МИНОБРНАУКИ РОССИИ
Федеральное государственное бюджетное образовательное учреждение
высшего образования
**«САРАТОВСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ
ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ИМЕНИ Н. Г. ЧЕРНЫШЕВСКОГО»**
Кафедра математической кибернетики и компьютерных наук

РАЗРАБОТКА ГРАФИЧЕСКОГО ИНТЕРФЕЙСА ПРИЛОЖЕНИЯ НА ЯЗЫКЕ PYTHON

КУРСОВАЯ РАБОТА

студента 2 курса 211 группы
направления 02.03.02 — Фундаментальная информатика и информационные технологии
факультета КНиИТ
Слепова Ильи Алексеевича

Научный руководитель
доцент, к. ф.-м. н. _____ С. В. Миронов

Заведующий кафедрой
к. ф.-м. н., доцент _____ А. С. Иванов

Саратов 2020

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

В данной работе подробно описаны этапы создания графического интерфейса для приложения на языке Python 3.7 с использованием дополнительных графических библиотек: Tkinter и glfw, последняя из которых предоставляет возможность работы со спецификацией OpenGL. Приложение демонстрирует возможности, доступные разработчикам при программировании на Python.

Язык Python является интерпретируемым императивным языком, что делает его чрезвычайно гибким и пригодным для выполнения широкого спектра задач: решение математических задач, анализ данных, веб-разработка, а также создание графического интерфейса задач.

Графическая спецификация OpenGL, считается одной из наиболее популярных для создания компьютерных игр, а следовательно графических интерфейсов, двумерной и трехмерной графики.

Целью данной работы является создание приложений с графическим интерфейсом, используя графические библиотеки Tkinter и glfw, а также язык Python. Таким образом, для достижения поставленной цели необходимо:

- изучить средства языка Python;
- изучить средства реализации графического приложения на Python;
- изучить графическую библиотеку Tkinter и спецификацию OpenGL для работы на Python;
- продемонстрировать на примерах работу графического интерфейса, реализованного на языке Python.

1 Описание элементов приложения

Сложно представить современное приложение без графического интерфейса, который позволяет пользователю удобно взаимодействовать с приложением. Графические интерфейс или же GUI состоит из всего, что пользователь видит в приложении: окна, кнопки, поля ввода-вывода информации и так далее. При работе с этими элементами интерфейса пользователь может взаимодействовать с видимыми объектами с помощью мыши, клавиатуры, сенсорного экрана и других устройств ввода. Такое разнообразие способов ввода информации, несомненно, улучшает пользовательский опыт, из-за чего графический интерфейс стал неотъемлемой частью бытовых и профессиональных инструментов.

1.1 Tkinter

Текст данного раздела написан на основе источников [1–4].

Одной из самых популярных графических библиотек в языке Python является Tkinter. Она позволяет создать приложение с любыми элементами интерфейса: кнопками, картинками, полями ввода-вывода текста и так далее.

Для начала работы с данной библиотекой, ее нужно импортировать в проект с помощью команды `from tkinter import *`, а затем создать объект с помощью интерпретатора `tk`. Tkinter является событийно-ориентированной библиотекой, из-за чего в приложениях подобного типа должен быть главный цикл — `mainloop()`, который не позволит приложению закрыться без выполнения какого-то определенного действия. В случае с Tkinter таким действием является метод `quit()`, который используется для выхода из интерпретатора и завершения цикла обработки событий.

```
1 from tkinter import *
2 from math import *
3
4 #Создание окна
5 window = Tk()
6 window.title("2d picture") #Заголовок
7 window.geometry("1000x1000") #Размер
8 window.configure(background = "gray") #Цвет заднего фона
```

Данный код позволяет нам создать примитивное приложение, результат

которого изображен на Рис. 1.

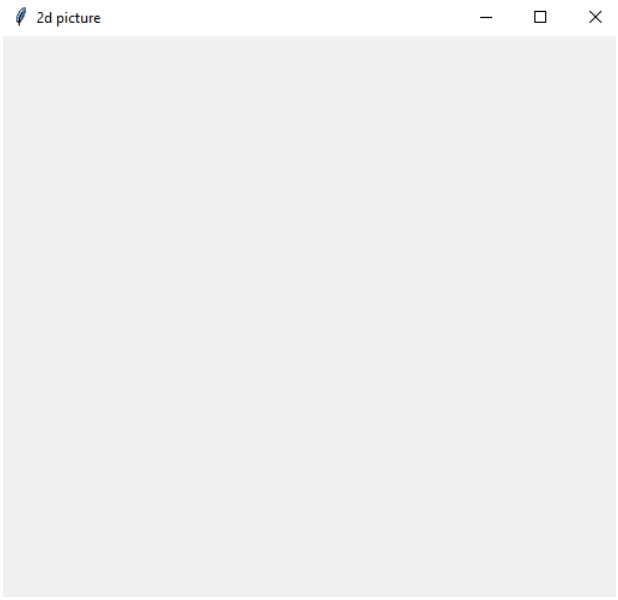

Рисунок 1 – Стандартное окно Tkinter

Так, `window = Tk()` — это объект, который отвечает за все действия с окном, `window.mainloop()` — метод, с помощью которого запускается цикл обработки событий, а методы `window.title()` и `window.geometry()` позволяют изменить заголовок и размер окна соответственно.

1.2 Label

Важной частью интерфейса являются текстовые метки, которые помогают пользователю ориентироваться в приложении. В Tkinter данный элемент называется `Label`, он позволяет вывести в окно статический текст, для этого достаточно лишь вызвать метод `Label()`:

```
18 label = Label(text="Text", justify=LEFT) #Конструктор
19 label.place(x = 50, y = 50) #Расположение
```

Как можно видеть, данный метод принимает несколько аргументов, которые позволяют выбрать окно для отображения, указать, какой именно текст будет отображаться, а также выбрать один из установленных шрифтов. Но данные аргументы — это не все доступные для текста настройки. У `Label` существует множество методов для более гибкой настройки отображения текста. Например, метод `anchor()` позволяет закрепить текст в определенном месте окна, `place()` позволяет установить положение текста в окне, `underline()`

подчеркивает один или несколько символов и так далее. Результат работы приложения изображен на Рис. 2.

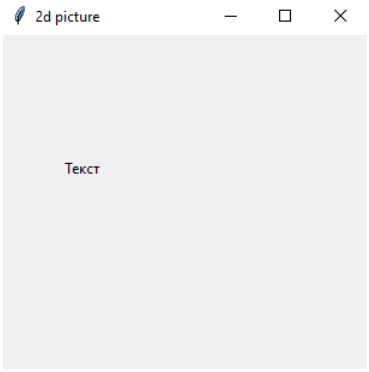

Рисунок 2 – Окно с текстом

1.3 Button

Кнопки одна из самых важных частей интерфейса, потому что именно через них происходит взаимодействие пользователя с приложением. Создать ее можно с помощью конструктора `Button()`, добавив в качестве аргумента окно, к которому будет привязана кнопка. Для добавления текста, нужно обратиться к элементу `text` объекта `button` так, будто это элемент массива, а прикрепить какое-либо событие на нажатие кнопки можно с помощью метода `bind()`.

```
22 Button = Button(window) #Конструктор
23 Button.bind("<Button>", print(2 + 2)) #Действие
24 Button["text"] = "Button" #Текст кнопки
25 Button.pack()
26 Button.place(x = 40, y = 130) #Расположение
27 window.mainloop()
```

Результат выполнения кода представлен на Рис. 3.

1.4 Entry

Как упоминалось ранее, важным элементом интерфейса являются поля ввода, с помощью которых можно передать в приложение любые данные. Для этого используется объект `Entry`, который представляет из себя простое поле для ввода текста. Конструктор данного объекта содержит два обязательных параметра:

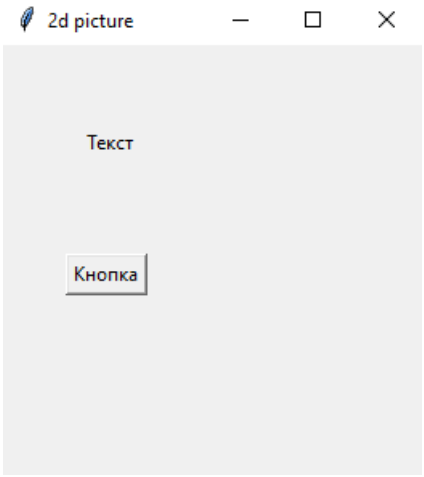

Рисунок 3 – Окно с кнопкой

```
13 entry = (Entry(window,width=20,bd=3)) #Конструктор
14 entry.pack()
15 entry.place(x = 10, y = 100) #Расположение
```

Где `master` — это окно, а `options` — список нескольких параметров: `bd` — толщина границы, `bg` — цвет фона, `font` — шрифт вводимого текста, `cursor` — изменение курсора мыши при наведении на данное поле, `width` — ширина элемента и `justify` — выравнивание текста, где значение `LEFT` выравнивает текст по левому краю, `RIGHT` — по правому, `CENTER` — по центру.

Полученное поле изображено на Рис. 4.

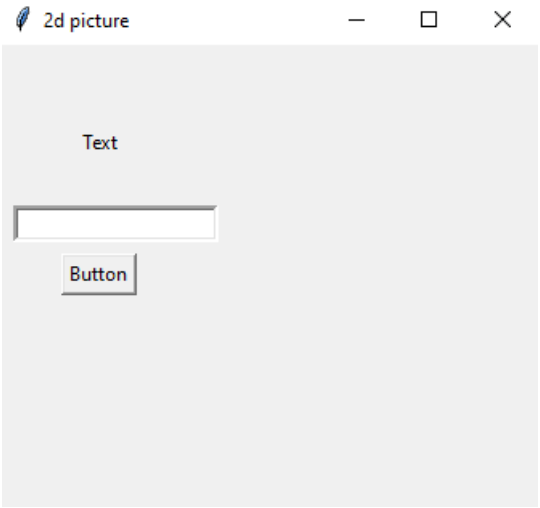

Рисунок 4 – Окно с полем ввода

1.5 Canvas

Основным для данного приложения объектом интерфейса является холст для рисования — `Canvas`, который используется для отображения различных фигур. Для создания пустого холста необходимо вызвать конструктор `Canvas()`, указав в аргументах окно, а также ширину и высоту холста.

```
9  windowCanvas = Canvas(window, width = 300, height = 300) #Размер холста
10 windowCanvas.pack()
```

Класс `Canvas` предоставляет множество методов для рисования фигур. Самым примитивным из них можно назвать `create_line()`, который проводит линию по указанным координатам и может принимать аргументы, изменяющие параметры линии: `width` определяет ширину, `fill` изменяет цвет, `arrowshape` позволяет добавить стрелку на конце линии и так далее. Результат работы приложения изображен на Рис. 5.

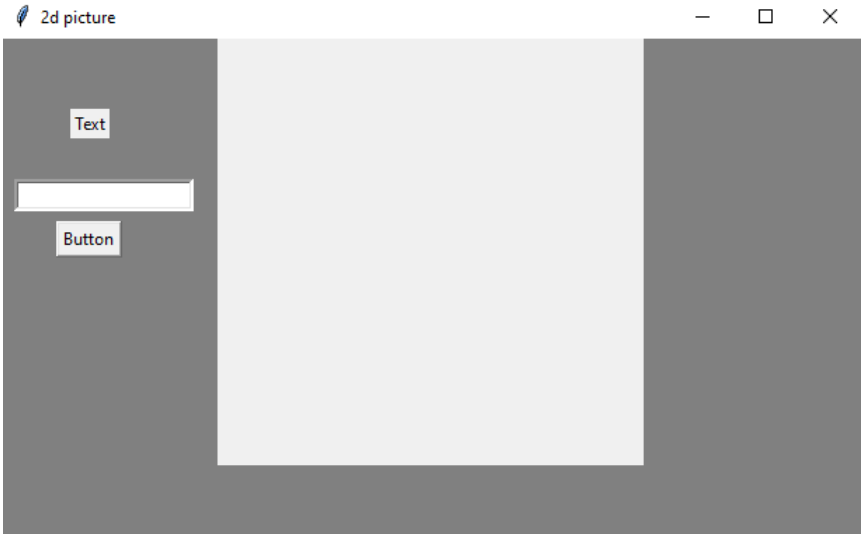

Рисунок 5 – Окно с холстом

Также разработчику доступны следующие методы: `create_rectangle()` для рисования прямоугольников, `create_oval()` для рисования окружностей, `create_polygon()` для рисования многоугольников, а также `create_arc()` для создания сегментов. Эти методы имеют одинаковые с `create_line()` параметры, но в `create_arc()` есть особые аргументы. Так, `start` изменяет градус начала сектора, `style` определяет, чем именно является фигура: сектором, сегментом или дугой.

1.6 Приложение с использованием Tkinter

В ходе изучения графической библиотеки Tkinter было реализовано приложение для отображения в окне изображения, а также функционирующими кнопками, которые позволяют изменить угол поворота изображения, масштабирование и смещение по осям Ox и Oy.

