Московский Авиационный Институт
(национальный исследовательский университет)

Факультет прикладной математики и физики
Кафедра вычислительной математики и программирования

Лабораторные работы по курсу «Компьютерная графика»

V семестр

Студентка Черыгова Е.Е.
Группа 8О-304Б

Москва, 2017 г.

Компьютерная графика
Лабораторная работа №1
«Работа с матрицами»
Черыгова Лиза, 8О-304Б

Задание

Написать программу, рисующую две трёхмерные кривые.

Требования

1). Дать возможность менять параметры кривой;

2). Дать возможность менять параметр разбиения кривой;

3). Дать возможность удобно вращать фигуру мышью;

4). Фигура должна выводится в центре окна и масштабироваться в зависимости от его размеров;

5). Параметры, которые нельзя использовать для настройки текущей (выбранной) кривой, должны быть скрыты или отключены;

6). Программа не должна лагать, у неё не должна течь память.

Варианты

Первая кривая у всех одинаковая:

$x(t) = A \cdot cos(t)$

$y(t) = B \cdot sin(t)$

$z(t) = C \cdot t$

Вторая кривая:

$x(t) = A \cdot t \cdot cos(t)$

$y(t) = A \cdot t \cdot sin(t)$

$z(t) = cos(sqrt(x \cdot x + y \cdot y))$

Теоретическая часть

Заменим координатную тройку (x, y, z), задающую точку в пространстве, на четверку чисел (x y z 1) или, более общо, на четверку (hx hy hz), $h \neq 0$.

Каждая точка пространства (кроме начальной точки O) может быть задана четверкой одновременно не равных нулю чисел; эта четверка чисел определена однозначно с точностью до общего множителя.

Предложенный переход к новому способу задания точек дает возможность воспользоваться матричной записью и в более сложных, трехмерных задачах. Любое аффинное преобразование в трехмерном пространстве может быть представлено в виде суперпозиции вращений, растяжений, отражений и переносов. Поэтому вполне уместно сначала подробно описать матрицы именно этих преобразований (ясно, что в данном случае порядок матриц должен быть равен четырем).

А. Матрицы вращения в пространстве

Матрица вращения вокруг оси абсцисс на угол φ:

$$[R_x] = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & cos(\varphi) & sin(\varphi) & 0 \\ 0 & -sin(\varphi) & cos(\varphi) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Матрица вращения вокруг оси ординат на угол ψ:

$$[R_y] = \begin{bmatrix} cos(\psi) & 0 & -sin(\psi) & 0 \\ 0 & 1 & 0 & 0 \\ sin(\psi) & 0 & cos(\psi) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Матрица вращения вокруг оси аппликат на угол χ:

$$[R_z] = \begin{bmatrix} cos(\chi) & sin(\chi) & 0 & 0 \\ -sin(\chi) & cos(\chi) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Замечание

Полезно обратить внимание на место знака "-" в каждой из трех приведенных матриц.

Б. Матрица растяжения (сжатия):

$$[D] = \begin{bmatrix} \alpha & 0 & 0 & 0 \\ 0 & \beta & 0 & 0 \\ 0 & 0 & \gamma & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

где $\alpha > 0$ коэффициент растяжения (сжатия) вдоль оси абсцисс;

$\beta > 0$ коэффициент растяжения (сжатия) вдоль оси ординат;

$\gamma > 0$ коэффициент растяжения (сжатия) вдоль оси аппликат).

В. Матрицы отражения

Матрица отражения относительно плоскости xy:

$$[M_z] = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Матрица отражения относительно плоскости yz:

$$[M_x] = \begin{bmatrix} -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Матрица отражения относительно плоскости zx:

$$[M_y] = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

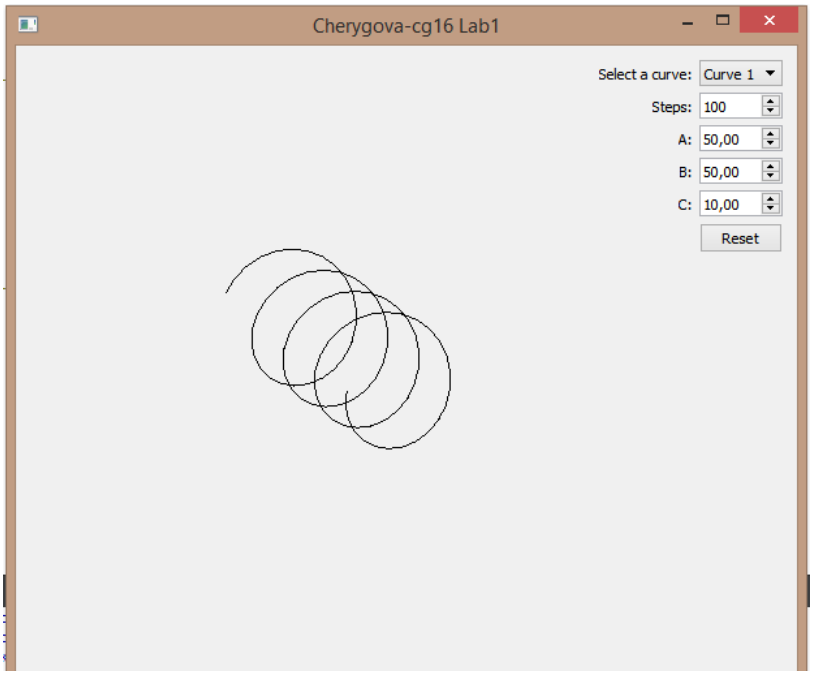
Г. Матрица переноса (здесь (λ, μ, ν) - вектор переноса):

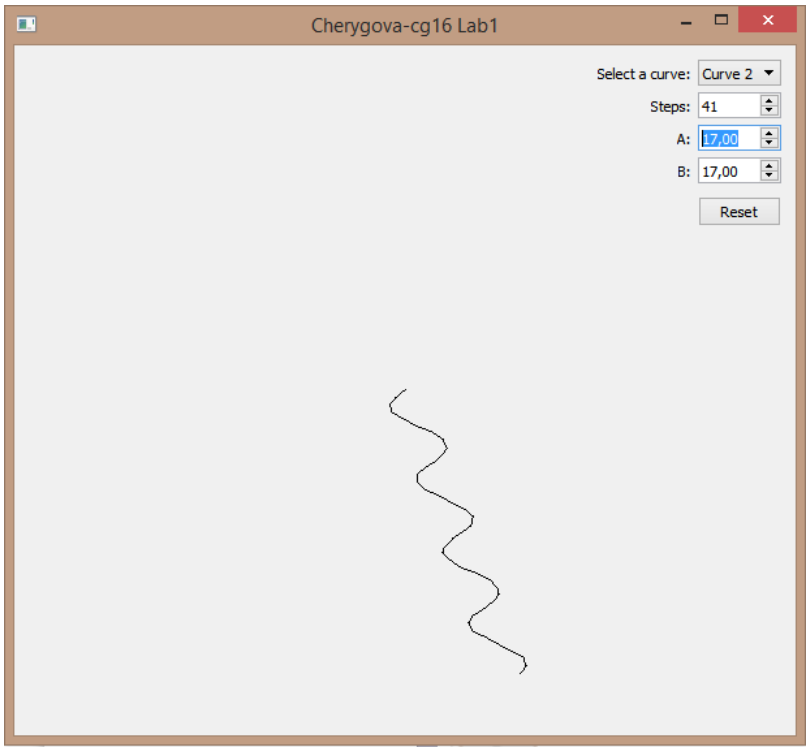
$$[T] = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ \lambda & \mu & \nu & 1 \end{bmatrix}$$

Замечание

Как и в двумерном случае, все выписанные матрицы невырождены.

Скриншоты


Cherygova-cg16 Lab1
Select a curve:
Curve 1
Steps:
100
A:
50,00
B:
50,00
C:
10,00
Reset


Cherygova-cg16 Lab1
Select a curve:
Curve 2
Steps:
41
A:
17,00
B:
17,00
Reset

Практическая часть

matrix4x4.hpp

```
#ifndef MATRIX4X4_H
#define MATRIX4X4_H

#include <vector>
#include <QtMath>

using std::vector;

class Matrix4x4
{
public:
    vector <vector<double>> M;

    Matrix4x4()
    {
        M.resize(4);
        for (int i = 0; i < 4; ++i)
        {
            M[i].resize(4);
            for (int j = 0; j < 4; ++j) M[i][j] = ((i == j) ? 1 : 0);
        }
    }
    Matrix4x4(double* a)
    {
        M.resize(4);
        for (int i = 0; i < 4; ++i)
        {
            M[i].resize(4);
            for (int j = 0; j < 4; ++j) M[i][j] = *(a + 4*i + j);
        }
```

```
}

Matrix4x4& operator =(const Matrix4x4 &N)
{
    for (int i = 0; i < 4; ++i)
    {
        for (int j = 0; j < 4; ++j) this->M[i][j] = N.M[i][j];
    }
    return *this;
}
friend Matrix4x4 operator *(const Matrix4x4& m1, const Matrix4x4& m2)
{
    Matrix4x4 res;
    for (int i = 0; i < 4; ++i)
    {
        for (int j = 0; j < 4; ++j)
        {
            double sum = 0;
            for (int k = 0; k < 4; ++k) sum += m1.M[i][k] * m2.M[k][j];

            res.M[i][j] = sum;
        }
    }
    return res;
}

static Matrix4x4 Translate(double l, double u, double n)
{
    double m[] = {1, 0, 0, 0,
                  0, 1, 0, 0,
                  0, 0, 1, 0,
                  l, u, n, 1};

    return Matrix4x4(m);
}
```

```
static Matrix4x4 Scale2D(double a, double b)
{
    double m[] = {a, 0, 0, 0,
                  0, b, 0, 0,
                  0, 0, 1, 0,
                  0, 0, 0, 1};

    return Matrix4x4(m);
}
static Matrix4x4 RotateX(double phi)
{
    double m[] = {1,    0,        0,      0,
                  0,  cos(phi), sin(phi), 0,
                  0, -sin(phi), cos(phi), 0,
                  0,    0,        0,      1};

    return Matrix4x4(m);
}
static Matrix4x4 RotateY(double psi)
{
    double m[] = {cos(psi), 0, -sin(psi), 0,
                     0,     1,     0,     0,
                  sin(psi), 0,  cos(psi), 0,
                     0,     0,     0,     1};

    return Matrix4x4(m);
}
static Matrix4x4 RotateZ(double hi)
{
    double m[] = { cos(hi), sin(hi),  0,   0,
                  -sin(hi), cos(hi),  0,   0,
                     0,       0,      1,   0,
                     0,       0,      0,   1};

    return Matrix4x4(m);
```

```
    }
};


#endif // MATRIX4X4_H
```

vector3d.hpp

```
#ifndef VECTOR3D_H
#define VECTOR3D_H

#include <vector>
#include <QPointF>
#include "matrix4x4.hpp"

using std::vector;

class Vector3D
{
private:
    vector<double> coords;

public:
    Vector3D()
    {
        coords.resize(3);
    }
    Vector3D(double x, double y, double z)
    {
        coords.resize(3);
        coords[0] = x;
        coords[1] = y;
        coords[2] = z;
    }

    QPointF GetPointF()
    {
        return QPointF(coords[0], coords[1]);
    }
```

```
    Vector3D& operator =(const Vector3D& source)
    {
        this->coords[0] = source.coords[0];
        this->coords[1] = source.coords[1];
        this->coords[2] = source.coords[2];
        return *this;
    }
    Vector3D& operator *(Matrix4x4 m)
    {
        coords[0] = coords[0]*m.M[0][0] + coords[1]*m.M[1][0]
                    + coords[2]*m.M[2][0] + m.M[3][0];
        coords[1] = coords[0]*m.M[0][1] + coords[1]*m.M[1][1]
                    + coords[2]*m.M[2][1] + m.M[3][1];
        coords[2] = coords[0]*m.M[0][2] + coords[1]*m.M[1][2]
                + coords[2]*m.M[2][2] + m.M[3][2];

        double denom = coords[0]*m.M[0][3] + coords[1]*m.M[1][3]
                + coords[2]*m.M[2][3] + m.M[3][3];
        if (denom != 1.0)
        {
            coords[0] /= denom;
            coords[1] /= denom;
            coords[2] /= denom;
        }

        return *this;
    }
};

#endif // VECTOR3D_H
```

