УДК 004.738.5

А. А. БУРЛАКОВ, П. Г. ГРОНСЬКИЙ
Хмельницький національний університет

ВИКОРИСТАННЯ СУЧАСНИХ ТЕХНОЛОГІЙ 3D-ПРОГРАМУВАННЯ ДЛЯ СТВОРЕННЯ ТРЬОХВИМІРНИХ ІГОР

Проводиться ґрунтовний аналіз сучасних технологій та інструментарію програмування при розробці трьохвимірних ігор. Показано використання бібліотеки XNA Framework — високопродуктивного кросплатформового API для розробки графічних додатків під Windows і XBOX 360.

There is carried out the fundamental analysis of modern technologies and tools of programming by developing the three-dimensional games. It has been shown the libraries XNA Framework use — high-productive cross-platform API for developing graphic applications under Windows and XBOX 360.

Ключові слова: інструментарій програмування, трьохвимірні ігри.

Одним з перспективних напрямків в індустрії розробки програмного забезпечення є створення трьохвимірних (3D) ігор. Яку б графічну бібліотеку або технологію на її базі не використовували при розробці 3D-моделей, в основі лежать фундаментальні знання з комп'ютерної графіки.

Для досягнення мети – створення 3D-ігор – найголовнішим завданням, що постає перед програмістом, є вибір оптимальної сучасної технології та інструментарію програмування. Ґрунтовний аналіз стану в цій сфері програмної інженерії дозволить ефективно реалізовувати поставлені задачі.

За останні роки комп'ютерна графіка отримала дуже широке поширення. Зараз 3D-вимірні зображення можна побачити скрізь, починаючи від простих комп'ютерних ігор і закінчуючи системами моделювання в реальному часі. Раніше, коли 3D-вимірна графіка існувала тільки на суперкомп'ютерах, не існувало єдиного стандарту в галузі графіки. Всі програми писалися з "нуля" або з використанням накопиченого досвіду, але в кожній програмі реалізовувалися свої методи для відображення графічної інформації. З приходом потужних процесорів і графічних прискорювачів тривимірна графіка стала реальністю для персональних комп'ютерів. Але в той же час виробники програмного забезпечення зіткнулися з серйозною проблемою – відсутністю будь-яких стандартів, які дозволяли писати програми, незалежні від обладнання та операційної системи.

Одним з перших таких стандартів, який існує і сьогодні, являється OpenGL. OpenGL – це графічний стандарт в області комп'ютерної графіки. На даний момент він є одним із найбільш популярних графічних стандартів у всьому світі. Розробники OpenGL – це найбільші фірми-розробники як устаткування, так і програмного забезпечення: Silicon Graphics, Inc., Microsoft, IBM Corporation, Sun Microsystems, Inc., Digital Equipment Corporation (DEC), Evans & Sutherland, Hewlett-Packard Corporation, Intel Corporation та Intergraph Corporation.

OpenGL перекладається як Відкрита Графічна Бібліотека (Open Graphics Library). Це означає, що OpenGL – це відкритий і мобільний стандарт. Програми, написані за допомогою OpenGL, можна переносити практично на будь-які платформи, отримуючи при цьому однаковий результат, будь це графічна станція або суперкомп'ютер. OpenGL звільняє програміста від написання програм для конкретного обладнання. Якщо пристрій підтримує якусь функцію, то ця функція виконується апаратно, якщо ні, то бібліотека виконує її програмно.

Випуск корпорацією Microsoft графічної бібліотеки DirectX поступово почав переорієнтовувати виробників ігор на Windows. Саме DirectX значно спростив роботу програмістів, взявши на себе керування мультимедійними апаратними засобами і маніпуляторами. Це дозволило створювати програми та ігри використовуючи стандартизовані набори команд і не підлаштовуватися під конкретні види відеокарт і звукових карт. З виходом нових версій DirectX істотно збільшилася продуктивність відеоадаптерів і кількість підтримуваних функцій.

Щоб вирішити, яка ж технологія краща, потрібно детально ознайомитись з їх перевагами та недоліками.

Що ж являє собою OpenGL? З точки зору програміста, OpenGL – це програмний інтерфейс для графічних пристроїв, таких як графічні прискорювачі. Він включає в себе близько 150 різних команд, за допомогою яких програміст може визначати різні об'єкти і виробляти рендеринг. Кажучи більш звичною мовою, ви визначаєте об'єкти, задаєте їх місце розташування у тривимірному просторі, визначаєте інші параметри (поворот, розтягування тощо), задаєте властивості об'єктів (колір, текстура, матеріал тощо), положення спостерігача, а бібліотека OpenGL подбає про те щоб відобразити все це на екрані.

OpenGL має добре продуману внутрішню структуру і досить простий процедурний інтерфейс. Незважаючи на це за допомогою OpenGL можна створювати складні і потужні програмні комплекси, витрачаючи при цьому мінімальний час у порівнянні з іншими графічними бібліотеками.

У деяких бібліотеках OpenGL (наприклад, під X Windows) є можливість зображувати результат не тільки на локальній машині, але також і по мережі. Додаток, який виробляє команди OpenGL, називається клієнтом, а програма, яка отримує ці команди і відображає результат – сервером.

