



中国科学技术大学

计算机图形学实验报告

学院：微电子学院

专业：电路与系统

姓名：刘宇阳

学号：SC24219012

完成时间：2024 年 12 月 25 日

中国科学技术大学

二〇二四 年 十二 月

目录

实验一 1

 一、 实验原理 1

 1. Bresenham 算法 1

 2. 中点椭圆绘制算法 1

 3. 多边形绘制 1

 4. 四联通漫水填充算法 1

 二、 算法实现思路 2

 1. 界面初始化 2

 2. 事件处理 2

 3. 绘图功能 2

 4. 填充功能 2

 三、 核心代码 2

 1. 绘制直线 2

 2. 绘制椭圆 3

 3. 绘制多边形 4

 4. 区域填充 4

 四、 实验结果 5

 1. 直线 5

 2. 椭圆 5

 3. 多边形 6

 4. 区域填充 6

实验二 7

 一、 实验原理 7

 1. B 样条曲线（BSpline） 7

 2. 贝塞尔曲线（Bezier Curve） 7

 3. de Casteljau 算法 7

 二、 算法实现思路 7

 1. B 样条曲线（BSpline） 7

 2. 贝塞尔曲线（Bezier Curve） 7

 3. de Casteljau 算法 7

 三、 核心代码 8

 1. B 样条曲线（BSpline） 8

 2. 贝塞尔曲线（Bezier Curve） 8

 3. de Casteljau 算法 9

 四、 实验结果 10

1. B 样条曲线 (BSpline)	10
2. 贝塞尔曲线 (Bezier Curve)	11
3. de Casteljau 算法	11
实验三	12
一、 实验原理	12
二、 算法实现思路	12
1. 科赫曲线 (Koch 曲线)	12
2. 蕨类植物	12
三、 核心代码	12
1. 科赫曲线 (Koch 曲线)	12
2. 蕨类植物	13
四、 实验结果	14
1. 科赫曲线 (Koch 曲线)	14
2. 蕨类植物	15
实验四	16
一、 实验原理	16
二、 算法实现思路	16
1. 初始化场景	16
2. 射线生成	16
3. 光线追踪	16
4. 光照计算	16
5. 显示渲染结果	17
三、 核心代码	17
1. trace_ray 函数	17
2. compute_lighting 函数	17
3. display_buffer 函数	18
四、 实验结果	19
附录	20
1. 实验一完整代码	20
2. 实验二完整代码	26
3. 实验三完整代码	30
4. 实验四完整代码	34

实验一

一、实验原理

1. Bresenham 算法

Bresenham 算法通过以下步骤来绘制直线：

- 1) 初始化起点和终点坐标。
- 2) 计算水平和垂直方向的距离差 (dx 和 dy)。
- 3) 根据 dx 和 dy 的大小，决定像素点的移动方向。
- 4) 在每一步中，根据误差判断是否需要跳过一个像素点。

2. 中点椭圆绘制算法

中点椭圆绘制算法通过以下步骤来绘制椭圆：

- 1) 初始化起点和终点坐标以及长轴和短轴的半径。
- 2) 计算像素点的初始坐标和误差值。
- 3) 在每次迭代中，根据误差值决定下一个像素点的位置，并更新误差值。

3. 多边形绘制

多边形绘制通过以下步骤来实现：

- 1) 用户点击画布上的多个点来定义多边形的顶点。
- 2) 每次点击后，连接前一个点和当前点形成一条线段。
- 3) 最后，连接最后一个点和第一个点以闭合多边形。

4. 四联通漫水填充算法

四联通漫水填充算法通过以下步骤来实现：

- 1) 在多边形区域内任选一个点作为种子点。
- 2) 将种子点入栈。
- 3) 如果栈非空，栈顶像素出栈，并将出栈像素设置为填充色，按右、上、左、下的顺序检查四邻域的像素，如果像素不在边界且未填色，则入栈。
- 4) 栈空结束。

二、算法实现思路

1. 界面初始化

- 使用 `tkinter` 创建一个主窗口。
- 在窗口中创建一个画布（`Canvas`），用于绘制图形。

2. 事件处理

- 绑定鼠标点击和右键点击事件，以便在用户交互时触发相应的操作。
- 实现鼠标移动事件，以便在绘制多边形时实时显示线条。

3. 绘图功能

- 直线：使用 `Bresenham` 算法实现直线绘制。
- 椭圆：使用中点椭圆绘制算法（`Midpoint Ellipse Algorithm`）实现椭圆绘制。
- 多边形：用户通过点击画布绘制多边形的顶点，最后通过闭合线条形成多边形。

4. 填充功能

- 四联通漫水填充：使用队列实现迭代式的四联通漫水填充算法（`Flood Fill Algorithm`），填充多边形内部。

三、核心代码

1. 绘制直线

```
def bresenham_line(self, x1, y1, x2, y2, color=(0, 0, 0)):
    dx = abs(x2 - x1)
    dy = abs(y2 - y1)
    sx = 1 if x1 < x2 else -1
    sy = 1 if y1 < y2 else -1
    err = dx - dy

    x, y = x1, y1
    while True:
        self.put_pixel(x, y, color)
        if x == x2 and y == y2:
```

```

        break
    e2 = 2 * err
    if e2 > -dy:
        err -= dy
        x += sx
    if e2 < dx:
        err += dx
        y += sy

```

2. 绘制椭圆

```

def midpoint_ellipse(self, center_x, center_y, a, b, color=(0, 0, 0)):
    x = 0
    y = b

    # Region 1
    d1 = (b * b) - (a * a * b) + (0.25 * a * a)
    dx = 2 * b * b * x
    dy = 2 * a * a * y

    while dx < dy:
        self.put_pixel(center_x + x, center_y + y, color)
        self.put_pixel(center_x - x, center_y + y, color)
        self.put_pixel(center_x + x, center_y - y, color)
        self.put_pixel(center_x - x, center_y - y, color)

        if d1 < 0:
            x += 1
            dx += 2 * b * b
            d1 += dx + b * b
        else:
            x += 1
            y -= 1
            dx += 2 * b * b
            dy -= 2 * a * a
            d1 += dx - dy + b * b

    # Region 2
    d2 = ((b * b) * ((x + 0.5) * (x + 0.5))) + \
        ((a * a) * ((y - 1) * (y - 1))) - \
        (a * a * b * b)

    while y >= 0:

```

```

        self.put_pixel(center_x + x, center_y + y, color)
        self.put_pixel(center_x - x, center_y + y, color)
        self.put_pixel(center_x + x, center_y - y, color)
        self.put_pixel(center_x - x, center_y - y, color)

    if d2 > 0:
        y -= 1
        dy -= 2 * a * a
        d2 += a * a - dy
    else:
        y -= 1
        x += 1
        dx += 2 * b * b
        dy -= 2 * a * a
        d2 += dx - dy + a * a

    self.update_canvas()

```

3. 绘制多边形

```

def on_click(self, event):
    if self.mode == "polygon":
        self.polygon_points.append((event.x, event.y))
        if len(self.polygon_points) > 1:
            self.bresenham_line(int(self.polygon_points[-2][0]),
int(self.polygon_points[-2][1]),
                                int(self.polygon_points[-1][0]),
int(self.polygon_points[-1][1]))