curve3d.hpp

```
#ifndef CURVE3D_H
#define CURVE3D_H

#include <vector>
#include <QtMath>
#include "vector3d.hpp"
#include "matrix4x4.hpp"

class Curve3D
{
private:
    double const tn = 2 * M_PI * 4;

public:
    vector <Vector3D> points;
    vector <double> params;

    Matrix4x4 Scaling, RotationX, RotationY, Translation;

    Curve3D() {}
    Curve3D(int NumPoints, int NumParams)
    {
        points.resize(NumPoints);
        params.resize(NumParams);
    }

    Vector3D Curve1(double t)
    {
        double x = params[0] * cos(t);
        double y = params[1] * sin(t);
        double z = params[2] * t;
        return Vector3D(x, y, z);
```

```
}
Vector3D Curve2(double t)
{
    double x = params[0] * sin(t);
    double y = params[1] * t;
    double z = cos(sqrt(x * x + y * y));
    return Vector3D(x, y, z);
}

void SetScaling(double a, double b)
{
    Scaling = Matrix4x4::Scale2D(a, b);
}
void SetRotationX(double phi)
{
    RotationX = Matrix4x4::RotateX(phi);
}
void SetRotationY(double psi)
{
    RotationY = Matrix4x4::RotateY(psi);
}
void SetTranslation(double l, double u, double n)
{
    Translation = Matrix4x4::Translate(l, u, n);
}
Matrix4x4 GetResultMatrix()
{
    return Scaling * RotationX * RotationY * Translation;
}

void FillVector(int CurveID)
{
    double dt = tn / points.size();
    for (uint i = 0; i < points.size(); ++i)
    {
```

```
            double t = dt * i;
            points[i] = CurveID ? Curve2(t) : Curve1(t);
            points[i] = points[i] * GetResultMatrix();
        }
    }
    void SetDefaultParams(int CurveID)
    {
        if (CurveID == 0)
        {
            params[0] = params[1] = 50;
            params[2] = 10;
        }
        if (CurveID == 1) params[0] = params[1] = 10;
    }
};

#endif // CURVE3D_H
```

curvepainter.hpp

```
#ifndef CURVEPAINTER_H
#define CURVEPAINTER_H

#include <QPainter>
#include "matrix4x4.hpp"
#include "curve3d.hpp"

class CurvePainter
{
public:
    vector<Curve3D> Curve;

    CurvePainter() {}

    void AddCurve(int NumPoints, int NumParams)
    {
        Curve.push_back(Curve3D(NumPoints, NumParams));
    }

    void CalculateCurve(int CurveID)
    {
        Curve[CurveID].FillVector(CurveID);
    }
    void PaintCurve(int CurveID, QPainter& Painter)
    {

for (uint i = 0; i < Curve[CurveID].points.size() - 1; ++i)
Painter.drawLine(Curve[CurveID].points[i].GetPointF(),
Curve[CurveID].points[i + 1].GetPointF());

    }
```

```
    double GetCurveParam(int CurveID, int ParamID)
    {
        return Curve[CurveID].params[ParamID];
    }
    void SetCurveParam(int CurveID, int ParamID, double arg)
    {
        Curve[CurveID].params[ParamID] = arg;
    }
};

#endif // CURVEPAINTER_H
```

Вывод

Проведена изначальная настройка рабочего окружения для разработки с использованием фреймворка Qt и IDE QtCreator под Windows. Пришлось основательно вникнуть в систему версий Qt и нюансами её работы с различными версиями компиляторов MS VC++ и MinGW. В результате получена комфортная среда для кроссплатформенной разработки с последними стабильными версиями используемых компонентов.

Непосредственно в процессе выполнения лабораторной познакомилась с особенностями построения графического интерфейса программ, написанных на C++. Не могу не отметить, что использование фреймворка Qt было не настолько удобным, как работа с .NET, однако конечный результат ничуть не уступает по функциональности и обладает большей производительностью.

Также освежила в памяти объектно-ориентированный подход для построения логики программы и познакомилась с использованием матричных вычислений для преобразования графических объектов. Работа с матрицами действительно сильно упрощает изменение (поворот, растяжение, сдвиг) объектов и делает преобразования нагляднее.

Компьютерная графика
Лабораторная работа №2
«Каркасные объекты»
Черыгова Лиза, 8О-304Б

Задание

Написать программу, рисующую проекцию трёхмерного каркасного объекта.

Требования

1). Рисовать грани объекта с помощью доступных функций рисования отрезка в координатах окна. При этом можно использовать OpenGL;

2). При запуске программы объект сразу должно быть хорошо виден;

3). Дать возможность возможность вращать фигуру (2 степени свободы) и изменять параметры фигуры;

4). Не рисовать для выпуклых фигур нелицевые грани;

5). Нарисовать оси системы координат;

6). Параметры, которые нельзя использовать для настройки текущей (выбранной) фигуры, должны быть скрыты или отключены;

7). Программа не должна лагать, у неё не должна течь память.

Варианты

Первые две фигуры у всех одинаковы:

1). N-угольная призма;

2). Поверхность z = A * (x*x + y*y) или параболоид.

Третья фигура - четырехугольная пирамида

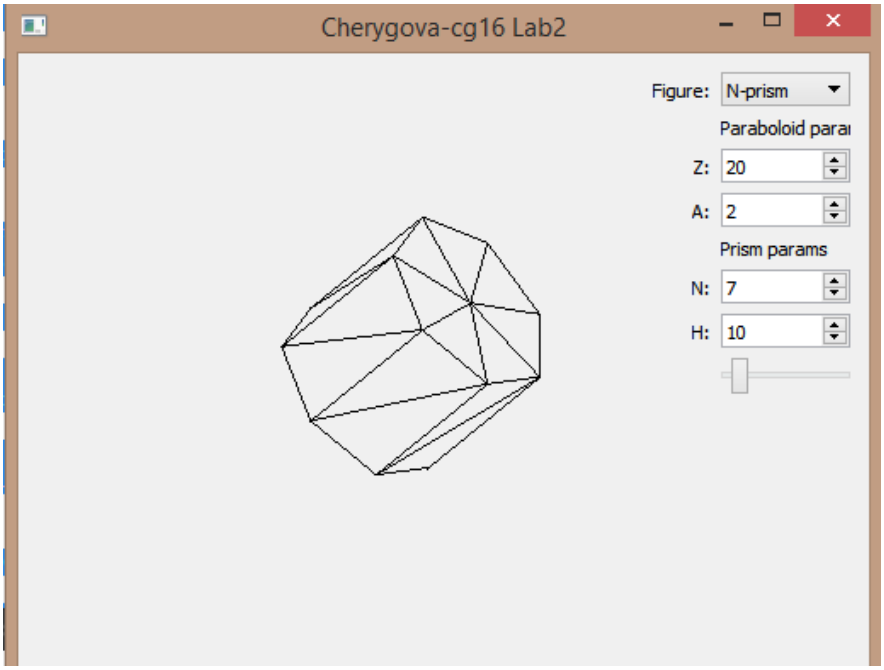
Теоретическая часть

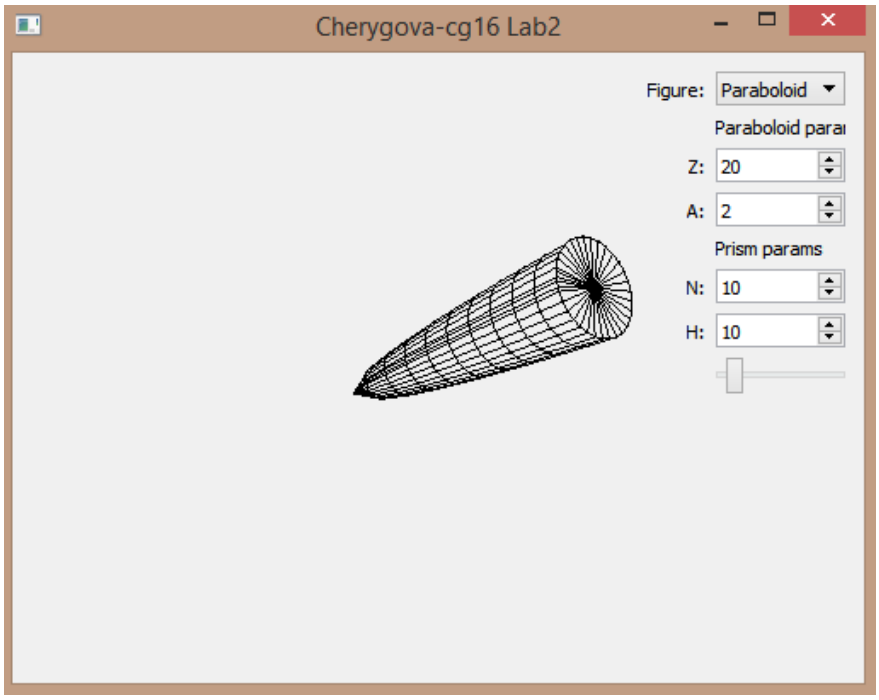
Каркасная модель — модель объекта в трёхмерной графике, представляющая собой совокупность вершин и рёбер, которая определяет форму отображаемого многогранного объекта.

Простейшая модель состоит из списка вершин, где каждой вершине соответствуют некоторые координаты в трёхмерном пространстве и списка отрезков-рёбер, где описана начальная и конечная вершина каждого ребра. В более сложных моделях рёбра могут описываться кривыми, например, кривыми Безье. В данной лабораторной работе грани многоугольников разбиваются на треугольники и четырехугольники. Термин «каркасная модель» происходит из конструкторского моделирования — первые некомпьютерные модели представляли собой каркас предмета, обтянутый тканью или голый «скелет» моделируемого предмета. Сегодня такие модели нередко используются в качестве арт-объектов.

Поскольку для отображения таких моделей требуется относительно немного вычислительных ресурсов, они широко применяются там, где требуется высокая производительность и большое число кадров в секунду, например, в программах для в конструирования трёхмерных объектов, разработки компьютерной графики и т. п. При отображении на двумерном дисплее можно спрятать те рёбра, которые находятся дальше от наблюдателя, закрасить грани. Таким образом, пользователь программы может легко взаимодействовать с моделью: поворачивать её, «видеть насквозь», изменять вершины и рёбра, не прибегая при этом к ресурсоёмкому «реалистичному» рендерингу.

Скриншоты


Cherygova-cg16 Lab2
Figure:
N-prism
Paraboloid para
Z:
20
A:
2
Prism params
N:
7
H:
10


Cherygova-cg16 Lab2
Figure:
Paraboloid
Paraboloid para
Z:
20
A:
2
Prism params
N:
10
H:
10

Практическая часть

body.h

```
#ifndef BODY_H
#define BODY_H

#include <QVector3D>
#include <QVector>
#include <QMatrix4x4>
#include <QPainter>
#include <QtMath>

struct Triangle
{
    QVector3D a, b, c;
    Triangle(QVector3D a = QVector3D(),
            QVector3D b = QVector3D(),
            QVector3D c = QVector3D()): a(a), b(b), c(c){}

    QVector3D normal();
    bool isVisible();
};

struct Nprizm
{
    int n ; //задаваемый параметр
    int level; //координата z
    int r;
    QVector <QVector3D> N_prizm ;

    QVector<QVector3D> return_prizm()
        {
            return N_prizm;
```

```
        }

 void Nprizm_method(int n_v,int level_v,int r_v){
        n = n_v;
        level = level_v;
        r = r_v;
        float da = 2.*M_PI/n;

        for (float phi = -M_PI ; phi < M_PI +da ; phi += da)
        {
QVector3D temp = QVector3D(r*cos(phi),r*sin(phi),level);
N_prizm.push_back(temp);
        }
    }

    QVector3D normal();
    bool isVisible();
};

struct Quadrangle
{
    QVector3D e, b, c, d;
    Quadrangle(QVector3D e = QVector3D(), QVector3D b = QVector3D(),
        QVector3D c = QVector3D(), QVector3D d = QVector3D()):
            e(e), b(b), c(c), d(d){}

    QVector3D normal();
    bool isVisible();
};

Triangle operator*(QMatrix4x4 m, Triangle t);

 Quadrangle operator*(QMatrix4x4 m, Quadrangle q);

struct Body
```

```
{
    QVector<Triangle> faces;
    QVector<Quadrangle> faces_q;
    QVector<Nprizm> prizm;

    Body(){}
    virtual ~Body(){}

    virtual void calculate(){}
    void draw(QPainter *p);
};

Body operator*(QMatrix4x4 m, Body b);

#endif // BODY_H
```

