Введение

Дисциплина «Архитектура графических устройств» входит в цикл направления специальности 1 40 05 01 12 «Информационные системы и технологии (в игровой индустрии)» компонента учреждения высшего образования.

Целью дисциплины является овладение теоретическими знаниями и практическими навыками по архитектуре графических процессоров, технологиям вычислений на графических процессорах и применения для этого различных библиотек.

Основными задачами дисциплины являются:

- изучение архитектуры графических процессоров;
- изучение технологий вычисления на графических процессорах;
- изучение современных библиотек для организации вычисления на графических процессорах.

1. Лабораторная работа № 1. Работа с цветовыми пространствами

Цель работы: познакомиться с представлением цвета в различных цветовых простанствах.

Для представления цвета в компьютерной графике используются разные цветовые пространства.

Для описания цвета пикселя на экране монитора наиболее удобно просранство RGB (red, green, blue – красный, зеленый, синий). Оно является аддитивным. Смысл этого в том, что для получения результирующего цвета, составляющие складываются. Например, для получения желтого цвета смешивается красная и зеленая составляющие, для фиолетового – красная и синяя. Выбор базовых цветов основан на физиологии человеческого глаза. Компоненты цвета обычно представляются однобайтовой целой переменной в интервале от 0 до 255 (8 бит на канал, 24 бита на пиксель) или числом с плавающей точкой в диапазоне от 0 до 1. Каждый пиксель монитора имеет 3 субпикселя, соответствующие 3-ем составляющим. Степень свечения субпикселя зависит от значения составляющей. 0 – отсутствует, 255 – имеет максимальную яркость. Таким образом (0, 0, 0) соответствует черному цвету, (255, 255, 255) – белому.

Для описания цвета пикселя при печати наиболее удобно пространство CMY или CMYK (cyan, magenta, yellow, key (black) – бирюзовый, пурпурный, желтый, черный). Оно является субстрактивным. Смысл этого в том, что для получения результирующего цвета базовые компоненты цвета вычитаются. Бирюзовый поглащает красный компонент и отражает зеленый и синий. Фиолетовый поглощает зеленый и отражает

красный и синий. Таким образом для получения синего цвета на бумаге нужно смешать составляющие, поглощающие красный и зеленый, то есть бирюзовый и фиолетовый. В графических редакторах трехкомпонентное пространство также кодируется числами в диапазоне от 0 до 255, четырехкомпонентное – числами в дипазоне от 0 до 100. Большее значение соответствует большему заполнению бумаги краской или тонером. Таким образом (0, 0, 0) – соответствует белому цвету, (255, 255, 255) – черному.

Если не учитывать цветовые профили оборудования (субпиксели монитора не идеально передают составляющие цвета; пигменты, используемые для производства чернил и тонера, также не идеальны, поэтому для цветокоррекции используются цветовые профили), то переход между RGB и CMY можно осуществить по следующим выражениям

$$\begin{cases} C = 255 - R, \\ M = 255 - G, \\ Y = 255 - B. \end{cases}$$

$$\begin{cases} R = 255 - C, \\ G = 255 - M, \\ B = 255 - Y. \end{cases}$$

Поскольку пространство RGB является трехкомпонентным, то его адекватно можно представить только в 3D пространстве. В целях унификации, а также чтобы уместить все оттенки цветов, видимых человеком на плоскости X0Y с координатами в пределах от 0 до 1, было разработано пространство XYZ.

Преобразование из RGB в XYZ выполняется по выражению

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = \frac{1}{0{,}17697} \cdot \begin{bmatrix} 0{,}49 & 0{,}31 & 0{,}2 \\ 0{,}17697 & 0{,}8124 & 0{,}01063 \\ 0 & 0{,}01 & 0{,}99 \end{bmatrix} \cdot \begin{bmatrix} R \\ G \\ B \end{bmatrix}.$$

Для человека привычнее оперировать другими понятиями: оттенок, насыщенность, яркость (светлость) цвета. Поэтому были разработаны цветовые пространства HSB (hue, saturation, brightness), HSI (hue, saturation, intencity), HSV (hue, saturation, value). Эти три пространства очень похожи. Hue – оттенок. Отсчитывается по углу: $0°$ – красный, $60°$ – желтый, $120°$ – зеленый, и так далее через весь спектр цветов до $300°$ – фиолетовый, и обратно к $360°$ – красному. Saturation – насыщенность, характеризует насколько цвет отличается от оттенков серого. Третий компонент (brightness, intencity, value) характеризует яркость или «светлоту», т.е. насколько цвет отличается от черного. Визуально эти цветовые пространства можно представить с помощью цилиндра или конуса. При представлении в виде конуса, повернутого основанием вниз на вершине, внизу находится черный цвет. На высоте от вершины к середине основания находятся градации серого цвета от черного до белого. На основании конуса

находится весь спектр по окружности с изменением от белого в центре, до насыщенных «чистых» цветов по краям.

Преобразовать цвет из RGB в HSI можно по следующим выражениям

$$\begin{cases} H = \arccos\left(\dfrac{\frac{1}{2}\cdot\big((R-G)+(R-B)\big)}{\sqrt{(R-G)^2+(R-B)\cdot(G-B)}}\right), \\ S = 1-\dfrac{3}{R+G+B}\cdot\min(R,G,B), \\ I = \dfrac{1}{3}\cdot(R+G+B). \end{cases}$$

Также используется в графическом программном обеспечении цветовое пространство $L^*a^*b^*$. Оно было разработано для устранения нелинейности системы XYZ с человеческой точки зрения. Координата L^* задает светлоту цвета. Координаты a^* и b^* – цветовой тон: a^* – расстояние от зеленого до красного, b^* – расстояние цвета от синего до желтого.

Конвертация из XYZ в $L^*a^*b^*$ производится по выражениям

$$\begin{cases} L^* = 116\cdot f\left(\dfrac{Y}{Y_n}\right)-16, \\ a^* = 500\cdot\left(f\left(\dfrac{X}{X_n}\right)-f\left(\dfrac{Y}{Y_n}\right)\right), \\ b^* = 200\cdot\left(f\left(\dfrac{Y}{Y_n}\right)-f\left(\dfrac{Z}{Z_n}\right)\right), \end{cases}$$

где:

- $f(t) = \begin{cases} \sqrt[3]{t}, \text{при } t > \delta^3, \\ \frac{t}{3\cdot\delta^2}+\frac{4}{29}, \text{при } t \le \delta^3, \end{cases}$
- $\delta = \frac{6}{29}$,
- X_n, Y_n, Z_n – значения координат «точки белого» в пространстве XYZ (для таблицы светимости D65 Международной комиссии по освещению $X_n = 95{,}0489$, $Y_n = 100$, $Z_n = 108{,}884$).

Обратная конвертация производится по выражениям

$$\begin{cases} X = X_n \cdot f^{-1}\left(\dfrac{L^* + 16}{116} + \dfrac{a^*}{500}\right), \\ Y = Y_n \cdot f^{-1}\left(\dfrac{L^* + 16}{116}\right), \\ Z = Z_n \cdot f^{-1}\left(\dfrac{L^* + 16}{116} - \dfrac{b^*}{200}\right), \end{cases}$$

где $f^{-1}(t) = \begin{cases} t^3, \text{при } t > \delta, \\ 3 \cdot \delta^2 \cdot \left(t - \frac{4}{29}\right), \text{при } t \le \delta. \end{cases}$

1.1. Задания на лабораторую работу

Разработать приложение (Windows Forms или WPF), реализующую функцию, указанную в табл. 1.1.

Таблица 1.1.

Задания на лабораторную работу № 1.

№ вар-та	Функции программы
1	Инструмент «Пипетка» с представлением цвета в пространствах RGB, HSI.
2	Инструмент «Пипетка» с представлением цвета в пространствах RGB, Lab.
3	Инструмент «Пипетка» с представлением цвета в пространствах RGB, XYZ.
4	Инструмент «Пипетка» с представлением цвета в пространствах RGB, CMY.
5	Преобразование исходного изображения в монохромное.
6	Создание изображения с прямоугольником, заполненным цветом, заданным пользователем в цветовом пространстве HSI.
7	Создание изображения с прямоугольником, заполненным цветом, заданным пользователем в цветовом пространстве Lab.
8	Создание изображения с прямоугольником, заполненным цветом, заданным пользователем в цветовом пространстве XYZ.
9	Создание изображения с прямоугольником, заполненным цветом, заданным пользователем в цветовом пространстве CMY.
10	Создание изображения с эллипсом, заполненным цветом, заданным пользователем в цветовом пространстве HSI.
11	Создание изображения с эллипсом, заполненным цветом, заданным пользователем в цветовом пространстве Lab.
12	Создание изображения с эллипсом, заполненным цветом, заданным пользователем в цветовом пространстве XYZ.

Продолжение таблицы 1.1.

13	Создание изображения с эллипсом, заполненным цветом, заданным пользователем в цветовом пространстве CMY.
14	Создание изображения, в котором каждый пиксель закрашен цветом R-составляющая которого пропорциональна координате пикселя по горизонтали, G – по вертикали. B = 0.
15	Создание изображения, в котором каждый пиксель закрашен цветом R-составляющая которого пропорциональна координате пикселя по горизонтали, B – по вертикали. G = 0.
16	Создание изображения, в котором каждый пиксель закрашен цветом G-составляющая которого пропорциональна координате пикселя по горизонтали, B – по вертикали. R = 0.
17	Создание изображения, в котором каждый пиксель закрашен цветом C-составляющая которого пропорциональна координате пикселя по горизонтали, M – по вертикали. Y = 0.
18	Создание изображения, в котором каждый пиксель закрашен цветом C-составляющая которого пропорциональна координате пикселя по горизонтали, Y – по вертикали. M = 0.
19	Создание изображения, в котором каждый пиксель закрашен цветом M-составляющая которого пропорциональна координате пикселя по горизонтали, Y – по вертикали. C = 0.
20	Создание изображения, в котором каждый пиксель закрашен цветом M-составляющая которого пропорциональна координате пикселя по горизонтали, Y – по вертикали. C = 1.
21	Создание изображения, в котором каждый пиксель закрашен цветом a-составляющая которого пропорциональна координате пикселя по горизонтали, b – по вертикали. L = 0,5.
22	Создание изображения, в котором каждый пиксель закрашен цветом H-составляющая которого пропорциональна координате пикселя по горизонтали, I – по вертикали. S = 1.
23	Создание изображения, в котором каждый пиксель закрашен цветом H-составляющая которого пропорциональна координате пикселя по горизонтали, S – по вертикали. I = 1.
24	Создание изображения, в котором каждый пиксель закрашен цветом S-составляющая которого пропорциональна координате пикселя по горизонтали, I – по вертикали. H задается пользователем.
25	Создание изображения с проекцией цветового RGB куба (шестиугольник), вид сверху.
26	Создание изображения с проекцией цветового RGB куба (шестиугольник), вид снизу.

Окончание таблицы 1.1.

27	Создание изображения с проекцией цветового конуса HSI (окружность), вид сверху.
28	Создание изображения с проекцией цветового конуса HSI (окружность), вид снизу.
29	Создание изображения с rg-хроматичностью.
30	Создание изображения с rg-хроматичностью в пространстве XYZ.

Контрольные вопросы:

- В чем разница между аддитивными и субстрактивными цветовыми пространствами?
- Как будет выглядеть треугольнк, если разрезать RGB куб по диаганалям через точки с координатами (1, 0, 0), (0, 1, 0), (0, 0, 1)?
- Как будет выглядеть RGB куб при взгляде со стороны вершины с белым цветом?
- Как будет выглядеть RGB куб при взгляде со стороны вершины с черным цветом?
- Что дает цветовое пространство XYZ?
- Что дает использование цветового пространства HSI?

2. Лабораторная работа № 2. Визуализация простейшей сцены с трехмерным объектом средствами библиотеки DirectX

Цель работы: ознакомиться с библиотекой DirectX, получить основные понятия о стадиях графического конвейера, получить практический опыт разработки простейшего приложения с использованием библиотеки DirectX.

Графический конвейер DirectX 11 имеет 9 стадий:

1. Входной сборщик (Input Assembler).
2. Вершинный шейдер (Vertex Shader).
3. Шейдер поверхности (Hull Shader).
4. Тесселятор (Tesselator).
5. Шейдер областей (Domain Shader).
6. Геометрический шейдер (Geometry Shader).
7. Растеризатор (Rasterizer).
8. Пиксельный шейдер (Pixel Shader).
9. Выходной сборщик (Output Merger).

Часть стадий являются программируемыми, часть только настраиваемы. Для создания простейшего приложения достаточно использования следующих стадий:

– входной сборщик для задания формата данных, поступающих на вход конвейера;
– вершинный шейдер для преобразования координат вершин объектов из пространства модели в пространство камеры и преобразования проецирования;
– растеризатор для получения областей кадра, соответствующих графическим примитивам;
– пиксельный шейдер для расчета цвета пикселей;
– выходной сборщик для получения результирующего изображения.

Будем использовать C# и объектно-ориентированную обертку DirectX-a SharpDX. В Microsoft Visual Studio создаем новый проект типа «Приложение Windows Forms (.NET Framework)». Версию платформы .NET можно выбрать 4.0 или 4.5.

В диспетчере пакетов NuGet необходимо добавить в проект следующие:
– SharpDX
– SharpDX.DXGI
– SharpDX.D3DCompiler
– SharpDX.Direct3D11
– SharpDX.Mathematics
– SharpDX.Desktop
– SharpDX.Diagnostics

Далее добавляем в проект манифест приложения. В контекстном меню проекта в обозревателе решения выбираем пункт «Добавить» → «Создать элемент...». В диалоговом окне приложения выбираем «Файл манифеста приложения». В манифесте нужно расскоментировать следующие строки.

`app.manifest`:

```
...


  <application xmlns="urn:schemas-microsoft-com:asm.v3">
    <windowsSettings>
      <dpiAware xmlns="http://schemas.microsoft.com/SMI/2005/WindowsSettings">true
</dpiAware>
    </windowsSettings>
  </application>
```

Удаляем форму `Form1` из приложения через контекстное меню в обозревателе решения. И убираем ее создание и отображение в `Program.cs`. Сразу можно запросить уровень поддерживаемых функций у устройства DirectX-a. Для этого вносим изменения в `Program.cs`.

Program.cs:

```
...

using SharpDX.Direct3D;
using Device11 = SharpDX.Direct3D11.Device;

namespace SimpleDirectXApp
{
    static class Program
    {
        /// <summary>
        /// Главная точка входа для приложения.
        /// </summary>
        [STAThread]
        static void Main()
        {
            if (!(Device11.GetSupportedFeatureLevel() == FeatureLevel.Level_11_0))
            {
                MessageBox.Show("DirectX11 Not Supported");
                return;
            }
        }
    }
}
```

Если далее продолжать добавлять функционал в класс Program, то скоро приложение превратится в абсолютно неподдерживаемый монолит. Поэтому разделим на классы, показанные на диаграммах на рис. 2.1 – 2.3.

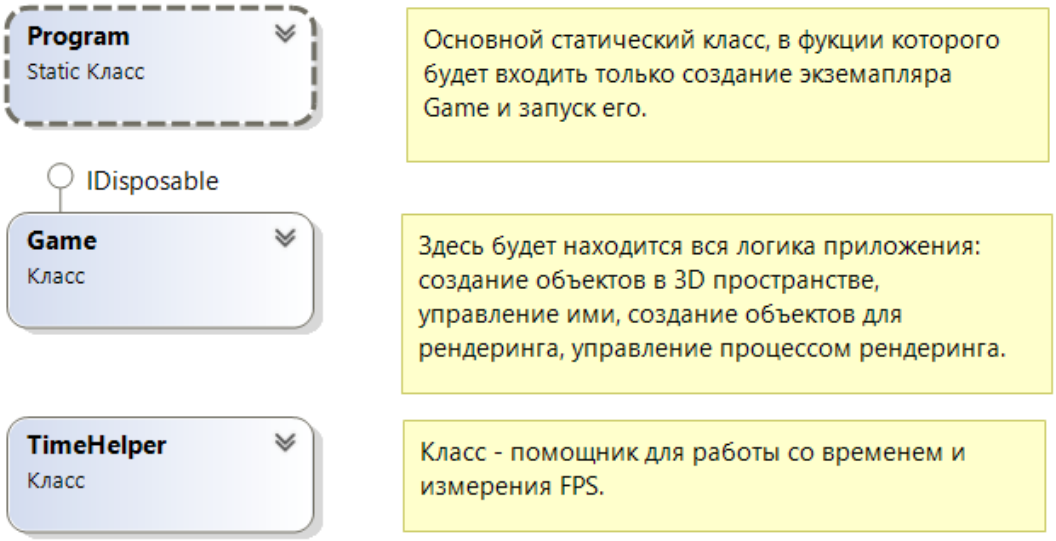

Рис. 2.1. «Основная» диаграмма классов.

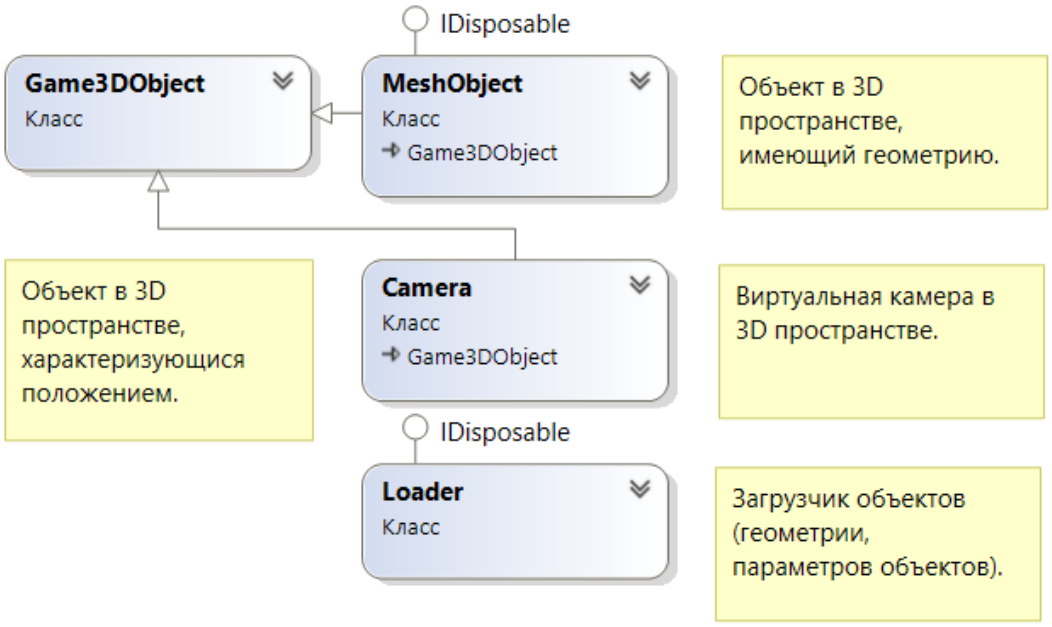

Рис. 2.2. Диаграмма классов, относящихся к объектам в 3D пространстве.

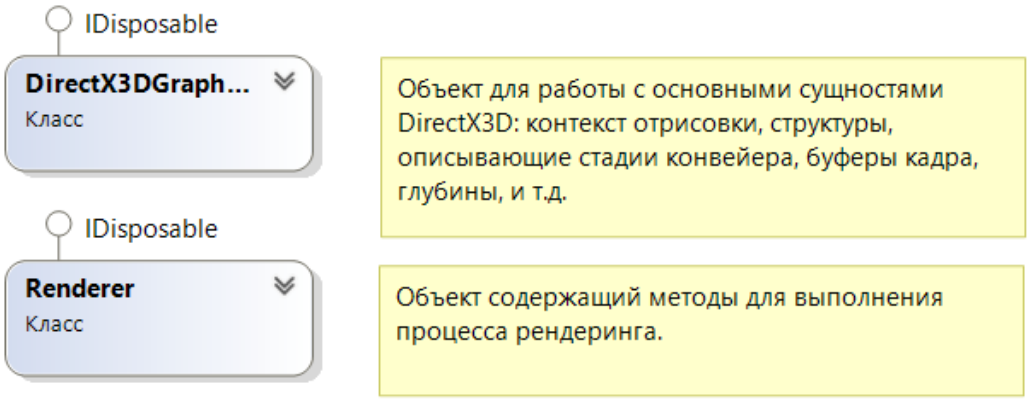

Рис. 2.3. Диаграмма классов, относящихся к непосредственной работе с графическим конвейером DirectX.

Для работы со временем и подсчета кадров в секунду (FPS) добавим класс `TimeHelper`. В нем для измерения временных интервалов и получения текущего времени воспользуемся высокоточным таймером `Stopwatch`, находящимся в пространстве имен `System.Diagnostics`. Для FPS большая точность не нужна, воспользуемся временем, выраженным в миллисекундах. А вот для времени виртуального мира нужно точнее, поскольку оно может использоваться в расчетах физики и генерирования событий. Поэтому воспользуемся счетчиком тактов таймера `Ticks`.