Основні можливості бібліотеки наступні:

1. Підтримка геометричних і растрових примітивів. На основі геометричних і растрових примітивів будуються всі об'єкти. З геометричних примітивів бібліотека надає точки, лінії, полігони, з растрових – бітовий масив (bitmap) та образ (image).

2. Обслуговування видових та модельних перетворень. За допомогою цих перетворень можна розташовувати об'єкти в просторі, обертати їх, змінювати форму, а також змінювати положення камери, з якої ведеться спостереження.

3. Робота з кольором. OpenGL надає програмісту можливість роботи з кольором у режимі RGBA (червоний – зелений – синій – альфа) або використовувати індексний режим, де колір обирається з палітри.

4. Забезпечення подвійної буферизації. OpenGL надає як одинарну, так і подвійну буферизацію. Подвійна буферизація використовується для того, щоб усунути мерехтіння при мультиплікації, тобто зображення кожного кадру спочатку малюється у другому (невидимому) буфері, а потім, коли кадр повністю намальований, весь буфер відображається на екрані.

5. Наявність функцій з накладення текстури, згладжування, освітлення, атмосферних ефектів (туман, дим). Все це також дозволяє надати об'єктам або сцені реалістичності, а також "відчути" тривимірне зображення.

Хоча бібліотека OpenGL і вважається однією з кращих бібліотек як для професійного застосування, так і для ігор, у неї існують і конкуренти. Звичайно, це Direct 3D з пакету DirectX.

Direct3D з пакету DirectX розроблений фірмою Microsoft. Direct3D створювався виключно для ігрових додатків.

DirectX – це високопродуктивна мультимедійна бібліотека для програмування додатків вимогливих до продуктивності відеопідсистеми, аудіосистеми. В основі DirectX лежить набір COM2-інтерфейсів, що надають програмістові доступ до апаратного забезпечення комп'ютера. Ці інтерфейси розроблені Microsoft в тісній співпраці з провідними виробниками апаратних пристроїв, такими як Intel, ATI, NVIDIA, Creative тощо. Тому інтерфейси DirectX дуже близькі до сучасного апаратного забезпечення і фактично пов'язані з апаратним забезпеченням через тонкий прошарок драйверів, минаючи стандартні інтерфейси Win32, такі як GDI і MCI.

Починаючи з перших версій, DirectX у своєму складі мав ряд інструментів:

1) DirectDraw – інструмент для растрової графіки;
2) Direct3D – інструмент для 3D графіки;
3) DirectInput – інструмент для отримання даних з миші, клавіатури тощо;
4) DirectPlay – інструмент для обміну даними в мережі;
5) DirectSound – інструмент для роботи зі звуком;
6) DirectMusic – інструмент відтворення музики;
7) DirectShow – інструмент для введення-виведення аудіо- і відеоданих;
8) DirectSetup – інсталятор;
9) DirectX Media Objects – підтримка потокових обробників (кодери, декодери).

Якщо порівнювати OpenGl та Direct 3D у плані переносимості програм з однієї платформи на іншу, то Direct3D буде працювати тільки на Intel-платформах під управлінням операційної системи Windows, у той час програми, написані за допомогою OpenGL можна успішно перенести на такі платформи, як Unix, Linux, SunOS , IRIX, Windows, MacOS і багато інших. А от у плані об'єктно-орієнтованого підходу OpenGL поступається Direct3D. OpenGL працює за принципом кінцевого автомата, переходячи з одного стану в інший, здійснюючи при цьому якісь перетворення. Ще однією перевагою Direct3D є підтримка дешевого обладнання, OpenGL підтримується не на всіх графічних картах. OpenGL легша, ніж Direct3D для вивчення основ графіки; OpenGL можна застосовувати, наприклад, для початкового вивчення тривимірної графіки.

З виходом Windows Vista з'явилася чергова, революційна 10 версія DirectX, яка істотно поліпшила реалістичність формованих зображень, значно збільшила продуктивність відеоадаптерів і дозволила розвантажити центральний процесор через перенесення великої частини функцій на відеочипсет. Це стало можливим завдяки зміні процесу взаємодії системи та мультимедійних пристроїв. DirectX 10 недосяжний для інших ОС.

Саме відсутність підтримки DirectX 10 і не дозволить запускати нові ігри під Windows XP. Втім, до 9-ї версії можливості OpenGL і DirectX зрівнялися, після чого намітилася тенденція до технологічного відставання OpenGL від DirectX.

У 2005 р. в продаж надійшла ігрова приставка Microsoft наступного покоління – XBOX 360. На цей раз Microsoft вирішила відмовитися від використання процесорів звичної архітектури x86 на користь процесора PowerPC, не сумісного з x86. Сама архітектура ігрової приставки так само значно відрізнялася від персонального комп'ютера, що працює під управлінням операційної системи Windows. Таким чином, вперше в історії у Microsoft виявилися дві несумісних ігрових платформи: Windows і XBOX 360. Ця обставина відчутно ускладнила життя розробникам ігор, так як написання програми з підтримкою обох платформ фактично зводиться до написання окремих додатків для кожної платформи, що під силу лише досить великим конторам.

