Московский Авиационный Институт
(Национальный Исследовательский Университет)

Факультет: “Информационные технологии и прикладная математика”
Кафедра: 806 “Вычислительная математика и программирование”

Лабораторная работа №6

по курсу “Компьютерная графика”

Студент	Полей-Добронравова А.В.
Группа	М8О-307Б-18
Преподаватель	Г.С.Филиппов
Вариант	4
Дата	
Оценка	

Москва, 2020

Создание шейдерных анимационных эффектов в OpenGL

Задача:

Для заданной поверхности обеспечить выполнение следующего шейдерного эффекта.

Вариант 4

Полушарие. Анимация. Вращение вокруг оси OY.

Описание

Программа написана на языке C++ в QTCreator, используется OpenGL. Реализация всей задачи идет в публичном классе
struct Window : QOpenGLWindow, QOpenGLFunctions

Шейдеры - это программы, которые конвертируют входные данные в выходные. При этом шейдеры являются изолированными программами, т.к. им не разрешается общаться друг с другом; единственная связь, которую они имеют, — это их входы и выходы.
(источник: https://ravesli.com/urok-5-shejdery-v-opengl/)

У шейдеров есть несколько типов. Vertex вершинный шейдер отвечает за определение положения точек для отрисовки. Fragment фрагментный шейдер отвечает за закраску фрагмента фигуры (области между точками).

Написание шейдеров на GLSL несколько отличается от обычных программ. В начале идет объявление версии, затем перечень переменных, определенных по следующим правилам:
Можно указывать спецификаторы. Существует два вида спецификатора:

1) Для указания вида входных параметров функции (in, out,inout)
2) Для формирования интерфейса шейдера (attribute, uniform, varying, const)

- **attribute**: для часто меняющейся информации, которую необходимо передавать *для каждой вершины* отдельно
- **uniform**: для относительно редко меняющейся информации, которая может быть использована как вершинным шейдером, так и фрагментным шейдером
- **varying**: для *интерполированной* информации, передающейся *от вершинного шейдера к фрагментному*
- **const**: для объявления *неизменяемых* идентификаторов, значения которых известны еще на этапе компиляции

Для передачи информации в шейдер используются *встроенные* и *определенные разработчиком* **attribute-**, **uniform-**, **varying**-переменные

Шейдеры у меня использованы классические, но для поворота я передаю вершинному шейдеру матрицу поворота, которая умножается на начальную позицию, вычисляя новые координаты точек. Создается матрица внутри основной программы. Переменная GLfloat gDegreesRotated принимает значения от 0 до 360 градусов, изменяясь по таймеру каждые 50 миллисекунд, и матрица поворота по оси Y меняется в зависимости от него rotation.rotate(gDegreesRotated, 0, 1, 0); Также для удобства подключен поворот с помощью мышки, и угол, на который поворачивается по остальным осям, также передается матрице поворота:
rotation.rotate(xRot, 1, 0, 0);
rotation.rotate(zRot, 0, 0, 1);

Чтобы работать удобно с vertex шейдером, я создала класс
```
struct Vertex
{
public:
 GLfloat m_position[3],
        m_color[3];
```
}; хранящий координату в 3Д пространстве и цвет в RGB значениях.

Использование шейдеров в OpenGL реализуется через создание QOpenGLShaderProgram, хранящей текущие шейдеры, буферы объектов для отрисовки. Для модификации или отрисовки наполнения программы нужно вызвать функцию этого класса bind(); Для конца работы с QOpenGLShaderProgram функцию класса release();
Итак, кроме нее необходимо создать QOpenGLBuffer, QOpenGLVertexArrayObject, которые в одновременной работе создадут задаваемую картинку. Алгоритм таков:
Создаются эти два новых объекта(выделяется память, вызывается функция create() для обоих), далее bind() этих двух объектов. Для QOpenGLBuffer настраиваем режим setUsagePattern(QOpenGLBuffer::StaticDraw); QOpenGLBuffer передается массив с точками единичного плоского объекта (полигон), allocate(c, sizeof(c)) выполняет перемещение этих данных в наш буфер.
После этого в самой QOpenGLShaderProgram идет настройка поступления данных в шейдеры из наши буферов через задание enableAttributeArray(); setAttributeBuffer(), и буферы мы отпускаем через release();
Так как для отрисовки многогранника нужно иметь несколько полигонов, у меня созданы массивы буферов для OpenGL каждого из них.
QVector<QOpenGLBuffer*> m_vbo1;
QVector<QOpenGLVertexArrayObject*> m_vao1;
отвечают за отрисовку нижней части полушария (дна).

