《计算机图形学》5 月报告

学号：171860578，姓名：张梓悦，2846728884@qq.com

2020 年 5 月 31 日

1 综述

在 3 月，我先对图形学相关算法进行了学习，并仔细研究了助教所提供的框架代码。最终完成了直线绘制、多边形绘制、椭圆绘制以及旋转、缩放、平移、裁剪的功能。并在 Cli 程序中添加了相应操作。

在 4 月，我继续补充了算法部分的代码，添加了曲线绘制算法，并测试了每一个指令的正确性，修改了一些代码中存在的 bug，完成了 Algorithm 以及 Cli 中所有需要添加的功能。并开始进行 qt 的学习，为 gui 部分的书写做好准备。

在 5 月，充分了解 qt 的功能原理后开始对 gui 部分进行了书写，完成了直线绘制、曲线绘制、多边形绘制、椭圆绘制以及旋转、缩放、平移、裁剪对应的 gui 操作，同时也为 gui 添加了重置画布、保存画布的功能，并为 gui 设计了更为人性化的操作方式。已在 5 月份完成了助教所要求的全部基础功能，预计在 6 月可以对 gui 做出更加美观更加人性化的设计并添加一些其他的拓展功能。

2 算法介绍

2.1 直线算法

2.1.1 DDA 算法

算法原理：

DDA（*Digital Differential Analyzer*）算法是一种利用计算两个坐标方向差分确定线段每个像素点的线段扫描转换算法。下面结合我编写的该算法`python`代码来做具体的解释。

定义线段两端点坐标分别为 (x0, y0), (x1, y1)。用直线方程 $y = mx + b$ 来表示绘制线段所在直线。先考虑两端点具有正斜率，且是从左端点到右端点进行处理的线段。假设 $m \leqslant 2$，则在 x 的单位间隔内（单位间隔为 1，即一个个像素点的移动），每个点取样并计算 y 的值。其中，初始时，$x_k = x0$，$y_k = y0$。由于 $y_k = mx_k + b$，且 $y_{k+1} = mx_{k+1} + b$，由于 $x_{k+1} = x_k + 1$，故 $y_{k+1} = mx_{k+1} + b = mx_k + m + 1$，因此

$$y_{k+1} = y_k + m \tag{1}$$

如果考虑到起始点横坐标 x0 大于终止点横坐标 x1，则让 x 和 y 的递增量 1 和 m 分别取负即可（即改为-1 和-m）。

若 m>1，则将 x 和 y 的规则交换。即在单位 y 间隔 (y=1) 取样，并计算每个连续的 x 值。因此有

$$x_{k+1} = x_k + (1/m); \tag{2}$$

如果考虑到起始点横坐标 y0 大于终止点横坐标 y1，则让 x 和 y 的递增量 1/m 和 1 分别取负即可（即改为-1/m 和-1）。

确定好 x 和 y 的递增量，即可循环绘制直线上的每个像素点。

综上可知 $y = kx + b, 0 < k < 1$ 时，x 每增加 1，y 增加 k，可令 $\Delta x = 1, \Delta y = k$。则可以从 x 的起始点 x_0 每次增加 Δx 直到 x_1，同时每次对 y 也增加 Δy，对于小数情况进行四舍五入即可，最终能够得到直线 $y = kx + b$。其他情况同理。

算法实现：

```
    if(abs(x1-x0)>=abs(y1-y0)):
        length=abs(x1-x0)
    else:
        length=abs(y1-y0)
        if (length==0):
            result.append((x0,y0))
            return result
        dx=(float)(x1-x0)/length
        dy=(float)(y1-y0)/length
        i=1
        x=x0
        y=y0
        while(i<=length):
            result.append((int(x+0.5),int(y+0.5)))
            x=x+dx
            y=y+dy
            i+=1
```