Подібна ізоляція двох платформ не влаштовувала Microsoft, тому виникла необхідність створення інструментарію, який дозволяє невеликим конторам і початківцям-розробникам створювати кросплатформенні програми, що працюють як на платформі Windows, так і на XBOX 360. В якості основи було вирішено використовувати платформу .NET: як відомо, .NET-додатки компілюються в проміжну мову IL, а фінальна компіляція в машинний код відбувається тільки під час запуску програми на конкретній системі.

Таким чином, .NET-додаткам взагалі байдуже, який процесор в даний момент встановлений в системі.

Але на практиці все виявилося дещо складніше:

1. .NET Framework містить потужну бібліотеку класів на всі випадки життя, включаючи розробку додатків баз даних, web-сервісів, web-сайтів. Зрозуміло, дана функціональність є, м'яко кажучи, дещо надмірною для ігрової приставки. Крім того, ряд класів .NET Framework сильно прив'язані до операційної системи Windows (простори імен System.Windows.Forms, System.Drawings), а перенесення Windows на ігрову приставку є вельми сумнівною ідеєю;

2. Managed DirectX є досить тонким налаштуванням над DirectX. А так як DirectX – це один з компонентів платформи Windows, Managed DirectX автоматично виявляється нестерпним API, прив'язаним до платформи Windows. Крім того, класи і структури Managed DirectX активно використовують функціональність з просторів імен System.Windows.Forms і System.Drawing.

Перша проблема була вирішена шляхом реалізації на XBOX лише підмножиною класів .NET Framework, відомих як .NET Compact Framework. Для вирішення другої проблеми був розроблений XNA Framework – високопродуктивний багатоплатформовий API для розробки графічних додатків для платформ Windows і XBOX 360.

Умовно всі компоненти XNA Framework можна розділити на 4 рівні абстракції (рис. 1):

1. Platform (Платформа) – найнижчий рівень, що містить платформозалежні API, такі як некерований DirectX. У переважній більшості випадків програма може нічого не знати про існування цього рівня, використовуючи компоненти більш високих рівнів абстракції пристроїв.

2. Core Framework (Основний Каркас) – нижній платформонезалежний рівень XNA, що забезпечує базову функціональність. Розміщується в збірці Microsoft.Xna.Framework.dll і містить 5 компонентів: Graphics (робота з графікою), Audio (робота зі звуком), Input (робота з пристроями введення-виведення), Math (математичні розрахунки), Storage (робота з файловою системою).

3. Класи і структури кожного компонента згруповані в простори імен і наведені в табл. 1. На платформі Windows перші три компонента (Graphics, Audio, Input) є надбудовами над DirectX, а компонент Storage – надбудовою над класами .NET Framework для роботи з файловою системою.

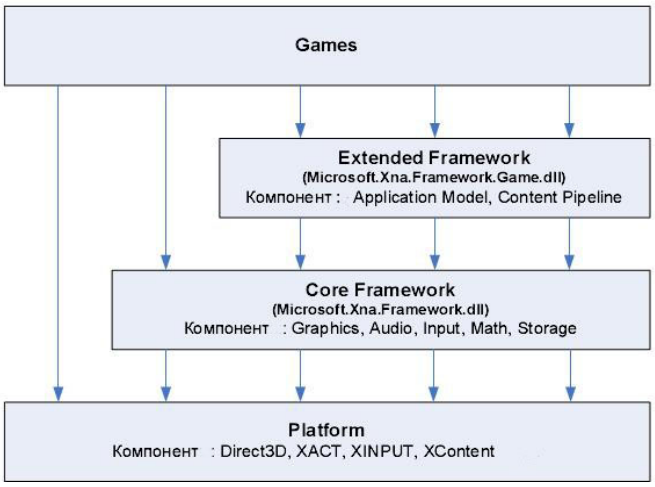

Рис. 1. Рівні XNA Framework

4. Extended Framework (розширений каркас) – набір високорівневих класів, які вирішують типові завдання, що постають перед розробником ігор: ініціалізація графічного пристрою, організація циклу обробки повідомлень, експорт моделей і текстур з графічних редакторів. По суті Extended Framework можна вважати універсальним ігровим движком (Game Engine) початкового рівня. Розміщується в збірці Microsoft.Xna.Framework.Game.dll.

5. Game – власне додаток користувача, тобто наші з вами програми. До слова, в комплект XNA входить кілька простих ігор (Starter Kits), які можна використовувати в якості заготовок для своїх додатків.

