МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МО ЭВМ

ОТЧЕТ

по лабораторной работе №5
по дисциплине «Компьютерная графика»
Тема: Использование шейдеров

Студент гр. 8383 _____ Киреев К.А.

Студент гр. 8383 _____ Муковский Д.В.

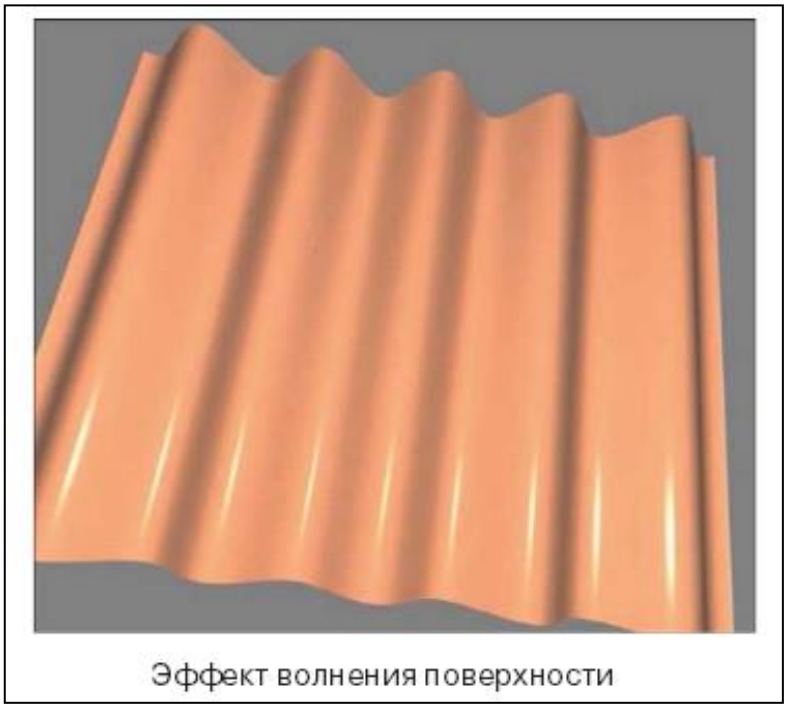
Преподаватель _____ Герасимова Т.В.

Санкт-Петербург
2021

Цель работы.

Разработать визуальный эффект по заданию, реализованный средствами языка шейдеров GLSL.

Вариант 24.

