**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра математического обеспечения и применения ЭВМ**

отчет

**По лабораторной работе № 5**

**по дисциплине «Компьютерная графика»**

Тема: **Расширения OpenGL, программируемый графический конвейер. Шейдеры.**

|  |  |  |
| --- | --- | --- |
| Студент гр. 0303 |  | Болкунов В.О. |
| Преподаватель |  | Герасимова Т.В. |

Санкт-Петербург

2023

**Задание**

Применение фильтра «Волновой эффект» при помощи шейдеров OpenGL.

**Основные теоретические положения**

**Шейдер** являетсячастью шейдерной программы, которая заменяет собой часть графического конвейера видеокарты. Тип шейдера зависит от того, какая часть конвейера будет заменена. Каждый шейдер должен выполнить свою обязательную работу, т. е. записать какие-то данные и передать их дальше по графическому конвейеру.

**Шейдерная программа** – это небольшая программа, состоящая из шейдеров (вершинного, фрагментного и др.), и выполняющаяся на графическом процессоре видео-карты.

Существует пять мест в графическом конвейере, куда могут быть встроены шейдеры. Соответственно шейдеры делятся на типы:

* вершинный шейдер (vertex shader);
* геометрический шейдер (geometric shader);
* фрагментный шейдер (fragment shader);
* два тесселяционных шейдера (tesselation), отвечающие за два разных этапа тесселяции (они доступны в OpenGL 4.0 и выше).

Дополнительно существуют вычислительные (compute) шейдеры, которые выполняются независимо от графического конвейера.

Разные шаги графического конвейера накладывают разные ограничения на работу шейдеров. Поэтому у каждого типа шейдеров есть своя специфика.

Для применения требуемого волнового эффекта потребуется реализовать вершинный шейдер, смещающий координаты вершин на некоторое расстояние по оси Y по гармоническому закону в зависимости от X координаты вершины. Фрагментному же шейдеру достаточно применять цвет вершины переданный ему из вершинного шейдера.

**Выполнение работы.**

В качестве основы изображения для применения шейдера была взята программа из лабораторной работы №3 – алгебраический фрактал.

В вершинном шейдере объявлено две **uniform** переменных *freq* и *amplitude* соответственно позволяющие регулировать частоту и амплитуду волнового эффекта. Также в вершинном шейдере присутствует **varying** переменная *vertex\_color*, позволяющая передавать цвет вершины в фрагментный шейдер.

В итоге формула смещения вершины в шейдере выглядит следующим образом:

Где – соответственно исходные координаты вершины, а – полученные в результате применения шейдера. – амплитуда волны, – частота.

Ниже представлено содержимое функции **main** фрагментного шейдера

vec4 pos = gl\_Vertex;  
pos.y = pos.y + sin(pos.x \* PI \* freq) \* amplitude;  
gl\_Position = gl\_ModelViewProjectionMatrix \* pos;  
vertex\_color = gl\_Color;

Фрагментный шейдер выполняет единственную функцию – устанавливает цвет фрагмента равным цвету вершины *vertex\_color*.

Код функции фрагментного шейдера состоит из строчки:

gl\_FragColor = vertex\_color;

Для выполнения работы были использованы язык Python 3.10 и библиотеки PyQt6, numpy и PyOpenGL.

Подключение графической библиотеки осуществляется с помощью виджета QOpenGLWidget

В нашем случае был создан виджет GLWidget наследуемый от данного класса, в котором с помощью переопределённых методов **initializeGL и paintGL** осуществляется соответственно подготовка кадра и отрисовка изображения.

В переопределённом методе **initializeGL** осуществляется компиляция шейдерной программы (модуль **shaders** представлен ниже) и установка её для использования в графическом конвейере.

