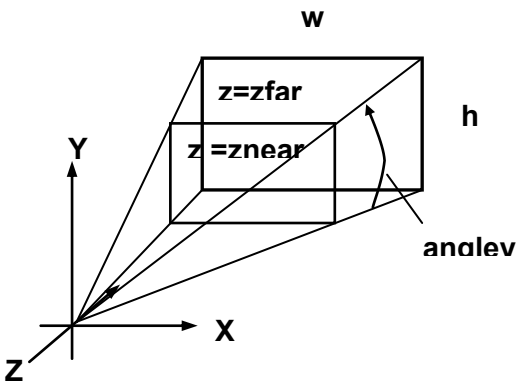


Рис. 3.4. Перспективна проекція

Перш ніж задавати матриці проекцій, не забудьте увімкнути режим роботи з потрібною матрицею командою `glMatrixMode(GL_PROJECTION)` і скинути поточну викликом `glLoadIdentity`. Наприклад:

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

3.4 Робоча область

Після застосування матриці проекцій на вхід наступного перетворення подаються так звані відсічені (*clipped*) координати. Потім знаходяться нормалізовані координати вершин за формулою:

$$x_n, y_n, z_n \ ^T = \ \frac{x_c}{w_c}, \frac{y_c}{w_c}, \frac{z_c}{w_c} \ ^T$$

Робоча область є прямокутником у віконній системі координат, розміри якого задаються командою:

glViewPort (*x*: GLint, *y*: GLint, *width*: GLint, *height*: GLint)

Значення всіх параметрів задаються в пікселях і визначають ширину і висоту робочої області з координатами лівого нижнього кута (*x*, *y*) у віконній системі координат. Розміри віконної системи координат визначаються поточними розмірами вікна застосування, точка (0, 0) знаходиться в лівому нижньому куті вікна.

Використовуючи параметри команди `glViewPort`, OpenGL обчислює віконні координати центра робочої області (O_x, O_y) за формулами:

$$O_x = x + \frac{width}{2};\ O_y = y + \frac{height}{2}$$

Нехай p_x=*width*, p_y=*height*, тоді можна знайти віконні координати кожної вершини:

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

При цьому цілі позитивні величини *n* і *f* задають мінімальну і максимальну глибину точки у вікні, за замовчуванням рівну 0 та 1 відповідно. Глибина кожної точки записується в спеціальний буфер глибини (*z-буфер*), що використовується для видалення невидимих ліній і поверхонь. Встановити значення *n* і *f* можна викликом функції

```
glDepthRange (n: GLclampd, f: GLclampd)
```

Контрольні питання

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»?

Розділ 4. Матеріали і освітлення

Для створення реалістичних зображень необхідно визначити як властивості самого об'єкта, так і властивості середовища, в якому він знаходиться . Перша група властивостей містить у собі параметри матеріалу, з якого зроблено об'єкт, способи накладення текстури на його поверхню, ступінь прозорості об'єкта. До другої групи можна віднести кількість і властивості джерел світла, рівень прозорості середовища, а також модель освітлення. Усі ці властивості можна задавати відповідними команди OpenGL.

4.1 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 у пункті 2.2.3. режим нормалізації нормалей, що включається викликом команди `glEnable(GL_NORMALIZE)`.

Для завдання глобальних параметрів освітлення використовуються команди

glLightModel[i f] (*pname*: GLenum, *param*: GLenum)
glLightModel[i f]v (*pname*: GLenum, *params*: ^GLtype)

Аргумент *pname* визначає, який параметр моделі освітлення буде настроюватися і може приймати наступні значення:

GL_LIGHT_MODEL_LOCAL_VIEWER	параметр *param* повинен бути булевим і задає положення спостерігача. Якщо він дорівнює GL_FALSE (за замовчуванням), то напрямок огляду вважається паралельним осі –*z*, в незалежності від положення у видових координатах. Якщо ж він дорівнює GL_TRUE, то спостерігач знаходиться в початку видової системи координат. Це може поліпшити якість освітлення, але ускладнює його розрахунок.
GL_LIGHT_MODEL_TWO_SIDE	параметр *param* повинний бути булевим і керує режимом розрахунку освітленості, як для лицьових, так і для зворотних граней. Якщо він дорівнює GL_FALSE (за замовчуванням), то

	освітленість розраховується тільки для лицьових граней, якщо `GL_TRUE` — то розрахунок проводиться і для зворотних граней.
`GL_LIGHT_MODEL_AMBIENT`	параметр *params* повинний містити чотири цілих чи дійсних числа, що визначають колір фонового освітлення навіть у випадку відсутності визначених джерел світла. Значення за умовчуванням: (0.2, 0.2, 0.2,1.0).

4.2 Специфікація матеріалів

Для завдання параметрів поточного матеріалу використовуються команди

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

За допомогою цих команд можна визначити розсіяний, дифузний і дзеркальний кольори матеріалу, а також ступінь дзеркального відображення й інтенсивність випромінювання світла, якщо об’єкт повинен світитися. Який саме параметр буде визначатися значенням *param*, залежить від значення *pname*:

`GL_AMBIENT`	параметр *params* повинний містити чотири цілих чи дійсних значення кольорів *RGBA* (за замовчуванням (0.2, 0.2, 0.2, 1.0)), які визначають розсіяний колір матеріалу (колір матеріалу в тіні)
`GL_DIFFUSE`	параметр *params* повинний містити чотири цілих чи дійсних значення кольорів *RGBA* (за замовчуванням (0.8, 0.8, 0.8, 1.0)), які визначають дифузний колір матеріалу
`GL_SPECULAR`	параметр *params* повинний містити чотири цілих чи дійсних значення кольорів *RGBA* (за замовчуванням (0.0, 0.0, 0.0, 1.0)), які визначають дзеркальний колір матеріалу
`GL_SHININESS`	параметр *params* повинний містити одне ціле чи дійсне значення в діапазоні від 0 (за замовчуванням) до 128, яке визначає ступінь дзеркального відображення матеріалу
`GL_EMISSION`	параметр *params* повинний містити чотири цілих чи дійсних значення кольорів *RGBA* (за замовчу-

	ванням (0.0, 0.0, 0.0, 1.0)), які визначають інтенсивність випромінюваного світла матеріалу
GL_AMBIENT_AND_DIFFUSE	еквівалентно двом викликам команди glMaterial зі значенням *pname* GL_AMBIENT і GL_DIFFUSE і однаковими значеннями *params*

З цього випливає, що виклик команди glMaterial[i f] можливий тільки для установки ступеня дзеркального відображення матеріалу (*shininess*). Команда glMaterial[i f]v використовується для завдання інших параметрів.

Параметр *face* визначає тип граней, для яких задається цей матеріал і може приймати значення GL_FRONT, GL_BACK або GL_FRONT_AND_BACK.

Якщо в сцені матеріали об'єктів розрізняються лише одним параметром, бажано спочатку встановити потрібний режим, викликавши glEnable з параметром GL_COLOR_MATERIAL, а потім використовувати команду

glColorMaterial (*face*: GLenum, *pname*: GLenum)

де параметр *face* має аналогічне значення, а параметр *pname* може приймати всі перераховані значення. Після цього значення обраного за допомогою *pname* властивості матеріалу для конкретного об'єкту (чи вершини) встановлюються викликом команди glColor, що дозволяє уникнути викликів більш ресурсоємної команди glMaterial і підвищує ефективність програми. Приклад визначення властивостей матеріалу:

```
const
  mat_dif : array [0..2] of GLFloat =(0.8,0.8,0.8);
  mat_amb : array [0..2] of GLFloat = (0.2, 0.2, 0.2);
  mat_spec :array [0..2] of GLFloat = (0.6, 0.6, 0.6);
  shininess = 0.7 * 128;
  …
  glMaterialfv (GL_FRONT_AND_BACK,GL_AMBIENT, @mat_amb);
  glMaterialfv (GL_FRONT_AND_BACK,GL_DIFFUSE, @mat_dif);
  glMaterialfv (GL_FRONT_AND_BACK,GL_SPECULAR, @mat_spec);
  glMaterialf  (GL_FRONT,GL_SHININESS, @shininess);
```

4.3 Опис джерел світла

Визначення властивостей матеріалу об'єкта має сенс, тільки якщо в сцені є джерела світла. Інакше всі об'єкти будуть чорними (точніше будуть мати колір, який дорівнює розсіяному кольору матеріалу). Додати в сцену джерело світла можна за допомогою команд:

glLight[i f] (*light*: GLenum, *pname*: GLenum, *param*: GLfloat)
glLight[i f]v (*light*: GLenum, *pname*: GLenum, *params*: ^GLfloat)

Параметр *light* однозначно визначає джерело світла. Він вибирається з набору спеціальних символічних імен виду `GL_LIGHTi`, де *i* повинно лежати в діапазоні від 0 до константи `GL_MAX_LIGHT`, яка звичайно не перевищує восьми.

Параметри *pname* і *params* мають сенс, аналогічний команді `glMaterial`. Розглянемо значення параметра *pname*:

`GL_SPOT_EXPONENT`	параметр *param* повинний містити ціле чи дійсне число від 0 (розсіяне світло за замовчуванням) до 128, що задає розподіл інтенсивності світла. Цей параметр описує рівень сфокусованості джерела світла
`GL_SPOT_CUTOFF`	параметр *param* повинний містити ціле чи дійсне число між 0 і 90 чи дорівнювати 180 (розсіяне світло за замовчуванням), яке визначає максимальний кут розсіювання світла. Значенням цього параметра є половина кута у вершині конусоподібного світлового потоку, створюваного джерелом
`GL_AMBIENT`	параметр *params* повинний містити чотири цілих чи дійсних значення кольорів *RGBA* (за замовчуванням (0, 0, 0, 1)), які визначають колір фонового освітлення
`GL_DIFFUSE`	параметр *params* повинний містити чотири цілих чи дійсних значення кольорів *RGBA* (за замовчуванням (1, 1, 1, 1) для `GL_LIGHT0` і (0, 0, 0, 1) для інших), які визначають колір дифузного освітлення
`GL_SPECULAR`	параметр *params* повинний містити чотири цілих чи дійсних значення кольорів *RGBA* (за замовчуванням (1, 1, 1, 1) для `GL_LIGHT0` і (0, 0, 0, 1) для інших), які визначають колір дзеркального відображення
`GL_POSITION`	параметр *params* повинний містити чотири цілих чи дійсних числа, що визначають положення джерела світла. Якщо значення компоненти *w* дорівнює 0.0, то джерело вважається нескінченно віддаленим і при розрахунку освітленості

	враховується тільки напрямок на точку (*x*,*y*,*z*), у противному випадку вважається, що джерело розташоване в точці (*x*,*y*,*z*,*w*). У першому випадку ослаблення світла при віддаленні від джерела не відбувається, тобто джерело вважається нескінченно віддаленим. Значення за замовчуванням: (0, 0, 1, 0)
`GL_SPOT_DIRECTION`	параметр *params* повинний містити чотири цілих чи дійсних числа, які визначають напрямок світла. Значення за замовчуванням: (0, 0, -1, 1). Ця характеристика джерела має сенс, якщо значення `GL_SPOT_CUTOFF` відмінне від 180 (яке, до речі, задано за замовчуванням)

`GL_CONSTANT_ATTENUATION`
`GL_LINEAR_ATTENUATION`
`GL_QUADRATIC_ATTENUATION`
	параметр *params* задає значення одного з трьох коефіцієнтів, що означають ослаблення інтенсивності світла при віддаленні від джерела. Допускаються тільки ненегативні значення. Якщо джерело не є спрямованим (див. `GL_POSITION`), то ослаблення обернено пропорційно сумі:
	$att_{constant} + att_{linear} \cdot d + att_{quadratic} \cdot d^2$
	де *d* — відстань між джерелом світла і освітлюваною їм вершиною, $att_{constant}$, att_{linear} і $att_{quadratic}$ дорівнюють параметрам, заданим за допомогою констант `GL_CONSTANT_ATTENUATION`, `GL_LINEAR_ATTENUATION` і `GL_QUADRATIC_ATTENUATION` відповідно.
	За замовчуванням ці параметри задаються трійкою (1, 0, 0), і фактично ослаблення не відбувається

При зміні положення джерела світла варто враховувати наступний факт: у OpenGL джерела світла є об’єктами, багато в чому такими ж, як багатокутники і точки. На них поширюється основне правило обробки координат у OpenGL — параметри, які описують положення у просторі, перетворюються поточною модельно-видовою матрицею в момент формування об’єкта, тобто в момент виклику відповідних команд OpenGL. Таким чином, формуючи джерело світла одночасно з об’єктом сцени чи камерою, його можна прив’язати до цього об’єкта. Чи, навпаки, сформувати

стаціонарне джерело світла, що буде залишатися на місці, поки інші об'єкти пересуваються.

