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

ОТЧЁТ ПО ЛАБОРАТОРНОЙ РАБОТЕ №1

по дисциплине Компьютерная графика

Тема: «Исследование математических методов представления и преобразования графических объектов на плоскости»

Студенты гр. 1307		Грунская Н.Д. Тростин М.Ю. Голубев М.А.
Преподаватель		Матвеева И.В.

Санкт-Петербург

2024

Цель работы:
Практическое закрепление теоретических знаний о представлении и преобразований графических объектов на плоскости

Постановка задачи:
Поворот плоского объекта (треугольника) относительно произвольной точки плоскости на заданный угол. Необходимо предусмотреть возможность редактирования положения точки

Краткая теоретическая информация
Для представления треугольниками с вершинами A, B, C используется матрица
$$\begin{bmatrix} x_A & y_A \\ x_B & y_B \\ x_C & y_C \end{bmatrix}$$

Для того, чтобы выполнить поворот треугольника вокруг некоторой точки d, необходимо для начала сместить все точки так, чтобы точка d лежала в начале координат. Выполнить это можно при помощи следующего преобразования:

$$\begin{bmatrix} x_A & y_A & 1 \\ x_B & y_B & 1 \\ x_C & y_C & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -x_d & -y_d & 1 \end{bmatrix} = \begin{bmatrix} x_A - x_d & y_A - y_d & 1 \\ x_B - x_d & y_B - y_d & 1 \\ x_C - x_d & y_C - y_d & 1 \end{bmatrix}$$

После чего можно осуществить поворот на произвольный угол α при помощи матрицы поворота:

$$\begin{bmatrix} x_A & y_A & 1 \\ x_B & y_B & 1 \\ x_C & y_C & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -x_d & -y_d & 1 \end{bmatrix} \cdot \begin{bmatrix} \cos(\alpha) & \sin(\alpha) & 0 \\ -\sin(\alpha) & \cos(\alpha) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Теперь преобразованный треугольник необходимо вернуть в исходную систему координат:

$$\begin{bmatrix} x_A & y_A & 1 \\ x_B & y_B & 1 \\ x_C & y_C & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -x_d & -y_d & 1 \end{bmatrix} \cdot \begin{bmatrix} \cos(\alpha) & \sin(\alpha) & 0 \\ -\sin(\alpha) & \cos(\alpha) & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ x_d & y_d & 1 \end{bmatrix}$$

Реализация алгоритма
Для реализации алгоритма был использован Python, а для визуализации результата был использован Tkinter
Функция поворота треугольника вокруг точки:

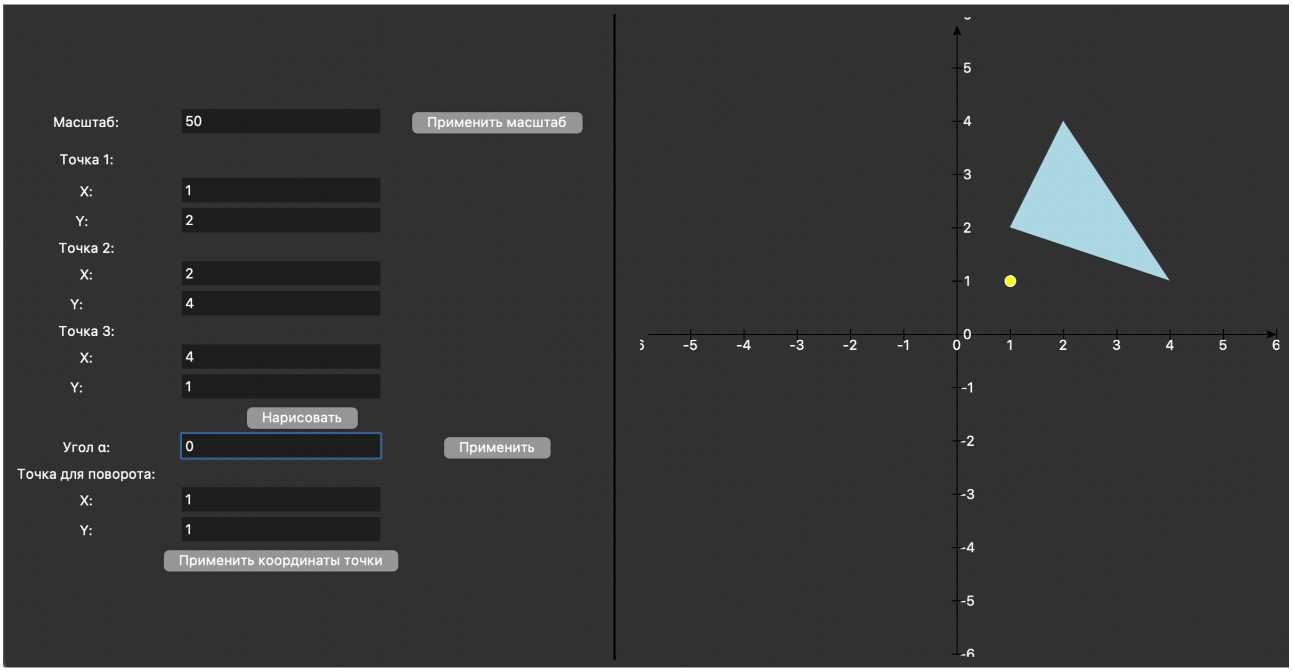
```
def roll_triangle():
```

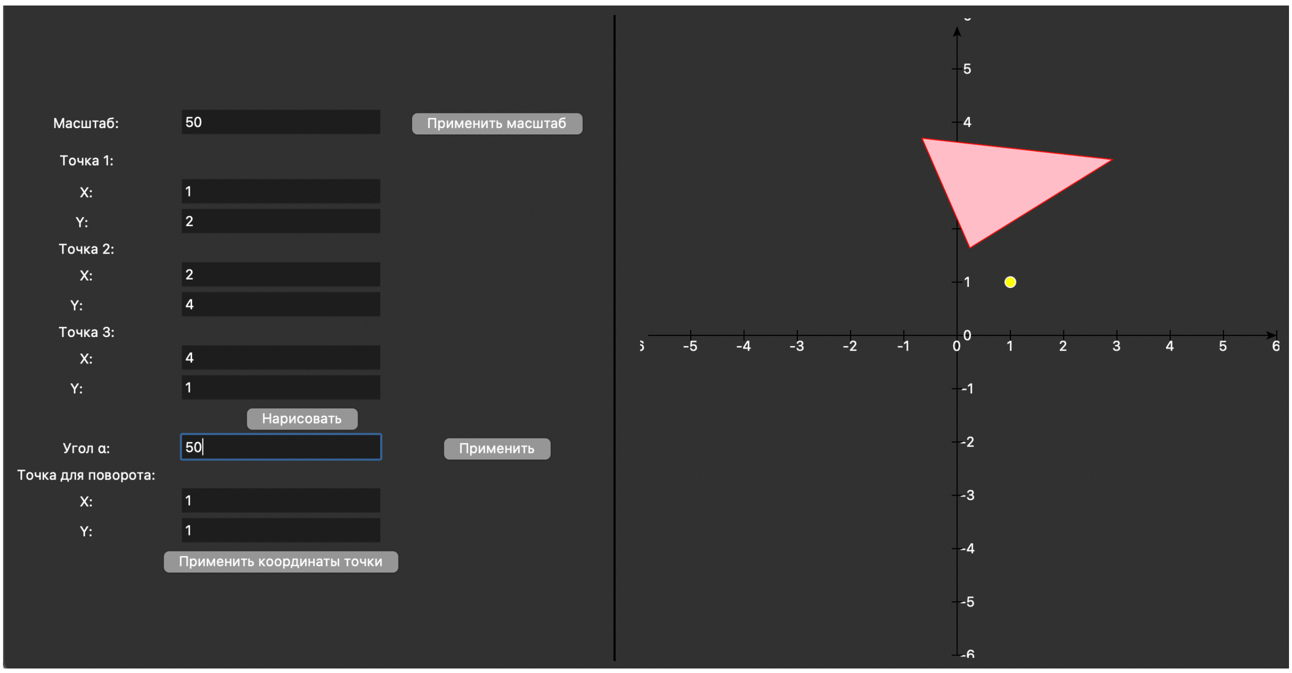
```
# Получаем координаты точки для поворота
set_dot()
# Получаем координаты вершин треугольника
get_coordinates()
draw_coord_lines()
# Получаем угол поворота
angle = float(entry_angle.get())
angle = math.radians(angle)

mat1 = [[x1, y1, 1], [x2, y2, 1], [x3, y3, 1]]
mat2 = [[1, 0, 0], [0, 1, 0], [dot_x * (-1), dot_y * (-1), 1]]
mat3 = [[math.cos(angle), math.sin(angle), 0],
[math.sin(angle) * (-1), math.cos(angle), 0], [0, 0, 1]]
mat4 = [[1, 0, 0], [0, 1, 0], [dot_x * 1, dot_y * 1, 1]]

result =
multiply_matrices(multiply_matrices(multiply_matrices(mat1, mat2),
mat3), mat4)
x1_new = result[0][0]
y1_new = result[0][1]
x2_new = result[1][0]
y2_new = result[1][1]
x3_new = result[2][0]
y3_new = result[2][1]

# Масштабируем координаты после вращения
x1_new_scaled = canvas_x / 2 + x1_new * scale
y1_new_scaled = canvas_y / 2 - y1_new * scale
x2_new_scaled = canvas_x / 2 + x2_new * scale
y2_new_scaled = canvas_y / 2 - y2_new * scale
x3_new_scaled = canvas_x / 2 + x3_new * scale
y3_new_scaled = canvas_y / 2 - y3_new * scale
```

```
    # Рисуем преобразованный треугольник
    canvas.create_polygon(x1_new_scaled, y1_new_scaled,
x2_new_scaled, y2_new_scaled, x3_new_scaled, y3_new_scaled,
                                   outline="red", fill="pink")
```

