**Московский авиационный институт**

**(Национальный исследовательский университет)**

Институт: «Информационные технологии и прикладная математика»

Кафедра: 806 «Вычислительная математика и программирование»

Дисциплина: «Компьютерная графика»

**Лабораторная работа № 6**

Тема: Создание шейдерных анимационных эффектов в OpenGL

Студент: Шиляева Н. С.

Группа: 80-304

Преподаватель: Чернышов Л.Н.

Дата:

Оценка:

Москва, 2020

1. Постановка задачи

Для поверхности, созданной в л.р. №5, обеспечить выполнение следующего шейдерного эффекта:

Анимация. Вращение относительно оси OY.

1. Решение задачи

Язык программирования: C#.

Выбранный язык удобен для решения задачи, так как он обладает удобными библиотеками для построения фигур и вывода их на экран.

OpenGL (Open Graphics Library) - спецификация, определяющая платформонезависимый программный интерфейс для написания приложений, использующих двумерную и трехмерную компьютерную графику.

В классе OpenGL так же объявлены константы из библиотеки OpenGL. Их имена идентичны оригинальным, но в силу специфики языка C# для обращения к ним необходимо указывать имя типа. Например, следующий вызов функции glEnable(GL\_BLEND) в нашем случае сводится к вызову метода Enable(OpenGL.GL\_BLEND). Само создание объекта типа OpenGL осуществляется при создании устройства вывода (класс OGL Device) и доступ к нему можно получить при помощи свойства gl данного объекта (RenderDevice) или объекта типа OGLDeviceUpdateArgs передаваемого в качестве параметра методу OnDevice Update(). Данный метод, как и сама работа с устройством OpenGL реализуются в параллельном потоке. Обращение к устройству OpenGL из другого потока не допускается (создание многопоточного рендера возможно, но это достаточно специфическая архитектура, например рендеринг частей экрана в текстуры а потом их объединение). Для большинства функций библиотеки OpenGL при отладке DEBUG конфигурации осуществляется проверка ошибок выполнения и их вывод в окно вывода Microsoft Visual Studio. Поэтому при отладке и написании кода связанного с OpenGL необходимо также контролировать ошибки библиотеки OpenGL в окне вывода.

Шейдер (Shader) — программа для процессора графической карты (GPU), управляющая поведением шейдерной стадии графического конвейера и занимающаяся обработкой соответствующих входных данных. Шейдеры делятся на три типа: Vertex shader (вершинный шейдер) — программа для вершинной шейдерной стадии конвейера, обрабатывающая вершинные данные. Вершинный шейдер может описывать, например, трансформацию вершин из объектного пространства в пространство камеры. Вершинный шейдер выполняется отдельно для каждой вершины. Geometry shader (геометрический шейдер) — шейдерные программы, выполняющиеся в конвейерной стадии геометрического шейдера. Геометрический шейдер работает с вершинными данными, но выполняется сразу для элемента геометрии, например, для треугольника, то есть на вход подаются три вершины. Кроме этих трёх вершин, возможно использование вспомогательных вершин (т.н. adjacent vertices). Геометрические шейдеры способны создавать новую геометрию, могут использоваться для создания частиц, изменения детализации модели «на лету», создание силуэтов и т.д. Fragment shader (пиксельный шейдер) — программа для пиксельной шейдерной стадии конвейера. Пиксельный шейдер выполняется для каждого фрагмента растеризованной геометрии. Обычно пиксельный шейдер занимается закраской геометрического объекта — наложение текстур, освещение, и наложение разных текстурных эффектов, таких как отражение, преломление, туман, Bump Mapping и пр. Пиксельные шейдеры также используются для пост-эффектов.

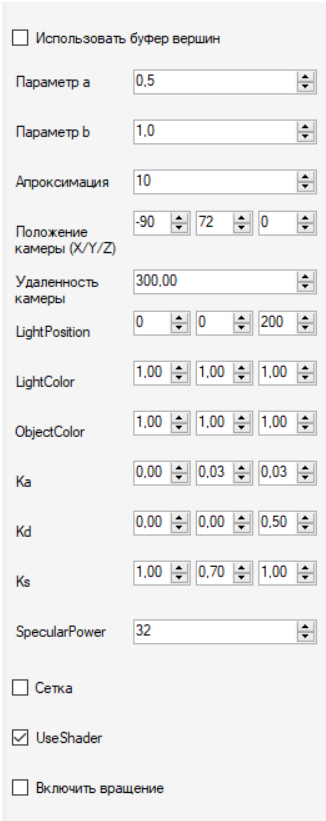
GLSL - высокоуровневый язык используемый для шейдеров OGL. По синтаксису и в целом внешне идентичен C (не С++), основными отличиями является отсутствие указателей, рекурсии и динамического выделения памяти.

Есть два основных типа шейдеров: вершинный и фрагментный. Для вычисления цвета пикселя часто необходим доступ к интерполированным данным вершин. Например, вычисляя влияние освещения на пиксель, нам нужна информация о нормали этого пикселя. Однако в OpenGL нормали являются атрибутами вершин, следовательно доступны лишь вершинному шейдеру. Архитектура графического конвейера OpenGL устроена таким образом, что после обработки всех вершин идет стадия экспорта данных вершин. После чего все обработанные вершины со всеми интерполируемыми значениями размещаются в кэше. Дальше растеризатор из них примитивы. Сборка примитива - это работа с плоскостями отсечения, CULLFACE, определение видимости примитива на экране, возможность отсечения в hierarchical-Z (то есть если растеризатор сможет сразу определить что примитив не виден, то отбросит его), и самое главное - определение пикселей под закраску (то есть выбираются какие пиксели будут закрашиваться этим примитивом). После того, как процессоры z-stencil исключают пиксели, не прошедшие z-stencil тесты, для оставшихся выполняется фрагментный шейдер. Таким образом, при вызове фрагментного шейдера вершины его полигона уже были обработаны, а поступающие значения интерполируются.