body.cpp

```
#include "body.h"
#include <QDebug>
QVector3D Triangle::normal()
{
    return QVector3D::crossProduct((b - a), (c - a));
}

QVector3D Quadrangle::normal()
{
    return QVector3D::crossProduct((c - b), (d - b));
}

QVector3D Nprizm::normal()
{
    return QVector3D::crossProduct((N_prizm[1] - N_prizm[0]),
                           (N_prizm[2] - N_prizm[0]));
}

bool Nprizm::isVisible(){
    if (this->normal().z()>0)
    {
        return true;
    }
    else return false;

}

bool Triangle::isVisible()
{
    if (this->normal().z()>0){
        return true;
    }
```

```
    else return false;

}

bool Quadrangle::isVisible()
{
    if (this->normal().z()>0){
        return true;
    }
    return false;
}

Triangle operator*(QMatrix4x4 m, Triangle t)
{
    return Triangle(m * t.a, m * t.b, m * t.c);
}

Quadrangle operator *(QMatrix4x4 m, Quadrangle q)
{
    return Quadrangle(m * q.c, m * q.b, m * q.c, m * q.d);
}

Nprizm operator *(QMatrix4x4 m, Nprizm pr)
{
    foreach (QVector3D prizmeach, pr.N_prizm) {
        pr.N_prizm.push_back(m * prizmeach);
    }
    return pr;
}

void Body::draw(QPainter *p)
{
    foreach(Triangle face, faces)
    {
        if(face.isVisible())
```

```
        {
            p->drawLine(face.a.toPointF(), face.b.toPointF());
            p->drawLine(face.b.toPointF(), face.c.toPointF());
            p->drawLine(face.c.toPointF(), face.a.toPointF());
        }
    }

    foreach (Quadrangle face_q, faces_q)
    {
        if(face_q.isVisible())
        {
p->drawLine(face_q.b.toPointF(), face_q.e.toPointF());
p->drawLine(face_q.e.toPointF(), face_q.d.toPointF());
p->drawLine(face_q.d.toPointF(), face_q.c.toPointF());
p->drawLine(face_q.c.toPointF(), face_q.b.toPointF());
        }
    }

    foreach(Nprizm prizmeach,prizm)
    {
        if(prizmeach.isVisible())
        {

for(int i = 0;i < prizmeach.N_prizm.size() - 1;i++)
            {

p->drawLine(prizmeach.N_prizm[i].toPointF(),
    prizmeach.N_prizm[i+1].toPointF());

            }
        }
    }

}
```

```
Body operator*(QMatrix4x4 m, Body b)
{
    Body res;
    foreach(Triangle face, b.faces)
        res.faces.push_back(m * face);

    foreach (Quadrangle face_q, b.faces_q) {
        res.faces_q.push_back(m * face_q);
    }
    foreach (Nprizm prizmeach, b.prizm) {
        res.prizm.push_back(m * prizmeach);
    }
    return res;
}
```

n_prizm.h

```
#ifndef N_PRIZM_H
#define N_PRIZM_H
#include "body.h"

struct N_prizm: public Body

{
    void calculate()
    {
        int n = 7;
        int level = 0;
        int level2 = 10;
        int r = 6;
        float da = 2.*M_PI/n;

        for (float phi = -M_PI ; phi < M_PI +da ; phi += da)
        {

faces.push_back(Triangle(QVector3D(0,0,level),
QVector3D(r*cos(phi+da),r*sin(phi+da),level),
QVector3D(r*cos(phi),r*sin(phi),level)));

faces.push_back(Triangle(QVector3D(0,0,level2),
QVector3D(r*cos(phi),r*sin(phi),level2),
QVector3D(r*cos(phi +da),r*sin(phi+da),level2)));

faces.push_back(Triangle(QVector3D(r*cos(phi),r*sin(phi),level),
QVector3D(r*cos(phi+da),r*sin(phi+da),level),
QVector3D(r*cos(phi),r*sin(phi),level2)));

faces.push_back(Triangle(QVector3D(r*cos(phi+da),r*sin(phi+da),level),
QVector3D(r*cos(phi+da),r*sin(phi+da),level2),
```

```
QVector3D(r*cos(phi),r*sin(phi),level2)));

        }
    }
};

#endif // N_PRIZM_H
```

paraboloid.h

```
#ifndef PARABOLOID_H
#define PARABOLOID_H
#include "body.h"
#include "math.h"
struct Paraboloid: public Body
{
    float z = 20;
    float a = 2.;

    void calculate()
    {
       float steps = 10;

       float dz = z/steps;
       float r;
       float r_kr;
       float r_kr2;
       float da =0.2;
       for(float z_i = 0;z_i < z;z_i+=dz)
       {

           r = sqrt(z_i/a);
           float z_i2 = z_i + dz;
           float r2 = sqrt(z_i2/a);
           r_kr = sqrt((z-dz)/a);
           r_kr2 = sqrt (z/a);

           for (float phi = -M_PI ; phi < M_PI +da ; phi += da)
           {

faces_q.push_back(Quadrangle(QVector3D(r*cos(phi),r*sin(phi),z_i),
            QVector3D(r2*cos(phi),r2*sin(phi),z_i2),
```

```
            QVector3D(r2*cos(phi +da),r2*sin(phi+da),z_i2),
            QVector3D(r*cos(phi+da),r*sin(phi+da),z_i)));

faces.push_back(Triangle(QVector3D(0,0,z),
            QVector3D(r_kr2*cos(phi +da),r_kr2*sin(phi+da),z),
            QVector3D(r_kr2*cos(phi),r_kr2*sin(phi),z)));

           }

       }


    }
};

#endif // PARABOLOID_H
```

piramid.h

```
#ifndef PIRAMID_H
#define PIRAMID_H

#include "body.h"

struct Piramid: public Body
{
    void calculate()
    {
        QVector3D b(0, 6, 0), c(0, 0, 0),
                d(10, 0, 0), e(10, 6, 0), a(5,3,10);
        faces.push_back(Triangle(b,e,d));
        faces.push_back(Triangle(d,c,b));
        faces.push_back(Triangle(b,c,a));
        faces.push_back(Triangle(c,d,a));
        faces.push_back(Triangle(d,e,a));
        faces.push_back(Triangle(e,b,a));
    }
};

#endif // PIRAMID_H
```

tetraedr.hpp

```
#ifndef TETRAEDER_HPP
#define TETRAEDER_HPP

#include "body.h"
struct Tetraeder: public Body
{
    void calculate()
```

```
    {

        QVector3D a(0, 0, 8), b(0, 6, 0),
                c(0, 0, 0), d(10, 0, 0);
        faces.push_back(Triangle(b, c, a));
        faces.push_back(Triangle(c, d, a));
        faces.push_back(Triangle(d, b, a));
        faces.push_back(Triangle(c, b, d));

    }
};

#endif // TETRAEDER_HPP
```

Вывод

Изучила способы построения каркасных объектов и тел вращения с помощью плоских многоугольников и заданной точностью и определения нелицевых граней. Линейная алгебра снова сильно пригодилась. Наиболее трудоёмкими частями были проектирование взаимодействия классов с интерфейсом и написание циклов, в которых выводятся координаты многоугольников.

Компьютерная графика
Лабораторная работа №3
«Закраска граней»
Черыгова Лиза, 8О-304Б

Задание

1). Написать программу, рисующую освещённый трёхмерный объект;

2). Реализовать модель освещения Блинна-Фонга (с halfway-вектором);

3). Реализовать любую другую модель освещения, принцип работы которой отличается от модели Блинна-Фонга (например, Кука-Торренса).

Требования

1). При запуске программы сразу должен быть виден красиво освещенный объект в хорошем ракурсе;

2). Грани объекта рисуются с помощью доступных функций закраски грани в координатах окна;

3). Геометрия объекта должна генерироваться в программе, а не откуда-то считываться;

4). У объекта должны быть изменяемые параметры (см. ЛР2);

5). Ориентация объекта хранится с помощью матрицы. Объект можно вращать в 3 или хотя бы 2 степенях свободы;

6). Параметры, которые нельзя использовать для настройки текущей (выбранной) фигуры, должны быть скрыты или отключены;

7). Программа не должна лагать, у неё не должна течь память.

Требования к освещению

1). Освещение «наклеено» на фигуру, т.е. остается неизменным при вращении «камеры» (но меняется при движении источника/наблюдателя);

2). Положение источника света и наблюдателя должны быть как-то обозначены в пространстве (напр., небольшими кружками);

3). Все важные параметры освещения должны быть изменяемы. Под важными параметрами понимаются:

- Цвет каждой из трёх компонент освещения (фоновой, рассеянной и бликовой);

- Параметры модели освещения (напр., коэффициент гладкости фигуры в модели Кука-Торренса);

- Положение источника и наблюдателя.

Варианты

Первые две фигуры у всех одинаковы:

1). Сфера;

2). Тор.

Третья фигура - четырехугольная пирамида

Теоретическая часть

Пусть заданы точечный источник света, расположенный в некоторой точке, поверхность, которая будет освещаться и наблюдатель. Будем считать, что наблюдатель точечный. Каждая точка поверхности имеет свои координаты и в ней определена нормаль к поверхности. Её освещенность складывается из трех компонент: фоновое освещение (ambient), рассеянный свет (diffuse) и бликовая составляющая (specular). Свойства источника определяют мощность излучения для каждой из этих компонент, а свойства материала поверхности определяют её способность воспринимать каждый вид освещения.

$$I = I_a + I_d + I_s$$

, где

I_a - фоновая составляющая (ambient);

I_d - рассеянная составляющая (diffuse);

I_s - зеркальная составляющая (specular).

Модель Фонга – классическая модель освещения. Модель представляет собой комбинацию диффузной составляющей (модели Ламберта) и зеркальной составляющей и работает таким образом, что кроме равномерного освещения на материале может еще появляться блик. Местонахождение блика на объекте, освещенном по модели Фонга, определяется из закона равенства углов падения и отражения. Если наблюдатель находится вблизи углов отражения, яркость соответствующей точки повышается.

Падающий и отраженный лучи лежат в одной плоскости с нормалью к отражающей поверхности в точке падения, и эта нормаль делит угол между лучами на две равные части. Т.о. отраженная составляющая освещенности в точке зависит от того, насколько близки направления на наблюдателя и отраженного луча.

Фоновое освещение это постоянная в каждой точке величина надбавки к освещению. Вычисляется фоновая составляющая освещения как:

$$I_a = k_a i_a$$

,

где

I_a - фоновая составляющая освещенности в точке,

k_a – свойство материала воспринимать фоновое освещение,

I_a – мощность фонового освещения.

Из формулы выше видно, что фоновая составляющая освещенности не зависит от пространственных координат освещаемой точки и источника. Поэтому при моделировании освещения, в большинстве случае, не имеет смысла брать более одного фонового источника света. Часто просто задается некое глобальное фоновое освещение всей сцены.

2. Рассеянный свет при попадании на поверхность рассеивается равномерно во все стороны. При расчете такого освещения учитывается только ориентация поверхности (нормаль) и направление на источник света. Рассеянная составляющая рассчитывается по закону косинусов (закон Ламберта):

$$I_d = k_d \cos(\vec{L}, \vec{N}) i_d = k_d (\vec{L} \cdot \vec{N}) i_d$$

,

где

I_d – рассеянная составляющая освещенности в точке,

k_d – свойство материала воспринимать рассеянное освещение,

i_d – мощность рассеянного освещения,

$\vec{L}$ – направление из точки на источник,

$\vec{N}$ - вектор нормали в точке.

3. Зеркальный свет при попадании на поверхность подчиняется следующему закону: “Падающий и отраженный лучи лежат в одной плоскости с нормалью к отражающей поверхности в точке падения, и эта нормаль делит угол между лучами на две равные части”. Т.о. отраженная составляющая освещенности в точке зависит от того, насколько близки направления на наблюдателя и отраженного луча.

В общем случае вектора $\vec{V}, \vec{L} u \vec{N}$ не лежат в одной плоскости.

$$I_s = k_s \cos^{\alpha}(\vec{R}, \vec{V}) i_s = k_s (\vec{R} \cdot \vec{V})^{\alpha} i_s$$

,

где

I_s – зеркальная составляющая освещенности в точке,

k_s – коэффициент зеркального отражения,

i_d – мощность зеркального освещения,

$\vec{R}$ – направление отраженного луча,

$\vec{V}$ - направление на наблюдателя,

α - коэффициент блеска, свойство материала.

Именно зеркальное отражение представляет наибольший интерес, но в то же время его расчет требует больших вычислительных затрат. При фиксированном положении поверхности относительно источников света фоновая и рассеянные составляющие освещения могут быть просчитаны единожды для всей сцены, т.к. их значение не зависит от направления взгляда. С зеркальной составляющей этот фокус не сработает и придется пересчитывать её каждый раз, когда взгляд меняет свое направление.

Во всех вычислениях выше, для рассеянной и зеркальной компонент, если скалярное произведение в правой части меньше нуля, то соответствующая компонента освещенности полагается равной нулю.

Упрощенный расчет зеркальной компоненты освещенности. Модель Блинна-Фонга

Для расчета отраженной компоненты требуется выполнить довольно громоздкие вычисления. Существует модель Блинна-Фонга, представляющая собой модель Фонга с упрощенным расчетом зеркального отражения. Вычислим в каждой точке вектор полупути $\vec{H}$ (halfway vector):

$$\vec{H} = \frac{\vec{L} + \vec{V}}{|\vec{L} + \vec{V}|} = (\vec{L} + \vec{V})_{norm}$$

который показывает ориентацию площадки, на которой будет максимальное отражение. Тогда величину $(\vec{R} \cdot \vec{V})^\alpha$ можно заменить величиной $(\vec{H} \cdot \vec{N})^\beta$.

Вектор $\vec{H}$ называется вектором полупути, т.к. если все три вектора $\vec{V}$, $\vec{L}$ и $\vec{N}$ лежат в одной плоскости, то угол между $\vec{H}$ и $\vec{N}$ составляет половину угла между $\vec{R}$ и $\vec{V}$.