Полный исходный код программы представлен в приложении

Пример работы приложения:





Выводы:
Были практически закреплены теоретические знания о представлении и преобразований графических объектов на плоскости

Приложение:

Ссылка на видео:

https://youtu.be/VsRlVSNx7eg

Приложение

```
import tkinter as tk
import math



def set_scale():
    global scale
    scale = int(entry_scale_entry.get())
    draw_coord_lines()



def set_dot():
    global dot_x
    global dot_y
    dot_x = int(entry_dot_x.get())
    dot_y = int(entry_dot_y.get())
    radius = 5
    canvas.create_oval(canvas_x / 2 + dot_x * scale - radius,
canvas_y / 2 - dot_y * scale - radius,
                       canvas_y / 2 + int(entry_dot_x.get()) *
scale + radius, canvas_y / 2 - dot_y * scale + radius,
                       fill="yellow")



def multiply_matrices(matrix1, matrix2):
    result = [[0, 0, 0], [0, 0, 0], [0, 0, 0]]

    for i in range(3):
        for j in range(3):
            for k in range(3):
                result[i][j] += matrix1[i][k] * matrix2[k][j]

    return result
```

```
def draw_coord_lines():
    canvas.delete("all")
    canvas.create_line(10, canvas_y / 2, canvas_x, canvas_y / 2,
fill="black", arrow=tk.LAST)  # Ось x
    canvas.create_line(canvas_x / 2, canvas_y, canvas_y / 2, 10,
fill="black", arrow=tk.LAST)  # Ось y

    # Рисуем деления на оси
    for i in range(-15, 15):
        canvas.create_line(canvas_x / 2 + i * scale, canvas_y / 2
- 5, canvas_x / 2 + i * scale, canvas_y / 2 + 5,
                               fill="black")  # Деления на оси x
        canvas.create_line(canvas_x / 2 - 5, canvas_y / 2 + i *
scale, canvas_x / 2 + 5, canvas_y / 2 + i * scale,
                               fill="black")  # Деления на оси y
        canvas.create_text(canvas_x / 2 + i * scale, canvas_y / 2
+ 10, text=str(i))  # Подписи к делениям на оси x
        canvas.create_text(canvas_x / 2 + 10, canvas_y / 2 - i *
scale, text=str(i))  # Подписи к делениям на оси y


def get_coordinates():
    global x1, x2, x3, y1, y2, y3
    x1 = float(entry1_x.get())
    y1 = float(entry1_y.get())
    x2 = float(entry2_x.get())
    y2 = float(entry2_y.get())
    x3 = float(entry3_x.get())
    y3 = float(entry3_y.get())


def draw_triangle():
```

```
    get_coordinates()
    draw_coord_lines()
    # Масштабируем координаты
    x1_scaled = canvas_x / 2 + x1 * scale
    y1_scaled = canvas_y / 2 - y1 * scale
    x2_scaled = canvas_x / 2 + x2 * scale
    y2_scaled = canvas_y / 2 - y2 * scale
    x3_scaled = canvas_x / 2 + x3 * scale
    y3_scaled = canvas_y / 2 - y3 * scale
    canvas.create_polygon(x1_scaled, y1_scaled, x2_scaled,
y2_scaled, x3_scaled, y3_scaled, fill="lightblue")


def roll_triangle():
    # Получаем координаты точки для поворота
    set_dot()
    # Получаем координаты вершин треугольника
    get_coordinates()
    draw_coord_lines()
    # Получаем угол поворота
    angle = float(entry_angle.get())
    angle = math.radians(angle)

    mat1 = [[x1, y1, 1], [x2, y2, 1], [x3, y3, 1]]
    mat2 = [[1, 0, 0], [0, 1, 0], [dot_x * (-1), dot_y * (-1), 1]]
    mat3 = [[math.cos(angle), math.sin(angle), 0],
[math.sin(angle) * (-1), math.cos(angle), 0], [0, 0, 1]]
    mat4 = [[1, 0, 0], [0, 1, 0], [dot_x * 1, dot_y * 1, 1]]

    result =
multiply_matrices(multiply_matrices(multiply_matrices(mat1, mat2),
mat3), mat4)

    x1_new = result[0][0]
```

```
    y1_new = result[0][1]
    x2_new = result[1][0]
    y2_new = result[1][1]
    x3_new = result[2][0]
    y3_new = result[2][1]

    # Масштабируем координаты после вращения
    x1_new_scaled = canvas_x / 2 + x1_new * scale
    y1_new_scaled = canvas_y / 2 - y1_new * scale
    x2_new_scaled = canvas_x / 2 + x2_new * scale
    y2_new_scaled = canvas_y / 2 - y2_new * scale
    x3_new_scaled = canvas_x / 2 + x3_new * scale
    y3_new_scaled = canvas_y / 2 - y3_new * scale

    # Рисуем преобразованный треугольник
    canvas.create_polygon(x1_new_scaled, y1_new_scaled,
x2_new_scaled, y2_new_scaled, x3_new_scaled, y3_new_scaled,
                          outline="red", fill="pink")



root = tk.Tk()
root.title("Треугольник")


input_frame = tk.Frame(root)
input_frame.grid(row=0, column=0, padx=10, pady=10)


canvas_x = 600
canvas_y = 600
canvas = tk.Canvas(root, width=canvas_x, height=canvas_y)
canvas.grid(row=0, column=2, padx=10, pady=10, columnspan=2,
sticky="w")


# Разделительная линия
```

```
separator = tk.Frame(root, height=canvas_y, width=2, bg="black")
separator.grid(row=0, column=1, padx=10, pady=10, sticky="ns")

entry_scale = tk.Label(input_frame, text="Масштаб:")
entry_scale.grid(row=0, column=0, padx=5, pady=5)
entry_scale_entry = tk.Entry(input_frame)
entry_scale_entry.grid(row=0, column=1, padx=5, pady=5)
button_scale = tk.Button(input_frame, text="Применить масштаб",
command=set_scale)
button_scale.grid(row=0, column=2, padx=5, pady=5)

label1 = tk.Label(input_frame, text="Точка 1:")
label1.grid(row=1, column=0, padx=5, pady=5)

label1_x = tk.Label(input_frame, text="X:")
label1_x.grid(row=2, column=0)

entry1_x = tk.Entry(input_frame)
entry1_x.grid(row=2, column=1)

label1_y = tk.Label(input_frame, text="Y:  ")
label1_y.grid(row=3, column=0)
entry1_y = tk.Entry(input_frame)
entry1_y.grid(row=3, column=1)

label2 = tk.Label(input_frame, text="Точка 2:")
label2.grid(row=4, column=0)

label2_x = tk.Label(input_frame, text="X:")
label2_x.grid(row=5, column=0)
entry2_x = tk.Entry(input_frame)
entry2_x.grid(row=5, column=1)
label2_y = tk.Label(input_frame, text="Y:     ")
```

```
label2_y.grid(row=6, column=0)
entry2_y = tk.Entry(input_frame)
entry2_y.grid(row=6, column=1)

label3 = tk.Label(input_frame, text="Точка 3:")
label3.grid(row=7, column=0)

label3_x = tk.Label(input_frame, text="X:")
label3_x.grid(row=8, column=0)
entry3_x = tk.Entry(input_frame)
entry3_x.grid(row=8, column=1)
label3_y = tk.Label(input_frame,text="Y:     ")
label3_y.grid(row=9, column=0)
entry3_y = tk.Entry(input_frame)
entry3_y.grid(row=9, column=1)

button_draw = tk.Button(input_frame, text="Нарисовать",
command=draw_triangle)

button_draw.grid(row=10, columnspan=3)

label_angle = tk.Label(input_frame, text="Угол α:")
label_angle.grid(row=11, column=0)  # Поле для ввода угла альфа
entry_angle = tk.Entry(input_frame)
entry_angle.grid(row=11, column=1)
button_scale = tk.Button(input_frame, text="Применить",
command=roll_triangle)
button_scale.grid(row=11, column=2)

global dot_x
global dot_y
label_dot = tk.Label(input_frame, text="Точка для поворота:")
```

```
label_dot.grid(row=12, column=0)
label_dot_x = tk.Label(input_frame, text="X:")
label_dot_x.grid(row=13, column=0)
entry_dot_x = tk.Entry(input_frame)
entry_dot_x.grid(row=13, column=1)
label_dot_y = tk.Label(input_frame, text="Y:")
label_dot_y.grid(row=14, column=0)
entry_dot_y = tk.Entry(input_frame)
entry_dot_y.grid(row=14, column=1)
button_dot = tk.Button(input_frame, text="Применить координаты точки", command=set_dot)
button_dot.grid(row=15, column=1)


# Рисуем координатную ось
canvas.create_line(10, canvas_y / 2, canvas_x - 10, canvas_y / 2, fill="black", arrow=tk.LAST)  # Ось x
canvas.create_line(canvas_x / 2, canvas_y - 10, canvas_x / 2, 10, fill="black", arrow=tk.LAST)  # Ось y


scale = 20  # Масштаб для координат


# Рисуем деления на оси
for i in range(-20, 20):
    canvas.create_line(canvas_x / 2 + i * scale, canvas_y / 2 - 5, canvas_x / 2 + i * scale, canvas_y / 2 + 5,
                       fill="black")  # Деления на оси x
    canvas.create_line(canvas_x / 2 - 5, canvas_y / 2 + i * scale, canvas_x / 2 + 5, canvas_y / 2 + i * scale,
                       fill="black")  # Деления на оси y
    canvas.create_text(canvas_x / 2 + i * scale, canvas_y / 2 + 10, text=str(i))  # Подписи к делениям на оси x
```

```
    canvas.create_text(canvas_x / 2 + 10, canvas_y / 2 - i * scale, text=str(i))  # Подписи к делениям на оси y

root.mainloop()
```

