**Московский авиационный институт**

**(Национальный исследовательский университет)**

Факультет: «Информационные технологии и прикладная математика»

Кафедра: 804 «Теория вероятностей и компьютерное моделирование»

Дисциплина: «Компьютерная графика»

**Курсовой проект**

Тема: Визуализация порции кинематической поверхности.

Студент: Шевчук П.В.

Группа: 80-304

Преподаватель: Чернышов Л.Н.

Дата: 25.12.18

Оценка:

Москва, 2018

**1. Постановка задачи**

Задание: Составить и отладить программу, обеспечивающую каркасную визуализацию порции поверхности заданного типа. Исходные данные готовятся самостоятельно и вводятся из файла или в панели ввода данных. Должна быть обеспечена возможность тестирования программы на различных наборах исходных данных. Программа должна обеспечивать выполнение аффинных преобразований для заданной порции поверхности, а также возможность управлять количеством изображаемых параметрических линий. Для визуализации параметрических линий поверхности разрешается использовать только функции отрисовки отрезков в экранных координатах.

Вариант 15: Кинематическая поверхность. Образующая - эллипс, направляющая - Cardinal Spline 3D.

**2. Решения задачи**

ЯП: C++

ОС: Linux Mint 18.2

Среда разработки: Qt Creator 4.7.2

Библиотеки: QOpenGLFunctions, QCustomPlot.

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

Основные функции:

* Функция customPlot позволяет отрисовать один график по

нескольким точкам и за вертикальной линией будет следовать трассировщик, который будет вставать в ближайшие к точки графика от вертикальной линии.

* Функции initializeGL, resizeGL и paintGL перекрывают методы

родительского класса QGLWidget. Обработчики событий от мыши перекрывают обработчики, унаследованные от QWidget.

* В конструкторе вызывается QGLWidget::setFormat, чтобы

задать контекст устройства отображения OpenGL, и инициализируются приватные переменные-члены класса.

* Функция initializeGL вызывается один раз, перед вызовом

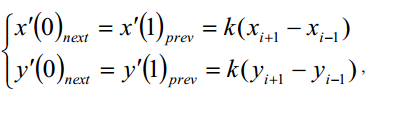
paintGL. Здесь выполняется настройка контекста отображения.

* Функция resizeGL вызывается один раз, перед paintGL, но после

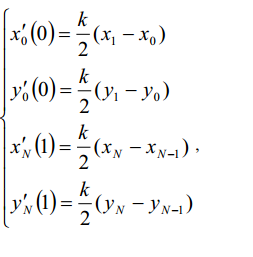
того, как будет вызвана функция initializeGL. Здесь настраивается область просмотра (viewport), проекция и прочие настройки, которые зависят от размера виджета.

Ход работы:

Создаем кинематическую поверхность из эллипсов. Эллипсы строим по семи точкам фундаментального сплайна. Фундаментальный сплайн (cardinal spline) получим, если в точках стыка:



На границах задаём:



Отрисовываем получившиеся после преобразования кривые, получая кинематическую поверхность. Сделано вращение относительно направления света. Также для простоты наблюдатель является источником света.

**3. Руководство по использованию программы**

При запуске программы открывается панель изменения данных и выдается график (рис.1).

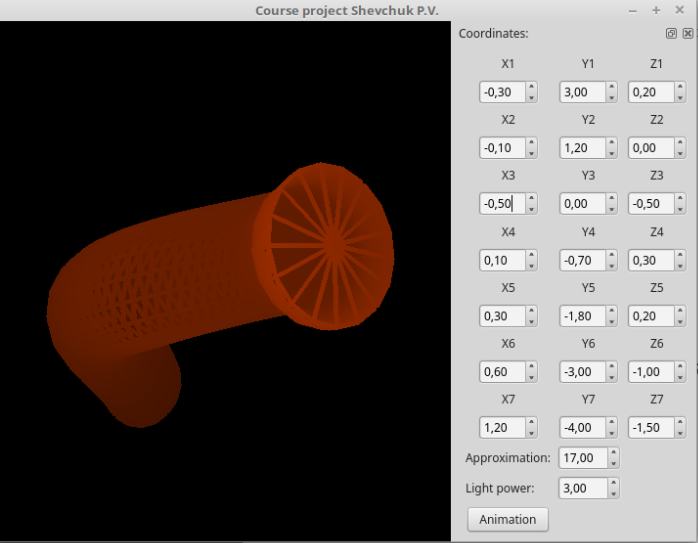


Рис 1. Значения по умолчанию

В окне вывода пользователь может менять положения точек, которые влияют на форму 3D-сплайна. Изменим положение третьей точки (рис.2).