`TimeHelper.cs`:

```
...

using System.Diagnostics;
```

```
7.
8.  namespace SimpleDirectXApp
9.  {
10.     class TimeHelper
11.     {
12.         private Stopwatch _stopwatch;
13.
14.         private int _framesCounter = 0;
15.
16.         private int _fps = 0;
17.         public int FPS { get => _fps; }
18.
19.         private long _previousFPSMeasurementTime;
20.
21.         private long _previousTicks;
22.
23.         private float _time;
24.         public float Time { get => _time; }
25.
26.         private float _deltaT;
27.         public float DeltaT { get => _deltaT; }
28.
29.         public TimeHelper()
30.         {
31.             _stopwatch = new Stopwatch();
32.             Reset();
33.         }
34.
35.         public void Reset()
36.         {
37.             _stopwatch.Reset();
38.             _framesCounter = 0;
39.             _fps = 0;
40.             _stopwatch.Start();
41.             _previousFPSMeasurementTime = _stopwatch.ElapsedMilliseconds;
42.             _previousTicks = _stopwatch.Elapsed.Ticks;
43.         }
44.
45.         public void Update()
46.         {
47.             long ticks = _stopwatch.Elapsed.Ticks;
48.             _time = (float)ticks / TimeSpan.TicksPerSecond;
49.             _deltaT = (float)(ticks - _previousTicks) / TimeSpan.TicksPerSecond;
50.             _previousTicks = ticks;
51.
52.             _framesCounter++;
53.             if (_stopwatch.ElapsedMilliseconds - _previousFPSMeasurementTime >=
    1000)
54.             {
55.                 _fps = _framesCounter;
56.                 _framesCounter = 0;
57.                 _previousFPSMeasurementTime = _stopwatch.ElapsedMilliseconds;
58.             }
59.         }
60.     }
61. }
```

В методе `Update` для получения времени количество тактов таймера делится на `TimeSpan.TicksPerSecond`, поскольку свойство таймера `Elapsed` является экземпляром структуры `TimeSpan`, а `TicksPerSecond`

является статической константой структуры, и обратится к константе можно только разименовав имя структуры. После расчета времени счетчик кадров инкрементируется и, если с момента прошлого обновления значения FPS прошло 1000 миллисекунд, FPS обновляется и счетчик сбрасывается.

Добавим класс `DirectX3DGraphics` для работы с основными объектами DirectX3D.

`DirectX3DGraphics.cs`:

```
...

using SharpDX;
using SharpDX.Direct3D;
using SharpDX.Direct3D11;
using SharpDX.DXGI;
using SharpDX.Windows;
using Device11 = SharpDX.Direct3D11.Device;

namespace SimpleDirectXApp
{
    class DirectX3DGraphics : IDisposable
    {
        private RenderForm _renderForm;
        public RenderForm RenderForm { get => _renderForm; }

        private SampleDescription _sampleDescription;
        public SampleDescription SampleDescription { get => _sampleDescription; }

        private SwapChainDescription _swapChainDescription;

        private Device11 _device;
        public Device11 Device { get => _device; }

        private SwapChain _swapChain;
        public SwapChain SwapChain { get => _swapChain; }

        private DeviceContext _deviceContext;
        public DeviceContext DeviceContext { get => _deviceContext; }

        private RasterizerStateDescription _rasterizerStateDescription;
        private RasterizerState _rasterizerState;

        private Factory _factory;

        private Texture2D _backBuffer;
        public Texture2D BackBuffer { get => _backBuffer; }

        private RenderTargetView _renderTargetView;

        private Texture2DDescription _depthStencilBufferDescription;

        private Texture2D _depthStencilBuffer;

        private DepthStencilView _depthStencilView;

        private bool _isFullScreen;
        public bool IsFullScreen
        {
            get { return _isFullScreen; }
```

```
            set
            {
                if (value != _isFullScreen )
                {
                    _isFullScreen = value;
                    _swapChain.SetFullscreenState(_isFullScreen, null);
                }
            }
        }
```

Рассмотрим назначение полей.

`_renderForm` – ссылка на форму, в которую будет производится рендеринг. Она потребуется для операций инициализации и изменения размера.

`_sampleDescription` – экземпляр структуры, описывающей сэмплирование при растеризации и качество сглаживания при сборке финального изображения, т.е. сколько будет для каждого пикселя выполнено проб на принадлежность графическому примитиву и каково должно быть качество сглаживания. Экземпляр структуры нужен, поскольку она используется в нескольких местах, и параметры сэмплинга должны быть одинаковы.

`_swapChainDescription`, `_swapChain` – структура описания параметров и, непосредственно, сам объект, представляющий собой цепочку буферов кадров.

`_device` – устройство, т.е. непосредственно сам графический ускоритель.

`_deviceContext` – контекст, выполняющий команды отрисовки.

`_rasterizerStateDescription`, `_rasterizerState` – структура описания параметров и, объект, соответствующий стадии растеризации.

`_factory` – фабрика, предназначенная для работы с видеоадаптерами и выполняющая работу при смене видеорежима, переключением между полноэкранным и оконным режимами.

`_backBuffer` – «задний» буфер, в который происходит рендеринг кадра перед отображением на экране.

`_renderTargetView` – объект доступа к «задниму» буферу, которая связывается с портом просмотра (области экрана в которую выводится кадр из «заднего» буфера).

`_depthStencilBufferDescription`, `_depthStencilBuffer` – структура описания параметров и буфер глубины (степень удаленности от камеры) и трафарета (шаблона).

`_depthStencilView` – объект для доступа к буферу глубины и трафарета.

`_isFullScreen` – признак полноэкранного режима.

В конструкторе происходит создание объектов, кроме «заднего» буфера, буфера глубины и трафарета, объектов доступа к ним. Это будет сделано позже, в методе изменения размеров.

DirectX3DGraphics.cs:

```
...


64.         public DirectX3DGraphics(RenderForm renderForm)
65.         {
66.             _renderForm = renderForm;
67.
68.             Configuration.EnableObjectTracking = true;
69.
70.             _sampleDescription = new SampleDescription(1, 0);
71.
72.             _swapChainDescription = new SwapChainDescription()
73.             {
74.                 BufferCount = 1,
75.                 ModeDescription =
76.                     new ModeDescription(_renderForm.ClientSize.Width,
    _renderForm.ClientSize.Height,
77.                         new Rational(60, 1), Format.R8G8B8A8_UNorm),
78.                 IsWindowed = true,
79.                 OutputHandle = _renderForm.Handle,
80.                 SampleDescription = _sampleDescription,
81.                 SwapEffect = SwapEffect.Discard,
82.                 Usage = Usage.RenderTargetOutput
83.             };
84.
85.             Device11.CreateWithSwapChain(
86.                 DriverType.Hardware,
87.                 DeviceCreationFlags.BgraSupport,
88.                 _swapChainDescription,
89.                 out _device,
90.                 out _swapChain);
91.             _deviceContext = _device.ImmediateContext;
92.
93.             _rasterizerStateDescription = new RasterizerStateDescription()
94.             {
95.                 FillMode = FillMode.Solid,
96.                 CullMode = CullMode.Back,
97.                 IsFrontCounterClockwise = true,
98.                 IsMultisampleEnabled = true,
99.                 IsAntialiasedLineEnabled = true,
100.                IsDepthClipEnabled = true
101.            };
102.
103.            _rasterizerState = new RasterizerState(_device,
    _rasterizerStateDescription);
104.            _deviceContext.Rasterizer.State = _rasterizerState;
105.
106.            _factory = _swapChain.GetParent<Factory>();
107.
108.            _factory.MakeWindowAssociation(_renderForm.Handle,
109.                WindowAssociationFlags.IgnoreAll);
110.
111.            _depthStencilBufferDescription = new Texture2DDescription()
112.            {
113.                Format = Format.D32_Float_S8X24_UInt,
114.                ArraySize = 1,
115.                MipLevels = 1,
```

```
116.                 Width = _renderForm.ClientSize.Width,
117.                 Height = _renderForm.ClientSize.Height,
118.                 SampleDescription = _sampleDescription,
119.                 Usage = ResourceUsage.Default,
120.                 BindFlags = BindFlags.DepthStencil,
121.                 CpuAccessFlags = CpuAccessFlags.None,
122.                 OptionFlags = ResourceOptionFlags.None
123.             };
124.         }
```

В строке 68 разрешаем отслеживание объектов на графическом ковейере для обеспечения возможности отладки.

В строке 70 задаем самый простой сэмплинг. Одна проба на пиксель (по его центру) на принадлежность примитиву, 0-ое качество сглаживания.

В строках 72 – 83 задаются параметры цепочки буферов кадров. Строка 74: количество буферов – 1. Для описания желаемого видеорежима (75 – 77) задаются ширина, высота, с помощью рациональной дроби `Rational` запрашиваем 60 кадров в секунду, формат цветности – 8 бит на канал (`Format.R8G8B8A8_UNorm`). Режим отображения – оконный (78). Для вывода изображения на экран DirectX-у необходим Windows HANDLE окна приложения (уникальный дискриптор окна, выдаваемый ядром операционной системы приложению), получаемый из `_renderForm` в строке 79. В строке 80 указываем, ранее созданную структуру, с описанием сэмплинга. 81: при смене кадра – отбрасывать предыдущее содержимое. 82: назначение – вывод результата рендеринга.

В строках 85 – 90 создаем устройство и цепочку буферов. Устройство запрашиваем аппаратное. С поддержкой формата буферов с цветовыми компонентами, расположенными в порядке Blue-Green-Red-Alpha. В данный момент это не обязательно, но позволит, при необходимости, выводить надписи и спрайты средствами DirectX2D поверх отрендеренного кадра.

В строке 91 получаем контекст, выполняющий команды отрисовки. Immediate означает, что контекст «синхронный», т.е. команды отправляются для выполнения на устройства сразу.

В строках 93 – 101 описываем настройки стадии растеризации. `FillMode` – режим заполнения примитивов: `Solid` – сплошное, т.е. поверхность примитива заполнена (можно указать `Wireframe` – каркасное – только границы примитивов). `CullMode` – режим отсечения невидимых граней (три варианта: `None` – показать все, `Front` – отсечь передние, `Back` – отсечь задние) выбираем отсечение задних. `IsFrontCounterClockwise` – определяет какие грани считать передними: обход вершин по часовой стрелке или против (выбираем вариант против часовой стрелки; это потом надо будет учитывать при задании геометрии объектов) при взгляде на нее со стороны камеры.

`IsMultisampleEnabled` – возможен-ли мультисэмплинг со сглаживанием. `IsAntialiasedLineEnabled` – рендеринг примиитивов типа «отрезок» со сглаживанием (к краям треугольников не относится). `IsDepthClipEnabled` – отсечение по глубине, т.е. по расстоянию от камеры.

В строках 103 – 104 создается объект стадии растеризации и устанавливается контексту выполнения команд отрисовки.

В строках 106 – 109 создается фабрика и выполняется ее ассоциация с окном приложения. `WindowAssociationFlags` отвечает за то, какие операции на себя берет фабрика, а какие – программа. Значение флага `WindowAssociationFlags.IgnoreAll` указывает на игнорирование Alt-Enter для переключения полноэкранного режима и PrintScreen, однако поведение в Windows 10 отличается.

В строках 111 – 123 создается структура описания параметров буфера глубины и трафарета. Формат буфера должен быть `Format.D32_Float_S8X24_UInt`: глубина (depth) 32-битный float (`D32_Float`), трафарет – 8-битное беззнаковое целое (`S8...UInt`), неиспользуемое дополнение до кратности 32 битам (`X24`). Размерность массива текстуры `ArraySize` – 1. Уровни уменьшения детализации `MipLevels` – 1. Ширина и высота – по размерам клиентской области формы. Режим сэмплинга указываем тот же, что и при описании цепочки буферов. `Usage` отвечает за доступ к ресурсу со стороны CPU и GPU. Задаем `ResourceUsage.Default`, что означает доступ на чтение и запись со стороны GPU. `BindFlags` указывает назначение создаваемого ресурса. Доступ со стороны CPU не нужен (`CpuAccessFlags.None`). Дополнительные опции не указываем.

В методе `Resize` производятся необходимые манипуляции при изменении размеров области отображения, в нашем случае – формы.

`DirectX3DGraphics.cs`:

```
...

126.         public void Resize()
127.         {
128.             Utilities.Dispose(ref _depthStencilView);
129.             Utilities.Dispose(ref _depthStencilBuffer);
130.             Utilities.Dispose(ref _renderTargetView);
131.             Utilities.Dispose(ref _backBuffer);
132.
133.             _swapChain.ResizeBuffers(_swapChainDescription.BufferCount,
134.                 _renderForm.ClientSize.Width, _renderForm.ClientSize.Height,
135.                 Format.Unknown, SwapChainFlags.None);
136.
137.             _backBuffer = Texture2D.FromSwapChain<Texture2D>(_swapChain, 0);
138.
139.             _renderTargetView = new RenderTargetView(_device, _backBuffer);
```

```
140.
141.             _depthStencilBufferDescription.Width =
    _renderForm.ClientSize.Width;
142.             _depthStencilBufferDescription.Height =
    _renderForm.ClientSize.Height;
143.             _depthStencilBuffer = new Texture2D(_device,
    _depthStencilBufferDescription);
144.
145.             _depthStencilView = new DepthStencilView(_device,
    _depthStencilBuffer);
146.
147.             _deviceContext.Rasterizer.SetViewport(
148.                 new Viewport(0, 0,
149.                 _renderForm.ClientSize.Width, _renderForm.ClientSize.Height,
150.                 0.0f, 1.0f)
151.                 );
152.             _deviceContext.OutputMerger.SetTargets(_depthStencilView,
    _renderTargetView);
153.         }
```

В строках 128 – 131 освобождаем ресурсы, которые необходимо пересоздать.

Строки 133 – 135: цепочке буферов задаем новые ширину и высоту для изменения размера «заднего» буфера, а в строке 137 получаем ресурс измененного буфера. 2-ой параметр `Texture2D.FromSwapChain` – индекс буфера (их может быть несколько) – 0, поскольку в нашем случае только один. В строке 139 создаем объект доступа к «задниму» буферу.

141 – 143: задаем размер и создаем буфер глубины и трафарета. А в строке 145 создаем объект для доступа к нему.

В строках 147 – 151 создаем область просмотра и указываем ее растеризатору. Заголовок конструктора `Viewport(int x, int y, int width, int height, float minDepth, float maxDepth)`. Первые 4 задают положение и размер области. 5-ый и 6-ой указывают в каких пределах глубины (расстояние от камеры) объекты будут отображаться. Причем значения глубины нормализуются, чтобы они лежали в пределах от 0 – у камеры до 1 – самые отделенные. В строке 152 связываем указываем стадии выходного сборщика объекты доступа к буферам.

В методе `ClearBuffers` производится очистка всех буферов, которая будет выполнятся перед рендерингом каждого кадра.

`DirectX3DGraphics.cs`:

```
...


155.         public void ClearBuffers(Color backgroundColor)
156.         {
157.             _deviceContext.ClearDepthStencilView(
158.                 _depthStencilView,
159.                 DepthStencilClearFlags.Depth | DepthStencilClearFlags.Stencil,
160.                 1.0f, 0
```

```
161.                 );
162.             _deviceContext.ClearRenderTargetView(_renderTargetView,
     backgroundColor);
163.         }
```

При очистке буфера глубины и трафарета последние два параметра (строка 160) задают значения, которыми очищаются глубина и трафареты.

В методе `Dispose` освобождаются все выделенные ресурсы в порядке, обратном их созданию.

`DirectX3DGraphics.cs`:

```
...


165.         public void Dispose()
166.         {
167.             Utilities.Dispose(ref _depthStencilView);
168.             Utilities.Dispose(ref _depthStencilBuffer);
169.             Utilities.Dispose(ref _renderTargetView);
170.             Utilities.Dispose(ref _backBuffer);
171.             Utilities.Dispose(ref _factory);
172.             Utilities.Dispose(ref _rasterizerState);
173.             Utilities.Dispose(ref _deviceContext);
174.             Utilities.Dispose(ref _swapChain);
175.             Utilities.Dispose(ref _device);
176.         }
177.     }
178. }
```

Дело в том, что DirectX работает на основе технологии COM (Component Object Model). При создании объектов они создаются внутри библиотек DirectX, а наша программа имеет дело только со ссылками на них.

Перед тем, как перейти к созданию класса `Renderer`, создадим шейдерные программы. Для этого необходимо добавить в проект 2 новых текстовых файла назвав их «`vertex.hlsl`» и «`pixel.hlsl`». Шейдерные программы в DirectX3D пишутся на языке HLSL (High Level Shader Language). Синтаксис языка основан на C. В свойствах обоих файлов необходимо установить «`Копировать в выходной каталог`» в значение «`Всегда копировать`».

Программа вершинного шейдера выполняется для каждой вершины геометрии объектов в 3D пространстве. В ее задачи входит преобразование координат вершин из пространства модели (объекта) в пространство камеры и проецирование, т.е. преобразование в пространство координат экрана.

Выполнить это можно с помощью матриц координатных преобразований. Кроме того, может использоваться для манипуляции фомы объектов.

Вначале задаем формат входных данных для вершинного шейдера.

`vertex.hlsl`:

```
1.  struct vertexData
2.  {
3.      float4 position : POSITION;
4.      float4 color    : COLOR;
5.  };
```

Для формат входных данных объявляем структуру. Поля структуры имеют тип `float4` – вектора из 4 элементов типа `float`. Полей – два: положение вершины в 3D пространстве `position` и цвет `color`. 4-ый компонент вектора координат `w` на входе должен быть у всех вершин равен 1. После выполнения координатных преобразований в нем получится значение, позволяющее вычислить глубину. 4-ый компонент вектора цвета – это прозрачность (Alpha-канал). Ключевые слова `POSITION` и `COLOR` – это семантика, которая используется для указания назначения передаваемых по графическому конвейеру данных. Все, передаваемые по конвейеру данные должны быть размечены соответствующей семантикой.

Далее задаем формат выходных данных, который одновременно является форматом входных данных для пиксельного шейдера.

`vertex.hlsl`:

```
...


7.  struct pixelData
8.  {
9.      float4 position : SV_POSITION;
10.     float4 color    : COLOR;
11. };
```

Разница только в семантике вектора координат. Префикс «`SV_`» (System Value) используется для указания куда передавать данные. В данном случае означает, что данные поступят на вход растеризатора.

Данные, общие для всех вершин объекта, например, матрицы координатный преобразований передаются через константные буферы.

`vertex.hlsl`:

```
...


cbuffer perObjectData : register(b0) {
    float4x4 worldViewProjectionMatrix;
    float    time;
    int      timeScaling;
    float2   _padding;
}
```

Для константных буферов указывается в какой регистр GPU его передавать `b0`, `b1`, … Объявление буферов аналогично структурам. Причем компоненты выравниваются по границе 16 байт, что соответствует 4-хкомпонентному вектору 4-хбайтовых переменных (типа `int`, `float`). Отступ добавляется автоматически. Поэтому лучше задать отступ до 16-тибайтовой границе самим как в шейдерной программе, так и в основной. Здесь для этого используется `_padding`.

Напишем функцию для вершинного шейдера.

`vertex.hlsl`:

```
...


pixelData vertexShader(vertexData input) {
    pixelData output = (pixelData)0;
    float4 position = input.position;

    float scale = 0.5f * sin(time * 0.785f) + 1.0f;
    if (timeScaling > 0) position.xyz = mul(scale, position.xyz);

    output.position = mul(position, worldViewProjectionMatrix);
    output.color = input.color;

    return output;
}
```

В строке 21 объявляется и инициализируется нулями переменная для результата вычислений. Строка 22: объявление временной переменной. В строке 24 вычисляется коэффициент масштабирования на основе значения времени из константного буфера. Строка 25: если установлен признак масштабирования умножаем первых 3 компонента координаты на коэффициент (в 4-ом `w` должна остаться единица). В HLSL нету перегрузки операторов, поэтому для умножения скаляра на вектор, вектора на вектор и т.д. используются встроенные функции. В строке 27 выполняются координатные преобразования (специфика DirectX3D в порядке следования операндов: слева – вектор, справа – матрица, хотя в большинстве случаев – наоборот). Строка 28: цвет не модифицируем.

Программа для пиксельного шейдера выполняется для каждого писеля каждого фрагмента, полученного со стадии растеризации.

`pixel.hlsl`:

```
1.  struct pixelData
2.  {
3.      float4 position : SV_POSITION;
4.      float4 color    : COLOR;
5.  };
6.
7.  float4 pixelShader(pixelData input) : SV_Target
8.  {
9.      return input.color;
10. }
```

Хотя пиксельный шейдер и не имеет доступа к координатам, но объявление формата входных данных должно быть в точности таким же как формат выходных данных вершинного шейдера. Выход пиксельного шейдера должен быть помечен семантикой `SV_Target`, которая указывает, что данные предназначены для выходного сборщика для формирования изображения кадра.

Теперь можно переходить к классу `Renderer`.

`Renderer.cs`:

```
...


6.    using System.Runtime.InteropServices;
7.    using SharpDX;
8.    using SharpDX.DXGI;
9.    using SharpDX.D3DCompiler;
10.   using SharpDX.Direct3D11;
11.   using SharpDX.Direct3D;
12.   using Buffer11 = SharpDX.Direct3D11.Buffer;
13.   using Device11 = SharpDX.Direct3D11.Device;
14.
15.   namespace SimpleDirectXApp
16.   {
17.       class Renderer : IDisposable
18.       {
19.           [StructLayout(LayoutKind.Sequential)]
20.           public struct VertexDataStruct
21.           {
22.               public Vector4 position;
23.               public Vector4 color;
24.           }
25.
26.           [StructLayout(LayoutKind.Sequential)]
27.           public struct PerObjectConstantBuffer
28.           {
29.               public Matrix worldViewProjectionMatrix;
30.               public float time;
31.               public int timeScaling;
```

```
32.             public Vector2 _padding;
33.         }
34.
35.         private DirectX3DGraphics _directX3DGraphics;
36.         private Device11 _device;
37.         private DeviceContext _deviceContext;
38.
39.         private VertexShader _vertexShader;
40.         private PixelShader _pixelShader;
41.         private ShaderSignature _shaderSignature;
42.         private InputLayout _inputLayout;
43.
44.         private PerObjectConstantBuffer _perObjectConstantBuffer;
45.         private Buffer11 _perObjectConstantBufferObject;
```

В строках 19 – 33 объявляются структуры для описания формата входных данных графического конвейера и константного буфера. Поля, их порядок, размерность должны соответствовать тому, как было сделано в шейдерных программах.