算法效果：图 1

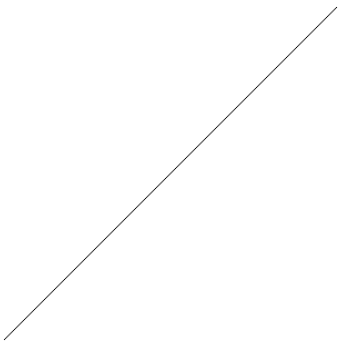

图 1: DDA 算法

2.1.2 Bresenham 算法

算法原理：

该算法避免了浮点运算，相当于 DDA 算法的一种改进算法。

当 x 方向是每一步的移动方向时（斜率小于 1），以每一小格的中点为界，如果当前的 yi 在中点下方，则 y 取 y_i; 如果当前的 yi 在中点上方，则 y 取 $y_i + 1$。这样我们只需仅维护一个判别变量 p 即可。如果 p 大于等于 0 则下一个 y 的值 +1，同时 p 加上 2*dy-2*dx。如果 p 小于 0 则下一个 y 的值不变，同时 p 加上 2*dy。

算法实现：

```
    dx=abs(x1-x0)
    dy=abs(y1-y0)
    if(dx==0 and dy==0):
        result.append((x0,y0))
        return result
    gradient_flag=0
    if(dx<dy):
        gradient_flag=1
    if(gradient_flag==1):
        x0,y0=y0,x0
        x1,y1=y1,x1
        dx,dy=dy,dx
    xx=1
    if(x1-x0<0):
        xx=-1
    yy=1
    if(y1-y0<0):
        yy=-1
    p=2*dy-dx
    x=x0
    y=y0
    result.append((x,y))
    while(x!=x1):
        if(p>=0):
            p+=2*dy-2*dx
            y+=yy
        else:
            p+=2*dy
        x+=xx
        if(gradient_flag):
            result.append((y,x))
        else:
            result.append((x,y))
```

算法效果：图 2

2.2 椭圆算法（中点圆生成算法）

算法原理：

类似于 Bresenham 直线算法的思想，利用判别式选择像素，只需要做简单的整数运算。即根据判别式是否大于 0，来选择下一个 y 的值是不变还是减去 1，同时根据条件修改判别式的值。同时可以利用对称性，只需遍历椭圆的四分之一即右上方部分就能画出整个椭圆。

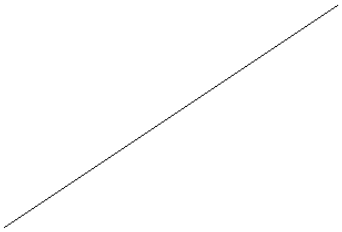

图 2: Bresenham 算法

算法实现：

```
    x0, y0 = p_list[0]
    x1, y1 = p_list[1]
    result = []
    xx=int((x0+x1)/2)
    yy=int((y0+y1)/2)
    a=int((abs(x1-x0))/2)
    b=int((abs(y1-y0))/2)
    p=float(b**2+a**2*(0.25-b))
    x=0
    y=b
    result.append((xx+x,yy+y))
    result.append((xx-x,yy+y))
    result.append((xx+x,yy-y))
    result.append((xx-x,yy-y))
    while(b**2*x<a**2*y):
        if(p<0):
            p+=float(b**2*(2*x+3))
        else:
            p+=float(b**2*(2*x+3)-a**2*(2*y-2))
            y-=1
        x+=1
        result.append((xx+x,yy+y))
        result.append((xx-x,yy+y))
        result.append((xx+x,yy-y))
        result.append((xx-x,yy-y))
    p=float((b*(x+0.5))**2+(a*(y-1))**2-(a*b)**2)
    while(y>0):
        if(p<0):
            p+=float(b**2*(2*x+2)+a**2*(-2*y+3))
            x+=1
        else:
            p+=float(a**2*(-2*y+3))
        y-=1
```

```
        result.append((xx+x,yy+y))
        result.append((xx-x,yy+y))
        result.append((xx+x,yy-y))
        result.append((xx-x,yy-y))
    return result
```