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

ОТЧЁТ ПО ЛАБОРАТОРНОЙ РАБОТЕ №2

по дисциплине Компьютерная графика

Тема: «Формирования различных кривых с использованием ортогонального проектирования на плоскость визуализации (экране дисплея)

Студенты гр. 1307	_______________	Грунская Н.Д. Тростин М.Ю. Голубев М.А.
Преподаватель	_______________	Матвеева И.В.

Санкт-Петербург

2024

Цель работы:
Практическое закрепление теоретических знаний о формировании кривых с использованием ортогонального проектирования на плоскость визуализации.

Постановка задачи:

Сформировать на плоскости кривую Безье на основе задающей ломаной, определяемой 3 и большим количеством точек. Обеспечить редактирование координат точек задающей ломаной с перерисовкой сплайна Безье.

Краткая теоретическая информация:

Точки задания этих кривых Безье только определяют ход кривой, сама строящаяся кривая в общем случае не проходит через внутренние точки задающего многоугольника

Особенности:

1. Подходит по касательной к внешним ребрам (сторонам) задающего многоугольника, а остальные точки определяют ход кривой. Они позволяют качественно оценить ход кривой в зависимости от вида задающего многоугольника.
2. Кривая задается параметрически в функции от независимого параметра.
3. Это кривая n-ой степени, т.е. сколько ребер у задающего многоугольника – такой степени и получается кривая. Влиять на степень кривой можно только изменением количества задающих ее точек.

Математически такая кривая описывается параметрическим уравнением:

$P(t) = \sum_{i=0}^{n} P_i \times N_{i,n}(t)$,где P(t) – полиномиальная функция,
Pi – вес (координаты) i-ой точки задания,
Ni, n – весовой коэффициент i-той вершины,
i – номер вершины (точки),
n – количество сторон задающего многоугольника
t – задающий параметр, причем 0 <= t <= 1

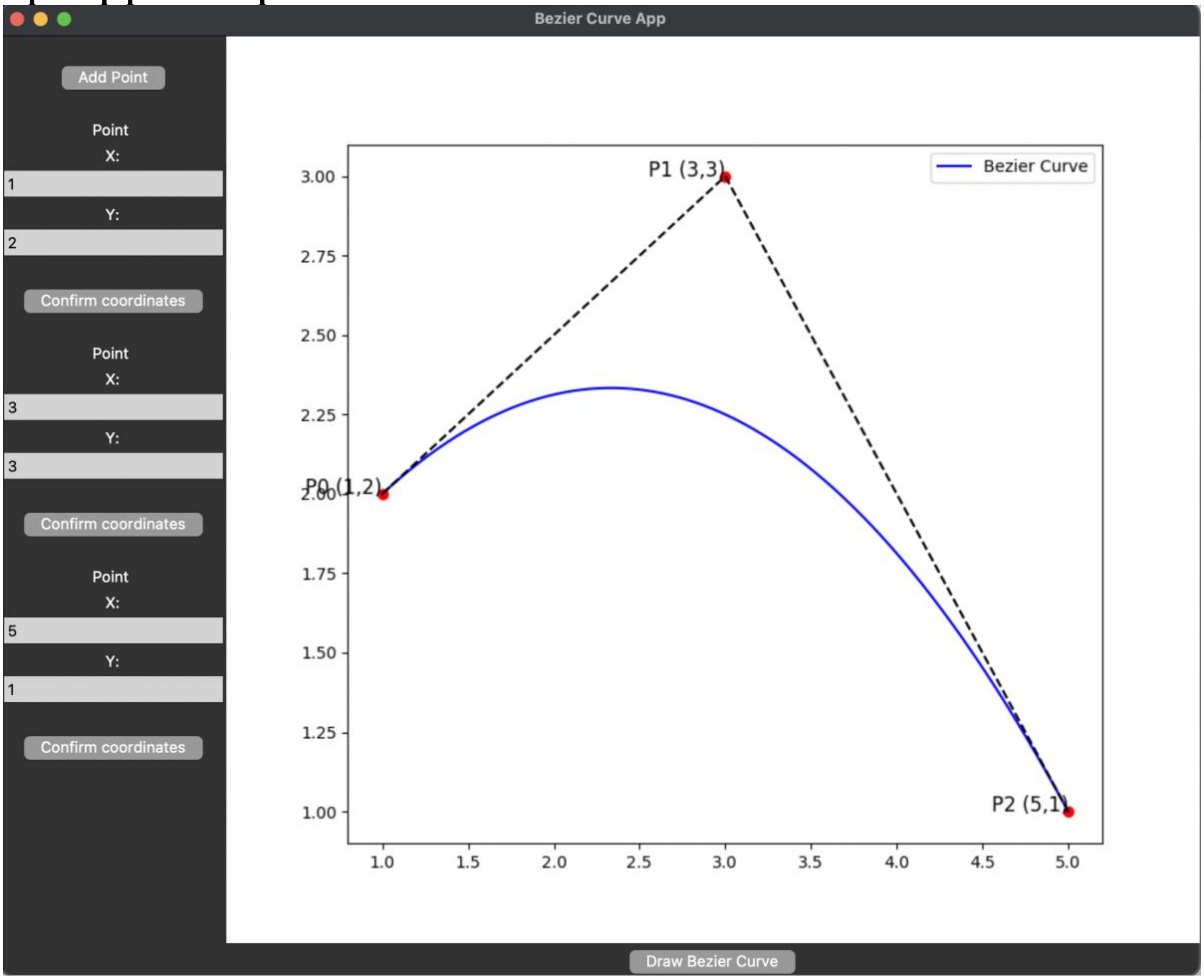
$$N_{i,n}(t) = \frac{n!}{i!(n-i)!} \times t^i \times (1-t)^{n-}$$

Если множество точек задания (n), то строим либо кривую n-ой степени, либо можем построить кривую невысокой степени (обычно кубическую, так как она позволяет обеспечить перегиб и может быть быстро построена). В последнем случаедля такой кривой нужно только четыре последовательные точки задания.