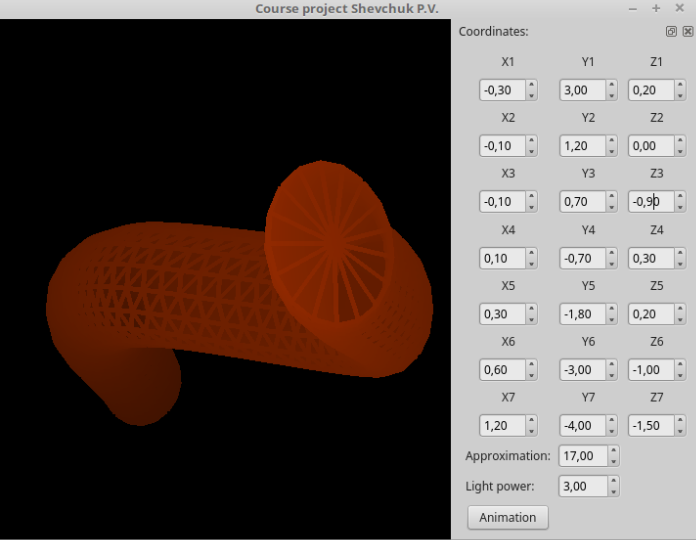


Рис 2. Значения X3 = -0,1; Y3 = 0,7; Z3 = -0,5

Также можно изменять степень аппроксимации фигуры и силу падающего света. Поставим анимацию, которая вращает тело относительно направления на источник света, уменьшим степень аппроксимации и увеличим силу падающего света (рис.3).

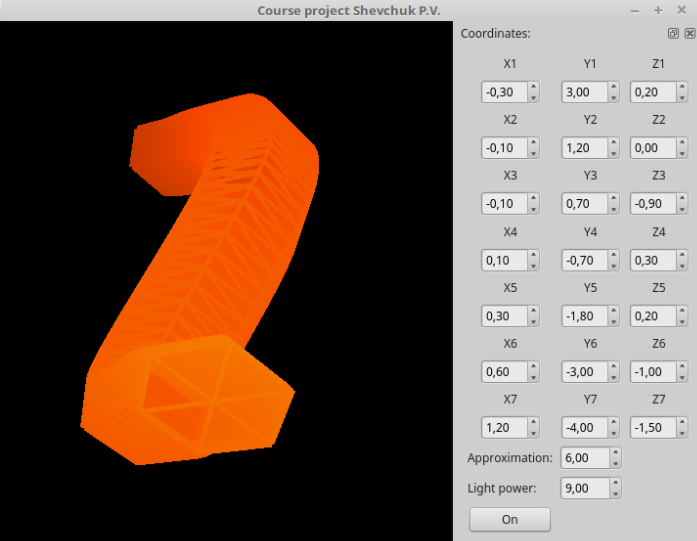


Рис 3. Включена анимация; степень аппроксимации равна 6;

сила падающего света равна 9

**4. Листинг программы**

#include "view.h"

#include "panel.h"

#include <QDebug>

#include <QPainter>

#include <cmath>

#include "simpleobject3d.h"

#include <QOpenGLContext>

#include <QtMath>

#include <algorithm>

// сигнал

View::View(QWidget \*parent) : QOpenGLWidget(parent), pan(nullptr) {

}

// инициализация панели

const Panel \*View::controlPanel() const { return pan; }

void View::setControlPanel(const Panel \*p) {

pan = p;

update();

}

// инициализация графика

void View::*initializeGL*() {

glClearColor(0.0f, 0.0f, 0.0f, 1.0f);

//glEnable(GL\_CULL\_FACE);

glEnable(GL\_DEPTH\_TEST);

glPolygonMode(GL\_FRONT, GL\_LINE);

glLineWidth(5.0f);

glLineStipple(10, 8);

scale = -4.0f;

animation = true;

lightPosition = QVector4D(0.0, 0.0, 0.0, 1.0);

initShaders();

init();