Загальні правила наступні:

1. Якщо положення джерела світла задається командою `glLight` перед визначенням положення віртуальної камери (наприклад, командою `glLookAt`), то буде вважатися, що координати (0,0,0) джерела знаходяться в точці спостереження і, отже, положення джерела світла визначається щодо положення спостерігача.
2. Якщо положення встановлюється між визначенням положення камери і перетвореннями модельно-видової матриці об'єкта, то воно фіксується, тобто в цьому випадку положення джерела світла задається у світових координатах.

Для використання освітлення спочатку треба встановити відповідний режим викликом команди `glEnable(GL_LIGHTING)`, а потім увімкнути потрібне джерело командою `glEnable(GL_LIGHTi)`.

Ще раз звернемо увагу на те, що при вимкненому освітленні колір вершини дорівнює поточному кольору, що задається командами `glColor`. При увімкненому освітленні колір вершини обчислюється виходячи з інформації про матеріал, нормалі і джерела світла.

При вимиканні освітлення візуалізація відбувається швидше, однак у такому випадку застосування повинно саме розраховувати кольори вершин.

Текст програми, що демонструє основні принципи визначення матеріалів і джерел світла, приведений в *дод. С 3*.

4.4 Створення ефекту туману

На завершення розглянемо одну цікаву і часто використовувану можливість OpenGL — створення ефекту туману. Легке затуманення сцени створює реалістичний ефект, а часто може сховати деякі артефакти, які з'являються, коли в сцені присутні віддалені об'єкти.

Туман у OpenGL реалізується шляхом зміни кольору об'єктів у сцені в залежності від їхньої глибини, тобто відстані до точки спостереження. Зміна кольору відбувається або для вершин примітивів, або для кожного пікселя на етапі растеризації в залежності від реалізації OpenGL.

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

Метод обчислення інтенсивності тумана у вершині можна визначити за допомогою команд

glFog[if] (*pname*: enum, param: T)
glFog[if]v (*pname*: enum, params: T)

Аргумент *pname* може приймати наступні значення:

<table>
<tr><td rowspan="5">GL_FOG_MODE</td><td colspan="2">аргумент 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 компонент кольору туману</td></tr>
</table>

Контрольні питання

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?

Розділ 5. Накладання текстури

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

Накладення текстури на поверхню об'єктів сцени підвищує її реалістичність, однак при цьому треба враховувати, що цей процес вимагає значних обчислювальних витрат, особливо якщо OpenGL не підтримується апаратно.

Для роботи з текстурою потрібно виконати наступну послідовність дій:

- вибрати зображення і перетворити його до потрібного формату;
- передати зображення в OpenGL;
- визначити, як текстура буде наноситися на об'єкт і як вона буде з ним взаємодіяти;
- зв'язати текстуру з об'єктом.

5.1 Підготовка текстури

Для використання текстури необхідно спочатку завантажити в пам'ять потрібне зображення і передати його OpenGL.

Зчитування графічних даних з файлу і їх перетворення можна проводити вручну. Можна також скористатися функцією, що входить до складу бібліотеки GLAUX (для її використання треба додатково підключити *glaux.lib*), яка сама проводить необхідні операції. Це функція

```
AUX_RGBImageRec* auxDIBImageLoad (const char *file)
```

де *file* — назва файлу з розширенням **.bmp* чи **.dib*. Функція повертає покажчик на область пам'яті, де зберігаються перетворені дані. Як зрозуміло із синтаксису, ця функція використовується в мові C++. При програмуванні в Delphi можна використати універсальну процедуру, реалізація якої приводиться нижче.

```
procedure PrepareImage(BmpFileName: AnsiString);
type
  PPixelArray = ^TPixelArray;
  TPixelArray = array [0..0] of Byte;
var
  Bitmap : TBitmap;
  Data : PPixelArray;
  BMInfo : TBitmapInfo;
  I, ImageSize : Integer;
  Temp : Byte;
  MemDC : HDC;
begin
```

```
    Bitmap := TBitmap.Create;
    Bitmap.LoadFromFile(BmpFileName);
    with BMinfo.bmiHeader do begin
      FillChar (BMInfo, SizeOf(BMInfo), 0);
      biSize := sizeof (TBitmapInfoHeader);
      biBitCount := 24;
      biWidth := Bitmap.Width;
      biHeight := Bitmap.Height;
      ImageSize := biWidth * biHeight;
      biPlanes := 1;
      biCompression := BI_RGB;
      MemDC := CreateCompatibleDC (0);
      GetMem (Data, ImageSize * 3);
      try
        GetDIBits (MemDC, Bitmap.Handle, 0, biHeight, Data,
                   BMInfo, DIB_RGB_COLORS);
        For I := 0 to ImageSize - 1 do begin
            Temp := Data [I * 3];
            Data [I * 3] := Data [I * 3 + 2];
            Data [I * 3 + 2] := Temp;
        end;
        glTexImage2d(GL_TEXTURE_2D, 0, 3, biWidth,
                     biHeight, 0, GL_RGB, GL_UNSIGNED_BYTE, Data);
      finally
        FreeMem (Data);
        DeleteDC (MemDC);
        Bitmap.Free;
      end;
    end;
  end;
```

При створенні образа текстури в пам'яті варто враховувати наступні вимоги.

По-перше, розміри текстури, як по горизонталі, так і по вертикалі повинні бути ступенями двійки. Ця вимога накладається для компактного розміщення текстури в текстурній пам'яті і сприяє її ефективному використанню. Працювати тільки з такими текстурними файлами звичайно незручно, тому після завантаження їх треба перетворити. Зміну розмірів текстури можна провести за допомогою команди

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

У якості значення параметра *format* звичайно використовується значення `GL_RGB` чи `GL_RGBA`, яке визначає формат збереження інформації. Параметри *widthin*, *heightin*, *widhtout*, *heightout* визначають розміри вхідного і вихідного зображень, а за допомогою *typein* і *typeout* задається тип елементів масивів, розташованих за адресами *datain* і *dataout*. Зазвичай, це

може бути тип `GL_UNSIGNED_BYTE`, `GL_SHORT`, `GL_INT` і т.д. Результат своєї роботи функція заносить в область пам'яті, на яку вказує параметр *dataout*.

По-друге, треба передбачити випадок, коли об'єкт після растеризації виявляється за розмірами значно меншим нанесеної на нього текстури. Чим менший об'єкт, тим меншою повинна бути текстура, що наноситься на нього, і тому вводиться поняття *рівнів деталізації текстури* (*mipmap*). Кожен рівень деталізації задає деяке зображення, що є, як правило, зменшеною в два рази копією оригіналу. Такий підхід дозволяє поліпшити якість нанесення текстури на об'єкт. Наприклад, для зображення розміром 2^mx2^n можна побудувати *max(m,n)+1* зменшених зображень, що відповідають різним рівням деталізації.

Ці два етапи створення образу текстури у внутрішній пам'яті OpenGL можна зробити за допомогою команди

gluBuild2DMipmaps (*target*: GLenum, *components*: GLint, *width*: GLint, *height*: GLint, *format*: GLenum, *type*: GLenum, *data*: pointer)

де параметр *target* повинен дорівнювати `GL_TEXTURE_2D`. Параметр *components* визначає кількість колірних компонентів текстури і може приймати наступні основні значення:

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

Параметри *width*, *height*, *data* визначають розміри і розташування текстури відповідно, а *format* і *type* мають аналогічне значення, що й у команди `gluScaleImage`.

Після виконання цієї команди текстура копіюється у внутрішню пам'ять OpenGL, і тому пам'ять, що зайнята вихідним зображенням, можна звільнити.

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

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

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

треба створити *n* ідентифікаторів текстур, які будуть записані в масив *textures*. Перед початком визначення властивостей чергової текстури варто зробити її поточною («прив'язати» текстуру), викликавши команду

glBindTexture (*target*: GLenum, *texture*: GLuint)

де *target* може приймати значення `GL_TEXTURE_1D` або `GL_TEXTURE_2D`, а параметр *texture* повинен дорівнювати ідентифікатору тієї текстури, до якої будуть відноситися наступні команди. Для того, щоб у процесі малювання зробити поточною текстуру з деяким ідентифікатором, досить знову викликати команду `glBindTexture` з відповідним значенням *target* і *texture*. Таким чином, команда `glBindTexture` вмикає режим створення текстури з ідентифікатором *texture*, якщо така текстура ще не створена, або режим її використання, тобто робить цю текстуру поточною.

Оскільки не кожна апаратура може оперувати текстурами великого розміру, доцільно обмежити розміри текстури до *256x256* чи *512x512* пікселів. Зазначимо, що використання невеликих текстур підвищує ефективність програми.

5.2 Накладання текстури на об'єкти

При накладанні текстури, як вже згадувалося, треба враховувати випадок, коли розміри текстури відрізняються від віконних розмірів об'єкта, на який вона накладається. При цьому можливе як розтягування, так і стиснення зображення, і те, як будуть проводитися ці перетворення, може серйозно вплинути на якість побудованого зображення. Для визначення положення точки на текстурі використовується параметрична система координат (*s*, *t*), причому значення *s* і *t* знаходяться у відрізку [0,1] (рис. 5.1).

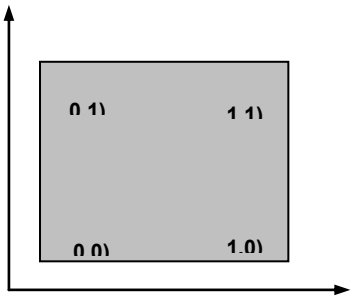


Рис. 5.1. Текстурні координати

Для зміни різних параметрів текстури застосовуються команди:

glTexParameter[i f] (*target*: GLenum, *pname*: GLenum, *param*: GLenum)
glTexParameter[i f]v (*target*: GLenum, *pname*: GLenum, *params*: ^ GLenum)

Параметр *target* може приймати значення `GL_TEXTURE_1D` або `GL_TEXTURE_2D`, *pname* визначає, яка властивість буде змінюватися, а за до-

помогою *param* чи *params* встановлюється нове значення. Можливі значення *pname*:

GL_TEXTURE_MIN_FILTER	параметр *param* визначає функцію, яка буде використовуватися для стиску текстури. При значенні GL_NEAREST буде використовуватися один (найближчий), а при значенні GL_LINEAR (значення за замовчуванням) чотири найближчих елементи текстури
GL_TEXTURE_MAG_FILTER	параметр *param* визначає функцію, яка буде використовуватися для збільшення (розтягання) текстури. При значенні GL_NEAREST буде використовуватися один (найближчий), а при значенні GL_LINEAR (значення за замовчуванням) чотири найближчих елементи текстури
GL_TEXTURE_WRAP_S	параметр *param* встановлює значення координати *s*, якщо воно не входить у відрізок [0,1]. При значенні GL_REPEAT (значення за замовчуванням) ціла частина *s* відкидається, і в результаті зображення розмножується по поверхні. При значенні GL_CLAMP використовуються крайові значення: 0 чи 1, що зручно використовувати, якщо на об'єкт накладається одне зображення
GL_TEXTURE_WRAP_T	аналогічно попередньому значенню, тільки для координати *t*

Використання режиму GL_NEAREST підвищує швидкість накладання текстури, однак при цьому знижується якість, тому що на відміну від GL_LINEAR інтерполяція не виконується.

Для того щоб визначити, як текстура буде взаємодіяти з матеріалом, з якого зроблений об'єкт, використовуються команди

```
glTexEnv[i f]  (target: GLenum, pname: GLenum, param: GLtype)
glTexEnv[i f]v (target: GLenum, pname: GLenum, params: ^GLtype)
```

Параметр *target* повинен дорівнювати GL_TEXTURE_ENV, а для *pname* розглянемо тільки одне значення GL_TEXTURE_ENV_MODE, яке найчастіше застосовується.