Таблиця 1

Згруповані в простори імен класи і структури кожного компонента

Простори імен	Відповідний компонент XNA	Призначення
Microsoft.Xna.Framework	Math	Математичні розрахунки: матрична алгебра, аналітична геометрія, перевірка зіткнень і т.д. XNA Framework виконує математичні розрахунки власними засобами, що в деяких випадках дещо підвищує продуктивність завдяки відсутності накладних витрат взаємодії з COM
Microsoft.Xna.Framework.Graphics		Робота з графікою

Microsoft.Xna.Framework.Graphics.PackedVector	Graphics	Робота з упакованими векторами. Прикладом упакованого вектора є 32-бітне число, що містить інформацію про яскравість червоного, синього і зеленого кольору
Microsoft.Xna.Framework.Audio	Audio	Робота із звуком
Microsoft.Xna.Framework.Input	Input	Робота з пристроями введення (клавіатура, миша, джойстики)
Microsoft.Xna.Framework.Storage	Storage	Робота з файловою системою поточної платформи: завантаження і збереження настройок застосування, "збережених ігор"

Завдяки XNA Framework, що дає нам багато допоміжних класів і простих способів для керування вікном гри, ігровими компонентами й файлами (моделями, текстурами, шейдерами і звуковими файлами), вам не треба турбуватися про власні формати файлів, написання власного класу управління вікном або про те, як завантажити модель у ваш «движок», а потім отримати її на екрані. Якщо ви раніше працювали з DirectX чи OpenGL, то можливо стикалися з тим, що приклади та уроки не здаються важкими, але як тільки ви захочете використовувати в «движку» власні текстури, тривимірні моделі або звукові файли, то зіштовхуєтеся з такими проблемами, як непідтримувані формати файлів, відсутність підтримки тривимірних моделей для вашого редактора або неможливість відтворення ваших звукових файлів. Потім ви або продовжуєте пошук, або пишете власні класи для підтримки цих можливостей. Класи для текстур і моделей існують, але це не означає, що вони досконалі. Іноді вам потрібна додаткова функціональність або більш простий спосіб завантаження і доступу до текстур і моделей. Тому ви пишете нові класи для управління текстурами і моделями, які всередині, як і раніше, використовують класи XNA, але роблять їх простіше для роботи з текстурами і моделями, а в подальшому – і з матеріалами і шейдерами також.

В розробці 2D-вимірних ігор використовується двовимірна система координат. Головна відмінність цієї системи від традиційної полягає в тому, що початок координат розташовано в лівому верхньому кутку екрану. Координати по осі X зростають зліва направо, а вісь Y виявляється перевернутою – її координати зростають зверху вниз.

При роботі з тривимірною графікою використовується декілька видів систем координат. Нам знадобиться ще одна вісь – вісь Z. Існує кілька варіантів тривимірних систем координат, зокрема, поширені так звані правостороння і лівостороння системи. Ми будемо користуватися правобічної системою – вона застосовується в XNA Framework. Її схематичне зображення наведено на рис. 2.

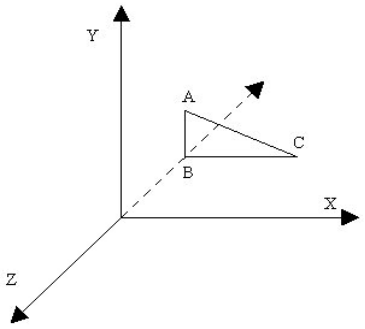

Рис. 2

Особливість цієї системи координат полягає в тому, що початок координат можна зіставити з лівим нижнім кутом монітора, позитивна частина осі X знаходиться праворуч від початку координат, позитивна частина осі Y – зверху, а позитивна частина осі Z – спереду. А це означає, що видима частина осі Z – це її негативна частина. Ця частина осі знаходиться якби "в глибині монітора" в той час, як позитивна частина знаходиться "попереду монітора".

У двовимірній системі координат існує поняття точки – її координати задаються двома значеннями: X й Y. Точки існують і в тривимірній системі координат – вони задаються вже трьома значеннями: X, Y, Z.

Знаючи координати вершин полігонів, з яких складається об’єкт, ми можемо розташувати його в просторі. Тепер потрібно розібратися зі зміною положення об’єктів у просторі. Існує кілька основних операцій, які можуть використовуватися для переміщення об’єктів у тривимірному просторі. Це – переміщення (translation), обертання (rotation) і масштабування (scale).

Результати роботи графічної підсистеми тривимірної гри ми бачимо на плоскому екрані монітора – змодельована комп’ютером тривимірна сцена проектується на двовимірну поверхню. При проектуванні потрібно обрати точку, яка виконує роль камери, що дозволяє бачити тривимірний простір. У свою чергу, об’єкти в тривимірному просторі можуть пересуватися відповідно до певних правил. Для управління всім цим використовуються кілька матриць. Це – світова матриця (World Matrix), матриця виду (View Matrix) і матриця проекції (Projection Matrix).

Матрицю можна представити у вигляді таблиці, що складається з m рядків і n стовпців. У комп’ютерній графіці застосовуються матриці 4×4. Перші три стовпці цієї матриці відповідають за

модифікацію координат X, Y, Z вершин об'єкта, який бере участь у трансформації.

Світова матриця дозволяє задавати перетворення – переміщення, обертання і трансформації об'єктів. Матриця виду дозволяє керувати камерою. Матриця проекції служить для налаштування проекції тривимірної сцени на екран.