}

// вызов при изменении виджета

void View::*resizeGL*(int w, int h) {

float aspect = w / static\_cast<float>(h);

projectionMatrix.setToIdentity();

projectionMatrix.perspective(45, aspect, 0.3f, 1000000000.0f);

}

// нарисовать график

void View::*paintGL*() {

glClear(GL\_COLOR\_BUFFER\_BIT | GL\_DEPTH\_BUFFER\_BIT);

QMatrix4x4 viewMatrix;

viewMatrix.setToIdentity();

viewMatrix.translate(lr, ud, scale);

viewMatrix.rotate(rotation);

m\_program.bind();

m\_program.setUniformValue("u\_projectionMatrix", projectionMatrix);

m\_program.setUniformValue("u\_viewMatrix", viewMatrix);

m\_program.setUniformValue("u\_lightPosition", lightPosition);

m\_program.setUniformValue("u\_lightPower", static\_cast<float>(pan->getLightpower()));

m\_program.setUniformValue("u\_ambientFactor", static\_cast<float>(pan->getAmbientfactor()));

m\_program.setUniformValue("u\_specularFactor", static\_cast<float>(pan->getSpecularfactor()));

m\_program.setUniformValue("u\_absorptionFactor", static\_cast<float>(pan->getAbsorptionfactor()));

for (auto &object : objects) {

object->draw(&m\_program, context()->functions());

}

}

// инициализация шейдера

void View::initShaders() {

if (!m\_program.addShaderFromSourceFile(QOpenGLShader::Vertex,

":/vshader.vsh")) {

close();

}

if (!m\_program.addShaderFromSourceFile(QOpenGLShader::Fragment,

":/fshader.fsh")) {

close();

}

if (!m\_program.*link*()) {

close();

}

}

// инициализация пирамиды

void View::initPyramid(float width) {

float width\_div\_2 = width / 2.0f;

QVector<VertexData> vertexes;

vertexes.append(VertexData(QVector3D(0.0, -width\_div\_2, 0.0),

QVector2D(0.0, 0.0), QVector3D(0.0, -1.0, 0.0)));

for (int i = 0; i < 17; ++i) {

vertexes.append(VertexData(

(QVector3D(width\_div\_2 \*

cos(2.0f \* static\_cast<float>(M\_PI) \* i / 17.0f),

-width\_div\_2,

width\_div\_2 \*

sin(2.0f \* static\_cast<float>(M\_PI) \* i / 17.0f))),

QVector2D(i % 2, (i + 1) % 2), QVector3D(0.0, -1.0, 0.0)));

}

QVector<GLuint> indexes;

GLuint offset = 0;

for (GLuint i = offset + 1; i < static\_cast<GLuint>(vertexes.size() - 1);

++i) {

indexes.append(i + 1);

indexes.append(offset);

indexes.append(i);

}

indexes.append(offset + 1);

indexes.append(offset);

indexes.append(offset + 17);

offset += 18;

vertexes.append(VertexData(QVector3D(0.0, width\_div\_2, 0.0),

QVector2D(1.0, 1.0), QVector3D(0.0, 1.0, 0.0)));

for (int i = 0; i < 17; ++i) {

vertexes.append(VertexData(

(QVector3D(width\_div\_2 / 2.0f \*

cos(2.0f \* static\_cast<float>(M\_PI) \* i / 17.0f),

width\_div\_2,

width\_div\_2 / 2.0f \*

sin(2.0f \* static\_cast<float>(M\_PI) \* i / 17.0f))),

QVector2D(0.0, 0.0), QVector3D(0.0, 1.0, 0.0)));

}

for (GLuint i = offset + 1; i < static\_cast<GLuint>(vertexes.size() - 1);

++i) {

indexes.append(i);

indexes.append(offset);

indexes.append(i + 1);

}

indexes.append(offset + 17);

indexes.append(offset);

indexes.append(offset + 1);

for (GLuint i = 1; i < 17; ++i) {

indexes.append(i + 1);

indexes.append(i);

indexes.append(i + offset);

indexes.append(i + 1);

indexes.append(i + offset);

indexes.append(i + 1 + offset);

}

indexes.append(1);

indexes.append(17);

indexes.append(17 + offset);

indexes.append(17 + offset);

indexes.append(1 + offset);

indexes.append(1);

objects.append(new SimpleObject3D(vertexes, indexes, QImage(":/cube.jpg"), true));