```

4. 区域填充

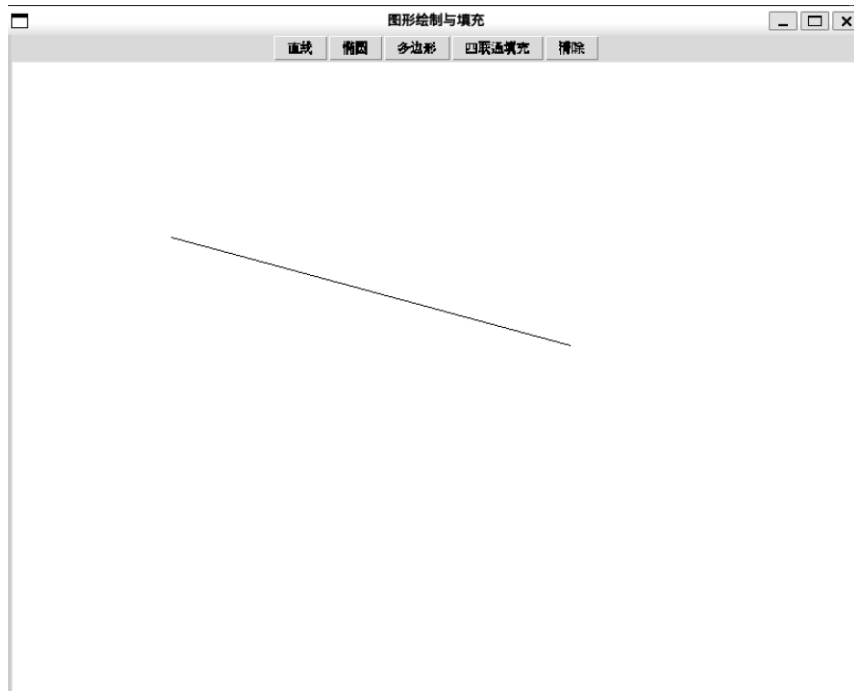
```

def perform_flood_fill(self, event, mode="four"):
    x, y = event.x, event.y
    boundary_color = (0, 0, 0) # 假设边界颜色为黑色
    fill_color = (255, 0, 0) # 填充颜色为红色
    if mode == "four":
        self.flood_fill_four_connected(x, y, fill_color=fill_color,
boundary_color=boundary_color)
        messagebox.showinfo("完成", "四联通填充完成")
    # 重新绑定鼠标事件
    self.canvas.bind("<Button-1>", self.on_click)

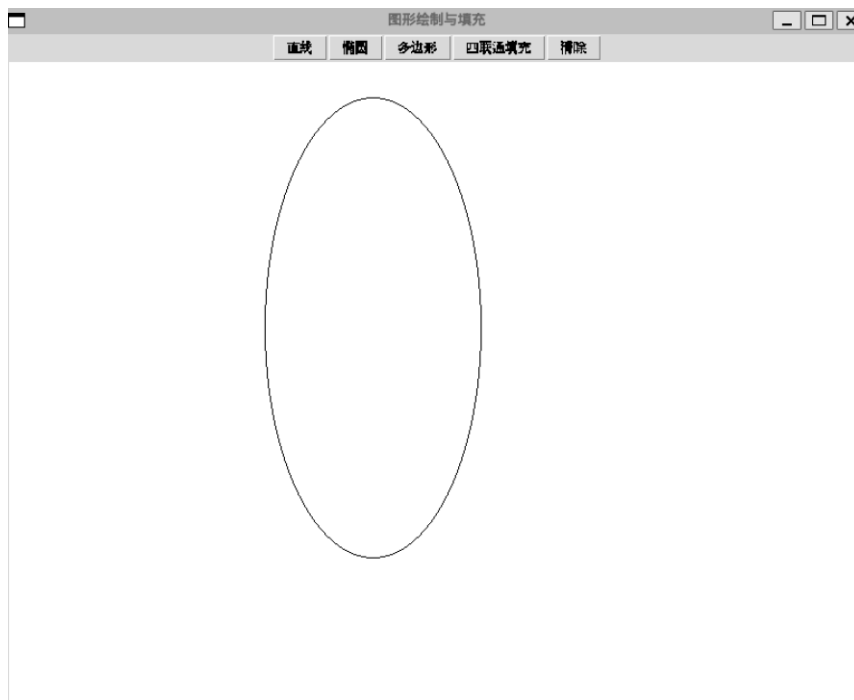
```

四、实验结果

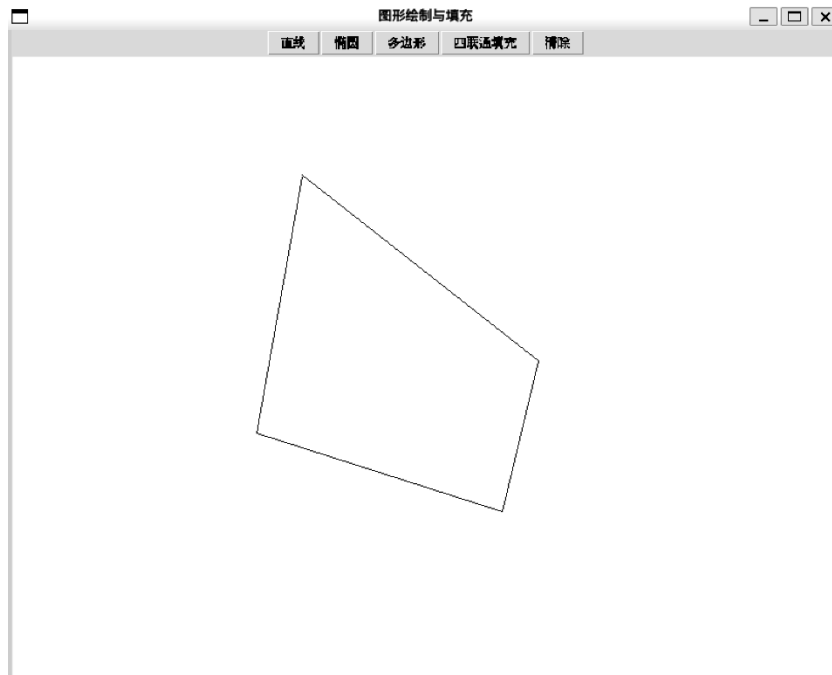
1. 直线



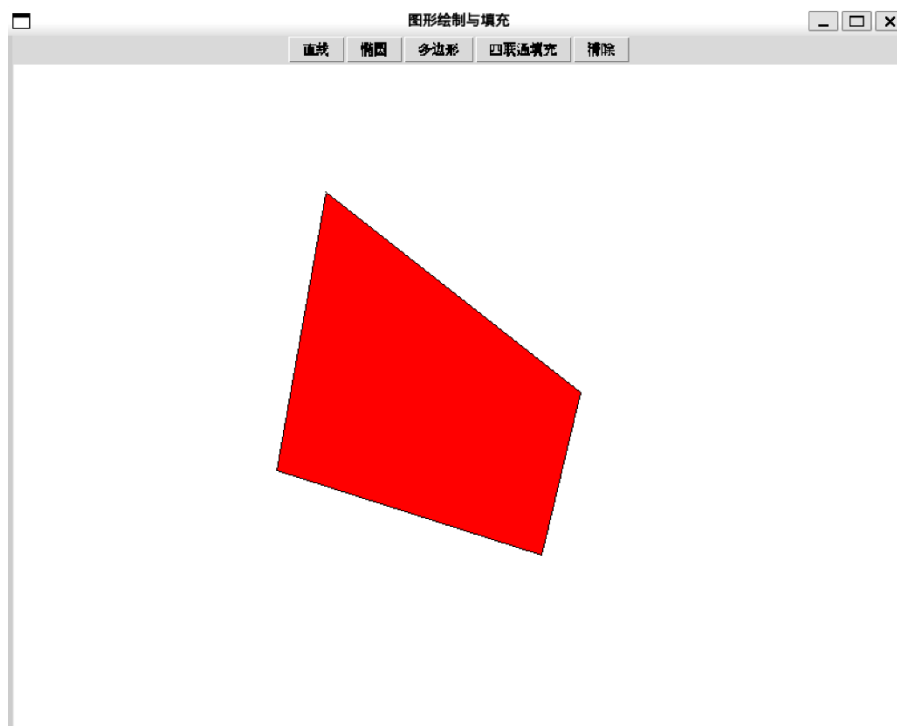
2. 椭圆



3. 多边形



4. 区域填充



实验二

一、实验原理

1. B 样条曲线 (BSpline)

B 样条曲线是一种参数化的曲线，由一组控制点和节点向量定义。

节点向量分为内部节点和边界节点。内部节点均匀分布，边界节点与第一个和最后一个控制点重合。

B 样条曲线的形状可以通过调整控制点的位置来改变。

2. 贝塞尔曲线 (Bezier Curve)

贝塞尔曲线是一种参数化的曲线，由一组控制点定义。

最常用的贝塞尔曲线是二次贝塞尔曲线和三次贝塞尔曲线。

贝塞尔曲线的形状可以通过调整控制点的位置来改变。

3. de Casteljau 算法

de Casteljau 算法是一种递归方法，用于计算贝塞尔曲线上的点。

该算法通过迭代地对控制点进行线性插值，最终得到贝塞尔曲线上的点。

二、算法实现思路

1. B 样条曲线 (BSpline)

- 定义节点向量和控制点。
- 使用递归公式计算 B 样条基函数。
- 绘制曲线时，根据参数 t (范围在 0 到 1 之间) 计算每个点的坐标。

2. 贝塞尔曲线 (Bezier Curve)

- 定义控制点。
- 使用 Bernstein 多项式定义贝塞尔曲线。
- 绘制曲线时，根据参数 t (范围在 0 到 1 之间) 计算每个点的坐标。

3. de Casteljau 算法

- 定义控制点。

- b) 递归地对控制点进行线性插值。
- c) 绘制每一层的插值点和最终的贝塞尔曲线。

三、核心代码

1. B 样条曲线 (BSpline)

```
def draw_bspline(self):
    points = np.array(self.control_points)
    n = len(points) - 1
    # 生成均匀二次 B 样条的节点
    t = np.linspace(0, 1, 100)
    # 绘制曲线
    curve_points = []
    for i in range(n-1):
        if i + 2 <= n:
            # 计算二次 B 样条基函数
            for tj in t:
                x = (1-tj)**2/2 * points[i][0] + \
                    (1/2 + tj - tj**2) * points[i+1][0] + \
                    tj**2/2 * points[i+2][0]

                y = (1-tj)**2/2 * points[i][1] + \
                    (1/2 + tj - tj**2) * points[i+1][1] + \
                    tj**2/2 * points[i+2][1]
                curve_points.append((x, y))
    # 绘制曲线段
    for i in range(len(curve_points)-1):
        self.canvas.create_line(curve_points[i][0], curve_points[i][1],
                                curve_points[i+1][0], curve_points[i+1][1],
                                fill="blue", width=2)
```

