**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 |  | Архипов В. А. |
| Преподаватель |  | Герасимова Т. В. |

Санкт-Петербург

2023

**Цель работы.**

Реализовать требуемый визуальный эффект с помощью шейдера.

**Задание.**

Реализовать фильтр «градиент» (Вариант 18).

**Выполнение работы.**

Работа была выполнена в среде разработки PyCharm на языке программирования Python 3.10. Для реализации пользовательского интерфейса были использованы библиотека PyQt6 и программа Qt Designer, создающая пользовательский интерфейс по построенному в ней макету. Для работы с графикой была использована библиотека PyOpenGL. Подключение графической библиотеки к пользовательскому интерфейсу было осуществлено с помощью виджета QOpenGLWidget из библиотеки PyQt6.

Макет пользовательского интерфейса был разработан в программе Qt Designer (см. рис. 1).

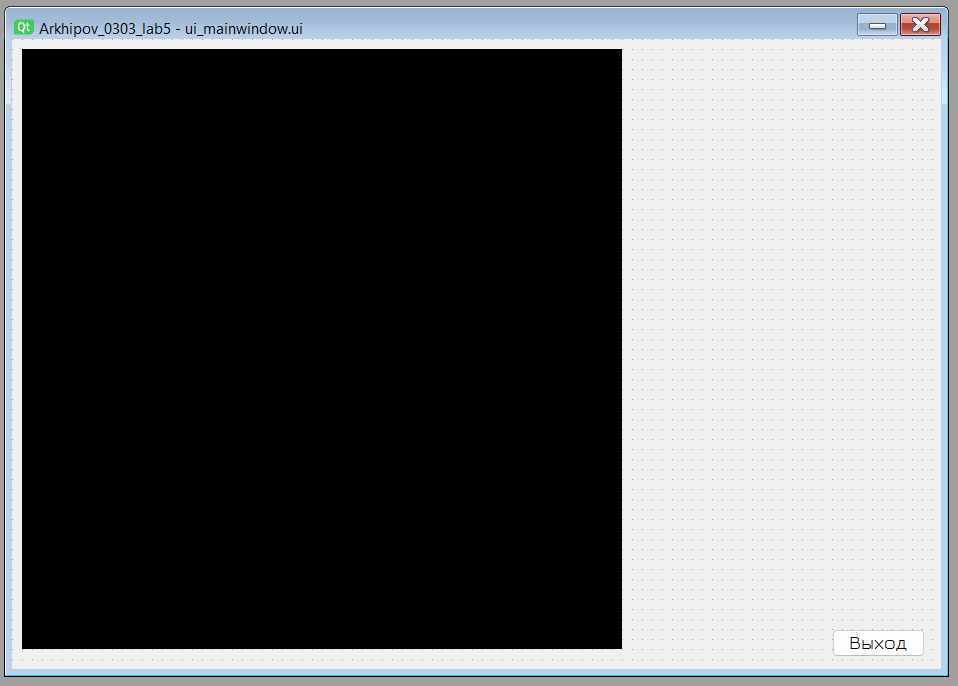


Рисунок – разработанный в Qt Designer макет пользовательского интерфейса

Класс главного окна MainWindow наследуется от базового для PyQt класса QMainWindow и содержит внутри себя виджет glWidget унаследованный от QOpenGLWidget, для отображения графики.

Для того чтобы продемонстрировать градиент, то есть плавный переход цветов, была выбрана фигура треугольник и реализован соответствующий алгоритм, обеспечивающий смешение трёх заданных цветов. Общая идея алгоритма заключается в расчете расстояния между каждой точкой внутри треугольника и всеми его вершинами, эти расстояния служат весами, которые буду участвовать в вычислении цвета для рассматриваемой точки. Теперь рассмотрим алгоритм более детально. На старте у нас есть треугольник , мы знаем координаты его вершин и цвета этих вершин. Первое, что необходимо сделать – это провести из вершин треугольника высоты на противолежащие стороны (см. рис. 2).

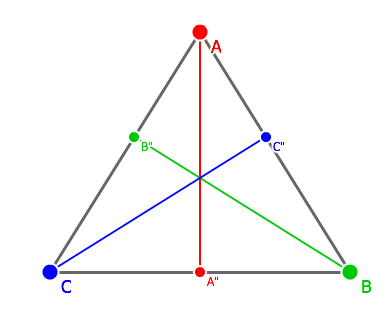


Рисунок – первый шаг алгоритма – построить высоты

Теперь рассмотрим работу алгоритма на одной конкретной точке (см. рис. 3).