}

// инициализация сплайна

void View::initSpline(float width)

{

if (!objects.empty()) {

objects[0]->~SimpleObject3D();

objects.pop\_back();

}

auto p0t = pan->getPoint(0);

auto p1t = pan->getPoint(1);

auto p2t = pan->getPoint(2);

auto p3t = pan->getPoint(3);

auto p4t = pan->getPoint(4);

auto p5t = pan->getPoint(5);

auto p6t = pan->getPoint(6);

auto p0 = QVector3D(std::get<0>(p0t), std::get<1>(p0t), std::get<2>(p0t));

auto p1 = QVector3D(std::get<0>(p1t), std::get<1>(p1t), std::get<2>(p1t));

auto p2 = QVector3D(std::get<0>(p2t), std::get<1>(p2t), std::get<2>(p2t));

auto p3 = QVector3D(std::get<0>(p3t), std::get<1>(p3t), std::get<2>(p3t));

auto p4 = QVector3D(std::get<0>(p4t), std::get<1>(p4t), std::get<2>(p4t));

auto p5 = QVector3D(std::get<0>(p5t), std::get<1>(p5t), std::get<2>(p5t));

auto p6 = QVector3D(std::get<0>(p6t), std::get<1>(p6t), std::get<2>(p6t));

QVector3D t1 = 0.5 \* (p2 - p0);

QVector3D t2 = 0.5 \* (p3 - p1);

QVector3D t3 = 0.5 \* (p4 - p2);

QVector3D t4 = 0.5 \* (p5 - p3);

QVector3D t5 = 0.5 \* (p6 - p4);

QVector<VertexData> curve;

interpolate(curve, p1, p2, t1, t2, 15);

interpolate(curve, p2, p3, t2, t3, 15);

interpolate(curve, p3, p4, t3, t4, 15);

interpolate(curve, p4, p5, t4, t5, 15);

QVector<GLuint> indexes;

for (size\_t i = 0; i < curve.size() - 1; ++i) {

indexes.append(i);

indexes.append(i + 1);

}

initCinematic(curve);

}

// нахождение промежуточных значений

void View::interpolate(QVector<VertexData> &res, QVector3D p1, QVector3D p2, QVector3D t1, QVector3D t2, size\_t steps)

{

for (float t = 0; t < steps; ++t) {

float s = t / steps;

float h1 = 2 \* static\_cast<float>(pow(s, 3)) - 3 \* static\_cast<float>(pow(s, 2)) + 1;

float h2 = -2 \* static\_cast<float>(pow(s, 3)) + 3 \* static\_cast<float>(pow(s, 2));

float h3 = static\_cast<float>(pow(s, 3)) - 2 \* static\_cast<float>(pow(s, 2)) + s;

float h4 = static\_cast<float>(pow(s, 3)) - static\_cast<float>(pow(s, 2));

res.push\_back(VertexData(h1 \* p1 + h2 \* p2 + h3 \* t1 + h4 \* t2, QVector2D(0.0, 0.0), QVector3D(0.0, 0.0, 0.0)));

}

}

// описание движение тела

void View::initCinematic(QVector<VertexData> &curve)

{

GLuint e = pan->getAccuracy();

QVector<VertexData> vertexes;

for (size\_t i = 0; i < curve.size(); ++i) {

for (size\_t j = 0; j < e; ++j) {

vertexes.push\_back(VertexData(QVector3D(cos(2.0 \* M\_PI \* j / e) \* 0.5 + curve[i].position.x(), curve[i].position.y(),

sin(2.0 \* M\_PI \* j / e) \* 0.5 + curve[i].position.z()), QVector2D(0.0, 0.0), QVector3D(0.0, 0.0, 0.0)));

}

}

vertexes.push\_back(VertexData(QVector3D(curve[0].position.x(), curve[0].position.y(), curve[0].position.z()), QVector2D(0.0,0.0), QVector3D(0.0,0.0,0.0)));

vertexes.push\_back(VertexData(QVector3D(curve.back().position.x(), curve.back().position.y(), curve.back().position.z()), QVector2D(0.0,0.0), QVector3D(0.0,0.0,0.0)));