QVector<QOpenGLBuffer*> m_vbo2;
QVector<QOpenGLVertexArrayObject*> m_vao2;
отвечают за верхушку полушария в виде треугольников.

QVector<QOpenGLBuffer*> m_vbo3;
QVector<QOpenGLVertexArrayObject*> m_vao3;
боковины полушария.

Функция *paintGL*() вызывается каждый раз после вызова update() по таймеру или другому сигналу. В ней идет формирование матрицы поворота и сама отрисовка через bind() программы, vao и vbo, а далее вызова glDrawArrays(MODE, начало считывания массива, количество элементов для считывания);

Цвет для отрисовки я создаю при формировании буферов. Для наглядности я сделала мешанину из цветов с точными границами, чтобы видеть каждый полигон многогранника.

Код

```
#include <QtGui/QGuiApplication>
#include <QtGui/QKeyEvent>

#include <QtGui/QOpenGLWindow>
#include <QtGui/QOpenGLBuffer>
#include <QtGui/QOpenGLFunctions>
#include <QtGui/QOpenGLShaderProgram>
#include <QtGui/QOpenGLVertexArrayObject>
#include <QtMath>
#include <QTimer>


static QString vertexShader =
    "#version 120\n"
    "\n"
    "attribute vec3 position;\n"
    "attribute vec3 color;\n"
    "\n"
    "varying vec3 v_color;\n"
    "uniform mat4 matrixRotation;"
    "\n"
    "void main()\n"
    "{\n"
    "    v_color = color;\n"
    "    gl_Position =  matrixRotation * vec4(position, 1);\n"
    "}\n"
    ;
```

```
static QString fragmentShader =
      "#version 120\n"
      "\n"
      "varying vec3 v_color;\n"
      "\n"
      "void main()\n"
      "{\n"
      "    gl_FragColor = vec4(v_color, 1);\n"
      "}\n"
      ;
struct Vertex
{
public:
 GLfloat m_position[3],
       m_color[3];
};

struct Window : QOpenGLWindow, QOpenGLFunctions
{
  Window()
  {
     gDegreesRotated = 0;
     xRot = yRot = zRot = 0;
     scale = 0.06;
     //setSurfaceType(OpenGLSurface);
     timer = new QTimer();
     timer->setSingleShot(true);
     connect(timer, SIGNAL(timeout()), this, SLOT(update()));
     timer->start(50);
  }

  Vertex* create(GLfloat x, GLfloat y, GLfloat z, GLfloat x1, GLfloat x2, GLfloat x3) {
     Vertex* dop = new Vertex;
     dop->m_position[0] = x;
     dop->m_position[1] = y;
     dop->m_position[2] = z;
     dop->m_color[0] = x1;
     dop->m_color[1] = x2;
     dop->m_color[2] = x3;
     return dop;
  }
  void createShaderProgram()
  {
     if ( !m_pgm.addShaderFromSourceCode( QOpenGLShader::Vertex, vertexShader)) {
        qDebug() << "Error in vertex shader:" << m_pgm.log();
        exit(1);
     }
     if ( !m_pgm.addShaderFromSourceCode( QOpenGLShader::Fragment,
fragmentShader)) {
        qDebug() << "Error in fragment shader:" << m_pgm.log();
```

```
        exit(1);
    }
    if ( !m_pgm.link() ) {
        qDebug() << "Error linking shader program:" << m_pgm.log();
        exit(1);
    }
}

void createGeometry()
{
        double toch = 5;
        double toch1 = M_PI / toch;
        double toch2 = M_PI / toch / 2;
        double radius = 1; // центр 0,0,0
        double h = 0;
        double i;
        QOpenGLBuffer* buf;
        QOpenGLVertexArrayObject* vaobuf;
        for(i = 2 * M_PI; i >= toch1; i = i - toch1) {
            buf = new QOpenGLBuffer(QOpenGLBuffer::VertexBuffer);
            vaobuf = new QOpenGLVertexArrayObject(this);
            vaobuf->create();
            vaobuf->bind();
            Vertex c[3] = {*create(0.00, 0.00, 0.0, 1.0, 0.0, 0.0),
                        *create(radius * cos(i),  radius * sin(i), 0.0, 0.0, 1.0, 0.0),
                        *create(radius * cos(i- toch1), radius * sin(i- toch1), 0.0, 0.0, 0.0, 1.0)};
            buf->create();
            buf->setUsagePattern(QOpenGLBuffer::StaticDraw); // StreamDraw
            buf->bind();
            buf->allocate(c, sizeof(c));
            m_pgm.enableAttributeArray("position");
            m_pgm.setAttributeBuffer("position", GL_FLOAT, offsetof(Vertex, m_position), 3,
sizeof(Vertex) );
            m_pgm.enableAttributeArray("color");
            m_pgm.setAttributeBuffer("color",  GL_FLOAT, offsetof(Vertex, m_color), 3,
sizeof(Vertex) );
            m_vbo1.push_back(buf);
            m_vao1.push_back(vaobuf);
            vaobuf->release();
            buf->release();
        }
        for(i = 0; i <= 2 * M_PI-toch1; i = i + toch1) {
            for (h = 0; h <= M_PI_2; h = h + toch2) {