Сначала была произведена инициализация окна с холстом, а затем описан метод, заполняющий массив `figure` семью массивами `path`, которые содержат в себе координаты линий различных частей изображения:

```
from tkinter import *
from math import *

#Создание окна
window = Tk()
window.title("2d picture") #Заголовок
window.geometry("1000x1000") #Размер
window.configure(background = "gray") #Цвет заднего фона
windowCanvas = Canvas(window, width = 1000, height = 1000) #Размер холста
windowCanvas.pack()

#Загрузка координат
def loadCoord():
```

Далее описаны поля и соответствующие переменные для ввода параметров, которые отвечают за преобразования изображения:

```
#Поле ввода смещения по оси Oy
yTranslateInput = (Entry(window, width=20, bd=3)) #Конструктор
yTranslateInput.pack()
yTranslateInput.place(x=10, y = 10) #Расположение

#Заголовок
labelYTranslate = Label(text="Oy translate", justify=LEFT) #Конструктор
labelYTranslate.place(x = yTranslateInput.winfo_width() + 150, y = 10) #Расположение

#Поле ввода смещения по оси Ox
xTranslateInput = (Entry(window, width=20, bd=3)) #Конструктор
xTranslateInput.pack()
xTranslateInput.place(x=10, y = 40) #Расположение

#Заголовок
labelXTranslate = Label(text="Ox translate", justify=LEFT) #Конструктор
```

```
225 labelXTranslate.place(x = xTranslateInput.winfo_width() + 150, y = 40) #Расположение
226
227 #Поле ввода значения угла поворота
228 rotateAngleInput = (Entry(window, width=20, bd=3)) #Конструктор
229 rotateAngleInput.pack()
230 rotateAngleInput.place(x = 10, y = 70) #Расположение
231
232 #Заголовок
233 labelRotateAngle = Label(text="Rotate angle", justify=LEFT) #Конструктор
234 labelRotateAngle.place(x = yTranslateInput.winfo_width() + 150, y = 70) #Расположение
235
236 #Поле ввода значения масштаба
237 scaleValueInput = (Entry(window,width=20,bd=3)) #Конструктор
238 scaleValueInput.pack()
239 scaleValueInput.place(x = 10, y = 100) #Расположение
240
241 #Заголовок
242 labelScaleValue = Label(text="Scale", justify=LEFT) #Конструктор
243 labelScaleValue.place(x = yTranslateInput.winfo_width() + 150, y = 100) #Расположение
244
```

Наконец, описывается метод `printImage()`, который отвечает за отрисовку изображения, а также за переменные, которые отвечают за его преобразования. Сначала очищается холст, потом инициализируются переменные для хранения информации о введенных в поля значениях масштаба и его установка относительно размеров окна:

```
246 def printImage(event):
247
248     #Переменные для хранения размеров окна
249     windowWidth = windowCanvas.winfo_width()
250     windowHeight = windowCanvas.winfo_height()
251
252     #Очистка холста
253     windowCanvas.delete('all')
254
255     #Переменная для хранения количества итераций цикла
256     i = 0
257
258     #Установка масштаба относительно размеров окна
259     if (windowWidth/windowHeight >= 270/190):
260         scale = windowHeight/190
261     else:
262         scale = windowWidth/270
```

```
263
264     #Получение масштаба из поля ввода
265     if scaleValueInput.get():
266         scaleValue = float(scaleValueInput.get())
267     else:
268         scaleValue = 1
```

Затем инициализируется переменная для стандартных значений координат, а также ввод величины угла из поля ввода и вычисление косинуса и синуса:

```
270     #Стандартные значения координат x и y относительно размеров окна
271     initialXCoordinate = float((windowWidth - 190*scale*scaleValue)/2)
272     initialYCoordinate = float((windowHeight - 270*scale*scaleValue)/2)
273
274     #Получение угла поворота из поля ввода
275     if (rotateAngleInput.get()):
276         rotateAngleValue = float(rotateAngleInput.get())
277     else:
278         rotateAngleValue = 0
279
280     #Вычисление угла, косинуса и синуса
281     rotateAngleValue = rotateAngleValue * pi / 180
282     angleCos = (cos(rotateAngleValue))
283     angleSin = (sin(rotateAngleValue))
```

Далее, вычисляются координаты центра окна и применяются все преобразования к ранее инициализированным координатам:

```
285     #Изменение координат относительно всех преобразований
286     initialXCoordinate = initialXCoordinate * angleCos - initialYCoordinate * angleSin
287     initialYCoordinate = initialYCoordinate * angleCos + initialXCoordinate * angleSin
288
289     #Координаты цетра окна
290     windowCenterXCoordinate = windowWidth / 2
291     windowCenterYCoordinate = windowHeight / 2
292
293     #Получение смещения по оси Ox из поля ввода
294     if xTranslateInput.get():
295         initialXCoordinate += int(xTranslateInput.get())
296
297     # Получение смещения по оси Oy из поля ввода
```

```
298     if yTranslateInput.get():
299         initialYCoordinate -= int(yTranslateInput.get())
```

В конце метода описан цикл отрисовки изображения с учетом всех преобразований:

```
302     for j in range(len(figure)):
303         while i < len(figure[j]):
304             windowCanvas.create_line((figure[j][i] * scale * scaleValue -
                ↪  windowCenterXCoordinate) * angleCos - (
305                         figure[j][i + 1] * scale * scaleValue - windowCenterYCoordinate) *
                            ↪  angleSin + windowCenterXCoordinate + initialXCoordinate,
306                         (figure[j][i + 1] * scale * scaleValue - windowCenterYCoordinate) *
                            ↪  angleCos + (
307                         figure[j][i] * scale * scaleValue - windowCenterXCoordinate) *
                            ↪  angleSin + windowCenterYCoordinate + initialYCoordinate,
308                         (figure[j][i + 2] * scale * scaleValue - windowCenterXCoordinate) *
                            ↪  angleCos - (
309                         figure[j][i + 3] * scale * scaleValue - windowCenterYCoordinate) *
                            ↪  angleSin + windowCenterXCoordinate + initialXCoordinate,
310                         (figure[j][i + 3] * scale * scaleValue - windowCenterYCoordinate) *
                            ↪  angleCos + (
311                         figure[j][i + 2] * scale * scaleValue - windowCenterXCoordinate) *
                            ↪  angleSin + windowCenterYCoordinate + initialYCoordinate,
312                         fill="orange", width=3)
313             i = i + 4
314         i = 0
```

Данная функция вызывается при нажатии кнопки, описанной следующим кодом:

```
317 drawButton = Button(window) #Конструктор
318 drawButton["text"] = "Draw" #Надпись
319 drawButton.bind("<Button-1>", printImage) #Функция при нажатии
320 window.bind("drawButton", printImage) #Окно отображения кнопки
321 drawButton.pack()
322 drawButton.place(x = 40, y = 130) #Расположение
323 window.mainloop()
```

Результат работы приложения изображен на Рис. 6.

Полный код программы приведен в приложении А.

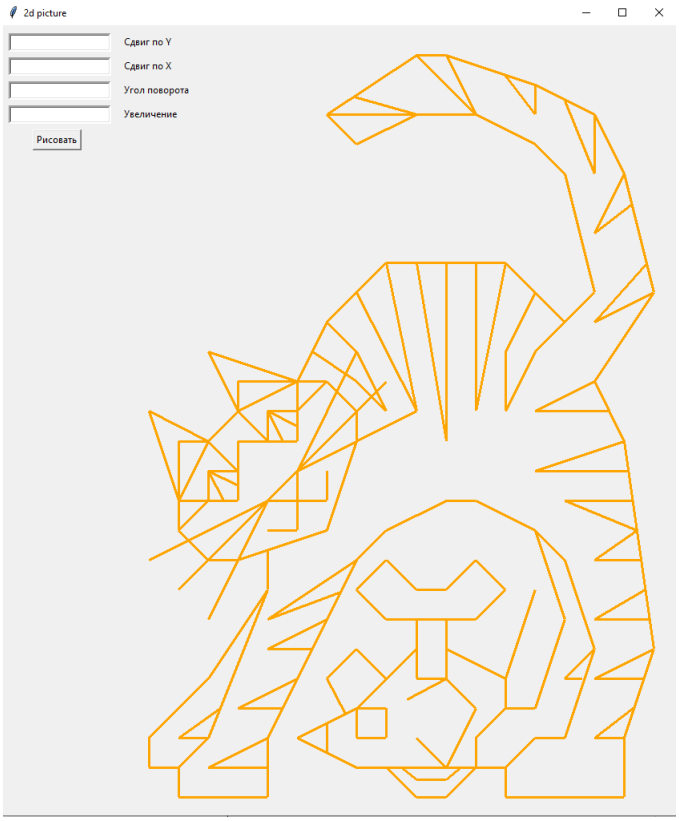

Рисунок 6 – Окно с выведенным изображением

2 OpenGL

Работа со спецификацией OpenGL описана на основе источников [3,5–8].

Как говорилось ранее, OpenGL является одной из самых популярных спецификаций программного интерфейса для получения интерактивных или программно-генерируемых изображений. Программы, написанные с использованием OpenGL, могут работать на любых платформах, демонстрируя одинаковый результат.

Чтобы работать со спецификацией OpenGL в Python, необходимо импортировать в приложение библиотеку glfw и пакеты *OpenGL.GL*. Также импортируются дополнительные библиотеки: numpy и pyrr, которые позволяют работать с матрицами и векторами:

```
1 import glfw
2 from OpenGL.GL import *
3 from OpenGL.GL.shaders import *
4 import numpy as np
5 import pyrr
```

Далее идет проверка, инициализирована ли библиотека glfw:

```
8 if not glfw.init():
9     raise Exception("glfw can not be initialized!")
```

После подключения библиотек происходит создание окна, с помощью метода `create_window()`, который получает в качестве аргументов размеры окна, его заголовок, режим работы монитора, библиотеки glfw, после этого указывается позиция окна на экране с помощью метода `set_window_pos`, затем происходит инициализация этого окна с помощью метода `make_content_current()`:

```
12 window = glfw.create_window(1280, 720, "My OpenGL window", None, None) #Конструктор
13 glfw.set_window_pos(window, 400, 200) #Расположение окна
14 glfw.make_context_current(window) #Выбор рабочегоокна
```

Затем идет проверка на корректность создания окна:

```
18 if not window:
19     glfw.terminate()
20     raise Exception("glfw window can not be created!")
```

Наконец, чтобы увидеть созданное окно, создается цикл, который работает, пока окно не закрыто, и содержит в себе методы `poll_events()` и `swap_buffers()`, организующие отображение окна.

Полученное окно представлено Рис. 7.

Рисунок 7 – Пустое окно

2.1 Шейдеры

Для упрощения работы с OpenGL используются шейдеры — программы, которые выполняются для каждого участка кода графическим процессором или же GPU, а написаны они на языке GLSL. Так, для работы с вершинами рисунка используется `Vertex Shader`:

```
50 vertex_src = """
51 # version 330
52 layout(location = 0) in vec3 aPosition;
53
54 uniform mat4 model;
55 uniform mat4 projection;
56
57 void main()
58 {
```

```
59      gl_Position = projection * model * vec4(aPosition, 1.0);
60  }
61  """
```

Данный шейдер отвечает за положение объекта в пространстве. Таким образом, переменная `aPosition` хранит в себе позицию объекта, `model` содержит в себе все преобразования объекта, а `projection` хранит данные о проекции.

Также существует шейдер, отвечающий за цвет объекта — `Fragment Shader`:

```
64  fragment_src = """
65  # version 330
66
67  out vec4 outColor;
68
69  void main()
70  {
71      outColor = vec4(0.0f,0.0f,0.0f,1.0f);
72  }
73  """
```

Он вычисляет цвет для каждого пикселя в формате RGBA с помощью вещественных чисел, из которых состоит четырехмерный вектор `outColor`.

2.2 Отрисовка объектов

Все сложные изображения состоят из простых фигур — линий, точек, многоугольников. К сожалению, в OpenGL не предусмотрено методов для рисования данных объектов, поэтому все эти функции описаны в утилите *glu32.dll*. Для создания примитивов предусмотрены параметры, используемые в методах `glDrawElement()`, `glDrawArrays()`, `glDrawBuffer()` и так далее. Самым простым параметром является `GL_POINTS`: с помощью него рисуются отдельные точки. Параметр `GL_LINES` используется для рисования линий. Он соединяет две указанные вершины, а также дает возможность указать параметры линии: толщину, цвет и сглаживание.

Далее рассмотрим параметры для отрисовки многоугольников. Так, для отрисовки треугольников используется `GL_TRIANGLES`, добавляющий те же

аргументы, что и `GL_LINES`, но его возможности можно расширить с помощью метода `glPolygonMode()`, который добавит два дополнительных параметра: `face`, указывающий на секторы, и `mode`, который отвечает за способ отрисовки треугольника — точками, линиями или область с заливкой. Для рисования фигур, содержащих более, чем 3 угла, используется параметр `GL_QUADS`, к которому применимо все, что описывалось для треугольника.

Результаты применения данных методов будут представлены в последующих разделах.

2.3 Приложение с использованием OpenGL

Для демонстрации ранее описанных возможностей библиотеки создано приложение, которое использует перспективную проекцию для наблюдения за вращающимся вокруг оси Oy объектом, который можно загрузить из файла.