2. 贝塞尔曲线 (Bezier Curve)

```
def bezier_curve(self):
    if len(self.control_points) < 2:
        return

    def bernstein(i, n, t):
        return comb(n, i) * (t ** i) * ((1 - t) ** (n - i))

    points = []
    n = len(self.control_points) - 1
```

```

for t in np.arange(0, 1.01, 0.01):
    x = y = 0
    for i in range(n + 1):
        basis = bernstein(i, n, t)
        x += basis * self.control_points[i][0]
        y += basis * self.control_points[i][1]
    points.append((x, y))

for i in range(len(points)-1):
    self.canvas.create_line(points[i][0], points[i][1],
                            points[i+1][0], points[i+1][1],
                            fill='red', width=2)

```

3. de Casteljau 算法

```

def casteljau_curve(self):
    if len(self.control_points) < 2:
        return

    def de_casteljau(points, t):
        """
        递归实现 de Casteljau 算法，并返回每一层的插值点。
        """
        layers = [points]
        while len(points) > 1:
            new_points = []
            for i in range(len(points)-1):
                x = (1 - t) * points[i][0] + t * points[i+1][0]
                y = (1 - t) * points[i][1] + t * points[i+1][1]
                new_points.append((x, y))
            layers.append(new_points)
            points = new_points
        return layers

    points_on_curve = []
    for t in np.arange(0, 1.01, 0.01):
        layers = de_casteljau(self.control_points, t)
        final_point = layers[-1][0]
        points_on_curve.append(final_point)

    # 可视化每一层的插值点和线条
    for layer in layers[:-1]: # 不绘制最后一层的单个点

```

```

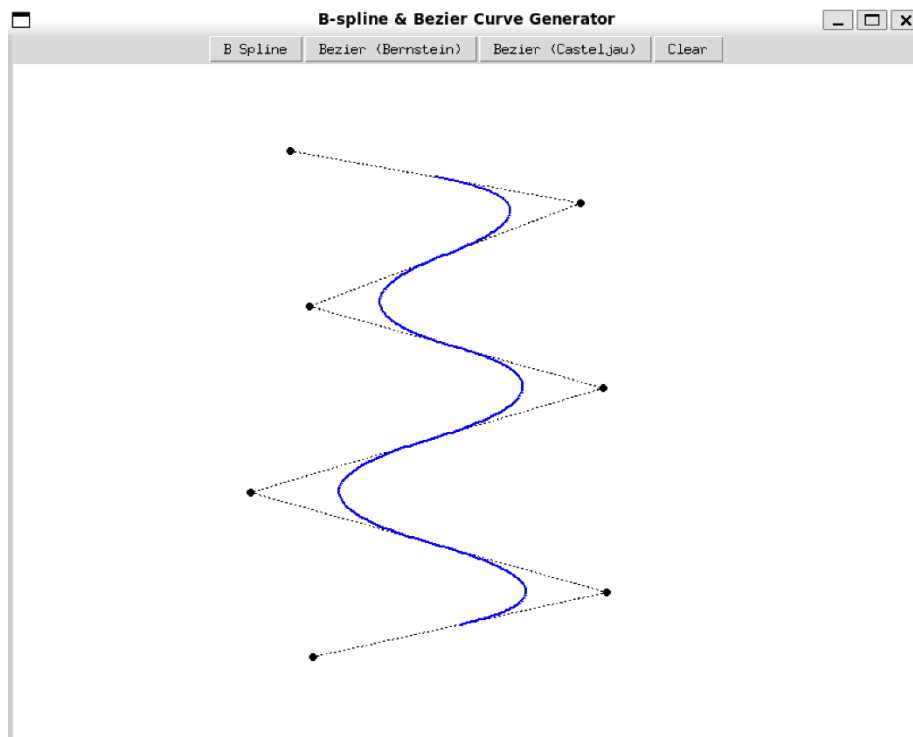
        for i in range(len(layer)-1):
            x1, y1 = layer[i]
            x2, y2 = layer[i+1]
            self.canvas.create_line(x1, y1, x2, y2, fill='lightgray', dash=(1,1))
        for point in layer:
            self.canvas.create_oval(point[0]-2, point[1]-2, point[0]+2,
point[1]+2, fill='orange', outline="")

    # 绘制贝塞尔曲线
    for i in range(len(points_on_curve)-1):
        self.canvas.create_line(points_on_curve[i][0], points_on_curve[i][1],
                                points_on_curve[i+1][0],
                                points_on_curve[i+1][1],
                                fill='green', width=2)

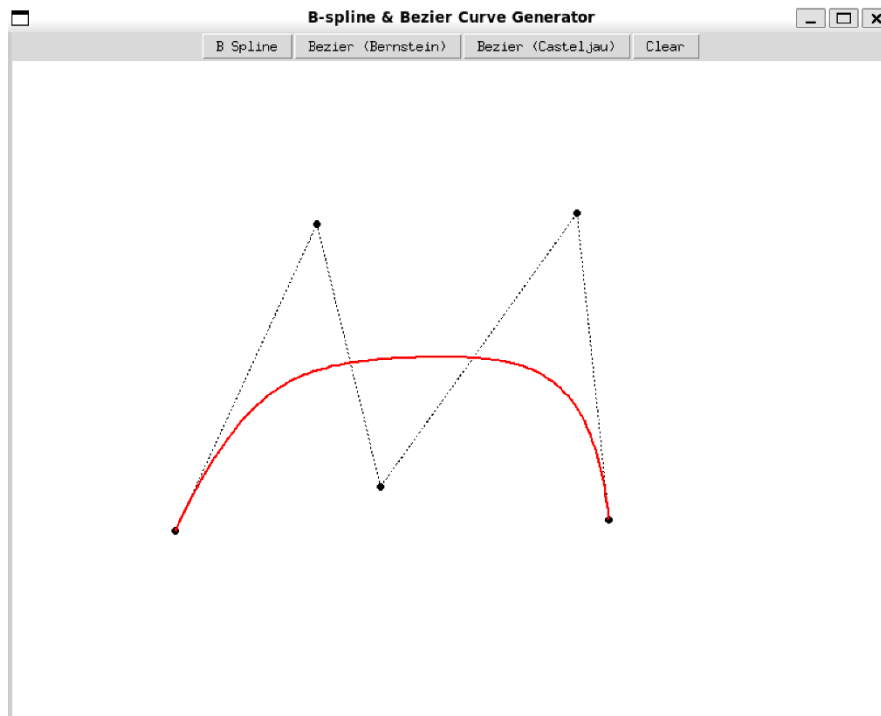
```

四、实验结果

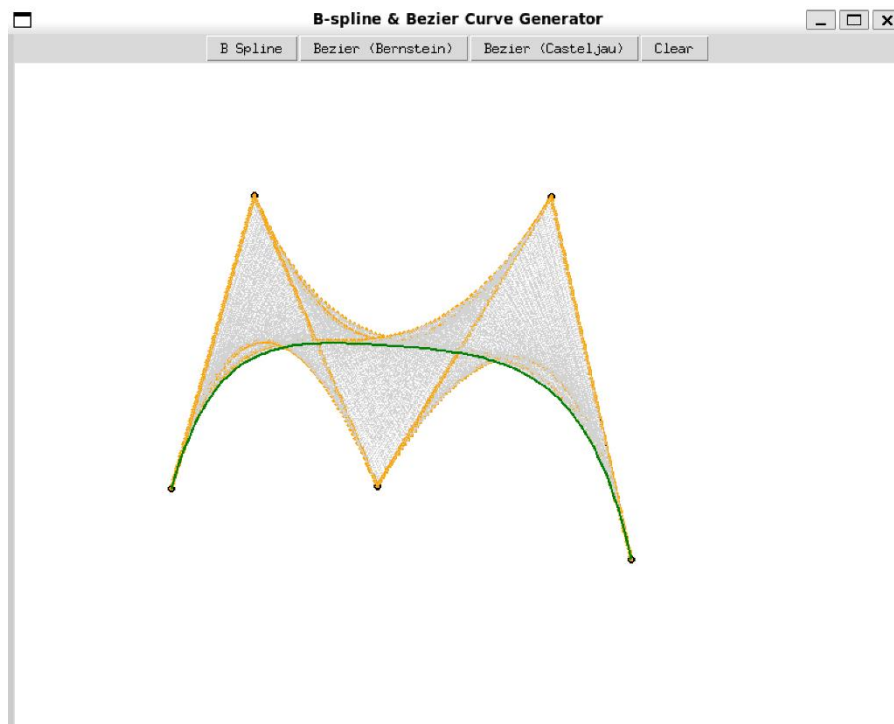
1. B 样条曲线（BSpline）



2. 贝塞尔曲线 (Bezier Curve)