QVector<GLuint> indexes;

for (GLuint i = 0; i < e; ++i) {

indexes.push\_back(i);

indexes.push\_back(vertexes.size() - 2);

indexes.push\_back((i + 1) % e);

}

for (GLuint i = 0; i < e; ++i) {

indexes.push\_back(e \* (curve.size() - 1) + i);

indexes.push\_back(e \* (curve.size() - 1) + (i + 1) % e);

indexes.push\_back(vertexes.size() - 1);

}

for (GLuint i = 0; i < curve.size() - 1; ++i) {

for (GLuint j = 0; j < e; ++j) {

indexes.push\_back(i \* e + j);

indexes.push\_back(i \* e + (j + 1) % e);

indexes.push\_back((i + 1) \* e + j);

indexes.push\_back((i + 1) \* e + j);

indexes.push\_back(i \* e + (j + 1) % e);

indexes.push\_back((i + 1) \* e + (j + 1) % e);

}

}

objects.append(new SimpleObject3D(vertexes, indexes, QImage(":/cube.jpg"), true));

}

void View::toggleTimer()

{

if (animation)

killTimer(timer);

else

timer = startTimer(30);

animation = not animation;

}

void View::initFigure(float width)

{

if (!objects.empty()) {

objects[0]->~SimpleObject3D();

objects.pop\_back();

}

float width\_div\_2 = width / 2.0f;

QVector<VertexData> vertexes;

auto cnt = static\_cast<float>(pan->getAccuracy());

float step = width / cnt;

auto a = static\_cast<float>(pan->getA());

vertexes.append(VertexData(QVector3D(0, -width\_div\_2, width\_div\_2), QVector2D(0.0, 0.0), QVector3D(0.0, -1.0, 0.0)));

for (int i = 1; i <= cnt; ++i) {

float z = i \* step;

vertexes.append(VertexData(QVector3D(0, -width\_div\_2, width\_div\_2 - z), QVector2D(1.0, 0.0), QVector3D(0.0, -1.0, 0.0)));

vertexes.append(VertexData(QVector3D(sqrt(z / a), -width\_div\_2, width\_div\_2 - z), QVector2D(0.0, 1.0), QVector3D(0.0, -1.0, 0.0)));

vertexes.append(VertexData(QVector3D(-sqrt(z / a), -width\_div\_2, width\_div\_2 - z), QVector2D(1.0, 1.0), QVector3D(0.0, -1.0, 0.0)));

}

QVector<GLuint> indexes;

indexes.append(1);

indexes.append(2);

indexes.append(0);

indexes.append(1);

indexes.append(0);

indexes.append(3);

for (GLuint i = 1; i < 3 \* (cnt - 1) + 1; i += 3) {

indexes.append(i + 3);

indexes.append(i + 4);

indexes.append(i);

indexes.append(i + 1);

indexes.append(i);

indexes.append(i + 4);

indexes.append(i);

indexes.append(i + 2);

indexes.append(i + 5);

indexes.append(i);

indexes.append(i + 5);

indexes.append(i + 3);

}

GLuint offset = vertexes.size();

vertexes.append(VertexData(QVector3D(0, width\_div\_2, width\_div\_2), QVector2D(0.0, 0.0), QVector3D(0.0, 1.0, 0.0)));

for (GLuint i = 1; i <= cnt; ++i) {

float z = i \* step;

vertexes.append(VertexData(QVector3D(0, width\_div\_2, width\_div\_2 - z), QVector2D(0.0, 0.0), QVector3D(0.0, 1.0, 0.0)));

vertexes.append(VertexData(QVector3D(sqrt(z / a), width\_div\_2, width\_div\_2 - z), QVector2D(0.0, 0.0), QVector3D(0.0, 1.0, 0.0)));

vertexes.append(VertexData(QVector3D(-sqrt(z / a), width\_div\_2, width\_div\_2 - z), QVector2D(0.0, 0.0), QVector3D(0.0, 1.0, 0.0)));

}

indexes.append(offset + 1);

indexes.append(offset + 0);

indexes.append(offset + 2);

indexes.append(offset + 1);

indexes.append(offset + 3);

indexes.append(offset + 0);

for (GLuint i = 1; i < 3 \* (cnt - 1) + 1; i += 3) {

indexes.append(offset + i + 3);

indexes.append(offset + i);

indexes.append(offset + i + 4);

indexes.append(offset + i + 1);

indexes.append(offset + i + 4);

indexes.append(offset + i);

indexes.append(offset + i);

indexes.append(offset + i + 5);

indexes.append(offset + i + 2);

indexes.append(offset + i);

indexes.append(offset + i + 3);

indexes.append(offset + i + 5);