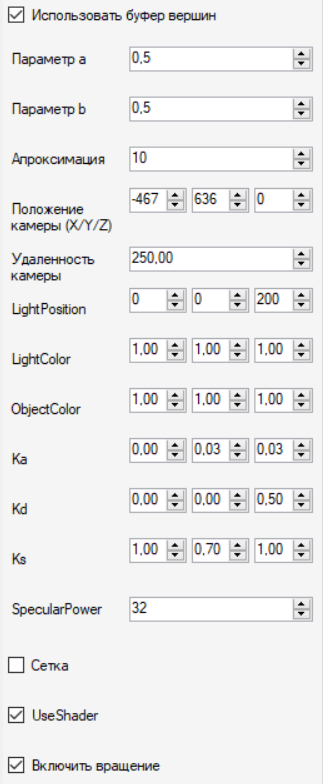
1. Набор тестов

Вводим все необходимые данные прямого эллиптического цилиндра, указывая параметры фигуры и освещения, ставим галочки, где это необходимо, чтобы проверить работу программы:

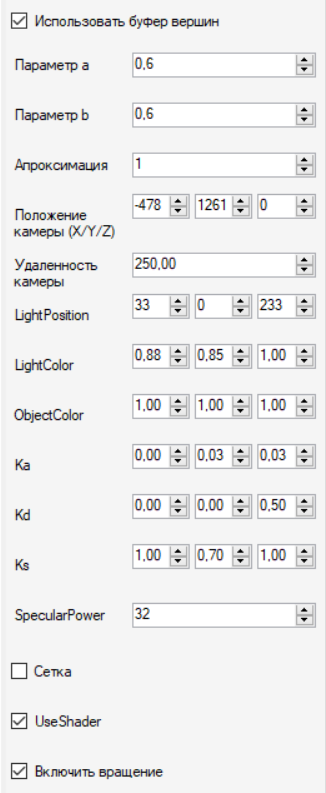
Тест 1:



Тест 2:



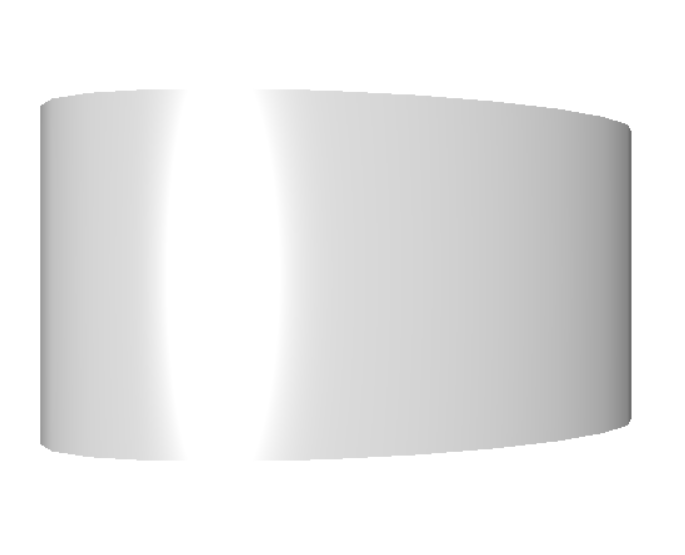
Тест 3:



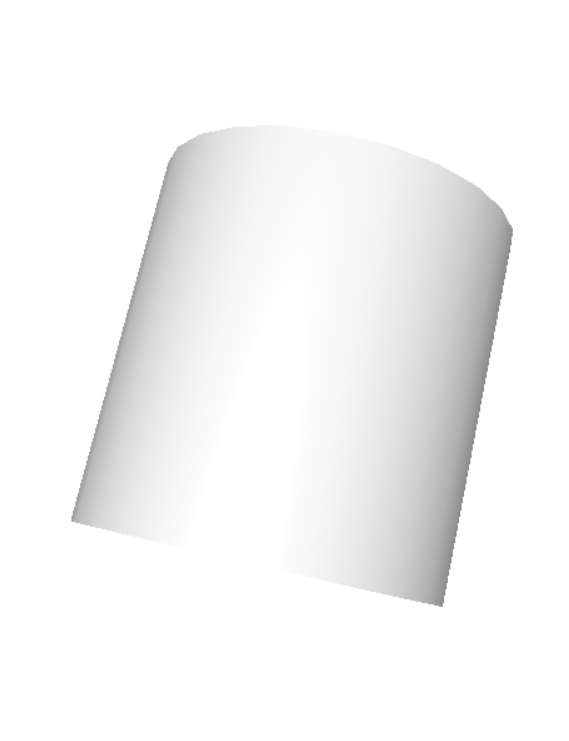
1. Результаты выполнения тестов

Наглядно видим выполнение программы по тестам, описанным ранее.

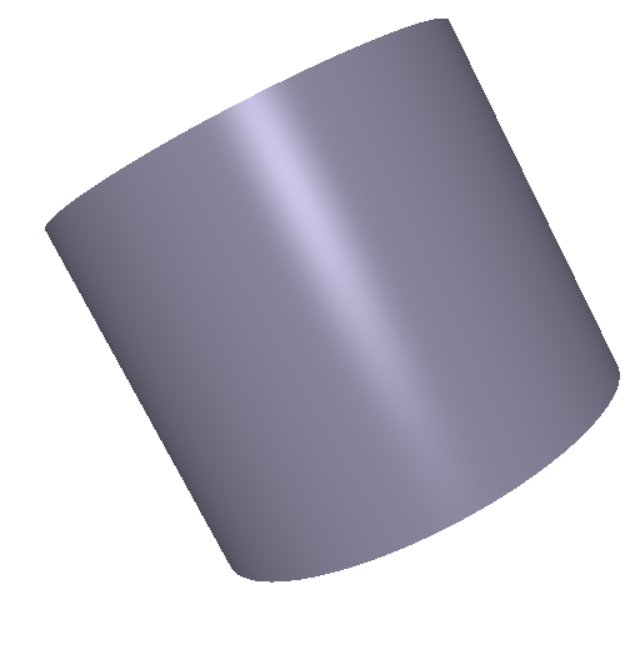
Тест 1:



Тест 2:



Тест 3:



1. Листинг программы

*Файл Program.cs*

using System;

using SharpGL;

using CGLabPlatform;

using System.Runtime.InteropServices;

using System.Windows.Forms;

using System.Diagnostics;

using System.Text;

// Шиляева Наталья; группа: М8О-304Б-18

// Создание и работа с классом приложения аналогична предыдущим примерам, только в

// в данном случае наследуемся от шаблона OGLApplicationTemplate<T>, в силу чего

// для вывода графики будет использоваться элемент управления OGLDevice работающий

// через OpenGL (далее OGL). Код OGLDevice размещается в Controls\OGLDevice.cs

public abstract class CGLabDemoOGL : OGLApplicationTemplate<CGLabDemoOGL>

{

[STAThread]

static void Main() { RunApplication(); }

#region Свойства

[DisplayCheckerProperty(false, "Использовать буфер вершин")]

public virtual bool useVBO { get; set; }

//[DisplayNumericProperty(100, 1, "Высота", 1)]

//public virtual double H { get; set; }

[DisplayNumericProperty(0.5, 0.1, "Параметр a", 0.1)]

public virtual double a { get; set; }

[DisplayNumericProperty(1, 0.1, "Параметр b", 0.1)]

public virtual double b { get; set; }

public double Height = 100; // высота цилиндра

[DisplayNumericProperty(10, 1, "Апроксимация", 1, 17)]

public abstract int Aprox { get; set; }

[DisplayNumericProperty(new[] { -90d, 72d, 0d }, 1, 0, "Положение камеры (X/Y/Z)")]

public virtual DVector3 cameraAngle

{

get { return Get<DVector3>(); }

set { if (Set(value)) UpdateModelViewMatrix(); }

}

[DisplayNumericProperty(300d, 0.1, 2, "Удаленность камеры")]

public virtual double cameraDistance

{

get { return Get<double>(); }

set { if (Set(value)) UpdateModelViewMatrix(); }

}

[DisplayNumericProperty(new[] { 0d, 0d, 200d }, 1, "LightPosition")]

public DVector3 LightPosition { get; set; }

[DisplayNumericProperty(new[] { 1d, 1d, 1d }, 0.01, "LightColor", 0, 1)]

public DVector3 LightColor { get; set; }

[DisplayNumericProperty(new[] { 1d, 1d, 1d }, 0.01, "ObjectColor", 0, 1)]

public DVector3 ObjectColor { get; set; }

[DisplayNumericProperty(new[] { 0d, 0.03d, 0.03d }, 0.01, "Ka", 0, 1)]

public DVector3 Ka { get; set; }

[DisplayNumericProperty(new[] { 0.0d, 0d, 0.5d }, 0.01, "Kd", 0, 1)]

public DVector3 Kd { get; set; }

[DisplayNumericProperty(new[] { 1d, 0.7d, 1d }, 0.01, "Ks", 0, 1)]

public DVector3 Ks { get; set; }

[DisplayNumericProperty(32, 1, "SpecularPower", 1, 100)]

public abstract int SpecularPower { get; set; }

[DisplayCheckerProperty(false, "Сетка")]

public virtual bool Grid { get; set; }

[DisplayCheckerProperty(true, "UseShader")]

public bool UseShader

{

get

{

return Get<bool>();

}

set

{

if (Set<bool>(value))

RenderDevice.AddScheduleTask((gl, s) => gl.UseProgram(value ? program : 0));

}

}

#endregion

// Само создание объекта типа OpenGL осуществляется при создании устройства вывода (класс OGLDevice)

// и доступ к нему можно получить при помощи свойства gl данного объекта (RenderDevice) или объекта

// типа OGLDeviceUpdateArgs передаваемого в качестве параметра методу OnDeviceUpdate(). Данный метод,

// как и сама работа с устройством OpenGL реализуются в параллельном потоке. Обращение к устройству

// OpenGL из другого потока не допускается (создание многопоточного рендера возможно, но это достаточно

// специфическая архитектура, например рендеринг частей экрана в текстуры а потом их объединение).

// Для большинства функций библиотеки OpenGL при отладке DEBUG конфигурации осуществляется проверка

// ошибок выполнения и их вывод в окно вывода Microsoft Visual Studio. Поэтому при отладке и написании

// кода связанного с OpenGL необходимо также контролировать ошибки библиотеки OpenGL в окне вывода.

uint program;

public uint[] attrib\_loc = new uint[2];

public uint[] uniform\_loc = new uint[3];

public uint[] buffers = new uint[2];

protected override void OnMainWindowLoad(object sender, EventArgs args)

{

base.VSPanelWidth = 260;

base.ValueStorage.RightColWidth = 60;

base.RenderDevice.VSync = 1;

#region Обработчики событий мыши и клавиатуры -------------------------------------------------------

RenderDevice.MouseWheel += (s, e) => cameraDistance += e.Delta / 1000.0;

#endregion

// Вся работа связанная с OGL должна выполнятся в одном потоке. Тут работа с OGL

// Метод OnMainWindowLoad() является событием возбуждаемым потоком

// пользовательского интерфейса (UI). Весь код ниже добавляется в диспетчер устройства

// вывода (метод AddScheduleTask() объекта RenderDevice) и выполняется несинхронно в контексте потока

// OGL. Сам диспетчер является очередью типа FIFO (First In First Out - т.е. задания обрабатываются

// строго в порядке их поступления) и гарантирует, что все задания добавленные в OnMainWindowLoad()