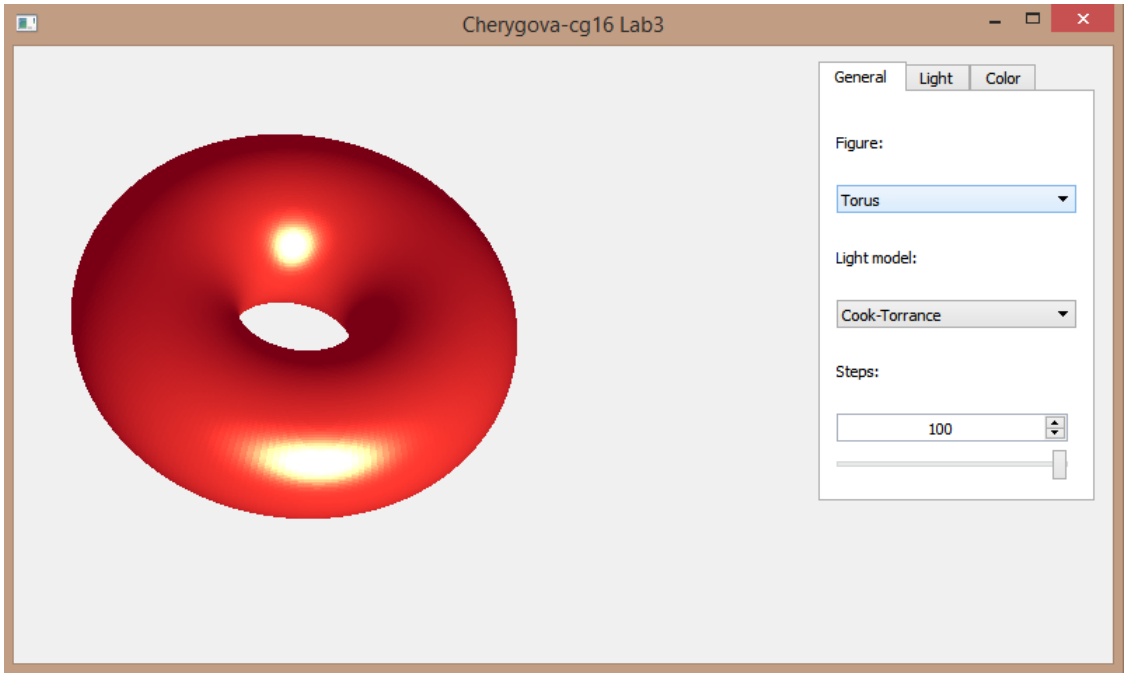
Модель отражения Блинна-Фонга никогда в точности не совпадает с моделью Фонга, однако можно подобрать соответствующие значения α и β, для которых распределения зеркальной составляющей по поверхности для обеих моделей будут очень близкими. Вместе с тем, в ряде случаев модель Блинна-Фонга требует значительно меньше вычислений, например в случае направленного бесконечно-удаленного источника.

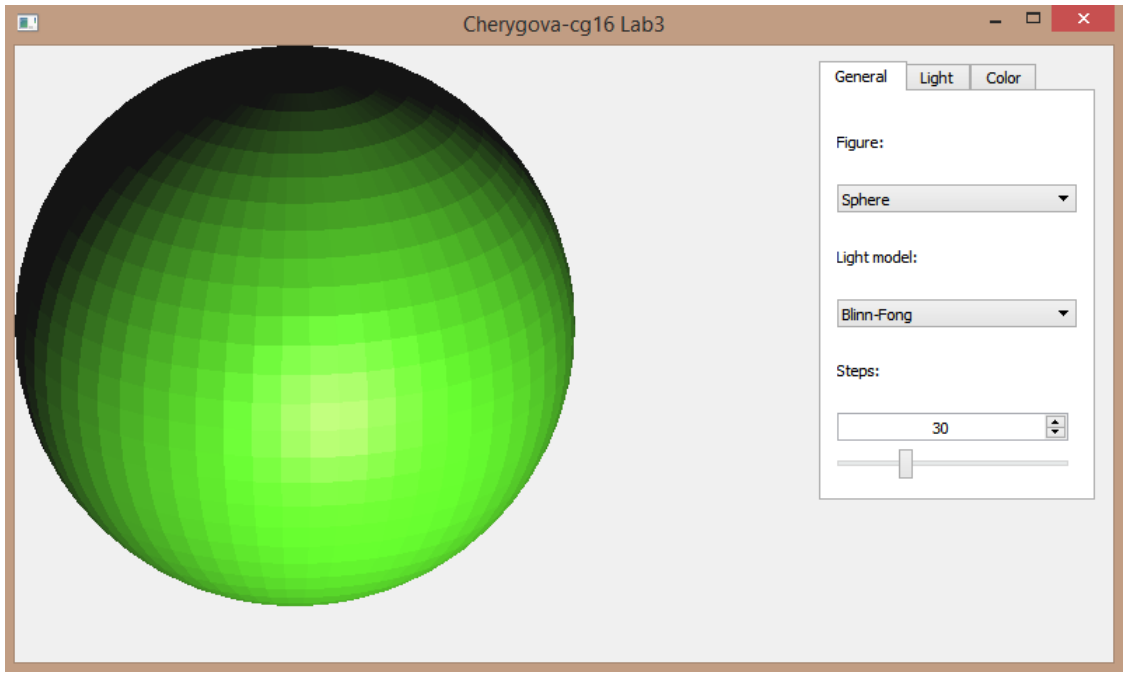
Учет цвета

Если используется цветовая модель RGB, то все расчеты, представленные выше проделываются для каждой компоненты R, G и B по отдельности. Если при этом освещать

поверхность синим рассеянным светом (0.0, 0.0, 1.0), а она воспринимает только красный рассеянный (1.0, 0.0, 0.0), то рассеянная составляющая освещенности во всех точках этой поверхности будет равна (0.0, 0.0, 0.0).

Скриншоты


Cherygova-cg16 Lab3
General
Light
Color
Figure:
Torus
Light model:
Cook-Torrance
Steps:
100


Cherygova-cg16 Lab3
General
Light
Color
Figure:
Sphere
Light model:
Blinn-Fong
Steps:
30