Рассмотрим назначение полей класса.

`_directX3DGraphics`, `_device`, `_deviceContext` предназначены для хранения ссылок на используемые классы.

`_vertexShader`, `_pixelShader` – шейдерные программы соответсвующих стадий конвейера.

`_shaderSignature` – объект, содержащий описание данных стадий конвейера, полученных при компиляции шейдерных программ. В нашем случае – только один объект, содержащий информацию о входных данных вершинного шейдера.

`_inputLayout` – объект, содержащий описание данных, подаваемых программой на входной сборщик.

`_perObjectConstantBuffer`, `_perObjectConstantBufferObject` – структура для передачи данных в константный буфер программы вершинного шейдера и объект доступа к буферу.

В конструкторе класса сохраняются в поля ссылки на используемые объекты, компилируются шейдерные программы, формируется описание формата входных данных.

`Renderer.cs`:

```
...


47.         public Renderer(DirectX3DGraphics directX3DGraphics)
48.         {
49.             _directX3DGraphics = directX3DGraphics;
50.             _device = _directX3DGraphics.Device;
51.             _deviceContext = _directX3DGraphics.DeviceContext;
52.
53.             CompilationResult vertexShaderByteCode =
```

```
54.                     ShaderBytecode.CompileFromFile("vertex.hlsl",
55.                     "vertexShader", "vs_5_0");
56.                 _vertexShader = new VertexShader(_device, vertexShaderByteCode);
57.                 CompilationResult pixelShaderByteCode =
58.                     ShaderBytecode.CompileFromFile("pixel.hlsl",
59.                     "pixelShader", "ps_5_0");
60.                 _pixelShader = new PixelShader(_device, pixelShaderByteCode);
61.
62.                 InputElement[] inputElements = new[]
63.                 {
64.                     new InputElement("POSITION", 0, Format.R32G32B32A32_Float,
65.                         0, 0),
66.                     new InputElement("COLOR", 0, Format.R32G32B32A32_Float,
67.                         16, 0)
68.                 };
69.
70.                 _shaderSignature =
71.                     ShaderSignature.GetInputSignature(vertexShaderByteCode);
72.                 _inputLayout = new InputLayout(_device, _shaderSignature,
    inputElements);
73.
74.                 Utilities.Dispose(ref vertexShaderByteCode);
75.                 Utilities.Dispose(ref pixelShaderByteCode);
76.
77.                 _deviceContext.InputAssembler.InputLayout = _inputLayout;
78.                 _deviceContext.VertexShader.Set(_vertexShader);
79.                 _deviceContext.PixelShader.Set(_pixelShader);
80.             }
```

В строках 49 – 51 сохраняем ссылки на используемые объекты.

Строки 53 – 60: загружаем и компилируем шейдерные программы. Здесь строковые константы «`vs_5_0`» и «`ps_5_0`» означают, соответственно вершинный (`vs`) и пиксельный (`ps`) шейдер и шейдерная модель (Shader Model) версии 5.0 (соответствует DirectX 11). Версия шейдерной модели определяет уровень доступного функционала в шейдерных программах.

В строках 62 – 68 задается описание формата входных данных для графического конвейера. Заголовок конструктора `InputElement(string name, int index, Format format, int offset, int slot)`. Параметр `name` должен соответствовать семантике, с которой он был объявлен в шейдерной программе. Параметр `index` нужен, если несколько полей были объявлены с одной и той же семантикой, если такого нет – 0 (как в нашем случае). Параметр `format` задает формат данных. В данном случае оба поля – 4-хкомпонентные векторы элементов типа `float`, соответственно указываем `Format.R32G32B32A32_Float`. Смещение в байтах от начала задает `offset`. Соответственно координата имеет смещение 0. Она занимает 16 байт (4 компонета вектора по 4 байта), тогда для цвета смещение – 16. Параметр `slot` (может быть от 0 до 15) указывает на какой слот входного сборщика подаются данные.

В строках 70 – 72 заданное в программе описание входных данных со стороны нашей программы ставится в соответствие с описпнием,

полученным при компиляции программы вершинного шейдера (`_shaderSignature`).

Поскольку результаты компиляции шейдерных программ (бинарный код) больше не нужны, в строках 74 – 75 освобождаются соответствующие ресурсы.

В строках 77 – 79 графическому конвейеру устанавливается формат входных данных и подключаются шейдерные программы.

В методе `CreateConstantBuffers` создаются константные буферы.

`Renderer.cs`:

```
...

82.         public void CreateConstantBuffers()
83.         {
84.             _perObjectConstantBufferObject = new Buffer11(
85.                 _device,
86.                 Utilities.SizeOf<PerObjectConstantBuffer>(),
87.                 ResourceUsage.Dynamic,
88.                 BindFlags.ConstantBuffer,
89.                 CpuAccessFlags.Write,
90.                 ResourceOptionFlags.None,
91.                 0);
92.         }
```

Поскольку константные буферы обновляются для каждого кадра или даже для объекта, то указываем `ResourceUsage.Dynamic` и возможность записи `CpuAccessFlags.Write`.

Для обновления данных в константном буфере предусмотрим метод `SetPerObjectConstantBuffer`.

`Renderer.cs`:

```
...

94.         public void SetPerObjectConstantBuffer(float time, int timeScaling)
95.         {
96.             _perObjectConstantBuffer.time = time;
97.             _perObjectConstantBuffer.timeScaling = timeScaling;
98.         }
```

Здесь обновляем два поля константного буфера.

Для выполнения действий в начале рендеринга каждого кадра предусмотрим мотод `BeginRender`.

`Renderer.cs`:

```
...


100.          public void BeginRender()
101.          {
102.              _directX3DGraphics.ClearBuffers(Color.Black);
103.          }
```

В данный момент нам нужно только очищение быферов.

Для обновления константного буфера предусмотрим метод `UpdatePerObjectConstantBuffers`.

`Renderer.cs`:

```
...


105.          public void UpdatePerObjectConstantBuffers(Matrix world, Matrix view,
106.              Matrix projection)
107.          {
108.              _perObjectConstantBuffer.worldViewProjectionMatrix =
109.                  Matrix.Multiply(Matrix.Multiply(world, view), projection);
110.              _perObjectConstantBuffer.worldViewProjectionMatrix.Transpose();
111.              DataStream dataStream;
112.              _deviceContext.MapSubresource(
113.                  _perObjectConstantBufferObject,
114.                  MapMode.WriteDiscard,
115.                  SharpDX.Direct3D11.MapFlags.None,
116.                  out dataStream);
117.              dataStream.Write(_perObjectConstantBuffer);
118.              _deviceContext.UnmapSubresource(_perObjectConstantBufferObject, 0);
119.              _deviceContext.VertexShader.SetConstantBuffer(0,
      _perObjectConstantBufferObject);
120.          }
```

Порядок перемножения матриц координатных преобразований (строки 108 – 109) в DirectX обратен обычному.

Перед подачей на графический конвейер результирующая матрица транспонируется в строке 110.

Обновление содержимого буфера выполняется в 4 этапа: создание потока вызовом `MapSubresource` в строках 112 – 116, запись в поток вызовом `Write` в строке 117, освобождение потока вызовом `UnmapSubresource` в строке 118, установка обновленного буфера на вход вершинного шейдера вызовом `SetConstantBuffer` в строке 119.

Метод рендера объекта `RenderMeshObject` создадим позже, после реализации класса `MeshObject`.

Метод `EndRender` предназначен для выполнения действий по завершению рендера кадра. Тут может выполнятся пост-обработка. В нашем случае – просто обновляем содержимое кадра с заменой содержимого предыдущего.

`Renderer.cs`:

```
...


122.          public void EndRender()
123.          {
124.              _directX3DGraphics.SwapChain.Present(1, PresentFlags.Restart);
125.          }
```

Первый параметр `SwapChain.Present` задает время ожидания для смены кадра в интервалах вертикальной синхронизации монитора. Задаем 1 для смены кадров с частотой текущего видеорежима.

В методе `Dispose` освобождаются все выделенные ресурсы в порядке, обратном их созданию.

`Renderer.cs`:

```
...


127.          public void Dispose()
128.          {
129.              Utilities.Dispose(ref _perObjectConstantBufferObject);
130.              Utilities.Dispose(ref _inputLayout);
131.              Utilities.Dispose(ref _shaderSignature);
132.              Utilities.Dispose(ref _pixelShader);
133.              Utilities.Dispose(ref _vertexShader);
134.          }
135.      }
136.  }
```

Добавим класс `Game3DObject` для представления объекта в 3D пространстве. Он будет характеризоваться позицией в 3D пространстве и углами поворота вокруг осей координат. Расположение осей координат в DirectX3D, углы поворота показаны на рис. 2.4. Центр координат пространства камеры находится в центре экрана. По осям X и Y координатное пространство лежит в пределах от -1 до 1. Ось Z направлена вглубь монитора по направлению нашего взгляда (что специфично именно для DirectX3D, поскольку, в большинстве случаев, в геометрии и САПР-ах принято наоборот). Углы поворота объектов отсчитываются по часовой стрелке, если смотреть со стороны стрелки оси к началу координат (это тоже специфично именно для DirectX3D, поскольку в геометрии принято против часовой стрелки).

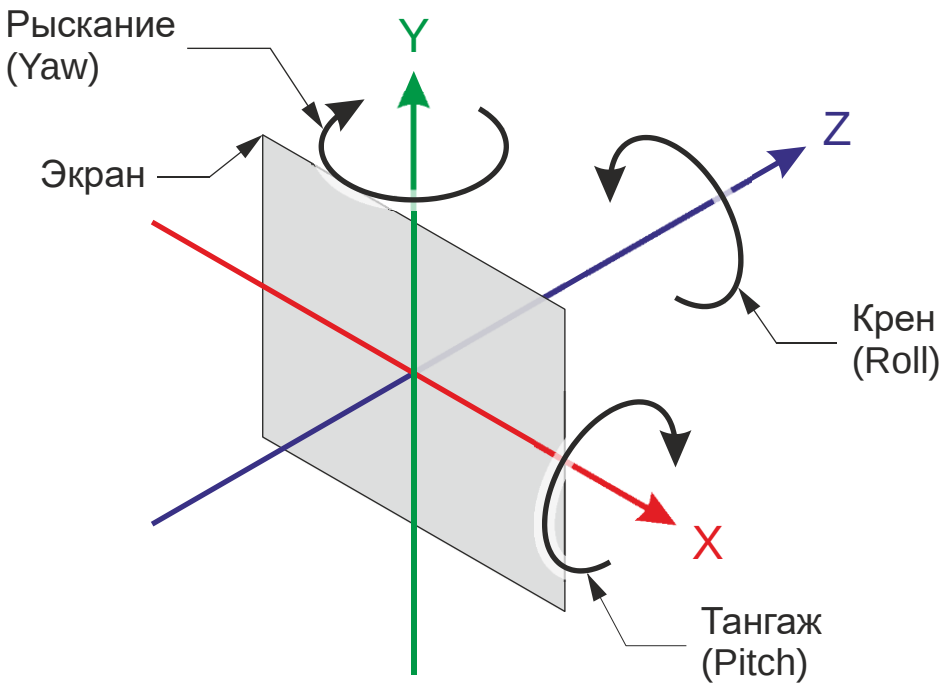

Рис. 2.4. Расположение осей относительно экрана в DirectX3D и углы поворота объектов в 3D пространстве.

Game3DObject.cs:

```
...

using SharpDX;

namespace SimpleDirectXApp
{
    class Game3DObject
    {
        internal Vector4 _position;
        public Vector4 Position { get => _position; }

        internal float _yaw;
        public float Yaw { get => _yaw; set => _yaw = value; }
        internal float _pitch;
        public float Pitch { get => _pitch; set => _pitch = value; }
        internal float _roll;
        public float Roll { get => _roll; set => _roll = value; }

        public Game3DObject(Vector4 position,
            float yaw = 0.0f, float pitch = 0.0f, float roll = 0.0f)
        {
            _position = position;
            _yaw = yaw;
            _pitch = pitch;
            _roll = roll;
        }

        private void LimitAngleByPlusMinusPi(ref float angle)
        {
            if (angle > MathUtil.Pi) angle -= MathUtil.TwoPi;
            else if (angle < -MathUtil.Pi) angle += MathUtil.TwoPi;
        }

```

```
37.         public virtual void YawBy(float deltaYaw)
38.         {
39.             _yaw += deltaYaw;
40.             LimitAngleByPlusMinusPi(ref _yaw);
41.         }
42.
43.         public virtual void PitchBy(float deltaPitch)
44.         {
45.             _pitch += deltaPitch;
46.             LimitAngleByPlusMinusPi(ref _pitch);
47.         }
48.
49.         public virtual void RollBy(float deltaRoll)
50.         {
51.             _roll += deltaRoll;
52.             LimitAngleByPlusMinusPi(ref _roll);
53.         }
54.
55.         public virtual void MoveBy(float deltaX, float deltaY, float deltaZ)
56.         {
57.             _position.X += deltaX;
58.             _position.Y += deltaY;
59.             _position.Z += deltaZ;
60.         }
61.
62.         public virtual void MoveTo(float x, float y, float z)
63.         {
64.             _position.X = x;
65.             _position.Y = y;
66.             _position.Z = z;
67.         }
68.
69.         public Matrix GetWorldMatrix()
70.         {
71.             return Matrix.Multiply(
72.                 Matrix.RotationYawPitchRoll(_yaw, _pitch, _roll),
73.                 Matrix.Translation((Vector3)_position)
74.                 );
75.         }
76.     }
77. }
```

Кроме свойств для углов предусмотрены мотоды изменения угла на заданное приращение (строки 37 – 53). При этом ограничивается значение углов поворота в интервале $[-\pi;\pi]$ вызовом метода `LimitAngleByPlusMinusPi`. Методы `MoveBy` и `MoveTo` служат для относительного и абсолютного перемещения объекта. Метод `GetWorldMatrix` предназначен для создания матрицы координатных преобразований из пространства модели в мировое на основе положения и углов поворота объекта (специфичен для DirectX3D порядок следования операндов при перемножении матриц: обычно слева – перенос, справа – поворот).

Создадим теперь наследника – класс `MeshObject` для объекта в 3D пространстве, имеющего геометрию.

MeshObject.cs:

```
...

using SharpDX;
using SharpDX.Direct3D11;
using Buffer11 = SharpDX.Direct3D11.Buffer;

namespace SimpleDirectXApp
{
    class MeshObject : Game3DObject, IDisposable
    {
        private DirectX3DGraphics _directX3DGraphics;

        private int _verticesCount;
        private Renderer.VertexDataStruct[] _vertices;
        private Buffer11 _vertexBufferObject;
        private VertexBufferBinding _vertexBufferBinding;
        public VertexBufferBinding VertexBufferBinding { get =>
    _vertexBufferBinding; }

        private int _indicesCount;
        public int IndicesCount { get => _indicesCount; }
        private uint[] _indices;
        private Buffer11 _indicesBufferObject;
        public Buffer11 IndicesBufferObject { get => _indicesBufferObject; }

        public MeshObject(DirectX3DGraphics directX3DGraphics,
            Vector4 position, float yaw, float pitch, float roll,
            Renderer.VertexDataStruct[] vertices, uint[] indices)
            : base(position, yaw, pitch, roll)
        {
            _directX3DGraphics = directX3DGraphics;
            _vertices = vertices;
            _verticesCount = _vertices.Length;
            _indices = indices;
            _indicesCount = _indices.Length;

            _vertexBufferObject = Buffer11.Create(
                _directX3DGraphics.Device,
                BindFlags.VertexBuffer,
                _vertices,
                Utilities.SizeOf<Renderer.VertexDataStruct>() * _verticesCount);
            _vertexBufferBinding = new VertexBufferBinding(
                _vertexBufferObject,
                Utilities.SizeOf<Renderer.VertexDataStruct>(),
                0);
            _indicesBufferObject = Buffer11.Create(
                _directX3DGraphics.Device,
                BindFlags.IndexBuffer,
                _indices,
                Utilities.SizeOf<uint>() * _indicesCount);
        }

        public void Dispose()
        {
            Utilities.Dispose(ref _indicesBufferObject);
            Utilities.Dispose(ref _vertexBufferObject);
        }
    }
}
```

Для хранения данных массива вершин предназначены поля `_vertices` и `_verticesCount`. Для массива индексов вершин – `_indices` и `_indicesCount`.

Для подачи данных на вход графического конвейера предназначены `_vertexBufferObject`, `_vertexBufferBinding`, `_indicesBufferObject` и соответствующие свойства, к которым будет обращпться `Renderer`. `_vertexBufferBinding` не является обязательным, но позволит использовать перегруженный метод входного сборщика с меньшим количеством переметров.

В конструкторе, в строках 33 – 37 сохраняем в полях класса необходимые данные. В строках 39 – 43 создается объект для буфера вершин. В строках 44 – 47 создается объект подключения (привязки) ко входному сборщику буфера вершин. Второй параметр `stride` указывает на смещение в буфере между элементами (в нашем случае оно равно размеру элемента – структуры `VertexDataStruct`). Третий параметр `offset` указывает на смещение от начала буфера для данного объекта (в одном буфере можно разместить данные геометрии нескольких объектов). В строках 48 – 52 создается буфер индексов. Дело в том, что одна и та же вершина может входить в состав нескольких графических примитивов. В этом случае при использовании массива индексов можно получить экономию объема используемой памяти и увеличение быстродействия при подаче данных на вход конвейера.

В методе Dispose (строки 55 – 59) освобождаются все выделенные ресурсы в порядке, обратном их созданию.

Теперь верномся к классу `Renderer` и добавим метод `RenderMeshObject` для отрисовки объекта в 3D пространстве.

`Renderer.cs`:

```
...


122.         public void RenderMeshObject(MeshObject meshObject)
123.         {
124.             _deviceContext.InputAssembler.PrimitiveTopology =
     PrimitiveTopology.TriangleList;
125.             _deviceContext.InputAssembler.SetVertexBuffers(0,
126.                 meshObject.VertexBufferBinding);
127.             _deviceContext.InputAssembler.SetIndexBuffer(
     meshObject.IndicesBufferObject,
128.                 Format.R32_UInt, 0);
129.             _deviceContext.DrawIndexed(meshObject.IndicesCount, 0, 0);
130.         }
```

Перед подачей данных на вход графического конвейера задается вид примитивов. Наиболее часто используется массив треугольников `PrimitiveTopology.TriangleList`. Далее подключаем ко входу конвейера буфер вершин. Первый параметр `SetVertexBuffers` – `slot`. Задаем то же значение, что и при описании формата входных данных в конструкторе класса. Затем подключаем буфер индексов. Формат указываем `Format.R32_UInt`, поскольку для массива индексов использовати 32-хбитные бесзнаковые целые. После этого вызываем команду отрисовки нашего объекта. Заголовок метода `DrawIndexed(int indexCount, int startIndexLocation, int baseVertexLocation)`. Первый параметр – количество индексов. Второй – с какого индекса начинать чтение из буфера. Третий – номер вершины в вершинном буфере, с которой начинается отсчет индесов (фактически это значение будет добавлено к каждому индексу, прочитанному из буфера, перед выборкой вершины).

Создадим класс Camera.

`Camera.cs`:

```
...

using SharpDX;

namespace SimpleDirectXApp
{
    class Camera : Game3DObject
    {
        private float _fovY;
        public float FOVY { get => _fovY; set => _fovY = value; }

        private float _aspect;
        public float Aspect { get => _aspect; set => _aspect = value; }

        public Camera(Vector4 position,
            float yaw = 0.0f, float pitch = 0.0f, float roll = 0.0f,
            float fovY = MathUtil.PiOverTwo, float aspect = 1.0f)
            : base(position, yaw, pitch, roll)
        {
            _fovY = fovY;
            _aspect = aspect;
        }

        public Matrix GetProjectionMatrix()
        {
            return Matrix.PerspectiveFovLH(_fovY, _aspect, 0.1f, 100.0f);
        }

        public Matrix GetViewMatrix()
        {
            Matrix rotation = Matrix.RotationYawPitchRoll(_yaw, _pitch, _roll);
            Vector3 viewTo = (Vector3)Vector4.Transform(Vector4.UnitZ, rotation);
            Vector3 viewUp = (Vector3)Vector4.Transform(Vector4.UnitY, rotation);
            return Matrix.LookAtLH((Vector3)_position,
                (Vector3)_position + viewTo, viewUp);
```

```
        }
    }
}
```

Поле `_fovY` (Field Of View) и соответствующее свойство указывают угол обзора камеры в вертикальной плоскости. Поле `_aspect` и соответствующее свойство предназначены для хранения соотношения ширины к высоте.

В конструкторе сохраняем значения в полях.

Метод `GetProjectionMatrix` предназначен для формирования матрицы проецирования. Третий и четвертый параметр метода `PerspectiveFovLH` задают пределы по оси Z. Те объекты, которые выходят за эти пределы, не отображаются. Глубина в соответствующем буфере для объектов на ближней границе будет равна 0, на дальней – 1.