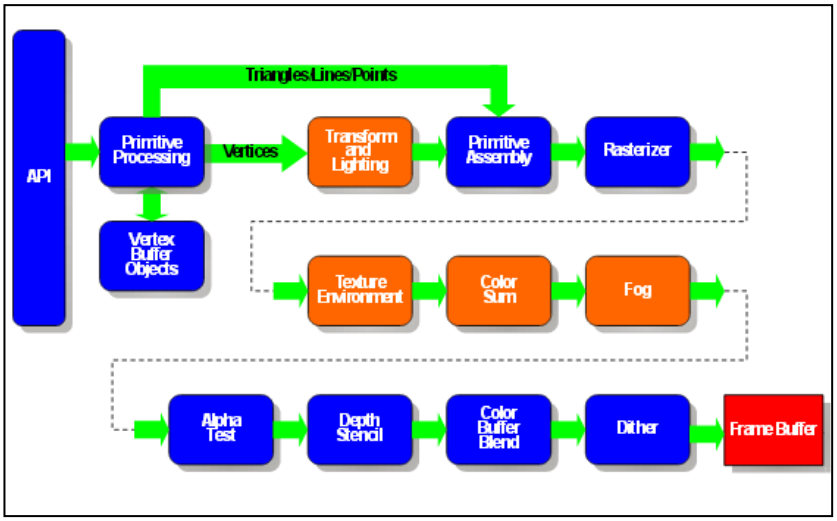


Эффект волнения поверхности

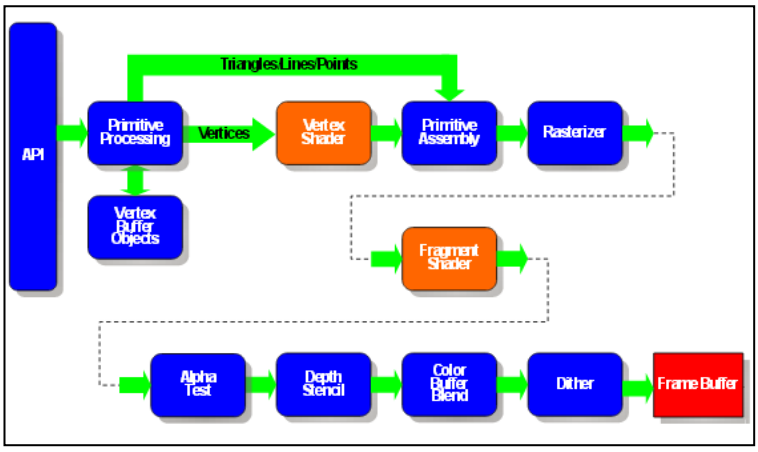
Анимация. Координата X изменяется по закону **X*cos(t)**, координата Y изменяется по закону Y = **Ysin(X+t)**

Теоретические сведения.

Конвейер с фиксированной функциональностью



Программированный графический конвейер



Программируемый графический конвейер позволяет обойти фиксированную функциональность стандартного графического конвейера OpenGL.

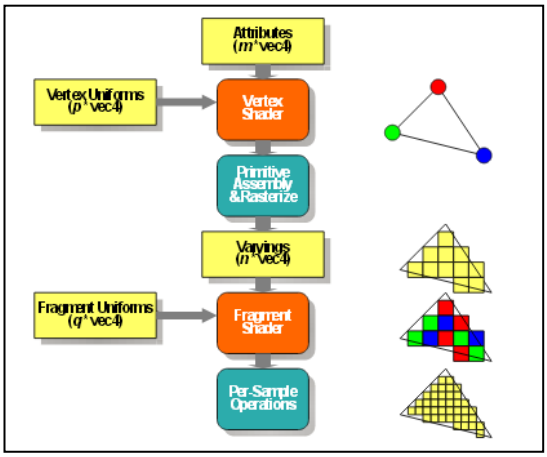
Для вершин – задать необычное преобразование вершин (обычное – это просто умножение координат вершин на модельную и видовую матрицу). Типичное применение - скелетная анимация, анимация волн.

Для геометрических примитивов, таких как треугольник, позволяет сформировать несколько иную геометрию, чем было, например, разбить треугольник на несколько более мелких.

Для фрагментов – позволяет определить цвет фрагмента (пикселя) в обход стандартных моделей освещения. Например, реализовать процедурные текстуры: дерева или мрамора.

Программа, используемая для расширения фиксированной функциональности OpenGL называется **шейдер,** соответственно различают 3 типа шейдеров вершинный, геометрический и фрагментный

Программируемая модель



Выполнение работы.

Шейдер

Загрузка шейдера *vshader*, в котором содержится следующий код на языке *GLSL*:

```
attribute vec4 vertex;
uniform mat4 mvp;
varying vec4 vtx;
uniform float time;

void main() {
    float t = vertex.x*5*cos(time);
    gl_Position = mvp * vec4(vertex.x, vertex.y*sin(t*0.5+time), vertex.z, 1.0);
    vtx = vertex;
}
```

Шейдер обеспечивает вершинную анимацию. Для этого нам понадобится переменная для хранения времени, или счетчик кадров. Но вершинные шейдеры не могут передавать переменные между разными вершинами, не говоря уж о кадрах. Поэтому нам понадобится объявить переменную-счетчик в приложении *OpenGL*, и передать её шейдеру как *uniform*-переменную. Назовём счетчик кадров "*time*", и получим в шейдере *uniform*-переменную с этим именем.

Здесь в качестве текущей амплитуды гармонического колебания берется значение, которое изменяется по закону *X*cos(t)*, координата Y изменяется по закону *Y = Ysin(X+t).*

В методе инициализации *initializeGL* происходит линковка шейдерных модулей:

```
//compile shaders
program = new QGLShaderProgram(this);
program->addShaderFromSourceFile(QGLShader::Vertex, ":/vshader.vsh");
program->addShaderFromSourceFile(QGLShader::Fragment, ":/fshader.fsh");
program->link();
program->bind();
```

Далее создается таймер для анимации:

```
QTimer *timer = new QTimer(this);
connect(timer, SIGNAL(timeout()), this, SLOT(refresh()));
timer->start(0);
```