// будут выполнены до первого вызова метода OnDeviceUpdate() (aka OnPaint)

#region Инициализация OGL и параметров рендера -----------------------------------------------------

RenderDevice.AddScheduleTask((gl, s) =>

{

//gl.Disable(OpenGL.GL\_DEPTH\_TEST);

gl.Enable(OpenGL.GL\_DEPTH\_TEST);

gl.Enable(OpenGL.GL\_CULL\_FACE);

gl.CullFace(OpenGL.GL\_BACK); // отбросить полигоны, повернутые изнанкой

gl.ClearColor(1, 1, 1, 1);

gl.FrontFace(OpenGL.GL\_CW); // обход вершин по часовой стрелки

gl.PolygonMode(OpenGL.GL\_FRONT, OpenGL.GL\_LINE);

gl.PolygonMode(OpenGL.GL\_BACK, OpenGL.GL\_FILL);

});

#endregion

#region Загрузка и компиляция шейдера

var errorhandler = new Action<string, object, object>((format, arg0, arg1) =>

{

string errormessage = String.Format(format, arg0, arg1);

Trace.WriteLine(errormessage);

throw new Exception(errormessage);

Application.Exit();

});

RenderDevice.AddScheduleTask((gl, s) =>

{

var parameters = new int[1];

uint uit\_ver = gl.CreateShader(OpenGL.GL\_VERTEX\_SHADER);

uint uit\_frag = gl.CreateShader(OpenGL.GL\_FRAGMENT\_SHADER);

var compile\_shader = new Action<uint>(shader =>

{

gl.CompileShader(shader);

// Проверяем была ли компиляция выполнена успешно

gl.GetShader(shader, OpenGL.GL\_COMPILE\_STATUS, parameters);

if (parameters[0] != OpenGL.GL\_TRUE)

{

// В случае если компиляция не удалась пытаемся добиться от OGL

// что именно ему не понравилось. Для этого вначале получаем длину

// сообщения, выделим память под него, а затем уже запрашиваем

// скопировать туда само сообщение. В случае C# это выглядит

// немного иначе, но суть таже.

gl.GetShader(shader, OpenGL.GL\_INFO\_LOG\_LENGTH, parameters);

StringBuilder strbuilder = new StringBuilder(parameters[0]);

gl.GetShaderInfoLog(shader, parameters[0], IntPtr.Zero, strbuilder);

errorhandler("OpenGL Error: ошибка во время компиляции {1}.\n{0}",

strbuilder.ToString(), shader == uit\_ver ? "VERTEX\_SHADER"

: shader == uit\_frag ? "FRAGMENT\_SHADER" : "??????????\_SHADER");

}

});

var shader\_vert = HelpUtils.GetTextFileFromRes("shader.vert");

var shader\_frag = HelpUtils.GetTextFileFromRes("shader.frag");

//return;

gl.ShaderSource(uit\_ver, shader\_vert);

gl.ShaderSource(uit\_frag, shader\_frag);

compile\_shader(uit\_ver);

compile\_shader(uit\_frag);

program = gl.CreateProgram();

gl.AttachShader(program, uit\_ver);

gl.AttachShader(program, uit\_frag);

//gl.LinkProgram(program);

gl.LinkProgram(program);

gl.GetProgram(program, OpenGL.GL\_LINK\_STATUS, parameters);

if (parameters[0] != OpenGL.GL\_TRUE)

errorhandler("OpenGL Error: ошибка линковки програмы шейдера", null, null);

attrib\_loc[0] = (uint)gl.GetAttribLocation(program, "position");

attrib\_loc[1] = (uint)gl.GetAttribLocation(program, "normal");

uniform\_loc[0] = (uint)gl.GetUniformLocation(program, "projection");

uniform\_loc[1] = (uint)gl.GetUniformLocation(program, "view");

uniform\_loc[2] = (uint)gl.GetUniformLocation(program, "model");

gl.UseProgram(program);

});

#endregion

#region Инициализация буфера вершин -----------------------------------------------------------------

RenderDevice.AddScheduleTask((gl, s) =>

{

gl.EnableClientState(OpenGL.GL\_VERTEX\_ARRAY);

gl.EnableClientState(OpenGL.GL\_NORMAL\_ARRAY);

gl.EnableClientState(OpenGL.GL\_INDEX\_ARRAY);

gl.EnableClientState(OpenGL.GL\_COLOR\_ARRAY);

gl.GenBuffers(buffers.Length, buffers);

}, this);

#endregion

#region Уничтожение буфера вершин по завершению работы OGL ------------------------------------------

RenderDevice.Closed += (s, e) => // Событие выполняется в контексте потока OGL при завершении работы

{

var gl = e.gl;

gl.DisableClientState(OpenGL.GL\_VERTEX\_ARRAY);

gl.DisableClientState(OpenGL.GL\_NORMAL\_ARRAY);

gl.DisableClientState(OpenGL.GL\_INDEX\_ARRAY);

gl.DisableClientState(OpenGL.GL\_COLOR\_ARRAY);

};

#endregion

#region Обновление матрицы проекции при изменении размеров окна и запуске приложения ----------------

RenderDevice.Resized += (s, e) =>

{

var gl = e.gl;

gl.MatrixMode(OpenGL.GL\_PROJECTION);

pMatrix = Perspective(60, (double)e.Width / e.Height, 0.1, 100);

gl.LoadMatrix(pMatrix.ToArray(true));

};

#endregion

}

public DMatrix4 mMatrix;

public DMatrix4 vMatrix;

public DMatrix4 pMatrix;

[DisplayCheckerProperty(false, "Включить вращение")]

public bool EnableRot

{

get { return \_EnableRot; }

set

{

\_EnableRot = value;

base.OnPropertyChanged();

}

}

private bool \_EnableRot;

private void UpdateModelViewMatrix() // метод вызывается при измении свойств cameraAngle и cameraDistance