3. de Casteljau 算法



实验三

一、实验原理

Fractal Generator 是一个使用 Python 和 Tkinter 库创建的图形用户界面 (GUI) 应用程序，用于生成和显示两种基本的分形图案：科赫曲线 (Koch Curve) 和蕨类植物 (Fern)。科赫曲线是一种递归生成的曲线，具有自相似的性质。蕨类植物则通过迭代变换矩阵来模拟其生长过程。

二、算法实现思路

1. 科赫曲线 (Koch 曲线)

- 科赫曲线是通过递归的方法生成的。基本思想是从一条线段开始，将其分成三等分，在中间一段上添加一个向外凸起的小三角形。
- 通过递归地对每一小段重复这个过程，可以生成越来越复杂的科赫曲线。

2. 蕨类植物

- 蕨类植物通过迭代变换矩阵来模拟其生长过程。基本思想是使用一组概率分布的变换矩阵，每次根据概率选择一个变换矩阵，将当前点进行变换。
- 通过多次迭代，可以生成一棵看似真实的蕨类植物

三、核心代码

1. 科赫曲线 (Koch 曲线)

```
def koch_curve_points(self, start, end, depth):
    if depth == 0:
        return [start, end]

    # Calculate required points
    dx = end[0] - start[0]
    dy = end[1] - start[1]

    p1 = start
    p2 = (start[0] + dx/3, start[1] + dy/3)

    # Calculate the position of the peak point
    angle = math.pi/3 # 60 degrees
    p3x = p2[0] + (dx/3)*math.cos(angle) - (dy/3)*math.sin(angle)
```

```

p3y = p2[1] + (dx/3)*math.sin(angle) + (dy/3)*math.cos(angle)
p3 = (p3x, p3y)

p4 = (start[0] + 2*dx/3, start[1] + 2*dy/3)
p5 = end

# Recursive calls
curve1 = self.koch_curve_points(p1, p2, depth-1)
curve2 = self.koch_curve_points(p2, p3, depth-1)
curve3 = self.koch_curve_points(p3, p4, depth-1)
curve4 = self.koch_curve_points(p4, p5, depth-1)

# Combine all points (removing duplicates at joints)
return curve1[:-1] + curve2[:-1] + curve3[:-1] + curve4

```

2. 蕨类植物

```

def draw_fern(self):
    self.clear_canvas()
    iterations = self.fern_iterations.get()

    # Transformation matrices and probabilities
    p = np.array([0.85, 0.07, 0.07, 0.01]) # Probabilities

    # Starting point
    x, y = 0, 0

    # Scale and translate coordinates to fit canvas
    def transform_coords(x, y):
        return (400 + x*70, 550 - y*70)

    for _ in range(iterations):
        r = np.random.rand()
        if r < p[0]:
            # Successive stems
            x_new = 0.85*x + 0.04*y
            y_new = -0.04*x + 0.85*y + 1.6
        elif r < p[0] + p[1]:
            # Left leaflet
            x_new = 0.2*x - 0.26*y
            y_new = 0.23*x + 0.22*y + 1.6
        elif r < p[0] + p[1] + p[2]:
            # Right leaflet

```



```

        x_new = -0.15*x + 0.28*y
        y_new = 0.26*x + 0.24*y + 0.44
    else:
        # Successive stems
        x_new = 0
        y_new = 0.16*y

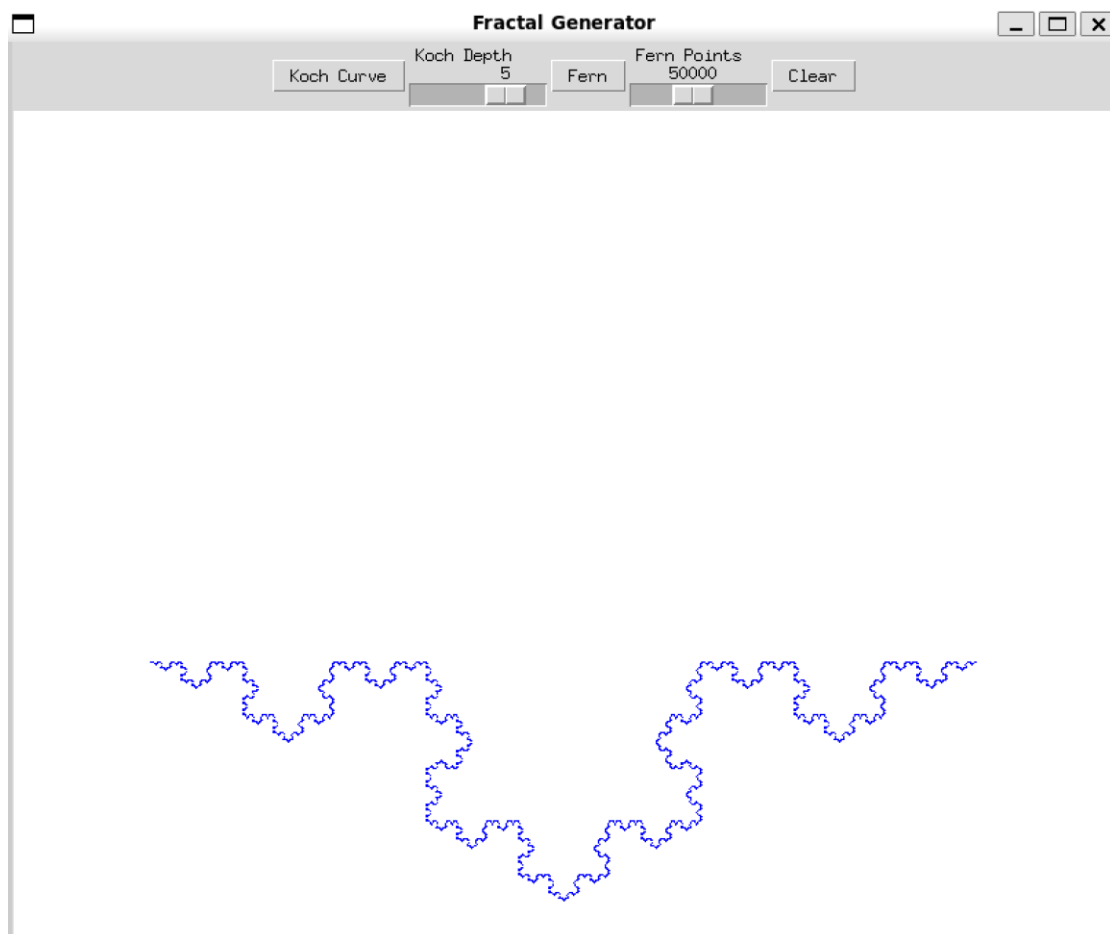
    x, y = x_new, y_new

    # Transform and plot point
    plot_x, plot_y = transform_coords(x, y)
    self.canvas.create_rectangle(plot_x, plot_y, plot_x+1, plot_y+1,
                                fill="dark green", outline="dark green")

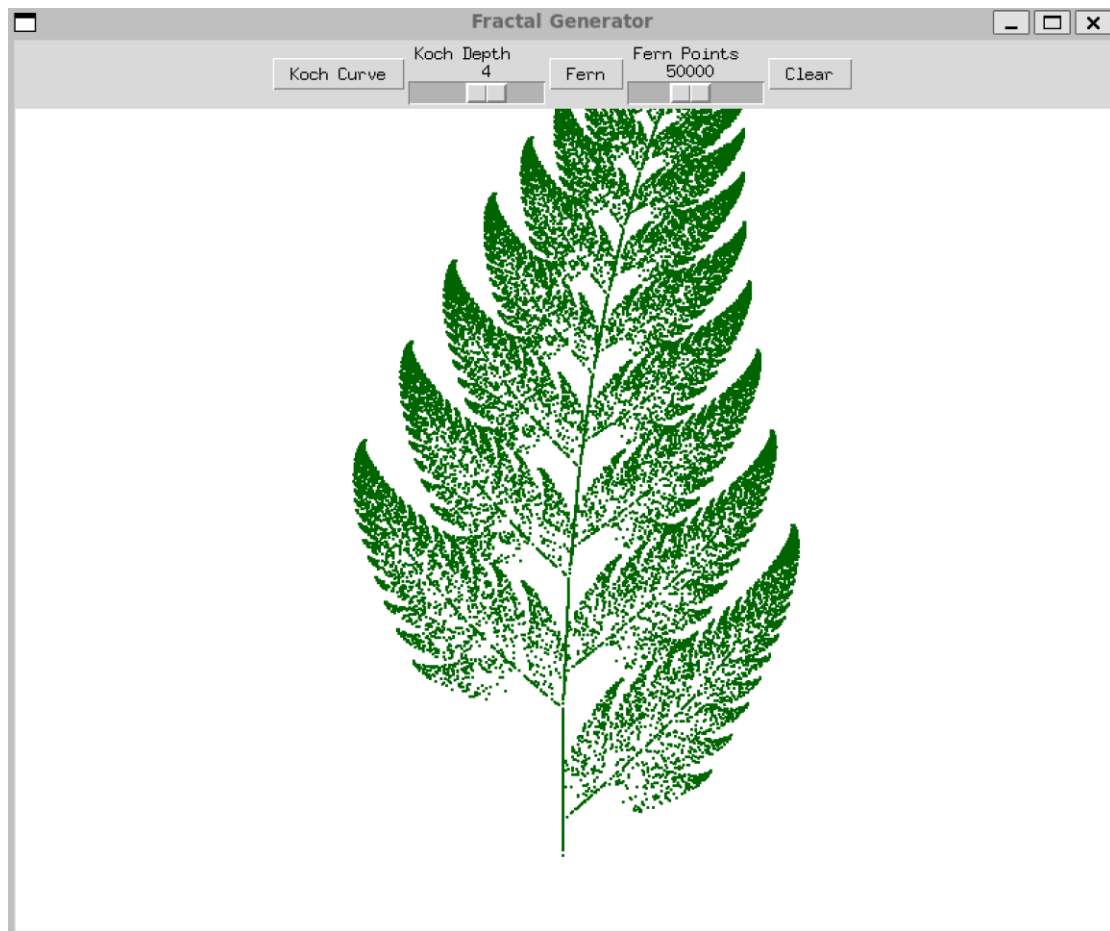
```