Практическая часть

widget.cpp

```
#include "widget.h"
#include "ui_widget.h"

#include <QMouseEvent>
#include <QPainter>
#include <QVector4D>
#include <QMatrix4x4>
#include <QVector3D>
#include <QMatrix3x3>
#include <QPointF>
#include <algorithm>
#include <cmath>
using std::max;

int steps = 30;
float resX = 1;
float resY = 1;
int check = 1;
int X = 100;
int Y = 200;
int Z = 300;
int Ired1 = 20;
int Ired2 = 90;
int Ired3 = 90;
int Igreen1 = 20;
int Igreen2 = 60;
int Igreen3 = 90;
int Iblue1 = 20;
int Iblue2 = 30;
int Iblue3 = 80;
```

```
int eli = 30; //коэффициент блеска, свойство материала
float m = 0.223; //шероховатость материала
QString sv = "Blinn-Fong";

Widget::Widget(QWidget *parent) :
    QWidget(parent), ui(new Ui::Widget)
{
    ui->setupUi(this);
    ui->StepsSlider->setValue(steps);
    ui->StepsSpinBox->setValue(steps);
    ui->LSourceSpinX->setValue(X);
    ui->LSourceSpinY->setValue(Y);
    ui->LSourceSpinZ->setValue(Z);
    ui->RedSlider1->setValue(Ired1);
    ui->RedSlider2->setValue(Ired2);
    ui->RedSlider3->setValue(Ired3);
    ui->GreenSlider1->setValue(Igreen1);
    ui->GreenSlider2->setValue(Igreen2);
    ui->GreenSlider3->setValue(Igreen3);
    ui->BlueSlider1->setValue(Iblue1);
    ui->BlueSlider2->setValue(Iblue2);
    ui->BlueSlider3->setValue(Iblue3);
    ui->CoeffSlider->setValue(eli);
}

Widget::~Widget()
{
    delete ui;
}

QVector3D crossProduct(QVector4D A, QVector4D B)
{
    return QVector3D::crossProduct(QVector3D(A), QVector3D(B));
}
float ColorOverflowControl(float c)
```

```
{
    c = c < 0 ? 0 : c;
    c = c > 255 ? 255 : c;
    return c;
}

float colorred (QVector3D N, QVector3D L,QVector3D V){
    L.normalize();
    N.normalize();
    V.normalize();
    float diff = Ired1
            + Ired2*(max(0.0F,QVector3D::dotProduct(L,N)))
+ Ired3*pow(max(0.0F,QVector3D::dotProduct(N,(L+V).normalized())),eli);
    return diff;
}
float colorgreen(QVector3D N, QVector3D L,QVector3D V){
    L.normalize();
    N.normalize();
    V.normalize();
    float diff = Igreen1+Igreen2*(max(0.0F,QVector3D::dotProduct(L,N)))
+Igreen3*pow(max(0.0F,QVector3D::dotProduct(N,(L+V).normalized())),eli);
    return diff;
}
float colorblue(QVector3D N, QVector3D L,QVector3D V){
    L.normalize();
    N.normalize();
    V.normalize();
    float diff = Iblue1+Iblue2*(max(0.0F,QVector3D::dotProduct(L,N)))
+Iblue3*pow(max(0.0F,QVector3D::dotProduct(N,(L+V).normalized())),eli);
    return diff;
}

float CookTorrance(QVector3D N, QVector3D L, QVector3D V, float m)
{
    float HN = QVector3D::dotProduct((L+V).normalized(),N);
```

```
    float VN = QVector3D::dotProduct(V,N);
    float LN = QVector3D::dotProduct(L,N);
    float VH = QVector3D::dotProduct(V,(L+V).normalized());
    float G = fmin(1,fmin((2*HN*VN)/VH,(2*HN*LN)/VH));
    float F = 1/(1+VN);
    float Dd = exp((pow((HN),2)-1)/(pow(m,2)*(pow((HN),2))))
                    / (M_PI*pow(m,2)*(pow((HN),4)));

    return (F*G*Dd)/(VN*LN);
}

float K_colorred(QVector3D N, QVector3D L,QVector3D V, float m){
    L.normalize();
    N.normalize();
    V.normalize();
    float diff = Ired1
            + Ired2*(max(0.0F,QVector3D::dotProduct(L,N)))
            + Ired3*max(0.0F,CookTorrance(N,L,V,m));
    return diff;
}
float K_colorgreen(QVector3D N, QVector3D L,QVector3D V, float m){
    L.normalize();
    N.normalize();
    V.normalize();
    float diff = Igreen1+Igreen2*(max(0.0F,QVector3D::dotProduct(L,N)))
            +Igreen3*max(0.0F,CookTorrance(N,L,V,m));
    return diff;
}
float K_colorblue(QVector3D N, QVector3D L,QVector3D V, float m){
    L.normalize();
    N.normalize();
    V.normalize();
    float diff = Iblue1+Iblue2*(max(0.0F,QVector3D::dotProduct(L,N)))
            +Iblue3*max(0.0F,CookTorrance(N,L,V,m));
    return diff;
```

```
}

QVector4D getObjectPoint(float a, float b) {
    if(check == 1)
    {
        float D=0.5;
        float A=0.3;
        return QVector4D((D+A*cos(b))*cos(a),
                         (D+A*cos(b))*sin(a),
                         A*sin(b),
                         1);
    }
    else
        return QVector4D(cos(a)*cos(b),
                         sin(b),
                         sin(a)*cos(b),
                         1);

}

void Widget::paintEvent(QPaintEvent *)
{
    float resz = (resX+resY)/2;
    QPainter p(this);

    if(check == 1)
    {
        QVector<QVector4D> object;
        float da = M_PI  /steps;

        for (float a = -M_PI; a < M_PI ; a += da)
        {
            for (float b = -M_PI; b < M_PI; b += da)
            {
                float aa = a + da;
```

```
        float bb = b + da;
        object.push_back(getObjectPoint(a, b));
        object.push_back(getObjectPoint(aa, b));
        object.push_back(getObjectPoint(aa, bb));
        object.push_back(getObjectPoint(a, bb));
    }
}
QMatrix4x4 e(1,0,0,0,
             0,1,0,0,
             0,0,1,0,
             0,0,0,1);
float C, S;
C = cos(alpha);
S = sin(alpha);
QMatrix4x4 rotate1(1,0,0,0,
                   0,C,-S,0,
                   0,S,C,0,
                   0,0,0,1);
C = cos(beta);
S = sin(beta);
QMatrix4x4 rotate2(C,0,S,0,
                   0,1,0,0,
                   -S,0,C,0,
                   0,0,0,1);
QMatrix4x4 translate(200,0,0,200,
                     0,200,0,200,
                     0,0,200,200,
                     0,0,0,1);
QMatrix4x4 M = translate * rotate1 * rotate2;
for (int i = 0; i < object.size(); i += 4)
{
    QVector4D A,B,C,D;
    A = M * object[i + 0];
    B = M * object[i + 1];
    C = M * object[i + 2];
```

```
            D = M * object[i + 3];
            QVector3D Zrenie = QVector3D(0,74,760);

            QVector3D N = crossProduct(A-B, A-C);
            QVector3D L = QVector3D(X, Y, Z);

            if (N.z() > 0)
            {
                QPointF ff[4];
                ff[0] = QPointF(A.x()*resz, A.y()*resz);
                ff[1] = QPointF(B.x()*resz, B.y()*resz);
                ff[2] = QPointF(C.x()*resz, C.y()*resz);
                ff[3] = QPointF(D.x()*resz, D.y()*resz);
                float diff = 0;
                float diff1 = 0;float diff2 = 0;float diff3 =0;
                if(sv == "Blinn-Fong")
                {
diff1 = ColorOverflowControl(colorred(N,L,Zrenie));
diff2 = ColorOverflowControl(colorgreen(N,L,Zrenie));
diff3 = ColorOverflowControl(colorblue(N,L,Zrenie));
                }
                else
                {
diff1 = ColorOverflowControl(K_colorred(N,L,Zrenie,m));
diff2 = ColorOverflowControl(K_colorgreen(N,L,Zrenie,m));
diff3 = ColorOverflowControl(K_colorblue(N,L,Zrenie,m));
                }
                ColorOverflowControl(diff);

                p.setPen(Qt::NoPen);
                p.setBrush(QColor(diff1,diff2,diff3));
                p.drawConvexPolygon(ff, 4);
            }
        }
    }
```

```
else
{
    if(check == 2)
    {
        QVector<QVector4D> object;
        float da = M_PI  / steps;
        for (float a =-M_PI ; a < M_PI ; a += da)
        {
            for (float b = -M_PI/2; b < M_PI/2; b += da)
            {
                float aa = a + da;
                float bb = b + da;

                object.push_back(getObjectPoint(a, b));
                object.push_back(getObjectPoint(a, bb));
                object.push_back(getObjectPoint(aa, b));
                object.push_back(getObjectPoint(a, bb));
                object.push_back(getObjectPoint(aa, bb));
                object.push_back(getObjectPoint(aa, b));
            }
        }
        QMatrix4x4 e(1,0,0,0,
                     0,1,0,0,
                     0,0,1,0,
                     0,0,0,1);
        float C, S;
        C = cos(alpha);
        S = sin(alpha);
        QMatrix4x4 rotate1(1,0,0,0,
                           0,C,-S,0,
                           0,S,C,0,
                           0,0,0,1);
        C = cos(beta);
        S = sin(beta);
```

```
            QMatrix4x4 rotate2(C,0,S,0,
                               0,1,0,0,
                               -S,0,C,0,
                               0,0,0,1);
            QMatrix4x4 translate(200,0,0,200,
                                 0,200,0,200,
                                 0,0,200,200,
                                 0,0,0,1);
            QMatrix4x4 M = translate * rotate1 * rotate2;
            for (int i = 0; i < object.size(); i += 3)
            {
                QVector4D A,B,C,D;
                A = M * object[i + 0];
                B = M * object[i + 1];
                C = M * object[i + 2];
                //D = M * object[i + 3];

                QVector3D Zrenie = QVector3D(0,74,760);
                QVector3D N = crossProduct(A-B, A-C);
                QVector3D L = QVector3D(X, Y, Z);
                if (N.z() > 0)
                {
QPointF ff[3];
ff[0] = QPointF(A.x()*resz, A.y()*resz);
ff[1] = QPointF(B.x()*resz, B.y()*resz);
ff[2] = QPointF(C.x()*resz, C.y()*resz);
//                  ff[3] = QPointF(D.x()*resz, D.y()*resz);
float diff = 0;
float diff1 = 0;float diff2 = 0;float diff3 =0;
if(sv == "Blinn-Fong")
                  {
diff1 = ColorOverflowControl(colorred(N,L,Zrenie));
diff2 = ColorOverflowControl(colorgreen(N,L,Zrenie));
diff3 = ColorOverflowControl(colorblue(N,L,Zrenie));
                    }
```

```
                    else
                    {
diff1 = ColorOverflowControl(K_colorred(N,L,Zrenie,m));
diff2 = ColorOverflowControl(K_colorgreen(N,L,Zrenie,m));
diff3 = ColorOverflowControl(K_colorblue(N,L,Zrenie,m));
                    }
                    ColorOverflowControl(diff);

                    p.setPen(Qt::NoPen);
                    p.setBrush(QColor(diff1,diff2,diff3));
                    p.drawConvexPolygon(ff, 3);
                }
            }
        }
    }
}

void Widget::mouseMoveEvent(QMouseEvent *e)
{
    alpha = e->y() *0.005;
    beta = e->x() *0.005;
    update();
}
void Widget::resizeEvent(QResizeEvent *event)
{
    resX = (float)event->size().width()/700;
    resY = (float)event->size().height()/500;
    update();
}

void Widget::on_Figure_currentIndexChanged(int index)
{
    check = index + 1;
    update();
}
```

```
void Widget::on_LightModel_currentTextChanged(const QString &arg)
{
    sv = arg;
    if (ui->LightModel->currentText() == "Blinn-Fong")
                        ui->CoeffSlider->setValue(eli);
    else ui->CoeffSlider->setValue((int)(m*100));
    update();
}
void Widget::on_StepsSlider_valueChanged(int value)
{
    steps = value;
    update();
}
void Widget::on_StepsSpinBox_valueChanged(int arg)
{
    steps = arg;
    update();
}


void Widget::on_LSourceSpinX_valueChanged(int arg)
{
    X = arg;
    update();
}
void Widget::on_LSourceSpinY_valueChanged(int arg)
{
    Y = arg;
    update();
}
void Widget::on_LSourceSpinZ_valueChanged(int arg)
{
    Z = arg;
    update();
}
```

```
void Widget::on_RedSlider1_valueChanged(int value)
{
    Ired1 = value;
    update();
}
void Widget::on_RedSlider2_valueChanged(int value)
{
    Ired2 = value;
    update();
}
void Widget::on_RedSlider3_valueChanged(int value)
{
    Ired3 = value;
    update();
}
void Widget::on_GreenSlider1_valueChanged(int value)
{
    Igreen1 = value;
    update();
}
void Widget::on_GreenSlider2_valueChanged(int value)
{
    Igreen2 = value;
    update();
}
void Widget::on_GreenSlider3_valueChanged(int value)
{
    Igreen3 = value;
    update();
}
void Widget::on_BlueSlider1_valueChanged(int value)
{
    Iblue1 = value;
```

```
    update();
}
void Widget::on_BlueSlider2_valueChanged(int value)
{
    Iblue2 = value;
    update();
}
void Widget::on_BlueSlider3_valueChanged(int value)
{
    Iblue3 = value;
    update();
}
void Widget::on_CoeffSlider_valueChanged(int value)
{
    if (ui->LightModel->currentIndex() == 0) eli = value;
    if (ui->LightModel->currentIndex() == 1) m = value / 100.0;
    update();
}
```

Вывод

Познакомилась с несколькими основными моделями освещения. Сложность состояла в том, чтобы не только понять основные принципы построения каждой модели и необходимые матричные преобразования для определения освещённости, но ещё и применить к ним цветовую модель RGB. Очень пригодился QTabWidget для построения компактного интерфейса с множеством элементов, задающих параметры тел и освещения.

Компьютерная графика
Лабораторная работа №4
«Современный OpenGL»
Черыгова Лиза, 8О-304Б

Задание

Написать программу, рисующую простую сцену с одним-двумя объектами.

Требования

1). Программа должна работать быстро и плавно (не забудьте отключить режим энергосбережения у ноутбука). Желательно, чтобы программа не потребляла 100

2). После запуска программа выводит версию OpenGL;

3). Программа показывает число выводимых кадров в секунду (FPS);

4). В программе использована хотя бы одна пара шейдеров
(вершинный + фрагментный);

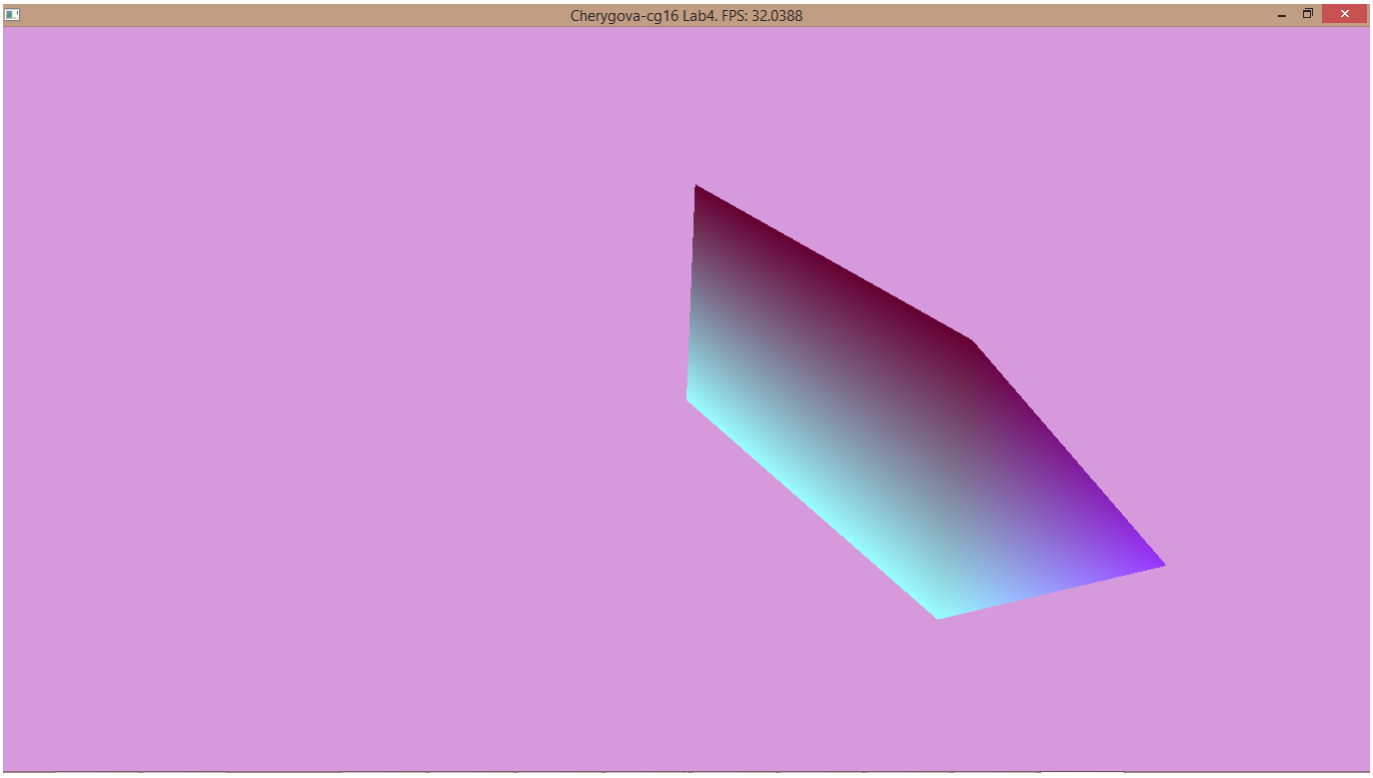
Теоретическая часть

Программы, работающие с трёхмерной графикой и видео (игры, GIS, CAD, CAM и др.), используют шейдеры для определения параметров геометрических объектов или изображения, для изменения изображения (для создания эффектов сдвига, отражения, преломления, затемнения с учётом заданных параметров поглощения и рассеяния света, для наложения текстур на геометрические объекты и др.).

Ранее разработчики игр реализовывали алгоритм создания изображений из геометрических объектов (рендеринг) вручную: составляли алгоритм определения видимых частей сцены, составляли алгоритм наложения текстур, составляли алгоритмы, создающие нестандартные видеоэффекты. Для ускорения рисования некоторые алгоритмы рендеринга были реализованы на аппаратном уровне — с помощью видеокарты. Разработчики игр могли использовать алгоритмы, реализуемые видеокартой, но не могли заставить видеокарту исполнять их собственные алгоритмы, например, для создания нестандартных эффектов. Нестандартные алгоритмы исполнялись на центральном процессоре — более медленном процессоре, по сравнению с процессорами видеокарты.

Для решения проблемы в видеокарты стали добавлять (аппаратно) алгоритмы, востребованные разработчиками. Вскоре стало ясно, что реализовать все алгоритмы невозможно и нецелесообразно; решили дать разработчикам доступ к видеокарте — позволить собирать блоки графического процессора в произвольные конвейеры, реализующие разные алгоритмы. Программы, предназначенные для выполнения на процессорах видеокарты, получили название «шейдеры». Были разработаны специальные языки для составления шейдеров. Теперь в видеокарты загружались не только данные о геометрических объектах («геометрия»), текстуры и другие данные, необходимые для рисования (формировании изображения), но и инструкции для GPU.