Найчастіше використовуються такі значення параметра *param*:

GL_MODULATE	кінцевий колір знаходиться як добуток кольору точки на поверхні і кольору відповідної їй точки на текстурі

GL_REPLACE	як кінцевий колір використовується колір точки на текстурі

5.3 Текстурні координати

Перед нанесенням текстури на об'єкт необхідно встановити відповідність між точками на поверхні об'єкту і на самій текстурі. Задавати цю відповідність можна двома методами: окремо для кожної вершини чи для усіх вершин, задавши параметри спеціальної функції відображення.

Перший метод реалізується за допомогою команд

```
glTexCoord[1 2 3 4][s i f d]  (coord: GLtype)
glTexCoord[1 2 3 4][s i f d]v (coords: ^GLtype)
```

Найчастіше використовуються команди виду `glTexCoord2*(`***s***`: GLtype,` ***t***`: GLtype)`, які задають поточні координати текстури. Поняття поточних координат текстури аналогічно поняттям поточного кольору і поточної нормалі, і є атрибутом вершини. Однак навіть для куба знаходження відповідних координат текстури є досить трудомістким заняттям, тому в бібліотеці GLU крім команд, що проводять побудову таких примітивів, як сфера, циліндр і диск, передбачене також накладення на них текстур. Для цього досить викликати команду

```
gluQuadricTexture (quadObject: ^GLUquadricObj, textureCoords: GLboolean)
```

з параметром *textureCoords*, який дорівнює `GL_TRUE`, і тоді поточна текстура буде автоматично накладатися на примітив.

Другий метод реалізується за допомогою команд

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

Параметр *coord* визначає, для якої координати задається формула, і може приймати значення `GL_S`, `GL_T`,а параметр *pname* може дорівнювати одному з наступних значень:

GL_TEXTURE_GEN_MODE	визначає функцію для накладення текстури. У цьому випадку аргумент *param* приймає значення:	
	GL_OBJECT_LINEAR	значення відповідної текстурної координати визначається відстанню до площини, що задається за допомогою значення *pname* GL_OBJECT_PLANE (див. нижче). Формула має наступний вигляд: $g = x \cdot x_p + y \cdot y_p + z \cdot z_p + w \cdot w_p$ де *g* — відповідна текстурна коор-

		дината (*s* чи *t*), *x*, *y*, *z*, *w* — координати відповідної точки, x_p, y_p, z_p, w_p — коефіцієнти рівняння площини. У формулі використовуються координати об’єкта
	GL_EYE_LINEAR	аналогічно попередньому значенню, тільки у формулі використовуються видові координати. Тобто координати текстури об’єкта в цьому випадку залежать від положення цього об’єкта
	GL_SPHERE_MAP	дозволяє емулювати віддзеркалювання від поверхні об’єкта. Текстура начебто «обертається» навколо об’єкта. Для даного методу використовуються видові координати і необхідне завдання нормалей
GL_OBJECT_PLANE	дозволяє задати площину, відстань до якої буде використовуватися при генерації координат, якщо встановлений режим GL_OBJECT_LINEAR. У цьому випадку параметр *params* є 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 завдання текстурних координат необхідно викликати команду glEnable з параметром GL_TEXTURE_GEN_S або GL_TEXTURE_GEN_P.

Для прикладу розглянемо, як можна задати дзеркальну текстуру. При такому накладенні текстури зображення буде начебто відбиватися від поверхні об’єкта, викликаючи цікавий оптичний ефект. Для цього спочатку треба створити два цілочисельних масиви коефіцієнтів *s_coeffs* і *t_coeffs* зі значеннями (1,0,0,1) і (0,1,0,1) відповідно, а потім викликати команди:

```
glEnable (GL_TEXTURE_GEN_S);
glTexGeni (GL_S, GL_TEXTURE_GEN_MODE, GL_EYE_LINEAR);
glTexGendv (GL_S, GL_EYE_PLANE, s_coeffs);
```

і такі ж команди для координати *t* з відповідними змінами.

Програма, що використовує накладення текстури, приведена в *дод. С2*.

Контрольні питання

1. Для чого використовуються рівні деталізації текстури (mipmapping)?
2. Як встановити відповідність між координатами об'єкту та текстури?
3. Які обмеження накладаються на файли текстури?
4. Що таке текстурні координати?

Розділ 6. Операції з пікселями

Після проведення всієї 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 2, графічна бібліотека OpenGL підтримує роботу з наступними буферами:

- кілька буферів кольору
- буфер глибини
- буфер-накопичувач (акумулятор)
- буфер маски

Група буферів кольору включає буфер кадру, але таких буферів може бути декілька. При використанні подвійної буферизації говорять про робочий (*front*) і фоновий (*back*) буфери. Як правило, у фоновому буфері програма створює зображення, що потім разом копіюється в робочий буфер. На екрані може з'явитися інформація тільки буферів кольору.

Буфер глибини призначений для видалення невидимих поверхонь, і пряма робота з ним потрібна вкрай рідко. Буфер-накопичувач можна застосовувати для різних операцій, більш докладно робота з ним описана в розділі 6.2. Буфер маски використовується для формування піксельних масок (трафаретів), які служать для вирізання із загального масиву тих пікселів, які варто вивести на екран.

6.1 Змішування зображень. 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 бу-

фері. Для змішування використовується четвертий компонент кольору — альфа-компонент, тому цей режим називають ще альфа-змішуванням. Програма може керувати інтенсивністю альфа-компоненти так само, як і інтенсивністю основних кольорів, тобто задавати значення інтенсивності для кожного пікселя чи кожної вершини примітиву.

Режим включається за допомогою команди `glEnable(GL_BLEND)`.

Визначити параметри змішування можна за допомогою команди:

`glBlendFunc`(*`src`*: enum, *`dst`*: enum)

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).

Приведемо 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:

```
glColor3f(1.0, 0.0, 0.0, 0.5);
glBegin(GL_TRIANGLES);
  glVertex3f(0.0, 0.0, 0.0);
  glVertex3f(1.0, 0.0, 0.0);
```

```
    glVertex3f(1.0, 1.0, 0.0);
glEnd;
```

Якщо в сцені є кілька прозорих об'єктів, що можуть перекривати один одного, коректне виведення можна гарантувати тільки у випадку виконання наступних умов:

- усі прозорі об'єкти виводяться після непрозорих;
- при виводі об'єкти з прозорістю повинні бути упорядковані по зменшенню глибини, тобто виводитися, починаючи з найбільш віддалених від спостерігача.

В OpenGL команди обробляються в порядку їхнього надходження, тому для реалізації перерахованих вимог досить розставити у відповідному порядку виклики команд `glVertex`, але і це в загальному випадку нетривіально.

6.2 Буфер-накопичувач

Буфер-накопичувач (*accumulation buffer*) — це один з додаткових буферів OpenGL. У ньому можна зберігати зображення для візуалізації, застосовуючи спеціальні операції до кожного пікселя. Буфер-накопичувач широко використовується для створення різних спецефектів.

Зображення береться з буфера, вибраного для зчитування командою

glReadBuffer(*buf*: enum)

Аргумент *buf* визначає буфер для зчитування. Значення *buf*, які дорівнюють `GL_BACK`, `GL_FRONT`, визначають відповідні буфери кольору для читання. `GL_BACK` задає як джерело пікселів фоновий буфер; `GL_FRONT` — поточний вміст вікна виводу. Команда має значення, якщо використовується подвійна буферизація. В іншому випадку використовується тільки один буфер, що відповідає вікну виведення.

Буфер-накопичувач є додатковим буфером кольору. Він не використовується для виведення зображень, але вони додаються в нього після виведення в один з буферів кольору. Застосовуючи описані нижче операції, можна потроху «накопичувати» зображення в буфері.

Потім отримане зображення переноситься з буфера-накопичувача в один з буферів кольору, обраний для запису командою

glDrawBuffer(*buf*: enum*)*

Значення *buf* аналогічне значенню відповідного аргументу в команді `glReadBuffer`.

Всі операції з буфером-накопичувачем контролюються командою

glAccum(*op*: enum, *value*: GLfloat)

Аргумент *op* задає операцію над пікселями і може приймати наступні значення:

GL_LOAD	піксель береться з буфера, обраного для читання, його значення збільшується на *value* і заноситься в буфер-накопичувач
GL_ACCUM	аналогічно попередньому, але отримане після множення значення складається з уже наявним у буфері
GL_MULT	ця операція множить значення кожного пікселя в буфері-накопичувачі на *value*
GL_ADD	аналогічно попередньому, тільки замість множення використовується додавання
GL_RETURN	зображення переноситься з буфера-накопичувача в буфер, обраний для запису. Перед цим значення кожного пікселя множиться на *value*

Слід зазначити, що для використання буфера-накопичувача немає необхідності викликати будь-яку команду glEnable. Досить ініціалізувати тільки сам буфер.

6.3 Буфер маски

При виводі пікселів у буфер кадру виникає необхідність виводити не всі пікселі, а тільки деяку їх підмножину, тобто накласти *трафарет* (*маску*) на зображення. Для цього OpenGL надає так званий буфер маски (*stencil buffer*) . Крім накладення маски, цей буфер надає ще кілька цікавих можливостей.

Перш ніж помістити піксель у буфер кадру, механізм візуалізації OpenGL дозволяє виконати порівняння (тест) між заданим значенням у буфері маски. Якщо тест проходить, піксель малюється в буфері кадру.

Механізм порівняння дуже гнучкий і контролюється наступними командами:

glStencilFunc (*func*: enum, *ref*: integer, *mask*: unsigned integer)
glStencilOp (*sfail*: enum, *dpfail*: enum, *dppass*: enum)

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 треба виконати команду `glEnable(GL_STENCIL_TEST)`.

Буфер маски використовується при створенні таких спецефектів, як тінь, віддзеркалення, плавні переходи однієї картинки в іншу тощо.

6.4 Керування растеризацією

Спосіб виконання растеризації примітивів можна частково регулювати командою

glHint (*target*: enum, *mode*: enum)

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

GL_FOG_HINT	точність обчислень при накладанні туману. Обчислення можуть виконуватися для кожного пікселя (найбільша точність) чи тільки у вершинах. Якщо реалізація OpenGL не підтримує попіксельного обчислення, то виконується тільки обчислення по вершинах
GL_LINE_SMOOTH_HINT	керування якістю прямих. При значенні *mode*, яке дорівнює GL_NICEST, зменшується ступінчастість прямих за рахунок більшого числа пікселів в прямих
GL_PERSPECTIVE_CORRECTION_HINT	точність інтерполяції координат при обчисленні кольорів і накладення текстури. Якщо реалізація OpenGL не підтримує режим GL_NICEST, то здійснюється лінійна інтерполяція координат
GL_POINT_SMOOTH_HINT	керування якістю точок. При значенні параметра *mode*, який дорівнює GL_NICEST точки малюються як кола
GL_POLYGON_SMOOTH_HINT	керування якістю виводу сторін багатокутника

Параметр *mode* інтерпретується в такий спосіб:

GL_FASTEST	використовується найбільш швидкий алгоритм
GL_NICEST	використовується алгоритм, що забезпечує кращу якість
GL_DONT_CARE	вибір алгоритму залежить від реалізації

Слід зауважити, що командою `glHint` програміст може тільки визначити свої побажання щодо того чи іншого аспекту растеризації примітивів. В конкретній реалізації OpenGL ці побажання можуть бути і проігноровані.

Зверніть увагу на те, що `glHint` не можна викликати між операторними дужками `glBegin`/`glEnd`.

Контрольні питання

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.

Розділ 7. 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.

7.1 Усунення ступінчастості

Почнемо з задачі усунення ступінчастості (*antialiasing*). Ефект ступінчастості (*aliasing*) виникає в результаті похибок растеризації примітивів у буфері кадру через скінченну (і, як правило, невелику) роздільну здатність. Є кілька підходів до вирішення даної проблеми. Наприклад, можна застосовувати фільтрацію отриманого зображення. Також цей ефект можна усувати на етапі растеризації, згладжуючи образ кожного примітива. Тут ми розглянемо прийом, що дозволяє усувати подібні артефакти для всієї сцени в цілому.

Для кожного кадру необхідно намалювати сцену кілька разів, на кожному проході трохи зсуваючи камеру відносно початкового положення. Положення камер, наприклад, можуть утворювати коло. Якщо зсув камери відносно малий, то похибки дискретизації проявляться по-різному, і, за рахунок їх усереднення, ми одержимо згладжене зображення.

Найпростіше зсувати положення спостерігача, але до цього потрібно обчислити величину зсуву так, щоб приведене до координат екрана значення не перевищувало, скажімо, половини розміру пікселя.