{

#region Обновление объектно-видовой матрицы ---------------------------------------------------------

RenderDevice.AddScheduleTask((gl, s) =>

{

gl.MatrixMode(OpenGL.GL\_MODELVIEW);

var deg2rad = Math.PI / 180; // Вращается камера, а не сам объект

double phi = deg2rad \* cameraAngle.X;

double teta = deg2rad \* cameraAngle.Y;

double psi = deg2rad \* cameraAngle.Z;

// матрицы поворота вокруг осей

DMatrix3 RX = new DMatrix3(1, 0, 0,

0, Math.Cos(phi), -Math.Sin(phi),

0, Math.Sin(phi), Math.Cos(phi));

DMatrix3 RY = new DMatrix3(Math.Cos(teta), 0, Math.Sin(teta),

0, 1, 0,

-Math.Sin(teta), 0, Math.Cos(teta));

DMatrix3 RZ = new DMatrix3(Math.Cos(psi), -Math.Sin(psi), 0,

Math.Sin(psi), Math.Cos(psi), 0,

0, 0, 1);

var cameraTransform = (RX \* RY) \* RZ;

var cameraPosition = cameraTransform \* new DVector3(0, 0, cameraDistance);

var cameraUpDirection = cameraTransform \* new DVector3(0, 1, 0);

// Мировая матрица (преобразование локальной системы координат в мировую)

mMatrix = DMatrix4.Identity; // нет никаких преобразований над объекта

// Видовая матрица (переход из мировой системы координат к системе координат камеры)

vMatrix = LookAt(DMatrix4.Identity, cameraPosition, DVector3.Zero, cameraUpDirection);

// матрица ModelView

var mvMatrix = vMatrix \* mMatrix;

gl.LoadMatrix(mvMatrix.ToArray(true));

//gl.Rotate(45, 1f, 0f, 0);

//gl.Rotate(-45, 0f, 1f, 0);

});

#endregion

}

{

var radians = (verticalAngle / 2) \* Math.PI / 180;

var sine = Math.Sin(radians);

if (nearPlane == farPlane || aspectRatio == 0 || sine == 0)

return DMatrix4.Zero;

var cotan = Math.Cos(radians) / sine;

var clip = farPlane - nearPlane;

return new DMatrix4(

cotan / aspectRatio, 0, 0, 0,

0, cotan, 0, 0,

0, 0, -(nearPlane + farPlane) / clip, -(2.0 \* nearPlane \* farPlane) / clip,

0, 0, -1.0, 1.0

);

}

private static DMatrix4 LookAt(DMatrix4 matrix, DVector3 eye, DVector3 center, DVector3 up)

{

var forward = (center - eye).Normalized();

if (forward.ApproxEqual(DVector3.Zero, 0.00001))

return matrix;

var side = (forward \* up).Normalized();

var upVector = side \* forward;

var result = matrix \* new DMatrix4(

+side.X, +side.Y, +side.Z, 0,

+upVector.X, +upVector.Y, +upVector.Z, 0,

-forward.X, -forward.Y, -forward.Z, 0,

0, 0, 0, 1

);

result.M14 -= result.M11 \* eye.X + result.M12 \* eye.Y + result.M13 \* eye.Z;

result.M24 -= result.M21 \* eye.X + result.M22 \* eye.Y + result.M23 \* eye.Z;

result.M34 -= result.M31 \* eye.X + result.M32 \* eye.Y + result.M33 \* eye.Z;

result.M44 -= result.M41 \* eye.X + result.M42 \* eye.Y + result.M43 \* eye.Z;

return result;

}

#region Count Figure

public int [] numbers = new int[60];

public void Count\_Aprox()

{

for (int i = 1, j = 0; i < 100; i++)

{

if (j >= numbers.Length) break;

if ((360 % i) == 0)

{

numbers[j] = i;

j++;

}

}

}

[StructLayout(LayoutKind.Sequential, Pack = 1)]

public struct Vertex

{

// Координата

public readonly float vx, vy, vz;

// Нормаль

public readonly float nx, ny, nz;

// Цвет

public readonly float r, g, b;

public Vertex(float vx, float vy, float vz, float nx, float ny, float nz, float r, float g, float b)

{

this.vx = vx; this.vy = vy; this.vz = vz;

this.nx = nx; this.ny = ny; this.nz = nz;

this.r = r; this.g = g; this.b = b;

}

}

public Vertex[] vertices; // массив вершин

public int SIZE = 0; // размер массива вершин

public void Count\_Vertex() // получение размера массива (расчёт кол-ва вершин) и его создание

{

SIZE = 0;

//////////////////////// вершины верхней крышки

for (int j = 1; j < Height; j += numbers[Aprox], TopCircles++)

{

CntVertInTopCircles = 0;

for (int phi = 0; phi <= 360; phi += numbers[Aprox], CntVertInTopCircles++)

{

SIZE++;

}

}

///////////////////////////////////////////////////////

for (int j = 1; j < Height; j += numbers[Aprox], BottomCircles++)

{

CntVertInBottomCircles = 0;

for (int phi = 0; phi <= 360; phi += numbers[Aprox], CntVertInBottomCircles++)

{

SIZE++;

}

}

//////////////////////////////////////////////////боковая поверхность

//////////////////////////////////////////////////боковая поверхность

int cnt\_step = (int)Height / numbers[Aprox];

double step = Height / (double)cnt\_step;

for (int j = 0; j <= cnt\_step; j++)

{

for (int i = 0, phi = 0; i < CntVertInBottomCircles - 1; i++, phi += numbers[Aprox])

{

SIZE++;

}

}

SIZE += 2;

vertices = new Vertex[SIZE];