Метод `GetViewMatrix` предназначен для формирования матрицы вида, т.е. преобразования из мировых координат в координаты камеры. Вначале создается матрица поворота. Затем она используется для поворота единичных векторов по осям Z и Y для формирования векторов направления взгляда и направления «вверх». Затем вызвается `Matrix.LookAtLH` для формирования матрицы вида.

Создадим класс `Loader`, который будет предназначен для создания объектов (а в будущем – и для загрузки текстур). В данный момент в классе предусмотрим метод для создания куба, показанного на рис. 2.5, в 3D пространстве.

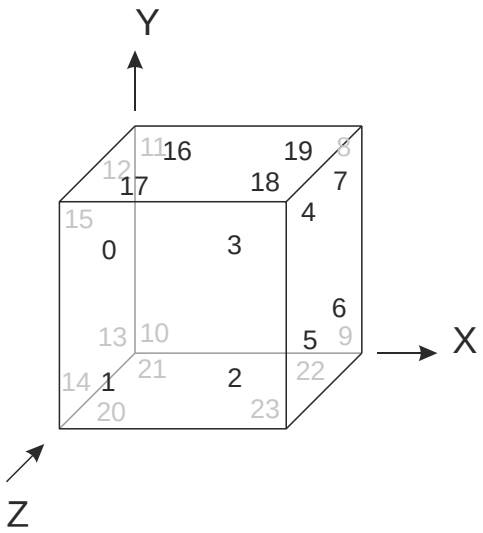

Рис. 2.5. Куб с нумерацией вершин.

Массив индексов вершин, входящих в примитивы (треугольники) составим так, чтобы обход каждого был против часовой стрелки при взгледе с фронтальной стороны треугольника:

– задняя грань куба: 8 – 9 – 10, 10 – 11 – 8;
– левая: 12 – 13 – 14, 14 – 15 – 12;
– нижняя: 20 – 21 – 22, 22 – 23 – 20;
– фронтальная: 0 – 1 – 2, 2 – 3 – 0;
– правая: 4 – 5 – 6, 6 – 7 – 4;
– верхняя: 16 – 17 – 18, 18 – 19 – 16.

`Loader.cs`:

```
...

using SharpDX;

namespace SimpleDirectXApp
{
    class Loader : IDisposable
    {
        private DirectX3DGraphics _directX3DGraphics;

        public Loader(DirectX3DGraphics directX3DGraphics)
        {
            _directX3DGraphics = directX3DGraphics;
        }

        public MeshObject MakeCube(Vector4 position, float yaw, float pitch, float roll)
        {
            Renderer.VertexDataStruct[] vertices =
                new Renderer.VertexDataStruct[24]
                {
                    new Renderer.VertexDataStruct // front 0
                    {
                        position = new Vector4(-1.0f, 1.0f, -1.0f, 1.0f),
                        color = new Vector4(0.0f, 1.0f, 1.0f, 1.0f)
                    },
                    new Renderer.VertexDataStruct // front 1
                    {
                        position = new Vector4(-1.0f, -1.0f, -1.0f, 1.0f),
                        color = new Vector4(0.0f, 1.0f, 1.0f, 1.0f)
                    },
                    new Renderer.VertexDataStruct // front 2
                    {
                        position = new Vector4(1.0f, -1.0f, -1.0f, 1.0f),
                        color = new Vector4(0.0f, 1.0f, 1.0f, 1.0f)
                    },
                    new Renderer.VertexDataStruct // front 3
                    {
                        position = new Vector4(1.0f, 1.0f, -1.0f, 1.0f),
                        color = new Vector4(0.0f, 1.0f, 1.0f, 1.0f)
                    },
                    new Renderer.VertexDataStruct // right 4
                    {
                        position = new Vector4(1.0f, 1.0f, -1.0f, 1.0f),
                        color = new Vector4(1.0f, 0.0f, 1.0f, 1.0f)
                    },
                    new Renderer.VertexDataStruct // right 5
```

```
50.                         {
51.                             position = new Vector4(1.0f, -1.0f, -1.0f, 1.0f),
52.                             color = new Vector4(1.0f, 0.0f, 1.0f, 1.0f)
53.                         },
54.                         new Renderer.VertexDataStruct // right 6
55.                         {
56.                             position = new Vector4(1.0f, -1.0f, 1.0f, 1.0f),
57.                             color = new Vector4(1.0f, 0.0f, 1.0f, 1.0f)
58.                         },
59.                         new Renderer.VertexDataStruct // right 7
60.                         {
61.                             position = new Vector4(1.0f, 1.0f, 1.0f, 1.0f),
62.                             color = new Vector4(1.0f, 0.0f, 1.0f, 1.0f)
63.                         },
64.                         new Renderer.VertexDataStruct // back 8
65.                         {
66.                             position = new Vector4(1.0f, 1.0f, 1.0f, 1.0f),
67.                             color = new Vector4(1.0f, 0.0f, 0.0f, 1.0f)
68.                         },
69.                         new Renderer.VertexDataStruct // back 9
70.                         {
71.                             position = new Vector4(1.0f, -1.0f, 1.0f, 1.0f),
72.                             color = new Vector4(1.0f, 0.0f, 0.0f, 1.0f)
73.                         },
74.                         new Renderer.VertexDataStruct // back 10
75.                         {
76.                             position = new Vector4(-1.0f, -1.0f, 1.0f, 1.0f),
77.                             color = new Vector4(1.0f, 0.0f, 0.0f, 1.0f)
78.                         },
79.                         new Renderer.VertexDataStruct // back 11
80.                         {
81.                             position = new Vector4(-1.0f, 1.0f, 1.0f, 1.0f),
82.                             color = new Vector4(1.0f, 0.0f, 0.0f, 1.0f)
83.                         },
84.                         new Renderer.VertexDataStruct // left 12
85.                         {
86.                             position = new Vector4(-1.0f, 1.0f, 1.0f, 1.0f),
87.                             color = new Vector4(0.0f, 1.0f, 0.0f, 1.0f)
88.                         },
89.                         new Renderer.VertexDataStruct // left 13
90.                         {
91.                             position = new Vector4(-1.0f, -1.0f, 1.0f, 1.0f),
92.                             color = new Vector4(0.0f, 1.0f, 0.0f, 1.0f)
93.                         },
94.                         new Renderer.VertexDataStruct // left 14
95.                         {
96.                             position = new Vector4(-1.0f, -1.0f, -1.0f, 1.0f),
97.                             color = new Vector4(0.0f, 1.0f, 0.0f, 1.0f)
98.                         },
99.                         new Renderer.VertexDataStruct // left 15
100.                        {
101.                            position = new Vector4(-1.0f, 1.0f, -1.0f, 1.0f),
102.                            color = new Vector4(0.0f, 1.0f, 0.0f, 1.0f)
103.                        },
104.                        new Renderer.VertexDataStruct // top 16
105.                        {
106.                            position = new Vector4(-1.0f, 1.0f, 1.0f, 1.0f),
107.                            color = new Vector4(1.0f, 1.0f, 0.0f, 1.0f)
108.                        },
109.                        new Renderer.VertexDataStruct // top 17
110.                        {
111.                            position = new Vector4(-1.0f, 1.0f, -1.0f, 1.0f),
112.                            color = new Vector4(1.0f, 1.0f, 0.0f, 1.0f)
113.                        },
```

```
                    new Renderer.VertexDataStruct // top 18
                    {
                        position = new Vector4(1.0f, 1.0f, -1.0f, 1.0f),
                        color = new Vector4(1.0f, 1.0f, 0.0f, 1.0f)
                    },
                    new Renderer.VertexDataStruct // top 19
                    {
                        position = new Vector4(1.0f, 1.0f, 1.0f, 1.0f),
                        color = new Vector4(1.0f, 1.0f, 0.0f, 1.0f)
                    },
                    new Renderer.VertexDataStruct // bottom 20
                    {
                        position = new Vector4(-1.0f, -1.0f, -1.0f, 1.0f),
                        color = new Vector4(0.0f, 0.0f, 1.0f, 1.0f)
                    },
                    new Renderer.VertexDataStruct // bottom 21
                    {
                        position = new Vector4(-1.0f, -1.0f, 1.0f, 1.0f),
                        color = new Vector4(0.0f, 0.0f, 1.0f, 1.0f)
                    },
                    new Renderer.VertexDataStruct // bottom 22
                    {
                        position = new Vector4(1.0f, -1.0f, 1.0f, 1.0f),
                        color = new Vector4(0.0f, 0.0f, 1.0f, 1.0f)
                    },
                    new Renderer.VertexDataStruct // bottom 23
                    {
                        position = new Vector4(1.0f, -1.0f, -1.0f, 1.0f),
                        color = new Vector4(0.0f, 0.0f, 1.0f, 1.0f)
                    }
                };
            uint[] indices = new uint[36]
            {
                8, 9, 10,       10, 11, 8,
                12, 13, 14,     14, 15, 12,
                20, 21, 22,     22, 23, 20,
                0, 1, 2,        2, 3, 0,
                4, 5, 6,        6, 7, 4,
                16, 17, 18,     18, 19, 16
            };

            return new MeshObject(_directX3DGraphics, position,
                yaw, pitch, roll, vertices, indices);
        }

        public void Dispose()
        {

        }
    }
}
```

В конструкторе сохраняем ссылку на `DirectX3DGraphics` в соответствующее поле.

В методе MakeCube объявляем массивы вершин, индексов в соответствии со сказанным выше и создаем экземпляр `MeshObject`, передав в конструктор массивы.

Метод освобождения ресурсов `Dispose` пока пустой.

Создадим класс `Game`, реализующий основную логику приложения.

`Game.cs`:

```
...

using SharpDX;
using SharpDX.Windows;

namespace SimpleDirectXApp
{
    class Game : IDisposable
    {
        RenderForm _renderForm;

        MeshObject _cube;
        Camera _camera;

        DirectX3DGraphics _directX3DGraphics;
        Renderer _renderer;

        TimeHelper _timeHelper;

        public Game()
        {
            _renderForm = new RenderForm();
            _renderForm.UserResized += RenderFormResizedCallback;
            _directX3DGraphics = new DirectX3DGraphics(_renderForm);
            _renderer = new Renderer(_directX3DGraphics);
            _renderer.CreateConstantBuffers();

            Loader loader = new Loader(_directX3DGraphics);
            _cube = loader.MakeCube(new Vector4(0.0f, 0.0f, 0.0f, 1.0f), 0.0f,
    0.0f, 0.0f);
            _camera = new Camera(new Vector4(0.0f, 2.0f, -10.0f, 1.0f));
            _timeHelper = new TimeHelper();
            loader.Dispose();
            loader = null;
        }

        public void RenderFormResizedCallback(object sender, EventArgs args)
        {
            _directX3DGraphics.Resize();
            _camera.Aspect = _renderForm.ClientSize.Width /
                (float)_renderForm.ClientSize.Height;
        }

        private bool _firstRun = true;

        public void RenderLoopCallback()
        {
            if (_firstRun)
            {
                RenderFormResizedCallback(this, EventArgs.Empty);
                _firstRun = false;
            }
            _timeHelper.Update();
            _renderForm.Text = "FPS: " + _timeHelper.FPS.ToString();
            _cube.YawBy(_timeHelper.DeltaT * MathUtil.TwoPi * 0.1f);

            Matrix viewMatrix = _camera.GetViewMatrix();
            Matrix projectionMatrix = _camera.GetProjectionMatrix();

```

```
62.             _renderer.BeginRender();
63.
64.             _renderer.SetPerObjectConstantBuffer(_timeHelper.Time, 1);
65.
66.             _renderer.UpdatePerObjectConstantBuffers(_cube.GetWorldMatrix(),
67.                 viewMatrix, projectionMatrix);
68.             _renderer.RenderMeshObject(_cube);
69.
70.             _renderer.EndRender();
71.         }
72.
73.         public void Run()
74.         {
75.             RenderLoop.Run(_renderForm, RenderLoopCallback);
76.         }
77.
78.         public void Dispose()
79.         {
80.             _cube.Dispose();
81.             _renderer.Dispose();
82.             _directX3DGraphics.Dispose();
83.         }
84.     }
85. }
```

Поля предусмотрим для всех необходимых объектов.

В контсрукторе создаем объекты в следующем порядке: `RenderForm` (устанавливаем обработчик события изменения размера), затем `DirectX3DGraphics`, затем `Renderer` (вызываем мотод создания константного буфера). После этого можно создавать объекты, расположенные в 3D пространстве (куб и камера) и `TimeHelper`. В конце конструктора вызываем мотод освобождения ресурсов `Loader`-а, поскольку он больше потребуется.

В обработчике события изменения размера `RenderFormResizedCallback` вызываем метод `Resize` объекта `DirectX3DGraphics` для изменения размеров буферов. Обновляем свойство `Aspect` камеры.

Поле `_firstRun`, инициализируемое истиной, нужно, чтобы перед рендерингом первого кадра вызвать `RenderFormResizedCallback` для изменения размеров буферов и установки верного значения соотношения сторон камеры.

В методе рендеринга кадра `RenderLoopCallback` вначале обновляется значение времени и FPS. Вращается куб с частотой 0,1 Гц. Затем получаем от камеры матрицы координатных преобразований. Непосредственно рендеринг начинается с вызова `BeginRender`. Затем обновляются значения в константном буфере (`SetPerObjectConstantBuffer`). Затем обновляем значения матриц координатных преобразований и обновляем константный буфер на

графическом конвейере (`UpdatePerObjectConstantBuffers`). Рендерим куб (`RenderMeshObject`). Завершается рендер вызовом `EndRender`.

Метод `Run` предназначен для запуска цикла `RenderLoop`.

В методе `Dispose` освобождаем выделенные ресурсы.

Все готово. Модифицируем `Program.cs` следующим образом.

`Program.cs`:

```
...


16.         [STAThread]
17.         static void Main()
18.         {
19.             if (!(Device11.GetSupportedFeatureLevel() == FeatureLevel.Level_11_0))

20.             {
21.                 MessageBox.Show("DirectX11 Not Supported");
22.                 return;
23.             }
24.
25.             Game game = new Game();
26.             game.Run();
27.             game.Dispose();
28.         }
```

После запуска видим вращающийся разноцветный куб и значение FPS в заголовке окна, как на рис. 2.6.

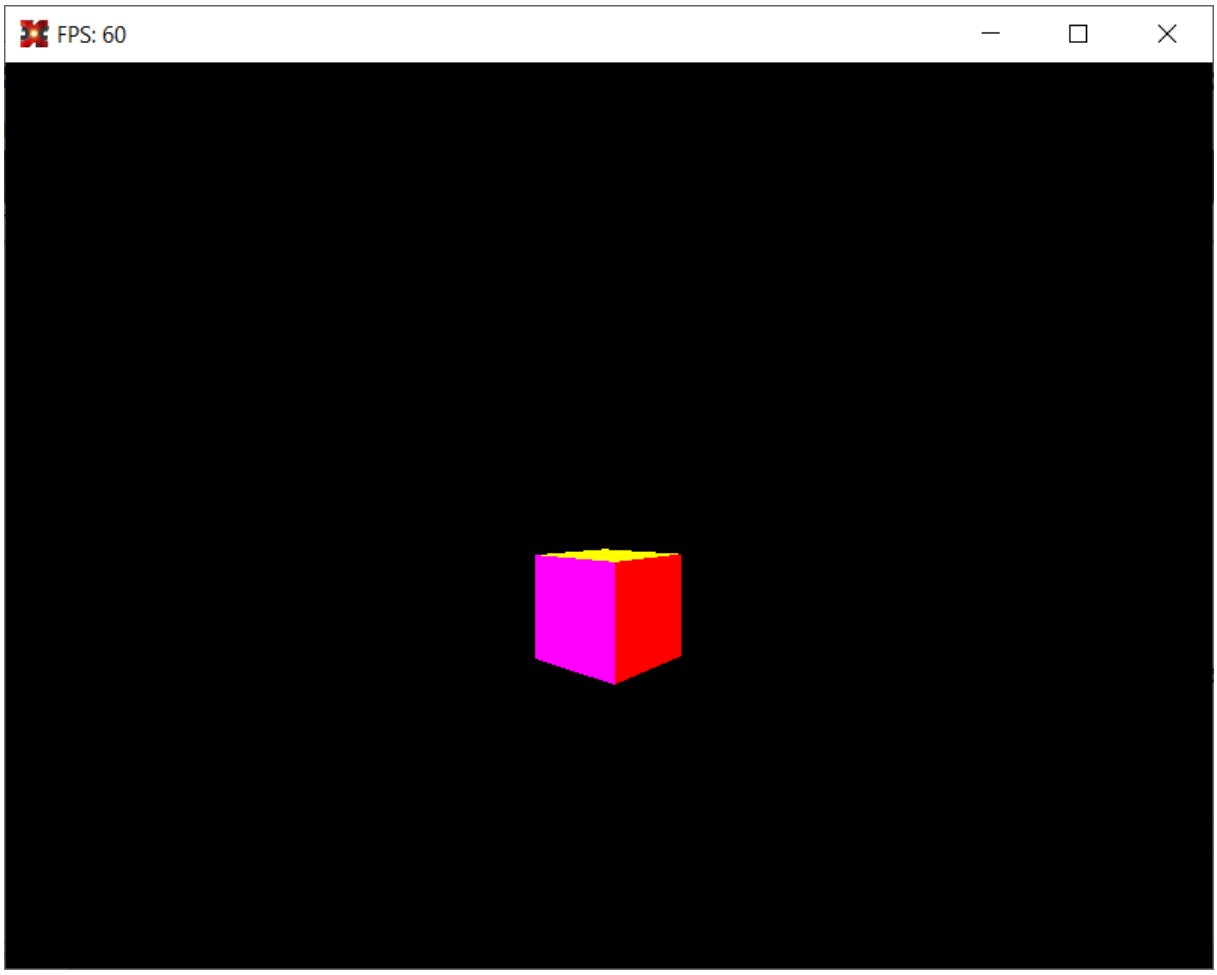

Рис. 2.6. Результат запуска.

2.1. Задания на лабораторую работу

Разработать программу для визуализации указанного в табл. 2.1 трехмерного тела с заданным режимом отображения средствами библиотеки DirectX. Грани тела окрасить в различные цвета.

Таблица 2.1.

Задания на лабораторную работу № 2.

№ вар-та	Трехмерное тело
1	Сфера, сплошная модель.
2	Цилиндр, сплошная модель.
3	Конус, сплошная модель.
4	Тетраэдр, сплошная модель.
5	Октаэдр, сплошная модель.
6	Икосаэдр, сплошная модель.
7	Додекаэдр, сплошная модель.
8	Кубооктаэдр, сплошная модель.

Продолжение таблицы 2.1.

9	Пирамида с квадратным основанием, сплошная модель.
10	Пирамида с пятиугольным основанием, сплошная модель.
11	Пирамида с шестиугольным основанием, сплошная модель.
12	Призма с треугольным основанием, сплошная модель.
13	Параллелепипед, сплошная модель.
14	Призма с пятиугольным основанием, сплошная модель.
15	Призма с шестиугольным основанием, сплошная модель.
16	Четырехугольная антипризма, сплошная модель.
17	Сфера, каркасная модель.
18	Цилиндр, каркасная модель.
19	Конус, каркасная модель.
20	Тетраэдр, каркасная модель.
21	Октаэдр, каркасная модель.
22	Икосаэдр, каркасная модель.
23	Додекаэдр, каркасная модель.
24	Кубооктаэдр, каркасная модель.
25	Пирамида с квадратным основанием, каркасная модель.
26	Пирамида с пятиугольным основанием, каркасная модель.
27	Четырехугольная антипризма, каркасная модель.
28	Пирамида с шестиугольным основанием, каркасная модель.
29	Призма с треугольным основанием, каркасная модель.
30	Параллелепипед, каркасная модель.
31	Призма с пятиугольным основанием, каркасная модель.
32	Призма с шестиугольным основанием, каркасная модель.

Контрольные вопросы:

- Модифицируйте приложение для отображения координатных осей.
- За что отвечает структура `SampleDescription`?
- В каком диапазоне хранится глубина в буфере глубины?
- Каково назначение объекта `DeviceContext`?
- За что отвечает поле `IsAntialiasedLineEnabled` структуры `RasterizerStateDescription`?
- По границе в сколько байт выравниваются константные буферы?

3. Лабораторная работа № 3. Визуализация простейшей сцены с трехмерным текстурированным объектом средствами библиотеки DirectX

Цель работы: ознакомиться с методами загрузки и наложения на 3D объекты текстур, получить практический опыт разработки простейшего приложения рендеринга сцены с текстурированными объектами средствами библиотеки DirectX.

Для добавления текстур в предыдущий пример необходимо добавить в диспетчере пакетов NuGet в проект SharpDX.Direct2D1.

В приложении добавится класс для хранения текстуры и связанных с ней объектов. Часть классов будет модифицировано. Т.о. объектную модель приложения будут иллюстрировать диаграммы классов, приведенные на рис. 2.1, 2.3, 3.1.