Скриншоты



widget.cpp

```
#include "widget.h"

Widget::Widget(QOpenGLWidget *parent):
QOpenGLWidget(parent),
fps_start(QTime::currentTime()),
fps_frames(0),
yaw(0), pitch(0), dist(30)
{
}

Widget::~Widget()
{
}

void Widget::initializeGL()
{
fps_start = QTime::currentTime();
initializeOpenGLFunctions();

qDebug("OpenGL version: %s", glGetString(GL_VERSION));
qDebug("OpenGL vendor: %s", glGetString(GL_VENDOR));
qDebug("OpenGL renderer: %s", glGetString(GL_RENDERER));
qDebug("OpenGL shading language version: %s",
                glGetString(GL_SHADING_LANGUAGE_VERSION));

myShader.addShaderFromSourceFile(QOpenGLShader::Vertex, ":/simple.vert");
myShader.addShaderFromSourceFile(QOpenGLShader::Fragment, ":/simple.frag");
    myShader.link();
```

```
init();

refresh.setInterval(30);
connect(&refresh, SIGNAL(timeout()), this, SLOT(update()));
refresh.start();
}

void Widget::init()
{
    GLfloat vertices[] = {
        // top
        0.0, -0.5, 1.0,
        -0.5, 0.0, 1.0,
        0.0, 0.0, 1.0,
        // bottom
        0.0, -0.5, 0.0,
        -0.5, 0.0, 0.0,
        0.0, 0.0, 0.0
    };
    glGenBuffers(1, &vbo_vertices);
    glBindBuffer(GL_ARRAY_BUFFER, vbo_vertices);
    glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);

    GLfloat colors[] = {
        // top
        0.4, 0.0, 0.2,
        0.6, 0.2, 1.0,
        0.6, 1.0, 1.0,
        // bottom
        0.4, 0.0, 0.2,
        0.6, 0.2, 1.0,
        0.6, 1.0, 1.0
    };
    glGenBuffers(1, &vbo_colors);
    glBindBuffer(GL_ARRAY_BUFFER, vbo_colors);
```

```
    glBufferData(GL_ARRAY_BUFFER, sizeof(colors), colors, GL_STATIC_DRAW);

    GLushort faces[] = {
        0, 1, 2,
        0, 1, 3,
        1, 3, 4,
        1, 5, 4,
        1, 2, 5,
        2, 0, 5,
        0, 3, 5,
        3, 4, 5
    };
    glGenBuffers(1, &ibo_faces);
    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, ibo_faces);
    glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(faces), faces, GL_STATIC_DRAW);

attr_coord3d = myShader.attributeLocation("coord3d");
attr_v_color = myShader.attributeLocation("v_color");
glEnableVertexAttribArray(attr_coord3d);
glEnableVertexAttribArray(attr_v_color);
}

void Widget::draw(QOpenGLShaderProgram *prog)
{
QMatrix4x4 anim;
anim.rotate(yaw, QVector3D(0, 1, 0));
anim.rotate(pitch, QVector3D(1, 0, 0));

QMatrix4x4 model;
model.translate(QVector3D(0.0, 0.0, 3.0));
QMatrix4x4 view;
view.lookAt(QVector3D(0.0, 1.0, 0.0),
                QVector3D(0.0, 0.0, 3.0), QVector3D(0.0, 1.0, 0.0));
QMatrix4x4 projection;
projection.perspective(dist, float(width())/height(), 0.1f, 10.0f);
```

```
QMatrix4x4 mvp = projection * view * model * anim;
prog->setUniformValue("mvp", mvp);

glEnableVertexAttribArray(attr_coord3d);
glBindBuffer(GL_ARRAY_BUFFER, vbo_vertices);
    glVertexAttribPointer(attr_coord3d, 3, GL_FLOAT, GL_FALSE, 0, 0);

glEnableVertexAttribArray(attr_v_color);
glBindBuffer(GL_ARRAY_BUFFER, vbo_colors);
    glVertexAttribPointer(attr_v_color, 3, GL_FLOAT, GL_FALSE, 0, 0);

    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, ibo_faces);
    glDrawElements(GL_TRIANGLES, 24, GL_UNSIGNED_SHORT, 0);

glDisableVertexAttribArray(attr_coord3d);
glDisableVertexAttribArray(attr_v_color);
}

void Widget::paintGL()
{
makeCurrent();

    glClearColor(.839, .6, .862, 2.);
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
glEnable(GL_DEPTH_TEST);

    myShader.bind();
draw(&myShader);
myShader.release();

doneCurrent();
measureFps();
}
```

```
void Widget::measureFps()
{
fps_frames++;
int delta_t =
(QTime::currentTime().msecsSinceStartOfDay() - fps_start.msecsSinceStartOfDay())

if(delta_t > 1000)
{
this->setWindowTitle(QString("Cherygova-cg16 Lab4. FPS: %1")
        .arg(1000. * fps_frames / delta_t));
fps_frames = 0;
fps_start = QTime::currentTime();
}
}

void Widget::mousePressEvent(QMouseEvent *e)
{
if(e->button())
    {
        pmouse[0] = e->x();
        pmouse[1] = e->y();
}
}

void Widget::wheelEvent(QWheelEvent *e)
{
if(e->delta() < 0)
        dist += 1.0;
else if(dist > 1.0)
        dist -= 1.0;

paintGL();
}

void Widget::mouseMoveEvent(QMouseEvent *e)
```

```
{
if(e->buttons())
    {
yaw +=(e->x() - pmouse[0]);
pitch -=(e->y() - pmouse[1]);
    }

    pmouse[0] = e->x();
    pmouse[1] = e->y();
    paintGL();
}
```

simple.frag

```
varying vec3 f_color;

void main(void) {
  gl_FragColor = vec4(f_color.x, f_color.y, f_color.z, 1.0);
}
```

simple.vert

```
attribute vec3 coord3d;
attribute vec3 v_color;
uniform mat4 mvp;
varying vec3 f_color;

void main(void) {
  gl_Position = mvp * vec4(coord3d, 1.0);
  f_color = v_color;
}
```

Вывод

OpenGL — открытый графический стандарт, реализации которого (библиотеки) позволяют работать с примитивами 2D и 3D графики, используя вычислительные возможности видеоадаптеров. Для выполнения лабораторной работы мне пришлось разобраться, как подключить OpenGL в проект и как использовать его вызовы. В ходе выполнения была написана простейшая шейдерная программа, состоящая из вершинного и фрагментного шейдеров и выполняющаяся на графическом процессоре видеокарты, которая рисует сцену с одним каркасным объектом, закрашенным через шейдеры.

Компьютерная графика
Лабораторная работа №5
«Современный OpenGL(2)»
Черыгова Лиза, 8О-304Б

Задание

Написать программу, рисующую простую сцену с несколькими объектами, по которой можно перемещаться.

Требования

1). Программа должна работать быстро и плавно (не забудьте отключить режим энергосбережения у ноутбука). Желательно, чтобы программа не потребляла 100

2). На сцене присутствует поверхность, заданная картой высот;

3). На сцене присутствуют несколько объектов, заданных поверхностями вращения и экструзивными формами;

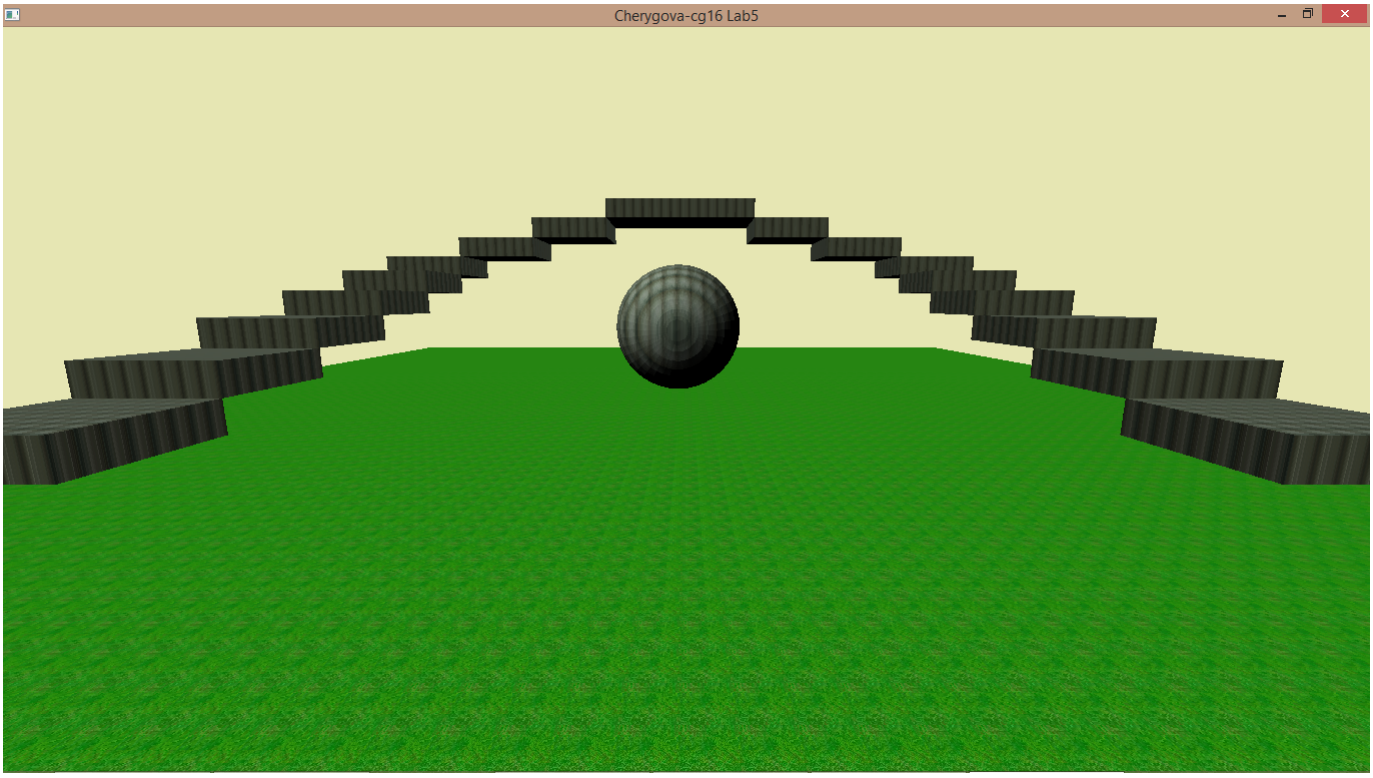
4). Все объекты покрыты, как минимум, обычной текстурой.

Теоретическая часть

Поле высот (англ. Heightmap) — двумерный массив, каждый элемент которого интерпретируется как высота. Часто используются в программах для создания ландшафтов (Terrain), чтобы хранить информацию о высоте каждой точки местности. Используются также в технологии бамп-маппинга (bump mapping).

Экструзия — процесс построения фигуры, при котором образующий её контур (чаще всего) движется вдоль некой траектории.

Скриншоты



Практическая часть

mesh.h

```
#ifndef MESH_H
#define MESH_H

#include <QVector3D>
#include <QVector>
#include <QString>
#include <QFile>
#include <QTextStream>
#include <QDebug>

#include <QOpenGLFunctions>
#include <QOpenGLShaderProgram>
#include <QOpenGLTexture>

struct Vertex
{
public:
QVector3D p;
QVector3D n;

Vertex(){}
Vertex(const Vertex &v): p(v.p), n(v.n){}
Vertex(const QVector3D &p, const QVector3D &n): p(p), n(n){}
Vertex(float x, float y, float z): p(x, y, z){}

};

class Mesh
{
private:
```

```
QVector<Vertex> vertices;
QOpenGLTexture *texture;

public:
Mesh() { texture = 0; }
~Mesh(){}

void loadRawTriangles(QString fileName);
void setTexture(QOpenGLTexture *t);
void draw(QOpenGLFunctions *funcs, QOpenGLShaderProgram *prog);

};


#endif // MESH_H
```

mesh.cpp

```
#include "mesh.h"

void Mesh::loadRawTriangles(QString fileName)
{
QFile file(fileName);
QVector<float> floats;
if (!file.open(QIODevice::ReadOnly))
throw QString("Can't open file ") + fileName;

QTextStream in(&file);
while(!in.atEnd())
{
QString line = in.readLine();
QStringList values = line.split(" ", QString::SkipEmptyParts);

if(values.empty())
continue;
else if(values[0] == "#")
continue;
else if(values.size() == 9)
{
QTextStream stream(&line);
for (int i = 0; i < 9; ++i)
{
float f;
stream >> f;
floats.push_back(f);
}

QVector3D A(floats[0], floats[1], floats[2]);
QVector3D B(floats[3], floats[4], floats[5]);
QVector3D C(floats[6], floats[7], floats[8]);
```

```
QVector3D N = QVector3D::crossProduct(A-B, A-C).normalized();

vertices.push_back(Vertex(A, N));
vertices.push_back(Vertex(B, N));
vertices.push_back(Vertex(C, N));
floats.clear();
} else
qDebug() << values.size() << "Warning: bad line " << line;
}
}

void Mesh::setTexture(QOpenGLTexture *t)
{
texture = t;
}

void Mesh::draw(QOpenGLFunctions *funcs, QOpenGLShaderProgram *prog)
{
int qt_vertex = prog->attributeLocation("qt_Vertex");
int qt_normal = prog->attributeLocation("qt_Normal");

funcs->glEnableVertexAttribArray(qt_vertex);
funcs->glEnableVertexAttribArray(qt_normal);

funcs->glVertexAttribPointer
(qt_vertex, 3, GL_FLOAT, false, 6 * sizeof(float), &vertices[0].p);

funcs->glVertexAttribPointer
(qt_normal, 3, GL_FLOAT, false, 6 * sizeof(float), &vertices[0].n);

if(texture)
{
prog->setUniformValue("sampler", 0);
texture->bind();
}
```

```
glDrawArrays(GL_TRIANGLES, 0, vertices.size());

if(texture)
texture->release();
}
```

Вывод

При построении полноценной сцены применила новые методы задания объектов: карту высот и экструзивные формы. Также впервые использовались текстуры для закраски граней и создания внешнего вида определённого материала. Для облегчения работы программы координаты объектов не высчитывались внутри неё, а задавались через считывание файлов, которые в свою очередь генерировались через отдельные небольшие программы. Отличительной особенностью является возможность перемещения по сцене, т.е. изменение положения зрителя. Всё это делает программу похожей на заготовок 3D-игры.