Розглянемо на простому прикладі завантаження та вивід простої 3D-вимірної моделі.

Для виведення моделі на екран необхідно додати в проект відповідні їй ресурси, в нашому випадку – модель у форматі .FBX і текстура у форматі .JPG. Далі слід завантажити модель в гру, налаштувати її властивості і вивести на екран.

У тривимірній графіці існує таке поняття, як Mesh (сітка, іноді використовують транслітерацію з цього слова – меш). Тривимірні моделі складаються з сіток – ліній, що з'єднують вершини. Модель може складатися з декількох взаємодіючих сіток, кожну з яких потрібно обробити при виведенні моделі.

Створимо новий проект для демонстрації техніки виведення тривимірної моделі. На рис. 3 можна побачити вікно Project Explorer цього проекту і вікно Properties для текстури, призначеної для накладення на модель.

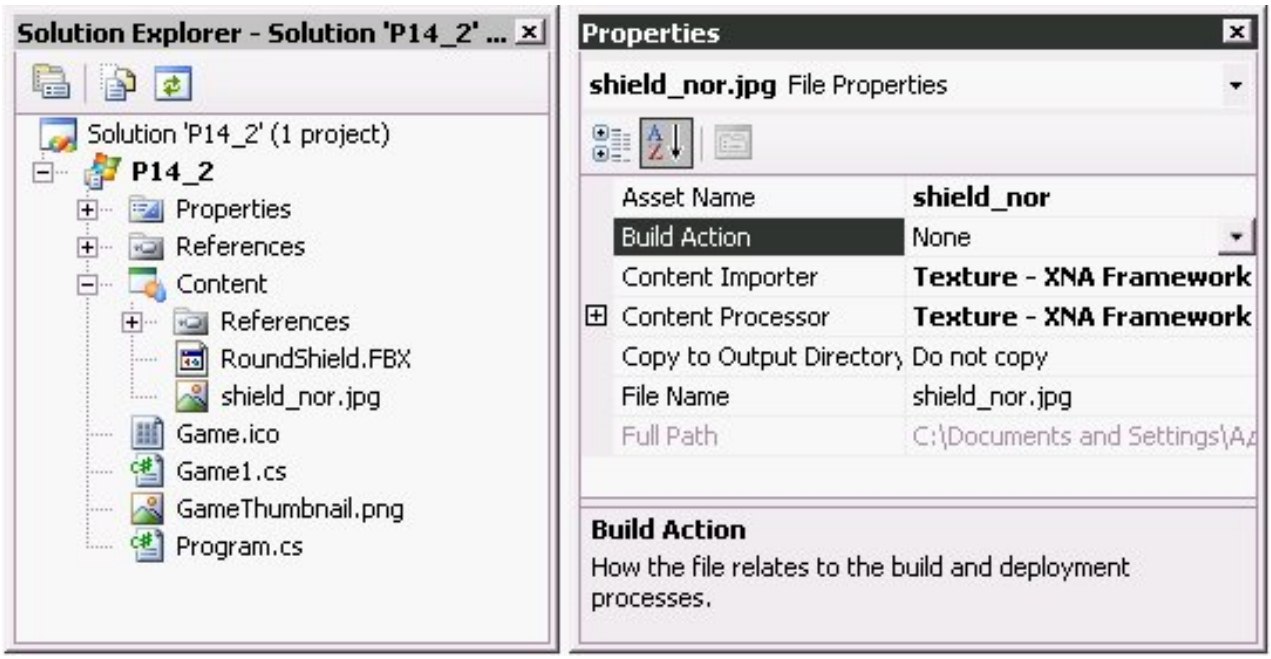

Рис. 3. Вікна Solution Explorer і Properties

Тривимірні моделі, які передбачають використання текстур, містять у своїх файлах вказівку на те, яким чином накладати текстуру на об'єкт і так само – на файл текстури. У нашому випадку адреса файлу текстури задана у відносному вираженні, тобто система буде шукати цей файл в тій же папці, в якій розташований файл тривимірної моделі. Текстура піддається перетворенням у відповідності з командами, записаними у файлі тривимірної моделі, тому ми відключили параметр Build Action у властивостях текстури.