Далее шейдерам через *uniform* передается матрица проекций *OpenGL:*

```
glPushMatrix();
float* mvp = new float[16];
glGetFloatv(GL_PROJECTION_MATRIX, mvp);
glMultMatrixf(mvp);
glGetFloatv(GL_MODELVIEW_MATRIX, mvp);
glPopMatrix();
QMatrix4x4 mvpMatrix(mvp);
program->setUniformValue("mvp", mvpMatrix);
```

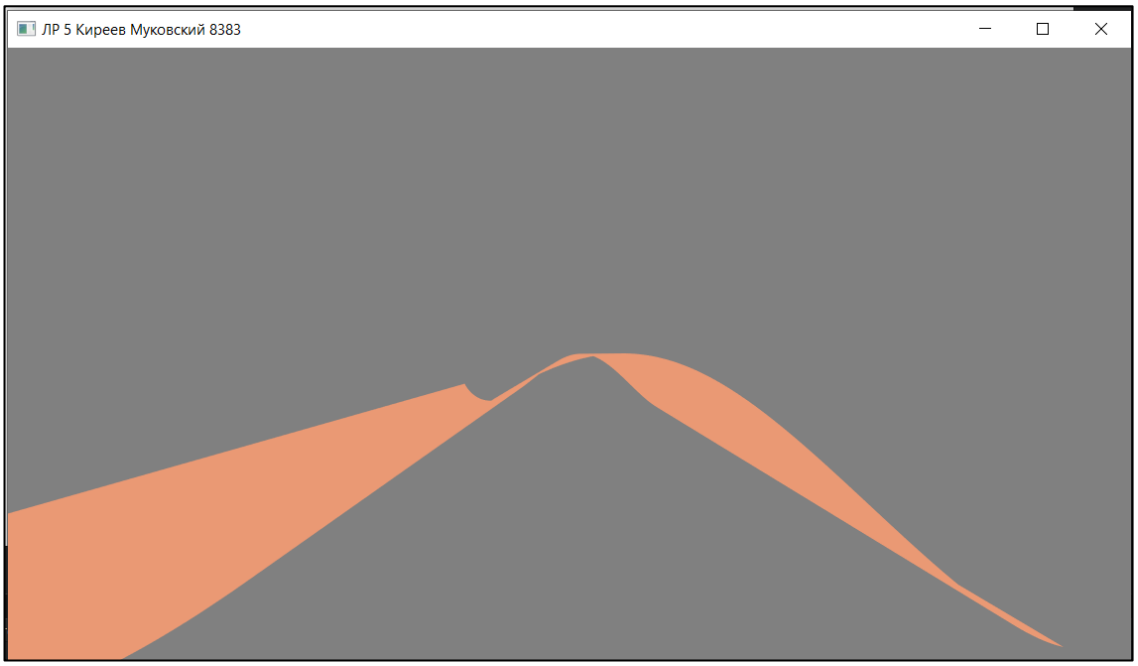
Изменение значения таймера *time*:

```
time += 0.1;
if (time > 2 * M_PI) time = 0.0;
program->setUniformValue("time", time);
update();
```

В методе *paintGL* происходит превращение кривой в поверхность на сцене:

```
for (int i = 0; i < graphPoints.size()-1; i++)
{
        float x1 = graphPoints.at(i).first;
        float y1 = graphPoints.at(i).second;
        float x2 = graphPoints.at(i+1).first;
        float y2 = graphPoints.at(i+1).second;
        glBegin(GL_POLYGON);
        glVertex3f(x1, y1, -j * GRID_SIZE);
        glVertex3f(x1, y1, -j * GRID_SIZE - GRID_SIZE);
        glVertex3f(x2, y2, -j * GRID_SIZE - GRID_SIZE);
        glVertex3f(x2, y2, -j * GRID_SIZE);
        glEnd();
}
```

Пример работы программы



Выводы.

В результате выполнения лабораторной работы была разработана программа для создания анимации на трехмерной поверхности по данным законам изменения координат и изучена работа с шейдерами в OpenGL.

ПРИЛОЖЕНИЕ А.

