计算机图形学: Experiment
进度报告

June 26, 2021

2021 春季

张思拓　181240078
181240078@smail.nju.edu.cn

目录

1 实验环境

1.1 操作系统

macOS Big Sur 11.2[1]
Windows 10

1.2 开发环境

Anaconda Python 3.8

- numpy 1.19.2
- pillow 8.0.1
- pyqt 5.9.2

2 实验进度

2.1 三月进度

- 在课程实验网站上下载 CG_demo 文件夹，获得实验框架代码，包括核心算法 cg_algorithms.py, 命令行界面 cg_cli.py, 用户交互界面 cg_gui.py. 初步尝试运行三份 python 文件，均能正常执行和退出，说明环境配置正确。
- 完成了 cg_algorithms.py 中 **DDA** 算法和 **Bresenham** 算法的实现。

2.2 四月进度

- 完成了核心算法中椭圆绘制的中点圆算法。
- 阅读 cg_cli.py 框架，实现了命令行界面程序中的指令。

2.3 五月进度

- 完成了核心算法中的平移变换、旋转变换以及缩放变换。
- 实现了 **Cohen-Sutherland** 裁剪算法和 **Liang-Barsky** 裁剪算法。
- 实现了用户交互界面程序 cg_gui.py 中线段、多边形、以及椭圆的绘制。

2.4 六月进度

- 完成了核心算法中的 **Bézier** 和 **B 样条**曲线绘制。
- 学习 PyQt 的使用方法，完成了旋转、缩放、平移，裁剪的 gui 操作，同时添加了重置画布、保存画布的操作。
- 完成了所要求的全部内容。
- 撰写完成进度报告和系统说明书，录制系统演示视频。

[1]在 macOS 系统和 Windows 系统下均进行了测试，其中报告和说明书中的截图在 Windows 下完成；由于 macOS 下录屏较为方便，因此演示视频在 macOS 下完成。两个平台下 PyQt 的窗口展示结果略有不同。

3 算法实现

3.1 线段绘制

3.1.1 DDA 画线算法

算法原理

DDA 画线算法在 x 或 y 方向上以单位增量对线段采样，计算确定另一方向对应整数值，逐步计算路径像素位置。取样方向取决于斜率的绝对值 k 的大小：当 $k \leq 1$ 时在 x 方向取样，当 $k > 1$ 时在 y 方向取样。

在具体实现时为避免分别讨论 x 和 y 方向取样，可以先计算得到步数 len，再根据步数统一确定 x 轴和 y 轴各自的步长 dx, dy，每次同时按各自步长递增即可。当 len 等于 0 时说明起点终点重合，单独讨论。

算法实现

```
x, y = x0, y0
len = abs((x1 - x0) if abs(x1 - x0) > abs(y1 - y0) else (y1 - y0))
if len == 0:
   result.append((x, y))
else:
   dx = (x1 - x0) / len
   dy = (y1 - y0) / len
   for i in range(len + 1):
      result.append((round(x), round(y)))
      x = x + dx
      y = y + dy
```

3.1.2 Bresenham 算法

算法原理

DDA 算法在每次增量操作时需对 x 和 y 增加增量 Δx 和 Δy，而误差会在浮点增量运算的过程中累加，导致偏离线段。

Bresenham 算法采用整数增量运算，在线段离散采样过程中每个点有两个候选像素。当前像素位置为 (x_k, y_k)，则两候选点分别为 (x_{k+1}, y_k) 和 (x_{k+1}, y_{k+1})，Bresenham 算法会根据这两点与真实值的差分，维护一个误差参数 $p_k = 2\Delta y x_k - 2\Delta x y_k + c$。若 $p_k > 0$ 说明 y_{k+1} 比 y_k 更接近线段，因此取较高像素 (x_{k+1}, y_{k+1})，反之取较低像素。决策参数可以增量计算。

具体代码参考 Github 上 Python Bresenham 进行实现，需要不断更新决策参数 p，并根据 p 是否大于零选择是否对 y 方向坐标递增。为便于实现，可以交换坐标轴位置和方向，计算各分量在 x 和 y 轴分别的增量。这样可以使用一套计算公式处理各种情况。

算法实现

```
dx = x1 - x0
dy = y1 - y0
xs = 1 if dx > 0 else -1
ys = 1 if dy > 0 else -1
dx = abs(dx)
dy = abs(dy)

if dx > dy:
   xx, xy, yx, yy = xs, 0, 0, ys # x增量在x方向的变化
else:
   dx, dy = dy, dx # 交换坐标轴
   xx, xy, yx, yy = 0, ys, xs, 0 # y增量在y方向的变化

p = 2 * dy - dx # bresenham算法决策参数
```

```
y = 0 # 距离起点增量

for x in range(dx + 1):
   result.append((x0 + x * xx + y * yx, y0 + x * xy + y * yy))
   if p >= 0:
      y += 1
      p -= 2 * dx
   p += 2 * dy
```

3.1.3 算法对比

相较于 DDA 算法，Bresenham 算法避免了像素在浮点数增量计算时连续叠加所产生的累积误差。同时避免了每次计算下一个像素点时所需的取整等运算的代价，转为从当前点出发，根据误差判断从两个候选像素中选择误差最小的像素点。