Если n > 4, то мы находимся в противоречии с количеством точек, так как формируя сегменты на основе точек задающего многоугольника в точках стыковки таких сплайнов кривая будет иметь излом (разрыв производных). Чтобы избежать такой ситуации учитывают то свойство кривых Безье, что к внешним ребрам задающего много угольника они подходят по касательной. Поэтому в третье ребро описания при формировании кубической кривой Безье добавляют дополнительную точку P` и ее считают за последнею точку текущего характеристического многоугольника и первой точкой следующего характеристического многоугольника. Обычно ее берут по середине ребра. Тогда кривая будет плавно переходить от предыдущей кубической кривой (кривой третьей степени) к последующей. А вся такая кривая представляет собой составную плавную кривую, состоящую из ряда сегментов, называется составной кривой Безье.

Для расчета последнего сегмента такой составной кривой Безье можно либо понизить степень строящегося участка кривой до второй, так как может остаться только три незадействованной точки, либо при расчете использовать последнюю точку дважды.

Пример работы приложения:



Выводы:

Были практически закреплены теоретические знания о формировании кривых с использованием ортогонального проектирования на плоскость визуализации.

Приложение

Ссылка на видео: : https://youtu.be/irFx5sxIQfg

Исходный код:

beizer_functions.py

```
import scipy.special


def bernstein_poly(i, n, t):
    return scipy.special.comb(n, i) * t ** i * (1 - t) ** (n - i)
```

lab2.py

```
import tkinter as tk
from tkinter import messagebox
import numpy as np
from matplotlib import pyplot as plt
from matplotlib.backends.backend_tkagg import FigureCanvasTkAgg
from bezier_functions import bernstein_poly


class Point(tk.Frame):
    def __init__(self, parent):
        super().__init__(parent)
        self.coordinates = [0, 0]
        self.confirm_button = tk.Button
        self.number = ''
        self.point_frame = tk.Frame(self, width=100, height=50)
        self.point_frame.pack()
        new_point_label = tk.Label(self.point_frame, text="Point " +
self.number)
        new_point_label.pack()

        x_label = tk.Label(self.point_frame, text="X:")
        x_label.pack()
        self.x_entry = tk.Entry(self.point_frame)
        self.x_entry.pack()

        y_label = tk.Label(self.point_frame, text="Y:")
        y_label.pack()
        self.y_entry = tk.Entry(self.point_frame)
        self.y_entry.pack()

        self.confirm_button = tk.Button(self.point_frame, text="Confirm
coordinates", command=self.confirm_coordinates)
        self.confirm_button.pack(pady=20)

    def confirm_coordinates(self):
        self.coordinates[0] = int(self.x_entry.get())
        self.coordinates[1] = int(self.y_entry.get())
        self.x_entry.config(bg="lightgrey", fg="black")
        self.y_entry.config(bg="lightgrey", fg="black")


class BezierCurveApp(tk.Tk):
    def __init__(self):
        super().__init__()

        self.canvas = None
        self.title("Bezier Curve App")
        self.geometry("1020x800")
```

```
        self.figs = []
        self.points = []
        self.point_coordinates = []

        self.menu_frame = tk.Frame(self, width=200, height=600)
        self.menu_frame.pack(side="left", fill="y")

        self.add_point_button = tk.Button(self.menu_frame, text="Add 
Point", command=self.add_point)
        self.add_point_button.pack(pady=20)

        self.draw_curve_button = tk.Button(self, text="Draw Bezier Curve", 
command=self.draw_curve)
        self.draw_curve_button.pack(side="bottom")

    def make_coordinates(self):
        points_correct = []
        for i in range(len(self.points)):
            a = [self.points[i].coordinates[0], 
self.points[i].coordinates[1]]
            points_correct.append(a)
        print(points_correct)
        return points_correct

    def add_point(self):
        new_point = Point(self.menu_frame)
        self.points.append(new_point)
        new_point.pack(side='top', fill="both", expand=False)

    def draw_curve(self):
        if len(self.points) < 2:
            messagebox.showerror("Error", "At least 2 points are required 
to draw the Bezier curve")
            return

        points = np.array(self.make_coordinates())

        t = np.linspace(0, 1, 1000)
        curve_x = np.zeros_like(t)
        curve_y = np.zeros_like(t)

        for i in range(len(points)):
            curve_x += points[i][0] * bernstein_poly(i, len(points) - 1, t)
            curve_y += points[i][1] * bernstein_poly(i, len(points) - 1, t)

        fig, ax = plt.subplots()
        ax.plot(curve_x, curve_y, label='Bezier Curve', color='blue')
        ax.scatter(points[:, 0], points[:, 1], color='red')  # Отображение 
точек управления
        for i, (x, y) in enumerate(points):
            ax.text(x, y, f'P{i} ({x},{y})', fontsize=12, ha='right')

        # Adding dashed lines connecting the control points
        for i in range(len(points) - 1):
            ax.plot([points[i][0], points[i + 1][0]], [points[i][1], 
points[i + 1][1]], 'k--')

        ax.legend()
        if self.canvas == None:
            self.canvas = FigureCanvasTkAgg(fig, master=self)
        else:
```

```
            self.canvas.get_tk_widget().destroy()
            self.canvas = FigureCanvasTkAgg(fig, master=self)
        self.canvas.draw()
        self.canvas.get_tk_widget().pack(side="right", fill="both",
expand=True)


if __name__ == "__main__":
    app = BezierCurveApp()
    app.mainloop()
```

МИНОБРНАУКИ РОССИИ

САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ

ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)

ОТЧЁТ ПО ЛАБОРАТОРНОЙ РАБОТЕ №3

по дисциплине Компьютерная графика

Тема: «Формирование различных поверхностей с использованием ее пространственного разворота и ортогонального проецирования на плоскость при ее визуализации (выводе на экран дисплея)»

Студенты гр. 1307		Грунская Н.Д. Тростин М.Ю. Голубев М.А.
Преподаватель		Матвеева И.В.

Санкт-Петербург

2024

Цель работы:
Практическое закрепление теоретических знаний о формировании различных поверхностей с использованием ее пространственного разворота и ортогонального проецирования на плоскость

Постановка задачи:
Сформировать билинейную поверхность на основе произвольного задания ее четырех угловых точек. Обеспечить ее поворот относительно осей X и Y

Краткая теоретическая информация:
Билинейные поверхности - трёхмерные поверхности, задающиеся в пространстве 4-х угловых точек поверхности
Тогда уравнение билинейной поверзности представляется как:
$$\overline{Q}(u,w) = \overline{P_{00}}(1-u)(1-w) + \overline{P_{01}}(1-u)w + \overline{P_{10}}u(1-w) + \overline{P_{11}}uw$$

Для поворота вокруг осей X и Y необходимо применить поворот на один и тот же угол для каждой из четырёх задающих точек

Реализация алгоритма:
Функция поворота точки вокруг оси X
```
def rotate_x(self, num, angle):
    x = self.points[num].coordinates[0]
    y = self.points[num].coordinates[1] * math.cos(angle)
- self.points[num].coordinates[2] * math.sin(angle)
    z = self.points[num].coordinates[1] * math.sin(angle)
+ self.points[num].coordinates[2] * math.cos(angle)
    return [x, y, z]
```