                if (h+ toch2 <= M_PI_2) { //края
                    buf = new QOpenGLBuffer(QOpenGLBuffer::VertexBuffer);
                    vaobuf = new QOpenGLVertexArrayObject(this);
                    vaobuf->create();
                    vaobuf->bind();
                    Vertex c[4] = {*create(radius * cos(i) * cos(h), radius * sin(i) * cos(h),
radius * sin(h), 1.0, 0.0, 0.0),
```

```
                                *create(radius * cos(i+ toch1) * cos(h), radius * sin(i+ toch1) *
cos(h), radius * sin(h), 0.0, 1.0, 0.0),
                                *create(radius * cos(i+ toch1) * cos(h+ toch2), radius * sin(i+
toch1) * cos(h+ toch2), radius * sin(h+ toch2), 0.0, 0.0, 1.0),
                                *create(radius * cos(i) * cos(h+ toch2), radius * sin(i) * cos(h+
toch2), radius * sin(h+ toch2), 1.0, 0.0, 0.0)};
                    buf->create();
                    buf->setUsagePattern(QOpenGLBuffer::StaticDraw);
                    buf->bind();
                    buf->allocate(c, sizeof(c));
                    m_pgm.enableAttributeArray("position");
                    m_pgm.setAttributeBuffer("position", GL_FLOAT, offsetof(Vertex,
m_position), 3, sizeof(Vertex) );
                    m_pgm.enableAttributeArray("color");
                    m_pgm.setAttributeBuffer("color",  GL_FLOAT, offsetof(Vertex, m_color),
3, sizeof(Vertex) );
                    m_vbo3.push_back(buf);
                    m_vao3.push_back(vaobuf);
                    vaobuf->release();
                    buf->release();
                }
                else {
                    buf = new QOpenGLBuffer(QOpenGLBuffer::VertexBuffer);
                    vaobuf = new QOpenGLVertexArrayObject(this);
                    vaobuf->create();
                    vaobuf->bind();
                     Vertex c[3] = {*create(0, 0, radius * sin(h+toch2), 1.0, 0.0, 0.0),
                                *create(radius * cos(i) * cos(h), radius * sin(i) * cos(h), radius *
sin(h), 0.0, 1.0, 0.0),
                                *create(radius * cos(i+ toch1) * cos(h), radius * sin(i+ toch1) *
cos(h), radius * sin(h), 0.0, 0.0, 1.0)};
                    buf->create();
                    buf->setUsagePattern(QOpenGLBuffer::StaticDraw);
                    buf->bind();
                    buf->allocate(c, sizeof(c));
                    m_pgm.enableAttributeArray("position");
                    m_pgm.setAttributeBuffer("position", GL_FLOAT, offsetof(Vertex,
m_position), 3, sizeof(Vertex) );
                    m_pgm.enableAttributeArray("color");
                    m_pgm.setAttributeBuffer("color",  GL_FLOAT, offsetof(Vertex,
m_color), 3, sizeof(Vertex) );
                    m_vbo2.push_back(buf);
                    m_vao2.push_back(vaobuf);
                    vaobuf->release();
                    buf->release();
                }
            }
        }
    }
```

```
void initializeGL()
{
    QOpenGLFunctions::initializeOpenGLFunctions();
    createShaderProgram(); m_pgm.bind();
    createGeometry();
}

void resizeGL(int w, int h)
{
    glViewport(0, 0, w, h);
    update();
}

void paintGL()
{
    if (gDegreesRotated < 360) {
            gDegreesRotated = gDegreesRotated + 1;
    }
    else {
        gDegreesRotated = 1;
    }
    glClear(GL_COLOR_BUFFER_BIT);
    QMatrix4x4 rotation;
    rotation.setToIdentity();
    rotation.rotate(gDegreesRotated, 0, 1, 0);
    rotation.rotate(xRot, 1, 0, 0);
    rotation.rotate(zRot, 0, 0, 1);
    m_pgm.bind();
    m_pgm.setUniformValue("matrixRotation", rotation);
    for (int i = 0; i < m_vao1.size(); i++) {
        m_vao1[i]->bind();
        m_vbo1[i]->bind();
            glDrawArrays(GL_TRIANGLES, 0, 3);
        m_vao1[i]->release();
        m_vbo1[i]->release();
    }
    for (int i = 0; i < m_vao2.size(); i++) {
        m_vao2[i]->bind();
        m_vbo2[i]->bind();
            glDrawArrays(GL_TRIANGLES, 0, 3);
        m_vao2[i]->release();
        m_vbo2[i]->release();
    }
    for (int i = 0; i < m_vao3.size(); i++) {
        m_vao3[i]->bind();
        m_vbo3[i]->bind();
            glDrawArrays(GL_QUADS, 0, 4);
        m_vao3[i]->release();
        m_vbo3[i]->release();
    }
```