四、实验结果

1. 科赫曲线（Koch 曲线）



2. 蕨类植物



实验四

一、实验原理

这个实验的目的是通过 Python 和 Tkinter 库创建一个简单的 3D 场景渲染器。渲染器使用光线追踪（Ray Tracing）的方法来计算每个像素的颜色，从而模拟光照效果和反射。

光线追踪的基本原理

1. **相机模型**：假设有一个虚拟的相机，它从一个特定的位置（**camera**）朝向屏幕（**Canvas**）发射射线。
2. **射线生成**：对于画布上的每一个像素，根据像素坐标计算出该像素在视图空间中的射线方向。
3. **光线追踪**：从相机位置沿着射线追踪，检查射线是否与场景中的物体相交。
4. **光照计算**：如果射线与物体相交，则计算交点处的光照效果，包括漫反射、镜面反射和环境光。
5. **颜色合成**：根据计算出的颜色值将像素在画布上绘制出来。

二、算法实现思路

1. 初始化场景

- a) 设置相机位置、光源位置和强度等参数。
- b) 定义场景中的物体，如球体和地板，并为其设置颜色、反射率和镜面反射指数。

2. 射线生成

根据像素坐标计算出射线的方向向量。

3. 光线追踪

- a) 从相机位置沿着射线方向追踪，检查射线是否与场景中的物体相交。
- b) 计算射线与物体的交点，并确定在该点的颜色（包括直接光照和反射）。

4. 光照计算

- a) 对于每个交点，根据光源的位置、法线、视图方向等计算漫反射和镜面反

射分量。

b) 环境光通过全局照明或者预设的环境颜色来处理。

5. 显示渲染结果

将渲染出的颜色值绘制到画布上。

三、核心代码

核心部分是光线追踪算法的实现，包括射线生成、物体检测和光照计算。以下是关键函数：

1. trace_ray 函数

这个函数负责沿着一条射线追踪，并计算交点处的颜色。

```
def trace_ray(self, origin, direction):
    aspect_ratio = self.width / self.height
    fov = math.pi / 3 # 60 degrees

    color = np.array([0.0, 0.0, 0.0])

    for y in range(self.height):
        for x in range(self.width):
            screen_x = (2 * (x + 0.5) / self.width - 1) * math.tan(fov/2) *
            aspect_ratio
            screen_y = (1 - 2 * (y + 0.5) / self.height) * math.tan(fov/2)
            direction = self.normalize(np.array([screen_x, screen_y, 1]))

            # Trace ray and store color in buffer
            color += self.trace_ray(self.camera, direction)

    return np.clip(color / (self.width * self.height), 0, 1)
```

2. compute_lighting 函数

这个函数计算射线与物体交点处的光照效果。

```
def compute_lighting(self, hit_point, normal, view_dir, specular, color):
    # Ambient light
    ambient_color = np.array([0.1, 0.1, 0.1])

    # Diffuse lighting
    diffuse_color = np.zeros(3)
```

```

for light in self.lights:
    light_dir = self.normalize(light - hit_point)
    diff = max(np.dot(normal, light_dir), 0)
    diffuse_color += color * diff

# Specular lighting
specular_color = np.zeros(3)
for light in self.lights:
    light_dir = self.normalize(light - hit_point)
    reflect_dir = self.reflect(-light_dir, normal)
    spec = max(np.dot(view_dir, reflect_dir), 0) ** specular
    specular_color += color * spec

return ambient_color + diffuse_color + specular_color

```

3. display_buffer 函数

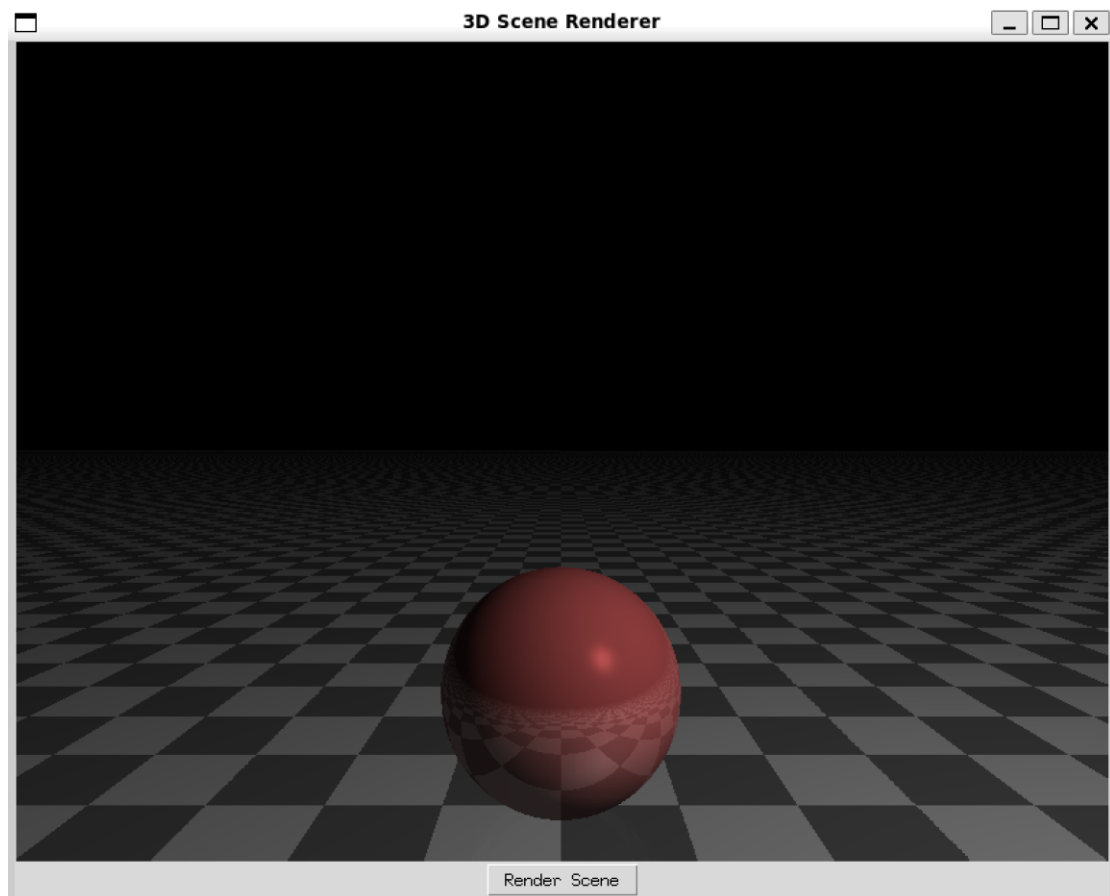
将渲染结果显示在画布上。

```

def display_buffer(self):
    for y in range(self.height):
        for x in range(self.width):
            r, g, b = self.buffer[y, x].astype(np.int32)
            color = f'#{r:02x}{g:02x}{b:02x}'
            self.canvas.create_rectangle(x, y, x+1, y+1, fill=color, outline=color)