Всі отримані зображення зберігаємо в буфері-накопичувачі з коефіцієнтом *1/n*, де *n* — число проходів для кожного кадру. Чим більше таких проходів — тим нижче продуктивність, але краще результат.

```
for i:=0 to samples_count do
begin
// звичайно samples_count лежить у межах від 5 до 10
  ShiftCamera(i); // зміщуємо камеру
  RenderScene();
  if i = 0 then
   // на першій ітерації завантажуємо зображення
   glAccum(GL_LOAD,1/samples_count);
  else
   // додаємо до вже існуючого
   glAccum(GL_ACCUM,1/samples_count);
end;
// Записуємо те, що вийшло, назад у вихідний буфер
glAccum(GL_RETURN,1.0);
```

Слід зазначити, що усунення ступінчастості відразу для всієї сцени, як правило, зв'язане із серйозним падінням продуктивності візуалізації, тому що вся сцена малюється кілька разів. Сучасні прискорювачі зазви-

чай апаратно реалізують інші методи, засновані на так називаному ресемплінзі зображень.

7.2 Побудова тіней

В OpenGL немає вбудованої підтримки побудови тіней на рівні базових команд. У значній мірі це пояснюється тим, що існує безліч алгоритмів їхньої побудови, які можуть бути реалізовані через функції OpenGL. Присутність тіней суттєво збільшує реалістичність тривимірного зображення, тому розглянемо один з прийомів їх побудови.

Більшість алгоритмів побудови тіней використовують модифіковані принципи перспективної проекції. Ми розглянемо один з найпростіших методів. З його допомогою можна одержувати тіні, що відкидаються тривимірним об'єктом на площину.

Загальний підхід такий: для всіх точок об'єкта знаходиться їхня проекція паралельно вектору, що з'єднує дану точку і точку, в якій знаходиться джерело світла, на деяку задану площину. Таким чином одержуємо новий об'єкт, що цілком належить заданій площині. Цей об'єкт і є тінню вихідного.

Розглянемо математичні основи даного методу. Нехай:

P — точка в тривимірному просторі, що відкидає тінь.

L — положення джерела світла, що освітлює дану точку.

$\boldsymbol{S} = a \cdot \boldsymbol{L} - \boldsymbol{P} \ - \boldsymbol{P}$ — точка, у яку відкидає тінь точка *P*, де *a* — параметр.

Припустимо, що тінь падає на площину *z*=0. У цьому випадку $a = \frac{z_p}{z_1 - z_p}$.

Таким чином:

$$x_s = \frac{x_p \cdot z_1 - x_1 \cdot z_p}{z_1 - z_p};\ y_s = \frac{y_p \cdot z_1 - y_1 \cdot z_p}{z_1 - z_p};\ z_s = 0$$

Введемо однорідні координати:

$$x_s = \frac{x_s'}{w_s'};\ y_s = \frac{y_s'}{w_s'};\ z_s = 0;\ z_s' = z_1 - z_p$$

Тоді координати *S* можуть бути отримані з використанням множення матриць у такий спосіб:

$$x_s'\ y_s'\ 0\ w_s' \ = \ x_s\ y_s\ z_s\ 1 \begin{matrix} z_1 & 0 & 0 & 0 \\ 0 & z_1 & 0 & 0 \\ -x_1 & -y_1 & 0 & -1 \\ 0 & 0 & 0 & z_1 \end{matrix}$$

Для того, щоб алгоритм міг розраховувати тінь, що падає на довільну площину, розглянемо довільну точку на лінії між *S* і *P*, представлену в однорідних координатах:

$a \cdot P + b \cdot L$, де *a* і *b* — скалярні параметри.

Наступна матриця задає площину через координати її нормалі:

$$\boldsymbol{G} = \begin{matrix} x_n \\ y_n \\ z_n \\ d \end{matrix}$$

Точка, у якій промінь, проведений від джерела світла через дану точку *P*, перетинає площину *G*, визначається параметрами *a* і *b*, які задовольняють наступне рівняння:

$$a \cdot \boldsymbol{P} + b \cdot \boldsymbol{L} \cdot \boldsymbol{G} = 0$$

Звідси одержуємо:

$$a \cdot \boldsymbol{P} \cdot \boldsymbol{G} + b \cdot \boldsymbol{L} \cdot \boldsymbol{G} = 0$$

Цьому рівнянню задовольняють:

$$a = \boldsymbol{L} \cdot \boldsymbol{G};\ b = -\boldsymbol{P} \cdot \boldsymbol{G}$$

Отже, координати шуканої точки:

$$\boldsymbol{S} = \boldsymbol{L} \cdot \boldsymbol{G} \cdot \boldsymbol{P} - \boldsymbol{P} \cdot \boldsymbol{G} \cdot \boldsymbol{L}$$

Користуючись асоціативністю матричного добутку, одержимо:

$$\boldsymbol{S} = \boldsymbol{P} \cdot \boldsymbol{L} \cdot \boldsymbol{G} \cdot \boldsymbol{I} - \boldsymbol{G} \cdot \boldsymbol{L}$$

де *I* — одинична матриця.

Матриця $\boldsymbol{L} \cdot \boldsymbol{G} \cdot \boldsymbol{I} - \boldsymbol{G} \cdot \boldsymbol{L}$ 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 $\boldsymbol{L} \cdot \boldsymbol{G} \cdot \boldsymbol{I} - \boldsymbol{G} \cdot \boldsymbol{L}$. Наступний код використовує її для побудови тіні на заданій площині:

```
      // Робимо тіні напівпрозорими з використанням змішання кольорів
(blending)
      glEnable(GL_BLEND);
      glBlendFunc(GL_SRC_ALPHA,GL_ONE_MINUS_SRC_ALPHA);
      glDisable(GL_LIGHTING);
      glColor4f(0.0, 0.0, 0.0, 0.5);
      glPushMatrix();
         // Проектуємо тінь
         glMultMatrixf(@floorShadow);
         // Формуємо зображення сцени в проекції
```

```
   RenderGeometry();
glPopMatrix();
glEnable(GL_LIGHTING);
glDisable(GL_BLEND);
// Формуємо зображення сцени в звичайному режимі
RenderGeometry();
```

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

```
procedure ShadowMatrix(plane, lightpos:array [0..3]of GLfloat;
                              var matrix:array[0..3, 0..3] of GLfloat)
{ параметри:  plane – коефіцієнти рівняння площини
              lightpos – координати джерела світла
  повертаємо: matrix – результуюча матриця }
var
 dot: GLFloat;
begin
  dot := plane[0]*lightpos[0] + plane[1]*lightpos[1] +
         plane[2]*lightpos[2] + plane[3]*lightpos[3];
  matrix[0,0] := dot - lightpos[0]*plane[0];
  matrix[1,0] := 0.0 - lightpos[0]*plane[1];
  matrix[2,0] := 0.0 - lightpos[0]*plane[2];
  matrix[3,0] := 0.0 - lightpos[0]*plane[3];

  matrix[0,1] := 0.0 - lightpos[1]*plane[0];
  matrix[1,1] := dot - lightpos[1]*plane[1];
  matrix[2,1] := 0.0 - lightpos[1]*plane[2];
  matrix[3,1] := 0.0 - lightpos[1]*plane[3];

  matrix[0,2] := 0.0 - lightpos[2]*plane[0];
  matrix[1,2] := 0.0 - lightpos[2]*plane[1];
  matrix[2,2] := dot - lightpos[2]*plane[2];
  matrix[3,2] := 0.0 - lightpos[2]*plane[3];

  matrix[0,3] := 0.0 - lightpos[3]*plane[0];
  matrix[1,3] := 0.0 - lightpos[3]*plane[1];
  matrix[2,3] := 0.0 - lightpos[3]*plane[2];
  matrix[3,3] := dot - lightpos[3]*plane[3];
end;
```

Але тіні, побудовані таким чином, мають ряд недоліків:

- описаний алгоритм припускає, що площини нескінченні, і не відсікає тіні по границі. Наприклад, якщо деякий об’єкт відкидає тінь на стіл, вона не буде відтинатися по границі, і, тим більше, «загортатися» на бічну поверхню столу;
- у деяких місцях тіні може спостерігатися ефект «подвійного змішування» (*reblending*), тобто темні плями в тих ділянках, де спроектовані трикутники перекривають один одного;

- зі збільшенням числа поверхонь складність алгоритму різко збільшується, тому що для кожної поверхні потрібно заново будувати всю сцену, навіть якщо проблема відсікання тіней по границі буде вирішена;
- тіні звичайно мають розмиті границі, а в приведеному 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 недолік можна використати буфер маски (див. п. 6.3).

Отже, задача — відсікти виведення геометрії (тіні, у даному випадку) по границі деякої довільної області й уникнути «подвійного змішування». Загальний алгоритм вирішення задачі з використанням буфера маски такий:

1. Очищуємо буфер маски значенням 0.

```
// Очищуємо буфер маски
glClearStencil(0);
// вмикаємо тест
glEnable(GL_STENCIL_TEST);
```

2. Відображаємо задану область відсікання, встановлюючи значення в буфері маски в 1.

```
// умова завжди виконана і значення в буфері буде дорівнювати 1
glStencilFunc (GL_ALWAYS, 0, $FFFFFFF);
// у будь-якому випадку заміняємо значення в буфері маски
glStencilOp (GL_REPLACE, GL_REPLACE, GL_REPLACE);
// виводимо геометрію, по якій потім буде відсічена тінь
RenderPlane();
```

3. Малюємо тіні в тих областях, де в буфері маски встановлені значення 1. Якщо тест проходить, встановлюємо в ці області значення 2.

```
// умова виконана і тест дає істину тільки якщо
// значення в буфері маски дорівнює 1
glStencilFunc (GL_EQUAL, 0, $FFFFFFF);
// значення в буфері дорівнює 2, якщо тінь вже виведена
glStencilOp (GL_KEEP, GL_KEEP, GL_INCR);
// виводимо тіні
RenderShadow();
```

Однак, навіть при застосуванні маскування залишаються невирішеними деякі проблеми пов’язані з роботою *z*-буфера. Зокрема, окремі ділянки тіней можуть стати невидимими. Для вирішення цієї проблеми можна спробувати трохи підняти тіні над площиною за допомогою модифікації рівняння, яке описує площину. Опис інших методів виходить за рамки даного посібника.

7.3 Дзеркальні відображення

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

Алгоритм використовує інтуїтивне представлення повної сцени з дзеркалом як складеної з двох: «дійсної» і «віртуальної» — яка знаходиться за дзеркалом. Отже, процес формування відображень складається з двох частин: візуалізації звичайної сцени та побудови і візуалізації віртуальної.

Для кожного об’єкта «дійсної» сцени будується його відбитий двійник, який спостерігач і побачить у дзеркалі (рис. 7.1).

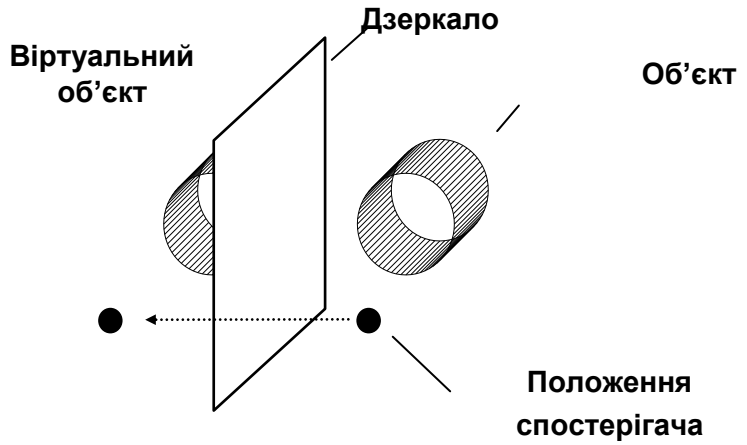


Рис. 7.1. Схема дзеркального відображення

Для ілюстрації розглянемо кімнату з дзеркалом на стіні. Кімната й об’єкти, що знаходяться в ній, виглядають у дзеркалі так, ніби дзеркало було вікном, а за ним була б ще одна така кімната з тими же об’єктами, але симетрично відображеними щодо площини, проведеної через поверхню дзеркала.

Спрощений варіант алгоритму створення плоского відображення складається з наступних кроків:

1. Формуємо сцену як звичайно, але без об’єктів-дзеркал.
2. З використанням буферу маски обмежуємо подальше виведення проекції дзеркала на екран.
3. Формуємо сцену, відображену відносно площини дзеркала. При цьому буфер маски дозволить обмежити виведення формою проекції об’єкта-дзеркала.