算法效果：图 3

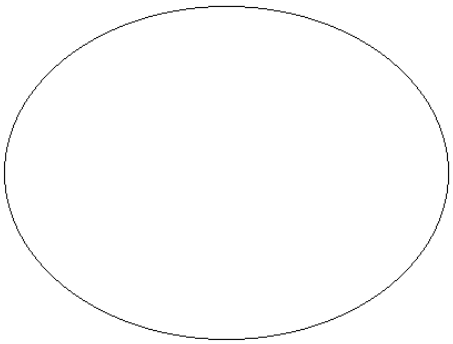

图 3: 中点圆生成算法

2.3 曲线算法

2.3.1 Bezier 算法

算法原理：

把 t 从 0 到 1 的移动过程中，在各个控制点连线的相应位置取点，并对相邻两条线上的点再次连线，重复以该过程使得没有可连接的两个点，即得到终的一个点。因此 t 从 0 到 1 的移动过程中计算出来的点的集合将构成目标曲线。

算法实现：

```
    def Bezier_point(n, t, control_point):
    while(n!=1):
        for i in range(0, n-1):
            x0,y0=control_point[i]
            x1,y1=control_point[i+1]
            x=float(x0*(1-t))+float(x1*t)
            y=float(y0*(1-t))+float(y1*t)
            control_point[i]=x,y
        n-=1
    return control_point[0]

    m=len(p_list)*10000
    for i in range(0, m):
        control_point=p_list.copy()
```

```
        t = float(i/m)
        x,y=Bezier_point(len(p_list), t, control_point)
        result.append((int(x+0.5),int(y+0.5)))
```

算法效果：图 4

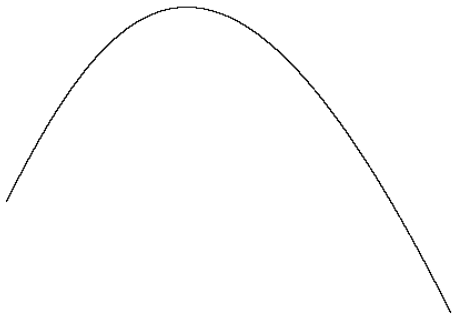

图 4: Bezier 算法

2.3.2 B-spline 算法

算法原理：

四个平面离散点可确定一条三次 B 样条曲线，可以先计算出三次 B 样条曲线的参数表达式，再由该参数表达式计算曲线上的每一个点即可。

算法实现：

```
    def deboox_cox(i, k, u):
    if k == 1:
        if i<=u and u<i+1:
            return 1
        else:
            return 0
    else:
        return (u-i)/(k-1)*deboox_cox(i,k-1,u)+(i+k-u)/(k-1)*deboox_cox(i+1,k-1,u)

    k = 3
    n=len(p_list)
    if(n<4):
        return result
    du=float(1/1000)
    u =float(k)

    while(u<n):
        x1,y1 = 0,0
        for i in range(0,n):
            x0,y0 = p_list[i]
```

```
            res=deboox_cox(i, k+1, u)
            x1 +=x0*res
            y1 +=y0*res
        result.append([round(x1), round(y1)])
        u+=du
```

算法效果：图 5

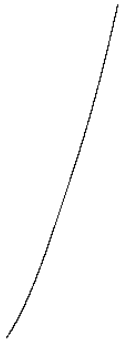

图 5: B-spline 算法

2.4 平移算法

算法原理：

不妨设 dx 为水平方向平移量，dy 为垂直方向平移量。

x,y 为原始坐标，x',y' 为平移后坐标，则：

$$\begin{cases} x' = x + dx \\ y' = y + dy \end{cases} \tag{3}$$

算法实现：

```
    result = []
    for x, y in p_list:
        result.append((x+dx,y+dy))
    return result
```

2.5 旋转算法

算法原理：

不妨设 x_r,y_r 为旋转中心点坐标，θ 为顺时针旋转角度。

x,y 为原始坐标，x',y' 为旋转后坐标，则：

$$\begin{cases} x' = x_r + (x - x_r)\cos\theta - (y - y_r)\sin\theta \\ y' = y_r + (x - x_r)\sin\theta + (y - y_r)\cos\theta \end{cases} \tag{4}$$