Компьютерная графика
Лабораторная работа №6
«Системы частиц»
Черыгова Лиза, 8О-304Б

Задание

Разработать физическую модель для выбранной системы частиц. Модель должна быть достаточно упрощенной для реализации в виде шейдера, а также должна, скорее, отражать внешний вид, а не имитировать физические процессы.

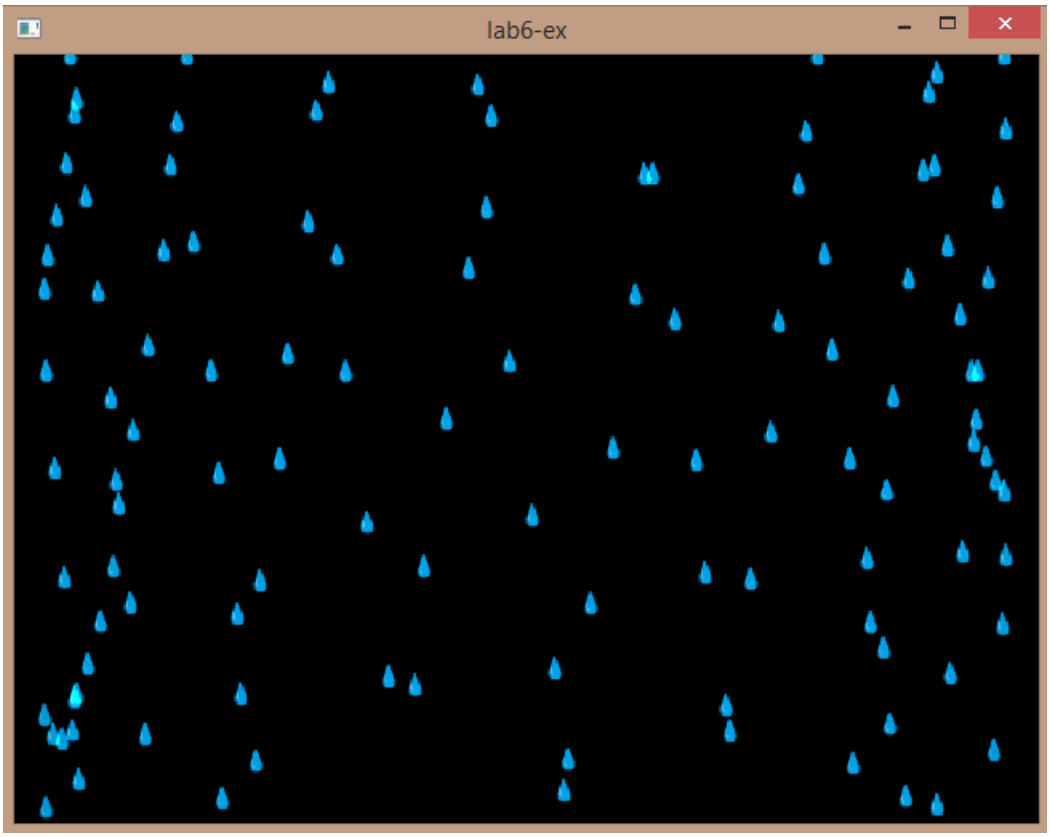
Требования

Дать пользователю возможность менять параметры модели.

Варианты

1). Всплески на поверхности воды от падения объектов, выстрелов и взрывов
2). Почва поднятая взрывом или выстрелом
3). Пыль и штукатурка, поднятые взрывом или выстрелом
4). Огонь
5). Искры от столкновения объектов
6). Дым 7). Молнии 8). Снег (модель должна отличаться от примера) 9). Дождь 10). Мелкие осколки разбитого стекла 11). Фонтан искр 12). Искры, отскакивающие от стен 13). Мелкие осколки разбитого предмета 14). Пузырьки воздуха под водой 15). Вихрь из пыли, песка, листьев или снега 16). Плавно опускающийся водяной пар 17). Падающая листва 18). Падающие перья 19). Брызги воды от водопада 20). Фейерверк 21). Мелкие горящие осколки разбитого предмета 22). Мелкие осколки разбитого предмета под водой

Скриншоты



Практическая часть

particle.frag

```
#version 120

uniform sampler2D sampler;
varying float alpha;

void main(void) {
    vec4 color = texture2D(sampler, gl_PointCoord);
    gl_FragColor = vec4(color.xyz, color.a*alpha);
}
```

particle.vert

```
varying float alpha;
uniform float time;
attribute vec3 vertex;

float atten(float d)
{
    return 1.0/(0.001 + d*d);
}

vec3 calc_pos(vec3 initial, vec3 vel, float time)//верхняя позиция
{
    return initial + vel*time;
}

vec3 random_vector(float param)
{
    return vec3(sin(param), cos(param),
        sin(param) * exp(cos(param) * sin(param)));
}

void main(void)
{


    vec3 rnd = random_vector(vertex.x);

    if (rnd.y > 0.94 || rnd.y < -0.94)
    {

        rnd.y=-1/0.2*cos(rnd.y*rnd.x*rnd.z);
    }
```

```
    vec3 pos = vec3(rnd.y, 1, 0.0);
    vec3 vel = vec3(0, -20.0 * abs(rnd.x), 0.0);


    float t = fract((time + vertex.x) / (100.0 * 2.5));

    vec3 current_pos = calc_pos(pos, vel, t);
    gl_Position =
        gl_ModelViewProjectionMatrix * vec4(current_pos, 1.0);
    alpha = 1.0 - t * t;
}
```

window.cpp

```
#include "window.h"
#include <cstdlib>

Window::Window(QWidget *parent):
QOpenGLWidget(parent),
frame(0),
    particles(1000)//
{
for(int i = 0; i < particles; ++i)
v.push_back(float(i));
}

void Window::initializeGL()
{
initializeOpenGLFunctions();

ps.addShaderFromSourceFile
        (QOpenGLShader::Vertex, ":/particle.vert");
ps.addShaderFromSourceFile
        (QOpenGLShader::Fragment, ":/particle.frag");
ps.link();

vao.create();
vao.bind();

buf.create();
buf.setUsagePattern(QOpenGLBuffer::StaticDraw);
buf.bind();
    buf.allocate(v.data(), v.size() * sizeof(v[0]));
        //закидываем точки в буфер на видеокарту

    texture = new QOpenGLTexture(QImage(":/particle.png"));
```

```
    timer.setInterval(15);
timer.setSingleShot(false);
QObject::connect(&timer, SIGNAL(timeout()), this, SLOT(update()));
    timer.start();
}

void Window::paintGL()
{
glClearColor(0, 0, 0, 0);
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

glEnable(GL_POINT_SPRITE);
glEnable(GL_ALPHA_TEST);
glEnable(GL_BLEND);
glDepthMask(GL_FALSE);
glBlendFunc(GL_SRC_ALPHA, GL_ONE);
glPointSize(20);

// vao.bind();
ps.bind();

//    timer2.start();

texture->bind();
    ps.setUniformValue("time", float(frame));
//       ps.setUniformValue("time", timer2.remainingTime());
ps.setUniformValue("alpha", 1.0f);
ps.setUniformValue("sampler", 0);

GLuint att_vertex = ps.attributeLocation("vertex");
glEnableVertexAttribArray(att_vertex);
glVertexAttribPointer(0, 1, GL_FLOAT, GL_FALSE, 0, 0);
glDrawArrays(GL_POINTS, 0, v.size());
```

```
ps.release();
// vao.release();

frame = (frame + 1) % 1000;
}

Window::~Window()
{

}
```

Вывод

В данной лабораторной работе я разработала физическую модель для дождя. Небольшое затруднение вызвала подборка траектории движения по осям.

Компьютерная графика
Лабораторная работа №7
«Построение плоских полиномиальных кривых»
Черыгова Лиза, 8О-304Б

Задание

Написать программу, рисующую полиномиальную кривую по контрольным точкам.

Требования Дать возможность интерактивно изменять положение контрольных точек, касательных, весов и натяжений. Число сегментов и точек известно заранее.

Варианты

Три кривые на выбор:

1). Сплайн непрерывной кривизны из двух сегментов по трем точкам и касательным в 1-ой и 3-ей точках;

2). Фундаментальная кривая (cardinal spline);

3). B-сплайн. Узловой вектор равномерный;

4). Кривая Безье (без ограничения порядка);

5). Интерполяционный многочлен Лагранжа;

6). NURBS, узловой вектор неравномерный, веса точек различны и изменяемы.

Теоретическая часть

Кривая Безье — параметрическая кривая, задаваемая выражением

$$B(t) = \sum_{i=0}^{n} P_i b_{i,n}(t), \quad 0 \leqslant t \leqslant 1$$

где P_i — функция компонент векторов опорных вершин,

а $b_{i,n}(t)$ — базисные функции кривой Безье, называемые также полиномами Бернштейна.

$$b_{i,n}(t) = \binom{n}{i} t^i (1-t)^{n-i}$$

,

где $\binom{n}{i} = \frac{n!}{i!(n-i)!}$ — число сочетаний из n по i,

где n — степень полинома, i — порядковый номер опорной вершины.

Интерполяционный многочлен Лагранжа - многочлен минимальной степени, принимающий данные значения в данном наборе точек. Для n+1 пар чисел $(x_0, y_0), (x_1, y_1), \ldots, (x_n, y_n)$, где все x_j различны, существует единственный многочлен L(x) степени не более n, для которого $L(x_j) = y_j$.

В простейшем случае (n=1) — это линейный многочлен, график которого — прямая, проходящая через две заданные точки.

Лагранж предложил способ вычисления таких многочленов:

$$L(x) = \sum_{i=0}^{n} y_i l_i(x)$$

где базисные полиномы определяются по формуле:

$$l_i(x) = \prod_{j=0, j\neq i}^{n} \frac{x - x_j}{x_i - x_j} = \frac{x - x_0}{x_i - x_0} \cdots \frac{x - x_{i-1}}{x_i - x_{i-1}} \cdot \frac{x - x_{i+1}}{x_i - x_{i+1}} \cdots \frac{x - x_n}{x_i - x_n}$$

$l_i(x)$ обладают следующими свойствами:

1). Являются многочленами степени n

2). $l_i(x_i) = 1$

3). $l_i(x_j) = 0$ при $j \neq i$

Отсюда следует, что L(x), как линейная комбинация $l_i(x)$, может иметь степень не больше n, и $L(x_i) = y_i$.

B-сплайн — сплайн-функция, имеющая наименьший носитель для заданной степени, порядка гладкости и разбиения области определения. Фундаментальная теорема устанавливает, что любая сплайн-функция для заданной степени, гладкости и области определения может быть представлена как линейная комбинация B-сплайнов той же степени и гладкости на той же области определения.