}

public int FIRST = 0, SECOND = 0, LAST = 0;

public int CntVertInCircle = 0;

public int Circles = 0;

public int CntVertInTopCircles = 0;

public int TopCircles = 0;

public int CntVertInBottomCircles = 0;

public int BottomCircles = 0;

public void Create\_Vertex()

{

CntVertInCircle = 0;

Circles = 0;

CntVertInTopCircles = 0;

TopCircles = 0;

CntVertInBottomCircles = 0;

BottomCircles = 0;

double X, Y, Z;

int iterator = 0;

//////////////////////// вершины верхней крышки

for (int j = 1; j < Height; j += numbers[Aprox], TopCircles++)

{

CntVertInTopCircles = 0;

for (int phi = 0; phi <= 360; phi += numbers[Aprox], CntVertInTopCircles++)

{

if ((360 - phi) < numbers[Aprox])

{

X = a \* (Height - j);

Y = 0;

//X \*= Math.Cos(cur\_time);

//Y \*= Math.Sin(cur\_time);

}

else

{

X = a \* (Height - j) \* Math.Cos(phi \* Math.PI / 180);

Y = b \* (-1) \* (Height - j) \* Math.Sin(phi \* Math.PI / 180);

//X \*= Math.Cos(cur\_time);

//Y \*= Math.Sin(cur\_time);

}

Z = (Height / 2);

iterator++;

}

}

///////////////////////////////////////////////////////

for (int j = 1; j < Height; j += numbers[Aprox], BottomCircles++)

{

CntVertInBottomCircles = 0;

for (int phi = 0; phi <= 360; phi += numbers[Aprox], CntVertInBottomCircles++)

{

if ((360 - phi) < numbers[Aprox])

{

X = a \* (Height - j);

Y = 0;

Z = ((-Height / 2d));

//X \*= Math.Cos(cur\_time);

//Y \*= Math.Sin(cur\_time);

}

else

{

X = a \* (Height - j) \* Math.Cos(phi \* Math.PI / 180);

Y = b \* (-1) \* (Height - j) \* Math.Sin(phi \* Math.PI / 180);

Z = ((-Height / 2d));

//X \*= Math.Cos(cur\_time);

//Y \*= Math.Sin(cur\_time);

}

iterator++;

}

}

//боковая поверхность

int cnt\_step = (int)Height / numbers[Aprox];

double step = Height / (double)cnt\_step;

for (int j = 0; j <= cnt\_step; j++, Circles++)

{

CntVertInCircle = 0;

{

if (i == 0) FIRST = iterator;

if ((i == 0) && (j < cnt\_step)) SECOND = iterator;

Z = ((-Height / 2.0 + (j) \* step / 1d));

Vertex tmp = vertices[TopCircles \* CntVertInTopCircles + i];

X = tmp.vx; ;

Y = tmp.vy; ;

LAST = iterator;

iterator++;

}

}

iterator++;

}

public int Polygons = 0; // число полигонов

public void Count\_Polygons()

{

Polygons = 0;

//разбивка на полигоны верхней крышки

for (int j = 1; j < TopCircles; j++)

{

for (int i = 0; i < CntVertInTopCircles - 1; i++)

{

Polygons += 3;

}

}

int z = 1;

for (int j = 1; j < TopCircles; j++)

{

for (int i = j \* CntVertInTopCircles; i < (j + 1) \* CntVertInTopCircles - 1; i++)

{

Polygons += 3;

}

z++;

}

for (int i = 0; i < CntVertInTopCircles - 1; i++)

{

Polygons += 3;

}

// разбивка на полигоны нижней крышки

for (int j = 1; j < BottomCircles; j++)

{

for (int i = 0; i < CntVertInBottomCircles - 1; i++)

{

Polygons += 3;

}

}

z = 1;

for (int j = 1; j < BottomCircles; j++)

{

for (int i = j \* CntVertInBottomCircles; i < (j + 1) \* CntVertInBottomCircles - 1; i++)

{

Polygons += 3;

}

z++;

}

for (int i = 0; i < CntVertInBottomCircles - 1; i++)

{

Polygons += 3;

}

//разбивка на полигоны бокового основания

for (int j = 1; j < Circles; j++)

{

for (int i = 0; i < CntVertInCircle - 1; i++)

{

Polygons += 3;

}

}

for (int j = 1; j < Circles; j++)

{

for (int i = 0; i < CntVertInCircle - 1; i++)

{

Polygons += 3;

}

}

for (int j = 1; j < Circles; j++)

{

for (int i = CntVertInCircle - 1; i < CntVertInCircle; i++)

{

Polygons += 3;

}

}

for (int j = 1; j < Circles; j++)

{

for (int i = CntVertInCircle - 1; i < CntVertInCircle; i++)

{

Polygons += 3;

}

}

indices = new int[Polygons];

}

public int[] indices; // массив индексов вершин

public void Create\_Polygons()

{

int iterator = 0;

int visited = 0;

//разбивка на полигоны верхней крышки

for (int j = 1; j < TopCircles; j++)

{

for (int i = 0; i < CntVertInTopCircles - 1; i++)

{

int a = (j - 1) \* CntVertInTopCircles + i + visited;

int b = (j - 1) \* CntVertInTopCircles + i + 1 + visited;

int c = j \* CntVertInTopCircles + i + visited;

indices[iterator] = a; iterator++;

indices[iterator] = b; iterator++;

indices[iterator] = c; iterator++;

Count\_Normal(a, b, c);

}

}

int z = 1;

for (int j = 1; j < TopCircles; j++)

{

for (int i = j \* CntVertInTopCircles; i < (j + 1) \* CntVertInTopCircles - 1; i++)

{

int a = i + visited;

int b = z + visited;

int c = i + 1 + visited;

indices[iterator] = a; iterator++;

indices[iterator] = b; iterator++;

indices[iterator] = c; iterator++;

z++;

Count\_Normal(a, b, c);

}

z++;

}

for (int i = 0; i < CntVertInTopCircles - 1; i++)