Ниже представлен фрагмент подключения шейдера в этом методе:

self.shader = shaders.createWaveProgram()gl.glUseProgram(self.shader)self.freqLocation = gl.glGetUniformLocation(self.shader, "freq")  
self.amplitudeLocation = gl.glGetUniformLocation(self.shader, "amplitude")

Сама отрисовка для гибкости использования осуществляется задаваемой функцией в поле **function**. В переопределённом методе resizeGL отслеживается изменение размера окна, устанавливается Viewport и посылается сигнал об изменении размера области отрисовки.

Задавав функцию рисования и вызвав метод **update** у GLWidget можно добиться рисования любых объектов в соответствии с заданной функцией.

В модуле drawing описаны функции для генерации и рисования фрактального изображения, перечислим содержимое данного модуля:

* Нормализованные векторы точек шестиугольника используемые для генерации шестиугольников на заданных уровнях

hexagon = np.array([np.array([np.cos(t), np.sin(t)]) for t in np.linspace(0, 2 \* np.pi, 7)[:-1] + np.pi / 6])

* Генерация n шестиугольников с радиусом r (расстоянием между соседними уровнями) по описанным выше формулам нахождения точек

def generate(n, r):

* Цветовая палитра фрактала

colors = [[0.06, 0.28, 0.66], [0.78, 0.0, 0.49], [0.66, 0.94, 0.0], [1.0, 0.65, 0.0]]

* Нормализованные векторы точек окружности для рисования окружностей в вершинах шестиугольников

circleVecs = [np.array([np.cos(t), np.sin(t)]) for t in np.linspace(0, 2 \* np.pi, 50)]

* Рисование окружности радиуса r в точке p и заданным цветом

def drawCircle(r, p, color):

* Рисование линий между точками сгенерированных шестиугольников по описанному алгоритму соединения точек

def drawLines(dots):

В модуле **shaders** реализованы функции для работы с шейдерами

* Чтение содержимого файла шейдера

def readShader(file):  
 return open(file, 'r').read()

* Создание и компиляция шейдера из файла

def createShader(shader\_type, file):  
 shader = gl.glCreateShader(shader\_type)  
 gl.glShaderSource(shader, readShader(file))  
 gl.glCompileShader(shader)  
 return shader

* Создание и линковка шейдерной программы

def createWaveProgram():  
 vertex = createShader(gl.GL\_VERTEX\_SHADER, './wave.vert')  
 fragment = createShader(gl.GL\_FRAGMENT\_SHADER, './wave.frag')  
 waveProgram = gl.glCreateProgram()  
 gl.glAttachShader(waveProgram, vertex)  
 gl.glAttachShader(waveProgram, fragment)  
 gl.glLinkProgram(waveProgram)  
 return waveProgram

Управление приложением осуществляется с помощью виджета **ControlPanel**, который содержит в себе два ползунка для регулирования количества уровней фрактала и для управления масштабом отображения фрактала (расстоянием между уровнями); и два ползунка для управления параметрами шейдера: для настройки частоты и амплитуды применяемого волнового эффекта.

Элементы управления и графический виджет объединены компонентом **MainWindow** (наследуемом от QMainWindow), в нём происходит связывание событий интерфейса управления с обновлениями изображения.

**Тестирование**

Возможные изображения полученные программой представлены на рисунках 1, 2 и 3.