Перейдем к созданию приложения. Сначала импортируются нужные библиотеки:

```
1 import glfw
2 from OpenGL.GL import *
3 from OpenGL.GL.shaders import *
4 import numpy as np
5 import pyrr
```

Затем происходит создание окна, а также инициализация библиотеки glfw и проверка корректности выполнения этих действий:

```
8  if not glfw.init():
9      raise Exception("glfw can not be initialized!")
10
11 #Создание окна
12 window = glfw.create_window(1280, 720, "My OpenGL window", None, None) #Конструктор
13 glfw.set_window_pos(window, 400, 200) #Расположение окна
14 glfw.make_context_current(window) #Выбор рабочегоокна
15 glClearColor(0.3, 0.45, 0.1, 1) #Цвет заднего фона
16
17 #Проверка корректности создания окна
18 if not window:
19     glfw.terminate()
20     raise Exception("glfw window can not be created!")
```

Далее описан метод `window_resize()`, которые принимает следующие аргументы: `window` — целевое окно, `width` — ширина окна и `height` — его высота. Данный метод позволяет изменять размеры окна, при этом объект всегда остается в середине:

```
23 def window_resize(window, width, height):
24     glViewport(0, 0, width, height)
25     projection = pyrr.matrix44.create_perspective_projection_matrix(45, width / height,
       ↪  0.1, 100)
26     glUniformMatrix4fv(proj_loc, 1, GL_FALSE, projection)
```

После чего вызывается метод `set_window_size_callback()`, который инициализирует ранее описанную функцию для OpenGL:

```
28 glfw.set_window_size_callback(window, window_resize)
```

После описаны переменные, а также цикл для считывания вершин изображения из файла:

```
31 file_name = "car.txt"
32 file = open(file_name, 'r')
33 figure = []
34 flag = True
35 while(flag):
36     cmd = file.readline().split()
37     if(cmd[0] == 'figure'):
38         flag = False
39     elif(cmd[0] == 'path'):
40         i = 0
41         while(i < int(cmd[1])):
42             cmd1 = file.readline().split()
43             if (cmd1[0].isalpha() == False):
44                 for j in range(3):
45                     figure.append(float(cmd1[j])*0.2)
46             i += 1
47 vertices = np.array(figure, dtype=np.float32)
```

Затем инициализируются фрагментный и вершинный шейдеры, и на их основе создается шейдерная программа:

```
#Вершинный шейдер
vertex_src = """
# version 330
layout(location = 0) in vec3 aPosition;

uniform mat4 model;
uniform mat4 projection;

void main()
{
    gl_Position = projection * model * vec4(aPosition, 1.0);
}
"""

#Фрагментный шейдер
fragment_src = """
# version 330

out vec4 outColor;

void main()
{
    outColor = vec4(0.0f,0.0f,0.0f,1.0f);
}
"""

#Компиляция шейдеров и шейдерной программы
vertexShader = compileShader(vertex_src, GL_VERTEX_SHADER)
fragmentShader = compileShader(fragment_src, GL_FRAGMENT_SHADER)
shader = compileProgram(vertexShader, fragmentShader)
```

С помощью метода `compileShader()` происходит компиляция шейдера, а с помощью `compileProgram()` компилируется шейдерная программа.

Далее создается объект *Vertex Buffer Object* или же *VBO*, с помощью которого данные передаются в GPU, эти функции выполняют метод `glGenBuffer()`, который принимает в качестве аргумента количество генерируемых буферов и, соответственно, создает буфер, а затем метод `glBindBuffer()`, который выбирает тип буфера и объект, которые передаются в первом и втором аргументах соответственно. После его объявления в буффер загружается массив с вершинами с помощью метода `glBufferData()`, который принимает 4 аргумента: тип, размер данных, сами данные, и режим использования. Обязательно описывается параметр `GL_STATIC_DRAW`, который делает данные

иммутабельными, показывая, что они используются для рисования. После загрузки данных в буфер, вызываются методы `glEnableVertexAttribArray()` и `glVertexAttribPointer()`, которые отвечают за описание расположения параметра вершинного шейдера в буфере. Так, первый метод активирует параметр, а второй описывает расположение с помощью 6 аргументов — первый соответствует значению `location` вершинного шейдера, второй и третий аргументы описывают количество и тип элементов буфера, пятый аргумент задает шаг в байтах для перехода к этому же атрибуту следующей вершины, а последний задает первоначальное смещение для чтения данных из буфера:

```
80 #Инициализация и заполнение буфера для передачи вершин изображения
81 objectVBO = glGenBuffers(1) #Конструктор
82 glBindBuffer(GL_ARRAY_BUFFER, objectVBO) #Выбор буфера
83 glBufferData(GL_ARRAY_BUFFER, vertices.nbytes, vertices, GL_STATIC_DRAW) #Заполнение
   ↪ буфера
84 glBufferData()
85
86 #Инициализация и заполнения массива вершин
87 glEnableVertexAttribArray(0)
88 glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, vertices.itemsize * 3, ctypes.c_void_p(0))
```

После этого описываются переменные для хранения параметров проекции и преобразований. Создание матриц происходит с помощью библиотеки *pyrr*, а также ее методов `matrix44.create_perspective_projection_matrix()` для создания проекции и `matrix44.create_from_translation()` для создания матрицы преобразований:

```
91 projection = pyrr.matrix44.create_perspective_projection_matrix(45, 1280/720, 0.1, 100)
92 translation = pyrr.matrix44.create_from_translation(pyrr.Vector3([0, 0, -3]))
```

Затем начинаются взаимодействия с шейдерной программой. Сначала мы выбираем программу с помощью метода `glUseProgram()`, принимающего в качестве аргумента шейдерную программу, а затем, с помощью метода `glGetUniformLocation()`, создаются 2 переменные, которые впоследствии будут использоваться для передачи данных в шейдерную программу с помощью методов `glUniformMatrix4fv()`, которая принимает 4 аргумента: переменную, в которую происходит передача данных, количество экземпляров,

`boolean` переменную, которая показывает, нужно ли транспонировать полученные матрицы, а также саму матрицу:

```
#Выбор используемой шейдерной программы
glUseProgram(shader)

#Создание перменных для передачи значений в шейдерную программу
model_loc = glGetUniformLocation(shader, "model")
proj_loc = glGetUniformLocation(shader, "projection")

#Передача значения проекции в шейдерную программу
glUniformMatrix4fv(proj_loc, 1, GL_FALSE, projection)
```

Далее следует цикл `while not glfw.window_should_close(window)`, который отвечает за отображения всего, чтобы описано ранее. Он работает до тех пор, пока окно не закрыто. Внутри используются функции, описанные ранее, поэтому не стоит заострять на них особого внимания и перейдем к общему описанию цикла. Сначала создается матрица `model`, которая содержит все преобразования объекта, стоит отметить первый аргумент конструктора, который отвечает за вращение объекта по оси Oy, после этого матрица передается в шейдерную программу и отрисовывается линиями с помощью метода `glDrawArrays()`. В конце вызывается метод `terminatte()`, обозначающий конец программы:

```
while not glfw.window_should_close(window):
    glfw.poll_events()

    #Очистка холста
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)

    #Матрица модели с вращением
    model = pyrr.matrix44.multiply(pyrr.Matrix44.from_y_rotation(0.8*glfw.get_time()),
    ↪  translation)

    #Передача матрицы модели в шейдерную программу
    glUniformMatrix4fv(model_loc, 1, GL_FALSE, model)

    #Отрисовка линий изображения
    glDrawArrays(GL_LINE_STRIP, 0, len(vertices))

    glfw.swap_buffers(window)
glfw.terminate()
```

Полученное изображение представлено Рис. 8 и 9.

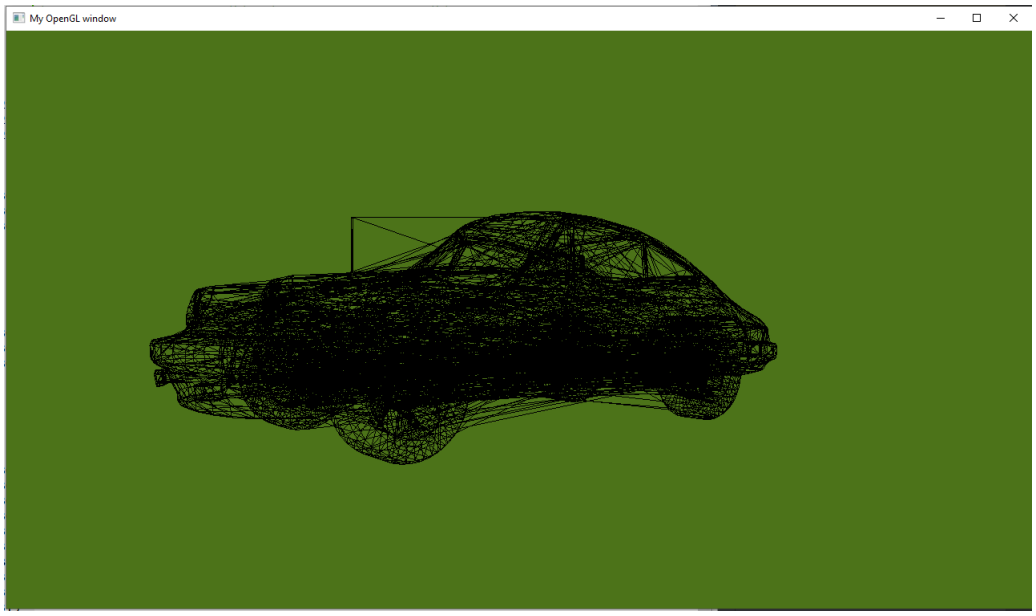

Рисунок 8 – Окно с изображением автомобиля

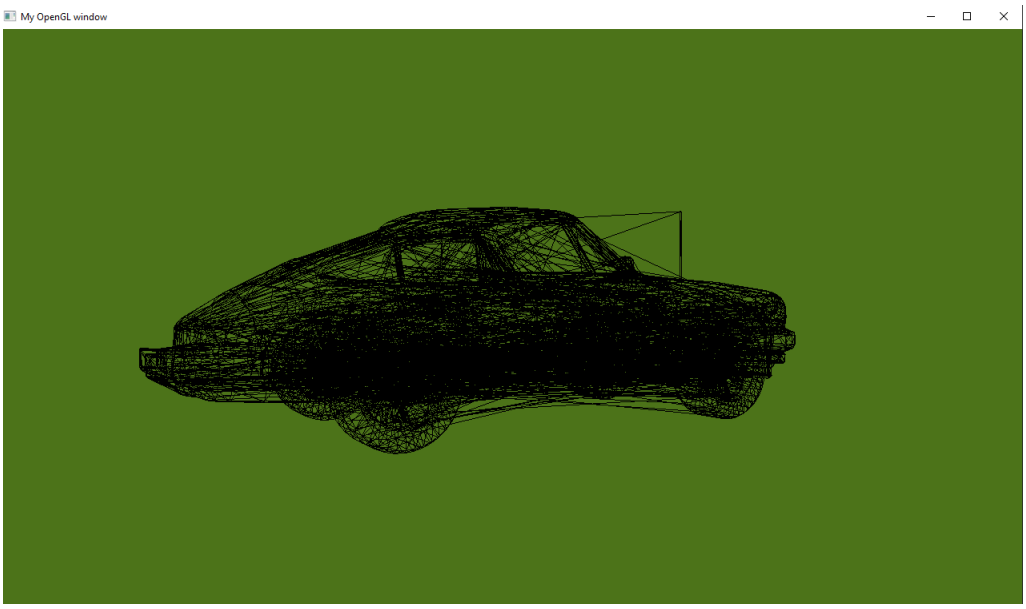

Рисунок 9 – Окно с повернутым по оси Oy изображением автомобиля

Полный код программы приведен в приложении Б.

2.4 Освещение

Информация о реализации системы освещения в OpenGL написана на основе источников [9–13].

Освещение в OpenGL основано на использовании приближенных к реальности упрощенных математических моделей. Одна из них называется моделью освещения Фонга и состоит из трех главных компонентов: *ambient* — фонового, который имитирует окружающий свет, чтобы объект не был абсолютно черным, *diffuse* — рассеянного, который имитирует направленный на объект источник света, а также *specular* — бликового, который имитирует блик, который появляется на блестящих объектах.

Чтобы такое освещение выглядело более реалистично, объекты должны по-разному реагировать на свет. Для этого используются цвета материалов для всех трех компонентов освещения, с добавлением еще одного — силы блеска или же *shininess*, который влияет на радиус бликов.

Очевидно, что в природе свет не может появиться без источника, поэтому в OpenGL также возможно реализовать несколько видов источников света с помощью шейдеров: направленный — моделируемый бесконечно удаленный источник света с лучами, идущими в одном направлении и независящими от расположения самого источника, точечный — источник, имеющий заданное положение в пространстве и распространяющий лучи равномерно во всех направлениях, а интенсивность падает с расстоянием, прожектор — источник, имеющий заданное положение, но его лучи распространяются только в одном направлении.

2.5 Приложение с освещением с использованием OpenGL

Для демонстрации ранее описанных возможностей освещения создано приложение, использующее модель освещения Фонга с двумя точечными источниками света: статичным и динамичным, которые освещают модель, состоящую из материалов.