{

int a = (TopCircles - 1) \* CntVertInTopCircles + i + visited;

int b = (TopCircles - 1) \* CntVertInTopCircles + i + 1 + visited;

int c = vertices.Length - 2;

indices[iterator] = a; iterator++;

indices[iterator] = b; iterator++;

indices[iterator] = c; iterator++;

Count\_Normal(a, b, c);

}

// разбивка на полигоны нижней крышки

visited += TopCircles \* CntVertInTopCircles;

for (int j = 1; j < BottomCircles; j++)

{

for (int i = 0; i < CntVertInBottomCircles - 1; i++)

{

int a = (j - 1) \* CntVertInBottomCircles + i + visited;

int b = j \* CntVertInBottomCircles + i + visited;

int c = (j - 1) \* CntVertInBottomCircles + i + 1 + visited;

indices[iterator] = a; iterator++;

indices[iterator] = b; iterator++;

indices[iterator] = c; iterator++;

Count\_Normal(a, b, c);

}

}

z = 1;

for (int j = 1; j < BottomCircles; j++)

{

for (int i = j \* CntVertInBottomCircles; i < (j + 1) \* CntVertInBottomCircles - 1; i++)

{

int a = i + visited;

int b = i + 1 + visited;

int c = z + visited;

indices[iterator] = a; iterator++;

indices[iterator] = b; iterator++;

indices[iterator] = c; iterator++;

z++;

Count\_Normal(a, b, c);

}

z++;

}

for (int i = 0; i < CntVertInBottomCircles - 1; i++)

{

int a = (BottomCircles - 1) \* CntVertInBottomCircles + i + visited;

int b = vertices.Length - 1;

int c = (BottomCircles - 1) \* CntVertInBottomCircles + i + 1 + visited;

indices[iterator] = a; iterator++;

indices[iterator] = b; iterator++;

indices[iterator] = c; iterator++;

Count\_Normal(a, b, c);

}

visited += BottomCircles \* CntVertInBottomCircles;

//разбивка на полигоны бокового основания

for (int j = 1; j < Circles; j++)

{

for (int i = 0; i < CntVertInCircle - 1; i++)

{

int a = (j - 1) \* CntVertInCircle + i + visited;

int c = j \* CntVertInCircle + i + visited;

int b = (j - 1) \* CntVertInCircle + i + 1 + visited;

indices[iterator] = a; iterator++;

indices[iterator] = b; iterator++;

indices[iterator] = c; iterator++;

//Count\_Normal(a, b, c);

}

}

for (int j = 1; j < Circles; j++)

{

for (int i = 0; i < CntVertInCircle - 1; i++)

{

int a = (j - 1) \* CntVertInCircle + i + 1 + visited;

int c = j \* CntVertInCircle + i + visited;

int b = j \* CntVertInCircle + i + 1 + visited;

indices[iterator] = a; iterator++;

indices[iterator] = b; iterator++;

indices[iterator] = c; iterator++;

//Count\_Side\_Normal(a, b, c);

}

}

for (int j = 1; j < Circles; j++)

{

for (int i = CntVertInCircle - 1; i < CntVertInCircle; i++)

{

int a = (j - 1) \* CntVertInCircle + visited;

int c = j \* CntVertInCircle + i + visited;

int b = (j) \* CntVertInCircle + visited;

indices[iterator] = a; iterator++;

indices[iterator] = b; iterator++;

indices[iterator] = c; iterator++;

//Count\_Normal(a, b, c);

}

}

for (int j = 1; j < Circles; j++)

{

for (int i = CntVertInCircle - 1; i < CntVertInCircle; i++)

{

int a = (j - 1) \* CntVertInCircle + i + visited;

int c = j \* CntVertInCircle + i + visited;

int b = (j - 1) \* CntVertInCircle + visited;

indices[iterator] = a; iterator++;

indices[iterator] = b; iterator++;

indices[iterator] = c; iterator++;

//Count\_Normal(a, b, c);

}

}

// расчёт нормалей для боковой поверхности

for (int j = 1; j < Circles; j++)

{

for (int i = 0; i < CntVertInCircle; i++)

{

int a = (j - 1) \* CntVertInCircle + i + visited;

int c = j \* CntVertInCircle + i + visited;

int b = (j - 1) \* CntVertInCircle + i + 1 + visited;

Count\_Normal(a, b, c);

//Count\_Side\_Normal(a, b, c); // нормаль = (1,0,0)

}

}

for (int j = 0; j < Circles; j++)

{

for (int i = CntVertInCircle - 1; i < CntVertInCircle; i++)

{

int a = (j - 1) \* CntVertInCircle + i + visited;

int c = j \* CntVertInCircle + i + visited;

int b = (j - 1) \* CntVertInCircle + visited;

//Count\_Normal(a, b, c);

}

}

Vertex tmp = vertices[FIRST];

tmp = vertices[SECOND];

public void Count\_Normal(int a, int b, int c)

{

DVector3 vec1 = new DVector3(vertices[a].vx, vertices[a].vy, vertices[a].vz);

DVector3 vec2 = new DVector3(vertices[b].vx, vertices[b].vy, vertices[b].vz);

DVector3 vec3 = new DVector3(vertices[c].vx, vertices[c].vy, vertices[c].vz);

DVector3 normal = (-1) \* DVector3.CrossProduct(vec1 - vec3, vec2 - vec1);

}

#endregion;

public void Draw\_Axes(OGLDeviceUpdateArgs e)

{

var gl = e.gl;

gl.UseProgram(0);

#region Axes

gl.LoadIdentity();

gl.PushMatrix();

//x - вправо

gl.Color(1f, 0f, 0);

gl.Begin(OpenGL.GL\_LINES);

gl.Vertex(0, 0, 0);

gl.Vertex(1000, 0, 0);

gl.End();