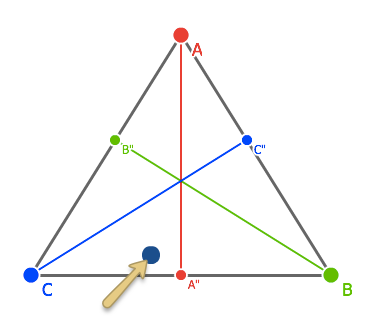


Рисунок – одна точка, для которой будет вычислен цвет

Каждая точка внутри треугольника характеризуется удалённостью от вершин треугольника. Эта информация крайне необходима для вычисления цвета в треугольнике, так как чем дальше точка от какой-то вершины, тем меньше цвета этой вершины она будет содержать, и наоборот, чем ближе точка к вершине, тем большую часть цвета этой вершины она будет содержать. Для измерения степени удалённости точки от определённой вершины будем использовать проекцию рассматриваемой точки на высоту, проведённую из этой самой вершины. Поэтому из рассматриваемой точки опускаем три

высоты (см. рис. 4).

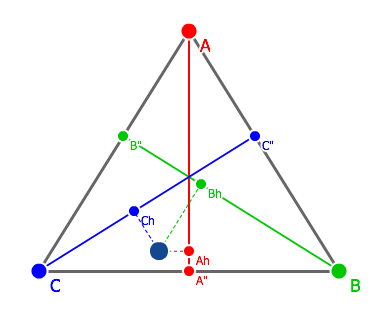


Рисунок – строим проекции рассматриваемой точки на высоту треугольника

Далее нам необходимо вычислить процентное содержание цвета каждой вершины в рассматриваемой точке. Для оценки удалённости точки от вершины можно воспользоваться формулой (Аналогично для остальных двух вершин). Чем больше значение этого коэффициента, тем ближе рассматриваемая точка к вершине, а значит, тем больше процентное содержание цвета в текущей точке.

Когда все коэффициенты будут вычислены, можно приступать к вычислению цвета в точке. Цвет будем вычислять по частям – отдельно посчитаем значения красной, синей и зеленой компонент. Чтобы найти значение красной компоненты итогового цвета воспользуемся формулой , где

– значение красной компоненты цвета вершины .

Аналогичные вычисление проводим для синей и зелёной компонент, после чего собираем их вместе и получаем искомый цвет. В результате вычислений цвета для каждой точки внутри треугольника получаем фигуру с плавными переходами.

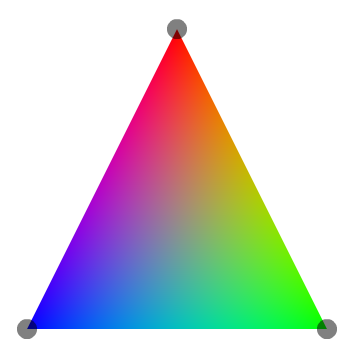


Рисунок – результат работы алгоритма

Теперь перейдем к рассмотрению программной реализации данного алгоритма. Для централизованного доступа к вершинам треугольника и цветам этих вершин были созданы два списка self.vertex и self.colors, хранящие вершины треугольника и их цвета соответственно.

Так как нам предстоит работать с пикселями, а видоизменять вершины нам не нужно, то в работе будет использован только фрагментный шейдер. Код шейдерной программы содержится в файле fragment\_shader.frag. Содержимое этого файла считывается функцией read\_file и передаётся для подключения в программу. Ниже представлен фрагмент кода, отвечающий за подключение шейдерной программы

self.shader = compileProgram(compileShader(read\_file('fragment\_shader.frag'), gl.GL\_FRAGMENT\_SHADER))

В результате своей работы шейдерная программа должна вернуть цвет рассматриваемой точки, представленный вектором из четырех вещественных чисел vec4.

Для осуществления работы шейдерной программы нам необходима информация о вершинах треугольника и их цветах, поэтому в программу передаются 6 uniform переменных: A,B,C – координаты вершин треугольников в формате vec2, A\_color, B\_color, C\_color – цвета этих вершин в формате vec3. Ниже представлен фрагмент кода, реализующий отправку переменных в шейдерную программу.

vertex\_a = gl.glGetUniformLocation(self.shader, 'A')  
vertex\_b = gl.glGetUniformLocation(self.shader, 'B')  
vertex\_c = gl.glGetUniformLocation(self.shader, 'C')  
  
vertex\_a\_color = gl.glGetUniformLocation(self.shader, 'A\_color')  
vertex\_b\_color = gl.glGetUniformLocation(self.shader, 'B\_color')  
vertex\_c\_color = gl.glGetUniformLocation(self.shader, 'C\_color')  
  
  
gl.glUseProgram(self.shader)  
gl.glUniform2f(vertex\_a, \*self.vertex[0])  
gl.glUniform2f(vertex\_b, \*self.vertex[1])  
gl.glUniform2f(vertex\_c, \*self.vertex[2])  
gl.glUniform3f(vertex\_a\_color, \*self.colors[0])  
gl.glUniform3f(vertex\_b\_color, \*self.colors[1])  
gl.glUniform3f(vertex\_c\_color, \*self.colors[2])