Перейдем к созданию приложения. Сначала импортируются нужные пакеты и библиотеки. Стоит отметить, что добавилась еще одна библиотека — glm, которая во многом схожа с pyrr, но имеет некоторые отличные методы и конструкторы для векторов и матриц:

```
import glfw
from OpenGL.GL import *
from OpenGL.GL.shaders import compileProgram, compileShader
import pyrr
import numpy as np
import glm
```

Далее описывается массив вершин и нормалей источника света. В нашем случае куб:

```
#Координаты вершин источника света
lightVertices = [
        -0.5, -0.5, -0.5,  0.0,  0.0, -1.0,
         0.5, -0.5, -0.5,  0.0,  0.0, -1.0,
         0.5,  0.5, -0.5,  0.0,  0.0, -1.0,
         0.5,  0.5, -0.5,  0.0,  0.0, -1.0,
        -0.5,  0.5, -0.5,  0.0,  0.0, -1.0,
        -0.5, -0.5, -0.5,  0.0,  0.0, -1.0,

        -0.5, -0.5,  0.5,  0.0,  0.0,  1.0,
         0.5, -0.5,  0.5,  0.0,  0.0,  1.0,
         0.5,  0.5,  0.5,  0.0,  0.0,  1.0,
         0.5,  0.5,  0.5,  0.0,  0.0,  1.0,
        -0.5,  0.5,  0.5,  0.0,  0.0,  1.0,
        -0.5, -0.5,  0.5,  0.0,  0.0,  1.0,

        -0.5,  0.5,  0.5, -1.0,  0.0,  0.0,
        -0.5,  0.5, -0.5, -1.0,  0.0,  0.0,
        -0.5, -0.5, -0.5, -1.0,  0.0,  0.0,
        -0.5, -0.5, -0.5, -1.0,  0.0,  0.0,
        -0.5, -0.5,  0.5, -1.0,  0.0,  0.0,
        -0.5,  0.5,  0.5, -1.0,  0.0,  0.0,

         0.5,  0.5,  0.5,  1.0,  0.0,  0.0,
         0.5,  0.5, -0.5,  1.0,  0.0,  0.0,
         0.5, -0.5, -0.5,  1.0,  0.0,  0.0,
         0.5, -0.5, -0.5,  1.0,  0.0,  0.0,
         0.5, -0.5,  0.5,  1.0,  0.0,  0.0,
         0.5,  0.5,  0.5,  1.0,  0.0,  0.0,

        -0.5, -0.5, -0.5,  0.0, -1.0,  0.0,
         0.5, -0.5, -0.5,  0.0, -1.0,  0.0,
         0.5, -0.5,  0.5,  0.0, -1.0,  0.0,
         0.5, -0.5,  0.5,  0.0, -1.0,  0.0,
        -0.5, -0.5,  0.5,  0.0, -1.0,  0.0,
```

```
        -0.5, -0.5, -0.5,  0.0, -1.0,  0.0,

        -0.5,  0.5, -0.5,  0.0,  1.0,  0.0,
         0.5,  0.5, -0.5,  0.0,  1.0,  0.0,
         0.5,  0.5,  0.5,  0.0,  1.0,  0.0,
         0.5,  0.5,  0.5,  0.0,  1.0,  0.0,
        -0.5,  0.5,  0.5,  0.0,  1.0,  0.0,
        -0.5,  0.5, -0.5,  0.0,  1.0,  0.0
  ]
lightCube = np.array(lightVertices, dtype= 'float32')
```

После чего описываются шейдеры. Первым стал вершинный шейдер:

```
#Вершинный шейдер объекта
vertex_src = """
# version 330
layout(location = 0) in vec3 a_position;
layout(location = 2) in vec3 a_normal;
uniform mat4 model;
uniform mat4 projection;
uniform mat4 view;
uniform mat4 modelInv;
uniform mat4 modelView;

out vec3 Normal;
out vec3 fragPos;

void main()
{
    gl_Position = projection * view * model * vec4(a_position, 1.0);
    fragPos = vec3(modelView * vec4(a_position, 1.0f));
    Normal = mat3(modelInv) * a_normal;
}
"""
```

Так, переменные `a_position` и `a_normal` содержат в себе позицию и нормали объекта, `model` и `modelView` хранят данные о всех преобразованиях объекта и сцены, `projection` данные о проекции, `view` информацию о наблюдаемой точке, `modelInv` обратную транспонированную матрицу `model`. В методе `main()` происходит вычисление переменной `gl_Position`, хранящей позиции объекта относительно преобразований, проекции и точки наблюдения. Также вычисляется переменная `vfragPos`, которая содержит позицию

отдельного фрагмента, и последней переменной стала `Normal`, которая хранит измененный относительно матрицы `modelInv` нормализующий вектор.

Затем описывается фрагментный шейдер:

```
#Фрагментный шейдер объекта
fragment_src = """
#version 330 core

struct Light {
    vec3 position;
    vec3 ambient;
    vec3 diffuse;
    vec3 specular;
};

struct Material {
    vec3 ambient;
    vec3 diffuse;
    vec3 specular;
    float shininess;
};

in vec3 fragPos;
in vec3 Normal;

out vec4 color;

uniform Material material;
uniform vec3 viewPos;
uniform Light light;

void main() {

    vec3 norm = normalize(Normal);
    vec3 lightDir = normalize(light.position - fragPos);
    float diff = max(dot(norm, lightDir), 0.0f);
    float spec;
    if (diff > 0.0){
        vec3 viewDir = normalize(viewPos - fragPos);
        vec3 reflectDir = reflect(-lightDir, norm);
        spec = pow(max(dot(viewDir, reflectDir), 0.0f), material.shininess);
    }
    else
        spec = 0.0;

    vec3 specular = light.specular * (spec * material.specular);
    vec3 diffuse = light.diffuse * (diff * material.diffuse);
```

```
119     vec3 ambient = light.ambient * material.ambient;
120 
121     vec3 result = ambient + diffuse + specular;
122 
123     color = vec4(result, 1.0f);
124 }
125 """
```

В нем описываются две структуры `Light` и `Material`, которые содержат данные о параметрах света и цвете материала соответственно, после фрагментный шейдер получает ранее описаные переменные `fragPos` и `Normal` из вершинного шейдера, далее описаны переменные структур и `viewPos`, хранящая обратный вектор произведения матрицы наблюдаемой точки и единичного вектора. В методе `main()` вычисляется цвет фрагмента, на который в данный момент направлены лучи освещения. Сначала проходит нормализация нормализующего вектора, после чего вычисляется направление света `lightDir`, затем происходит вычисление коэффициента рассеивания, если данная переменная отлична от нуля, происходит вычисление бликового коэффициента `spec`, иначе он равен нулю. После происходит вычисление всех параметров света относительно ранее вычисленных коэффициентов и материалов объекта, после чего все они складываются и вычисляется цвет в формате RGBA.

Перейдем к шейдерам, описывающим источник счета. Вершинный шейдер:

```
127 #Вершинный шейдер источника света
128 vertex_light = """
129 # version 330
130 layout (location = 0) in vec3 a_position;
131 
132 uniform mat4 model;
133 uniform mat4 view;
134 uniform mat4 projection;
135 
136 void main()
137 {
138   gl_Position = projection * view * model * vec4(a_position, 1.0);
139 }
140 """
```

Данный шейдер отвечает за положение объекта в пространстве. Таким образом, переменная `aPosition` хранит в себе позицию объекта, `model` содержит в себе все преобразования объекта, `view` информацию о наблюдаемой точке, а `projection` хранит данные о проекции. В методе `main()` вычисляется позиция объекта относительно всех этих данных.

После описывается фрагментный шейдер:

```
142 #Фрагментный шейдер источника света
143 fragment_light = """
144 #version 330 core
145 out vec4 color;
146 void main() {
147   color = vec4(1.0f);
148 }
149 """
```

В данном шейдере происходит установка цвета объекта, в нашем случае белого.

Далее описываются классы материалов и объекта. Стоит отметить, что в строках 170-221 также происходит загрузка модели из файла:

```
151 #Класс для хранения значений параметров материала
152 class Material:
153     def __init__(self):
154         self.ambient = []
155         self.diffuse = []
156         self.specular = []
157         self.shininess = []
158 
159 #Класс объекта
160 class ObjLoader:
161     def __init__(self):
162         self.vert_coords = []
163         self.text_coords = []
164         self.norm_coords = []
165         self.material = Material()
166 
167         self.model = []
168 
169     #Загрузка информации об объекте из файла
170     def load_model(self):
171         file_name = "car_triangles.txt"
172         file = open(file_name, 'r')
```

```
        ambient = []
        diffuse = []
        text_coords = []
        specular = []
        figure = []
        shininess = 1.0
        flag = True
        while (flag):
            cmd = file.readline().split()
            if (cmd[0] == 'figure'):
                flag = False
            elif (cmd[0] == 'color'):
                ambient = [float(cmd[1]) / 255, float(cmd[2]) / 255, float(cmd[3]) / 255]
                diffuse = ambient
                specular = ambient
            elif (cmd[0] == 'ambient'):
                ambient = [float(cmd[1]), float(cmd[2]), float(cmd[3])]
            elif (cmd[0] == 'diffuse'):
                diffuse = [float(cmd[1]), float(cmd[2]), float(cmd[3])]
            elif (cmd[0] == 'specular'):
                specular = [float(cmd[1]), float(cmd[2]), float(cmd[3])]
            elif (cmd[0] == 'shininess'):
                shininess = float(cmd[1])
            elif (cmd[0] == 'mesh'):
                N = int(cmd[1])
                K = int(cmd[2])
                while (N > 0):
                    cmd1 = file.readline().split()
                    for i in range(6):
                        if i < 3:
                            self.vert_coords.append(float(cmd1[i]))
                        else:
                            self.norm_coords.append(float(cmd1[i]))
                    N -= 1
                while (K > 0):
                    cmd2 = file.readline().split()
                    for i in range(3):
                        text_coords.append(float(cmd2[i]))
                    K -= 1
                self.text_coords = np.array(text_coords, dtype = 'uint32')
                self.vert_coords = np.array(self.vert_coords, dtype = 'float32')
                self.norm_coords = np.array(self.norm_coords, dtype = 'float32')
                self.material.ambient = ambient
                self.material.diffuse = diffuse
                self.material.specular = specular
                self.material.shininess = shininess
                figure.extend(self.vert_coords)
                figure.extend(self.norm_coords)
```

```
221                 self.model = np.array(figure, dtype = 'float32')
```

Далее описываются координаты положения камеры, а также функция, которая позволяет перемещать камеру с помощью клавиатуры:

```
239 #Координаты камеры
240 global cameraXCoordinate
241 global cameraZCoordinate
242
243 cameraXCoordinate = 0
244 cameraZCoordinate = 25
245
246 #Управление камерой с помощью клавиатуры
247 def key_input(window, key, scancode, action, mode):
248     global cameraXCoordinate, cameraZCoordinate
249     if key == glfw.KEY_W and action == glfw.REPEAT:
250         cameraZCoordinate -= 0.5
251     if key == glfw.KEY_S and action == glfw.REPEAT:
252         cameraZCoordinate += 0.5
253     if key == glfw.KEY_A and action == glfw.REPEAT:
254         cameraXCoordinate -= 0.5
255     if key == glfw.KEY_D and action == glfw.REPEAT:
256         cameraXCoordinate += 0.5
```

После идет создание ранее описанного метода `window_resize()` , а также инициализация библиотеки glfw и создание окна:

```
258 glfw.set_window_pos(window, 400, 200)
259 glfw.set_window_size_callback(window, window_resize)
260 glfw.set_key_callback(window, key_input)
261 glfw.make_context_current(window)
262 glEnable(GL_DEPTH_TEST)
263 glClearColor(0.3, 0.3, 0.3, 1.0)
```

Затем происходит компиляция шейдерных программ для объекта, освещения и источника света:

```
267 shader = compileProgram(compileShader(vertex_src, GL_VERTEX_SHADER),
    ↪ compileShader(fragment_src, GL_FRAGMENT_SHADER))
268 lightShader = compileProgram(compileShader(vertex_light, GL_VERTEX_SHADER),
    ↪ compileShader(fragment_light, GL_FRAGMENT_SHADER))
```

Далее создаются и заполняются буферы и массив вершин для объекта с помощью ранее описанных методов. Стоит отметить, что для хранения координат объекта будет использоваться *Element Buffer Object* или же *EBO*:

```
#Создание буферов и массива для объекта
objectVAO = glGenVertexArrays(1)
objectVBO = glGenBuffers(1)
objectEBO = glGenBuffers(1)

#Заполнение вершинного буфера
glBindBuffer(GL_ARRAY_BUFFER, objectVBO)
glBufferData(GL_ARRAY_BUFFER, obj.model.nbytes, obj.model, GL_STATIC_DRAW)

#Заполнение массива вершин
glBindVertexArray(objectVAO)
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, obj.model.itemsize * 3, ctypes.c_void_p(0))
glEnableVertexAttribArray(0)

#Заполнение массива индексов
glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, obj.model.itemsize * 3,
↪  ctypes.c_void_p(20))
glEnableVertexAttribArray(1)

#Заполнение элементного буфера
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, objectEBO)
glBufferData(GL_ELEMENT_ARRAY_BUFFER, obj.text_coords.nbytes, obj.text_coords,
↪  GL_STATIC_DRAW)
glBindBuffer(GL_ARRAY_BUFFER, 0)
glBindVertexArray(0)
```

После создаются массив и буфер для источника света:

```
#Создание буфера и массива для источников света
lightCubeVAO = glGenVertexArrays(1)
lightCubeVBO = glGenBuffers(1)

#Заполнение вершинного буфера
glBindBuffer(GL_ARRAY_BUFFER, lightCubeVBO)
glBufferData(GL_ARRAY_BUFFER, lightCube.itemsize * 4, lightCube, GL_STATIC_DRAW)

#Заполнение массива вершин
glBindVertexArray(lightCubeVAO)
glVertexAttribPointer(0,3,GL_FLOAT,GL_FALSE, lightCube.itemsize * 3, ctypes.c_void_p(0))
glEnableVertexAttribArray(0)
```

```
310
311 #Заполнение массива индексов
312 glVertexAttribPointer(1,3,GL_FLOAT,GL_FALSE, lightCube.itemsize * 3, ctypes.c_void_p(12))
313 glEnableVertexAttribArray(1)
```

Наконец, создаются буфер и массив для освещения:

```
315 #Создание массива для отображения блика света
316 lightVAO = glGenVertexArrays(1)
317
318 #Заполнение вершинного буфера
319 glBindBuffer(GL_ARRAY_BUFFER, lightCubeVBO)
320 glBufferData(GL_ARRAY_BUFFER, lightCube.nbytes, lightCube, GL_STATIC_DRAW)
321
322 #Заполнение массива вершин
323 glBindVertexArray(lightVAO)
324 glVertexAttribPointer(0,3,GL_FLOAT,GL_FALSE, lightCube.itemsize * 6, ctypes.c_void_p(0))
325 glEnableVertexAttribArray(0)
```

Теперь создаются матрицы проекции, преобразований, а также положения статичного источника света:

```
327 projection = pyrr.matrix44.create_perspective_projection_matrix(90, 1280/720, 1, 2000)
    ↪ #Матрица проекции
328 translation = pyrr.matrix44.create_from_translation(pyrr.Vector3([0, 0, 0])) #Матрица
    ↪ преобразований
329 staticLightModel =  pyrr.matrix44.create_from_translation(pyrr.Vector3([7, 5, -4]))
    ↪ #Матрица преобразований статичного источника света
```

Данные матрицы создаются по аналогии с предыдущим приложением, стоит отметить только матрицу `staticLightModel`, которая содержит в себе позицию статичного источника света, координаты которого были выбраны для лучшей демонстрации результата.