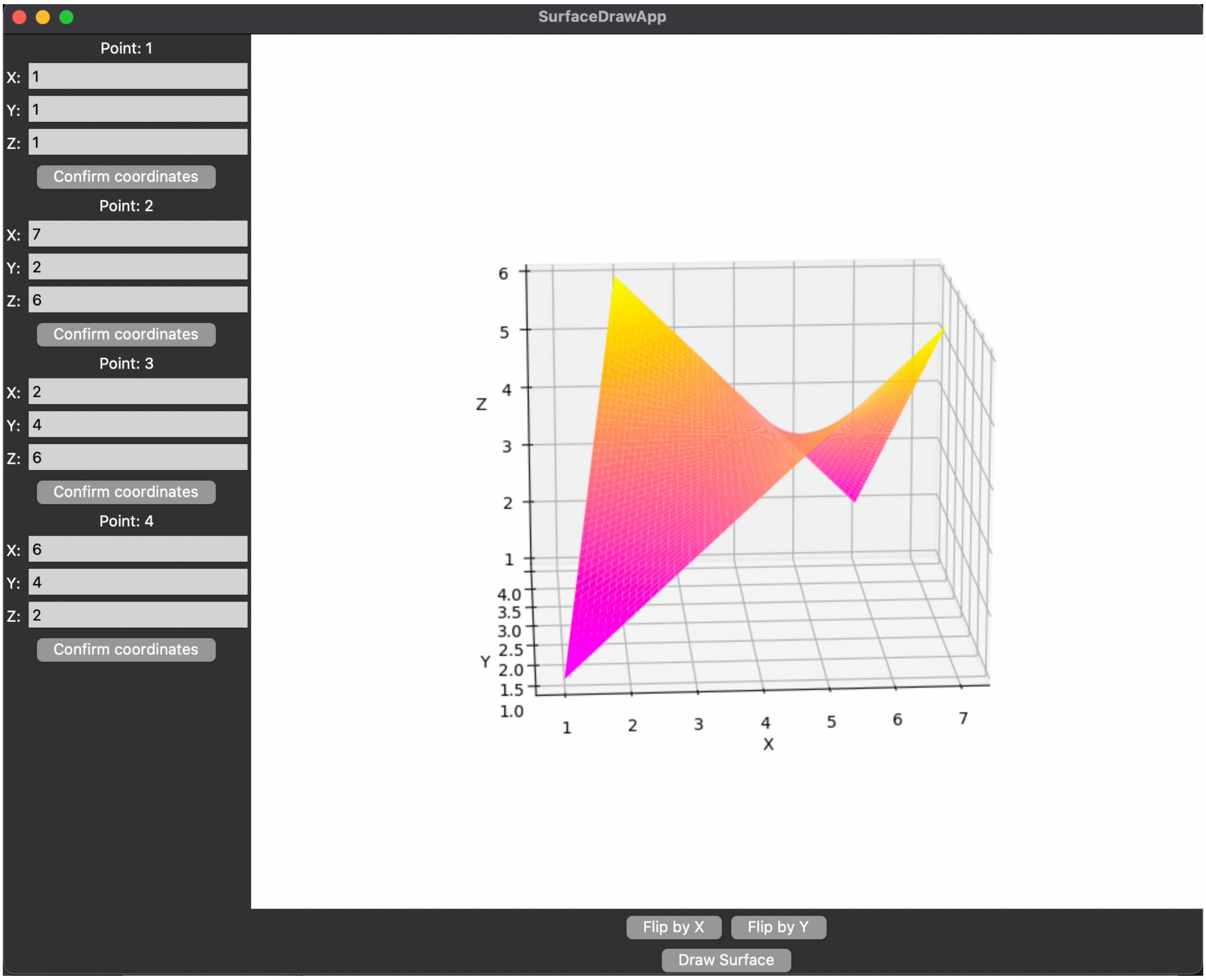
Функция разворота фигуры вокруг оси X
```
def flip_by_x(self):
    angle = math.pi
    for i in range(4):
        self.points[i].coordinates = self.rotate_x(i,
angle)
    self.draw_curve()
```

Функция поворота точки вокруг оси Y
```
def rotate_y(self, num, angle):
    x = self.points[num].coordinates[0] * math.cos(angle)
+ self.points[num].coordinates[2] * math.sin(angle)
    y = self.points[num].coordinates[1]
    z = -self.points[num].coordinates[0] *
math.sin(angle) + self.points[num].coordinates[2] *
math.cos(angle)
    return [x, y, z]
```

Функция разворота фигуры вокруг оси Y

```
def flip_by_y(self):
    angle = math.pi
    for i in range(4):
        self.points[i].coordinates = self.rotate_y(i, 
angle)
    self.draw_curve()
```

Пример работы приложения:



Выводы:

В ходе выполнения работы были практически закреплены теоретические знания о формировании различных поверхностей с использованием ее пространственного разворота и ортогонального проецирования на плоскость

Приложение
Ссылка на видео:

Исходный код програмы:

```
import math
import tkinter as tk
from tkinter import messagebox
import numpy as np
from matplotlib import pyplot as plt
from matplotlib.backends.backend_tkagg import
FigureCanvasTkAgg

import math
import tkinter as tk
from tkinter import messagebox
import numpy as np
from matplotlib import pyplot as plt
from matplotlib.backends.backend_tkagg import
FigureCanvasTkAgg

global frequency
frequency = 100


class Point(tk.Frame):
    def __init__(self, parent, number):
        super().__init__(parent)

        self.coordinates = [0, 0, 0]
        self.number = number
        self.confirm_button = tk.Button
        self.point_frame = tk.Frame(self, width=100,
height=50)
        self.point_frame.pack()
        new_point_label = tk.Label(self.point_frame,
text="Point: " + self.number)
        new_point_label.pack()

        # literally small x with his little labels and
entry's in his little frame
        self.x_frame = tk.Frame(self, width=50,
height=25)
        x_label = tk.Label(self.x_frame, text="X:")
        x_label.pack(side="left")
        self.x_entry = tk.Entry(self.x_frame)
```

```
        self.x_entry.pack(side='right')
        self.x_frame.pack()

        # stupid y (I hate him)
        self.y_frame = tk.Frame(self, width=50,
height=25)
        y_label = tk.Label(self.y_frame, text="Y:")
        y_label.pack(side="left")
        self.y_entry = tk.Entry(self.y_frame)
        self.y_entry.pack(side='right')
        self.y_frame.pack()

        # ZZZZZZZZZ do I even have to sa anything LOL
        self.z_frame = tk.Frame(self, width=50,
height=25)
        z_label = tk.Label(self.z_frame, text="Z:")
        z_label.pack(side="left")
        self.z_entry = tk.Entry(self.z_frame)
        self.z_entry.pack(side='right')
        self.z_frame.pack()

        self.confirm_button = tk.Button(self,
text="Confirm coordinates",
command=self.confirm_coordinates)
        self.confirm_button.pack()

    def confirm_coordinates(self):
        self.coordinates[0] = int(self.x_entry.get())
        self.coordinates[1] = int(self.y_entry.get())
        self.coordinates[2] = int(self.z_entry.get())
        self.x_entry.config(bg="lightgrey", fg="black")
        self.y_entry.config(bg="lightgrey", fg="black")
        self.z_entry.config(bg="lightgrey", fg="black")


class SurfaceApp(tk.Tk):
    def __init__(self):
        super().__init__()

        self.fig = None
        self.ax = None
        self.canvas = None

        self.title("SurfaceDrawApp")
        self.geometry("1020x800")
```

```
        self.points = []

        self.menu_frame = tk.Frame(self, width=200,
height=600)
        self.menu_frame.pack(side="left", fill="y")

        self.draw_surface_button = tk.Button(self,
text="Draw Surface", command=self.draw_curve)
        self.draw_surface_button.pack(side="bottom")

        for i in range(4):
            point = Point(self.menu_frame, str(i + 1))
            self.points.append(point)
            point.pack(side='top', fill="both",
expand=False)

        self.buttons_frame = tk.Frame(self, width=200,
height=100)

        self.flip_by_x_button =
tk.Button(self.buttons_frame, text="Flip by X",
command=self.flip_by_x)
        self.flip_by_x_button.pack(side="left")
        self.flip_by_y_button =
tk.Button(self.buttons_frame, text="Flip by Y",
command=self.flip_by_y)
        self.flip_by_y_button.pack(side="right")

        self.buttons_frame.pack(side="bottom")

    def make_coordinates(self):
        points_correct = []
        for i in range(len(self.points)):
            a = [self.points[i].coordinates[0],
self.points[i].coordinates[1],
self.points[i].coordinates[2]]
            points_correct.append(a)
        print(points_correct)
        return points_correct

    def set_coordinates(self):

        coordinates = self.make_coordinates()

        u = np.linspace(0, 1, frequency)  # параметр u
```

```
        v = np.linspace(0, 1, frequency)  # параметр v
        x_values = []

        x1 = coordinates[0][0]
        x2 = coordinates[1][0]
        x3 = coordinates[2][0]
        x4 = coordinates[3][0]
        for i in range(len(u)):
            for j in range(len(v)):
                x = x1 * (1 - u[i]) * (1 - v[j]) + x2 *
v[j] * (1 - u[i]) + x3 * (1 - v[j]) * u[i] + x4 * u[i] *
v[j]
                x_values.append(x)

        y_values = []

        y1 = coordinates[0][1]
        y2 = coordinates[1][1]
        y3 = coordinates[2][1]
        y4 = coordinates[3][1]
        for i in range(len(u)):
            for j in range(len(v)):
                y = y1 * (1 - u[i]) * (1 - v[j]) + y2 *
v[j] * (1 - u[i]) + y3 * (1 - v[j]) * u[i] + y4 * u[i] *
v[j]
                y_values.append(y)

        z_values = []

        z1 = coordinates[0][2]
        z2 = coordinates[1][2]
        z3 = coordinates[2][2]
        z4 = coordinates[3][2]
        for i in range(len(u)):
            for j in range(len(v)):
                z = z1 * (1 - u[i]) * (1 - v[j]) + z2 *
v[j] * (1 - u[i]) + z3 * (1 - v[j]) * u[i] + z4 * u[i] *
v[j]
                z_values.append(z)

        x_mesh = np.array(x_values).reshape((frequency,
frequency))  # assuming you have 100 points in u and v
        y_mesh = np.array(y_values).reshape((frequency,
frequency))  # assuming you have 100 points in u and v
```

```
        z_mesh = np.array(z_values).reshape((frequency,
frequency))

        return [x_mesh, y_mesh, z_mesh]

    def rotate_x(self, num, angle):
        x = self.points[num].coordinates[0]
        y = self.points[num].coordinates[1] *
math.cos(angle) - self.points[num].coordinates[2] *
math.sin(angle)
        z = self.points[num].coordinates[1] *
math.sin(angle) + self.points[num].coordinates[2] *
math.cos(angle)
        return [x, y, z]

    def rotate_y(self, num, angle):
        x = self.points[num].coordinates[0] *
math.cos(angle) + self.points[num].coordinates[2] *
math.sin(angle)
        y = self.points[num].coordinates[1]
        z = -self.points[num].coordinates[0] *
math.sin(angle) + self.points[num].coordinates[2] *
math.cos(angle)
        return [x, y, z]

    def flip_by_x(self):
        angle = math.pi
        for i in range(4):
            self.points[i].coordinates = self.rotate_x(i,
angle)
        self.draw_curve()

    def flip_by_y(self):
        angle = math.pi
        for i in range(4):
            self.points[i].coordinates = self.rotate_y(i,
angle)
        self.draw_curve()

    def draw_curve(self):
        if len(self.points) < 4:
            messagebox.showerror("Error", "At least 4
points are required to draw the surface")
            return
```

```
        if self.canvas is not None:
            self.fig.clf()
            self.canvas.get_tk_widget().pack_forget()
            self.canvas.get_tk_widget().destroy()

        # assuming you have 100 points in u and v
        self.fig = plt.figure()
        self.ax = self.fig.add_subplot(111,
projection='3d')
        settled_coordinates = self.set_coordinates()
        surf =
self.ax.plot_surface(settled_coordinates[0],
settled_coordinates[1], settled_coordinates[2],
                                    cmap='spring')

        self.ax.set_xlabel("X", loc='right')
        self.ax.set_ylabel("Y")
        self.ax.set_zlabel("Z")

        self.canvas = FigureCanvasTkAgg(self.fig,
master=self)
        self.canvas.draw()
        self.canvas.get_tk_widget().pack(side="right",
fill="both", expand=True)


if __name__ == "__main__":
    app = SurfaceApp()
    app.mainloop()
```