Ця послідовність дій дозволить одержати переконливий ефект відображення. Розглянемо етапи більш докладно.

Спочатку необхідно намалювати сцену як звичайно. Не будемо зупинятися на цьому етапі докладно. Зауважимо тільки, що, очищаючи буфери OpenGL безпосередньо перед малюванням, потрібно не забути очистити буфер маски:

```
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT | GL_STENCIL_BUFFER_BIT);
```

Під час візуалізації сцени краще не малювати об'єкти, що потім стануть дзеркальними.

На другому етапі необхідно обмежити подальше виведення проекції дзеркального об'єкта на екран.

Для цього налагоджуємо буфер маски і малюємо дзеркало

```
glEnable(GL_STENCIL_TEST);
// умова завжди виконана і значення в буфері буде дорівнювати 1
glStencilFunc(GL_ALWAYS, 1, 0);
glStencilOp(GL_KEEP, GL_KEEP, GL_REPLACE);
RenderMirrorObject();
```

У результаті ми одержали:

- у буфері кадру — коректно зображена сцена, за винятком області дзеркала;
- в області дзеркала (там, де ми хочемо бачити відображення) значення буфера маски дорівнює 1.

На третьому етапі потрібно намалювати сцену, відображену відносно площини дзеркального об'єкта.

Спочатку налагоджуємо матрицю відображення. Матриця відображення повинна дзеркально відбивати всю геометрію відносно площини, у якій лежить об'єкт-дзеркало. Її можна одержати, наприклад, за допомогою такої функції (спробуйте одержати цю матрицю самостійно як вправу):

```
procedure ReflectionMatrix(plane_point, plane_normal: array [0..2] of GLfloat; var reflection_matrix:array [0..3] of GLfloat)
var
  pv: GLfloat;
begin
  p:=plane_point[0]*plane_normal[0]+plane_point[1]*plane_normal[1]+
     plane_point[2]*plane_normal[2];

  reflection_matrix[0,0]=1-2*plane_normal[0]*plane_normal[0];
  reflection_matrix[1,0]=0-2*plane_normal[0]*plane_normal[1];
  reflection_matrix[2,0]=0-2*plane_normal[0]*plane_normal[2];
  reflection_matrix[3,0]=2*p*plane_normal[0];

  reflection_matrix[0,1]=0-2*plane_normal[0]*plane_normal[1];
  reflection_matrix[1,1]=1-2*plane_normal[1]*plane_normal[1];
  reflection_matrix[2,1]=0-2*plane_normal[1]*plane_normal[2];
```

```
  reflection_matrix[3,1]=2*p*plane_normal[1];

  reflection_matrix[0,2]=0-2*plane_normal[0]*plane_normal[2];
  reflection_matrix[1,2]=0-2*plane_normal[1]*plane_normal[2];
  reflection_matrix[2,2]=1-2*plane_normal[2]*plane_normal[2];
  reflection_matrix[3,2]=2*p*plane_normal[2];

  reflection_matrix[0,3] = 0;
  reflection_matrix[1,3] = 0;
  reflection_matrix[2,3] = 0;
  reflection_matrix[3,3] = 1;
end;
```

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

```
// умова виконана і тест дає істину тільки якщо
// значення в буфері маски дорівнює 1
glStencilFunc (GL_EQUAL, 0, $FFFFFFF);
// нічого не змінюємо в буфері
glStencilOp(GL_KEEP, GL_KEEP, GL_KEEP);
```

і малюємо сцену ще раз (без дзеркальних об’єктів)

```
glPushMatrix();
glMultMatrixf(@reflection_matrix);
RenderScene();
GlPopMatrix();
```

Нарешті, відключаємо маскування

```
glDisable(GL_STENCIL_TEST);
```

В разі потреби, після цього можна ще раз вивести дзеркальний об’єкт, наприклад, з альфа-змішуванням — для створення ефекту помутніння дзеркала і т.д.

Існує кілька модифікацій цього алгоритму, які відрізняються послідовністю дій, обмеженнями на геометрію і т.д.

Контрольні питання

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.

Розділ 8. Оптимізація програм

Для мінімізації часу формування та виведення зображень необхідно притримуватися багатьох класичних рекомендацій по програмуванню на OpenGL [3,5,6]: інтенсивно використовувати заздалегідь підготовлені списки команд, векторну форму завдання параметрів функцій, перехоплення повідомлень Windows об'єктам інтерфейсу з користувачем тощо. Загалом, тема оптимізації графічних програм є захоплюючою і практично невичерпною, а тому не може бути з достатньою детальністю розглянута в рамках даного посібника. Наведемо лише окремі поради і прийоми з підвищення надійності і продуктивності графічних застосувань на основі використання бібліотеки OpenGL.

8.1 Поради з підвищення надійності програм

В літературі та Інтернет – джерелах можна знайти багато порад з надійного програмування графічних застосувань. Наведемо лише окремі з них:

- завжди перевіряйте програми на наявність помилок. Для виявлення помилок викликайте команду `glGetError` відразу після рендерінгу сцени;
- враховуйте реакцію бібліотеки на помилки в залежності від версії OpenGL. Наприклад, OpenGl 1.1 ігнорує матричні операції, що викликаються між командами `glBegin` та `glEnd`, але наступні версії можуть реагувати іншим чином. Семантика помилок OpenGL також може змінюватись між синтаксично сумісними версіями;
- якщо необхідно згорнути усі геометричні побудови у окремій площині, то треба використовувати проекційну матрицю. При використанні матриці видового перетворення необхідний кінцевий результат не гарантується;
- не бажано робити багато послідовних змін окремої матриці. Наприклад, не виконуйте анімацію обертанням шляхом неперервного виклику команди `glRotate` із зростаючим кутом повороту. Краще використовувати команду `glLoadIdentity` для ініціалізації даної матриці у кожному кадрі, а потім викликати команду `glRotate` з необхідним кутом для цього кадру;
- не варто очікувати повідомлення про помилки під час створення списку зображень. Команди у межах списку зображень генерують помилки тільки під час використання списку;

- розміщуйте найближчу площину видимого об'єму якомога далі від точки спостереження для того, щоб оптимізувати роботу буферу глибини;
- використовуйте команду `glFlush` для примусового викликання усіх попередніх команд OpenGL;
- використовуйте весь діапазон буферу-накопичувача. Наприклад, при накопичуванні чотирьох зображень масштабуйте кожне зображення 1/4 від об'єму зображень, що накопичуються;
- якщо необхідно точна двохвимірна растеризація, то треба ретельно визначити ортогональну проекцію і вершини примітивів, які необхідно растеризувати. Ортогональна проекція повинна бути визначена з цілочисельними координатами: `gluOrtho2D(0, width, 0, height)`, де *width* та *height* — розміри області перегляду. Для растеризації дані проекційної матриці, вершини багатокутників та позиції піксельних образів повинні бути задані цілочисельними координатами;
- уникайте використання від'ємних значень координати вершини *w* та координати текстури *q*. OpenGL не може проводити вірне відсікання, а також може робити помилки при інтерполяції та тонуванні примітивів, що задані такими координатами;
- не прогнозуйте точність виконання операцій, виходячи з типів даних для параметрів команд OpenGL. Наприклад, якщо Ви використовуєте команду `glRotated`, то необов'язково, що конвеєр, який обробляє геометричні об'єкти , збереже точність, очікувану для числа з плаваючою точкою подвійної точності на протязі всієї операції. Можливо, що параметри команди `glRotated`, будуть перетворені у інші типи даних перед обробкою.

8.2 Прийоми підвищення продуктивності застосувань

Для підвищення продуктивності прикладних програм використовуйте наступні прийоми:

- використовуйте команду `glColorMaterial` тільки тоді, коли властивість матеріалу швидко змінюється (наприклад, у кожній вершині). Використовуйте команду `glMaterial` для рідких змін чи при швидкій зміні більше, ніж однієї властивості матеріалу;
- завжди використовуйте команду `glLoadIdentity` для ініціалізації матриці замість завантаження власної копії одиничної матриці;
- використовуйте спеціальні виклики матриць, такі як `glRotate`, `glTranslate`, `glScale`, замість створення власних матриць обертання, переміщення чи масштабування та виклику команди `glMultMatrix`;

- використовуйте функції запитів, коли прикладна програма вимагає тільки декілька значень параметрів стану для своїх обчислень. Якщо програмі потрібно декілька значень з однієї групи атрибутів, то використовуйте команди `glPushAttrib` та `glPopAttrib`, для збереження та відновлення значень;
- використовуйте списки зображень для інкапсуляції викликів рендерінга жорстко заданих об'єктів, які будуть багаторазово відображуватись;
- 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 у клієнт-серверних середовищах;
- встановлюйте нормалі одиничної довжини, якщо це можливо, та уникайте додаткових витрат режиму `GL_NORMALIZE`. Уникайте використання команди `glScale` при застосуванні освітлення, тому що для нормальної роботи у цьому випадку необхідне включення режиму `GL_NORMALIZE`;
- встановіть режим `glShadeModel` у стан `GL_FLAT`, якщо не потрібен режим згладжування (*smooth shading*);
- використовуйте, якщо можливо, один виклик команди `glClear` для одного кадру;
- використовуйте один виклик `glBegin(GL_TRIANGLES)` для відображення множини незалежних трикутників замість використання множини викликів `glBegin(GL_TRIANGLES)` чи `glBegin(GL_POLYGON)`. Аналогічно застосовуйте виклики команд `glBegin(GL_QUADS)` та `glBegin(GL_LINES)`;
- застосування масивів вершин зменшує додаткові витрати на виклик функцій;
- використовуйте векторні форми команд, для передачі попередньо обчислених даних, та скалярні форми команд, для того щоб передати значення, що будуть обчислені відразу після виклику;
- якщо не має необхідності, уникайте використання різних режимів для передньої (*front*) та задньої (*back*) сторін многокутників.

Контрольні питання

1. Перелічіть відомі Вам поради з підвищення надійності графічних застосувань.
2. Які Вам відомі прийоми підвищення продуктивності застосувань?
3. Як підвищити ефективність виконання афінних перетворень в OpenGL?

Додаток А. Структура GLUT-застосування

Розглянемо побудову консольного застосування за допомогою бібліотеки GLUT. Як зазначалося раніше, ця бібліотека не входить до складу бібліотек, що поставляються фірмою Microsoft. Тому для підключення бібліотеки необхідно отримати дистрибутив бібліотеки та провести модифікацію заголовного файлу. При модифікації заголовних файлів необхідно змінити значення констант opengl32 та glu32.

Бібліотека GLUT забезпечує єдиний інтерфейс для роботи з вікнами незалежно від платформи, тому описувана нижче структура застосування залишається незмінною для операційних систем Windows, Linux та інших.

За своїм призначенням функції GLUT можуть бути класифіковані на кілька груп:

- ініціалізація;
- початок обробки подій;
- керування вікнами;
- керування меню;
- реєстрація функцій із зворотнім викликом;
- керування індексованою палітрою кольорів;
- відображення шрифтів;
- відображення додаткових геометричних фігур (тор, конус та ін.).

Ініціалізація забезпечується функцією:

```
glutInit (int *argcp, char **argv)
```

Змінна *argcp* це покажчик на стандартну змінну *argc*, що описується в функції main(), а *argv* — покажчик на параметри, передані програмі при запуску, які описуються там же. Ця функція проводить необхідні початкові дії для побудови вікна застосування і лише кілька функцій GLUT можуть бути викликані до неї. До них відносяться:

```
glutInitWindowPosition(int x, int y)
glutInitWindowSize(int width, int height)
glutInitDisplayMode(unsigned int mode)
```

Перші дві функції задають відповідно положення і розмір вікна, а остання функція визначає різні режими відображення інформації, що можуть спільно задаватися з використанням операції побітового «АБО» (|):

GLUT_RGBA	режим RGBA. Використовується за замовчуванням, якщо не встановлені явно режими GLUT_RGBA або GLUT_INDEX
GLUT_RGB	те ж, що і GLUT_RGBA
GLUT_INDEX	режим індексованих кольорів (використання палітри). Ска-

	совує GLUT_RGBA
GLUT_SINGLE	вікно з одиночним буфером. Використовується за умовчуванням
GLUT_DOUBLE	вікно з подвійним буфером. Скасовує GLUT_SINGLE
GLUT_STENCIL	вікно з буфером маски
GLUT_ACCUM	вікно з буфером-накопичувачем
GLUT_DEPTH	вікно з буфером глибини

Це неповний список значень параметрів для даної функції, однак для більшості випадків цього буває досить.