Розглянемо детально код, який ми написали у Visual Studio 2010:

```
using System;
using System.Collections.Generic;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Audio;
using Microsoft.Xna.Framework.Content;
using Microsoft.Xna.Framework.GamerServices;
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Input;
using Microsoft.Xna.Framework.Net;
using Microsoft.Xna.Framework.Storage;

namespace 3d_model
{
   public class Game1 : Microsoft.Xna.Framework.Game
   {
      GraphicsDeviceManager graphics;
      // Модель для виводу
      Model myModel;
      //Світова матрица
      Matrix WorldMatrix;
      //Відповідність сторін вікна виводу
      float aspectRatio;
      //Кут повороту для обертання моделі
      float angle;
      public Game1()
      {
         graphics = new GraphicsDeviceManager(this);
         Content.RootDirectory = "Content";
      }
      protected override void Initialize()
      {
         // TODO: Add your initialization logic here
         base.Initialize();
      }
      protected override void LoadContent()
      {
         //завантаження моделі
         myModel = Content.Load<Model>("RoundShield");
         //Установка співвідношення сторін вікна виводу
         aspectRatio = (float)graphics.GraphicsDevice.Viewport.Width /
         (float)graphics.GraphicsDevice.Viewport.Height;
      }
      protected override void UnloadContent()
      {
         // TODO: Unload any non ContentManager content here
      }

      protected override void Update(GameTime gameTime)
      {
```

```
        //збільшимо кут повороту на 1
        angle++;
        //встановимо новую світову матрицу
        //поворот по осі Y рівний куту angle, поворот по X -
        //angle, поділеному на 5
        WorldMatrix = Matrix.CreateRotationY(MathHelper.ToRadians(angle))*
           Matrix .CreateRotationX (MathHelper .ToRadians (angle /5.0f));
        //Для кожної сітки в моделі
        foreach (ModelMesh mesh in myModel.Meshes)
        {
           //Для кожного ефекту в сітці
           foreach (BasicEffect effect in mesh.Effects)
           {
              //Встановити освітлення за замовчуванням
              effect.LightingEnabled = true;
              effect.EnableDefaultLighting();
             //Встановити матрицю
              effect.World = WorldMatrix;
              effect.View = Matrix.CreateLookAt(new Vector3(0.0f, 0.0f, 119.0f), Vector3.Zero, Vector3.Up);
              effect.Projection = Matrix.CreatePerspectiveFieldOfView(MathHelper.ToRadians(45.0f),
                 aspectRatio, 1.0f, 1000.0f);
           }
        }
        base.Update(gameTime);
     }
     protected override void Draw(GameTime gameTime)
     {
        graphics.GraphicsDevice.Clear(Color.CornflowerBlue);
        // Вивід моделі
        foreach (ModelMesh mesh in myModel.Meshes)
        {
           mesh.Draw();
        }
        base.Draw(gameTime);
     }
  }
}
```

Після цього ми можемо побачити результат роботи шейдера в XNA (рис. 4).

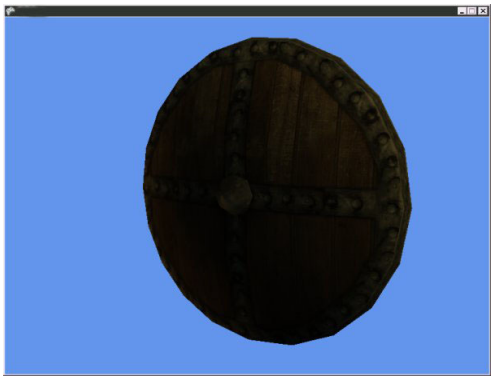

Рис. 4

XNA Framework підтримує шейдери на High Level Shader Language (HLSL). Опис шейдерів зберігається в .Fx-файлі. Зараз ми розглянемо структуру шейдерної програми і наведемо приклад найпростішого шейдера на HLSL. Також буде описано те, як працювати з шейдерами в XNA Game Studio.

Шейдерна програма схожа на звичайний файл з текстами програм на будь-якій С-подібній мові. Така програма компілюється в машинний код для відеокарти і потім вбудовується в графічний конвеєр на етапах обробки вершин і пікселів. Для того, щоб почати створювати власний шейдер, необхідно додати в папку Content новий файл з ефектом (.Fx-файл) – рис. 5.

У вікні, що з'явилося, необхідно обрати Effect File і ввести ім'я файлу (рис. 6).