Приложение:

Ссылка на видео:
https://www.youtube.com/watch?v=QfSVIGJbpH8

МИНОБРНАУКИ РОССИИ

САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ

ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)

ОТЧЁТ ПО ЛАБОРАТОРНОЙ РАБОТЕ №4

по дисциплине Компьютерная графика

Тема: «Исследование алгоритмов отсечения отрезков и многоугольников окнами различного вида»

Студенты гр. 1307		Грунская Н.Д. Тростин М.Ю. Голубев М.А.
Преподаватель		Матвеева И.В.

Санкт-Петербург

2024

Цель работы:
Практическое закрепление теоретических знаний об алгоритмах отсечения отрезков и многоугольников окнами различного вида

Постановка задачи:
Обеспечить реализацию алгоритма отсечения массива произвольных отрезков заданным прямоугольным окном с использование алгоритма Коэна-Сазерленда. Вначале следует вывести на экран сгенерированные отрезки полностью, а затем другим цветом или яркостью те, которые полностью или частично попадают в область окна

Краткая теоретическая информация:

Схема Коэна-Сазерленда - один из первых алгоримов для быстрого отсечения линий

Этот алгоритм сравнивает концы отрезка с границами отсекателя (некоторой области на экране) и на основе результатов определяет, полностью ли отрезок видим, полностью ли скрыт или пересекается с отсекателем.

Точки классифицируются относительно границ отсекателя по коду в двоичной форме (например, верх, низ, право, лево).

Всего четыре границы окна формируют девять областей, а на рис. 1 перечислены значения двоичного кода во всех этих областях.

Рис. 1 - Девять областей окна с двоичными кодами

После определения кодов областей для всех конечных точек, определяют линии, которые полностью лежат внутри окна и очевидно лежащие снаружи.

Если конечные точки линий имеют код области 0000, эти отрезки целиком вмещаются в окно, они сохраняются.

Любая линия, конечные точки которой имеют 1 в одинаковых разрядах кода области, лежит полностью за пределами окна, и этот отрезок удаляется.

Если линии с помощью тестов кодов области нельзя отнести к полностью внешним или полностью внутренним, выполняется проверка на пересечения с границами окон.

Реализация алгоритма:
Определение двоичных кодов областей
```
INSIDE = 0   # 0000
LEFT = 1     # 0001
RIGHT = 2    # 0010
BOTTOM = 4   # 0100
TOP = 8      # 1000
```

Функция для определения двоичного кода точки

```
def compute_code(x, y):
    code = INSIDE
    if x < x_min:
        code |= LEFT
    elif x > x_max:
        code |= RIGHT
    if y < y_min:
        code |= BOTTOM
    elif y > y_max:
        code |= TOP
    return code
```

Функция для отображения отрезка с заданными координатами границ

```
def compute_and_draw(x1, y1, x2, y2):
    my_canvas.create_line(x1, y1, x2, y2, fill = 'blue')
    my_canvas.grid(row = 6, column = 0)
    code1 = compute_code(x1, y1)
    code2 = compute_code(x2, y2)
    accept = False
    while True:
        if code1 == 0 and code2 == 0:
            accept = True
            break
        elif (code1 & code2) != 0:
            break
        else:
            x = 1.0
            y = 1.0
            if code1 != 0:
                code_out = code1
            else:
                code_out = code2
            if code_out & TOP:
                x = x1 + ((x2 - x1) / (y2 - y1)) * (y_max
- y1)
                y = y_max
            elif code_out & BOTTOM:
                x = x1 + ((x2 - x1) / (y2 - y1)) * (y_min
- y1)
                y = y_min
            elif code_out & RIGHT:
                y = y1 + ((y2 - y1) / (x2 - x1)) * (x_max
- x1)
                x = x_max
            elif code_out & LEFT:
```

```
                y = y1 + ((y2 - y1) / (x2 - x1)) * (x_min
- x1)
                x = x_min
            if code_out == code1:
                x1 = x
                y1 = y
                code1 = compute_code(x1, y1)
            else:
                x2 = x
                y2 = y
                code2 = compute_code(x2, y2)
    if accept:
        my_canvas.create_line(x1, y1, x2, y2, fill =
'blue',  width = 1)
        my_canvas.grid(row = 6, column = 0)

        second_canvas.create_line(x1, y1, x2, y2, fill =
'green')
        second_canvas.grid(row = 6, column = 1)
    else:
        print("Line rejected")
```

Пример работы приложения:

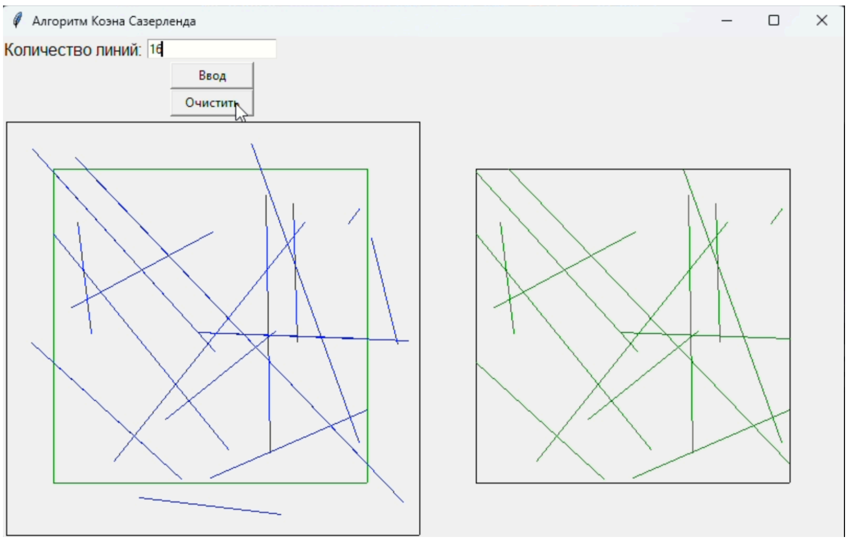


Рис. 2 - Пример работы приложения

Выводы:
В ходе выполнения работы были практически закреплены теоретические знания об алгоритмах отсечения отрезков и многоугольников окнами различного вида