```

四、实验结果



附录

1. 实验一完整代码

```
import tkinter as tk
from tkinter import messagebox
from PIL import Image, ImageTk
import math
from collections import deque

class DrawingApp:
    def __init__(self, root):
        self.root = root
        self.root.title("图形绘制与填充")

        # 创建画布和 PhotoImage
        self.canvas_width = 800
        self.canvas_height = 600
        self.image = Image.new("RGB", (self.canvas_width, self.canvas_height),
"white")
        self.photo = ImageTk.PhotoImage(self.image)

        self.canvas = tk.Canvas(root, width=self.canvas_width,
height=self.canvas_height, bg='white')
        self.canvas.pack(side=tk.BOTTOM)
        self.canvas_image = self.canvas.create_image((0, 0), anchor=tk.NW,
image=self.photo)

        # 创建控制按钮
        self.control_frame = tk.Frame(root)
        self.control_frame.pack(side=tk.TOP)

        tk.Button(self.control_frame, text="直线", command=lambda:
self.set_mode("line")).pack(side=tk.LEFT)
        tk.Button(self.control_frame, text="椭圆", command=lambda:
self.set_mode("ellipse")).pack(side=tk.LEFT)
        tk.Button(self.control_frame, text="多边形", command=lambda:
self.set_mode("polygon")).pack(side=tk.LEFT)
        tk.Button(self.control_frame, text="四联通填充", command=lambda:
self.set_fill_mode("flood_fill_four")).pack(side=tk.LEFT)
        tk.Button(self.control_frame, text="清除",
command=self.clear_canvas).pack(side=tk.LEFT)
```

```

# 初始化变量
self.mode = "line"
self.fill_mode = None
self.points = []
self.polygon_points = []
self.drawing = False

# 绑定鼠标事件
self.canvas.bind("<Button-1>", self.on_click)
self.canvas.bind("<Button-3>", self.on_right_click)
self.canvas.bind("<Motion>", self.on_motion)

def set_mode(self, mode):
    self.mode = mode
    self.fill_mode = None
    self.points = []
    self.polygon_points = []
    self.drawing = False
    print(f'模式切换为: {self.mode}')

def set_fill_mode(self, fill_mode):
    self.fill_mode = fill_mode
    self.mode = "fill"
    print(f'填充模式切换为: {self.fill_mode}')

def clear_canvas(self):
    self.image = Image.new("RGB", (self.canvas_width, self.canvas_height),
"white")
    self.photo = ImageTk.PhotoImage(self.image)
    self.canvas.itemconfig(self.canvas_image, image=self.photo)
    self.points = []
    self.polygon_points = []
    self.drawing = False
    self.fill_mode = None
    self.mode = "line"
    print("画布已清除")

def put_pixel(self, x, y, color=(0, 0, 0)):
    if 0 <= x < self.canvas_width and 0 <= y < self.canvas_height:
        self.image.putpixel((x, y), color)

def update_canvas(self):
    self.photo = ImageTk.PhotoImage(self.image)
    self.canvas.itemconfig(self.canvas_image, image=self.photo)

```



```

def bresenham_line(self, x1, y1, x2, y2, color=(0, 0, 0)):
    dx = abs(x2 - x1)
    dy = abs(y2 - y1)
    sx = 1 if x1 < x2 else -1
    sy = 1 if y1 < y2 else -1
    err = dx - dy

    x, y = x1, y1
    while True:
        self.put_pixel(x, y, color)
        if x == x2 and y == y2:
            break
        e2 = 2 * err
        if e2 > -dy:
            err -= dy
            x += sx
        if e2 < dx:
            err += dx
            y += sy
    self.update_canvas()

def midpoint_ellipse(self, center_x, center_y, a, b, color=(0, 0, 0)):
    x = 0
    y = b

    # Region 1
    d1 = (b * b) - (a * a * b) + (0.25 * a * a)
    dx = 2 * b * b * x
    dy = 2 * a * a * y

    while dx < dy:
        self.put_pixel(center_x + x, center_y + y, color)
        self.put_pixel(center_x - x, center_y + y, color)
        self.put_pixel(center_x + x, center_y - y, color)
        self.put_pixel(center_x - x, center_y - y, color)

        if d1 < 0:
            x += 1
            dx += 2 * b * b
            d1 += dx + b * b
        else:
            x += 1
            y -= 1

```

```

        dx += 2 * b * b
        dy -= 2 * a * a
        d1 += dx - dy + b * b

# Region 2
d2 = ((b * b) * ((x + 0.5) * (x + 0.5))) + \
      ((a * a) * ((y - 1) * (y - 1))) - \
      (a * a * b * b)

while y >= 0:
    self.put_pixel(center_x + x, center_y + y, color)
    self.put_pixel(center_x - x, center_y + y, color)
    self.put_pixel(center_x + x, center_y - y, color)
    self.put_pixel(center_x - x, center_y - y, color)

    if d2 > 0:
        y -= 1
        dy -= 2 * a * a
        d2 += a * a - dy
    else:
        y -= 1
        x += 1
        dx += 2 * b * b
        dy -= 2 * a * a
        d2 += dx - dy + a * a
    self.update_canvas()

def scan_line_fill(self, points, fill_color=(255, 0, 0)):
    if len(points) < 3:
        messagebox.showerror("错误", "多边形至少需要三个点")
        return

    # 找到多边形的边界
    min_y = min(p[1] for p in points)
    max_y = max(p[1] for p in points)

    # 对每一条扫描线
    for y in range(int(min_y), int(max_y) + 1):
        intersections = []

        # 计算扫描线与多边形边的交点
        for i in range(len(points)):
            j = (i + 1) % len(points)
            y1, y2 = points[i][1], points[j][1]

```

```

        if min(y1, y2) <= y <= max(y1, y2):
            if y1 != y2:
                x = points[i][0] + (points[j][0] - points[i][0]) * (y - y1) / (y2 -
y1)

                intersections.append(x)

    # 对交点进行排序
    intersections.sort()

    # 两两配对填充
    for i in range(0, len(intersections), 2):
        if i + 1 < len(intersections):
            x_start = int(math.ceil(intersections[i]))
            x_end = int(math.floor(intersections[i+1]))
            for x in range(x_start, x_end + 1):
                self.put_pixel(x, y, fill_color)
    self.update_canvas()

def flood_fill_four_connected(self, x, y, fill_color=(255, 0, 0), boundary_color=(0, 0,
0)):
    # 获取目标颜色
    target_pixel = self.get_pixel_color(x, y)
    if target_pixel == fill_color or target_pixel == boundary_color:
        return

    # 使用队列实现迭代式的四联通漫水填充
    queue = deque()
    queue.append((x, y))
    while queue:
        current_x, current_y = queue.popleft()
        if 0 <= current_x < self.canvas_width and 0 <= current_y <
self.canvas_height:
            current_color = self.get_pixel_color(current_x, current_y)
            if current_color == target_pixel and current_color != boundary_color:
                self.put_pixel(current_x, current_y, fill_color)
                queue.append((current_x + 1, current_y))
                queue.append((current_x - 1, current_y))
                queue.append((current_x, current_y + 1))
                queue.append((current_x, current_y - 1))
    self.update_canvas()

def get_pixel_color(self, x, y):
    # 获取指定像素的颜色
    return self.image.getpixel((x, y))

```

```

def on_click(self, event):
    if self.mode == "line":
        self.points.append((event.x, event.y))
        if len(self.points) == 2:
            self.bresenham_line(int(self.points[0][0]), int(self.points[0][1]),
                                int(self.points[1][0]), int(self.points[1][1]))
            self.points = []

    elif self.mode == "ellipse":
        self.points.append((event.x, event.y))
        if len(self.points) == 2:
            center_x = self.points[0][0]
            center_y = self.points[0][1]
            dx = abs(self.points[1][0] - center_x)
            dy = abs(self.points[1][1] - center_y)
            self.midpoint_ellipse(int(center_x), int(center_y), int(dx), int(dy))
            self.points = []

    elif self.mode == "polygon":
        self.polygon_points.append((event.x, event.y))
        if len(self.polygon_points) > 1:
            self.bresenham_line(int(self.polygon_points[-2][0]),
                                int(self.polygon_points[-2][1]),
                                int(self.polygon_points[-1][0]),
                                int(self.polygon_points[-1][1]))

    elif self.mode == "fill":
        if self.fill_mode == "flood_fill_four":
            if len(self.polygon_points) > 2:
                # 提示用户点击填充点
                messagebox.showinfo("提示", "请点击多边形内部的一个点进行四联通填充")
                # 绑定填充点击事件
                self.canvas.bind("<Button-1>", lambda e:
self.perform_flood_fill(e, mode="four"))
            else:
                messagebox.showerror("错误", "多边形至少需要三个点进行填充")

    def perform_flood_fill(self, event, mode="four"):
        x, y = event.x, event.y
        boundary_color = (0, 0, 0) # 假设边界颜色为黑色
        fill_color = (255, 0, 0) # 填充颜色为红色