算法实现：

```
angel=float(r*math.pi/180)
cos=math.cos(angel)
sin=math.sin(angel)
result = []
for x0, y0 in p_list:
    x1=int(float(x)+float((x0-x)*cos)-float((y0-y)*sin)+0.5)
    y1=int(float(y)+float((x0-x)*sin)+float((y0-y)*cos)+0.5)
    result.append((x1,y1))
return result
```

2.6 缩放算法

算法原理：
不妨设 x_f,y_f 为缩放中心点坐标，s 为缩放倍数。
x,y 为原始坐标，x',y' 为缩放后坐标，则：

$$\begin{cases} x' = x \cdot s + x_f \cdot (1 - s) \\ y' = y \cdot s + y_f \cdot (1 - s) \end{cases} \tag{5}$$

算法实现：

```
result = []
for x0, y0 in p_list:
    x1=int(float(x0*s)+float(x*(1-s))+0.5)
    y1=int(float(y0*s)+float(y*(1-s))+0.5)
    result.append((x1,y1))
return result
```

2.7 线段裁剪算法

2.7.1 Cohen-Sutherland 算法

算法原理：
先将平面由裁剪框分成 9 个部分，对于任一端点 (x, y)，根据其坐标所在的区域，赋予一个 4 位的二进制码，判断图形元素是否落在裁剪窗口之内，如果有一部分在窗口之外再通过求交运算找出其位于内部的部分。

(1) 若 $x < x_{min}$，对应 code+1

(2) 若 $x > x_{max}$，对应 code+2

(3) 若 $y < y_{min}$，对应 code+4

(4) 若 $y > y_{max}$，对应 code+8

裁剪一条线段时，先求出端点 a 和 b 的编码 code1 和 code2，接下来进行如下算法：

(1) 如果 code1 和 code2 按位或的结果为 0，则说明 a 和 b 均在窗口内，即线段完全位于窗口之内，直接返回原来的端点值即可。

(2) 如果 code1 和 code2 按位与的结果不等于 0。说明 a 和 b 同时在窗口的上、下、左或右方，即线段完全位于窗口的外部，返回空或者一对相同的端点即可。

(3) 求出线段与窗口边界的交点，并用该交点的坐标值替换端点的坐标值。即在该交点处将线段分为两部分。

(4) 再次计算 code1 和 code2 的值，如果 code1 和 code2 按位或的结果不为 0，则裁剪失败，返回空或者一对相同的端点。

(5) 保存修改后的端点坐标

(6) 算法结束。

算法实现：

```
    x0,y0 = p_list[0]
    x1,y1 = p_list[1]
    code1=0
    code2=0
    if(x0<x_min):
        code1+=1
    if(x0>x_max):
        code1+=2
    if(y0<y_min):
        code1+=4
    if(y0>y_max):
        code1+=8
    if(x1<x_min):
        code2+=1
    if(x1>x_max):
        code2+=2
    if(y1<y_min):
        code2+=4
    if(y1>y_max):
        code2+=8

    if((code1|code2)==0):
        result=p_list
    elif((code1&code2)!=0):
        result=[[0,0],[0,0]]
    else:
        code=code1|code2
        if(code&1):
            yy=int(float((x_min-x1)*(y0-y1)/(x0-x1))+float(y1)+0.5)
            if(x0<x_min):
                x0=x_min
                y0=yy
            elif(x1<x_min):
```

```
            x1=x_min
            y1=yy
    if(code&2):
        yy=int(float((x_max-x1)*(y0-y1)/(x0-x1))+float(y1)+0.5)
        if(x0>x_max):
            x0=x_max
            y0=yy
        elif(x1>x_max):
            x1=x_max
            y1=yy
    if(code&4):
        xx=int(float((y_min-y1)*(x0-x1)/(y0-y1))+float(x1)+0.5)
        if(y0<y_min):
            x0=xx
            y0=y_min
        elif(y1<y_min):
            x1=xx
            y1=y_min
    if(code&8):
        xx=int(float((y_max-y1)*(x0-x1)/(y0-y1))+float(x1)+0.5)
        if(y0>y_max):
            x0=xx
            y0=y_max
        elif(y1>y_max):
            x1=xx
            y1=y_max
    code1=0
    code2=0
    if(x0<x_min):
        code1+=1
    if(x0>x_max):
        code1+=2
    if(y0<y_min):
        code1+=4
    if(y0>y_max):
        code1+=8
    if(x1<x_min):
        code2+=1
    if(x1>x_max):
        code2+=2
    if(y1<y_min):
        code2+=4
    if(y1>y_max):
        code2+=8
    if((code1|code2)==0):
        result=[[x0,y0], [x1,y1]]
    else:
        result=[[0,0],[0,0]]
```