Приложение
Ссылка на видео:

Исходный код программы:

```
import random
from tkinter import *

INSIDE = 0  # 0000
LEFT = 1    # 0001
RIGHT = 2   # 0010
BOTTOM = 4  # 0100
TOP = 8     # 1000

x_max = 350
y_max = 350
x_min = 50
y_min = 50

def compute_code(x, y):
    code = INSIDE
    if x < x_min:
        code |= LEFT
    elif x > x_max:
        code |= RIGHT
    if y < y_min:
        code |= BOTTOM
    elif y > y_max:
        code |= TOP
    return code

def compute_and_draw(x1, y1, x2, y2):
    my_canvas.create_line(x1, y1, x2, y2, fill = 'blue')
    my_canvas.grid(row = 6, column = 0)
    code1 = compute_code(x1, y1)
    code2 = compute_code(x2, y2)
    accept = False
    while True:

        if code1 == 0 and code2 == 0:
            accept = True
            break
        elif (code1 & code2) != 0:
            break
        else:
            x = 1.0
```

```
            y = 1.0
            if code1 != 0:
                code_out = code1
            else:
                code_out = code2

            # y = y1 + slope * (x - x1),
            # x = x1 + (1 / slope) * (y - y1)
            if code_out & TOP:
                x = x1 + ((x2 - x1) / (y2 - y1)) * (y_max
- y1)
                y = y_max
            elif code_out & BOTTOM:
                x = x1 + ((x2 - x1) / (y2 - y1)) * (y_min
- y1)
                y = y_min
            elif code_out & RIGHT:
                y = y1 + ((y2 - y1) / (x2 - x1)) * (x_max
- x1)
                x = x_max

            elif code_out & LEFT:
                y = y1 + ((y2 - y1) / (x2 - x1)) * (x_min
- x1)
                x = x_min

            if code_out == code1:
                x1 = x
                y1 = y
                code1 = compute_code(x1, y1)

            else:
                x2 = x
                y2 = y
                code2 = compute_code(x2, y2)

    if accept:
        my_canvas.create_line(x1, y1, x2, y2, fill =
'blue',  width = 1)
        my_canvas.grid(row = 6, column = 0)

        second_canvas.create_line(x1, y1, x2, y2, fill =
'green')
        second_canvas.grid(row = 6, column = 1)
```

```
    else:
        print("Line rejected")

def generate_lines():
    n = int(entry_n_lines.get())
    for i in range(n):
        x1 = float(random.randint(5, 400))
        y1 = float(random.randint(5, 400))
        x2 = float(random.randint(5, 400))
        y2 = float(random.randint(5, 400))
        compute_and_draw(x1, y1, x2, y2)

def clear_lines():
    my_canvas.delete("all")
    second_canvas.delete("all")

    my_canvas.create_line(5,400,400,400, fill = 'black')
    my_canvas.create_line(400,5,400,400, fill = 'black')
    my_canvas.create_line(5,5,5,400, fill = 'black')
    my_canvas.create_line(5,5,400,5, fill = 'black')

    my_canvas.create_line(50,350,350,350, fill = 'green')
    my_canvas.create_line(350,50,350,350, fill = 'green')
    my_canvas.create_line(50,50,50,350, fill = 'green')
    my_canvas.create_line(50,50,350,50, fill = 'green')

    second_canvas.create_line(50,350,350,350, fill =
'black')
    second_canvas.create_line(350,50,350,350, fill =
'black')
    second_canvas.create_line(50,50,50,350, fill =
'black')
    second_canvas.create_line(50,50,350,50, fill =
'black')

#   ui
my_window = Tk()
my_window.title("Алгоритм Коэна Сазерленда")

#   input
Label(my_window, text = "Количество линий: ", fg =
"black", font = "none 12").grid(row = 1, column = 0,
sticky = W)
entry_n_lines = Entry(my_window, width = 20, bg= "white")
```

```
entry_n_lines.grid (row = 1, column = 0)
#   button
Button(my_window, text = "Ввод", width = 10, command =
generate_lines).grid(row = 4, column = 0)

Button(my_window, text = "Очистить", width = 10, command =
clear_lines).grid(row = 5, column = 0)
#   canvas
my_canvas = Canvas(my_window, width = 400, height = 400)
my_canvas.grid(row = 6, column = 0)
my_canvas.create_line(5,400,400,400, fill = 'black')
my_canvas.create_line(400,5,400,400, fill = 'black')
my_canvas.create_line(5,5,5,400, fill = 'black')
my_canvas.create_line(5,5,400,5, fill = 'black')

my_canvas.create_line(50,350,350,350, fill = 'green')
my_canvas.create_line(350,50,350,350, fill = 'green')
my_canvas.create_line(50,50,50,350, fill = 'green')
my_canvas.create_line(50,50,350,50, fill = 'green')

second_canvas = Canvas(my_window, width = 400, height =
400)
second_canvas.grid(row = 6, column = 1)
second_canvas.create_line(50,350,350,350, fill = 'black')
second_canvas.create_line(350,50,350,350, fill = 'black')
second_canvas.create_line(50,50,50,350, fill = 'black')
second_canvas.create_line(50,50,350,50, fill = 'black')

my_window.mainloop()
```

Приложение:

Ссылка на видео:
https://www.youtube.com/watch?v=kJI2IWfSwnk

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

ОТЧЁТ ПО ЛАБОРАТОРНОЙ РАБОТЕ №2

по дисциплине Компьютерная графика

Тема: «Формирования различных кривых с использованием ортогонального проектирования на плоскость визуализации (экране дисплея)

Студенты гр. 1307	_______________	Грунская Н.Д. Тростин М.Ю. Голубев М.А.
Преподаватель	_______________	Матвеева И.В.

Санкт-Петербург

2024

Цель работы:
Практическое закрепление теоретических знаний о формировании кривых с использованием ортогонального проектирования на плоскость визуализации.

Постановка задачи:

Сформировать на плоскости кривую Безье на основе задающей ломаной, определяемой 3 и большим количеством точек. Обеспечить редактирование координат точек задающей ломаной с перерисовкой сплайна Безье.

Краткая теоретическая информация:

Точки задания этих кривых Безье только определяют ход кривой, сама строящаяся кривая в общем случае не проходит через внутренние точки задающего многоугольника

Особенности:

1. Подходит по касательной к внешним ребрам (сторонам) задающего многоугольника, а остальные точки определяют ход кривой. Они позволяют качественно оценить ход кривой в зависимости от вида задающего многоугольника.
2. Кривая задается параметрически в функции от независимого параметра.
3. Это кривая n-ой степени, т.е. сколько ребер у задающего многоугольника – такой степени и получается кривая. Влиять на степень кривой можно только изменением количества задающих ее точек.

Математически такая кривая описывается параметрическим уравнением:

$P(t) = \sum_{i=0}^{n} P_i \times N_{i,n}(t)$,где P(t) – полиномиальная функция,
Pi – вес (координаты) i-ой точки задания,
Ni, n – весовой коэффициент i-той вершины,
i – номер вершины (точки),
n – количество сторон задающего многоугольника
t – задающий параметр, причем 0 <= t <= 1