Робота з буфером маски і буфером-накопичувачем описана в розділі 6.

Функції бібліотеки GLUT реалізують так званий подійно-керований механізм. Це означає, що є деякій внутрішній цикл, що запускається після відповідної ініціалізації й обробляє одну за одною всі події, оголошені під час ініціалізації. До подій відносяться: клік миші, закриття вікна, зміна властивостей вікна, зміна позиції курсору, натискання клавіші, і «порожня» (*idle*) подія, коли нічого не відбувається. Для проведення періодичної перевірки здійснення тієї чи іншої події треба зареєструвати функцію, що буде його обробляти. Для цього використовуються функції виду:

```
void glutDisplayFunc(void (*func)(void))
void glutReshapeFunc(void (*func)(int width, int height))
void glutMouseFunc (void (*func)(int button, int state, int x, int y))
void glutIdleFunc(void (*func)(void))
void glutMotionFunc(void (*func)(int x, int y))
void glutPassiveMotionFunc(void (*func)(int x, int y))
```

Параметром для них є ім'я відповідної функції заданого типу. За допомогою glutDisplayFunc задається функція малювання для вікна застосування, що викликається при необхідності створення чи оновлення зображення. Для явного виклику оновлення вікна, іноді зручно використовувати функцію glutPostRedisplay.

Через glutReshapeFunc встановлюється функція обробки зміни розмірів вікна користувачем, якій передаються нові розміри.

glutMouseFunc визначає функцію — обробник команд від миші, а glutIdleFunc задає функцію, що буде викликатися щоразу, коли немає подій від користувача.

Функція, визначена glutMotionFunc викликається тоді, коли користувач рухає мишу, утримуючи натиснутою кнопку. glutPassiveMotionFunc ре-

єструє функцію, що викликається, коли користувач рухає мишу і не натиснуто жодної кнопки.

Контроль усіх подій відбувається усередині нескінченного циклу в функції `glutMainLoop`, яка звичайно викликається наприкінці будь-якої програми, що використовує GLUT.

Додаток В. Примітиви бібліотек GLU і GLUT

Розглянемо стандартні команди побудови примітивів, що реалізовані в бібліотеках GLU і GLUT.

Щоб побудувати примітив з бібліотеки GLU, треба спочатку створити покажчик на quadric-об'єкт за допомогою команди `gluNewQuadric`, а потім викликати одну з команд `gluSphere`, `gluCylinder`, `gluDisk`, `gluPartialDisk`.

Розглянемо ці команди окремо:

```
gluSphere(qobj: ^GLUquadricObj, radius: GLdouble, slices: GLint, stacks:
                  GLint)
```

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

```
gluCylinder(qobj:   ^GLUquadricObj,   baseRadius:   GLdouble,   topRadius:
                  GLdouble,   height:   GLdouble,   slices:   GLint,   stacks:
                  GLint)
```

Дана функція будує циліндр без основ (тобто кільце), поздовжня вісь рівнобіжна осі *Z*, задня основа має радіус *baseRadius*, і розташована в площині *Z* = 0, а передня основа має радіус *topRadius* і розташована в площині *Z* = *height*. Якщо задати один з радіусів рівним нулю, то буде побудований конус.

Параметри *slices* і *stacks* мають аналогічну семантику, що й у попередній команді.

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

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

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

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

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

```
glutSolidSphere(radius: GLdouble, slices: GLint, stacks: GLint)
glutWireSphere(radius: GLdouble, slices: GLint, stacks: GLint)
```

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

```
glutSolidCube(size: GLdouble)
glutWireCube(size: GLdouble)
```

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

```
glutSolidCone(base: GLdouble, height: GLdouble, slices: GLint, stacks:
                  GLint)
glutWireCone(base: GLdouble, height:GLdouble, slices: GLint, stacks: GLint)
```

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

```
glutSolidTorus(innerRadius: GLdouble, outerRadius: GLdouble, nsides: GLint,
                  rings: GLint)
glutWireTorus(innerRadius: GLdouble, outerRadius: GLdouble, nsides: GLint,
                  rings: GLint)
```

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

```
glutSolidTetrahedron
glutWireTetrahedron
```

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

```
glutSolidOctahedron
glutWireOctahedron
```

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

```
glutSolidDodecahedron
glutWireDodecahedron
```

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

```
glutSolidIcosahedron
glutWireIcosahedron
```

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

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

Додаток С. Демонстраційні програми

С 1. Приклад 1: Модель освітлення OpenGL

Програма призначена для демонстрації моделі освітлення OpenGL на прикладі спрощеної астрономічної моделі «Червоний малюк», що складається з зірки та планети, що обертається навколо неї (зірка та планета створені за допомогою quadric-об'єктів). У програмі задаються характеристики джерела світла та характеристики матеріалів. Текст програми:

```
unit Main;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
Dialogs,
  OpenGL, Menus, ExtCtrls;
type
  TfrmGL = class(TForm)
    Timer1: TTimer;
    procedure FormCreate(Sender: TObject);
    procedure FormPaint(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
    procedure FormKeyDown(Sender: TObject; var Key: Word;
      Shift: TShiftState);
    procedure FormResize(Sender: TObject);
    procedure Timer1Timer(Sender: TObject);
  private
    DC : HDC;
    hrc: HGLRC;
    quadObj : GLUquadricObj;
  end;
var
  frmGL: TfrmGL;
  year : Integer = 0;
  day : Integer = 0;
implementation
{$R *.DFM}
procedure yearAdd;
begin
    year := (year + 5);
    If year > 360 then year := 0;
end;
{Перемалювання вікна}
procedure TfrmGL.FormPaint(Sender: TObject);
const
  sColor: array [0..3] of GLfloat = (1, 0.75, 0, 1);
  pColor: array [0..3] of GLfloat = (0.4, 0, 0.2, 1);
  mColor: array [0..3] of GLfloat = (0.7, 0.4, 0.5, 1);
  black : array [0..3] of GLfloat = (0, 0, 0, 1);
begin
  // очищення буферу кольору та глибини
```

```
  glClear(GL_COLOR_BUFFER_BIT OR GL_DEPTH_BUFFER_BIT);
  glPushMatrix;
    // малюємо сонце
    glPushMatrix;
      glRotatef (90.0, 1.0, 0.0, 0.0);  // повертаємо сонце прямо
      //задаємо випромінювання світла
      glMaterialfv(GL_FRONT_AND_BACK, GL_EMISSION, @sColor);
      gluSphere (quadObj, 1.0, 15, 10);
      //вимикаємо випромінювання
      glMaterialfv(GL_FRONT_AND_BACK, GL_EMISSION, @black);
    glPopMatrix;
    // малюємо червону планету
    glRotatef (year, 0.0, 1.0, 0.0);
    glTranslatef (2.0, 0.0, 0.0);
    glRotatef (90.0, 1.0, 0.0, 0.0);  // повертаємо прямо
    //встановлення характеристик відбиття світла для планети
    glMaterialfv(GL_FRONT,GL_AMBIENT_AND_DIFFUSE, @pColor);
    gluSphere (quadObj, 0.2, 10, 10);
    glMaterialfv(GL_FRONT, GL_AMBIENT_AND_DIFFUSE, @mColor);
  glPopMatrix;
  SwapBuffers(DC);
end;
{Встановдення формату пікселя}
procedure SetDCPixelFormat (hdc : HDC);
var
  pfd : TPixelFormatDescriptor;
  nPixelFormat : Integer;
begin
  FillChar (pfd, SizeOf (pfd), 0);
  pfd.dwFlags  := PFD_DRAW_TO_WINDOW or PFD_SUPPORT_OPENGL or
                  PFD_DOUBLEBUFFER;
  nPixelFormat := ChoosePixelFormat (hdc, @pfd);
  SetPixelFormat (hdc, nPixelFormat, @pfd);
end;
{Створення форми}
procedure TfrmGL.FormCreate(Sender: TObject);
const
  lcol : Array [0..3] of GLFloat = (0.97, 0.956, 0, 1);
  lpos : Array [0..3] of GLFloat = (0, 0, 0, 1.5);
  spec : array [0..3] of GLfloat = (0.5, 0.5, 0.5, 0.5);
begin
  DC := GetDC (Handle);
  SetDCPixelFormat(DC);
  hrc := wglCreateContext(DC);
  wglMakeCurrent(DC, hrc);
  quadObj := gluNewQuadric; //створюємо quadric-об'єкт
  gluQuadricDrawStyle (quadObj, GLU_Fill);
  // Задаємо характеристики світла
  glLightfv(GL_LIGHT0, GL_POSITION, @lpos);
  glLightfv(GL_LIGHT0, GL_DIFFUSE, @lcol);
  glMaterialf(GL_FRONT, GL_SHININESS, 60);
  glMaterialfv(GL_FRONT, GL_SPECULAR, @spec);
```

```
  glShadeModel (GL_SMOOTH); //встановлюємо режим згладжування
  glEnable(GL_LIGHTING); // вмикаємо режим освітлення
  glEnable(GL_LIGHT0); //вмикаємо джерело освітлення
  glEnable(GL_DEPTH_TEST); // встановлюємо режим тестування глибини
  Timer1.Enabled:=true;
end;
{Закінчення роботи програми}
procedure TfrmGL.FormDestroy(Sender: TObject);
begin
  gluDeleteQuadric (quadObj); //видаляємо quadric-об'єкт
  //звільнюємо контексти пристрою та відображення
  wglMakeCurrent(0, 0);
  wglDeleteContext(hrc);
  ReleaseDC (Handle, DC);
  DeleteDC (DC);
end;
procedure TfrmGL.FormKeyDown(Sender: TObject; var Key: Word;
  Shift: TShiftState);
begin
  If Key = VK_ESCAPE then Close;
end;
procedure TfrmGL.FormResize(Sender: TObject);
begin
  glViewport(0, 0, ClientWidth, ClientHeight);
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity;
  gluPerspective(60.0, ClientWidth / ClientHeight, 1.0, 20.0);
  glMatrixMode(GL_MODELVIEW);
  glLoadIdentity;
  glTranslatef (0.0, 0.0, -5.0);
  InvalidateRect(Handle, nil, False);
end;
procedure TfrmGL.Timer1Timer(Sender: TObject);
begin
  yearAdd;
  InvalidateRect(Handle, nil, False);
end;
end.
```

С 2. Приклад 2: Накладення текстури

Результатом виконання цієї програми є побудова куба, на грані якого нанесена текстура. Текстура задається растровим зображенням (рис. С2.1) розміром 384×256 пікселів. Дане зображення розрізається на окремі зображення розміром 64×64 пікселів, які й наносяться на грані куба.