2.7.2 Liang-Barsky 算法

算法原理：

先计算 p 数组：p=$[x0-x1, x1-x0, y0-y1, y1-y0]$

再计算 q 数组：q=$[x0-x\_min, x\_max-x0, y0-y\_min, y\_max-y0]$

接下来可分情况讨论，考虑 p 和 q 数组中每一对 p、q：

(1) 若 $p=0, q<0$，表明直线与裁剪框平行，但是在裁剪框外面。因此需要直接舍弃该直线，即返回空或者一对相同的端点。

(2) 若 $p=0, q\geq 0$，表明直线与裁剪框平行，且其延长线必然经过裁剪框的内部，接下来需要计算该线是在框的内部、外部还是与其相交

(3) 若 $p<0$，表明直线从裁剪边界的外部延伸到内部

(4) 若 $p>0$，表明直线从裁剪边界的内部延伸到外部

(5) 对于 (3) 和 (4) 的情况，找到直线与边界的交点，并计算出在该窗口内线段对应的参数 u_1 以及 u_2。u_1 由使直线是从外部延伸到窗口内部的边界决定。u_2 由使直线是从内部延伸到窗口外部的边界决定。先计算 $r=q/p$，则 $u_1=max(r,0)$，$u_2=min(r,1)$，如果 $u_1>u_2$，则这条直线完全在窗口外面，需要舍弃。否则，可根据 u_1 以及 u_2 这两个值，计算出裁剪后的线段的端点，最后返回这两个新端点即可。

算法实现：

```
    x0,y0 = p_list[0]
    x1,y1 = p_list[1]
    p=[x0-x1,x1-x0,y0-y1,y1-y0]
    q=[x0-x_min,x_max-x0,y0-y_min,y_max-y0]
    u1=float(0)
    u2=float(1)
    flag=False
    for i in range(0,4):
        if(p[i]==0 and q[i]<0):
            flag=True
        else:
            if(p[i]==0):
                continue
            r=float(q[i]/p[i])
            if(p[i]<0):
                u1=max(float(0),r)
            else:
                u2=min(float(1),r)
            if(u1>u2):
                flag=True
    if(flag==False):
        xx0=int(x0+u1*(x1-x0)+0.5)
        yy0=int(y0+u1*(y1-y0)+0.5)
        xx1=int(x0+u2*(x1-x0)+0.5)
        yy1=int(y0+u2*(y1-y0)+0.5)
        result=[[xx0,yy0],[xx1,yy1]]
```

3 系统介绍

3.1 采用的系统框架

本系统命令行部分采用了助教所提供的 cg_cli.py 框架，支持所有合法的指令：
重置画布 resetCanvas width height
保存画布 saveCanvas name
设置画笔颜色 setColor R G B
绘制线段 drawLine id x0 y0 x1 y1 algorithm
绘制多边形 drawPolygon id x0 y0 x1 y1 x2 y2 ... algorithm
绘制椭圆 drawEllipse id x0 y0 x1 x1
绘制曲线 drawCurve id x0 y0 x1 y1 x2 y2 ... algorithm
图元平移 translate id dx dy
图元旋转 rotate id x y r
图元缩放 scale id x y s
对线段裁剪 clip id x0 y0 x1 y1 algorithm

本系统图形界面部分采用了助教所提供的 cg_gui.py 框架，以鼠标交互的方式，通过鼠标事件获取所需参数并调用核心算法模块中的算法将图元绘制到屏幕上，或对图元进行编辑，能够实现命令行所对应的所有功能:
保存画布
设置画笔颜色
绘制线段
绘制多边形
绘制椭圆
绘制曲线
图元平移
图元旋转
图元缩放
对线段裁剪
退出程序