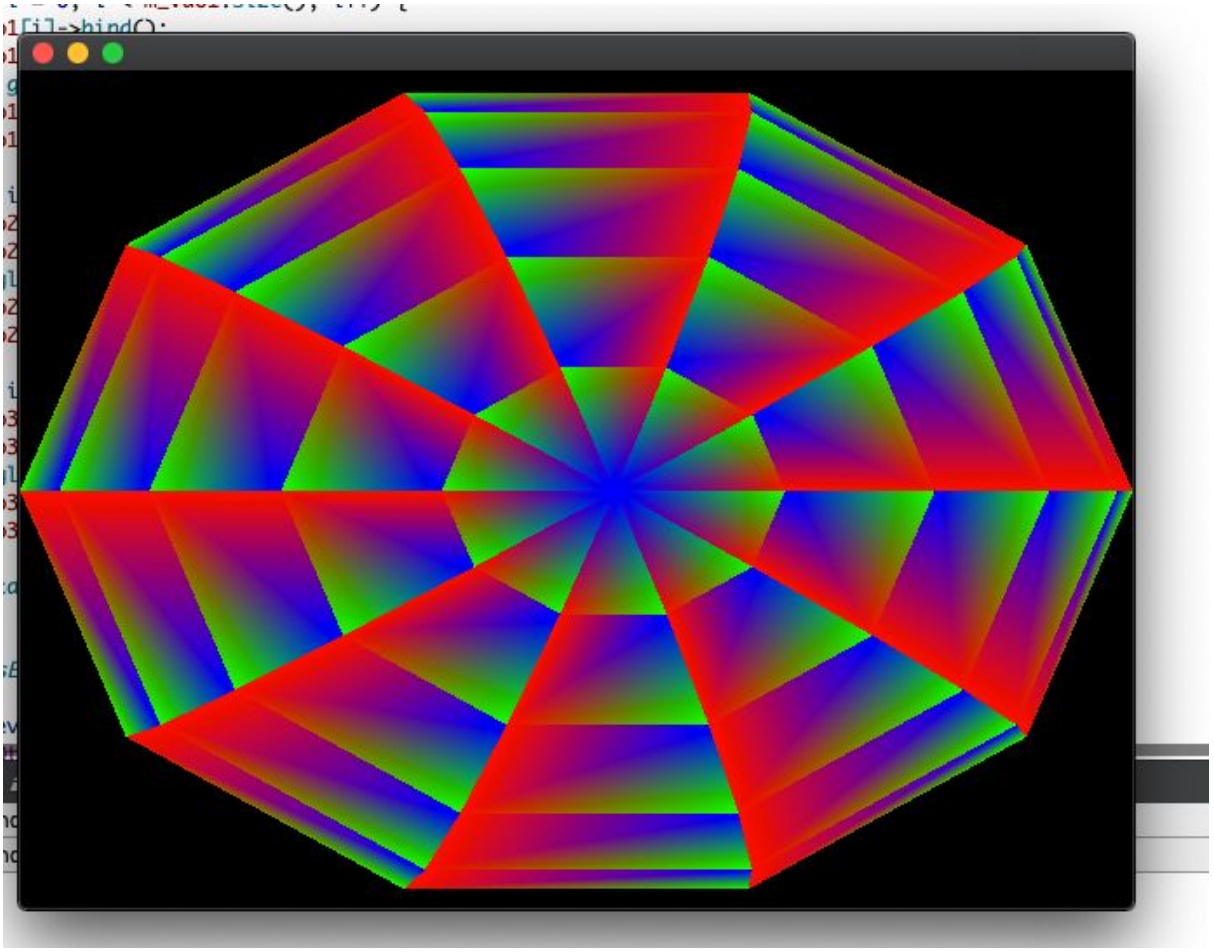
```
    timer->start(50);
}

void keyPressEvent(QKeyEvent * ev)
{
    switch (ev->key()) {
        case Qt::Key_Escape:
            exit(0);
            break;
        default:
            QWindow::keyPressEvent(ev);
            break;
    }

}

void mousePressEvent(QMouseEvent *event)
{
    lastPos = event->pos();
}

// перемещение мыши
void mouseMoveEvent(QMouseEvent *event)
{
    int dx = event->x() - lastPos.x();
    int dy = event->y() - lastPos.y();

    if (event->buttons() & Qt::LeftButton) {
        setXRotation(xRot + dy);
        setYRotation(yRot + dx);
    }
    if (event->buttons() & Qt::RightButton) {
        setXRotation(xRot + dy);
        setZRotation(zRot + dx);
    }
    lastPos = event->pos();
}
// обнуление периода
static void qNormalizeAngle(int &angle)
{
    while (angle < 0)
        angle += 360;
    while (angle > 360)
        angle -= 360;
}

// поворот меша на угол angle, относительно оси oX
void setXRotation(int angle)
{
    qNormalizeAngle(angle);
```

```
        if (angle != xRot) {
            xRot = angle;
            update();
        }
    }

    // поворот меша на угол angle, относительно оси oY
    void setYRotation(int angle)
    {
        qNormalizeAngle(angle);
        if (angle != yRot) {