Также для упрощения работы была реализована структура line, которая представляет собой линию, проходящую через две точки p1 и p2 представленные вектором из двух вещественных чисел.

struct line{  
 vec2 p1;  
 vec2 p2;  
};

Для получения координат текущего рассматриваемого фрагмента используется встроенная переменная gl\_FragCoord, но она возвращает положение точки не в нормализованных координатах, а в пикселях, что неудобно для дальнейшей работы с точкой. Поэтому была реализована функция transform\_coords, которая превращает пиксельные координаты в нормализованные. Помимо этого, были реализованы следующие вспомогательные функции:

- float get\_distance(vec2 p1, vec2 p2) – вычисляет расстояние между точками p1 и p2.

- vec2 intersection(line l1, line l2) – находит точку пересечения прямых l1 и l2.

- line height(vec2 point, line ln) – находит высоту, опущенную из точки point на прямую ln.

В функции main() шейдерной программы происходит задание сторон треугольника, построение высот, преобразование координат текущей рассматриваемой точки в нормализованные и проекция этой точки на проведённые высоты треугольника. Далее для каждой вершины происходит вычисление весового коэффициента цвета, после чего вычисляются красная, синяя, зеленая компоненты результирующего цвета и передаются в переменную color.

**Демонстрация.**

Ниже приведены примеры работы реализованного приложения.