3.2 命令行设计思路

3.2.1 指令识别

每次读取一整行指令，由 split 函数分解该行指令，通过每一条指令的第一个单词，区别每一条指令的操作。同时解析剩下的单词，构造出函数调用时的参数，从而正确执行出每条指令对应的操作。

3.2.2 信息存储

对于直线、曲线、多边形、椭圆，将其关键点信息存到以 id 为键值得 hash 表中，每当需要修改图像时即可通过 hash 表快速访问图像信息从而进行修改操作。绘制图像时可遍历 hash 表中的每一个图像，分别调用相应的绘制函数进行绘制即可。

3.3 GUI 界面设计思路

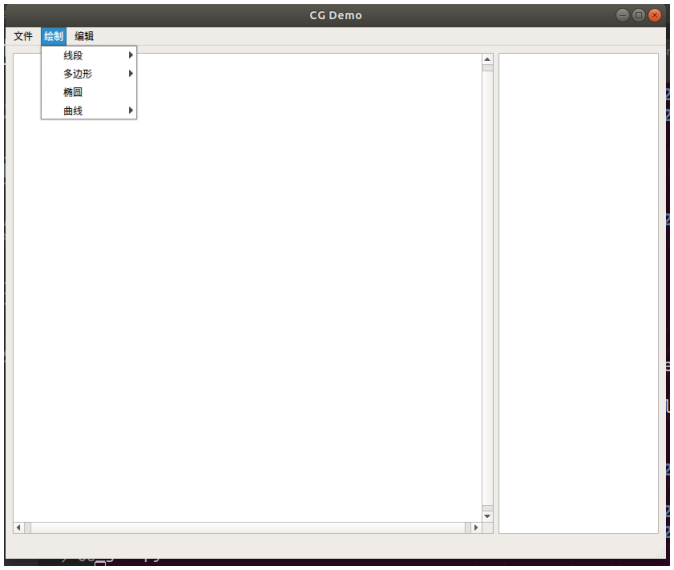

图 6: GUI 界面

3.3.1 直线绘制

先选定绘制直线所用的算法，再在画布中捕获鼠标的起点以及终点，最后调用直线绘制函数，从而在画布中绘制出一条完整的直线。

3.3.2 椭圆绘制

在画布中捕获鼠标的起点以及终点，从而确定椭圆的左上角以及右下角坐标。最后调用椭圆绘制函数，从而在画布中绘制出一个完整的椭圆。

3.3.3 多边形绘制

先选择算法，选点过程参考了 WINDOWS 绘图工具的实现，先同绘制直线的方式一样获取两个初始点，再按下鼠标确定新的点，如果该点的位置距离第一个初始点的距离平方小于 50，则结束多边形选点同时不再保存这一个结束点。最后调用多变形绘制算法进行多边形的绘制。