Кривая, построенная на основе B-сплайн-базиса, описывается следующим образом

$$\bar{p}(t) = \sum_{i=0}^{n} \bar{P}_i N_{ik}(t)$$

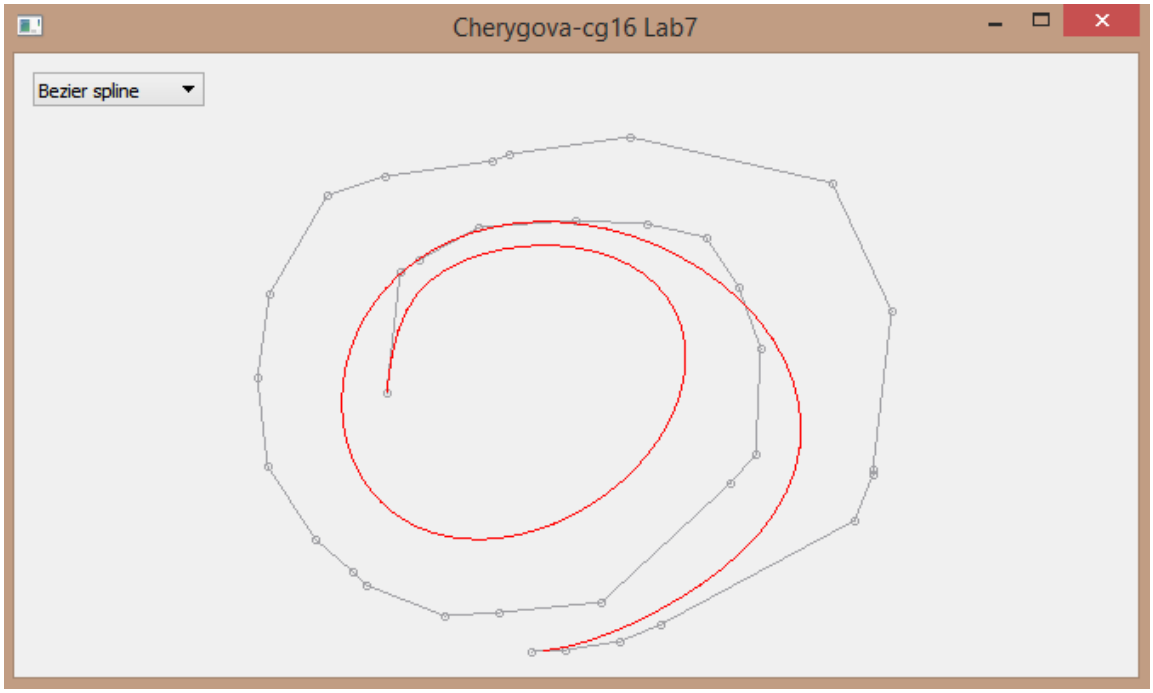
где $\bar{p}(t)$ - радиус-вектор точек на кривой, $\bar{P}_i$ - вершины аппроксимируемой ломаной (всего вершин n+1), а $N_{ik}(t)$ - весовая функция i-й нормализованной B-сплайн базисной кривой порядка k (т. е. степени k-1), задаваемая рекуррентными соотношениями:

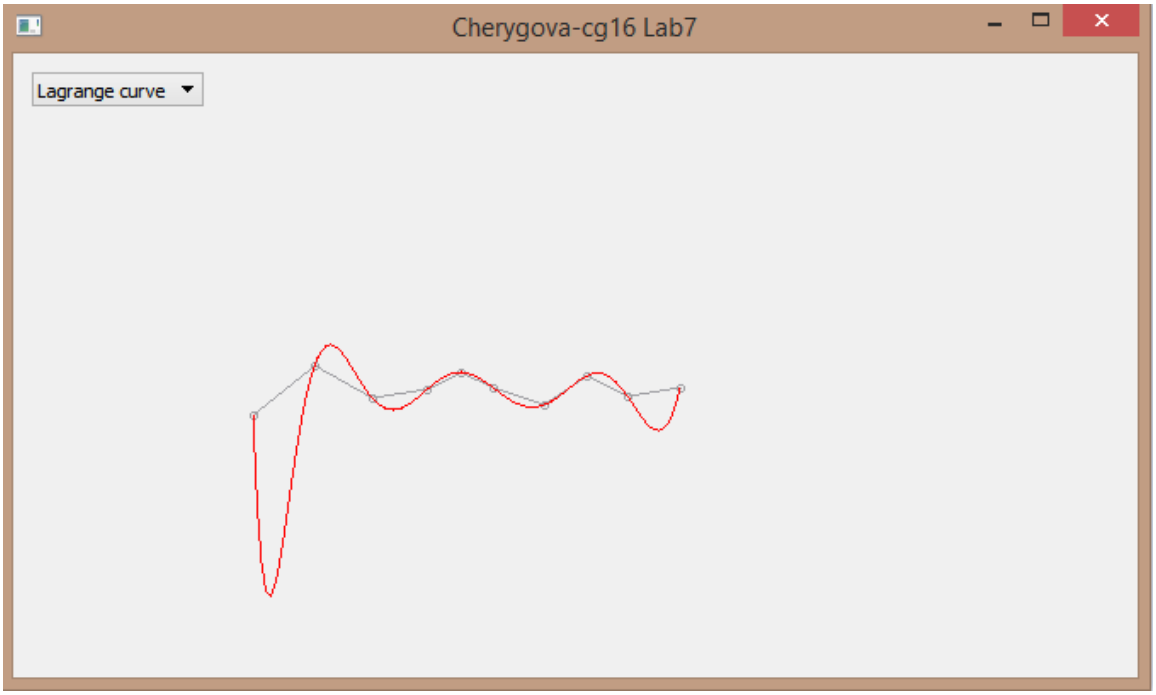
$$N_{i1} = \begin{cases} 1, \ \textit{если } x_i \leq t \leq x_{i-1} \\ 0, \ \textit{если } t \notin (x_i, x_{i-1}) \end{cases}$$

$$N_{ik}(t) = \frac{(t - x_i)N_{i,k-1}(t)}{x_{i+k-1} - x_i} - \frac{(x_{i+k} - t)N_{i+1,k-1}(t)}{x_{i+k} - x_{i+1}}$$

Здесь x_i – элементы узлового вектора, а t – параметр, изменяющийся в диапазоне от 0 до $t_{max} = (n - k + 2)$.

Скриншоты


Cherygova-cg16 Lab7
Bezier spline


Cherygova-cg16 Lab7
Lagrange curve

Практическая часть

b-spline.hpp

```
#ifndef BSPLINE_HPP
#define BSPLINE_HPP

#include <QPointF>
#include <vector>

using namespace std;


class BSpline
{
private:
    int anchors_cnt = 7;
    int order = 4;
    int steps = 100;

    vector<int> knots;

    double N(const vector<int> &T, int i, int order, float t)
    {

        if(order == 1)
            return T.at(i) <= t && t < T.at(i+1);

        double nA = t - T.at(i);
        double dA = T.at(i + order - 1) - T.at(i);

        double nB = T.at(i + order) - t;
        double dB = T.at(i + order) - T.at(i+1);
```

```
        double A = (dA == 0)? 0 : nA / dA;
        double B = (dB == 0)? 0: nB / dB;
return A * N(T, i, order - 1, t) + B * N(T, i + 1, order - 1, t);
    }

public:
    BSpline()
    {
        CalculateKnots();
    }

    void CalculateKnots()
    {

        for(int i = 0; i < order; ++i)
            knots.push_back(0);
        for(int i = 1; i < anchors_cnt - (order - 1); ++i)
            knots.push_back(i);
        for(int i = 0; i < order; ++i)
            knots.push_back(anchors_cnt - (order - 1));
    }

    vector<QPointF> Calculate(vector<QPointF> AnchorPoints)
    {
        vector<QPointF> res;

        for(int s = 0; s < steps; ++s)
        {
            double t = s / float(steps) * order;
            QPointF pt(0.0, 0.0);

            for(int i = 0; i < anchors_cnt; ++i)
                pt += N(knots, i, order, t) * AnchorPoints[i];
            res.push_back(pt);
        }
```

```
        res.push_back(AnchorPoints.back());

        return res;
    }
};

#endif // BSPLINE_HPP
```

bezier-spline.hpp

```
#ifndef BEZIERSPLINE_HPP
#define BEZIERSPLINE_HPP

#include <QPointF>
#include <vector>
#include <cmath>

using namespace std;


class BezierSpline
{
private:
    int steps = 100;

public:
    BezierSpline() {}

    double factorial(double n)
    {
      return (n == 1 || n == 0) ? 1 : factorial(n - 1) * n;
    }

    double B(double i, double n, double t)
    {
return factorial(n) /
(factorial(i) * factorial(n - i)) * pow(t, i) * pow(1 - t, n - i);
    }

    vector<QPointF> Calculate(vector<QPointF> AnchorPoints)
    {
        vector<QPointF> res;
```

```
        for (int s = 0; s < steps; ++s)
        {
            double t = s / (float)steps;
            QPointF pt(0.0, 0.0);

            for(uint i = 0; i < AnchorPoints.size(); ++i)
        pt += B(i, AnchorPoints.size() - 1, t) * AnchorPoints[i];
        res.push_back(pt);
        }

        return res;
    }
};

#endif // BEZIERSPLINE_HPP
```

lagrange-curve.hpp

```
#ifndef LAGRANGECURVE_HPP
#define LAGRANGECURVE_HPP

#include <QPointF>
#include <vector>

using namespace std;


class LagrangeSpline
{
private:
    int steps = 100;
    vector<QPointF> AnchorPoints;

public:
    LagrangeSpline() {}

    double l(uint i, double x)
    {
        double res = 1;
        for (uint j = 0; j < AnchorPoints.size(); ++j)
        {
            if (i == j) continue;
            res *= (x - AnchorPoints[j].x()) /
                (AnchorPoints[i].x() - AnchorPoints[j].x());
        }
        return res;
    }

    double L(double x)
    {
```

```
        double res = 0;

        for (uint i = 0; i < AnchorPoints.size(); ++i)
            res += AnchorPoints[i].y() * l(i, x);

        return res;
    }

    vector<QPointF> Calculate(vector<QPointF> AnchorPoints)
    {
        this->AnchorPoints = AnchorPoints;

        vector<QPointF> res;
        double t = (AnchorPoints.back().x() -
                AnchorPoints.front().x()) / steps;

        for (double s = AnchorPoints.front().x();
            s <= AnchorPoints.back().x(); s += t)
            res.push_back(QPointF(s, L(s)));

        return res;
    }
};

#endif // LAGRANGECURVE_HPP
```

widget.cpp

```
#include "widget.h"
#include "ui_widget.h"

#include <QPainter>

Widget::Widget(QWidget *parent) :
    QWidget(parent),
    ui(new Ui::Widget)
{
    ui->setupUi(this);
}

Widget::~Widget()
{
    delete ui;
}

void Widget::mousePressEvent(QMouseEvent *e)
{
    if (ui->SelectAlgorithm->currentIndex() != 0)
    // если в ComboBox выбрано не "None"
    {
if (e->button() == Qt::LeftButton)
// если была нажата левая клавиша мыши

 AnchorPoints.push_back(QPointF(e->x(), e->y()));
// вставляем координаты мыши в вектор опорных точек

if (e->button() == Qt::RightButton && AnchorPoints.size() > 0)
        // если правая клавиша мыши
AnchorPoints.pop_back(); // удаляем последнюю точку
repaint();
```

```
    }
}

void Widget::DrawAnchorPoints(QPainter& painter)
{
    if (ui->SelectAlgorithm->currentIndex() != 0)
    // если в ComboBox выбрано не "None"
    {
        painter.setPen(QColor(Qt::gray));

        for(uint i = 0; i < AnchorPoints.size(); ++i)
        {
if (i > 0) painter.drawLine(AnchorPoints[i-1], AnchorPoints[i]);
painter.drawEllipse(AnchorPoints[i], 2, 2);
        }
    }
}

void Widget::DrawCurve(QPainter& painter)
{
    if (ui->SelectAlgorithm->currentIndex() != 0 &&
            CurvePoints.size() > 1)
    {
        painter.setPen(QColor(Qt::red));

        for(uint i = 0; i < CurvePoints.size() - 1; ++i)
            painter.drawLine(CurvePoints[i], CurvePoints[i+1]);
    }
}

void Widget::paintEvent(QPaintEvent *)
{
    QPainter painter(this);

    DrawAnchorPoints(painter);
```

```
    if (AnchorPoints.size() > 1)
    {
        switch (ui->SelectAlgorithm->currentIndex())
        {
        case 0: break;
        case 1: CurvePoints = bezier.Calculate(AnchorPoints);
            break;
        case 2: CurvePoints = lagrange.Calculate(AnchorPoints);
            break;
        case 3: CurvePoints = bspline.Calculate(AnchorPoints);
            break;
        }

        DrawCurve(painter);
    }
}

void Widget::on_SelectAlgorithm_currentIndexChanged(int index)
{
    if (index == 0) AnchorPoints.clear();
    repaint();
}
```

Вывод

Из трёх реализованных алгоритмов наиболее удачным получились линии Безье: количество опорных точек может быть очень большим благодаря использованию чисел вещественного типа в вычислениях (позволяет избежать переполнения). Интерполяционный многочлен Лагранжа оказался хорош только для малого количества точек — для длинных кривых придётся использовать разбиение на небольшие участки. B-Spline'ы сложнее в реализации, однако дают наиболее красивый вид кривой.