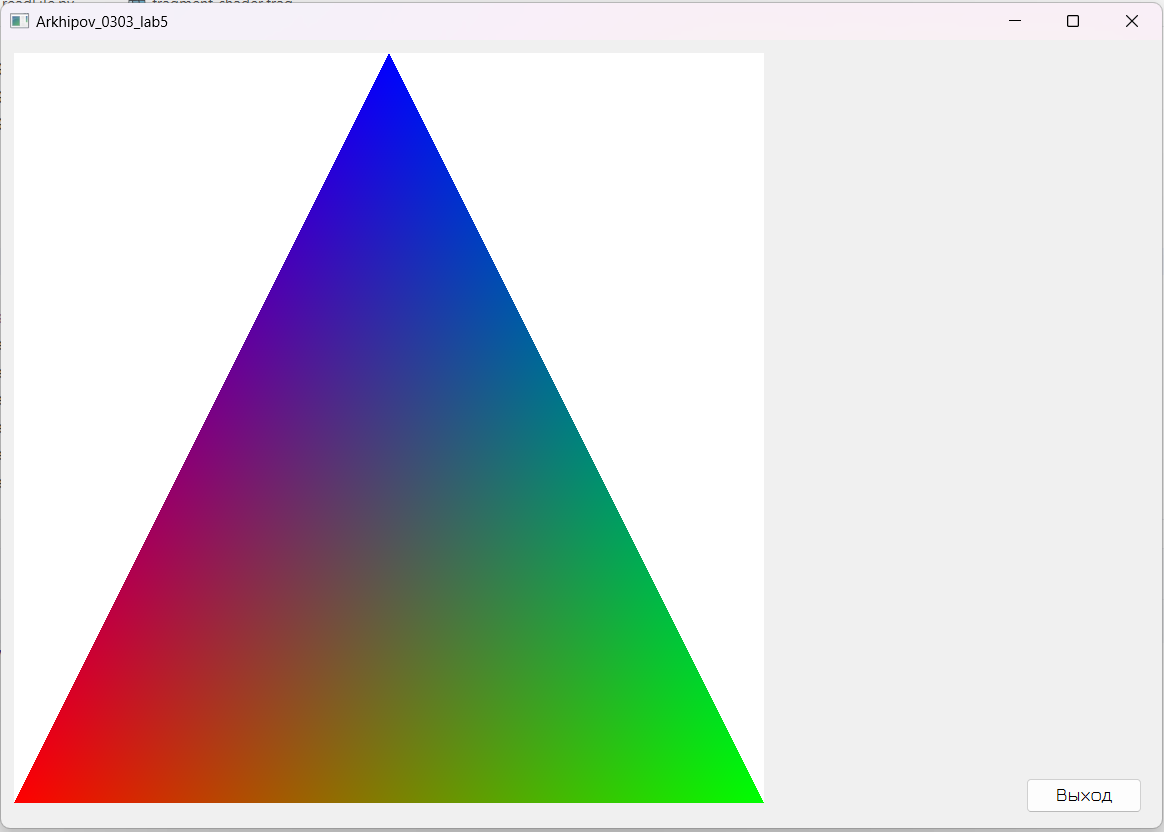


Рисунок – результат работы реализованного шейдера

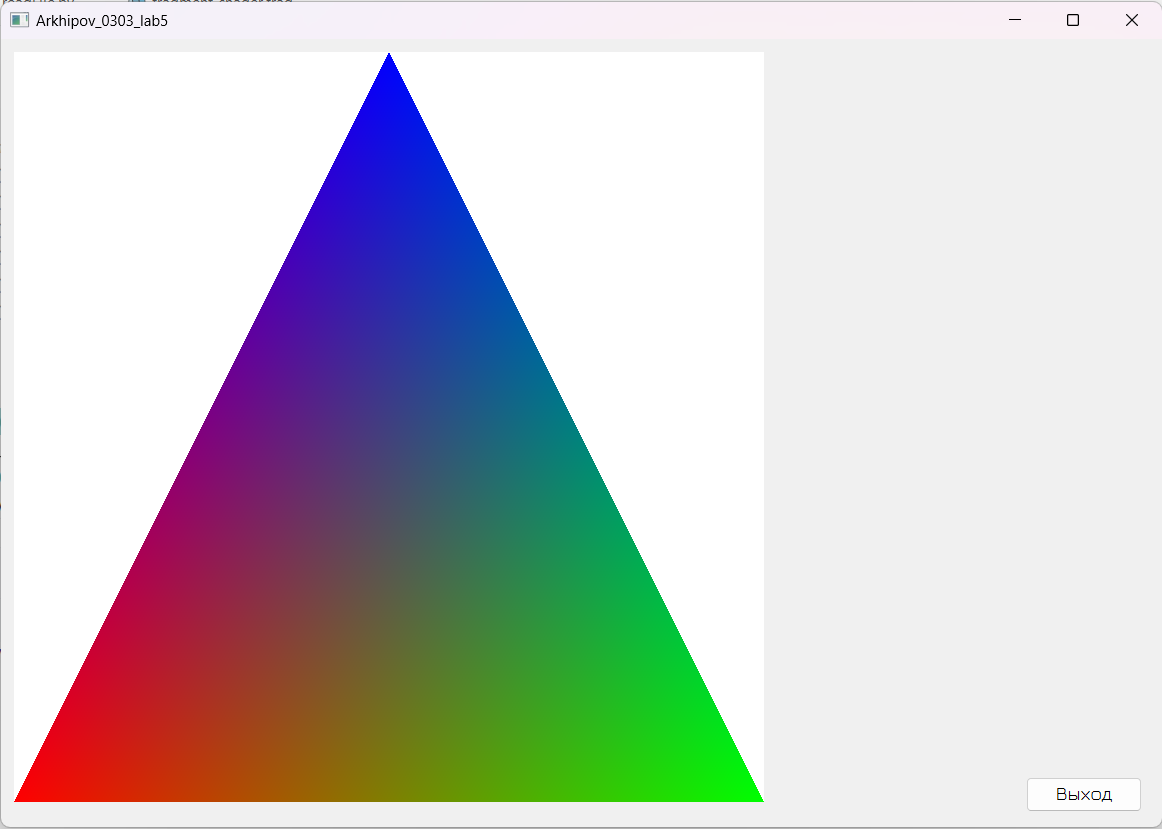


Рисунок – результат работы шейдера, встроенного в OpenGL

Как мы видим, результаты работы реализованного шейдера и шейдера, который стоит внутри OpenGL по умолчанию, идентичны.