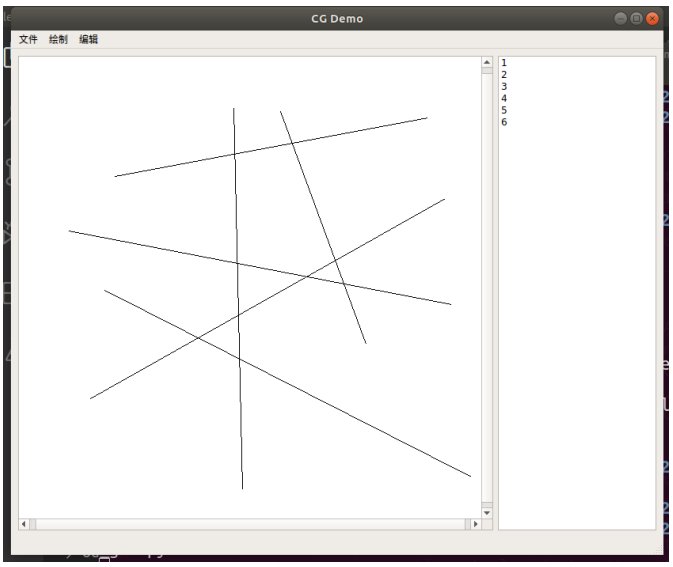

图 7: 直线绘制

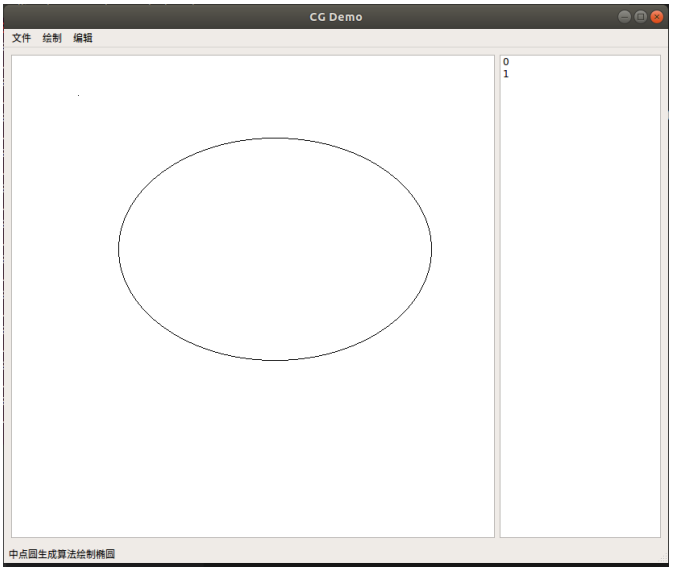

图 8: 椭圆绘制

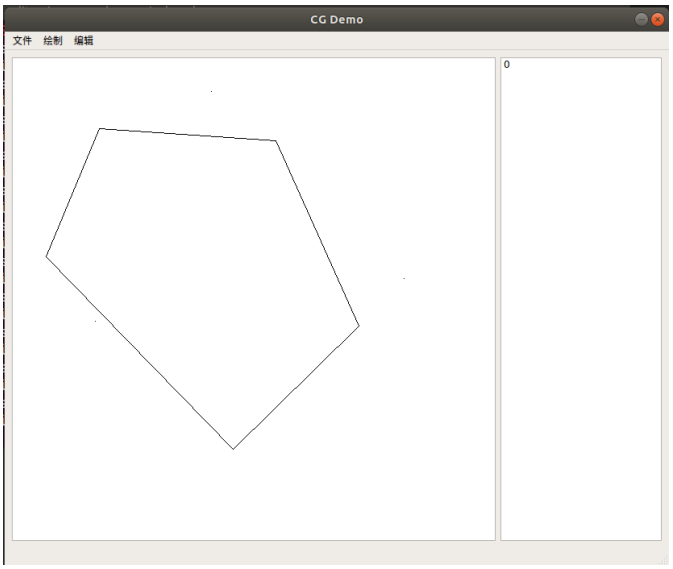

图 9: 多边形绘制

3.3.4 曲线绘制

先选择算法，然后通过对话框输入需要的控制点数，最后通过鼠标在画布中依次选点，然后调用对应的曲线绘制函数即可绘制出曲线。

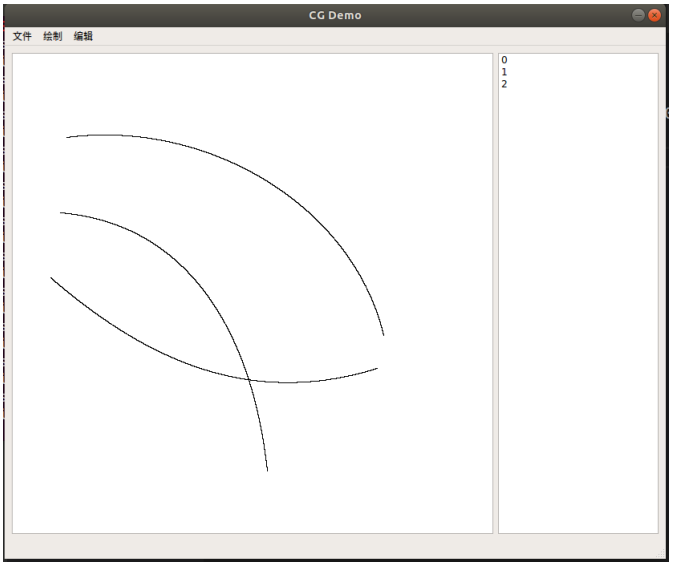

图 10: 曲线绘制

3.3.5 编辑图元

先通过右边控制栏选择图元，然后可以对图元进行平移、旋转、缩放以及裁剪操作，除旋转外都可以直接通过拖拽鼠标进行操作，旋转时需要先通过点击画布确定旋转中心，然后通过鼠标滚轮进行旋转。