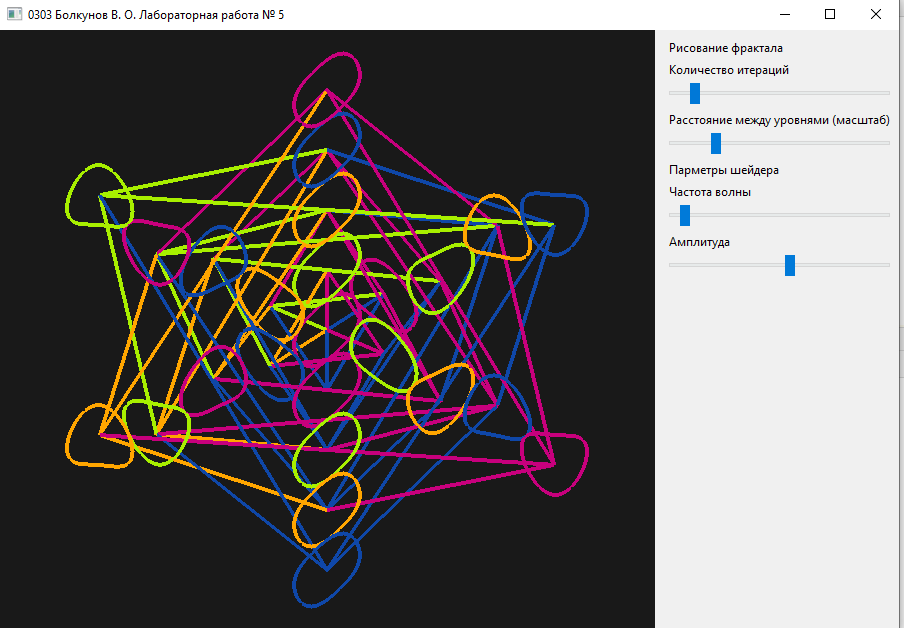


Рисунок 1: волновой эффект с небольшой частотой

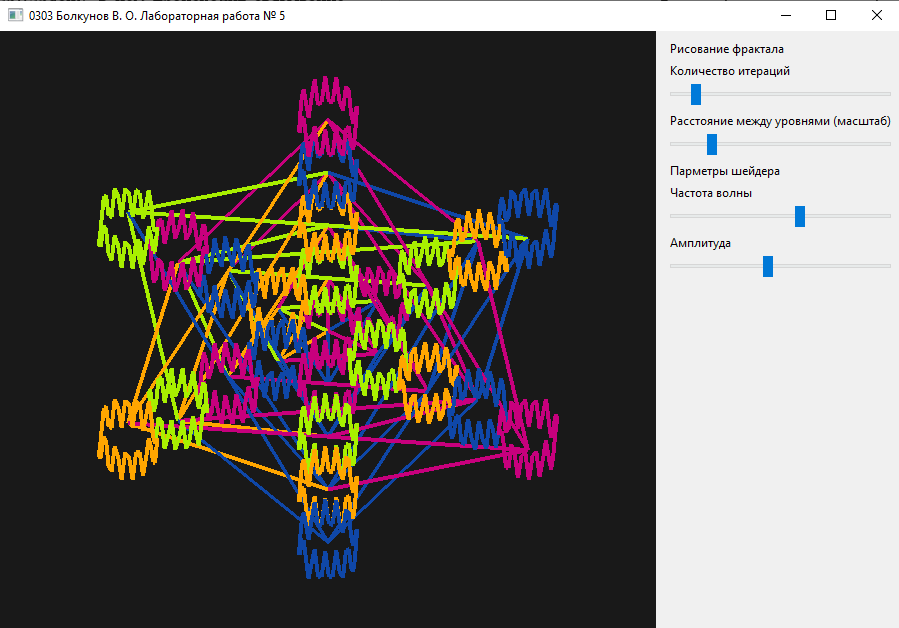


Рисунок 2: волновой эффект с большей частотой

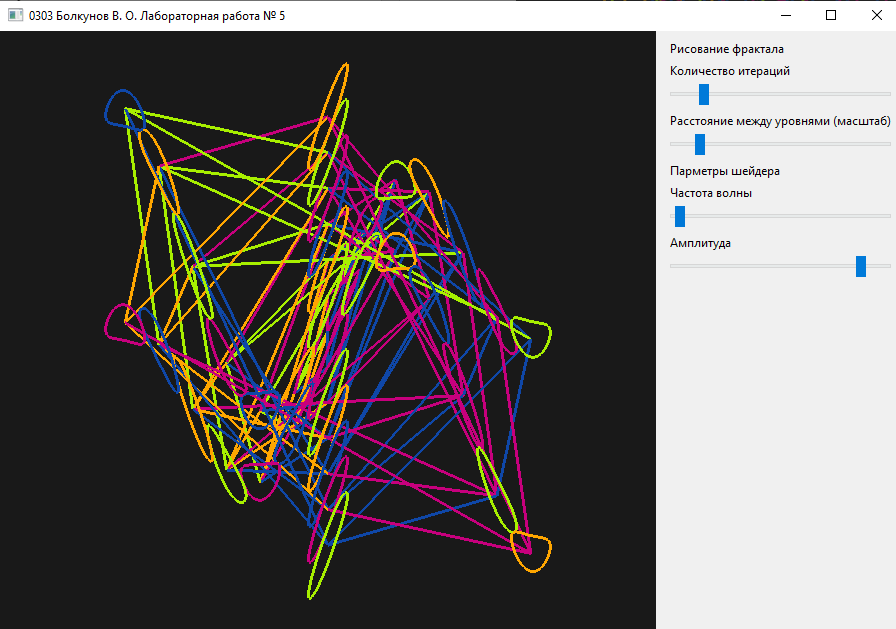
****

Рисунок 3: волновой эффект с большой амплитудой

**Выводы:**

В ходе лабораторной работы были изучены принципы работы с языком шейдеров в OpenGL – GLSL; на основе предыдущей лабораторной работы была создана шейдерная программа, применяющая фильтр волнового эффекта к исходному изображению.

**ПРИЛОЖЕНИЕ А. ИСХОДНЫЙ КОД**

***Файл main.py***

import sys  
  
from PyQt6.QtWidgets import QApplication  
  
from MainWindow import MainWindow  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 app = QApplication(sys.argv)  
 w = MainWindow()  
 w.show()  
 sys.exit(app.exec())

***Файл GLWidget.py***

from OpenGL import GL as gl  
from PyQt6 import QtCore  
from PyQt6.QtOpenGLWidgets import QOpenGLWidget  
  
import shaders  
  
  
*# Виджет OpenGL*class GLWidget(QOpenGLWidget):  
 viewPortResized = QtCore.pyqtSignal((int, int))  
  
 def \_\_init\_\_(self, parent=None):  
 super().\_\_init\_\_(parent)  
 *# Функция вызываемая в цикле отрисовки (при обновлениях)* self.function = None  