}

vertexes.append(VertexData(QVector3D(0, -width\_div\_2, width\_div\_2), QVector2D(0.0, 0.0), QVector3D(0.0, 0.0, 1.0)));

for (GLuint i = 1; i <= cnt; ++i) {

float z = i \* step;

vertexes.append(VertexData(QVector3D(0, -width\_div\_2, width\_div\_2 - z), QVector2D(0.0, 0.0), QVector3D(0.0, -1.0, 0.0)));

float dx1 = sqrt(z / a) - vertexes[(i == 1 ? vertexes.size() - 2 : vertexes.size() - 3)].position.x();

float dz1 = width\_div\_2 - z - vertexes[(i == 1 ? vertexes.size() - 2 : vertexes.size() - 3)].position.z();

vertexes.append(VertexData(QVector3D(sqrt(z / a), -width\_div\_2, width\_div\_2 - z), QVector2D(0.0, 0.0), QVector3D(-dz1, 0.0, dx1)));

float dx2 = -sqrt(z / a) - vertexes[vertexes.size() - 3].position.x();

float dz2 = width\_div\_2 - z - vertexes[vertexes.size() - 3].position.z();

vertexes.append(VertexData(QVector3D(-sqrt(z / a), -width\_div\_2, width\_div\_2 - z), QVector2D(0.0, 0.0), QVector3D(dz2, 0.0, -dx2)));

}

vertexes.append(VertexData(QVector3D(0, width\_div\_2, width\_div\_2), QVector2D(0.0, 0.0), QVector3D(0.0, 0.0, 1.0)));

for (GLuint i = 1; i <= cnt; ++i) {

float z = i \* step;

vertexes.append(VertexData(QVector3D(0, width\_div\_2, width\_div\_2 - z), QVector2D(0.0, 0.0), QVector3D(0.0, 1.0, 0.0)));

float dx1 = sqrt(z / a) - vertexes[(i == 1 ? vertexes.size() - 2 : vertexes.size() - 3)].position.x();

float dz1 = width\_div\_2 - z - vertexes[(i == 1 ? vertexes.size() - 2 : vertexes.size() - 3)].position.z();

vertexes.append(VertexData(QVector3D(sqrt(z / a), width\_div\_2, width\_div\_2 - z), QVector2D(0.0, 0.0), QVector3D(-dz1, 0.0, dx1)));

float dx2 = -sqrt(z / a) - vertexes[vertexes.size() - 3].position.x();

float dz2 = width\_div\_2 - z - vertexes[vertexes.size() - 3].position.z();

vertexes.append(VertexData(QVector3D(-sqrt(z / a), width\_div\_2, width\_div\_2 - z), QVector2D(0.0, 0.0), QVector3D(dz2, 0.0, -dx2)));

}

for (GLuint i = 3; i <= cnt \* 3; i += 3) {

indexes.append(i + offset \* 3);

indexes.append(i + offset \* 2);

indexes.append(i - 3 + offset \* 2);

indexes.append(i + offset \* 3);

indexes.append(i - 3 + offset \* 2);

indexes.append(i - 3 + offset \* 3);

}

indexes.append(offset \* 3);

indexes.append(offset \* 2);

indexes.append(offset \* 3 + 2);

indexes.append(offset \* 2);

indexes.append(offset \* 2 + 2);

indexes.append(offset \* 3 + 2);

for (GLuint i = 5; i <= cnt \* 3; i += 3) {

indexes.append(i + offset \* 3);

indexes.append(i - 3 + offset \* 2);

indexes.append(i + offset \* 2);

indexes.append(i + offset \* 3);

indexes.append(i + offset \* 3 - 3);

indexes.append(i - 3 + offset \* 2);