            yRot = angle;
            update();
        }
    }

    // поворот меша на угол angle, относительно оси oZ
    void setZRotation(int angle)
    {
        qNormalizeAngle(angle);
        if (angle != zRot) {
            zRot = angle;
            update();
        }
    }

    QOpenGLShaderProgram     m_pgm;
    QVector<QOpenGLBuffer*> m_vbo1;
    QVector<QOpenGLBuffer*> m_vbo2;
    QVector<QOpenGLBuffer*> m_vbo3;
    QVector<QOpenGLVertexArrayObject*> m_vao1;
    QVector<QOpenGLVertexArrayObject*> m_vao2;
    QVector<QOpenGLVertexArrayObject*> m_vao3;
    QTimer* timer;
    GLfloat gDegreesRotated;
    int xRot;
    int yRot;
    int zRot;
    double scale;
    bool flag = true;
    QPoint lastPos;
signals:
    void clicked();
};

int main(int argc, char *argv[])
{
    QGuiApplication a(argc,argv);
    Window w;
    w.setWidth(640); w.setHeight(480);
```

```
    w.show();
    return a.exec();
}
```

Пример работы

Выполнение самой анимации представлено на защите лабораторной, здесь показана закраска полушария:



Вывод

Шейдеры обрабатываются видеокартой компьютера, если она имеется. Их идея и использование заключается в инкапсулировании самого процесса отрисовки, основная работа остается в вычислении входных параметров для шейдера.