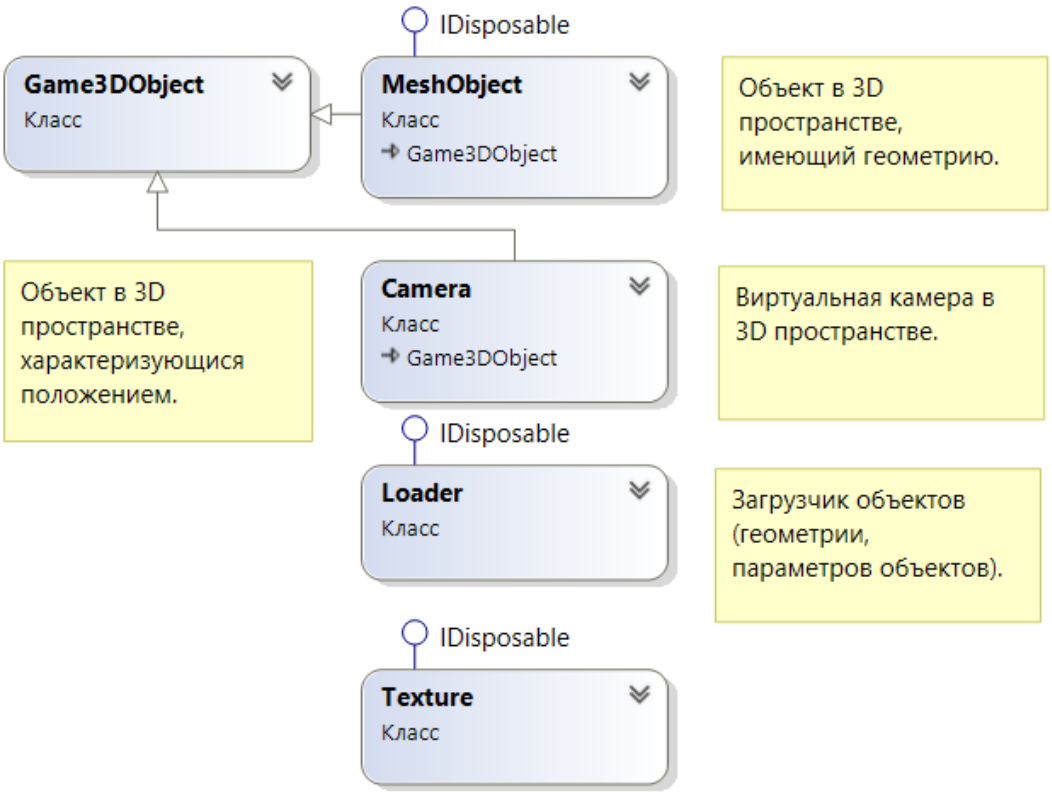

Рис. 3.1. Диаграмма классов, относящихся к объектам в 3D пространстве при использовании текстур.

Добавим в проект класс `Texture`.

`Texture.cs`:

```
...
```

```
using SharpDX;
using SharpDX.Direct3D11;

namespace SimpleDirectXApp
{
    class Texture : IDisposable
    {
        private Texture2D _textureObject;
        public Texture2D TextureObject { get => _textureObject; }

        private ShaderResourceView _shaderResourceView;
        public ShaderResourceView ShaderResourceView { get => _shaderResourceView; }

        private int _width;
        public int Width { get => _width; }

        private int _height;
        public int Height { get => _height; }

        private SamplerState _samplerState;
        public SamplerState SamplerState { get => _samplerState; }

        public Texture(Texture2D textureObject, ShaderResourceView shaderResourceView,
            int width, int height, SamplerState samplerState)
        {
            _textureObject = textureObject;
            _shaderResourceView = shaderResourceView;
            _width = width;
            _height = height;
            _samplerState = samplerState;
        }

        public void Dispose()
        {
            Utilities.Dispose(ref _shaderResourceView);
            Utilities.Dispose(ref _textureObject);
        }
    }
}
```

Поле `_textureObject` и связанное с ним поле предназначены для хранения и доступа к соответствующему объекту DirectX.

Поле `_shaderResourceView` и соответствующее свойство являются объектом доступа шейдерной программы к текстуре, котоый будет непосредственно использоваться при рендеринге.

Поля `_width` и `_height` – ширина и высота текстуры, соответственно.

Поле `_samplerState` и соответствующее свойство являются описанием параметров текстурного интерполятора, используемого для выбора из текстуры цвета для пикселя в кадре.

В конструкторе класса просто копируем значения из параметров в поля класса.

Метод `Dispose` предназначен для освобождения использованных ресурсов.

Изменим шейдерные программы, поскольку теперь вместо цвета по графическому конвейеру будут передаваться текстурные координаты. Также нужно добавить текстуру и текстурный интерполятор в пиксельный шейдер.

`vertex.hlsl`:

```
1.  struct vertexData
2.  {
3.      float4 position  : POSITION;
4.      float2 texCoord0 : TEXCOORD0;
5.  };
6.
7.  struct pixelData
8.  {
9.      float4 position  : SV_POSITION;
10.     float2 texCoord0 : TEXCOORD0;
11. };
12.


...


27.     output.position = mul(position, worldViewProjectionMatrix);
28.     output.texCoord0 = input.texCoord0;
29.
30.     return output;
31. }
```

Теперь вместо цвета с семантикой `COLOR` второе поле структур, описывающих данные, поступающих на вход вершинного шейдера и с его выхода на вход пиксельного, задает текстурные координаты, описываемые семантикой `TEXCOORD0`, `TEXCOORD1` и т.д. Для текстурных координат нам достаточно 2-хкомпонентного вектора (текстуры могут быть и 3D). В вершинном шейдере модифицировать текстурные координаты не будем.

`pixel.hlsl`:

```
1.  struct pixelData
2.  {
3.      float4 position  : SV_POSITION;
4.      float2 texCoord0 : TEXCOORD0;
5.  };
6.
7.  Texture2D meshTexture : register(t0);
8.  sampler   meshSampler : register(s0);
9.
10. float4 pixelShader(pixelData input) : SV_Target
11. {
12.     float4 texColor = {1, 1, 1, 1};
13.     texColor = meshTexture.Sample(meshSampler, input.texCoord0);
14.     return texColor;
```

```
15. }
```

В строках 7 и 8 объявляем объекты для текстуры и параметров текстурного интерполятора. Текстуры располагаются в регистрах `t0`, `t1` и т.д. Параметры интерполятора (сэмплера) – `s0`, `s1` и т.д. В строке 12 объявляем переменную для цвета пикселя кадра и инициализируем единицами. В строке 13 обращаемся к интерполятору для расчета цвета пикселя кадра.

Модифицируем класс `Renderer` для рендера текстурированных объектов.

`Renderer.cs`:

```
...

15.  namespace SimpleDirectXApp
16.  {
17.      class Renderer : IDisposable
18.      {
19.          [StructLayout(LayoutKind.Sequential)]
20.          public struct VertexDataStruct
21.          {
22.              public Vector4 position;
23.              public Vector2 texCoord;
24.          }


...


47.          private SamplerState _anisotropicSampler;
48.          public SamplerState AnisotropicSampler { get => _anisotropicSampler; }
49.
50.          public Renderer(DirectX3DGraphics directX3DGraphics)
51.          {


...


84.              SamplerStateDescription samplerStateDescription =
85.                  new SamplerStateDescription
86.                  {
87.                      Filter = Filter.Anisotropic,
88.                      AddressU = TextureAddressMode.Clamp,
89.                      AddressV = TextureAddressMode.Clamp,
90.                      AddressW = TextureAddressMode.Clamp,
91.                      MipLodBias = 0.0f,
92.                      MaximumAnisotropy = 16,
93.                      ComparisonFunction = Comparison.Never,
94.                      BorderColor = new SharpDX.Mathematics.Interop.RawColor4(
    1.0f, 1.0f, 1.0f, 1.0f),
95.                      MinimumLod = 0,
96.                      MaximumLod = float.MaxValue
97.                  };
98.              _anisotropicSampler = new SamplerState(_directX3DGraphics.Device,
99.                  samplerStateDescription);
100.         }
```

```
...

143.          public void SetTexture(Texture texture)
144.          {
145.              _deviceContext.PixelShader.SetShaderResource(0,
146.                  texture.ShaderResourceView);
147.              _deviceContext.PixelShader.SetSampler(0,
148.                  texture.SamplerState);
149.          }

...

166.          public void Dispose()
167.          {
168.              Utilities.Dispose(ref _anisotropicSampler);

...

174.          }
175.      }
176.  }
```

Структура `VertexDataStruct` теперь вместо цвета, во втором поле будет содержать текстурные координаты (2-хкомпонентный вектор).

Перед конструктором добавим поле `_anisotropicSampler` и соответствующее свойство для хранения параметров текстурного интерполятора. Будем использовать самый затратный по вычислениям, но дающий наилучшее качество способ фильтрации текстур – анизотропный.

В конце конструктора добавим создание объекта с параметрами интерполятора. Рассмотрим поля структуры `SamplerStateDescription`. `Filter` задает способ фильтрации текстуры, т.е. алгоритм расчета результирующего цвета пикселя кадра. Есть вырианты выборки ближайшего текселя (текстурного пикселя), линейной интерполяции (на самом деле билинейной) и анизотропной (усредняется цвет всех текселей попадающих в область, соответствующую пикселю кадра). `AddressU`, `AddressV`, `AddressW` задают поведение при выходе текстурных координат за пределы [0,0; 1,0]: повторять текстуру (замостить графический примитив плиткой), повторять с зеркалированием, повторять цвет крайнего пикселя, использовать цвет бордюра. На данный момент выберем повторять цвет крайнего пикселя. `MipLodBias` пока рассматривать не будем. `MaximumAnisotropy` задает ограничение на качество анизотропной фильтрации (1 – 16, большее значение дает лучшее качество). `ComparisonFunction` задает функцию сравнения с прошлым значением. Поскольку нам это не надо, задаем `Comparison.Never`. `MinimumLod`, `MaximumLod` задают минимальное и максимальное значения для уровня деталей (LOD – Level Of Details). Это используется когда текстура имеет несколько вариантов уменьшения и (или) увеличения для использования на

разном удалении от камеры. Поскольку в данный момент мы этим не пользуемся, задаем значения по умолчанию: 0 и максимальное значение для типа `float`.

Перед методом `RenderMeshObject` добавим метод `SetTexture` для выбора текстуры объекта. В нем используем методы `SetShaderResource` и `SetSampler` для установки нужной текстуры и параметров интерполятора. Первый параметр этих методов `slot` должен быть равен номеру регистров, который был задан программе пиксельного шейдера (мы использовали `t0` и `s0`, соответственно при вызове обоих методов указываем `slot` равным 0).

Вначале метода освобождения ресурсов `Dispose` добавляем строку для освобождения объекта, хранящего параметры текстурного интерполятора.

Добавим в проект текстуру. Развертка куба с номерами вершин показана на рис. 3.2. На рисунке показано начало и направление отсчета текстурных координат для двумерной текстуры, и указаны текстурные координаты для верхних вершин левой грани нашего куба (текстура квадратная). Текстурные координаты $(u; v)$ отсчитываются от вернего левого края (координата $(0; 0)$) до нижнего правого (координата $(1; 1)$). Причем отсчет ведется не по центрам пикселей текстуры, а по сетке линий между пикселями. Размер текстуры по ширине и высоте желательно должен быть кратен степени двойки, а еще лучше равен степени двойки (например 512×128).

Формат текстур может быть различным. Используем для загрузки текстур WIC (Windows Imaging Component). Он может конвертировать изображения из многих форматов растровых изображений (tif, gif, jpg, png и т.д.). В рассматриваемом примере используется формат png. На рис. 3.3 представлена текстура куба. Размер 512×512 пикселей, формат – png.

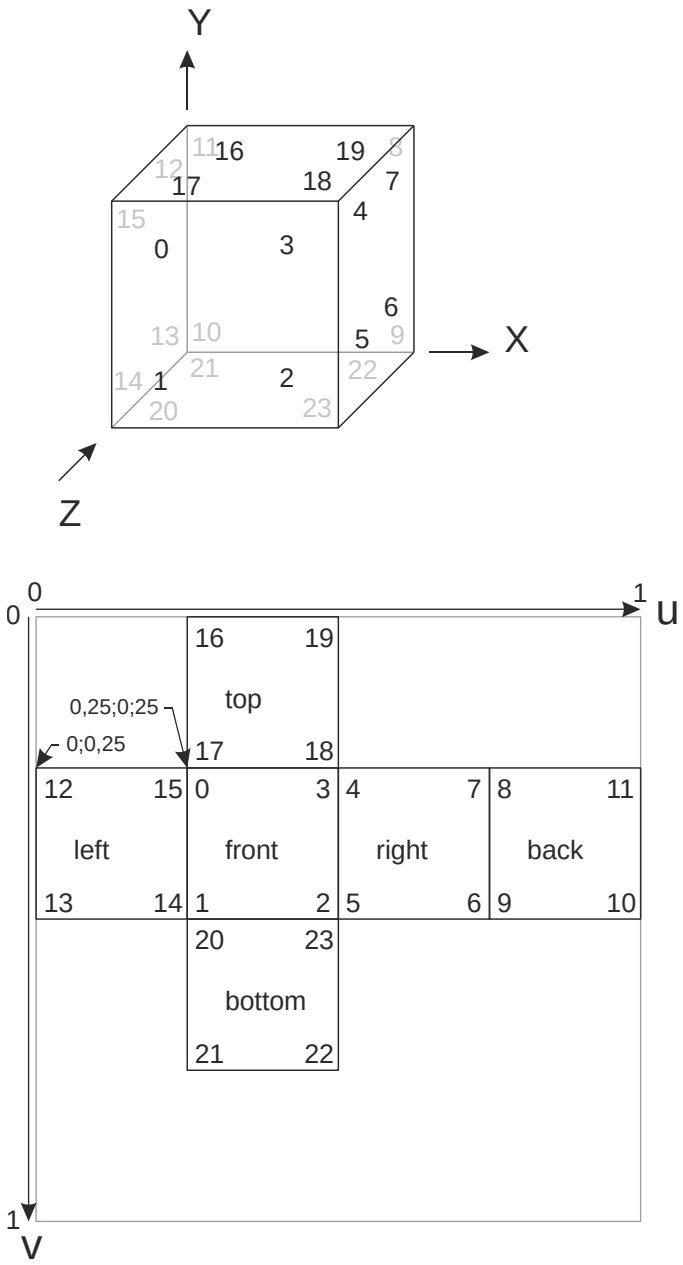

Рис. 3.2. Куб и его развертка.

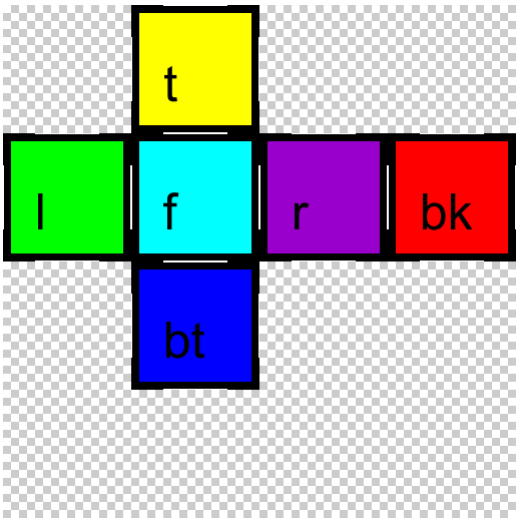

Рис. 3.3. Текстура куба.

Модифицируем класс `Loader` для поддержки загрузки текстур и задания текстурных координат вершин в методе `MakeCube`.

`Loader.cs`:

```
...

using SharpDX.DXGI;
using SharpDX.WIC;
using SharpDX.Direct3D;
using SharpDX.Direct3D11;

namespace SimpleDirectXApp
{
    class Loader : IDisposable
    {
        private DirectX3DGraphics _directX3DGraphics;
        private ImagingFactory _imagingFactory;

        public Loader(DirectX3DGraphics directX3DGraphics)
        {
            _directX3DGraphics = directX3DGraphics;
            _imagingFactory = new ImagingFactory();
        }

        public Texture LoadTextureFromFile(string fileName,
            SamplerState samplerState)
        {
            BitmapDecoder decoder = new BitmapDecoder(_imagingFactory,
                fileName, DecodeOptions.CacheOnDemand);
            BitmapFrameDecode bitmapFirstFrame = decoder.GetFrame(0);

            Utilities.Dispose(ref decoder);

            FormatConverter imageFormatConverter = new FormatConverter(
_imagingFactory);
            imageFormatConverter.Initialize(bitmapFirstFrame,
                PixelFormat.Format32bppRGBA, BitmapDitherType.None, null, 0.0,
                BitmapPaletteType.Custom);
            int stride = imageFormatConverter.Size.Width * 4;
            DataStream buffer = new DataStream(
                imageFormatConverter.Size.Height * stride, true, true);
```

```
41.                 imageFormatConverter.CopyPixels(stride, buffer);
42.
43.                 int width = imageFormatConverter.Size.Width;
44.                 int height = imageFormatConverter.Size.Height;
45.
46.                 Texture2DDescription textureDescription = new Texture2DDescription()
47.                 {
48.                     Width = width,
49.                     Height = height,
50.                     MipLevels = 1,
51.                     ArraySize = 1,
52.                     Format = Format.R8G8B8A8_UNorm,
53.                     SampleDescription = _directX3DGraphics.SampleDescription,
54.                     Usage = ResourceUsage.Default,
55.                     BindFlags = BindFlags.ShaderResource | BindFlags.RenderTarget,
56.                     CpuAccessFlags = CpuAccessFlags.None,
57.                     OptionFlags = ResourceOptionFlags.None
58.                 };
59.                 Texture2D textureObject = new Texture2D(_directX3DGraphics.Device,
60.                     textureDescription, new DataRectangle(buffer.DataPointer,
      stride));
61.                 ShaderResourceViewDescription shaderResourceViewDescription =
62.                     new ShaderResourceViewDescription()
63.                     {
64.                         Dimension = ShaderResourceViewDimension.Texture2D,
65.                         Format = Format.R8G8B8A8_UNorm,
66.                         Texture2D =
67.                             new ShaderResourceViewDescription.Texture2DResource
68.                             {
69.                                 MostDetailedMip = 0,
70.                                 MipLevels = -1
71.                             }
72.                     };
73.                 ShaderResourceView shaderResourceView =
74.                     new ShaderResourceView(_directX3DGraphics.Device, textureObject,
75.                     shaderResourceViewDescription);
76.
77.                 Utilities.Dispose(ref imageFormatConverter);
78.
79.                 return new Texture(textureObject, shaderResourceView, width, height,
80.                     samplerState);
81.             }
```

В классе добавляем поле `_imagingFactory`.

В конструкторе дабавляем создание экземпляра `ImagingFactory`.

После конструктора добавляем метод загрузки текстуры из файла `LoadTextureFromFile`. Порядок загрузки текстуры из файла с использованием WIC следующий. Вначале создается экземпляр декодера `BitmapDecoder` (строки 28 – 29). Последний параметр конструктора задает способ кэширования метаданных изображения: по запросу, или при загрузке. Затем (строка 30) получаем декодированный нужный фрейм (кадр) из изображения. Декодер больше не нужен, освобождаем (строка 32). Затем создаем и инициализируем параметры конвертера формата изображения (строки 34 – 37). Заголовок метода `Initialize(BitmapSource sourceRef, Guid dstFormat, BitmapDitherType dither, Palette`

paletteRef, double alphaThresholdPercent, BitmapPaletteType paletteTranslate). sourceRef – декодированный фрейм. dstFormat задает формат исходного изображения. dither – способ смешения цветов (при использовании палитры). paletteRef – палитра. alphaThresholdPercent задает порог при какой непрозрачности (которая сохраняется в альфа-канал) пиксель будет считаться прозрачным. paletteTranslate задает способ преобразования палитры. В строке 38 вычисляем смещение между строками. В строках 39 – 41 создается буфер, в который выполняется конвертация изображения. Строки 43 – 44: получаем размеры изображения. В строках 46 – 58 задаются параметры текстуры. Width, Height – размеры. MipLevels – количество уровней минификации текстуры (генерация текстур уменьшенного размера для использования для более отдаленных от камеры объектов). Поскольку не используем, задаем 1. ArraySize – размерность для массива текстур. У нас одна текстура, поэтому задаем 1. Формат задаем Format.R8G8B8A8_UNorm – 4 канала, 8 бит на канал, бесзнаковые значения. Параметры мультисэмплинга SampleDescription должны быть такие же, как использовались при создании цепочки буферов и буфера глубины и трафарета. Usage: даем GPU полный доступ к текстуре. BindFlags – ресурс шейдера. CpuAccessFlags задаем без доступа со стороны CPU. OptionFlags в данный момент не используется. В строках 59 – 60 копируем из буфера данные и создаем текстуру. В строках 61 – 72 задаем параметры объекта доступа к текстуре. Размерность – двемерная. Формат такой же, как и при описании текстуры. Параметры двумерной текстуры для нашего случая без минификации. В строках 73 – 75 создается объект доступа к текстуре. В строке 77 освобождается уже не нужный конвертер формата. В строках 79 – 80 создается и возвращается экземпляр нашего класса Texture.