//y - вверх

gl.Color(0, 1f, 0);

gl.Begin(OpenGL.GL\_LINES);

gl.Vertex(0, 0, 0);

gl.Vertex(0, 1000, 0);

gl.End();

//z - на нас

gl.Color(0f, 0f, 1f);

gl.Begin(OpenGL.GL\_LINES);

gl.Vertex(0, 0, 0);

gl.Vertex(0, 0, 2000);

gl.End();

gl.PopMatrix();

UpdateModelViewMatrix();

#endregion;

if(UseShader)

gl.UseProgram(program);

}

public double angleX = 0;

public double angleY = 0;

public double angleZ = 0;

public void Rotation()

{

if (EnableRot)

{

cameraAngle += new DVector3(0, 0.1, 0);

}

}

public float cur\_time = 0;

public void Shader(OGLDeviceUpdateArgs e, int offset\_vx, int offset\_nx, int mode)

{

unsafe

{

var gl = e.gl;

gl.EnableVertexAttribArray(attrib\_loc[0]);

gl.EnableVertexAttribArray(attrib\_loc[1]);

fixed (Vertex\* vrt = vertices)

}

int objectColorLoc = gl.GetUniformLocation(program, "objectColor");

int lightColorLoc = gl.GetUniformLocation(program, "lightColor");

int lightPosLoc = gl.GetUniformLocation(program, "lightPos");

int viewPosLoc = gl.GetUniformLocation(program, "viewPos");

int AmbientPosLoc = gl.GetUniformLocation(program, "ambient");

int DiffusePosLoc = gl.GetUniformLocation(program, "diffuse");

int SpectrPosLoc = gl.GetUniformLocation(program, "spectr");

int PowerLoc = gl.GetUniformLocation(program, "power");

gl.Uniform3(viewPosLoc, (float)cameraAngle.X, (float)cameraAngle.Y, (float)cameraAngle.Z);

gl.Uniform3(lightPosLoc, (float)LightPosition.X, (float)LightPosition.Y, (float)LightPosition.Z);

int intime = gl.GetUniformLocation(program, "time");

cur\_time += 0.005f;

gl.Uniform3(viewPosLoc, (float)LightPosition.X, (float)LightPosition.Y, (float)LightPosition.Z);

gl.Uniform1(PowerLoc, SpecularPower);

gl.UniformMatrix4((int)uniform\_loc[0], 1, false, pMatrix.ToFloatArray(true));

gl.UniformMatrix4((int)uniform\_loc[1], 1, false, vMatrix.ToFloatArray(true));

gl.UniformMatrix4((int)uniform\_loc[2], 1, false, mMatrix.ToFloatArray(true));

}

}

protected unsafe override void OnDeviceUpdate(object s, OGLDeviceUpdateArgs e)

{

var gl = e.gl;

Rotation();

UpdateModelViewMatrix();

// Очищаем буфер экрана и буфер глубины (иначе рисоваться все будет поверх старого)

gl.Clear(OpenGL.GL\_COLOR\_BUFFER\_BIT | OpenGL.GL\_DEPTH\_BUFFER\_BIT | OpenGL.GL\_STENCIL\_BUFFER\_BIT);

Count\_Aprox();

Count\_Vertex();

Create\_Vertex();

Count\_Polygons();

Create\_Polygons();

//Draw\_Axes(e);

int offset\_vx = (int)Marshal.OffsetOf(typeof(Vertex), "vx");

int offset\_nx = (int)Marshal.OffsetOf(typeof(Vertex), "nx");

int offset\_col = (int)Marshal.OffsetOf(typeof(Vertex), "r");

gl.Enable(OpenGL.GL\_NORMALIZE);

// Рендинг сцены реализуется одним из двух методов - VB (Vertex Buffer) или VA (Vertex Array),

// в зависимости от выбранного пользователем режима.

if (!useVBO)

#region Рендинг сцены методом VA (Vertex Array) -----------------------------------------------------

{

fixed (Vertex\* vrt = vertices)

{

gl.VertexPointer(3, OpenGL.GL\_FLOAT, sizeof(Vertex), IntPtr.Add((IntPtr)vrt, offset\_vx));

gl.NormalPointer(OpenGL.GL\_FLOAT, sizeof(Vertex), IntPtr.Add((IntPtr)vrt, offset\_nx));

gl.ColorPointer(3, OpenGL.GL\_FLOAT, sizeof(Vertex), IntPtr.Add((IntPtr)vrt, offset\_col));

}

fixed (int\* ptr = indices)

{

gl.IndexPointer(OpenGL.GL\_INT, 0, (IntPtr)(&ptr[0]));

}

if (UseShader)

{

Shader(e, offset\_vx, offset\_nx, 0);

}

fixed (int\* ptr = indices)

{

gl.DrawElements(OpenGL.GL\_TRIANGLES, indices.Length, (IntPtr)(&ptr[0]));

}

gl.DisableVertexAttribArray(attrib\_loc[0]);

gl.DisableVertexAttribArray(attrib\_loc[1]);

if (Grid)

gl.PolygonMode(OpenGL.GL\_FRONT\_AND\_BACK, OpenGL.GL\_LINE);

else

gl.PolygonMode(OpenGL.GL\_FRONT, OpenGL.GL\_FILL);

}

#endregion

}

}

1. Вывод

В ходе данной лабораторной работы получены основные навыки по методам закраски и отрисовки плоскостей с помощью технологий OpenGL.

Список литературы

1. Шилдт, Герберт. Ш57 С# 4.0: полное руководство. : Пер. с англ. — М. : ООО "И.Д. Вильямс", 2011. — 1056 с.: ил. — Парал. тит. англ. (дата обращения: 14.11.2020).
2. Прайс. C# 7 и .NET Core. Кросс-платформенная разработка для профессионалов, 3-е издание. -М.: Питер, 2018 - 640 с. (дата обращения: 15.11.2020).