```

```

        if mode == "four":
            self.flood_fill_four_connected(x, y, fill_color=fill_color,
boundary_color=boundary_color)
            messagebox.showinfo("完成", "四联通填充完成")
            # 重新绑定鼠标事件
            self.canvas.bind("<Button-1>", self.on_click)

    def on_right_click(self, event):
        if self.mode == "polygon" and len(self.polygon_points) > 2:
            self.bresenham_line(int(self.polygon_points[-1][0]),
int(self.polygon_points[-1][1]),
                                int(self.polygon_points[0][0]),
int(self.polygon_points[0][1]))
            self.drawing = True

    def on_motion(self, event):
        pass

    def fill_polygon(self):
        if self.mode == "fill" and self.fill_mode == "flood_fill_four":
            if len(self.polygon_points) > 2:
                # 提示用户点击填充点
                messagebox.showinfo("提示", "请点击多边形内部的一个点进行四
联通填充")
                # 绑定填充点点击事件
                self.canvas.bind("<Button-1>", lambda e: self.perform_flood_fill(e,
mode="four"))
            else:
                messagebox.showerror("错误", "多边形至少需要三个点进行填充")
        else:
            messagebox.showerror("错误", "请选择一个填充模式并绘制多边形")

def main():
    root = tk.Tk()
    app = DrawingApp(root)
    root.mainloop()

if __name__ == "__main__":
    main()

```

2. 实验二完整代码

```
import tkinter as tk
```

```

import numpy as np
from scipy.special import comb # 确保正确导入 comb

class CurveDrawer:
    def __init__(self, root):
        self.root = root
        self.root.title("B-spline & Bezier Curve Generator")

        # 创建画布
        self.canvas = tk.Canvas(root, width=800, height=600, bg='white')
        self.canvas.pack(side=tk.BOTTOM)

        # 控制按钮
        self.control_frame = tk.Frame(root)
        self.control_frame.pack(side=tk.TOP)

        tk.Button(self.control_frame, text="B Spline", command=lambda:
self.set_mode("bspline")).pack(side=tk.LEFT)
        tk.Button(self.control_frame, text="Bezier (Bernstein)", command=lambda:
self.set_mode("bezier")).pack(side=tk.LEFT)
        tk.Button(self.control_frame, text="Bezier (Casteljau)", command=lambda:
self.set_mode("casteljau")).pack(side=tk.LEFT) # 新按钮
        tk.Button(self.control_frame,
text="Clear",command=self.clear_canvas).pack(side=tk.LEFT)

        # 初始化变量
        self.mode = "bspline"
        self.control_points = []

        # 绑定鼠标事件
        self.canvas.bind("<Button-1>", self.on_click)
        self.canvas.bind("<Double-Button-1>", self.on_double_click)

    def set_mode(self, mode):
        self.mode = mode
        self.clear_canvas()

    def clear_canvas(self):
        self.canvas.delete("all")
        self.control_points = []

    def draw_point(self, x, y, color="black"):
        self.canvas.create_oval(x-3, y-3, x+3, y+3, fill=color)

```

```

def draw_control_polygon(self):
    if len(self.control_points) > 1:
        for i in range(len(self.control_points)-1):
            x1, y1 = self.control_points[i]
            x2, y2 = self.control_points[i+1]
            self.canvas.create_line(x1, y1, x2, y2, dash=(2,2))

def draw_bspline(self):
    points = np.array(self.control_points)
    n = len(points) - 1

    # 生成均匀二次 B 样条的节点
    t = np.linspace(0, 1, 100)

    # 绘制曲线
    curve_points = []
    for i in range(n-1):
        if i + 2 <= n:
            # 计算二次 B 样条基函数
            for tj in t:
                x = (1-tj)**2/2 * points[i][0] + \
                    (1/2 + tj - tj**2) * points[i+1][0] + \
                    tj**2/2 * points[i+2][0]

                y = (1-tj)**2/2 * points[i][1] + \
                    (1/2 + tj - tj**2) * points[i+1][1] + \
                    tj**2/2 * points[i+2][1]

                curve_points.append((x, y))

    # 绘制曲线段
    for i in range(len(curve_points)-1):
        self.canvas.create_line(curve_points[i][0], curve_points[i][1],
                                curve_points[i+1][0], curve_points[i+1][1],
                                fill="blue", width=2)

def bezier_curve(self):
    if len(self.control_points) < 2:
        return

    def bernstein(i, n, t):
        return comb(n, i) * (t ** i) * ((1 - t) ** (n - i))

    points = []

```

```

n = len(self.control_points) - 1

for t in np.arange(0, 1.01, 0.01):
    x = y = 0
    for i in range(n + 1):
        basis = bernstein(i, n, t)
        x += basis * self.control_points[i][0]
        y += basis * self.control_points[i][1]
    points.append((x, y))

for i in range(len(points)-1):
    self.canvas.create_line(points[i][0], points[i][1],
                           points[i+1][0], points[i+1][1],
                           fill='red', width=2)

def casteljau_curve(self):
    if len(self.control_points) < 2:
        return

def de_casteljau(points, t):
    """
    递归实现 de Casteljau 算法，并返回每一层的插值点。
    """
    layers = [points]
    while len(points) > 1:
        new_points = []
        for i in range(len(points)-1):
            x = (1 - t) * points[i][0] + t * points[i+1][0]
            y = (1 - t) * points[i][1] + t * points[i+1][1]
            new_points.append((x, y))
        layers.append(new_points)
        points = new_points
    return layers

points_on_curve = []
for t in np.arange(0, 1.01, 0.01):
    layers = de_casteljau(self.control_points, t)
    final_point = layers[-1][0]
    points_on_curve.append(final_point)

# 可视化每一层的插值点和线条
for layer in layers[:-1]: # 不绘制最后一层的单个点
    for i in range(len(layer)-1):
        x1, y1 = layer[i]

```



```

        x2, y2 = layer[i+1]
        self.canvas.create_line(x1, y1, x2, y2, fill='lightgray', dash=(1,1))
    for point in layer:
        self.canvas.create_oval(point[0]-2, point[1]-2, point[0]+2,
point[1]+2, fill='orange', outline=")

    # 绘制贝塞尔曲线
    for i in range(len(points_on_curve)-1):
        self.canvas.create_line(points_on_curve[i][0], points_on_curve[i][1],
                                points_on_curve[i+1][0],
points_on_curve[i+1][1],
                                fill='green', width=2)

    def on_click(self, event):
        self.control_points.append((event.x, event.y))
        self.draw_point(event.x, event.y)
        self.draw_control_polygon()

    def on_double_click(self, event):
        if self.mode == "bspline":
            self.draw_bspline()
        elif self.mode == "bezier":
            self.bezier_curve()
        elif self.mode == "casteljau":
            self.casteljau_curve()

def main():
    root = tk.Tk()
    app = CurveDrawer(root)
    root.mainloop()

if __name__ == "__main__":
    main()

```

3. 实验三完整代码

```

import tkinter as tk
import numpy as np
import math

class FractalGenerator:
    def __init__(self, root):
        self.root = root

```

```

self.root.title("Fractal Generator")

# Create canvas
self.canvas = tk.Canvas(root, width=800, height=600, bg='white')
self.canvas.pack(side=tk.BOTTOM)

# Control panel
self.control_frame = tk.Frame(root)
self.control_frame.pack(side=tk.TOP)

# Koch curve controls
tk.Button(self.control_frame, text="Koch Curve",
command=self.draw_koch).pack(side=tk.LEFT)
self.koch_depth = tk.Scale(self.control_frame, from_=1, to=6,
orient=tk.HORIZONTAL, label="Koch Depth")
self.koch_depth.pack(side=tk.LEFT)
self.koch_depth.set(4)

# Fern controls
tk.Button(self.control_frame, text="Fern",
command=self.draw_fern).pack(side=tk.LEFT)
self.fern_iterations = tk.Scale(self.control_frame, from_=10000, to=100000,
orient=tk.HORIZONTAL, label="Fern
Points")
self.fern_iterations.pack(side=tk.LEFT)
self.fern_iterations.set(50000)

# Clear button
tk.Button(self.control_frame, text="Clear",
command=self.clear_canvas).pack(side=tk.LEFT)

def clear_canvas(self):
    self.canvas.delete("all")

def draw_line(self, x1, y1, x2, y2, color="black"):
    self.canvas.create_line(x1, y1, x2, y2, fill=color)

def koch_curve_points(self, start, end, depth):
    if depth == 0:
        return [start, end]

# Calculate required points
dx = end[0] - start[0]
dy = end[1] - start[1]