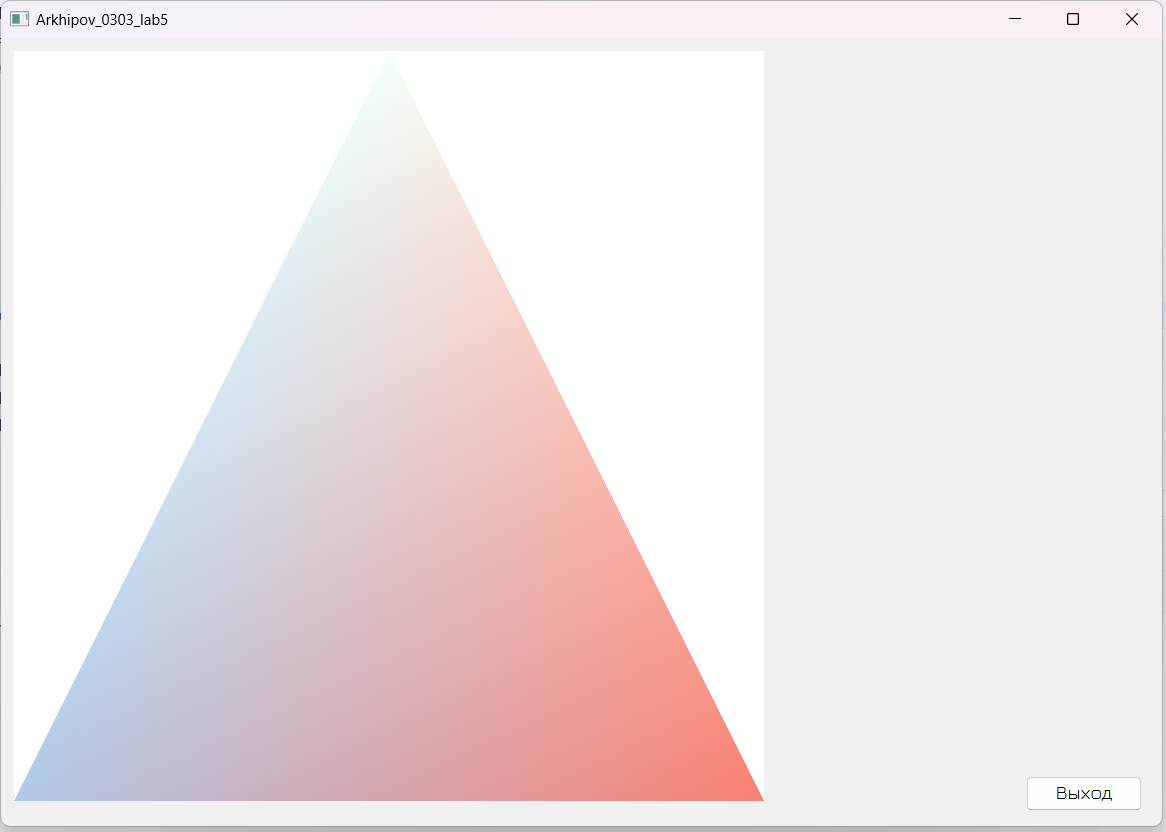


Рисунок – результат работы реализованного шейдера

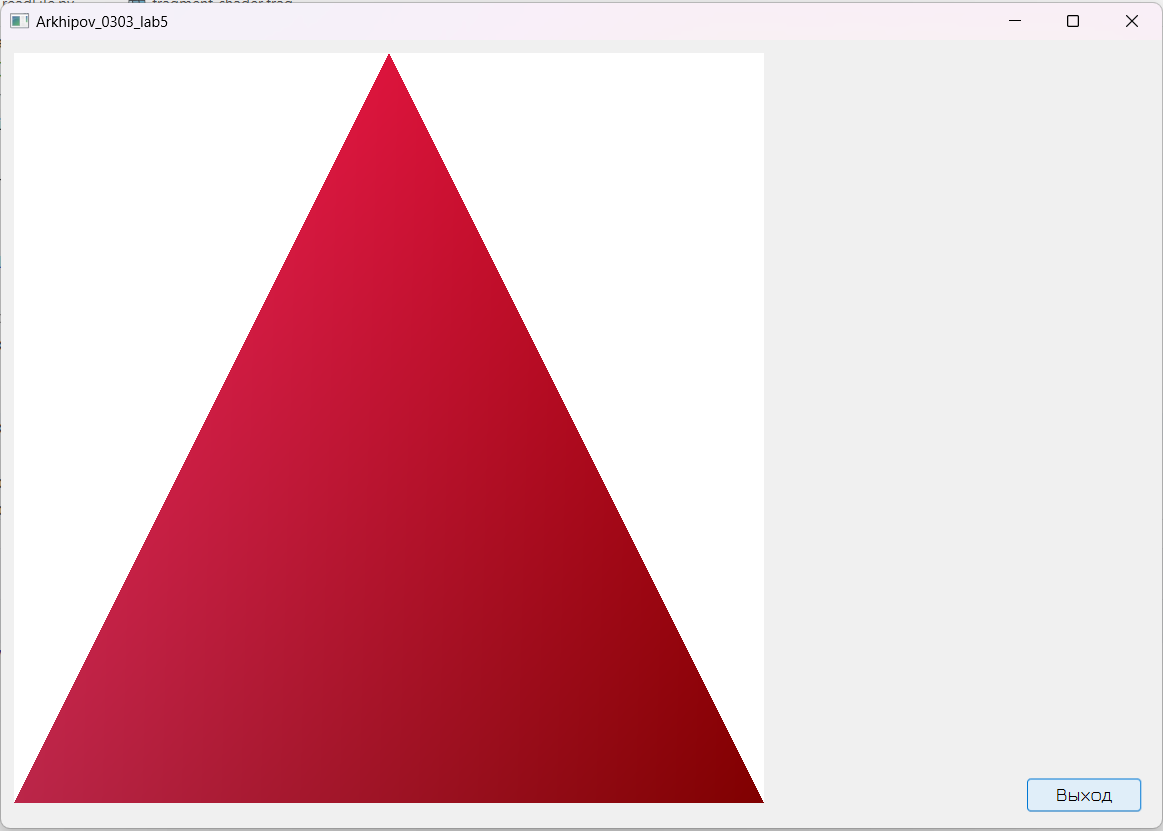


Рисунок – результат работы реализованного шейдера

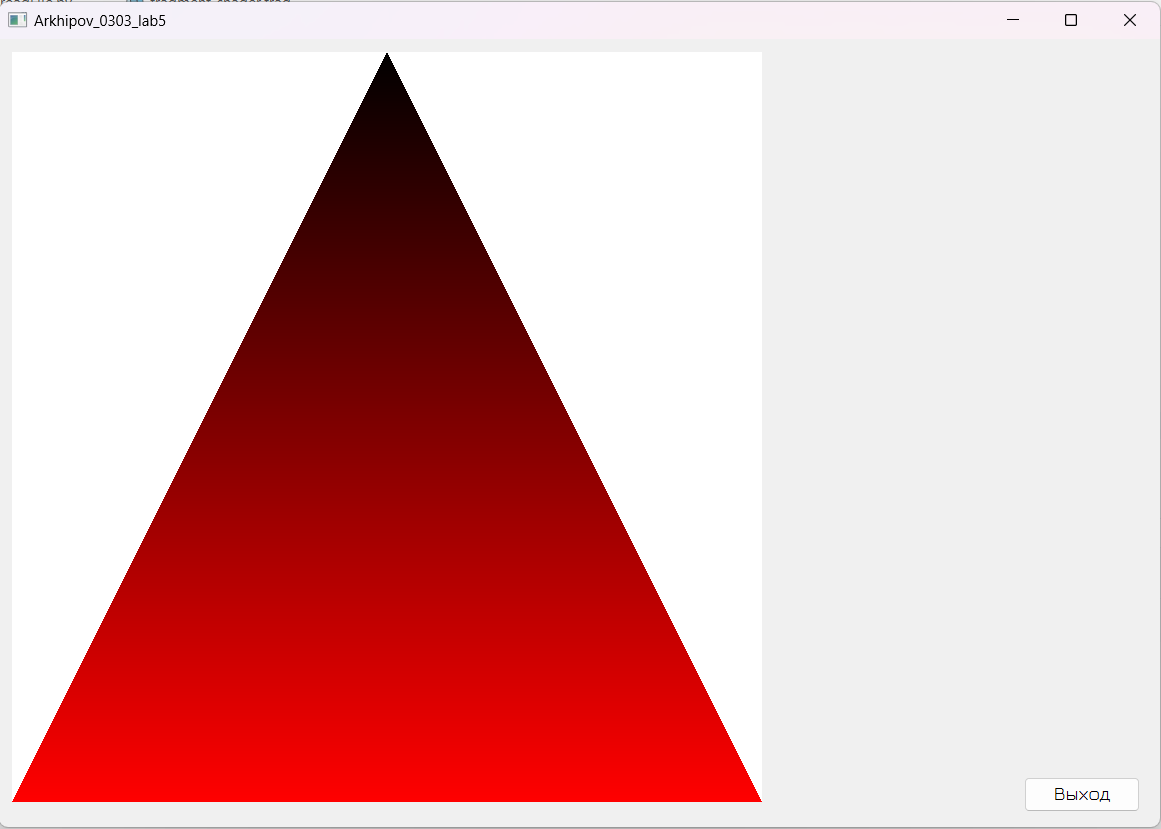


Рисунок – результат работы реализованного шейдера

Разработанный программный код см. в приложении А.

**Вывод.**

В результате выполнения лабораторной работы была разработана программа, реализующая эффект градиента с помощью шейдерной программы.

**ПРИЛОЖЕНИЕ А**

**РАЗРАБОТАННЫЙ ПРОГРАММНЫЙ КОД**

Название файла: main.py

import sys  
from PyQt6.QtWidgets import QApplication  
from MainWindow import MainWindow  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 app = QApplication(sys.argv)  
 main\_window = MainWindow()  
 main\_window.setFixedSize(929, 630)  
 sys.exit(app.exec())

Название файла: glWidget.py

from PyQt6.QtOpenGLWidgets import QOpenGLWidget  
from OpenGL import GL as gl  
from OpenGL.GL.shaders import compileShader, compileProgram  
from PyQt6 import QtCore  
  