Перейдем к выбору шейдерной программы, а также инициализации всех необходимых переменных:

```
331 #Выбор используемого шейдера
332 glUseProgram(shader)
333
```

```
334 #Инициализация переменных для передачи данных в шейдерную программу
335 #####################################################################
336 lightPos = glGetUniformLocation(shader, "light.position")
337 lightAmbient = glGetUniformLocation(shader, "light.ambient")
338 lightDiffuse = glGetUniformLocation(shader, "light.diffuse")
339 lightSpecular = glGetUniformLocation(shader, "light.specular")
340 lightView = glGetUniformLocation(lightShader, "view")
341 lightProjection = glGetUniformLocation(lightShader, "projection")
342 lightModel = glGetUniformLocation(lightShader, "model")
343
344 materialAmbient = glGetUniformLocation(shader, "material.ambient")
345 materialDiffuse = glGetUniformLocation(shader, "material.diffuse")
346 materialSpecular = glGetUniformLocation(shader, "material.specular")
347 materialShinnes = glGetUniformLocation(shader, "material.shininess")
348
349 sceneModel = glGetUniformLocation(shader, "model")
350 sceneProjection = glGetUniformLocation(shader, "projection")
351 sceneView = glGetUniformLocation(shader, "view")
352 modelView = glGetUniformLocation(shader, "modelView")
353 modelInv = glGetUniformLocation(shader, "modelInv")
354 viewPos = glGetUniformLocation(shader, "viewPos")
355 #####################################################################
```

Так, в первом блоке инициализируются все переменные, относящиеся к освещению и источникам света, затем все, что относятся к материалам, а в последнем блоке все, что относится к объекту и сцене.

Далее в шейдерную программу передаются переменные, которые не будут изменяться во время выполнения программы и при этом не относятся к источникам света:

```
357 #Передача неизменяемых данных в шейдерную программу
358 #####################################################################
359 glUniformMatrix4fv(sceneProjection, 1, GL_FALSE, projection)
360 glUniform3fv(lightAmbient, 1, obj.material.ambient)
361 glUniform3fv(lightDiffuse, 1, obj.material.diffuse)
362 glUniform3fv(lightSpecular, 1, obj.material.specular)
363 glUniform3fv(materialAmbient, 1, obj.material.ambient)
364 glUniform3fv(materialDiffuse, 1, obj.material.diffuse)
365 glUniform3fv(materialSpecular, 1, obj.material.specular)
366 glUniform1f(materialShinnes, obj.material.shininess)
367 #####################################################################
```

Стоит отметить, что метод `glUniform3fv()` используется для передачи вектора из трех элементов и принимает три аргумента: переменная, в которую происходит передача данных, количество векторов, а также сам вектор. Также стоит обратить внимание на метод `glUniform1f()`, который используется для передачи одного числа и принимает два аргумента: переменная, в которую происходит передача данных, и число.

Наконец, перейдем к описанию цикла вывода изображения на экран. Сначала вычисляются положение камеры, а также скорость движения динамичного источника света и радиус, на котором этот источник движется относительно объекта, после чего идет вычисление координат направленного света в переменной `inverseView`, и, относительно обоих источников света, вычисляется его положение на объекте и заносится в переменную `lightPosition`, после чего эти переменные передаются в шейдерную программу:

```
375     #Матрица с положением камеры
376     view = pyrr.matrix44.create_look_at(pyrr.Vector3([cameraXCoordinate, 0,
        ↪  cameraZCoordinate]), pyrr.Vector3([cameraXCoordinate, 0, 0]), pyrr.Vector3([0, 1,
        ↪  0]))
377
378     #Матрицы для движения источника света вокруг заданной точки
379     move = pyrr.Matrix44.from_y_rotation(glfw.get_time() * 1)
380     lightM = pyrr.matrix44.create_from_translation(pyrr.Vector3([15,0,5]))
381     dynamicLightModel = move * lightM
382
383     #Матрица отображения света
384     inverseView = np.linalg.inv(view) * np.array(glm.vec4(0,0,0,1))
385
386     #Матрица позиции света на объекте
387     lightPosition = np.array(dynamicLightModel * staticLightModel * pyrr.Vector4([0, 0, 0,
        ↪  1]))
388     lightPosition = np.array([lightPosition[0],lightPosition[1],-lightPosition[2]])
389
390     #Передача в шейдерную программу данных об положении камеры и света
391     glUseProgram(shader)
392     glUniform3fv(lightPos, 1, lightPosition)
393     glUniform3fv(viewPos, 1, inverseView)
394     glUniformMatrix4fv(sceneView, 1, GL_FALSE, view)
```

Затем создается `model` — матрица преобразований на основе преобразований объекта, а на ее основе создается обратная транспонированная матрица `modeli`, после чего эти данные передаются в шейдерную программу и про-

исходит отрисовка объекта с помощью метода `glDrawElements()`, который получает 4 аргумента: режим, размер массива, тип данных и индексы:

```
396     #Матрица модели
397     model = pyrr.matrix44.multiply(pyrr.Matrix44.from_y_rotation(3), translation)
398     modeli = np.linalg.inv(model).transpose()
399
400     #Передача в шейдерную программу данных об объекте
401     glBindVertexArray(objectVAO)
402     glUniformMatrix4fv(sceneModel, 1, GL_FALSE, model)
403     glUniformMatrix4fv(modelView, 1, GL_FALSE, model)
404     glUniformMatrix4fv(modelInv, 1, GL_FALSE, modeli)
405
406     #Отрисовка объекта
407     glDrawElements(GL_TRIANGLES, len(obj.text_coords), GL_UNSIGNED_INT, None)
408     glBindVertexArray(0)
```

После чего все данные передаются в шейдерную программу объектов света, создавая при этом 2 объекта: статичный и динамичный, который движется вокруг автомобиля:

```
410     #Отрисовка динамичного источника света
411     glUseProgram(lightShader)
412     glUniformMatrix4fv(lightView, 1, GL_FALSE, view)
413     glUniformMatrix4fv(lightProjection, 1, GL_FALSE, projection)
414     glUniformMatrix4fv(lightModel, 1, GL_FALSE, staticLightModel)
415     glBindVertexArray(lightVAO)
416     glDrawArrays(GL_TRIANGLES, 0, 36)
417     glBindVertexArray(0)
418
419     #Отрисовка динамичного источника света
420     glUseProgram(lightShader)
421     glUniformMatrix4fv(lightView, 1, GL_FALSE, view)
422     glUniformMatrix4fv(lightProjection, 1, GL_FALSE, projection)
423     glUniformMatrix4fv(lightModel, 1, GL_FALSE, dynamicLightModel)
424     glBindVertexArray(lightVAO)
425     glDrawArrays(GL_TRIANGLES, 0, 36)
426     glBindVertexArray(0)
```

Результат работы приложения изображен на Рис. 10, 11, 12 и 13.

Полный код программы приведен в приложении В.

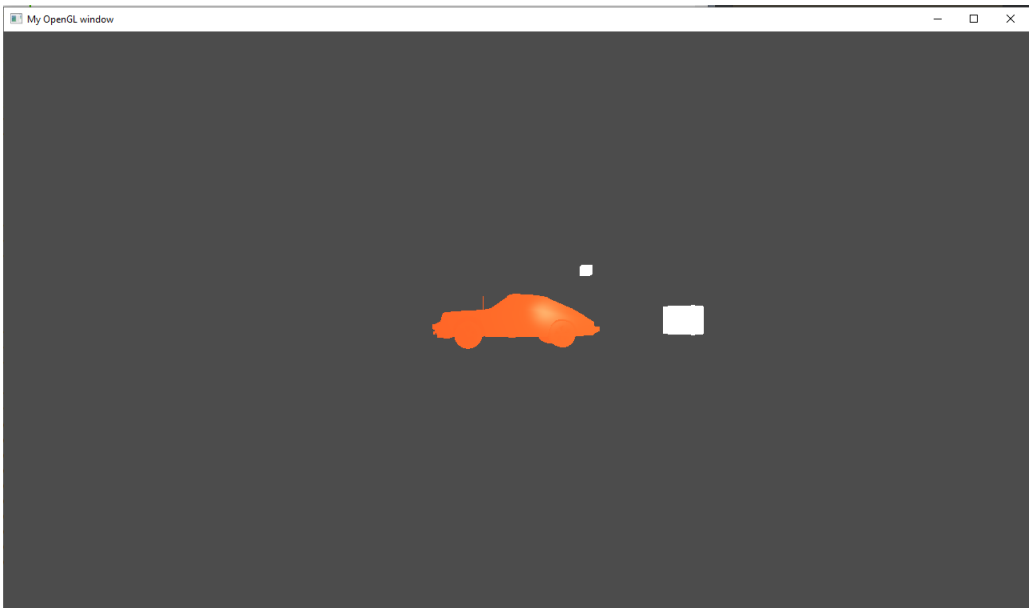

Рисунок 10 – Окно с изображением автомобиля и двух источников света. Свет направлен на заднее крыло

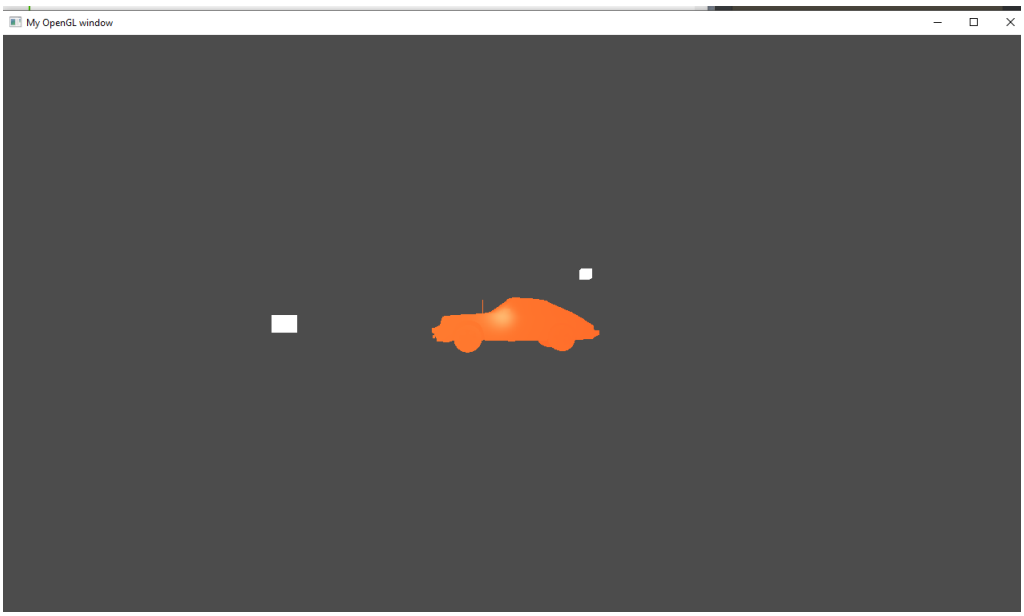

Рисунок 11 – Окно с изображением автомобиля и двух источников света. Свет направлен на дверь

Рисунок 12 – Окно с изображением автомобиля и двух источников света. Камера перемещена ближе к автомобилю, свет падает на переднее колесо

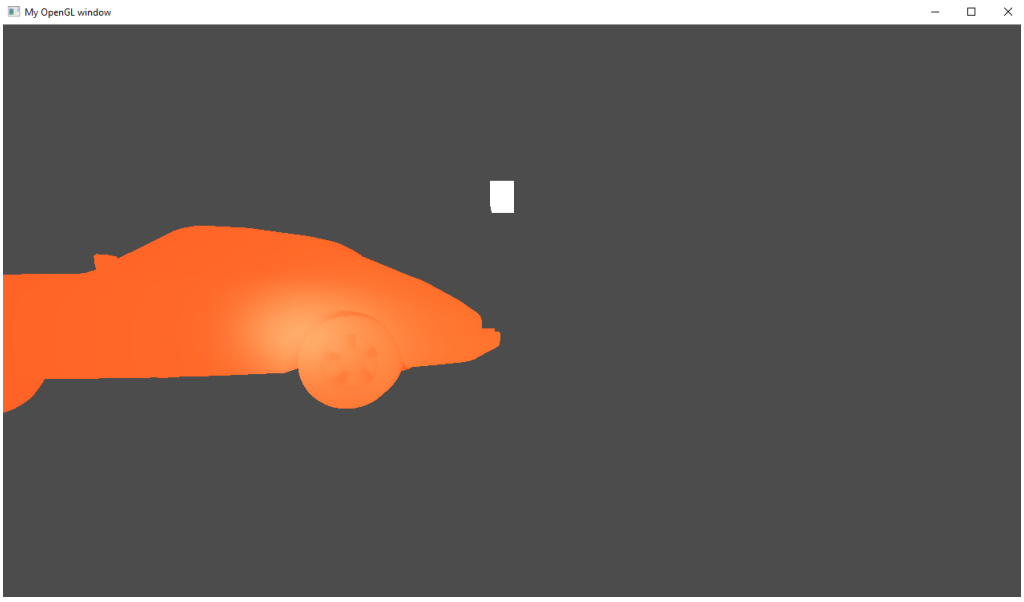

Рисунок 13 – Окно с изображением автомобиля и двух источников света. Камера перемещена вправо, свет падает на заднее колесо

ЗАКЛЮЧЕНИЕ

В ходе выполнения работы все поставленные задачи были выполнены: изучен функционал языка Python, освоены библиотеки для создания графического интерфейса приложения, реализованного на языке Python, изучены и реализованы в ходе создания приложения методы библиотеки Tkinter, а также возможности библиотеки glfw, предоставляющей инструменты для работы со спецификацией OpenGL.