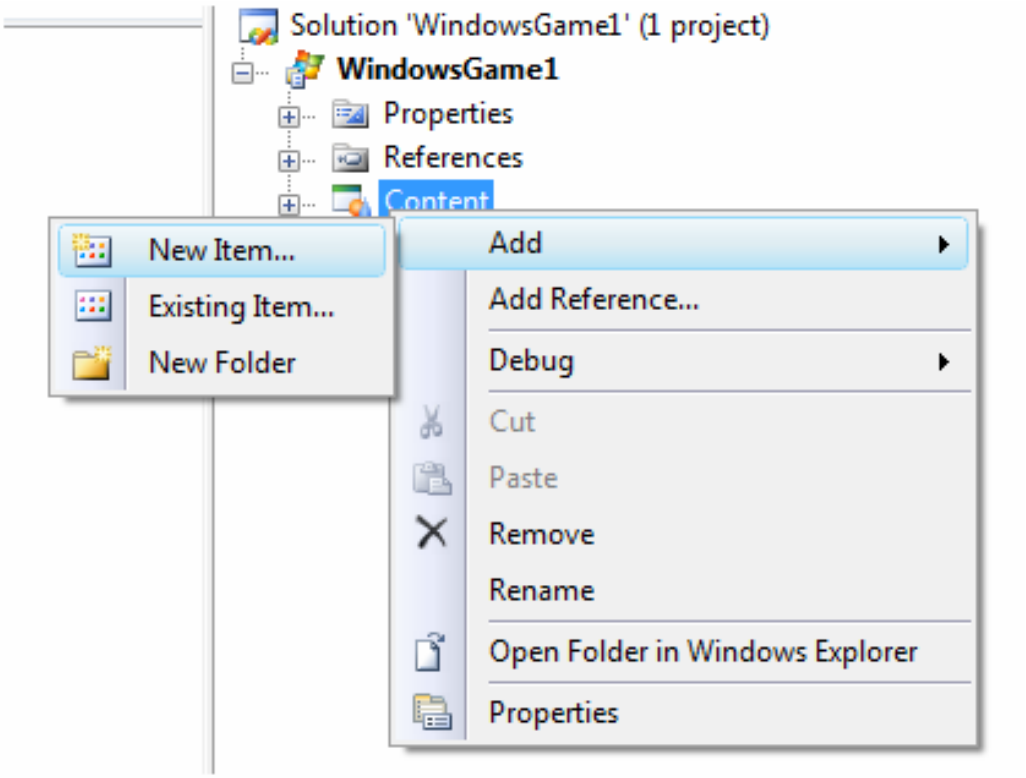

Рис. 5

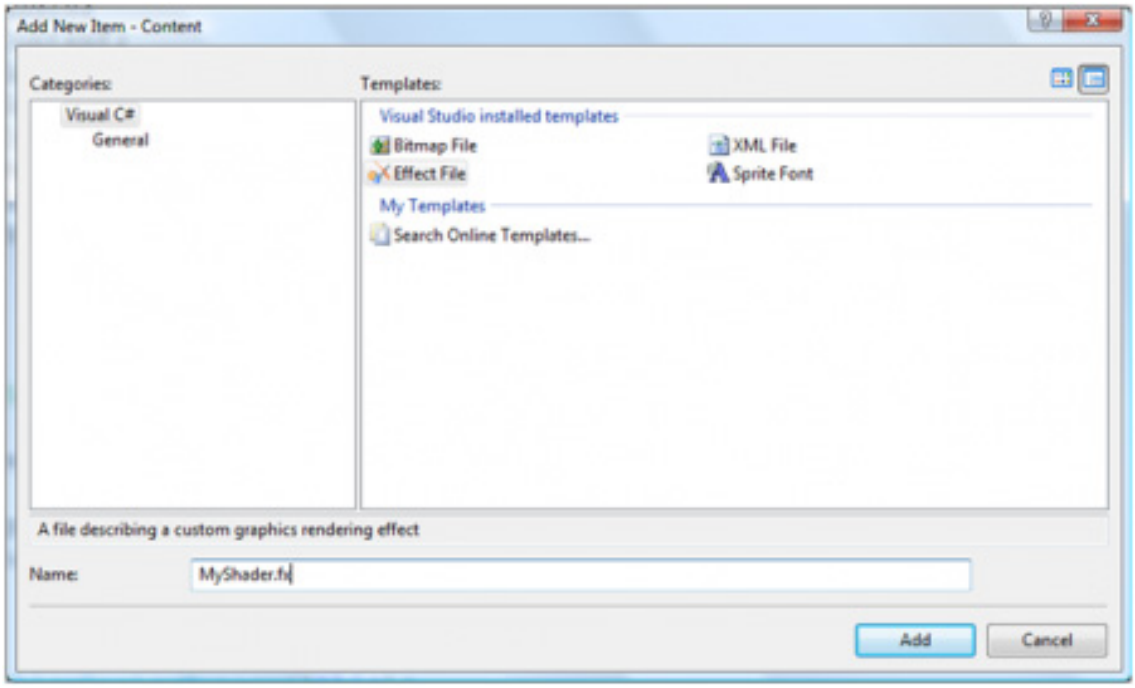

Рис. 6

XNA Game Studio створить не порожній файл, а файл з уже написаним найпростішим шейдером. Розглянемо вихідний код даного шейдера:

```
float4x4 World;
float4x4 View;
float4x4 Projection;
// TODO: add effect parameters here.
struct VertexShaderInput
{
float4 Position : POSITION0
// TODO: add input channels such as texture
// coordinates and vertex colors here
}
struct VertexShaderOutput
{
float4 Position : POSITION0;
// TODO: add vertex shader outputs such as colors and texture
// coordinates here. These values will automatically be interpolated
// over the triangle, and provided as input to your pixel shader
}
VertexShaderOutput VertexShaderFunction(VertexShaderInput input)
{
VertexShaderOutput output;
float4 worldPosition = mul(input.Position, World);
float4 viewPosition = mul(worldPosition, View);
output.Position = mul(viewPosition, Projection);
// TODO: add your vertex shader code here.
return output;
}
float4 PixelShaderFunction(VertexShaderOutput input) : COLOR0
{
// TODO: add your pixel shader code here.
return
float4(1, 0, 0, 1);
}
technique Technique1
{
pass Pass1
{
// TODO: set renderstates here.
VertexShader = compile
vs_1_1 VertexShaderFunction();
PixelShader = compile
ps_1_1 PixelShaderFunction();
}
}
```

Шейдер має три змінні: матриці World, View, Projection. Вони не мають значення за замовчуванням і повинні бути заповнені в програмі на XNA Framework, тобто у вашій майбутній грі.