3.3.6 重置画布

通过对话框进行画布大小的选择，可以通过输入数字或者拉动滑动条来修改大小，修改后将清空画布中的全部图元。

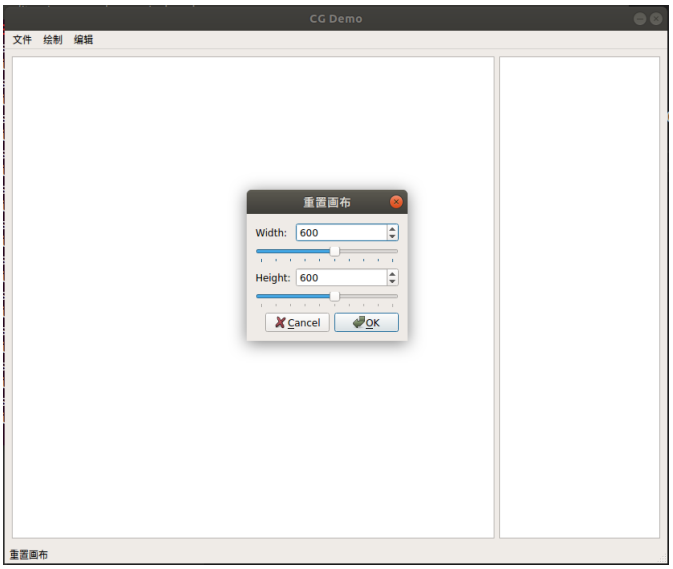

图 11: 重置画布

3.3.7 保存画布

通过对话狂输入目标文件名，调用 qt 中的 grab 操作保存画布并将其输出到对应文件中。

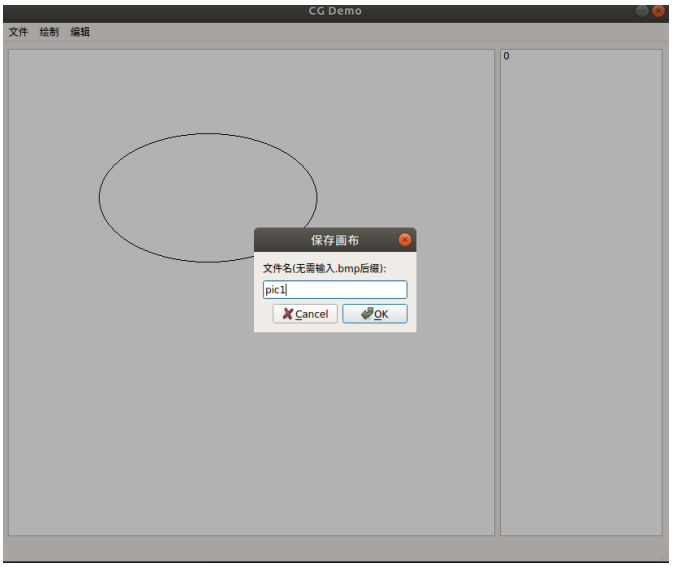

图 12: 保存画布

4 总结

通过这次图形学大作业，不仅让我们实现了一个功能全面的绘图工具，更大大加深了我们对课内所学的图形学算法理解，使我们受益匪浅。

5 引用

[1] 孙正兴, 计算机图形学课程 PPT, 2020