Для создания приложения с использованием библиотеки Tkinter были использованы следующие элементы интерфейса:

— «Tkinter»;
— «Label»;
— «Entry»;
— «Button»;
— «Canvas».

Для создания приложений с использованием библиотеки glfw были использованы следующие инструменты:

— методы для создания окна и инициализации библиотеки glfw;
— методы для генерации, инициализации и создания массивов вершин: `glGenVertexArray()` и `glVertexAttribPointer()`;
— методы для генерации, инициализации и создания буферов: `glGenBuffers()`, `glBindBuffer()` и `glBufferData()`;
— вершинные и фрагментные шейдеры, методы для взаимодействия с ними: `glGetUniformLocation()`, `glUniformMatrix4fv()`, `glUniform3fv()`, `glUniform1f()`;
— методы для отрисовки объектов и их параметры: `glDrawArrays()` и `glDrawElements()`, параметры `GL_TRIANGLES` и `GL_LINE_STRIP`.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1 Что такое Tkinter [Электронный ресурс]. — URL: https://younglinux.info/tkinter/tkinter.php (Дата обращения 1.05.2020). Загл. с экр. Яз. рус.

2 Модуль Tkinter. Создание прафического интерфейса пользователя с помощью языка программирования Python [Электронный ресурс]. — URL: http://kabinet-vplaksina.narod.ru/olderfiles/5/Modul_tkinter.pdf (Дата обращения 1.05.2020). Загл. с экр. Яз. рус.

3 Введение в Tkinter [Электронный ресурс]. — URL: https://habr.com/ru/post/133337/ (Дата обращения 1.05.2020). Загл. с экр. Яз. рус.

4 Уроки для изучения Tkinter [Электронный ресурс]. — URL: https://python-scripts.com/tkinter-introduction (Дата обращения 1.05.2020). Загл. с экр. Яз. рус.

5 OpenGL documentation [Электронный ресурс]. — URL: https://www.opengl.org.ru/docs/pg/chapter1.html (Дата обращения 7.05.2020). Загл. с экр. Яз. англ.

6 PyOpenGL documentation. — URL: http://pyopengl.sourceforge.net/documentation/index.html (Дата обращения 10.05.2020). Загл. с экр. Яз. англ.

7 Matrices – Pyrr 0.10.1 documentation [Электронный ресурс]. — URL: https://pyrr.readthedocs.io/en/latest/oo_api_matrix.html#module-pyrr.objects.matrix44 (Дата обращения 4.05.2020). Загл. с экр. Яз. англ.

8 *Порев, В. Н.* Компьютерная графика. Учебное пособие / В. Н. Порев. — Санкт-Петербург: БВХ-Петербург, 2004.

9 LearnOpenGl – Basic Lighting [Электронный ресурс]. — URL: https://learnopengl.com/Lighting/Basic-Lighting (Дата обращения 10.05.2020). Загл. с экр. Яз. англ.

10 LearnOpenGl – Materials [Электронный ресурс]. — URL: https://learnopengl.com/Lighting/Materials (Дата обращения 8.05.2020). Загл. с экр. Яз. англ.

11 LearnOpenGl – Light casters [Электронный ресурс]. — URL: https://learnopengl.com/Lighting/Light-casters (Дата обращения 10.05.2020). Загл. с экр. Яз. англ.

12 Learn OpenGL. часть 2.3. – Материалы [Электронный ресурс]. — URL: https://habr.com/ru/post/336166/ (Дата обращения 6.05.2020). Загл. с экр. Яз. рус.

13 *Kessenich, J.* OpenGL Programming Guide: The Official Guide to Learning OpenGL, 9th edition / J. Kessenich, G. Sellers, D. Shreiner. — Addison-Wesley, 2016.

ПРИЛОЖЕНИЕ А
Программный код файла FirstApp.py на языке Python

```
from tkinter import *
from math import *

#Создание окна
window = Tk()
window.title("2d picture") #Заголовок
window.geometry("1000x1000") #Размер
window.configure(background = "gray") #Цвет заднего фона
windowCanvas = Canvas(window, width = 1000, height = 1000) #Размер холста
windowCanvas.pack()

#Загрузка координат
def loadCoord():
    figure = []
    path = [40, 130, 30, 110,
            30, 110, 60, 120,
            60, 120, 70, 120,
            70, 120, 80, 130,
            80, 130, 80, 140,
            80, 140, 70, 170,
            70, 170, 40, 180,
            40, 180, 30, 180,
            30, 180, 20, 170,
            20, 170, 20, 160,
            20, 160, 10, 130,
            10, 130, 30, 140,
            30, 140, 40, 130,
            150, 100, 130, 80,
            130, 80, 90, 80,
            90, 80, 70, 100,
            70, 100, 60, 120,
            50, 177, 50, 190,
            50, 190, 30, 220,
            30, 220, 10, 240,
            10, 240, 10, 250,
            10, 250, 20, 250,
            50, 190, 30, 240,
            30, 240, 20, 250,
            20, 250, 20, 260,
            20, 260, 50, 260,
            50, 260, 50, 240,
            50, 240, 80, 180,
            80, 180, 90, 170,
            90, 170, 110, 160,
```

```
            110, 160, 120, 160,
            120, 160, 140, 170,
            140, 170, 150, 180,
            150, 180, 160, 210,
            160, 210, 150, 240,
            150, 240, 140, 240,
            140, 240, 130, 250,
            130, 250, 130, 260,
            130, 260, 170, 260,
            170, 260, 170, 240,
            170, 240, 180, 210,
            180, 210, 170, 140,
            170, 140, 160, 120,
            160, 120, 180, 90,
            180, 90, 170, 50,
            170, 50, 160, 30,
            160, 30, 140, 20,
            140, 20, 110, 10,
            110, 10, 100, 10,
            100, 10, 70, 30,
            70, 30, 80, 40,
            80, 40, 100, 30,
            100, 30, 120, 30,
            120, 30, 140, 40,
            140, 40, 150, 50,
            150, 50, 160, 90,
            160, 90, 150, 100,
            140, 170, 150, 200,
            150, 200, 140, 230,
            140, 230, 130, 230,
            130, 230, 120, 240,
            120, 240, 120, 250,
            120, 250, 130, 250,
            50, 170, 60, 170,
            60, 170, 60, 150,
            60, 150, 50, 160,
            50, 160, 70, 160,
            70, 160, 70, 150,
            60, 120, 60, 140,
            60, 140, 40, 140,
            40, 140, 40, 160,
            40, 160, 20, 160,
            20, 160, 30, 140,
            30, 140, 40, 150,
            30, 140, 20, 140,
            20, 140, 20, 160,
            40, 130, 40, 120,
            40, 120, 60, 120,
```

```
            60, 120, 40, 130,
            40, 130, 50, 140,
            20, 170, 30, 160,
            60, 130, 70, 120]
    path1 = [60, 150, 80, 110,
             60, 150, 90, 120,
             60, 150, 100, 130,
             50, 160, 10, 180,
             50, 160, 20, 190,
             50, 160, 30, 200]
    path2 = [90, 250, 100, 260,
             100, 260, 110, 260,
             110, 260, 120, 250,
             95, 250, 100, 254,
             100, 254, 110, 254,
             110, 254, 115, 250]
    path3 = [30, 160, 30, 150,
             30, 150, 40, 150,
             40, 155, 30, 150,
             30, 150, 35, 160,
             50, 140, 50, 130,
             50, 130, 60, 130,
             60, 135, 50, 130,
             50, 130, 55, 140]
    path4 = [90, 220, 80, 210,
             80, 210, 70, 220,
             70, 220, 76, 232,
             100, 240, 110, 250,
             110, 250, 120, 230,
             120, 230, 110, 220,
             110, 220, 97, 227]
    path5 = [130, 230, 130, 220,
             130, 220, 140, 190,
             130, 220, 110, 210,
             120, 200, 90, 200,
             90, 200, 80, 190,
             80, 190, 90, 180,
             90, 180, 100, 190,
             100, 190, 110, 190,
             110, 190, 120, 180,
             120, 180, 130, 190,
             130, 190, 120, 200,
             110, 200, 110, 220,
             110, 220, 100, 220,
             100, 220, 100, 200,
             100, 210, 80, 230,
             80, 230, 60, 240,
             60, 240, 80, 250,
```

```
            80, 250, 120, 250,
            70, 235, 70, 245,
            80, 230, 80, 240,
            80, 240, 90, 240,
            90, 240, 90, 230,
            90, 230, 80, 230]
path6 = [34, 230, 20, 240,
            20, 240, 30, 240,
            50, 250, 30, 250,
            30, 250, 50, 240,
            55, 230, 40, 230,
            40, 230, 60, 220,
            65, 210, 50, 210,
            50, 210, 70, 200,
            74.5, 191, 50, 200,
            50, 200, 80, 180,
            65, 110, 80, 120,
            80, 120, 90, 130,
            90, 130, 80, 110,
            80, 110, 70, 100,
            80, 90, 100, 130,
            100, 130, 90, 80,
            100, 80, 110, 140,
            110, 140, 110, 80,
            120, 80, 120, 130,
            120, 130, 130, 80,
            140, 90, 130, 110,
            130, 110, 130, 130,
            130, 130, 140, 110,
            140, 110, 150, 100,
            160, 120, 140, 130,
            140, 130, 165, 130,
            170, 140, 140, 150,
            140, 150, 171.5, 150,
            173, 160, 150, 160,
            150, 160, 174.4, 170,
            174.40, 170, 160, 180,
            160, 180, 176, 180,
            177, 190, 160, 200,
            160, 200, 178.5, 200,
            180, 210, 160, 220,
            160, 220, 176.5, 220,
            173, 230, 160, 240,
            160, 240, 170, 240,
            156, 220, 150, 220,
            150, 220, 160, 210,
            180, 90, 160, 100,
            160, 100, 178, 80,
```

```
189                 170, 50, 160, 70,
190                 160, 70, 173, 60,
191                 160, 30, 160, 50,
192                 160, 50, 150, 25.5,
193                 140, 20, 140, 30,
194                 140, 30, 130, 17.0,
195                 110, 10, 120, 30,
196                 120, 30, 100, 10,
197                 70, 30, 100, 30,
198                 100, 30, 79, 24]
199     figure.append(path)
200     figure.append(path1)
201     figure.append(path2)
202     figure.append(path3)
203     figure.append(path4)
204     figure.append(path5)
205     figure.append(path6)
206     return figure
207 figure = loadCoord()
208
209 #Поле ввода смещения по оси Oy
210 yTranslateInput = (Entry(window, width=20, bd=3)) #Конструктор
211 yTranslateInput.pack()
212 yTranslateInput.place(x=10, y = 10) #Расположение
213
214 #Заголовок
215 labelYTranslate = Label(text="Oy translate", justify=LEFT) #Конструктор
216 labelYTranslate.place(x = yTranslateInput.winfo_width() + 150, y = 10) #Расположение
217
218 #Поле ввода смещения по оси Ox
219 xTranslateInput = (Entry(window, width=20, bd=3)) #Конструктор
220 xTranslateInput.pack()
221 xTranslateInput.place(x=10, y = 40) #Расположение
222
223 #Заголовок
224 labelXTranslate = Label(text="Ox translate", justify=LEFT) #Конструктор
225 labelXTranslate.place(x = xTranslateInput.winfo_width() + 150, y = 40) #Расположение
226
227 #Поле ввода значения угла поворота
228 rotateAngleInput = (Entry(window, width=20, bd=3)) #Конструктор
229 rotateAngleInput.pack()
230 rotateAngleInput.place(x = 10, y = 70) #Расположение
231
232 #Заголовок
233 labelRotateAngle = Label(text="Rotate angle", justify=LEFT) #Конструктор
234 labelRotateAngle.place(x = yTranslateInput.winfo_width() + 150, y = 70) #Расположение
235
236 #Поле ввода значения масштаба
```

```
237 scaleValueInput = (Entry(window,width=20,bd=3)) #Конструктор
238 scaleValueInput.pack()
239 scaleValueInput.place(x = 10, y = 100) #Расположение
240
241 #Заголовок
242 labelScaleValue = Label(text="Scale", justify=LEFT) #Конструктор
243 labelScaleValue.place(x = yTranslateInput.winfo_width() + 150, y = 100) #Расположение
244
245 #Функция вывода изображения на экран
246 def printImage(event):
247
248     #Переменные для хранения размеров окна
249     windowWidth = windowCanvas.winfo_width()
250     windowHeight = windowCanvas.winfo_height()
251
252     #Очистка холста
253     windowCanvas.delete('all')
254
255     #Переменная для хранения количества итераций цикла
256     i = 0
257
258     #Установка масштаба относительно размеров окна
259     if (windowWidth/windowHeight >= 270/190):
260         scale = windowHeight/190
261     else:
262         scale = windowWidth/270
263
264     #Получение масштаба из поля ввода
265     if scaleValueInput.get():
266         scaleValue = float(scaleValueInput.get())
267     else:
268         scaleValue = 1
269
270     #Стандартные значения координат x и y относительно размеров окна
271     initialXCoordinate = float((windowWidth - 190*scale*scaleValue)/2)
272     initialYCoordinate = float((windowHeight - 270*scale*scaleValue)/2)
273
274     #Получение угла поворота из поля ввода
275     if (rotateAngleInput.get()):
276         rotateAngleValue = float(rotateAngleInput.get())
277     else:
278         rotateAngleValue = 0
279
280     #Вычисление угла, косинуса и синуса
281     rotateAngleValue = rotateAngleValue * pi / 180
282     angleCos = (cos(rotateAngleValue))
283     angleSin = (sin(rotateAngleValue))
284
```

```
285     #Изменение координат относительно всех преобразований
286     initialXCoordinate = initialXCoordinate * angleCos - initialYCoordinate * angleSin
287     initialYCoordinate = initialYCoordinate * angleCos + initialXCoordinate * angleSin
288 
289     #Координаты цетра окна
290     windowCenterXCoordinate = windowWidth / 2
291     windowCenterYCoordinate = windowHeight / 2
292 
293     #Получение смещения по оси Ox из поля ввода
294     if xTranslateInput.get():
295         initialXCoordinate += int(xTranslateInput.get())
296 
297     # Получение смещения по оси Oy из поля ввода
298     if yTranslateInput.get():
299         initialYCoordinate -= int(yTranslateInput.get())
300 
301     #Цикл отрисовки линий относительно всех преобразований
302     for j in range(len(figure)):
303         while i < len(figure[j]):
304             windowCanvas.create_line((figure[j][i] * scale * scaleValue -
                ↪  windowCenterXCoordinate) * angleCos - (
305                         figure[j][i + 1] * scale * scaleValue - windowCenterYCoordinate) *
                            ↪  angleSin + windowCenterXCoordinate + initialXCoordinate,
306                         (figure[j][i + 1] * scale * scaleValue - windowCenterYCoordinate) *
                            ↪  angleCos + (
307                         figure[j][i] * scale * scaleValue - windowCenterXCoordinate) *
                            ↪  angleSin + windowCenterYCoordinate + initialYCoordinate,
308                         (figure[j][i + 2] * scale * scaleValue - windowCenterXCoordinate) *
                            ↪  angleCos - (
309                         figure[j][i + 3] * scale * scaleValue - windowCenterYCoordinate) *
                            ↪  angleSin + windowCenterXCoordinate + initialXCoordinate,
310                         (figure[j][i + 3] * scale * scaleValue - windowCenterYCoordinate) *
                            ↪  angleCos + (
311                         figure[j][i + 2] * scale * scaleValue - windowCenterXCoordinate) *
                            ↪  angleSin + windowCenterYCoordinate + initialYCoordinate,
312                         fill="orange", width=3)
313             i = i + 4
314         i = 0
315 
316 #Кнопка отрисовки
317 drawButton = Button(window) #Конструктор
318 drawButton["text"] = "Draw" #Надпись
319 drawButton.bind("<Button-1>", printImage) #Функция при нажатии
320 window.bind("drawButton", printImage) #Окно отображения кнопки
321 drawButton.pack()
322 drawButton.place(x = 40, y = 130) #Расположение
```