*# Функция для чтения содержимого файла*def read\_file(file\_name, descriptor='r'):  
 file = open(file\_name, descriptor)  
 content = file.read()  
 return content  
  
class glWidget(QOpenGLWidget):  
 def \_\_init\_\_(self, main\_window):  
 super().\_\_init\_\_(parent=main\_window.ui.centralwidget)  
 *# Ссылка на родительское окно* self.mw = main\_window  
  
 self.vertex = [[-1, -1],[1, -1],[0, 1]]  
  
 self.demo = 1  
  
 if self.demo == 1:  
 self.colors = [[1, 0, 0],[0, 1, 0],[0, 0, 1]]  
 elif self.demo == 2:  
 self.colors = [[x / 255 for x in [176, 201, 230]], [x / 255 for x in [250, 128, 114]],  
 [x / 255 for x in [245, 255, 250]]]  
 elif self.demo == 3:  
 self.colors = [[x / 255 for x in [187, 38, 73]], [x / 255 for x in [128, 0, 0]],  
 [x / 255 for x in [220, 20, 60]]]  
 else:  
 self.colors = [[1, 0, 0], [1, 0, 0], [0, 0, 0]]  
  
 main\_window.ui.openGLWidget = self  
 main\_window.ui.openGLWidget.setGeometry(QtCore.QRect(10, 10, 600, 600))  
 main\_window.ui.openGLWidget.setObjectName("openGLWidget")  
  
  
 *# Настройка состояния. Вызывается один раз в самом начале* def initializeGL(self):  
 gl.glClearColor(1, 1, 1, 1)  
 gl.glClear(gl.GL\_COLOR\_BUFFER\_BIT | gl.GL\_DEPTH\_BUFFER\_BIT)  
  
 self.shader = compileProgram(compileShader(read\_file('fragment\_shader.frag'), gl.GL\_FRAGMENT\_SHADER))  
  
 vertex\_a = gl.glGetUniformLocation(self.shader, 'A')  
 vertex\_b = gl.glGetUniformLocation(self.shader, 'B')  
 vertex\_c = gl.glGetUniformLocation(self.shader, 'C')  
  
 vertex\_a\_color = gl.glGetUniformLocation(self.shader, 'A\_color')  
 vertex\_b\_color = gl.glGetUniformLocation(self.shader, 'B\_color')  
 vertex\_c\_color = gl.glGetUniformLocation(self.shader, 'C\_color')  
  
  
 gl.glUseProgram(self.shader)  
 gl.glUniform2f(vertex\_a, \*self.vertex[0])  
 gl.glUniform2f(vertex\_b, \*self.vertex[1])  
 gl.glUniform2f(vertex\_c, \*self.vertex[2])  
 gl.glUniform3f(vertex\_a\_color, \*self.colors[0])  
 gl.glUniform3f(vertex\_b\_color, \*self.colors[1])  
 gl.glUniform3f(vertex\_c\_color, \*self.colors[2])  
  
  
 def paintGL(self):  
 gl.glPointSize(5)  
 gl.glBegin(gl.GL\_TRIANGLES)  
 for i in range(3):  
 gl.glColor3fv(self.colors[i])  
 gl.glVertex2fv(self.vertex[i])  
 gl.glEnd()

Название файла: MainWindow.py

import sys  
from PyQt6.QtWidgets import QMainWindow  
from glWidget import glWidget  
from ui\_mainwindow import Ui\_MainWindow  
  
from OpenGL import GL as gl  
from time import sleep  
  
  
class MainWindow(QMainWindow):  
  
 def \_\_init\_\_(self):  
 super().\_\_init\_\_()  
 self.ui = Ui\_MainWindow()  
 self.ui.setupUi(self)  
  
 *# Создание виджета для отображения графики* self.ui.openGLWidget = glWidget(self)  
  
 *# Подписка элементов интерфейса на события* self.ui.pushButton.clicked.connect(self.leave)  
  
  
 self.ui.openGLWidget.update()  
  
  
 *# Демонстрация окна* self.show()  
  
  
 def leave(self):  
 print('You entered "Quit" button')  
 sys.exit()

Название файла: ui\_mainwindow.py

*# Form implementation generated from reading ui file 'ui\_mainwindow.ui'  
#  
# Created by: PyQt6 UI code generator 6.4.2  
#  
# WARNING: Any manual changes made to this file will be lost when pyuic6 is  
# run again. Do not edit this file unless you know what you are doing.*from PyQt6 import QtCore, QtGui, QtWidgets  
  
  
class Ui\_MainWindow(object):  
 def \_\_init\_\_(self):  
 self.openGLWidget = None  
  
 def setupUi(self, MainWindow):  
 MainWindow.setObjectName("MainWindow")  
 MainWindow.resize(929, 630)  
 font = QtGui.QFont()  
 font.setFamily("MS Sans Serif")  
 MainWindow.setFont(font)  
 self.centralwidget = QtWidgets.QWidget(parent=MainWindow)  
 self.centralwidget.setObjectName("centralwidget")  
 self.pushButton = QtWidgets.QPushButton(parent=self.centralwidget)  
 self.pushButton.setGeometry(QtCore.QRect(820, 590, 93, 28))  
 font = QtGui.QFont()  
 font.setFamily("Jura")  
 font.setPointSize(12)  
 self.pushButton.setFont(font)  
 self.pushButton.setObjectName("pushButton")  
 MainWindow.setCentralWidget(self.centralwidget)  
  