 self.shader = None  
  
 def resizeGL(self, w: int, h: int) -> None:  
 gl.glViewport(0, 0, w, h)  
 self.viewPortResized.emit(w, h)  
  
 *# Функция вызываемая перед любым обновлением* def initializeGL(self):  
 *# Заливка кадра* gl.glClearColor(0.1, 0.1, 0.1, 1)  
 *# Очистка буферов (цвета и глубины)* gl.glClear(gl.GL\_COLOR\_BUFFER\_BIT | gl.GL\_DEPTH\_BUFFER\_BIT)  
 *# Создание программы шейдера* self.shader = shaders.createWaveProgram()  
 *# Подключение шейдера* gl.glUseProgram(self.shader)  
 *# Указатели для uniform параметров* self.freqLocation = gl.glGetUniformLocation(self.shader, "freq")  
 self.amplitudeLocation = gl.glGetUniformLocation(self.shader, "amplitude")  
  
 *# Функция вызываемая при обновлении (посредством update или при изменении размеров)* def paintGL(self):  
 *# Вызов рендер-функции* if self.function is not None:  
 self.function()  
  
 *# Изменить частоту волны в шейдере* def setFreq(self, freq: float):  
 if self.shader is not None:  
 gl.glUniform1f(self.freqLocation, freq)  
  
 *# Изменить амплитуду в шейдере* def setAmplitude(self, amplitude: float):  
 if self.shader is not None:  
 gl.glUniform1f(self.amplitudeLocation, amplitude)