ИСХОДНЫЙ КОД ПРОГРАММЫ WIDGET.CPP

```
#include "widget.h"
#include <QDebug>
#define binomial(n, i) (factorial(n) / (factorial(i) * factorial(n - i)))


float p1=1.0f, p2=1.0f, p3=1.0f, p4=1.0f, p5=1.0f,p6=1.0f;
float g_Weight[] = {p1, p2, p3, p4, p5, p6};

Widget::Widget(QWidget* parent) : QGLWidget(parent)
{

    isDrag = false;
    time = 0.0;
    d = 4;
}


void Widget::initializeGL()
{
    //qglClearColor(Qt::white);
    glClearColor(0.5, 0.5, 0.5, 0);
    glPointSize(POINT_SIZE);
    glEnable(GL_DEPTH_TEST);
    glEnable(GL_BLEND);

    //compile shaders
    program = new QGLShaderProgram(this);
    program->addShaderFromSourceFile(QGLShader::Vertex, ":/vshader.vsh");
    program->addShaderFromSourceFile(QGLShader::Fragment, ":/fshader.fsh");
    program->link();
    program->bind();

    //set projection matrix
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    glFrustum(-1, 1, -1, 1, 1, 1);

    //set model-view matrix
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    glTranslatef(0.3, 0.3, -2);

    //get model-view-projection matrix for vertex shader
    glPushMatrix();
    float* mvp = new float[16];
```

```
    glGetFloatv(GL_PROJECTION_MATRIX, mvp);
    glMultMatrixf(mvp);
    glGetFloatv(GL_MODELVIEW_MATRIX, mvp);
    glPopMatrix();
    QMatrix4x4 mvpMatrix(mvp);
    program->setUniformValue("mvp", mvpMatrix);

    surfaceColor = new QVector4D(0.917, 0.6, 0.454, 1);

    QTimer *timer = new QTimer(this);
    connect(timer, SIGNAL(timeout()), this, SLOT(refresh()));
    timer->start(0);

//    glFrontFace(GL_CW);
//    glCullFace(GL_FRONT);
//    glEnable(GL_CULL_FACE);
//    glEnable(GL_DEPTH_TEST);

//    QMatrix4x4 mMatrix;
//    QMatrix4x4 vMatrix;

////    QMatrix4x4 cameraTransformation;
////    cameraTransformation.rotate(25, 0, 1, 0); // mAlpha = 25
////    cameraTransformation.rotate(25, 1, 0, 0);  // mBeta = 25

////    QVector3D cameraPosition = cameraTransformation * QVector3D(0, 0, 10);
////   QVector3D cameraUpDirection = cameraTransformation * QVector3D(0, 1, 10);

////    vMatrix.lookAt(cameraPosition, QVector3D(0, 0, 0), cameraUpDirection);
//     vMatrix.lookAt(QVector3D(0.0f, 0.0f, 3.0f), QVector3D(0.0f, 0.0f, 0.0f),
QVector3D(0.0f, 1.0f, 0.0f));
}

void Widget::resizeGL(int nWidth, int nHeight)
{
    glViewport(0, 0, nWidth, nHeight);
}

void Widget::paintGL()
{

    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glBlendFunc(GL_ONE, GL_ONE_MINUS_SRC_ALPHA);

    //поверхность
    for (int j = 0; j < 20; j++) {
        for (int i = 0; i < graphPoints.size()-1; i++) {
```

```
            float x1 = graphPoints.at(i).first;
            float y1 = graphPoints.at(i).second;
            float x2 = graphPoints.at(i+1).first;
            float y2 = graphPoints.at(i+1).second;

            program->setUniformValue("color", *surfaceColor);

            glBegin(GL_POLYGON);
            glVertex3f(x1, y1, -j * GRID_SIZE);
            glVertex3f(x1, y1, -j * GRID_SIZE - GRID_SIZE);
            glVertex3f(x2, y2, -j * GRID_SIZE - GRID_SIZE);
            glVertex3f(x2, y2, -j * GRID_SIZE);
            glEnd();
        }
    }

    program->setUniformValue("lighting", false);

    //nodes
    program->setUniformValue("color", QColor(0, 31, 1, 1));
//    glBegin(GL_POINTS);
//    for (int i = 0; i < nodes.size(); i++)
//        glVertex3f(nodes.at(i).first, nodes.at(i).second, -0.4901);
//    glEnd();

}


void Widget::recountGraphPoints() {
    QList<point> newGraphPoints;