 self.retranslateUi(MainWindow)  
 QtCore.QMetaObject.connectSlotsByName(MainWindow)  
  
 def retranslateUi(self, MainWindow):  
 \_translate = QtCore.QCoreApplication.translate  
 MainWindow.setWindowTitle(\_translate("MainWindow", "Arkhipov\_0303\_lab5"))  
 self.pushButton.setText(\_translate("MainWindow", "Выход"))  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 import sys  
 app = QtWidgets.QApplication(sys.argv)  
 MainWindow = QtWidgets.QMainWindow()  
 ui = Ui\_MainWindow()  
 ui.setupUi(MainWindow)  
 MainWindow.show()  
 sys.exit(app.exec())

Название файла: fragment\_shader.frag

#version 430  
  
out vec4 color;  
  
uniform vec2 A;  
uniform vec2 B;  
uniform vec2 C;  
  
uniform vec3 A\_color;  
uniform vec3 B\_color;  
uniform vec3 C\_color;  
  
struct line{  
 vec2 p1;  
 vec2 p2;  
};  
  
*// Трансформация координат*vec2 transform\_coords(vec4 point){  
 float x = point.x;  
 if (x < 375) {  
 x = -((375 - x) / 375);  
 } else {  
 x = (x-375)/375;  
 }  
  
 float y = point.y;  
 if (y<375){  
 y = -(375-y)/375;  
 } else {  
 y = (y-375)/375;  
 }  
  
 return vec2(x, y);  
}  
  
*// Расстояние между двумя точками*float get\_distance(vec2 p1, vec2 p2){  
 return sqrt(pow((p1.x - p2.x), 2.0) + pow((p1.y - p2.y), 2.0));  
}  
  
*// Пересечение прямых l1 и l2*vec2 intersection(line l1, line l2){  
 float d = (l2.p2.y - l2.p1.y) \* (l1.p2.x - l1.p1.x) - (l2.p2.x - l2.p1.x) \* (l1.p2.y - l1.p1.y);  
 float a = (l2.p2.x - l2.p1.x) \* (l1.p1.y - l2.p1.y) - (l2.p2.y - l2.p1.y) \* (l1.p1.x - l2.p1.x);  
  
 float x0 = l1.p1.x + a \* (l1.p2.x-l1.p1.x)/d;  
 float y0 = l1.p1.y + a \* (l1.p2.y-l1.p1.y)/d;  
  
 return vec2(x0,y0);  
}  
  
*// Высота из точки point на прямую ln*line height(vec2 point, line ln){  
 vec2 dir = vec2(0,0);  
 dir.x = ln.p2.x - ln.p1.x;  
 dir.y = ln.p2.y - ln.p1.y;  
  
 vec2 point2 = vec2(0,0);  
 if(dir.y != 0){  
 float temp = dir.x\*point.x + dir.y \* point.y;  
 point2.x = point.x != 1 ? 1 : 2;  
 point2.y = (-dir.x \* point2.x + temp) / dir.y;  
 } else {  
 point2.x = point.x;  
 point2.y = ln.p1.y;  
 }  
  
 return line(point, intersection(ln, line(point, point2)));  
}  
  
  
void main()  
{  
 line AC = line(A, C);  
 line AB = line(A, B);  
 line CB = line(C, B);  
 vec2 O = transform\_coords(gl\_FragCoord);  
  
 *// Высота из точки A на сторону BC* line AH = height(A, CB);  
 line O\_AH = height(O, AH);  
 float pA = 1 - get\_distance(A, O\_AH.p2)/ get\_distance(A, AH.p2);  
  
 *// Высота из точки C на сторону AB* line CH = height(C, AB);  
 line O\_CH = height(O, CH);  
 float pC = 1- get\_distance(C, O\_CH.p2)/ get\_distance(C, CH.p2);  
  
 *// Высота из точки B на сторону AC* line BH = height(B,AC);  
 line O\_BH = height(O, BH);  
 float pB = 1 - get\_distance(B, O\_BH.p2)/ get\_distance(B, BH.p2);  
  
 float red = (A\_color.r \* pA + B\_color.r \* pB + C\_color.r \* pC);  
 float green = (A\_color.g \* pA + B\_color.g \* pB + C\_color.g \* pC);  
 float blue = (A\_color.b \* pA + B\_color.b \* pB + C\_color.b \* pC);  
  
 color = vec4(red, green, blue, 1);  
}