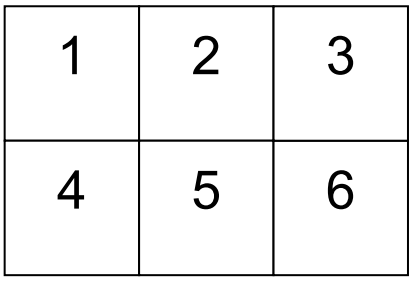


Рис. С2.1. Растрове зображення текстури

Текст програми:

```
unit Main;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
  Dialogs,OpenGl, ExtCtrls;
type
  TForm1 = class(TForm)
    Timer1: TTimer;
    procedure FormCreate(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
    procedure FormResize(Sender: TObject);
    procedure FormPaint(Sender: TObject);
    procedure FormKeyDown(Sender: TObject; var Key: Word;
      Shift: TShiftState);
    procedure Timer1Timer(Sender: TObject);
  private
    { Private declarations }
    DC : HDC;
    hrc: HGLRC;
  public
    { Public declarations }
  end;
type
  TSquare = class
  private
    { Private declarations }
  public
    { Public declarations }
  procedure PrepareImage(bmap: string);
  procedure initlist(CB: GLint);
  procedure SiccorsBitMap(bm:string; src_x1, src_y1, src_x2,
                          src_y2:integer);
end;
var
  Form1: TForm1;
  Square1:TSquare;
implementation
{$R *.DFM}
{Встановлення формату пікселя}
```

```
procedure SetDCPixelFormat (hdc : HDC);
var
  pfd : TPixelFormatDescriptor;
  nPixelFormat : Integer;
begin
  FillChar (pfd, SizeOf (pfd), 0);
  pfd.dwFlags  := PFD_DRAW_TO_WINDOW or PFD_SUPPORT_OPENGL or
                  PFD_DOUBLEBUFFER;
  nPixelFormat := ChoosePixelFormat (hdc, @pfd);
  SetPixelFormat (hdc, nPixelFormat, @pfd);
end;
procedure TForm1.FormCreate(Sender: TObject);
begin
  DC := GetDC (Handle);
  SetDCPixelFormat(DC);
  hrc := wglCreateContext(DC);
  wglMakeCurrent(DC, hrc);
  glShadeModel (GL_FLAT);
  Square1:=TSquare.Create;
  Square1.initlist(1);
  glEnable(GL_DEPTH_TEST);
  Timer1.Enabled:=true;
end;
procedure TForm1.FormDestroy(Sender: TObject);
begin
  // видалення списку
  glDeleteLists(1,1);
  //звільнюємо контексти пристрою та відображення
  wglMakeCurrent(0, 0);
  wglDeleteContext(hrc);
  ReleaseDC (Handle, DC);
  DeleteDC (DC);
end;
procedure TForm1.FormResize(Sender: TObject);
begin
  glViewport(0, 0, ClientWidth, ClientHeight);
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity;
  gluPerspective(60.0, ClientWidth / ClientHeight, 1.0, 20.0);
  glMatrixMode(GL_MODELVIEW);
  glLoadIdentity;
  glTranslatef (0.0, 0.0, -5.0);
  InvalidateRect(Handle, nil, False);
end;
procedure TForm1.FormPaint(Sender: TObject);
begin
  // очищення буферу кольору та глибини
  glClear(GL_COLOR_BUFFER_BIT or GL_DEPTH_BUFFER_BIT);
  glRotatef (2, 0.0, 1.0, 0.0);
  glCallList(1);//виклик створеного списку багатокутника
  SwapBuffers(DC);
end;
```

```
procedure TForm1.FormKeyDown(Sender: TObject; var Key: Word;
  Shift: TShiftState);
begin
 If Key = VK_ESCAPE then Close;
end;
// Підготовка текстури
procedure TSquare.PrepareImage(bmap: string);
type
  PPixelArray = ^TPixelArray;
  TPixelArray = array [0..0] of Byte;
var
  Bitmap : TBitmap;
  Data : PPixelArray;
  BMInfo : TBitmapInfo;
  I, ImageSize : Integer;
  Temp : Byte;
  MemDC : HDC;
begin
  Bitmap := TBitmap.Create;
  Bitmap.LoadFromFile (bmap);
  with BMinfo.bmiHeader do begin
    FillChar (BMInfo, SizeOf(BMInfo), 0);
    biSize := sizeof (TBitmapInfoHeader);
    biBitCount := 24;
    biWidth := Bitmap.Width;
    biHeight := Bitmap.Height;
    ImageSize := biWidth * biHeight;
    biPlanes := 1;
    biCompression := BI_RGB;
    MemDC := CreateCompatibleDC (0);
    GetMem (Data, ImageSize * 3);
    try
      GetDIBits (MemDC, Bitmap.Handle, 0, biHeight, Data,
                 BMInfo, DIB_RGB_COLORS);
      For I := 0 to ImageSize - 1 do begin
          Temp := Data [I * 3];
          Data [I * 3] := Data [I * 3 + 2];
          Data [I * 3 + 2] := Temp;
      end;
      glTexImage2d(GL_TEXTURE_2D, 0, 3, biWidth,
                   biHeight, 0, GL_RGB, GL_UNSIGNED_BYTE, Data);
    finally
      FreeMem (Data);
      DeleteDC (MemDC);
      Bitmap.Free;
    end;
  end;
end;
//процедура розрізання вихідного зображення
procedure TSquare.SiccorsBitmap(bm:string; src_x1, src_y1, src_x2,
                                src_y2:integer);
var LoadBitmap,CopyBitmap,SaveBitmap: TBitmap;
```

```
        DestRect, SrcRect: TRect;
    begin
      LoadBitmap:=TBitmap.Create;
      CopyBitmap:=TBitmap.Create;
      SaveBitmap:=TBitmap.Create;
      CopyBitmap.Height:=64;
      CopyBitmap.Width:=64;
      LoadBitmap.LoadFromFile(bm);
      CopyBitmap.Canvas.CopyMode:=cmSrcCopy;
      DestRect:=Rect(0,0,64,64);
      SrcRect:=Rect(src_x1,src_y1,src_x2,src_y2);
      Copybitmap.Canvas.BrushCopy(DestRect,     LoadBitmap,     SrcRect,
clBlack);
      CopyBitmap.Canvas.CopyRect(DestRect,LoadBitmap.Canvas,SrcRect);
      SaveBitmap.Assign(CopyBitmap);
      SaveBitmap.SaveToFile('xxx.bmp');
      CopyBitmap.free;
      SaveBitmap.free;
      LoadBitmap.free;
    end;
    procedure TSquare.initlist(CB: GLint);
    begin
      //встановлення параметрів текстури
      glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_linear);
      glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_linear);
      glTexEnvi(GL_Texture_ENV, GL_Texture_Env_Mode, Gl_Decal);
      glRenderMode(GL_RENDER);
      glEnable(GL_TEXTURE_2D);
      //Створення та ініціалізація елементу списка
      glNewList(CB, GL_COMPILE);
        glColor3f (0.4, 0.5, 0.2);
        glPolygonMode (GL_Front,Gl_Fill);
        glPushMatrix;
          glRotatef(-45.0,1.0, 0.0, 0.0);
          glRotatef(225,0.0, 0.0, 1.0);
          SiccorsBitmap('numbers.bmp',0,0,128,128);
          prepareImage('xxx.bmp');
          glBegin(GL_QUADS);
            glNormal3f(-1.0,  0.0,  0.0);
            glTexCoord2d (1.0, 0.0); glVertex3f(-0.5, -0.5,-0.5); //1
            glTexCoord2d (1.0, 1.0); glVertex3f(-0.5, -0.5, 0.5); //2
            glTexCoord2d (0.0, 1.0); glVertex3f(-0.5,  0.5, 0.5); //3
            glTexCoord2d (0.0, 0.0); glVertex3f(-0.5,  0.5,-0.5); //4
          glEnd;
          SiccorsBitmap('numbers.bmp',128,0,256,128);
          prepareImage('xxx.bmp');
          glBegin(GL_QUADS);
            glNormal3f( 0.0,  1.0,  0.0);
            glTexCoord2d (1.0, 0.0); glVertex3f(-0.5,0.5,-0.5); //4
            glTexCoord2d (1.0, 1.0); glVertex3f(-0.5, 0.5, 0.5); //3
            glTexCoord2d (0.0, 1.0); glVertex3f( 0.5,  0.5, 0.5); //5
            glTexCoord2d (0.0, 0.0); glVertex3f( 0.5,  0.5,-0.5); //6
```

```
        glEnd;
        SiccorsBitmap('numbers.bmp',256,0,384,128);
        prepareImage('xxx.bmp');
        glBegin(GL_QUADS);
          glNormal3f( 1.0,  0.0,  0.0);
          glTexCoord2d (1.0, 1.0); glVertex3f( 0.5, 0.5,-0.5); //6
          glTexCoord2d (1.0, 0.0); glVertex3f( 0.5,  0.5, 0.5); //5
          glTexCoord2d (0.0, 0.0); glVertex3f( 0.5, -0.5, 0.5); //7
          glTexCoord2d (0.0, 1.0); glVertex3f( 0.5, -0.5,-0.5); //8
        glEnd;
        SiccorsBitmap('numbers.bmp',0,128,128,256);
        prepareImage('xxx.bmp');
        glBegin(GL_QUADS);
          glNormal3f( 0.0, -1.0,  0.0);
          glTexCoord2d (1.0, 0.0); glVertex3f( 0.5, -0.5,-0.5); //8
          glTexCoord2d (1.0, 1.0); glVertex3f( 0.5, -0.5, 0.5); //7
          glTexCoord2d (0.0, 1.0); glVertex3f(-0.5, -0.5, 0.5); //2
          glTexCoord2d (0.0, 0.0); glVertex3f(-0.5, -0.5,-0.5); //1
        glEnd;
        SiccorsBitmap('numbers.bmp',256,128,384,256);
        prepareImage('xxx.bmp');
        glBegin(GL_QUADS);
          glNormal3f( 0.0,  0.0,  1.0);
          glTexCoord2d (0.0, 1.0); glVertex3f( 0.5, -0.5, 0.5); //7
          glTexCoord2d (0.0, 0.0); glVertex3f( 0.5, 0.5, 0.5); //5
          glTexCoord2d (1.0, 0.0); glVertex3f(-0.5,  0.5, 0.5); //3
          glTexCoord2d (1.0, 1.0); glVertex3f(-0.5, -0.5, 0.5); //2
        glEnd;
        SiccorsBitmap('numbers.bmp',128,128,256,256);
        prepareImage('xxx.bmp');
        glBegin(GL_QUADS);
          glNormal3f( 0.0,  0.0, -1.0);
          glTexCoord2d (0.0, 1.0); glVertex3f( 0.5,  0.5,-0.5);
          glTexCoord2d (0.0, 0.0); glVertex3f( 0.5, -0.5,-0.5);
          glTexCoord2d (1.0, 0.0); glVertex3f(-0.5, -0.5,-0.5);
          glTexCoord2d (1.0, 1.0);
          glVertex3f(-0.5,  0.5,-0.5);
        glEnd;
      glPopMatrix;
    glEndList;
end;
procedure TForm1.Timer1Timer(Sender: TObject);
begin
  InvalidateRect(Handle, nil, False);
end;
end.
```

С 3. Приклад 3: Демонстрація ефекту тіні

Дана програма демонструє малювання тіні для простого об'єкту. У програмі описана користувацька процедура для малювання тіні з урахуванням того, що усі грані куба паралельні координатним площинам. Тінь

малюється у вигляді 6 окремих сірих багатокутників, для кожної грані об'єкту. Також у програмі використовуються окремі функції бібліотеки GLUT.