Loader.cs:

```
...


83.         public MeshObject MakeCube(Vector4 position, float yaw, float pitch,
    float roll)
84.         {
85.             Renderer.VertexDataStruct[] vertices =
86.                 new Renderer.VertexDataStruct[24]
87.                 {
88.                     new Renderer.VertexDataStruct // front 0
89.                     {
90.                         position = new Vector4(-1.0f, 1.0f, -1.0f, 1.0f),
91.                         texCoord = new Vector2(0.25f, 0.25f)
92.                     },
93.                     new Renderer.VertexDataStruct // front 1
94.                     {
95.                         position = new Vector4(-1.0f, -1.0f, -1.0f, 1.0f),
96.                         texCoord = new Vector2(0.25f, 0.5f)
97.                     },
98.                     new Renderer.VertexDataStruct // front 2
```

```
99.                         {
100.                            position = new Vector4(1.0f, -1.0f, -1.0f, 1.0f),
101.                            texCoord = new Vector2(0.5f, 0.5f)
102.                        },
103.                        new Renderer.VertexDataStruct // front 3
104.                        {
105.                            position = new Vector4(1.0f, 1.0f, -1.0f, 1.0f),
106.                            texCoord = new Vector2(0.5f, 0.25f)
107.                        },
108.                        new Renderer.VertexDataStruct // right 4
109.                        {
110.                            position = new Vector4(1.0f, 1.0f, -1.0f, 1.0f),
111.                            texCoord = new Vector2(0.5f, 0.25f)
112.                        },
113.                        new Renderer.VertexDataStruct // right 5
114.                        {
115.                            position = new Vector4(1.0f, -1.0f, -1.0f, 1.0f),
116.                            texCoord = new Vector2(0.5f, 0.5f)
117.                        },
118.                        new Renderer.VertexDataStruct // right 6
119.                        {
120.                            position = new Vector4(1.0f, -1.0f, 1.0f, 1.0f),
121.                            texCoord = new Vector2(0.75f, 0.5f)
122.                        },
123.                        new Renderer.VertexDataStruct // right 7
124.                        {
125.                            position = new Vector4(1.0f, 1.0f, 1.0f, 1.0f),
126.                            texCoord = new Vector2(0.75f, 0.25f)
127.                        },
128.                        new Renderer.VertexDataStruct // back 8
129.                        {
130.                            position = new Vector4(1.0f, 1.0f, 1.0f, 1.0f),
131.                            texCoord = new Vector2(0.75f, 0.25f)
132.                        },
133.                        new Renderer.VertexDataStruct // back 9
134.                        {
135.                            position = new Vector4(1.0f, -1.0f, 1.0f, 1.0f),
136.                            texCoord = new Vector2(0.75f, 0.5f)
137.                        },
138.                        new Renderer.VertexDataStruct // back 10
139.                        {
140.                            position = new Vector4(-1.0f, -1.0f, 1.0f, 1.0f),
141.                            texCoord = new Vector2(1.0f, 0.5f)
142.                        },
143.                        new Renderer.VertexDataStruct // back 11
144.                        {
145.                            position = new Vector4(-1.0f, 1.0f, 1.0f, 1.0f),
146.                            texCoord = new Vector2(1.0f, 0.25f)
147.                        },
148.                        new Renderer.VertexDataStruct // left 12
149.                        {
150.                            position = new Vector4(-1.0f, 1.0f, 1.0f, 1.0f),
151.                            texCoord = new Vector2(0.0f, 0.25f)
152.                        },
153.                        new Renderer.VertexDataStruct // left 13
154.                        {
155.                            position = new Vector4(-1.0f, -1.0f, 1.0f, 1.0f),
156.                            texCoord = new Vector2(0.0f, 0.5f)
157.                        },
158.                        new Renderer.VertexDataStruct // left 14
159.                        {
160.                            position = new Vector4(-1.0f, -1.0f, -1.0f, 1.0f),
161.                            texCoord = new Vector2(0.25f, 0.5f)
162.                        },
```

```
163.                         new Renderer.VertexDataStruct // left 15
164.                         {
165.                             position = new Vector4(-1.0f, 1.0f, -1.0f, 1.0f),
166.                             texCoord = new Vector2(0.25f, 0.25f)
167.                         },
168.                         new Renderer.VertexDataStruct // top 16
169.                         {
170.                             position = new Vector4(-1.0f, 1.0f, 1.0f, 1.0f),
171.                             texCoord = new Vector2(0.25f, 0.0f)
172.                         },
173.                         new Renderer.VertexDataStruct // top 17
174.                         {
175.                             position = new Vector4(-1.0f, 1.0f, -1.0f, 1.0f),
176.                             texCoord = new Vector2(0.25f, 0.25f)
177.                         },
178.                         new Renderer.VertexDataStruct // top 18
179.                         {
180.                             position = new Vector4(1.0f, 1.0f, -1.0f, 1.0f),
181.                             texCoord = new Vector2(0.5f, 0.25f)
182.                         },
183.                         new Renderer.VertexDataStruct // top 19
184.                         {
185.                             position = new Vector4(1.0f, 1.0f, 1.0f, 1.0f),
186.                             texCoord = new Vector2(0.5f, 0.0f)
187.                         },
188.                         new Renderer.VertexDataStruct // bottom 20
189.                         {
190.                             position = new Vector4(-1.0f, -1.0f, -1.0f, 1.0f),
191.                             texCoord = new Vector2(0.25f, 0.5f)
192.                         },
193.                         new Renderer.VertexDataStruct // bottom 21
194.                         {
195.                             position = new Vector4(-1.0f, -1.0f, 1.0f, 1.0f),
196.                             texCoord = new Vector2(0.25f, 0.75f)
197.                         },
198.                         new Renderer.VertexDataStruct // bottom 22
199.                         {
200.                             position = new Vector4(1.0f, -1.0f, 1.0f, 1.0f),
201.                             texCoord = new Vector2(0.5f, 0.75f)
202.                         },
203.                         new Renderer.VertexDataStruct // bottom 23
204.                         {
205.                             position = new Vector4(1.0f, -1.0f, -1.0f, 1.0f),
206.                             texCoord = new Vector2(0.5f, 0.5f)
207.                         }
208.                     };

...

221.         }
222.
223.         public void Dispose()
224.         {
225.             Utilities.Dispose(ref _imagingFactory);
226.         }
227.     }
228. }
```

Метод `MakeCube` модифицируем для задания текстурных координат вершин. В методе Dispose добавляем освобождение экземпляра `ImagingFactory`.

После запуска видим вращающийся текстурированный куб, как на рис. 3.4.

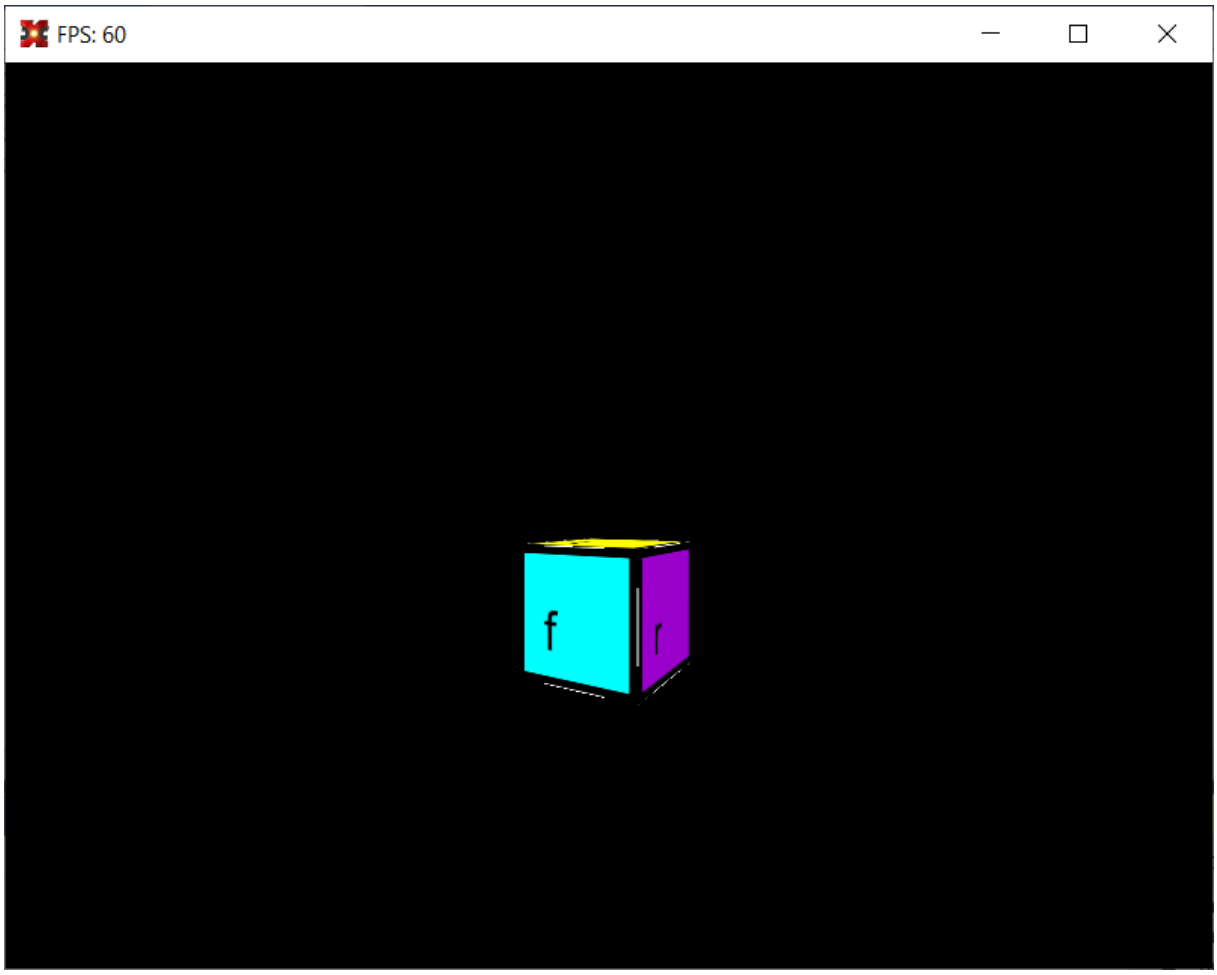

Рис. 3.4. Результат запуска.

3.1. Задания на лабораторую работу

Разработать программу для визуализации указанного в табл. 3.1 трехмерного тела средствами библиотеки DirectX. На трехмерное тело наложить текстуру.

Таблица 3.1.

Задания на лабораторную работу № 3.

№ вар-та	Трехмерное тело
1	Сфера, сплошная модель.
2	Цилиндр, сплошная модель.
3	Конус, сплошная модель.

Продолжение таблицы 3.1.

4	Тетраэдр, сплошная модель.
5	Октаэдр, сплошная модель.
6	Икосаэдр, сплошная модель.
7	Додекаэдр, сплошная модель.
8	Кубооктаэдр.
9	Пирамида с квадратным основанием.
10	Пирамида с пятиугольным основанием.
11	Пирамида с шестиугольным основанием.
12	Призма с треугольным основанием.
13	Параллелепипед.
14	Призма с пятиугольным основанием.
15	Призма с шестиугольным основанием.
16	Четырехугольная антипризма.
17	Сфера.
18	Цилиндр.
19	Конус.
20	Тетраэдр.
21	Октаэдр.
22	Икосаэдр.
23	Додекаэдр.
24	Кубооктаэдр.
25	Пирамида с квадратным основанием.
26	Пирамида с пятиугольным основанием.
27	Четырехугольная антипризма.
28	Пирамида с шестиугольным основанием.
29	Призма с треугольным основанием.
30	Параллелепипед.
31	Призма с пятиугольным основанием.
32	Призма с шестиугольным основанием.

4. Лабораторная работа № 4. Визуализация простейшей сцены с трехмерным объектом средствами библиотеки OpenGL

Цель работы: ознакомиться с библиотекой OpenGL, получить основные понятия о стадиях графического конвейера, получить практический опыт разработки простейшего приложения с использованием библиотеки OpenGL.

Графический конвейер OpenGL 4.4 имеет 9 стадии:

1. Входной сборщик (Input Assembler) в OpenGL именуется толкателем вершин (Vertex Puller).
2. Вершинный шейдер (Vertex Shader).
3. Шейдер управления тесселяцией (Tesselation Control Shader).
4. Тесселятор (Tesselator).
5. Шейдер расчета тесселяции (Tesselation Evaluation Shader).
6. Геометрический шейдер (Geometry Shader).
7. Обратная связь (Transform Feedback).
8. Растеризатор (Rasterizer).
9. Фрагментный шейдер (Fragment Shader).
10. Стадия по-фрагментных операций (Per-fragment Operations).

Часть стадий являются программируемыми, часть только настраиваемы. Для создания простейшего приложения достаточно использования следующих стадий:

- входной сборщик;
- вершинный шейдер;
- растеризатор;
- фрагментный шейдер;
- стадия по-фрагментных операций.

Будем использовать C# и объектно-ориентированную обертку OpenGL-а OpenTK. В Microsoft Visual Studio создаем новый проект типа «Приложение Windows Forms (.NET Framework)». Версию платформы .NET выбраем 4.5.

В диспетчере пакетов NuGet необходимо добавить в проект пакет OpenTK. Далее необходимо выполнить следующие действия:

1. Удалить из проекта форму `Form1`.
2. Добавить в проект новую форму `MainWindow`.
3. Перейти к коду, закрыть конструктор и удалить из проекта `MainWindow.Designer.cs`. Удалить из конструктора формы вызов метода `InitializeComponent`.
4. В раздел подключений добавить:

```
10. using OpenTK;
11. using OpenTK.Graphics;
12. using OpenTK.Graphics.OpenGL4;
13. using OpenTK.Input;
14. using System.IO;
15. using System.Diagnostics;
```

5. Изменить родителя класса `MainWindow` на `GameWindow`.
6. В Program.cs изменить метод Main следующим образом:

```
15.         static void Main()
16.         {
17.             MainWindow window = new MainWindow();
18.             window.Run(60);
19.         }
```

После этих действий проект должен собираться и запускаться, отображая пустое черное окно.

Создадим шейдерные программы. Для этого необходимо добавить в проект 2 новых текстовых файла назвав их «`vertexShader.glsl`» и «`fragmentShader.glsl`». Шейдерные программы в OpenGL пишутся на языке GLSL (Graphics Library Shader Language). Синтаксис языка основан на C. В свойствах обоих файлов необходимо установить «`Копировать в выходной каталог`» в значение «`Всегда копировать`».

Назначение программы вершинного шейдера аналогично соответствующему в DirectX – манипуляции с геометрией объектов.

`vertexShader.glsl`:

```
1.  #version 440 core
```

В первой строке задается требуемая версия спецификации GLSL и профиль (`core` – основной, т.е. будут использоваться только возможности указанной версии; `compatibility` – режим совместимости с прошлыми версиями и т.д.).

Далее объявляется формат входных данных.

`vertexShader.glsl`:

```
...


3.  layout (location = 0) in vec4 position;
4.  layout (location = 1) in vec4 color;
```

Входные данные объявляются с ключевым словом `in`. С помощью ключевого слова `layout` задается расположение элементов входных данных. Расположение задается конструкцией вида `(location = x)`. В нашем случае первым элементом (с индексом 0) для вершины будет координата, вторым – цвет. тип для обоих атрибутов вершины – 4-хкомпонентный вектор чисел с плавающей точкой `float`.

Объявим формат выходных данных.

`vertexShader.glsl`:

```
...
```

```
6.  out vec4 fragColor;
```

Выходные данные объявляются с ключевым словом `out`. На вход фрагментного шейдера нам достаточно подать только цвет, поэтому только один элемент выходных данных.

Данные, общие для всех вершин объекта, например, матрицы координатный преобразований объявляются как константные, с ключевым словом `uniform`.

`vertexShader.glsl`:

```
...


8.  uniform mat4 mvpMatrix;
```

Напишем функцию вершинного шейдера.

`vertexShader.glsl`:

```
...


10. void main(void)
11. {
12.     gl_Position = mvpMatrix * position;
13.     fragColor = color;
14. }
```

Функция вершинного шейдера записывает преобразованную координату во встроенную переменную `gl_Position`. Порядок перемножения при преобразовании: матрица – слева, вектор – справа. Цвет со входа копируется на выход без изменений.

Программа для фрагментного шейдера выполняется для каждого пикселя каждого фрагмента.

`fragmentShader.glsl`:

```
1.  #version 440 core
2.
3.  in vec4 fragColor;
4.  out vec4 color;
5.
6.  void main(void)
7.  {
8.      color = fragColor;
9.  }
```

Результирующий цвет пикселя нужно объявить как выходные данные, встроенной переменной нету.

Вернемся к классу `MainWindow`. Добавим поля.

`MainWindow.cs`:

```
...

19.     public partial class MainWindow : GameWindow
20.     {
21.         private Color4 backColor = new Color4(0.1f, 0.1f, 0.3f, 1.0f);
22.
23.         private const float fovY = (float)Math.PI / 4;
24.         private const float nearDistance = 0.1f;
25.         private const float farDistance = 100.0f;
26.
27.         private float cameraPositionAngleXAxis = 0.0f;
28.         private float cameraPositionAngleYAxis = 0.0f;
29.         private float cameraPositionAngleDelta = (float)Math.PI / 36;
30.         private float cameraPositionDistanceFromOrigin = 5.0f;
31.         private Vector4 startEye = new Vector4(0.0f, 0.0f, 0.0f, 1.0f);
32.         private Vector4 viewTarget = new Vector4(0.0f, 0.0f, 0.0f, 1.0f);
33.         private Vector4 viewYUp = new Vector4(0.0f, 1.0f, 0.0f, 1.0f);
34.
35.         private int program;
36.
37.         private int cubeVertexArray;
38.         private int cubeVertexBuffer;
39.         private int cubeColorBuffer;
40.
41.         private double time;
42.
43.         private Matrix4 modelMatrix;
44.         private Matrix4 viewMatrix;
45.         private Matrix4 projectionMatrix;
46.         private Matrix4 mvpMatrix;
47.
48.         private int mvpMatrixLocation;
```

Рассмотрим назначение полей.

`backColor` – цвет фона.

`fovY`, `nearDistance`, `farDistance` – параметры камеры: угол обзора и пределы расстояния от камеры для отсечения.

`cameraPositionAngleXAxis`, `cameraPositionAngleYAxis`, `cameraPositionAngleDelta` – углы поворота камеры и шаг изменения углов. В данном случае вращать будем не куб, а камеру вокруг него.

`cameraPositionDistanceFromOrigin` – расстояние камеры от центра мировых координат.

`startEye`, `viewTarget`, `viewYUp` – вектора для формирования матрицы вида: положение наблюдателя; точка, куда направлен взгляд; направление вверх.

`program` – имя шейдерной программы (в терминологии OpenGL вместо идентификатора используется «имя»).

`cubeVertexArray` – имя массива, хранящего перечень атрибутов вершин.

`cubeVertexBuffer`, `cubeColorBuffer` – имена буферов координат и цветов вершин.

`time` – текущее время.

`modelMatrix`, `viewMatrix`, `projectionMatrix`, `mvpMatrix` – матрицы координатных преобразований и результирующая.

`mvpMatrixLocation` – ссылка на константную переменную шейдерной программы с матрицей координатных преобразований.

В конструкторе обратимся к конструктору родителя для задания параметров OpenGL и окна.

`MainWindow.cs`:

```
...


50.         public MainWindow()
51.             : base(
52.                   800, 600, GraphicsMode.Default, "", GameWindowFlags.Default,
53.                   DisplayDevice.Default, 4, 4, GraphicsContextFlags.Debug
54.                   )
55.         {
56.             Title += "OpenGL version: " + GL.GetString(StringName.Version);
57.             Load += MainWindow_Load;
58.             Resize += MainWindow_Resize;
59.             KeyDown += MainWindow_KeyDown;
60.             RenderFrame += MainWindow_RenderFrame;
61.         }
```

Заголовок конструктора родительского класса `GameWindow(int width, int height, GraphicsMode mode, string title, GameWindowFlags options, DisplayDevice device, int major, int minor, GraphicsContextFlags flags)`. Первые 2 параметра задают размеры окна (или разрешение видеорежима для полноэкранного режима). Объект `GraphicsMode` позволяет задать параметры графического режима, например, цветность, количество бит на пиксель для буферов глубины и шаблона, и т.д. `title` – заголовок окна. `options` позволяет выбрать полноэкранный режим или запретить изменение размеров окна. `device` позволяет выбрать видеоадаптер. `major` и `minor` задают желаемую версию спецификации OpenGL. `flags` задают различные опции графического контекста. Мы с помощью `GraphicsContextFlags.Debug` включаем дополнительную отладочную информацию (ценой уменьшения производительности).

В строке 56 в заголовок окна добавляем версию OpenGL.
В строках 57 – 60 добавляем обработчики нужным событиям.

Добавим метод компиляции шейдерной программы (для одной стадии конвейера).

`MainWindow.cs`:

```
...


63.         private int CompileShader(ShaderType type, string path)
64.         {
65.             var shader = GL.CreateShader(type);
66.             var src = File.ReadAllText(path);
67.             GL.ShaderSource(shader, src);
68.             GL.CompileShader(shader);
69.             var info = GL.GetShaderInfoLog(shader);
70.             if (!string.IsNullOrWhiteSpace(info))
71.             {
72.                 Debug.WriteLine($"GL.CompileShader [{type}] had info
    log: {info}");
73.             }
74.             return shader;
75.         }
```

В строке 65 создается имя программы шейдера. В строке 66 читается программа из файла и, в строках 67 – 68 она компилируется. В строках 69 – 73 читается журнал компиляции, который при успешной компиляции будет пустым. Если возникли ошибки – выводится в консоль отладки.

В методе `CreateProgram` создается программа для всего конвейера.