ПРИЛОЖЕНИЕ Б
Программный код файла SecondApp.py на языке Python

```
import glfw
from OpenGL.GL import *
from OpenGL.GL.shaders import *
import numpy as np
import pyrr

#Проверка корректности инициализации библиотеки glfw
if not glfw.init():
    raise Exception("glfw can not be initialized!")

#Создание окна
window = glfw.create_window(1280, 720, "My OpenGL window", None, None) #Конструктор
glfw.set_window_pos(window, 400, 200) #Расположение окна
glfw.make_context_current(window) #Выбор рабочегоокна
glClearColor(0.3, 0.45, 0.1, 1) #Цвет заднего фона

#Проверка корректности создания окна
if not window:
    glfw.terminate()
    raise Exception("glfw window can not be created!")

#Функция изменения размера окна
def window_resize(window, width, height):
    glViewport(0, 0, width, height)
    projection = pyrr.matrix44.create_perspective_projection_matrix(45, width / height,
    ↪  0.1, 100)
    glUniformMatrix4fv(proj_loc, 1, GL_FALSE, projection)

glfw.set_window_size_callback(window, window_resize)

#Считывание координат из файла
file_name = "car.txt"
file = open(file_name, 'r')
figure = []
flag = True
while(flag):
    cmd = file.readline().split()
    if(cmd[0] == 'figure'):
        flag = False
    elif(cmd[0] == 'path'):
        i = 0
        while(i < int(cmd[1])):
            cmd1 = file.readline().split()
            if (cmd1[0].isalpha() == False):
```

```
                for j in range(3):
                    figure.append(float(cmd1[j])*0.2)
            i += 1
vertices = np.array(figure, dtype=np.float32)

#Вершинный шейдер
vertex_src = """
# version 330
layout(location = 0) in vec3 aPosition;

uniform mat4 model;
uniform mat4 projection;

void main()
{
    gl_Position = projection * model * vec4(aPosition, 1.0);
}
"""

#Фрагментный шейдер
fragment_src = """
# version 330

out vec4 outColor;

void main()
{
    outColor = vec4(0.0f,0.0f,0.0f,1.0f);
}
"""

#Компиляция шейдеров и шейдерной программы
vertexShader = compileShader(vertex_src, GL_VERTEX_SHADER)
fragmentShader = compileShader(fragment_src, GL_FRAGMENT_SHADER)
shader = compileProgram(vertexShader, fragmentShader)

#Инициализация и заполнение буфера для передачи вершин изображения
objectVBO = glGenBuffers(1) #Конструктор
glBindBuffer(GL_ARRAY_BUFFER, objectVBO) #Выбор буфера
glBufferData(GL_ARRAY_BUFFER, vertices.nbytes, vertices, GL_STATIC_DRAW) #Заполнение
↪ буфера
glBufferData()

#Инициализация и заполнения массива вершин
glEnableVertexAttribArray(0)
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, vertices.itemsize * 3, ctypes.c_void_p(0))

#Матрицы проекции и преобразований
```

```
projection = pyrr.matrix44.create_perspective_projection_matrix(45, 1280/720, 0.1, 100)
translation = pyrr.matrix44.create_from_translation(pyrr.Vector3([0, 0, -3]))

#Выбор используемой шейдерной программы
glUseProgram(shader)

#Создание перменных для передачи значений в шейдерную программу
model_loc = glGetUniformLocation(shader, "model")
proj_loc = glGetUniformLocation(shader, "projection")

#Передача значения проекции в шейдерную программу
glUniformMatrix4fv(proj_loc, 1, GL_FALSE, projection)

#Цикл отрисовки
while not glfw.window_should_close(window):
    glfw.poll_events()

    #Очистка холста
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)

    #Матрица модели с вращением
    model = pyrr.matrix44.multiply(pyrr.Matrix44.from_y_rotation(0.8*glfw.get_time()),
    ↪  translation)

    #Передача матрицы модели в шейдерную программу
    glUniformMatrix4fv(model_loc, 1, GL_FALSE, model)

    #Отрисовка линий изображения
    glDrawArrays(GL_LINE_STRIP, 0, len(vertices))

    glfw.swap_buffers(window)
glfw.terminate()
```

ПРИЛОЖЕНИЕ В
Программный код файла ThirdApp.py на языке Python

```
import glfw
from OpenGL.GL import *
from OpenGL.GL.shaders import compileProgram, compileShader
import pyrr
import numpy as np
import glm

#Координаты вершин источника света
lightVertices = [
        -0.5, -0.5, -0.5,  0.0,  0.0, -1.0,
         0.5, -0.5, -0.5,  0.0,  0.0, -1.0,
         0.5,  0.5, -0.5,  0.0,  0.0, -1.0,
         0.5,  0.5, -0.5,  0.0,  0.0, -1.0,
        -0.5,  0.5, -0.5,  0.0,  0.0, -1.0,
        -0.5, -0.5, -0.5,  0.0,  0.0, -1.0,

        -0.5, -0.5,  0.5,  0.0,  0.0,  1.0,
         0.5, -0.5,  0.5,  0.0,  0.0,  1.0,
         0.5,  0.5,  0.5,  0.0,  0.0,  1.0,
         0.5,  0.5,  0.5,  0.0,  0.0,  1.0,
        -0.5,  0.5,  0.5,  0.0,  0.0,  1.0,
        -0.5, -0.5,  0.5,  0.0,  0.0,  1.0,

        -0.5,  0.5,  0.5, -1.0,  0.0,  0.0,
        -0.5,  0.5, -0.5, -1.0,  0.0,  0.0,
        -0.5, -0.5, -0.5, -1.0,  0.0,  0.0,
        -0.5, -0.5, -0.5, -1.0,  0.0,  0.0,
        -0.5, -0.5,  0.5, -1.0,  0.0,  0.0,
        -0.5,  0.5,  0.5, -1.0,  0.0,  0.0,

         0.5,  0.5,  0.5,  1.0,  0.0,  0.0,
         0.5,  0.5, -0.5,  1.0,  0.0,  0.0,
         0.5, -0.5, -0.5,  1.0,  0.0,  0.0,
         0.5, -0.5, -0.5,  1.0,  0.0,  0.0,
         0.5, -0.5,  0.5,  1.0,  0.0,  0.0,
         0.5,  0.5,  0.5,  1.0,  0.0,  0.0,

        -0.5, -0.5, -0.5,  0.0, -1.0,  0.0,
         0.5, -0.5, -0.5,  0.0, -1.0,  0.0,
         0.5, -0.5,  0.5,  0.0, -1.0,  0.0,
         0.5, -0.5,  0.5,  0.0, -1.0,  0.0,
        -0.5, -0.5,  0.5,  0.0, -1.0,  0.0,
        -0.5, -0.5, -0.5,  0.0, -1.0,  0.0,