***Файл drawing.py***

from OpenGL import GL as gl  
import numpy as np  
  
*# Нормализованные векторы точек шестиугольника*hexagon = np.array([  
 np.array([np.cos(t), np.sin(t)])  
 for t in np.linspace(0, 2 \* np.pi, 7)[:-1] + np.pi / 6  
])  
  
  
*# Генерация n уровней вложенных шестиугольников на расстоянии r*def generate(n, r):  
 return [i \* r \* hexagon for i in range(n)]  
  
  
*# Палитра цветов*colors = [  
 [0.06, 0.28, 0.66],  
 [0.78, 0.0, 0.49],  
 [0.66, 0.94, 0.0],  
 [1.0, 0.65, 0.0]  
]  
  
*# Векторы точек окружности*circleVecs = [np.array([np.cos(t), np.sin(t)]) for t in np.linspace(0, 2 \* np.pi, 50)]  
  
  
*# Рисование окружности радиуса r в точке p*def drawCircle(r, p, color):  
 gl.glBegin(gl.GL\_LINE\_STRIP)  
 gl.glColor3dv(color)  
 for a in circleVecs:  
 gl.glVertex2dv(p + a \* r)  
 gl.glEnd()  
  
  
*# Рисованиее линий списка уровней шестиугольников*def drawLines(dots):  
 gl.glBegin(gl.GL\_LINES)  
 *# Уровни* for i in range(1, len(dots)):  
 *# Точки в уровне* for j in range(6):  
 gl.glColor3dv(colors[j % 4])  
 *# Соединение точки с двумя другими на своём уровне* for k in range(1, 3):  
 gl.glVertex2dv(dots[i][j])  
 gl.glVertex2dv(dots[i][j - 2 \* k])  
 *# Соединение точки с двумя точками предыдущего уровня* gl.glVertex2dv(dots[i][j])  
 gl.glVertex2dv(dots[i - 1][j - 1])  
 gl.glVertex2dv(dots[i][j])  
 gl.glVertex2dv(dots[i - 1][(j + 1) % 6])  
 gl.glEnd()

***Файл ControlPanel.py***

from PyQt6.QtCore import Qt  
from PyQt6.QtWidgets import QWidget, QVBoxLayout, QSlider, QLabel  
  
  
*# Виджет панели управления*class ControlPanel(QWidget):  
 def \_\_init\_\_(self, parent=None):  
 super().\_\_init\_\_(parent)  
 lt = QVBoxLayout(self)  
 self.setLayout(lt)  
  
 drawLabel = QLabel('Рисование фрактала', self)  
 levelsLabel = QLabel('Количество итераций', self)  
 self.iterations = QSlider(Qt.Orientation.Horizontal, self)  
 radiusLabel = QLabel('Расстояние между уровнями (масштаб)', self)  
 self.radius = QSlider(Qt.Orientation.Horizontal, self)  
  
 shaderLable = QLabel('Парметры шейдера', self)  
 waveLable = QLabel('Частота волны', self)  
 self.waves = QSlider(Qt.Orientation.Horizontal, self)  
 amplitudeLable = QLabel('Амплитуда', self)  
 self.amplitude = QSlider(Qt.Orientation.Horizontal, self)  
  
 *#* for i in [  
 drawLabel, levelsLabel, self.iterations, radiusLabel, self.radius,  
 shaderLable, waveLable, self.waves, amplitudeLable, self.amplitude  
 ]:  
 lt.addWidget(i)  
 lt.addStretch()

***Файл MainWindow.py***

from PyQt6 import QtCore  
from PyQt6.QtWidgets import QMainWindow, QSplitter  
from ControlPanel import ControlPanel  
from GLWidget import GLWidget  
from drawing import \*  
  