`MainWindow.cs`:

```
...


77.         private int CreateProgram()
78.         {
79.             var program = GL.CreateProgram();
80.             var shaders = new List<int>();
81.             shaders.Add(CompileShader(ShaderType.VertexShader,
    "vertexShader.glsl"));
82.             shaders.Add(CompileShader(ShaderType.FragmentShader,
    "fragmentShader.glsl"));
83.             foreach (var shader in shaders)
84.             {
85.                 GL.AttachShader(program, shader);
86.             }
87.             GL.LinkProgram(program);
88.             var info = GL.GetProgramInfoLog(program);
89.             if (!string.IsNullOrWhiteSpace(info))
90.             {
91.                 Debug.WriteLine($"GL.LinkProgram had info log: {info}");
92.             }
93.             foreach (var shader in shaders)
94.             {
```

```
95.                     GL.DetachShader(program, shader);
96.                     GL.DeleteShader(shader);
97.                 }
98.                 return program;
99.             }
```

В строке 79 создается имя для программы для графического конвейера. В строках 80 – 82 создается список скомпилированных программ стадий. В цикле (строки 83 – 86) программы всех стадий подключаются к общей. В строке 87 производится сборка. В строках 88 – 92 читается журнал сборки. Если не пустой – выводится в консоль отладки. После компоноовки общей программы, программы отдельных стадий, не нужны, поэтому в строках 93 – 97 они освобождаются.

Событие `Load` происходит после создания контекста OpenGL, но перед входом в основной цикл. Тут самое место разместить всю логику инициализации: создание массивов атрибутов вершин (положение, цвет), буферов для них, массива перечня атрибутов и т.д. А также зададим некоторые параметры настраиваемых стадий конвейера.

Массив внршин задавать будем иначе, чем в программе с DirectX. Нумерация вершин показана на рис. 4.1. На рисунке также показано направление осей координат в OpenGL. Ось Z, в отличии от DirectX, направлена к нам. Цвет будем задавать вдоль осей (компоненты: X – красный, Y – зеленый, Z – синий) в пределах $[0; 1]$. В результате получим цветовой куб пространства RGB.

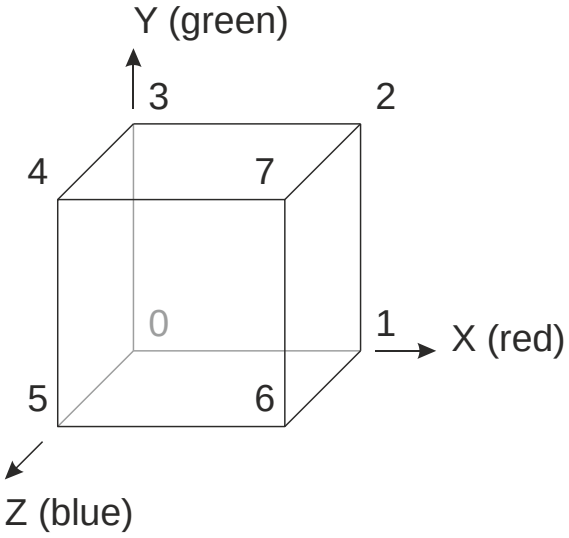

Рис. 4.1. Куб с нумерацией вершин.

Объединение вершин в треугольники и порядок обхода:

- задняя грань: 0 – 3 – 2, 2 – 1 – 0;
- левая: 0 – 5 – 4, 4 – 3 – 0;

– нижняя: 0 – 1 – 6, 6 – 5 – 0;
– фронтальная: 7 – 4 – 5, 5 – 6 – 7;
– правая: 7 – 6 – 1, 1 – 2 – 7;
– верхняя: 7 – 2 – 3, 3 – 4 – 7.

MainWindow.cs:

```
...

        private void MainWindow_Load(object sender, EventArgs e)
        {
            Vector4[] cubeVertices =
            {
                new Vector4(-0.5f, -0.5f, -0.5f, 1.0f), // #0 bottom left back black
                new Vector4(0.5f, -0.5f, -0.5f, 1.0f),  // #1 bottom right back red
                new Vector4(0.5f, 0.5f, -0.5f, 1.0f),   // #2 top right back yellow
                new Vector4(-0.5f, 0.5f, -0.5f, 1.0f),  // #3 top left back green
                new Vector4(-0.5f, 0.5f, 0.5f, 1.0f),   // #4 top left front cyan
                new Vector4(-0.5f, -0.5f, 0.5f, 1.0f),  // #5 bottom left front blue
                new Vector4(0.5f, -0.5f, 0.5f, 1.0f),   // #6 bottom right front magenta
                new Vector4(0.5f, 0.5f, 0.5f, 1.0f)     // #7 top right front white
            };
            Color4[] cubeColors =
            {
                new Color4(0.0f, 0.0f, 0.0f, 0.4f),   // #0 bottom left back black
                new Color4(1.0f, 0.0f, 0.0f, 0.4f),   // #1 bottom right back red
                new Color4(1.0f, 1.0f, 0.0f, 0.4f),   // #2 top right back yellow
                new Color4(0.0f, 1.0f, 0.0f, 0.4f),   // #3 top left back green
                new Color4(0.0f, 1.0f, 1.0f, 0.4f),   // #4 top left front cyan
                new Color4(0.0f, 0.0f, 1.0f, 0.4f),   // #5 bottom left front blue
                new Color4(1.0f, 0.0f, 1.0f, 0.4f),   // #6 bottom right front magenta
                new Color4(1.0f, 1.0f, 1.0f, 0.4f)    // #7 top right front white
            };

            CursorVisible = true;

            program = CreateProgram();
            mvpMatrixLocation = GL.GetUniformLocation(program, "mvpMatrix");

            cubeVertexArray = GL.GenVertexArray();
            GL.BindVertexArray(cubeVertexArray);
            cubeVertexBuffer = GL.GenBuffer();
            GL.BindBuffer(BufferTarget.ArrayBuffer, cubeVertexBuffer);
            Vector4[] vertices =
            {
```

```
137.                 cubeVertices[0], cubeVertices[3], cubeVertices[2],
    cubeVertices[2], cubeVertices[1], cubeVertices[0], // back
138.                 cubeVertices[0], cubeVertices[5], cubeVertices[4],
    cubeVertices[4], cubeVertices[3], cubeVertices[0], // left
139.                 cubeVertices[0], cubeVertices[1], cubeVertices[6],
    cubeVertices[6], cubeVertices[5], cubeVertices[0], // bottom
140.                 cubeVertices[7], cubeVertices[4], cubeVertices[5],
    cubeVertices[5], cubeVertices[6], cubeVertices[7], // front
141.                 cubeVertices[7], cubeVertices[6], cubeVertices[1],
    cubeVertices[1], cubeVertices[2], cubeVertices[7], // right
142.                 cubeVertices[7], cubeVertices[2], cubeVertices[3],
    cubeVertices[3], cubeVertices[4], cubeVertices[7]  // top
143.             };
144.             GL.BufferData(BufferTarget.ArrayBuffer, Vector4.SizeInBytes *
    vertices.Length, vertices, BufferUsageHint.StaticDraw);
145.             GL.EnableVertexAttribArray(0);
146.             GL.VertexAttribPointer(0, 4, VertexAttribPointerType.Float, false,
    0, 0);
147.
148.             cubeColorBuffer = GL.GenBuffer();
149.             GL.BindBuffer(BufferTarget.ArrayBuffer, cubeColorBuffer);
150.             Color4[] colors =
151.             {
152.                 cubeColors[0], cubeColors[3], cubeColors[2], cubeColors[2], cube
    Colors[1], cubeColors[0], // back
153.                 cubeColors[0], cubeColors[5], cubeColors[4], cubeColors[4], cube
    Colors[3], cubeColors[0], // left
154.                 cubeColors[0], cubeColors[1], cubeColors[6], cubeColors[6], cube
    Colors[5], cubeColors[0], // bottom
155.                 cubeColors[7], cubeColors[4], cubeColors[5], cubeColors[5], cube
    Colors[6], cubeColors[7], // front
156.                 cubeColors[7], cubeColors[6], cubeColors[1], cubeColors[1], cube
    Colors[2], cubeColors[7], // right
157.                 cubeColors[7], cubeColors[2], cubeColors[3], cubeColors[3], cube
    Colors[4], cubeColors[7]  // top
158.             };
159.
160.             GL.BufferData(BufferTarget.ArrayBuffer, 16 * colors.Length, colors,
    BufferUsageHint.StaticDraw);
161.             GL.EnableVertexAttribArray(1);
162.             GL.VertexAttribPointer(1, 4, VertexAttribPointerType.Float, false,
    0, 0);
163.
164.             GL.PolygonMode(MaterialFace.Front, PolygonMode.Fill);
165.
166.             GL.Enable(EnableCap.DepthTest);
167.             GL.DepthFunc(DepthFunction.Less);
168.         }
```

В строках 103 – 124 задаются массивы координат и цветов 8-ми вершин куба.

Строка 126: отображение курсора мыши.

Строки 128 – 129: вызываем метода создания шейдерной программы и получаем ссылку на константную переменную шейдерной программы с матрицей координатных преобразований.

В строках 131 – 132 создаем имя перечня атрибутов вершин и подключаем его к контексту.

Строки 133 – 134: создаем имя буфера координат и подключаем.

В строках 135 – 143 задаем массив вершин по треугольникам и, в строке 144 указываем его как источник данных для буфера координат. Первый параметр `BufferData` указывает назначение, `BufferTarget.ArrayBuffer` означает массив атрибутов вершин. Второй параметр – размер данных. Третий – непосредственно данные, т.е. массив вершин. Последний указывает на частоту обновления данных в буфере. `BufferUsageHint.StaticDraw` – данные будут меняться редко (не каждый кадр).

В строках 145 – 146 в массив атрибутов вершин добавляется запись, соответствующая буферу координат. Параметр `EnableVertexAttribArray` и первый параметр `VertexAttribPointer` должны быть равны location, заданным в шейдерной программе. Второй параметр `VertexAttribPointer` задает размер элемента буфера в его компонентах, а третий – тип данных компонента. Пятый (выполнение нормализации данных при конвертации) для типа `float` игнорируется. Последние два: `stride` и `offset` задают смещение в буфере между элементами и смещение от начала буфера. Если `stride = 0`, то фактическое значение определяется на основе типа данных и размера элемента.

В строках 148 – 162 производятся аналогичные действия для цвета вершин.

В строке 164 задаем режим рендеринга примитивов. Видимая сторона – фронтальная (обход вершин против часовой стрелки), сплошная модель.

В строках 166 – 167 включается тест глубины и задается способ теста. При `DepthFunction.Less` если глубина фрагмента меньше значения в буфере глубины, то тест пройден, иначе фрагмент отбрасывается.

Событие `Resize` происходи после изменения размера окна. В нем надо обновить область просмотра и матрицу проецирования для учета возможного изменения соотношения сторон кадра.

`MainWindow.cs`:

```
...


170.         private void MainWindow_Resize(object sender, EventArgs e)
171.         {
172.             GL.Viewport(0, 0, Width, Height);
173.
174.             float aspectRatio = Width / (float)Height;
175.             projectionMatrix = Matrix4.CreatePerspectiveFieldOfView(fovY,
     aspectRatio, nearDistance, farDistance);
176.         }
```

В обработчике события клавиатуры `KeyDown` вращаем камеру.

`MainWindow.cs`:

```
...

178.         private void MainWindow_KeyDown(object sender, KeyboardKeyEventArgs e)
179.         {
180.             switch (e.Key)
181.             {
182.                 case Key.A:
183.                     {
184.                         cameraPositionAngleYAxis += cameraPositionAngleDelta;
185.                         break;
186.                     }
187.                 case Key.D:
188.                     {
189.                         cameraPositionAngleYAxis -= cameraPositionAngleDelta;
190.                         break;
191.                     }
192.                 case Key.S:
193.                     {
194.                         cameraPositionAngleXAxis -= cameraPositionAngleDelta;
195.                         break;
196.                     }
197.                 case Key.W:
198.                     {
199.                         cameraPositionAngleXAxis += cameraPositionAngleDelta;
200.                         break;
201.                     }
202.                 case Key.Escape:
203.                     {
204.                         Exit();
205.                         break;
206.                     }
207.             }
208.         }
```

Обновление матриц координатных преобразований перед рендерингом кадра выделим в отдельный метод.

`MainWindow.cs`:

```
...

210.         private void UpdateMatrices()
211.         {
212.             modelMatrix = Matrix4.Identity;
213.
214.             if (cameraPositionAngleXAxis > (float)Math.PI)
    { cameraPositionAngleXAxis -= (float)Math.PI * 2.0f; }
215.             else if (cameraPositionAngleXAxis < (float)-Math.PI)
    { cameraPositionAngleXAxis += (float)Math.PI * 2.0f; }
216.             if (cameraPositionAngleYAxis > (float)Math.PI)
    { cameraPositionAngleYAxis -= (float)Math.PI * 2.0f; }
217.             else if (cameraPositionAngleYAxis < (float)-Math.PI)
    { cameraPositionAngleYAxis += (float)Math.PI * 2.0f; }
218.             Vector4 eye = startEye;
219.             eye.Z = cameraPositionDistanceFromOrigin;
220.             viewMatrix = Matrix4.CreateRotationY(cameraPositionAngleYAxis) *
```

```
221.                 Matrix4.CreateRotationX(cameraPositionAngleXAxis) *
222.                 Matrix4.LookAt(
223.                     eye.X, eye.Y, eye.Z,
224.                     viewTarget.X, viewTarget.Y, viewTarget.Z,
225.                     viewYUp.X, viewYUp.Y, viewYUp.Z
226.                     );
227.
228.             mvpMatrix = modelMatrix * viewMatrix * projectionMatrix;
229.         }
```

В матрицу преобразования из пространства модели в мировое в строке 221 занесем единичную, поскольку куб будет статичен.

В строках 214 – 217 ограничивается поворот камеры диапазоном $[-\pi;\pi]$.

В строках 218 – 219 задаем положение точки обзора `eye`.

В строках 220 – 226 создаем матрица вида для направления обзора на `viewTarget` и применяем поворот на углы `cameraPositionAngleXAxis` и `cameraPositionAngleYAxis`.

В строке 228 получаем значение результирующей матрицы. Порядок перемножения матриц таков, поскольку в OpenGL используется расположение элементов матрицы в памяти «column major».

Рендеринг кадра осуществляется в методе обработки события `RenderFrame`.

`MainWindow.cs`:

```
...


231.         private void MainWindow_RenderFrame(object sender, FrameEventArgs e)
232.         {
233.             //Title = $" (VSync: {VSync}) FPS: {1f / e.Time:0}";
234.
235.             time += e.Time;
236.
237.             UpdateMatrices();
238.
239.             GL.ClearColor(backColor);
240.             GL.Clear(ClearBufferMask.ColorBufferBit |
    ClearBufferMask.DepthBufferBit);
241.
242.             GL.UseProgram(program);
243.
244.             GL.UniformMatrix4(mvpMatrixLocation, false, ref mvpMatrix);
245.
246.             GL.DrawArrays(PrimitiveType.Triangles, 0, 36);
247.
248.             SwapBuffers();
249.         }
```

Для отображения в заголовке окна значения FPS можно раскомментировать строку 233.

В строке 235 обновляется текущее значение времени (в `e.Time` содержится значение временного интервала между прошлым и текущим кадрами). На данный момент значение времени не используется.

Затем, в строке 237, обновляем матрицы координатных преобразований.

В строке 239 задаем цвет фона, а в 240-ой очищаем буфер кадра и буфер глубины.

В строке 242 указываем используемую шейдерную программу.

В строке 244 обновляем значение матрицы координатных преобразований в памяти GPU. Второй параметр метода `UniformMatrix4 transpose` задает способ расположения матрицы в памяти. Для положения «column major» задаем `false`.

В строке 246 вызывается команда рендеринга объекта. Первый параметр задает тип примитива. Второй – с какого начинать. Третий – количество вершин.

Строка 248: после завершения команд рендеринга вызываем команду смены буферов кадра.

Для корректного освобождения очистки перекроем метод `Dispose(bool manual)`.

`MainWindow.cs`:

```
...


        protected override void Dispose(bool manual)
        {
            GL.DisableVertexAttribArray(1);
            GL.DisableVertexAttribArray(0);
            GL.DeleteBuffer(cubeColorBuffer);
            GL.DeleteBuffer(cubeVertexBuffer);
            GL.DeleteVertexArray(cubeVertexArray);
            GL.DeleteProgram(program);
        }
```

Освобождение ресурсов производится в порядке, обратном созданию.

После запуска получаем куб цветовой модели RGB (см. рис. 4.2) и вращение камеры вокруг него клавишами W, A, S, D.

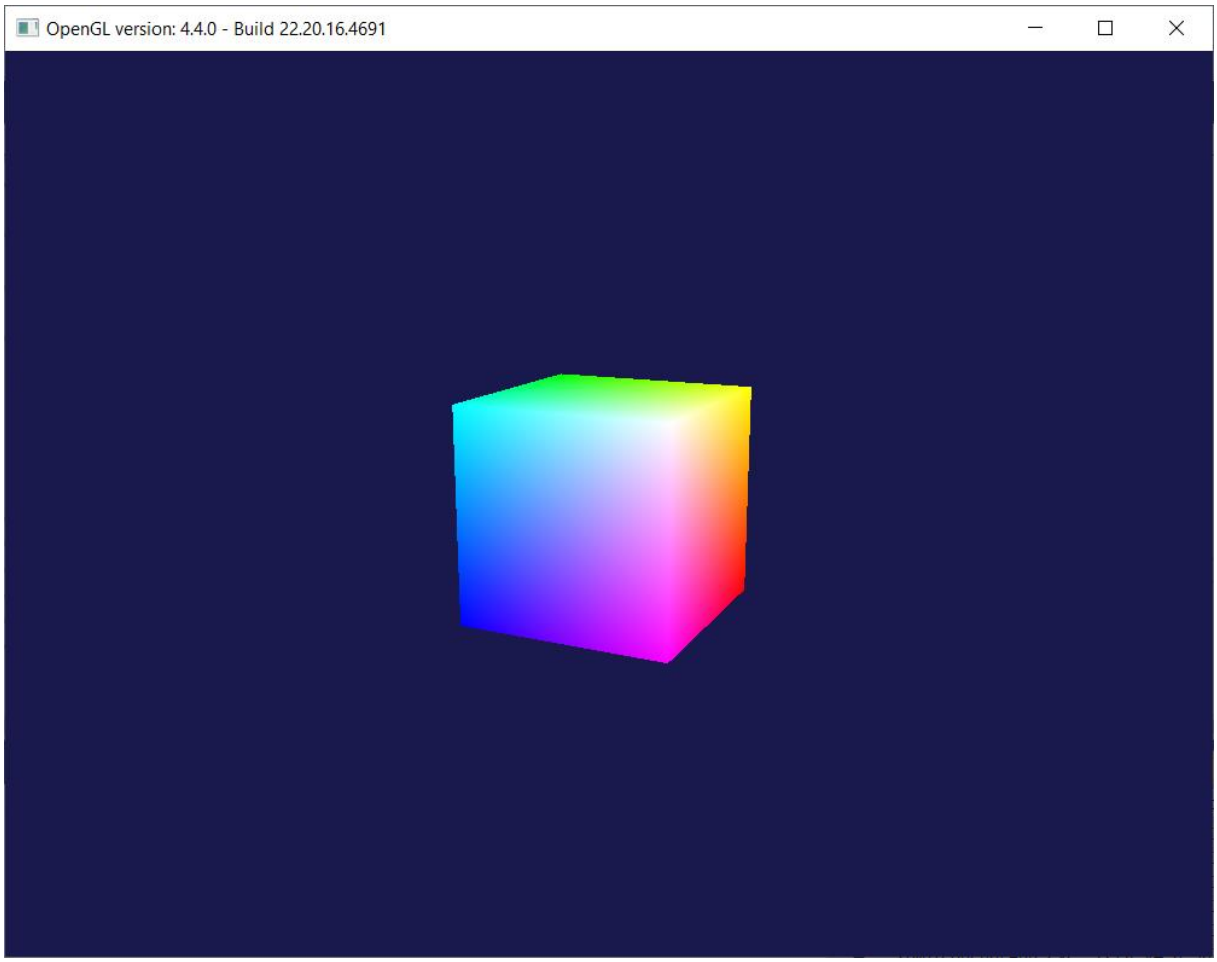

Рис. 4.2. Результат запуска.

4.1. Задания на лабораторую работу

Разработать программу для визуализации указанного в табл. 4.1 трехмерного тела с заданным режимом отображения средствами библиотеки OpenGL. Грани тела окрасить в различные цвета.

Таблица 4.1.

Задания на лабораторную работу № 4.

№ вар-та	Трехмерное тело
1	Сфера, сплошная модель.
2	Цилиндр, сплошная модель.
3	Конус, сплошная модель.
4	Тетраэдр, сплошная модель.
5	Октаэдр, сплошная модель.
6	Икосаэдр, сплошная модель.
7	Додекаэдр, сплошная модель.
8	Кубооктаэдр, сплошная модель.

Продолжение таблицы 4.1.

9	Пирамида с квадратным основанием, сплошная модель.
10	Пирамида с пятиугольным основанием, сплошная модель.
11	Пирамида с шестиугольным основанием, сплошная модель.
12	Призма с треугольным основанием, сплошная модель.
13	Параллелепипед, сплошная модель.
14	Призма с пятиугольным основанием, сплошная модель.
15	Призма с шестиугольным основанием, сплошная модель.
16	Четырехугольная антипризма, сплошная модель.
17	Сфера, каркасная модель.
18	Цилиндр, каркасная модель.
19	Конус, каркасная модель.
20	Тетраэдр, каркасная модель.
21	Октаэдр, каркасная модель.
22	Икосаэдр, каркасная модель.
23	Додекаэдр, каркасная модель.
24	Кубооктаэдр, каркасная модель.
25	Пирамида с квадратным основанием, каркасная модель.
26	Пирамида с пятиугольным основанием, каркасная модель.
27	Четырехугольная антипризма, каркасная модель.
28	Пирамида с шестиугольным основанием, каркасная модель.
29	Призма с треугольным основанием, каркасная модель.
30	Параллелепипед, каркасная модель.
31	Призма с пятиугольным основанием, каркасная модель.
32	Призма с шестиугольным основанием, каркасная модель.