```

```
        -0.5,  0.5, -0.5,  0.0,  1.0,  0.0,
         0.5,  0.5, -0.5,  0.0,  1.0,  0.0,
         0.5,  0.5,  0.5,  0.0,  1.0,  0.0,
         0.5,  0.5,  0.5,  0.0,  1.0,  0.0,
        -0.5,  0.5,  0.5,  0.0,  1.0,  0.0,
        -0.5,  0.5, -0.5,  0.0,  1.0,  0.0
  ]
lightCube = np.array(lightVertices, dtype= 'float32')

#Вершинный шейдер объекта
vertex_src = """
# version 330
layout(location = 0) in vec3 a_position;
layout(location = 2) in vec3 a_normal;
uniform mat4 model;
uniform mat4 projection;
uniform mat4 view;
uniform mat4 modelInv;
uniform mat4 modelView;

out vec3 Normal;
out vec3 fragPos;

void main()
{
    gl_Position = projection * view * model * vec4(a_position, 1.0);
    fragPos = vec3(modelView * vec4(a_position, 1.0f));
    Normal = mat3(modelInv) * a_normal;
}
"""

#Фрагментный шейдер объекта
fragment_src = """
#version 330 core

struct Light {
    vec3 position;
    vec3 ambient;
    vec3 diffuse;
    vec3 specular;
};

struct Material {
    vec3 ambient;
    vec3 diffuse;
    vec3 specular;
    float shininess;
};
```

```
93
94  in vec3 fragPos;
95  in vec3 Normal;
96
97  out vec4 color;
98
99  uniform Material material;
100 uniform vec3 viewPos;
101 uniform Light light;
102
103 void main() {
104
105     vec3 norm = normalize(Normal);
106     vec3 lightDir = normalize(light.position - fragPos);
107     float diff = max(dot(norm, lightDir), 0.0f);
108     float spec;
109     if (diff > 0.0){
110         vec3 viewDir = normalize(viewPos - fragPos);
111         vec3 reflectDir = reflect(-lightDir, norm);
112         spec = pow(max(dot(viewDir, reflectDir), 0.0f), material.shininess);
113     }
114     else
115         spec = 0.0;
116
117     vec3 specular = light.specular * (spec * material.specular);
118     vec3 diffuse = light.diffuse * (diff * material.diffuse);
119     vec3 ambient = light.ambient * material.ambient;
120
121     vec3 result = ambient + diffuse + specular;
122
123     color = vec4(result, 1.0f);
124 }
125 """
126
127 #Вершинный шейдер источника света
128 vertex_light = """
129 # version 330
130 layout (location = 0) in vec3 a_position;
131
132 uniform mat4 model;
133 uniform mat4 view;
134 uniform mat4 projection;
135
136 void main()
137 {
138   gl_Position = projection * view * model * vec4(a_position, 1.0);
139 }
140 """
```

```
141
142 #Фрагментный шейдер источника света
143 fragment_light = """
144 #version 330 core
145 out vec4 color;
146 void main() {
147   color = vec4(1.0f);
148 }
149 """
150
151 #Класс для хранения значений параметров материала
152 class Material:
153     def __init__(self):
154         self.ambient = []
155         self.diffuse = []
156         self.specular = []
157         self.shininess = []
158
159 #Класс объекта
160 class ObjLoader:
161     def __init__(self):
162         self.vert_coords = []
163         self.text_coords = []
164         self.norm_coords = []
165         self.material = Material()
166
167         self.model = []
168
169     #Загрузка информации об объекте из файла
170     def load_model(self):
171         file_name = "car_triangles.txt"
172         file = open(file_name, 'r')
173         ambient = []
174         diffuse = []
175         text_coords = []
176         specular = []
177         figure = []
178         shininess = 1.0
179         flag = True
180         while (flag):
181             cmd = file.readline().split()
182             if (cmd[0] == 'figure'):
183                 flag = False
184             elif (cmd[0] == 'color'):
185                 ambient = [float(cmd[1]) / 255, float(cmd[2]) / 255, float(cmd[3]) / 255]
186                 diffuse = ambient
187                 specular = ambient
188             elif (cmd[0] == 'ambient'):
```

```
189                 ambient = [float(cmd[1]), float(cmd[2]), float(cmd[3])]
190             elif (cmd[0] == 'diffuse'):
191                 diffuse = [float(cmd[1]), float(cmd[2]), float(cmd[3])]
192             elif (cmd[0] == 'specular'):
193                 specular = [float(cmd[1]), float(cmd[2]), float(cmd[3])]
194             elif (cmd[0] == 'shininess'):
195                 shininess = float(cmd[1])
196             elif (cmd[0] == 'mesh'):
197                 N = int(cmd[1])
198                 K = int(cmd[2])
199                 while (N > 0):
200                     cmd1 = file.readline().split()
201                     for i in range(6):
202                         if i < 3:
203                             self.vert_coords.append(float(cmd1[i]))
204                         else:
205                             self.norm_coords.append(float(cmd1[i]))
206                     N -= 1
207                 while (K > 0):
208                     cmd2 = file.readline().split()
209                     for i in range(3):
210                         text_coords.append(float(cmd2[i]))
211                     K -= 1
212                 self.text_coords = np.array(text_coords, dtype = 'uint32')
213                 self.vert_coords = np.array(self.vert_coords, dtype = 'float32')
214                 self.norm_coords = np.array(self.norm_coords, dtype = 'float32')
215                 self.material.ambient = ambient
216                 self.material.diffuse = diffuse
217                 self.material.specular = specular
218                 self.material.shininess = shininess
219                 figure.extend(self.vert_coords)
220                 figure.extend(self.norm_coords)
221                 self.model = np.array(figure, dtype = 'float32')
222 
223 #Функция изменения размеров окна
224 def window_resize(window, width, height):
225     glViewport(0, 0, width, height)
226 
227 #Проверка корректности инициализации библиотеки
228 if not glfw.init():
229     raise Exception("glfw can not be initialized!")
230 
231 #Создание окна
232 window = glfw.create_window(1280, 720, "My OpenGL window", None, None)
233 
234 #Проверка корректности создания окна
235 if not window:
236     glfw.terminate()
```

```
237     raise Exception("glfw window can not be created!")
238 
239 #Координаты камеры
240 global cameraXCoordinate
241 global cameraZCoordinate
242 
243 cameraXCoordinate = 0
244 cameraZCoordinate = 25
245 
246 #Управление камерой с помощью клавиатуры
247 def key_input(window, key, scancode, action, mode):
248     global cameraXCoordinate, cameraZCoordinate
249     if key == glfw.KEY_W and action == glfw.REPEAT:
250         cameraZCoordinate -= 0.5
251     if key == glfw.KEY_S and action == glfw.REPEAT:
252         cameraZCoordinate += 0.5
253     if key == glfw.KEY_A and action == glfw.REPEAT:
254         cameraXCoordinate -= 0.5
255     if key == glfw.KEY_D and action == glfw.REPEAT:
256         cameraXCoordinate += 0.5
257 
258 glfw.set_window_pos(window, 400, 200)
259 glfw.set_window_size_callback(window, window_resize)
260 glfw.set_key_callback(window, key_input)
261 glfw.make_context_current(window)
262 glEnable(GL_DEPTH_TEST)
263 glClearColor(0.3, 0.3, 0.3, 1.0)
264 
265 
266 #Компиляция шейдерных программ для объекта и источников света
267 shader = compileProgram(compileShader(vertex_src, GL_VERTEX_SHADER),
    ↪  compileShader(fragment_src, GL_FRAGMENT_SHADER))
268 lightShader = compileProgram(compileShader(vertex_light, GL_VERTEX_SHADER),
    ↪  compileShader(fragment_light, GL_FRAGMENT_SHADER))
269 
270 #Загрузка объекта
271 obj = ObjLoader()
272 obj.load_model()
273 
274 #Создание буферов и массива для объекта
275 objectVAO = glGenVertexArrays(1)
276 objectVBO = glGenBuffers(1)
277 objectEBO = glGenBuffers(1)
278 
279 #Заполнение вершинного буфера
280 glBindBuffer(GL_ARRAY_BUFFER, objectVBO)
281 glBufferData(GL_ARRAY_BUFFER, obj.model.nbytes, obj.model, GL_STATIC_DRAW)
282 
```

```
283 #Заполнение массива вершин
284 glBindVertexArray(objectVAO)
285 glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, obj.model.itemsize * 3, ctypes.c_void_p(0))
286 glEnableVertexAttribArray(0)
287
288 #Заполнение массива индексов
289 glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, obj.model.itemsize * 3,
    ↪  ctypes.c_void_p(20))
290 glEnableVertexAttribArray(1)
291
292 #Заполнение элементного буфера
293 glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, objectEBO)
294 glBufferData(GL_ELEMENT_ARRAY_BUFFER, obj.text_coords.nbytes, obj.text_coords,
    ↪  GL_STATIC_DRAW)
295 glBindBuffer(GL_ARRAY_BUFFER, 0)
296 glBindVertexArray(0)
297
298 #Создание буфера и массива для источников света
299 lightCubeVAO = glGenVertexArrays(1)
300 lightCubeVBO = glGenBuffers(1)
301
302 #Заполнение вершинного буфера
303 glBindBuffer(GL_ARRAY_BUFFER, lightCubeVBO)
304 glBufferData(GL_ARRAY_BUFFER, lightCube.itemsize * 4, lightCube, GL_STATIC_DRAW)
305
306 #Заполнение массива вершин
307 glBindVertexArray(lightCubeVAO)
308 glVertexAttribPointer(0,3,GL_FLOAT,GL_FALSE, lightCube.itemsize * 3, ctypes.c_void_p(0))
309 glEnableVertexAttribArray(0)
310
311 #Заполнение массива индексов
312 glVertexAttribPointer(1,3,GL_FLOAT,GL_FALSE, lightCube.itemsize * 3, ctypes.c_void_p(12))
313 glEnableVertexAttribArray(1)
314
315 #Создание массива для отображения блика света
316 lightVAO = glGenVertexArrays(1)
317
318 #Заполнение вершинного буфера
319 glBindBuffer(GL_ARRAY_BUFFER, lightCubeVBO)
320 glBufferData(GL_ARRAY_BUFFER, lightCube.nbytes, lightCube, GL_STATIC_DRAW)
321
322 #Заполнение массива вершин
323 glBindVertexArray(lightVAO)
324 glVertexAttribPointer(0,3,GL_FLOAT,GL_FALSE, lightCube.itemsize * 6, ctypes.c_void_p(0))
325 glEnableVertexAttribArray(0)
326
327 projection = pyrr.matrix44.create_perspective_projection_matrix(90, 1280/720, 1, 2000)
    ↪  #Матрица проекции
```

```
328 translation = pyrr.matrix44.create_from_translation(pyrr.Vector3([0, 0, 0])) #Матрица
    ↪ преобразований
329 staticLightModel =  pyrr.matrix44.create_from_translation(pyrr.Vector3([7, 5, -4]))
    ↪ #Матрица преобразований статичного источника света
330
331 #Выбор используемого шейдера
332 glUseProgram(shader)
333
334 #Инициализация переменных для передачи данных в шейдерную программу
335 ######################################################################
336 lightPos = glGetUniformLocation(shader, "light.position")
337 lightAmbient = glGetUniformLocation(shader, "light.ambient")
338 lightDiffuse = glGetUniformLocation(shader, "light.diffuse")
339 lightSpecular = glGetUniformLocation(shader, "light.specular")
340 lightView = glGetUniformLocation(lightShader, "view")
341 lightProjection = glGetUniformLocation(lightShader, "projection")
342 lightModel = glGetUniformLocation(lightShader, "model")
343
344 materialAmbient = glGetUniformLocation(shader, "material.ambient")
345 materialDiffuse = glGetUniformLocation(shader, "material.diffuse")
346 materialSpecular = glGetUniformLocation(shader, "material.specular")
347 materialShinnes = glGetUniformLocation(shader, "material.shininess")
348
349 sceneModel = glGetUniformLocation(shader, "model")
350 sceneProjection = glGetUniformLocation(shader, "projection")
351 sceneView = glGetUniformLocation(shader, "view")
352 modelView = glGetUniformLocation(shader, "modelView")
353 modelInv = glGetUniformLocation(shader, "modelInv")
354 viewPos = glGetUniformLocation(shader, "viewPos")
355 ######################################################################
356
357 #Передача неизменяемых данных в шейдерную программу
358 ######################################################################
359 glUniformMatrix4fv(sceneProjection, 1, GL_FALSE, projection)
360 glUniform3fv(lightAmbient, 1, obj.material.ambient)
361 glUniform3fv(lightDiffuse, 1, obj.material.diffuse)
362 glUniform3fv(lightSpecular, 1, obj.material.specular)
363 glUniform3fv(materialAmbient, 1, obj.material.ambient)
364 glUniform3fv(materialDiffuse, 1, obj.material.diffuse)
365 glUniform3fv(materialSpecular, 1, obj.material.specular)
366 glUniform1f(materialShinnes, obj.material.shininess)
367 ######################################################################
368
369 #Цикл отрисовки изображения
370 while not glfw.window_should_close(window):
371     glfw.poll_events()
372
373     glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)
```

```
374
375     #Матрица с положением камеры
376     view = pyrr.matrix44.create_look_at(pyrr.Vector3([cameraXCoordinate, 0,
        ↪  cameraZCoordinate]), pyrr.Vector3([cameraXCoordinate, 0, 0]), pyrr.Vector3([0, 1,
        ↪  0]))
377
378     #Матрицы для движения источника света вокруг заданной точки
379     move = pyrr.Matrix44.from_y_rotation(glfw.get_time() * 1)
380     lightM = pyrr.matrix44.create_from_translation(pyrr.Vector3([15,0,5]))
381     dynamicLightModel = move * lightM
382
383     #Матрица отображения света
384     inverseView = np.linalg.inv(view) * np.array(glm.vec4(0,0,0,1))
385
386     #Матрица позиции света на объекте
387     lightPosition = np.array(dynamicLightModel * staticLightModel * pyrr.Vector4([0, 0, 0,
        ↪  1]))
388     lightPosition = np.array([lightPosition[0],lightPosition[1],-lightPosition[2]])
389
390     #Передача в шейдерную программу данных об положении камеры и света
391     glUseProgram(shader)
392     glUniform3fv(lightPos, 1, lightPosition)
393     glUniform3fv(viewPos, 1, inverseView)
394     glUniformMatrix4fv(sceneView, 1, GL_FALSE, view)
395
396     #Матрица модели
397     model = pyrr.matrix44.multiply(pyrr.Matrix44.from_y_rotation(3), translation)
398     modeli = np.linalg.inv(model).transpose()
399
400     #Передача в шейдерную программу данных об объекте
401     glBindVertexArray(objectVAO)
402     glUniformMatrix4fv(sceneModel, 1, GL_FALSE, model)
403     glUniformMatrix4fv(modelView, 1, GL_FALSE, model)
404     glUniformMatrix4fv(modelInv, 1, GL_FALSE, modeli)
405
406     #Отрисовка объекта
407     glDrawElements(GL_TRIANGLES, len(obj.text_coords), GL_UNSIGNED_INT, None)
408     glBindVertexArray(0)
409
410     #Отрисовка динамичного источника света
411     glUseProgram(lightShader)
412     glUniformMatrix4fv(lightView, 1, GL_FALSE, view)
413     glUniformMatrix4fv(lightProjection, 1, GL_FALSE, projection)
414     glUniformMatrix4fv(lightModel, 1, GL_FALSE, staticLightModel)
415     glBindVertexArray(lightVAO)
416     glDrawArrays(GL_TRIANGLES, 0, 36)
417     glBindVertexArray(0)
418
```

```
419     #Отрисовка динамичного источника света
420     glUseProgram(lightShader)
421     glUniformMatrix4fv(lightView, 1, GL_FALSE, view)
422     glUniformMatrix4fv(lightProjection, 1, GL_FALSE, projection)
423     glUniformMatrix4fv(lightModel, 1, GL_FALSE, dynamicLightModel)
424     glBindVertexArray(lightVAO)
425     glDrawArrays(GL_TRIANGLES, 0, 36)
426     glBindVertexArray(0)
427 
428     glfw.swap_buffers(window)
429 glfw.terminate()
```