    point p1 = nodes[0];
    for(double t = 0.0;t <= 1.0; t += 0.0001)
    {
        newGraphPoints.append(p1);
        point p2 = bezier(t);
        p1 = p2;
    }
    this->graphPoints = newGraphPoints;
}

void Widget::mousePressEvent(QMouseEvent *e) {

    float x = (float) e->x() / this->width() * 2 - 1;
    float y = - (float) e->y() / this->height() * 2 + 1;
    float pointSize = (float) 1 / this->width() * (POINT_SIZE + 10);
```

```
    for (int i = 0; i < nodes.size(); i++)
        if ((fabs(x - nodes.at(i).first) < pointSize) &&
                (fabs(y - nodes.at(i).second) < pointSize)) {
            isDrag = true;
            dragNodeIndex = i;
        }
}

void Widget::mouseReleaseEvent(QMouseEvent *e) {
    isDrag = false;
}

void Widget::mouseMoveEvent(QMouseEvent *e) {
    if (isDrag) {
        float x = (float) e->x() / this->width() * 2 - 1;
        float y = - (float) e->y() / this->height() * 2 + 1;
        nodes[dragNodeIndex].first = x;
        nodes[dragNodeIndex].second = y;
        recountGraphPoints();
        update();
    }
}

void Widget::refresh() {
    time += 0.1;
    if (time > 2 * M_PI) time = 0.0;
    program->setUniformValue("time", time);
    update();
}

Widget::~Widget() {

}

double Widget::bernstein(int i, int n, double t)
{
    return binomial(n, i) * pow(t, i) * pow(1 - t, n - i);
}

point Widget::bezier(double t)
{
    const int n = nodes.size() - 1;
    point sum (0,0);

    for (int i = 0; i <= n; i++)
    {
        sum.first+=nodes.at(i).first*bernstein(i, n, t);
        sum.second+=nodes.at(i).second*bernstein(i, n, t);
```

```
    }
    return sum;
}

double Widget::factorial(double n)
{
    double fact = 1.0;
    for (int i = 2; i <= n; i++)
        fact *= i;
    return fact;
}


float Widget::B(float x, int n, int d){
    if(d == 0)
    {
        if(knots[n] <= x && x < knots[n+1])
        {
            return 1.0f;
        }
        return 0.0f;
    }
    float a = B(x,n,d-1);
    float b = B(x,n+1,d-1);
    float c = 0.0f, e = 0.0f;

    if(a != 0.0f)
    {
        c = (x - knots[n]) / (knots[n+d] - knots[n]);
    }
    if(b != 0)
    {
        e = (knots[n+d+1] - x) / (knots[n+d+1] - knots[n+1]);
    }
    return (a*c + b*e);
}


void Widget::CrKnotVector(){
    QVector <float> knots;

    for(int i = 0; i < d; i++)
    {
        knots.append(0.0f);
    }

    for(int i=0; i < points.length()-d+1; i++)
    {
```

```
        knots.append((float)i);
    }

    for(int i=0; i< d;i++)
    {
        knots.append((float)(points.length()-d));
    }

    this->knots=knots;
}

void Widget::Bspline(){
//    QList<point> newGraphPoints;
//    point p1 = nodes[0];
//    for(double t = 0.0;t <= 1.0; t += 0.0001)
//    {
//        newGraphPoints.append(p1);
//        point p2 = bezier(t);
//        p1 = p2;
//    }
//    this->graphPoints = newGraphPoints;

    if(d+1 >= points.length())
    {
        return;
    }

    glColor3d(1,0.8,0);
    glBegin(GL_LINE_STRIP);

    CrKnotVector();

    float xmin=knots[0];
    float xmax=knots.last();

    float delta = xmax - xmin;
    float step = delta/300;

    for(float t = xmin; t < xmax; t += step){
        float x = 0.0f, y = 0.0f;

        for(int i = 0; i < points.length(); i++){
            x+=B(t,i,d) * points[i].x() * g_Weight[i];
            y+=B(t,i,d) * points[i].y() * g_Weight[i];
        }

        glVertex2f(x,y);
    }
```

```
    glVertex2f(points.last().x(),points.last().y());
    glEnd();
}
```