Контрольные вопросы:

- Модифицируйте приложение для отображения координатных осей.
- Что означает `uniform` при объявлении переменной в шейдерной программе?
- Что задает `location` в шейдерной программе?
- Для чего предназначен Vertex Array Object?
- Какие действия нужно выполнить при изменении размеров окна?
- Для чего предназначены `BindBuffer` и `BufferData`?
- Для чего предназначен `PolygonMode`? Какие есть варианты?

5. Лабораторная работа № 5. Визуализация простейшей сцены с трехмерным текстурированным объектом средствами библиотеки OpenGL

Цель работы: ознакомиться с методами загрузки и наложения на 3D объекты текстур, получить практический опыт разработки простейшего приложения рендеринга сцены с текстурированными объектами средствами библиотеки OpenGL.

Изменим шейдерные программы, поскольку теперь вместо цвета по графическому конвейеру будут передаваться текстурные координаты.

`vertexShader.glsl`:

```
1.  #version 440 core
2.
3.  layout (location = 0) in vec4 position;
4.  layout (location = 1) in vec2 inTextureCoordinate;
5.
6.  out vec2 textureCoordinate;
7.
8.  uniform mat4 mvpMatrix;
9.
10. void main(void)
11. {
12.     gl_Position = mvpMatrix * position;
13.     textureCoordinate = inTextureCoordinate;
14. }
```

В строке 4 теперь вместо 4-хкомпонентного цвета 2-хкомпонентный вектор текстурных координат. Строка 6: выходными данными вершинного шейдера также являются текстурные координаты. Строка 13: передаем дальше по конвейеру текстурные координаты без изменений.

`vertexShader.glsl`:

```
1.  #version 440 core
2.
3.  in vec2 textureCoordinate;
4.  uniform sampler2D textureObject;
5.  out vec4 color;
6.
7.  void main(void)
8.  {
9.      color = texelFetch(textureObject, ivec2(textureCoordinate.st), 0);
10. }
```

Входными данными фрагментного шейдера являются текстурные координаты (строка 3). Добавился текстурный интерполятор `sampler2D` в строке 4. В строке 9 производится выборка тексела вызовом функции

`texelFetch`. Также, как и в DirectX, к компонентам векторов можно обращаться через имена компонентов (`x`, `y`, `z` – координаты вершин, `s`, `t` – текстурные координаты), в том числе комбинируя их в вектора (`xy`, `st` и т.д.). Несмотря на то, что координаты передаются в сэмплер в виде 2-хкомпонентного вектора целых чисел (`ivec2`), по графическому конвейеру они должны передаваться вектором чисел с плавающей точкой, поскольку целые и ряд других типов не интерполируются для пикселей фрагмента, а получают одинаковое значение для всего примитива.

Перейдем к классу `MainWindow`. Добавим поля.

`MainWindow.cs`:

```
...

37.         private int cubeVertexArray;
38.         private int cubeVertexBuffer;
39.         private Bitmap bitmap;
40.         private int cubeTextureCoordsBuffer;
41.         private int texture;
```

Вместо массива цветов вершин в строках 39 – 41 объявлены: `bitmap` – объект для чтения текстуры из файла, `cubeTextureCoordsBuffer` – имя буфера массива текстурных координат, texture – имя объекта, соответствующего текстуре (фактически это параметры интерполятора для текстуры и, соответствует этот объект объявленному во фрагментном шейдере `textureObject` типа `sampler2D`).

Добавим в проект текстуру. Развертка куба с номерами вершин показана на рис. 5.1. На рисунке показано начало и направление отсчета текстурных координат для двумерной текстуры, и указаны текстурные координаты для верхних вершин левой грани нашего куба. Текстурные координаты $(s; t)$ отсчитываются от вернего левого края (координата $(0; 0)$) до нижнего правого (координата $(512; 384)$). Причем отсчет ведется не по центрам пикселей текстуры, а по сетке линий между пикселями. Размер текстуры по ширине и высоте желательно должен быть кратен степени двойки, а еще лучше равен степени двойки (например 512×128).

Также добавим в проект текстуру. Для простоты будем использовать формат bmp. При этом мы сможем загрузить текстуру стандартными средствами .NET. Текстура куба показана на рис. 5.2 (размер 512×384 пикселей).

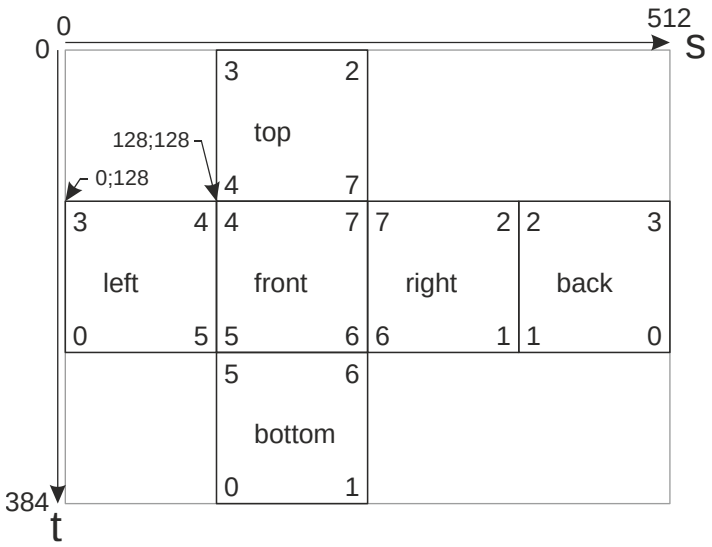

Рис. 5.1. Развертка куба с текстурными координатами.

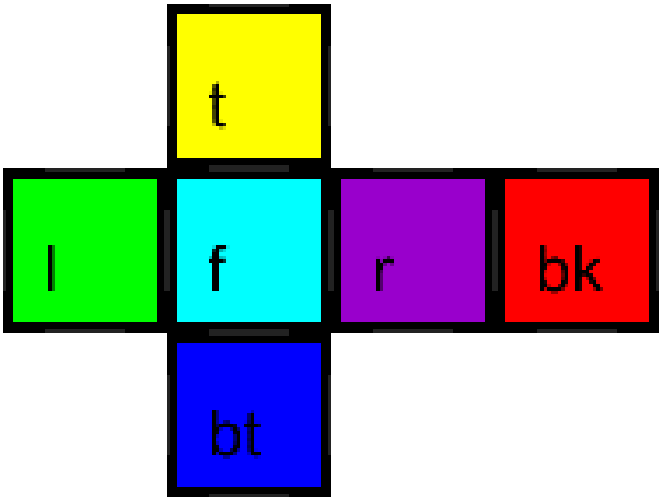

Рис. 5.2. Текстура куба.

Изменим метод обработки события `OnLoad` для поддержки работы с текстурами.

`MainWindow.cs`:

```
...


103.          private void MainWindow_Load(object sender, EventArgs e)
104.          {


...


137.              GL.VertexAttribPointer(0, 4, VertexAttribPointerType.Float, false,
      0, 0);
138.
139.              Vector2[] cubeTexCoords =
140.              {
141.                  new Vector2(512, 256), new Vector2(512, 128), new Vector2(384,
      128),      // back
```

```
                    new Vector2(384, 128), new Vector2(384, 256), new Vector2(512, 256),
                    new Vector2(0, 256), new Vector2(128, 256), new Vector2(128, 128),       // left
                    new Vector2(128, 128), new Vector2(0, 128), new Vector2(0, 256),
                    new Vector2(128, 384), new Vector2(256, 384), new Vector2(256, 256),     // bottom
                    new Vector2(256, 256), new Vector2(128, 256), new Vector2(128, 384),
                    new Vector2(256, 128), new Vector2(128, 128), new Vector2(128, 256),     // front
                    new Vector2(128, 256), new Vector2(256, 256), new Vector2(256, 128),
                    new Vector2(256, 128), new Vector2(256, 256), new Vector2(384, 256),     // right
                    new Vector2(384, 256), new Vector2(384, 128), new Vector2(256, 128),
                    new Vector2(256, 128), new Vector2(256, 0), new Vector2(128, 0),         // top
                    new Vector2(128, 0), new Vector2(128, 128), new Vector2(256, 128)
                };

                cubeTextureCoordsBuffer = GL.GenBuffer();
                GL.BindBuffer(BufferTarget.ArrayBuffer, cubeTextureCoordsBuffer);
                GL.BufferData(BufferTarget.ArrayBuffer, Vector2.SizeInBytes * cubeTexCoords.Length, cubeTexCoords, BufferUsageHint.StaticDraw);
                GL.EnableVertexAttribArray(1);
                GL.VertexAttribPointer(1, 2, VertexAttribPointerType.Float, false, 0, 0);

                bitmap = new Bitmap("Tex.bmp");
                Vector3[] textureData = new Vector3[bitmap.Height * bitmap.Width];
                int i = 0;
                for (int y = 0; y < bitmap.Height; y++)
                {
                    for (int x = 0; x < bitmap.Width; x++)
                    {
                        Color p = bitmap.GetPixel(x, y);
                        textureData[i++] = new Vector3(p.R / 255f, p.G / 255f, p.B / 255f);
                    }
                }
                texture = GL.GenTexture();
                GL.BindTexture(TextureTarget.Texture2D, texture);
                GL.TexImage2D(TextureTarget.Texture2D, 0, PixelInternalFormat.Rgb32f, bitmap.Width, bitmap.Height, 0, PixelFormat.Rgb, PixelType.Float, textureData);
                GL.TexParameterI(TextureTarget.Texture2D, TextureParameterName.TextureMagFilter, new int[] { (int)TextureMagFilter.Nearest });
                GL.TexParameterI(TextureTarget.Texture2D, TextureParameterName.TextureMinFilter, new int[] { (int)TextureMinFilter.Nearest });

                GL.PolygonMode(MaterialFace.Front, PolygonMode.Fill);

                GL.Enable(EnableCap.DepthTest);
                GL.DepthFunc(DepthFunction.Less);
            }
```

Удалим массив `cubeColors`, бывший в строках 114 – 124, массив `colors`, бывший в строках 150 – 158. Также действия, с ним связанные: в строках 148 – 149 вызовы `GenBuffer` и `BindBuffer`. Находившиеся в строках 160 – 162 вызовы `BufferData`, `EnableVertexAttribArray`, `VertexAttribPointer` изменим. Изменения рассмотрим позже.

После строки 137 с вызовом `VertexAttribPointer(0, …` добавим (в строках 139 – 153) массив текстурных координат вершин с учетом того, как вершины расположены в буфере.

В строке 155 создается имя для буфера текстурных координат, а в строке 156 он подключается к контексту. В строке 157 связываем его с данными из массива текстурных координат.

В строках 158 и 159 задаются параметры атрибута для текстурных координат. Изменилась только размерность атрибута. Ранее был 4-хкопонентный цвет, сейчас – 2-хкомпонентные текстурные координаты.

В строке 161 создаем `Bitmap` и читаем изображение из файла.

В строке 162 создаем массив для данных цвета пикселей. Элементы – 3-хкомпонентный вектор чисел с плавающей точкой. Соответственно, в строках 163 – 171 конвертируем в этот формат данные, прочитанные из файла.

В строке 172 создаем имя для текстуры и, в строке 173 подключаем ее к контексту.

В строках 174 – 176 задаем формат и параметры текстуры. Заголовок конструктора `TexImage2D<T8>(TextureTarget target, int level, PixelInternalFormat internalformat, int width, int height, int border, PixelFormat format, PixelType type, T8[] pixels)`. Первый параметр задает вид текстуры: 1D, 2D, 3D и т.д. Второй параметр задает уровень детализации (LOD – Level Of Details). Третий задает формат текстуры: каналы и их разрядность. 4-ый и 5-ый – размеры текстуры. 6-ой – ширина рамки. 7-ой и 8-ой задают формат данных массива, который мы передаем OpenGL. Последний параметр – непосредственно данные текстуры. В строках 175 и 176 задаются параметры фильтрации текстуры при использовании увеличенного и уменьшенного размеров. Самый простой вариант – `TextureMagFilter.Nearest`.

Изменим метод `UpdateMatrices`.

`MainWindow.cs`:

```
...


224.         private void UpdateMatrices()
225.         {
226.             modelMatrix = Matrix4.CreateRotationX((float)(2.0f * Math.PI *
     0.02f * time)) *
```

```
227.                 Matrix4.CreateRotationY((float)(2.0f * Math.PI * 0.007f * time))
    ;

...

244.         }
```

В строках 226 – 227 метода вместо единичной матрицы для преобразования из координат модели в мировые зададим вращение куба в зависимости от времени.

Метод рендеринга кадра остается без изменений.

Изменим метод освобождения ресурсов.

MainWindow.cs:

```
...

266.         protected override void Dispose(bool manual)
267.         {
268.             GL.DisableVertexAttribArray(1);
269.             GL.DisableVertexAttribArray(0);
270.             GL.DeleteTexture(texture);
271.             GL.DeleteBuffer(cubeTextureCoordsBuffer);
272.             GL.DeleteBuffer(cubeVertexBuffer);
273.             GL.DeleteVertexArray(cubeVertexArray);
274.             GL.DeleteProgram(program);
275.         }
276.     }
277. }
```

В строке 270 добавилось удатение текстуры. А в строке 271 изменился удаляемый буфер.

После запуска получаем вращающийся куб, как на рис. 5.3, и перемещение вокруг него камеры клавишами W, A, S, D.

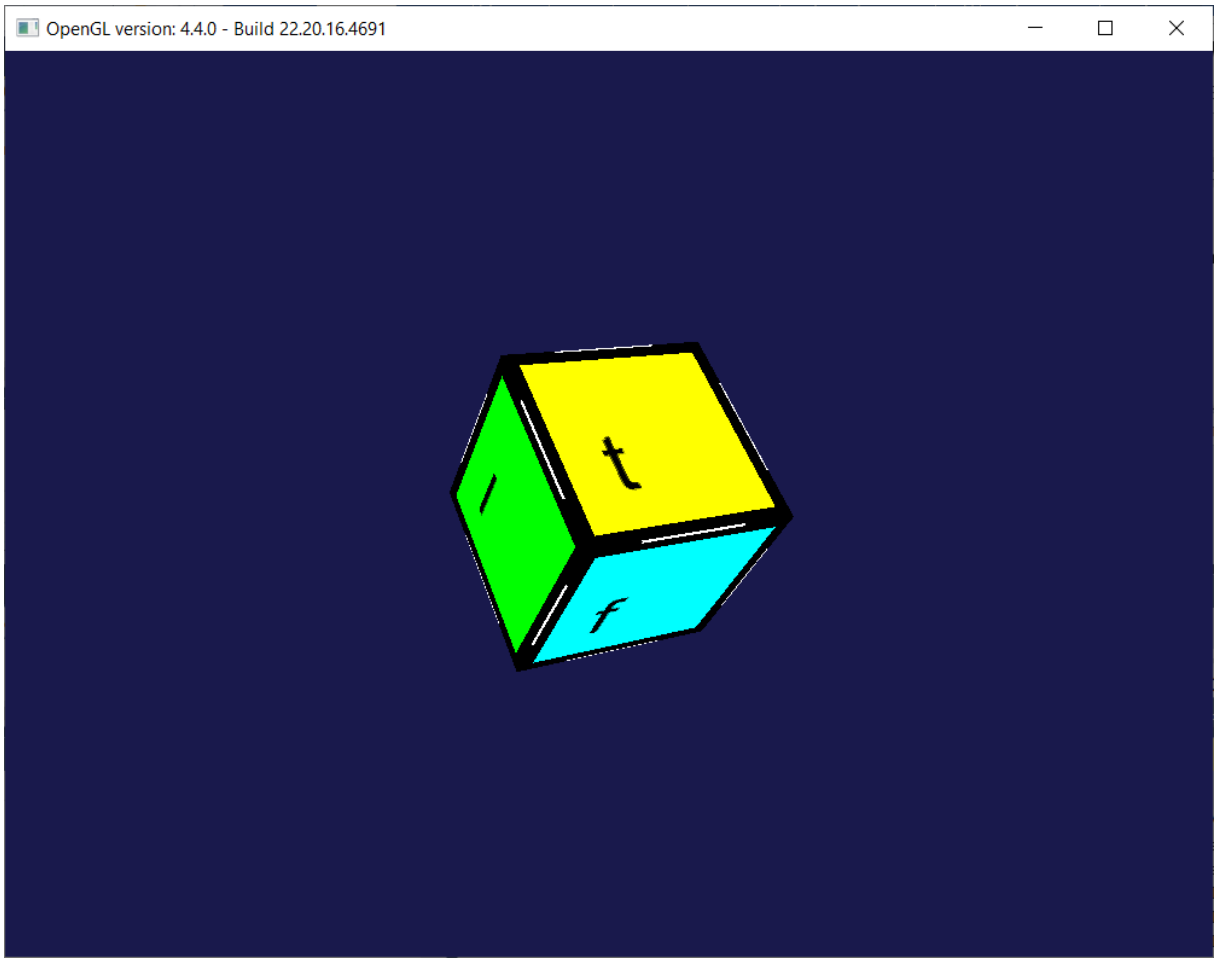

Рис. 5.3. Результат запуска.

5.1. Задания на лабораторную работу

Разработать программу для визуализации указанного трехмерного тела средствами библиотеки OpenGL. На трехмерное тело наложить текстуру.

Таблица 5.1.

Задание на лабораторную работу № 5.

№ вар-та	Трехмерное тело
1	Сфера, сплошная модель.
2	Цилиндр, сплошная модель.
3	Конус, сплошная модель.
4	Тетраэдр, сплошная модель.
5	Октаэдр, сплошная модель.
6	Икосаэдр, сплошная модель.
7	Додекаэдр, сплошная модель.
8	Кубооктаэдр.

Продолжение таблицы 5.1.

9	Пирамида с квадратным основанием.
10	Пирамида с пятиугольным основанием.
11	Пирамида с шестиугольным основанием.
12	Призма с треугольным основанием.
13	Параллелепипед.
14	Призма с пятиугольным основанием.
15	Призма с шестиугольным основанием.
16	Четырехугольная антипризма.
17	Сфера.
18	Цилиндр.
19	Конус.
20	Тетраэдр.
21	Октаэдр.
22	Икосаэдр.
23	Додекаэдр.
24	Кубооктаэдр.
25	Пирамида с квадратным основанием.
26	Пирамида с пятиугольным основанием.
27	Четырехугольная антипризма.
28	Пирамида с шестиугольным основанием.
29	Призма с треугольным основанием.
30	Параллелепипед.
31	Призма с пятиугольным основанием.
32	Призма с шестиугольным основанием.

Литература

1. Jamie Wong. Цвет: от шестнадцатеричных кодов до глаза [Электронный ресурс] / Jamie Wong. [Пер. с англ.] – Режим доступа: свободный. – URL: https://habr.com/ru/post/353582/ – Загл. с экрана (дата обращения: 01.06.2020).
2. Горьков, А. О цветовых пространствах [Электронный ресурс] / Горьков А. Режим доступа: свободный. – URL: https://habr.com/ru/post/181580/ – Загл. с экрана (дата обращения: 01.06.2020).
3. HSL and HSV [Электронный ресурс] / Режим доступа: свободный. – URL: https://en.wikipedia.org/wiki/HSL_and_HSV – Загл. с экрана (дата обращения: 01.06.2020).
4. CIELAB color space [Электронный ресурс] / Режим доступа: свободный. – URL: https://en.wikipedia.org/wiki/CIELAB_color_space – Загл. с экрана (дата обращения: 01.06.2020).
5. Kevin Gee. Introduction to the Direct3D 11 Graphics Pipeline [Электронный ресурс] / Kevin Gee. Режим доступа: свободный. – URL: https://www.nvidia.com/content/nvision2008/tech_presentations/Game_Developer_Track/NVISION08-Direct3D_11_Overview.pdf – Загл. с экрана (дата обращения: 01.06.2020).
6. Frank D. Luna. Introduction to 3D Game Programming with Directx 11 / Frank D. Luna. – Dulles: MERCURY LEARNING AND INFORMATION LLC, 2012. – 752 p.
7. Allen Sherrod, Wendy Jones. Beginning DirectX 11 Game Programming / Allen Sherrod, Wendy Jones. – Boston: Course Technology, 2012. – 372 p.

Содержание

Введение 3
1. Лабораторная работа № 1. Работа с цветовыми пространствами 3
   1.1. Задания на лабораторую работу 6
2. Лабораторная работа № 2. Визуализация простейшей сцены с трехмерным объектом средствами библиотеки DirectX 8
   2.1. Задания на лабораторую работу 40
3. Лабораторная работа № 3. Визуализация простейшей сцены с трехмерным текстурированным объектом средствами библиотеки DirectX 42
   3.1. Задания на лабораторую работу 54
4. Лабораторная работа № 4. Визуализация простейшей сцены с трехмерным объектом средствами библиотеки OpenGL 55
   4.1. Задания на лабораторую работу 69
5. Лабораторная работа № 5. Визуализация простейшей сцены с трехмерным текстурированным объектом средствами библиотеки OpenGL 71
   5.1. Задания на лабораторную работу 77
Литература 79