Не складно здогадатися, що ці змінні відповідають за Світову матрицю, матрицю Виду і матрицю Проекції.

Далі йде опис двох структур: VertexShaderInput і VertexShaderOutput. Ці структури будуть

використовуватися в якості вхідних і вихідних параметрів вершинного шейдера. Зверніть увагу на те, що структури складаються тільки з одного компонента – позиції вершини.

Далі йде опис вершинного шейдера, розглянемо його більш докладно:

```
VertexShaderOutput VertexShaderFunction(VertexShaderInput input){

VertexShaderOutput output;

float4 worldPosition = mul(input.Position, World);

float4 viewPosition = mul(worldPosition, View);

output.Position = mul(viewPosition, Projection);

// TODO: add your vertex shader code here.

return output;}
```

Спочатку створюється змінна типу VertexShaderOutput, яка буде зберігати значення, що повертається. Зверніть увагу на те, що необхідно заповнити всі елементи, які існують в структурі вихідного значення, в іншому випадку компілятор поверне повідомлення про помилку.

Далі створюється worldPosition, яка буде зберігати значення світових координат. Отримати це значення можна, коли домножимо локальні координати вершини, передані в якості елемента вхідної структури, на Світову матрицю. Це можна зробити за допомогою функції mul, яка перемножує матриці і повертає результат перемноження.

Далі аналогічним способом обчислюються значення координат виду і проекції і останні зберігаються у вихідній структурі. Виклик “return output;” повертає з функції значення змінної output.

Тепер розглянемо уважніше піксельний шейдер:

```
float4 PixelShaderFunction(VertexShaderOutput input) : COLOR0 {

// TODO: add your pixel shader code here.

Return float4(1, 0, 0, 1); }
```

Функція піксельного шейдера в даному випадку повертає тільки колір пікселя і для неї вказана семантика COLOR0, яка вказує, що значення є саме кольором. Зверніть увагу на те, що в якості вхідного параметра піксельний шейдер приймає значення типу VertexShaderOutput, в якому містяться вже інтерпольовані значення параметрів, передані з вершинного шейдера.

Даний піксельний шейдер жодним чином не використовує вхідні параметри і зафарбовує кожен піксель об’єкта червоним непрозорим кольором (четверта компонента кольору відповідає за прозорість, вона також має назву альфа-канал).

Також у файлі є опис єдиної техніки, названої Technique1, яка має один прохід і вказує, які функції слід використовувати в якості вершинного і піксельного шейдера.

Тепер розглянемо, що може відтворити XNA з допомогою рейдерів: На рис. 7 зображена вода, яка створена з допомогою шейдерів.

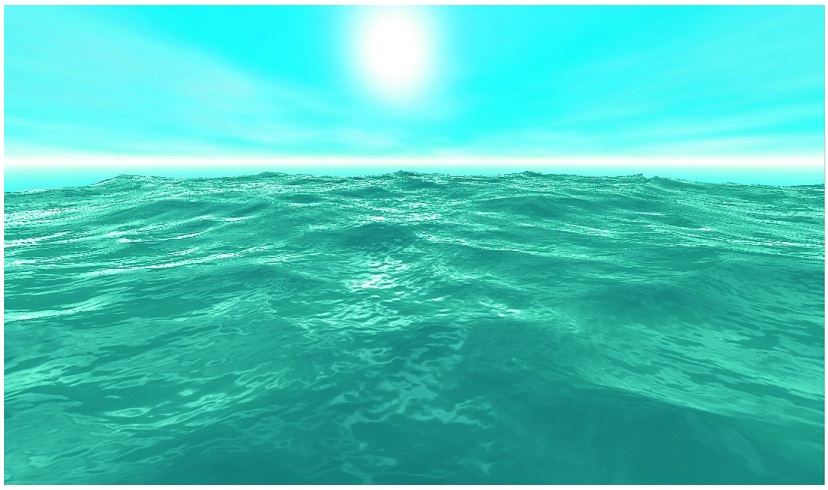

Рис. 7

В результаті проведеного дослідження було встановлено, що високорівнева графічна бібліотека XNA Framework від Microsoft є в наш час найбільш ефективним інструментом для розробки трьохвимірних ігор.

Список використаних джерел

1. Основи роботи з XNA [Електронний ресурс]. – Режим доступу : http://www.netlib.narod.ru/library/book0052/toc.htm. – Дата звернення : 20.11.2010.

2. Сайт спільноти розробників [Електронний ресурс]. – Режим доступу : http://www.xnadev.ru. – Дата звернення : 15.10.2010.

3. Офіційний сайт XNA розробників [Електронний ресурс]. – Режим доступу : www.creators.xna.com. – Дата звернення : 10.10.2010.

4. Форум XNA розробників [Електронний ресурс]. – Режим доступу : http://www.cyberforum.ru/graphics/thread7014.html. – Дата звернення : 15.09.2010.

5. Курс в Інтернет університеті [Електронний ресурс]. – Режим доступу : http://www.intuit.ru/department/se/intxna/. – Дата звернення : 15.09.2010.