```

```

p1 = start
p2 = (start[0] + dx/3, start[1] + dy/3)

# Calculate the position of the peak point
angle = math.pi/3 # 60 degrees
p3x = p2[0] + (dx/3)*math.cos(angle) - (dy/3)*math.sin(angle)
p3y = p2[1] + (dx/3)*math.sin(angle) + (dy/3)*math.cos(angle)
p3 = (p3x, p3y)

p4 = (start[0] + 2*dx/3, start[1] + 2*dy/3)
p5 = end

# Recursive calls
curve1 = self.koch_curve_points(p1, p2, depth-1)
curve2 = self.koch_curve_points(p2, p3, depth-1)
curve3 = self.koch_curve_points(p3, p4, depth-1)
curve4 = self.koch_curve_points(p4, p5, depth-1)

# Combine all points (removing duplicates at joints)
return curve1[:-1] + curve2[:-1] + curve3[:-1] + curve4

def draw_koch(self):
    self.clear_canvas()
    depth = self.koch_depth.get()

    # Set up initial line (centered horizontally, in lower third of canvas)
    start_x = 100
    end_x = 700
    y = 400

    points = self.koch_curve_points((start_x, y), (end_x, y), depth)

    # Draw the curve
    for i in range(len(points)-1):
        x1, y1 = points[i]
        x2, y2 = points[i+1]
        self.draw_line(x1, y1, x2, y2, "blue")

def draw_fern(self):
    self.clear_canvas()
    iterations = self.fern_iterations.get()

    # Transformation matrices and probabilities

```

```

p = np.array([0.85, 0.07, 0.07, 0.01]) # Probabilities

# Starting point
x, y = 0, 0

# Scale and translate coordinates to fit canvas
def transform_coords(x, y):
    return (400 + x*70, 550 - y*70)

# Draw points
for _ in range(iterations):
    r = np.random.random()
    if r < p[0]:
        # Stem
        x_new = 0.85*x + 0.04*y
        y_new = -0.04*x + 0.85*y + 1.6
    elif r < p[0] + p[1]:
        # Left leaflet
        x_new = 0.2*x - 0.26*y
        y_new = 0.23*x + 0.22*y + 1.6
    elif r < p[0] + p[1] + p[2]:
        # Right leaflet
        x_new = -0.15*x + 0.28*y
        y_new = 0.26*x + 0.24*y + 0.44
    else:
        # Successive stems
        x_new = 0
        y_new = 0.16*y

    x, y = x_new, y_new

# Transform and plot point
plot_x, plot_y = transform_coords(x, y)
self.canvas.create_rectangle(plot_x, plot_y, plot_x+1, plot_y+1,
                             fill="dark green", outline="dark green")

def main():
    root = tk.Tk()
    app = FractalGenerator(root)
    root.mainloop()

if __name__ == "__main__":
    main()

```

4. 实验四完整代码

```
import tkinter as tk
import numpy as np
import math

class Scene3DRenderer:
    def __init__(self, root):
        self.root = root
        self.root.title("3D Scene Renderer")

        # Canvas setup
        self.width = 800
        self.height = 600
        self.canvas = tk.Canvas(root, width=self.width, height=self.height, bg='black')
        self.canvas.pack()

        # Create image buffer
        self.buffer = np.zeros((self.height, self.width, 3), dtype=np.float64)
        self.z_buffer = np.full((self.height, self.width), float('inf'))

        # Scene parameters
        self.camera = np.array([0, 2, -6])
        self.light_pos = np.array([5, 5, -5])
        self.light_color = np.array([1, 1, 1])
        self.ambient = 0.1

        # Initialize scene objects
        self.init_scene()

        # Render button
        tk.Button(root, text="Render Scene", command=self.render_scene).pack()

    def init_scene(self):
        # Define scene objects
        self.sphere = {
            'center': np.array([0, 0, 0]),
            'radius': 1.0,
            'color': np.array([0.7, 0.3, 0.3]),
            'specular': 50,
            'reflectivity': 0.3
        }

        self.floor = {
```

```

        'y': -1,
        'color': np.array([0.5, 0.5, 0.5]),
        'specular': 10,
        'reflectivity': 0.1
    }

    # Checkerboard texture for floor
    self.checker_size = 1.0

def normalize(self, vector):
    return vector / np.linalg.norm(vector)

def reflect(self, vector, normal):
    return vector - 2 * np.dot(vector, normal) * normal

def sphere_intersect(self, origin, direction):
    b = 2 * np.dot(direction, origin - self.sphere['center'])
    c = np.linalg.norm(origin - self.sphere['center'])**2 - self.sphere['radius']**2
    delta = b**2 - 4*c

    if delta < 0:
        return None

    t1 = (-b - np.sqrt(delta)) / 2
    t2 = (-b + np.sqrt(delta)) / 2

    if t1 < 0 and t2 < 0:
        return None

    return min(t for t in [t1, t2] if t > 0)

def floor_intersect(self, origin, direction):
    if abs(direction[1]) < 1e-6:
        return None

    t = -(origin[1] - self.floor['y']) / direction[1]
    return t if t > 0 else None

def get_floor_color(self, point):
    # Checkerboard pattern
    x = math.floor(point[0] / self.checker_size)
    z = math.floor(point[2] / self.checker_size)
    if (x + z) % 2 == 0:
        return self.floor['color']

```

```

        return self.floor['color'] * 0.5

def compute_lighting(self, point, normal, view_dir, specular, color):
    # Ambient light
    intensity = self.ambient

    # Direction to light
    light_dir = self.normalize(self.light_pos - point)

    # Diffuse lighting
    diffuse = max(np.dot(normal, light_dir), 0)
    intensity += diffuse

    # Specular lighting
    if specular > 0:
        reflect_dir = self.reflect(-light_dir, normal)
        spec = max(np.dot(reflect_dir, view_dir), 0) ** specular
        intensity += 0.5 * spec

    return np.clip(color * intensity, 0, 1)

def trace_ray(self, origin, direction, depth=3):
    if depth <= 0:
        return np.zeros(3)

    # Check sphere intersection
    t_sphere = self.sphere_intersect(origin, direction)

    # Check floor intersection
    t_floor = self.floor_intersect(origin, direction)

    # Find closest intersection
    if t_sphere is None and t_floor is None:
        return np.zeros(3)

    if t_sphere is None:
        t = t_floor
        hit_point = origin + direction * t
        normal = np.array([0, 1, 0])
        color = self.get_floor_color(hit_point)
        specular = self.floor['specular']
        reflectivity = self.floor['reflectivity']
    elif t_floor is None:
        t = t_sphere

```

```

        hit_point = origin + direction * t
        normal = self.normalize(hit_point - self.sphere['center'])
        color = self.sphere['color']
        specular = self.sphere['specular']
        reflectivity = self.sphere['reflectivity']
    else:
        t = min(t_sphere, t_floor)
        if t == t_sphere:
            hit_point = origin + direction * t
            normal = self.normalize(hit_point - self.sphere['center'])
            color = self.sphere['color']
            specular = self.sphere['specular']
            reflectivity = self.sphere['reflectivity']
        else:
            hit_point = origin + direction * t
            normal = np.array([0, 1, 0])
            color = self.get_floor_color(hit_point)
            specular = self.floor['specular']
            reflectivity = self.floor['reflectivity']

    # Compute lighting
    view_dir = self.normalize(-direction)
    color = self.compute_lighting(hit_point, normal, view_dir, specular, color)

    # Compute reflection
    if reflectivity > 0:
        reflect_dir = self.reflect(direction, normal)
        reflect_origin = hit_point + normal * 1e-4 # Offset to avoid self-
intersection
        reflect_color = self.trace_ray(reflect_origin, reflect_dir, depth-1)
        color = color * (1 - reflectivity) + reflect_color * reflectivity

    return color

def render_scene(self):
    aspect_ratio = self.width / self.height
    fov = math.pi / 3 # 60 degrees

    for y in range(self.height):
        for x in range(self.width):
            # Calculate ray direction
            screen_x = (2 * (x + 0.5) / self.width - 1) * math.tan(fov/2) *
aspect_ratio
            screen_y = (1 - 2 * (y + 0.5) / self.height) * math.tan(fov/2)

```



```

        direction = self.normalize(np.array([screen_x, screen_y, 1]))

        # Trace ray and store color in buffer
        color = self.trace_ray(self.camera, direction)
        self.buffer[y, x] = np.clip(color * 255, 0, 255)

    # Display the rendered image
    self.display_buffer()

def display_buffer(self):
    # Convert buffer to hex colors and display on canvas
    for y in range(self.height):
        for x in range(self.width):
            r, g, b = self.buffer[y, x].astype(np.int32)
            color = f'#{r:02x}{g:02x}{b:02x}'
            self.canvas.create_rectangle(x, y, x+1, y+1, fill=color, outline=color)

def main():
    root = tk.Tk()
    app = Scene3DRenderer(root)
    root.mainloop()

if __name__ == "__main__":
    main()

```