Текст програми:

```
unit Main;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
  Dialogs, ExtCtrls, ComCtrls, StdCtrls, Menus, Buttons, OpenGL;
const
  // колір туману
  fogColor : Array [0..3] of GLfloat = (0.5, 0.5, 0.5, 1.0);
  // колір площини
  glfSquareAmbient:Array[0..3]of GLfloat=(0.247,0.199,0.0745,1.0);
  glfSquareDiffuse:Array[0..3]of GLfloat=(0.751,0.606,0.22648,1.0);
  glfSquareSpecular:Array[0..3]of GLfloat=(0.6282,0.556,0.366,1.0);
  // джерело світла
  glfLightAmbient:Array[0..3] of GLfloat = (0.25, 0.25, 0.25, 1.0);
  glfLightDiffuse : Array[0..3] of GLfloat = (1.0, 1.0, 1.0, 1.0);
  glfLightSpecular: Array[0..3] of GLfloat = (1.0, 1.0, 1.0, 1.0);
  glfLightPosition: Array[0..3] of GLfloat = (0.0, 0.0, 20.0, 1.0);
  glfLightModelAmbient:Array[0..3] of GLfloat=(0.25,0.25,0.25,1.0);
  // позиція першого джерела світла
  LightPosition : Array[0..3] of GLfloat = (0.0, 0.0, 15.0, 1.0);
  // позиція другого джерела світла
  glfLight1Position: Array[0..3]of GLfloat=(15.0, 15.0, -5.0, 1.0);
type
  TfrmGL = class(TForm)
    procedure Init;
    procedure SetProjection(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure SetDCPixelFormat;
    procedure FormDestroy(Sender: TObject);
    procedure FormKeyDown(Sender: TObject; var Key: Word;
                          Shift: TShiftState);
  public
    DC : HDC;
    hrc : HGLRC;
    cubeX, cubeY, cubeZ : GLfloat;
    cubeL, cubeH, cubeW : GLfloat;
    AddX, AddY, AddZ : GLfloat;            // початкові зсуви
    SquareLength : GLfloat;                // площина
    procedure Square;
    procedure Shadow;
  protected
    procedure WMPaint(var Msg: TWMPaint); message WM_PAINT;
  end;
var
  frmGL: TfrmGL;
implementation
uses DGLUT;
```

```
{$R *.DFM}
{Реакція на події клавіатури}
procedure TfrmGL.FormKeyDown(Sender: TObject; var Key: Word;
                             Shift: TShiftState);
begin
  If Key = VK_ESCAPE then Close;
  If Key = VK_LEFT then begin
     cubeX := cubeX + 0.1; InvalidateRect(Handle, nil, False);
  end;
  If Key = VK_RIGHT then begin
     cubeX := cubeX - 0.1; InvalidateRect(Handle, nil, False);
  end;
  If Key = VK_UP then begin
     cubeZ := cubeZ + 0.1; InvalidateRect(Handle, nil, False);
  end;
  If Key = VK_DOWN then begin
     cubeZ := cubeZ - 0.1; InvalidateRect(Handle, nil, False);
  end;
end;
{Малювання тіні}
procedure TfrmGL.Shadow;
  // розрахунок точки тіні для однієї точки об'єкта
  procedure OneShadow (x, y, z, h : GLfloat; var x1, y1 : GLfloat);
  begin
    x1 := x * LightPosition [2] / (LightPosition [2] - (z + h));
    If LightPosition [0] < x
       then begin If x1 > 0 then x1 := LightPosition [0] + x1 end
       else begin If x1 > 0 then x1 := LightPosition [0] - x1 end;
    y1 := y * LightPosition [2] / (LightPosition [2] - (z + h));
    If LightPosition [1] < y
       then begin If y1 > 0 then y1 := LightPosition [1] + y1 end
       else begin If y1 > 0 then y1 := LightPosition [1] - y1 end;
    If x1 < 0 then x1 := 0 else
       If x1 > SquareLength then x1 := SquareLength;
    If y1 < 0 then y1 := 0 else
       If y1 > SquareLength then y1 := SquareLength;
  end;
var
  x1, y1, x2, y2, x3, y3, x4, y4 : GLfloat;
  wrkx1, wrky1, wrkx2, wrky2, wrkx3, wrky3, wrkx4, wrky4 : GLfloat;
begin
  OneShadow (cubeX + cubeL, cubeY + cubeH, cubeZ, cubeW, x1, y1);
  OneShadow (cubeX, cubeY + cubeH, cubeZ, cubeW, x2, y2);
  OneShadow (cubeX, cubeY, cubeZ, cubeW, x3, y3);
  OneShadow (cubeX + cubeL, cubeY, cubeZ, cubeW, x4, y4);
  If cubeZ + cubeW >= 0 then
  begin
    glBegin (GL_QUADS);
      glVertex3f (x1, y1, -0.99); glVertex3f (x2, y2, -0.99);
      glVertex3f (x3, y3, -0.99); glVertex3f (x4, y4, -0.99);
    glEnd;
  end;
```

```
  If cubeZ >= 0 then
  begin
     wrkx1 := x1; wrky1 := y1; wrkx2 := x2; wrky2 := y2;
     wrkx3 := x3; wrky3 := y3; wrkx4 := x4; wrky4 := y4;
     OneShadow (cubeX + cubeL, cubeY + cubeH, cubeZ, 0, x1, y1);
     OneShadow (cubeX, cubeY + cubeH, cubeZ, 0, x2, y2);
     OneShadow (cubeX, cubeY, cubeZ, 0, x3, y3);
     OneShadow (cubeX + cubeL, cubeY, cubeZ, 0, x4, y4);
     glBegin (GL_QUADS);
       glVertex3f (x1, y1, -0.99); glVertex3f (x2, y2, -0.99);
       glVertex3f (x3, y3, -0.99); glVertex3f (x4, y4, -0.99);
       glVertex3f (wrkx2, wrky2, -0.99);
       glVertex3f (x2, y2, -0.99); glVertex3f (x3, y3, -0.99);
       glVertex3f (wrkx3, wrky3, -0.99);
       glVertex3f (wrkx1, wrky1, -0.99);
       glVertex3f (wrkx4, wrky4, -0.99);
       glVertex3f (x4, y4, -0.99); glVertex3f (x1, y1, -0.99);
       glVertex3f (wrkx1, wrky1, -0.99);
       glVertex3f (x1, y1, -0.99); glVertex3f (x2, y2, -0.99);
       glVertex3f (wrkx2, wrky2, -0.99);
       glVertex3f (wrkx3, wrky3, -0.99);
       glVertex3f (x3, y3, -0.99); glVertex3f (x4, y4, -0.99);
       glVertex3f (wrkx4, wrky4, -0.99);
     glEnd;
  end;
end;
{Малювання площини}
procedure TfrmGL.Square;
begin
  glPushAttrib (GL_ALL_ATTRIB_BITS );
    glMaterialfv(GL_FRONT, GL_AMBIENT, @glfSquareAmbient);
    glMaterialfv(GL_FRONT, GL_DIFFUSE, @glfSquareDiffuse);
    glMaterialfv(GL_FRONT, GL_SPECULAR, @glfSquareSpecular);
    glMaterialf (GL_FRONT, GL_SHININESS, 90.2);
    glBegin(GL_QUADS);
      glNormal3f(squarelength / 2.0, squarelength / 2.0, -1.0);
      glVertex3f(squarelength, squarelength, -1.0);
      glVertex3f(0.0, squarelength, -1.0);
      glVertex3f(0.0, 0.0, -1.0);
      glVertex3f(squarelength, 0.0, -1.0);
    glEnd;
  glPopAttrib;
end;
{Ініціалізація}
procedure TfrmGL.Init;
begin
  glEnable (GL_FOG); glEnable(GL_NORMALIZE);
  glEnable(GL_DEPTH_TEST);
  glLightfv(GL_LIGHT0, GL_AMBIENT, @glfLightambient);
  glLightfv(GL_LIGHT0, GL_DIFFUSE, @glfLightdiffuse);
  glLightfv(GL_LIGHT0, GL_SPECULAR, @glfLightspecular);
  glLightfv(GL_LIGHT0, GL_POSITION, @glfLightposition);
```

```
  glLightModelfv(GL_LIGHT_MODEL_AMBIENT, @glfLightmodelambient);
  // друге джерело світла
  glLightfv(GL_LIGHT1, GL_AMBIENT, @glfLightambient);
  glLightfv(GL_LIGHT1, GL_DIFFUSE, @glfLightdiffuse);
  glLightfv(GL_LIGHT1, GL_SPECULAR, @glfLightspecular);
  glLightfv(GL_LIGHT1, GL_POSITION, @glfLight1position);
  glEnable(GL_LIGHTING); glEnable(GL_LIGHT0); glEnable(GL_LIGHT1);
end;
{Зміна розмірів вікна}
procedure TfrmGL.SetProjection(Sender: TObject);
begin
  glViewport(0, 0, ClientWidth, ClientHeight);
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity;
  glFrustum (-0.5, 0.5, -0.5, 0.5, 1.0, 100.0);
  glMatrixMode(GL_MODELVIEW);
  glLoadIdentity;
  glTranslatef(0.0, 0.0, -32.0); glRotatef(120.0, 1.0, 0.0, 0.0);
  glRotatef(180.0, 0.0, 1.0, 0.0); glRotatef(40.0, 0.0, 0.0, 1.0);
  InvalidateRect(Handle, nil, False);
end;
{Початок роботи програми}
procedure TfrmGL.FormCreate(Sender: TObject);
begin
  DC := GetDC(Handle);
  SetDCPixelFormat;
  hrc := wglCreateContext(DC);
  wglMakeCurrent(DC, hrc);
  Init;
  // параметри туману
  glFogi(GL_FOG_MODE, GL_EXP);
  glFogfv(GL_FOG_COLOR, @fogColor);
  glFogf(GL_FOG_DENSITY, 0.015);
  SquareLength := 50.0;
  AddX := 0; AddY := 0; AddZ := 0;
  cubeX := 1.0; cubeY := 2.0; cubeZ := 3.0;
  cubeL := 1.0; cubeH := 2.0; cubeW := 3.0;
end;
{Аналог події OnPaint}
procedure TfrmGL.WMPaint(var Msg: TWMPaint);
var
  ps : TPaintStruct;
const
  CubeColor : Array [0..3] of GLfloat = (1.0, 0.0, 0.0, 0.0);
begin
  BeginPaint(Handle, ps);
    glClear(GL_COLOR_BUFFER_BIT or GL_DEPTH_BUFFER_BIT);
    glPushMatrix;
      glTranslatef(AddX, AddY, AddZ);
      glEnable (GL_LIGHT1);
        Square;
      glDisable (GL_LIGHT1);
```

```
      glPushAttrib (GL_ALL_ATTRIB_BITS );
        glMaterialfv(GL_FRONT, GL_AMBIENT_AND_DIFFUSE, @CubeColor);
        glPushMatrix;
          glTranslatef (cubeX, cubeY, cubeZ);
          glScalef (cubeL, cubeH, cubeW);
          glutSolidCube (1.0);
        glPopMatrix;
      glPopAttrib;
      Shadow;
    glPopMatrix;
    SwapBuffers(DC);
  EndPaint(Handle, ps);
end;
{Кінець роботи програми}
procedure TfrmGL.FormDestroy(Sender: TObject);
begin
  wglMakeCurrent(0, 0);
  wglDeleteContext(hrc);
  ReleaseDC(Handle, DC);
  DeleteDC (DC);
end;
{Формат пікселів}
procedure TfrmGL.SetDCPixelFormat;
var
  nPixelFormat: Integer;
  pfd: TPixelFormatDescriptor;
begin
  FillChar(pfd, SizeOf(pfd), 0);
  with pfd do begin
    nSize     := sizeof(pfd);
    nVersion  := 1;
    dwFlags   := PFD_DRAW_TO_WINDOW or PFD_SUPPORT_OPENGL or
                 PFD_DOUBLEBUFFER;
    iPixelType:= PFD_TYPE_RGBA;
    cColorBits:= 24;
    cDepthBits:= 32;
    iLayerType:= PFD_MAIN_PLANE;
  end;
  nPixelFormat := ChoosePixelFormat(DC, @pfd);
  SetPixelFormat(DC, nPixelFormat, @pfd);
end;
end.
```

Список літератури

1. *Геометрическое* моделирование и машинная графика в САПР: Учебник / В.Е. Михайленко, В.Н. Кислоокий, А.А. Лященко и др. — К.: Выща шк., 1991. — 374 с.
2. *Демченко В.В., Анпілогова В.О.* Функціональна модель графічних застосувань на основі OpenGL // Сучасні проблеми геометричного моделювання: Зб. праць міжнар. наук.-практ. конф. — Харків, 2001. — С. 194-196.
3. *Баяковский Ю.М., Игнатенко А.В., Фролов А.И.* Графическая библиотека OpenGL: Учебно-методическое пособие. — М.: Изд-во Моск. ун-та, 2003. – 132 с.
4. *Гилой В.* Интерактивная машинная графика. — М.: Мир, 1981. — 384 с.
5. *Краснов М.В.* OpenGL. Графика в проектах Delphi. — СПб.: БХВ — Санкт-Петербург, 2000. — 352 с.
6. *Тихомиров Ю.* Программирование трехмерной графики. — СПб.: БХВ — Санкт-Петербург, 1999. — 256 с.
7. *Нікольський Ю.В., Пасічник В.В., Щербина Ю.М.* Дискретна математика. — К.: Видавнича група BHV, 2007. — 368 с.
8. *Шикин Е.В., Боресков А.В.* Компьютерная графика. Полигональные модели. — М.: ДИАЛОГ-МИФИ, 2001. — 464 с.

Алфавітний покажчик

M

О

П

Р

С

Т

Навчальне видання

Лященко Анатолій Антонович
Демченко Віктор Вікторович
Бородавка Євген Володимирович
Смирнов Володимир В'ячеславович

ГЕОМЕТРИЧНЕ МОДЕЛЮВАННЯ І КОМП'ЮТЕРНА ГРАФІКА: ВИКОРИСТАННЯ БІБЛІОТЕКИ OPENGL

Навчальний посібник

Редагування та коректура .....
Комп'ютерна верстка .....
Оформлення обкладинки *Є.В. Бородавки*

Підписано до друку 29.01.2003. Формат 60x84 1/16.
Папір офсетний. Гарнітура Аріал. Друк на різографі.
Ум. друк. арк. 0,93. Облік.-вид. арк. 1,0.
Ум. фарбовідб. 9. Тираж 50 прим. Вид. № 3/I. Зам. № 16/1-03.

КНУБА, Повітрофлотський проспект, 31, Київ, Україна, 03680

Віддруковано в редакційно-видавничому відділі
Київського національного університету будівництва і архітектури
Київ 2008