}

vertexes.append(VertexData(QVector3D(sqrt(width / a), width\_div\_2, -width\_div\_2), QVector2D(0.0, 0.0), QVector3D(0.0, 0.0, -1.0)));

vertexes.append(VertexData(QVector3D(sqrt(width / a), -width\_div\_2, -width\_div\_2), QVector2D(0.0, 0.0), QVector3D(0.0, 0.0, -1.0)));

vertexes.append(VertexData(QVector3D(-sqrt(width / a), width\_div\_2, -width\_div\_2), QVector2D(0.0, 0.0), QVector3D(0.0, 0.0, -1.0)));

vertexes.append(VertexData(QVector3D(-sqrt(width / a), -width\_div\_2, -width\_div\_2), QVector2D(0.0, 0.0), QVector3D(0.0, 0.0, -1.0)));

indexes.append(vertexes.size() - 4);

indexes.append(vertexes.size() - 3);

indexes.append(vertexes.size() - 2);

indexes.append(vertexes.size() - 3);

indexes.append(vertexes.size() - 1);

indexes.append(vertexes.size() - 2);

objects.append(new SimpleObject3D(vertexes, indexes, QImage(":/cube.jpg"), true));

}

// счётчик

void View::*timerEvent*(QTimerEvent\*)

{

float angle = 2.0f;

QVector3D axis = QVector3D(0.0, 0.0, 1.0);

rotation = QQuaternion::fromAxisAndAngle(axis, angle) \* rotation;

update();

}

// мышка

void View::*mousePressEvent*(QMouseEvent \*event) {

setFocus();

if (event->buttons() == Qt::LeftButton) {

mousePosition = QVector2D(event->localPos());

}

event->accept();

}

void View::*mouseMoveEvent*(QMouseEvent \*event) {

if (event->buttons() != Qt::LeftButton) {

return;

}

QVector2D diff = QVector2D(event->localPos()) - mousePosition;

mousePosition = QVector2D(event->localPos());

float angle = diff.length() / 2.0f;

QVector3D axis = QVector3D(diff.y(), diff.x(), 0);

rotation = QQuaternion::fromAxisAndAngle(axis, angle) \* rotation;

update();

}

void View::*keyPressEvent*(QKeyEvent \*event)

{

if (event->key() == Qt::Key\_A) {

lr -= 0.1;

}

if (event->key() == Qt::Key\_D) {

lr += 0.1;

}

if (event->key() == Qt::Key\_W) {

ud -= 0.1;

}

if (event->key() == Qt::Key\_S) {

ud += 0.1;

}

update();

}

void View::*wheelEvent*(QWheelEvent \*event) {

scale += (event->angleDelta().y() / 150.0f);

scale = std::min(scale, -1.0f);

scale = std::max(scale, -20.0f);

update();

}

void View::init() {

auto p0 = QVector3D(-0.3f, 2 \* 1.5f, 0.2f);

auto p1 = QVector3D(-0.1f, 2 \* 0.6f, 0.0f);

auto p2 = QVector3D(-0.5f, 2 \* 0.0f, -0.5f);

auto p3 = QVector3D(0.1f, -2 \* 0.35f, 0.3f);

auto p4 = QVector3D(0.3f, -2 \* 0.9f, 0.2f);

auto p5 = QVector3D(0.6f, -2 \* 1.5f, -1.0f);

auto p6 = QVector3D(1.2f, -2 \* 2.0f, -1.5f);

auto p7 = QVector3D(2.0f, -2 \* 3.0, -2.0f);

QVector<VertexData> curve;

interpolate(curve, p1, p2, t1, t2, 15);

interpolate(curve, p2, p3, t2, t3, 15);

interpolate(curve, p3, p4, t3, t4, 15);

interpolate(curve, p4, p5, t4, t5, 15);

QVector<GLuint> indexes;

for (size\_t i = 0; i < curve.size() - 1; ++i) {

indexes.append(i);

indexes.append(i + 1);

}

init Cinematic(curve);

}

**5. Вывод**

Благодаря проделанной работе удалось освоить создание ​ порций поверхностей​ в среде разработки Qt на языке программирования C++.

**Список литературы**

1. Фундаментальные сплайны [Электронный ресурс].

URL: <http://sci.sernam.ru/book_rec.php?id=135>

2. Поверхность движения [Электронный ресурс].

URL: <http://stu.sernam.ru/book_g_mod.php?id=36>

3. Трёхмерная графика [Электронный ресурс].

URL: <http://www.linuxcenter.ru/lib/books/qt3/qt3_12.phtml>