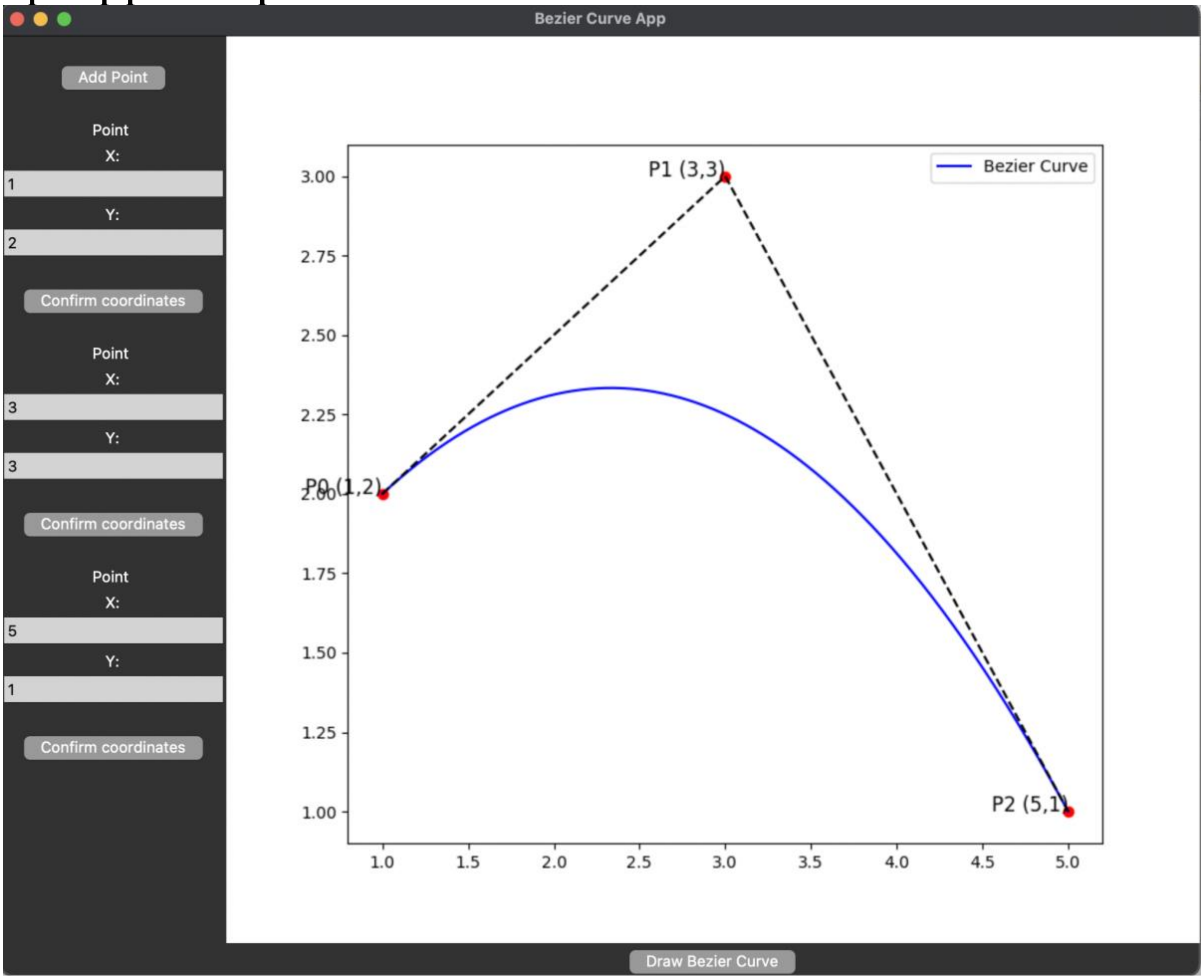
$$N_{i,n}(t) = \frac{n!}{i!(n-i)!} \times t^i \times (1-t)^{n-}$$

Если множество точек задания (n), то строим либо кривую n-ой степени, либо можем построить кривую невысокой степени (обычно кубическую, так как она позволяет обеспечить перегиб и может быть быстро построена). В последнем случаедля такой кривой нужно только четыре последовательные точки задания.

Если n > 4, то мы находимся в противоречии с количеством точек, так как формируя сегменты на основе точек задающего многоугольника в точках стыковки таких сплайнов кривая будет иметь излом (разрыв производных). Чтобы избежать такой ситуации учитывают то свойство кривых Безье, что к внешним ребрам задающего много угольника они подходят по касательной. Поэтому в третье ребро описания при формировании кубической кривой Безье добавляют дополнительную точку P` и ее считают за последнею точку текущего характеристического многоугольника и первой точкой следующего характеристического многоугольника. Обычно ее берут по середине ребра. Тогда кривая будет плавно переходить от предыдущей кубической кривой (кривой третьей степени) к последующей. А вся такая кривая представляет собой составную плавную кривую, состоящую из ряда сегментов, называется составной кривой Безье.

Для расчета последнего сегмента такой составной кривой Безье можно либо понизить степень строящегося участка кривой до второй, так как может остаться только три незадействованной точки, либо при расчете использовать последнюю точку дважды.

Пример работы приложения:



Выводы:

Были практически закреплены теоретические знания о формировании кривых с использованием ортогонального проектирования на плоскость визуализации.

Приложение

Ссылка на видео: : https://youtu.be/irFx5sxIQfg

Исходный код:

beizer_functions.py

```
import scipy.special


def bernstein_poly(i, n, t):
    return scipy.special.comb(n, i) * t ** i * (1 - t) ** (n - i)
```

lab2.py

```
import tkinter as tk
from tkinter import messagebox
import numpy as np
from matplotlib import pyplot as plt
from matplotlib.backends.backend_tkagg import FigureCanvasTkAgg
from bezier_functions import bernstein_poly


class Point(tk.Frame):
    def __init__(self, parent):
        super().__init__(parent)
        self.coordinates = [0, 0]
        self.confirm_button = tk.Button
        self.number = ''
        self.point_frame = tk.Frame(self, width=100, height=50)
        self.point_frame.pack()
        new_point_label = tk.Label(self.point_frame, text="Point " +
self.number)
        new_point_label.pack()

        x_label = tk.Label(self.point_frame, text="X:")
        x_label.pack()
        self.x_entry = tk.Entry(self.point_frame)
        self.x_entry.pack()

        y_label = tk.Label(self.point_frame, text="Y:")
        y_label.pack()
        self.y_entry = tk.Entry(self.point_frame)
        self.y_entry.pack()

        self.confirm_button = tk.Button(self.point_frame, text="Confirm
coordinates", command=self.confirm_coordinates)
        self.confirm_button.pack(pady=20)

    def confirm_coordinates(self):
        self.coordinates[0] = int(self.x_entry.get())
        self.coordinates[1] = int(self.y_entry.get())
        self.x_entry.config(bg="lightgrey", fg="black")
        self.y_entry.config(bg="lightgrey", fg="black")


class BezierCurveApp(tk.Tk):
    def __init__(self):
        super().__init__()

        self.canvas = None
        self.title("Bezier Curve App")
        self.geometry("1020x800")
```

```
        self.figs = []
        self.points = []
        self.point_coordinates = []

        self.menu_frame = tk.Frame(self, width=200, height=600)
        self.menu_frame.pack(side="left", fill="y")

        self.add_point_button = tk.Button(self.menu_frame, text="Add
Point", command=self.add_point)
        self.add_point_button.pack(pady=20)

        self.draw_curve_button = tk.Button(self, text="Draw Bezier Curve",
command=self.draw_curve)
        self.draw_curve_button.pack(side="bottom")

    def make_coordinates(self):
        points_correct = []
        for i in range(len(self.points)):
            a = [self.points[i].coordinates[0],
self.points[i].coordinates[1]]
            points_correct.append(a)
        print(points_correct)
        return points_correct

    def add_point(self):
        new_point = Point(self.menu_frame)
        self.points.append(new_point)
        new_point.pack(side='top', fill="both", expand=False)

    def draw_curve(self):
        if len(self.points) < 2:
            messagebox.showerror("Error", "At least 2 points are required
to draw the Bezier curve")
            return

        points = np.array(self.make_coordinates())

        t = np.linspace(0, 1, 1000)
        curve_x = np.zeros_like(t)
        curve_y = np.zeros_like(t)

        for i in range(len(points)):
            curve_x += points[i][0] * bernstein_poly(i, len(points) - 1, t)
            curve_y += points[i][1] * bernstein_poly(i, len(points) - 1, t)

        fig, ax = plt.subplots()
        ax.plot(curve_x, curve_y, label='Bezier Curve', color='blue')
        ax.scatter(points[:, 0], points[:, 1], color='red')  # Отображение
точек управления
        for i, (x, y) in enumerate(points):
            ax.text(x, y, f'P{i} ({x},{y})', fontsize=12, ha='right')

        # Adding dashed lines connecting the control points
        for i in range(len(points) - 1):
            ax.plot([points[i][0], points[i + 1][0]], [points[i][1],
points[i + 1][1]], 'k--')

        ax.legend()
        if self.canvas == None:
            self.canvas = FigureCanvasTkAgg(fig, master=self)
        else:
```

```
            self.canvas.get_tk_widget().destroy()
            self.canvas = FigureCanvasTkAgg(fig, master=self)
        self.canvas.draw()
        self.canvas.get_tk_widget().pack(side="right", fill="both",
expand=True)


if __name__ == "__main__":
    app = BezierCurveApp()
    app.mainloop()
```