  
*# Главное окно*class MainWindow(QMainWindow):  
 def \_\_init\_\_(self):  
 super().\_\_init\_\_()  
 self.setWindowTitle("0303 Болкунов В. О. Лабораторная работа № 5")  
 self.control = ControlPanel(self)  
 self.glwidget = GLWidget(self)  
 sp = QSplitter(self)  
 sp.addWidget(self.glwidget)  
 sp.addWidget(self.control)  
 sp.setStretchFactor(0, 1)  
  
 self.setCentralWidget(sp)  
 self.resize(900, 600)  
  
 *# Задаём рендер-функцию* self.glwidget.function = self.renderFunction  
  
 self.radiusFraction = 1000  
 self.lengthFraction = 100  
 self.amplitudeFraction = 1000  
  
 self.control.radius.setMaximum(self.radiusFraction)  
 self.control.iterations.setMaximum(50)  
 self.control.iterations.setTickInterval(1)  
 self.control.amplitude.setMaximum(500)  
 self.control.amplitude.setMinimum(-500)  
 self.control.waves.setMaximum(10000)  
  
 self.control.radius.valueChanged.connect(self.redraw)  
 self.control.iterations.valueChanged.connect(self.redraw)  
 self.control.waves.valueChanged.connect(self.redraw)  
 self.control.amplitude.valueChanged.connect(self.redraw)  
  
 self.control.waves.setValue(500)  
 self.control.amplitude.setValue(50)  
  
 self.control.radius.setValue(200)  
 self.control.iterations.setValue(5)  
 self.redraw()  
  
 def renderFunction(self):  
 self.glwidget.setFreq(self.control.waves.value() / self.lengthFraction)  
 self.glwidget.setAmplitude(self.control.amplitude.value() / self.amplitudeFraction)  
  
 radius = self.control.radius.value() / self.radiusFraction  
 *# Размытие цвета между вершинами* gl.glShadeModel(gl.GL\_SMOOTH)  
 *# Ширина линий* gl.glLineWidth(1 + 15 \* radius)  
 *# Генерация шестиугольников* dots = generate(self.control.iterations.value(), radius)  
 *# Рисование их линий* drawLines(dots)  
 *# Рисование окружностей в вершинах шестиугольников* for i in range(len(dots)):  
 for p in range(len(dots[i])):  
 drawCircle(radius / 2, dots[i][p], colors[(i + p) % 4])  
  
 *# Вызов обновления изображения* @QtCore.pyqtSlot()  
 def redraw(self):  
 self.glwidget.update()

***Файл shaders.py***

from OpenGL import GL as gl  
  
  
*# Загрузка файла шейдера*def readShader(file):  
 return open(file, 'r').read()  
  
  
*# Создание шейдера из файла*def createShader(shader\_type, file):  
 shader = gl.glCreateShader(shader\_type)  
 gl.glShaderSource(shader, readShader(file))  
 gl.glCompileShader(shader)  
 return shader  
  
  
*# Компиляция шейдерной программы*def createWaveProgram():  
 vertex = createShader(gl.GL\_VERTEX\_SHADER, './wave.vert')  
 fragment = createShader(gl.GL\_FRAGMENT\_SHADER, './wave.frag')  
 waveProgram = gl.glCreateProgram()  
 gl.glAttachShader(waveProgram, vertex)  
 gl.glAttachShader(waveProgram, fragment)  
 gl.glLinkProgram(waveProgram)  
 return waveProgram

***Файл wave.vert***

#define PI 3.1415926535897932384626433832795  
  
uniform float freq;  
uniform float amplitude;  
  
varying vec4 vertex\_color;  
  
void main() {  
 vec4 pos = gl\_Vertex;  
 pos.y = pos.y + sin(pos.x \* PI \* freq) \* amplitude;  
 gl\_Position = gl\_ModelViewProjectionMatrix \* pos;  
 vertex\_color = gl\_Color;  
}

***Файл wave.frag***

varying vec4 vertex\_color;  
  
void main() {  
 gl\_FragColor = vertex\_color;  
}