从实现结果上来看，Bresenham 算法的绘制效果要优于 DDA 算法，表现上锯齿更少，直线更加平滑。

3.2 多边形绘制

多边形是由定点所组成的线段依次连接所构成的图形，因此可以使用之前实现的画线算法绘制分别每一段线段即可。代码如下：

```
for i in range(len(p_list)):
   line = draw_line([p_list[i - 1], p_list[i]], algorithm)
   result += line
```

3.3 椭圆绘制——中点圆算法

算法原理

中点椭圆算法的思想和 Bresenham 画线算法的基本思想一样，都是每一步从当前点出发，从候选点中挑选出误差最小的像素值。又由于椭圆的对称性，我们只需处理第一象限的点即可，又根据椭圆切线斜率，可以将第一象限的椭圆分为两个部分：区域 1 斜率绝对值小于 1，沿 x 方向步进；区域 2 斜率绝对值大于 1，沿 y 方向步进。

以区域 1 为例，若当前像素为 (x_k, y_k)，那么两候选像素分别为 (x_{k+1}, y_k) 和 (x_{k+1}, y_{k-1})，所谓中点圆算法就是取这两个候选像素的中点，带入椭圆函数求值：

$$p_k = r_y^2(x_k + 1)^2 + r_x^2(y_k - 1/2)^2 - r_x^2 r_y^2$$

若 $p_k < 0$ 表明中点位于椭圆内，说明较高像素 (x_{k+1}, y_k) 离椭圆更近，反之说明较低像素离椭圆更近，选择离椭圆更近的像素。

算法实现

```
x0, y0 = p_list[0]
x1, y1 = p_list[1]
result = []

xc = int((x0 + x1) / 2) # 椭圆中心点横坐标
yc = int((y0 + y1) / 2) # 椭圆中心点纵坐标
rx = int(abs(x1 - x0) / 2)
ry = int(abs(y1 - y0) / 2)

p = ry**2 - rx**2*ry + rx**2 / 4 # 决策参数初始值
x, y = 0, ry
while ry**2*x < rx**2*y:
   result.append((x, y))
```

```
    x = x + 1
    if p < 0:
        p = p + 2*ry**2*x + ry**2
    else:
        y = y - 1
        p = p + 2*ry**2*x - 2*rx**2*y + ry**2
p = (ry*(x + 0.5))**2 + rx**2*(y - 1) - rx**2*ry**2
while y > 0:
    result.append((x, y))
    y = y - 1
    if p < 0:
        x = x + 1
        p = p + 2*ry**2*x - 2*rx**2*y + rx**2
    else:
        p = p - 2*rx**2*y + rx**2
result.append((rx, 0))

mirror = []
for x, y in result:
    mirror.append((-x, y))
    mirror.append((x, -y))
    mirror.append((-x, -y))
tmp = result + mirror
result = [(x+xc, y+yc) for x, y in tmp]
```

3.4 曲线绘制

3.4.1 Bézier 曲线

算法原理

Bézier 曲线是由一组控制顶点构成的多边形控制，勾画形成的曲线。曲线的形状逼近这些控制点。一条 n 次 Bézier 曲线可以被表示为它的 n+1 个控制顶点的加权和，其中权是 Bernstein 基函数。即：

$$P(u) = \sum_{k=0}^{n} P_k \cdot BEZ_{k,n}(u), \quad 0 \le u \le 1$$

我们可以利用 Bernstein 基函数的降阶公式，通过递归计算的方式得到 Bézier 曲线上点的坐标。降阶公式为：

$$BEZ_{k,n}(u) = (1-u)BEZ_{k,n-1}(u) + uBEZ_{k-1,n-1}(u)$$

在实现时使用德卡斯特里奥 (de Casteljau) 算法，先从曲线的控制点出发，根据上面的降阶公式：$P_i^r = (1-u)P_i^{r-1} + uP_{i+1}^{r-1}$，

$$P_i^r = \begin{cases} P_i^0 & r = 0 \\ (1-u)P_i^{r-1} + uP_{i+1}^{r-1} & \text{otherwise} \end{cases}$$

给定一个 u，随着 r 不断增加，对应控制点数递减，直到剩下一个顶点时即得到了所求的点。

算法实现

```
def bezier_point(control_points, t):
    if len(control_points) == 1:
        result, = control_points
        return round(result[0]), round(result[1])
    control_linestring = zip(control_points[:-1], control_points[1:])
    return bezier_point([((1 - t) * p1[0] + t * p2[0], (1 - t) * p1[1] + t * p2[1]) for p1,
        p2 in control_linestring], t)
```

3.4.2 B 样条曲线

算法原理

类似地，B 样条曲线是由 B 样条基函数定义的：给定 n+1 个控制顶点 $\{P_i\}$ 和 n+k+2 个参数节点向量：$U_{n,k} = \{u_i \mid i = 0, 1, ..., n+k+1, u_i \leq u_{i+1}\}$，则 k+1 阶 (k 次)B 样条曲线的数学定义为：

$$P(u) = \sum_{i=0}^{n} P_i B_{i,k+1}(u), u \in [u_k, u_{n+1}]$$

同样的，B 样条基函数也可根据 deBoox-Cox 递推关系定义：

$$B_{i,k+1}(u) = \frac{u - u_i}{u_{i+k} - u_i} B_{i,k}(u) + \frac{u_{i+k+1} - u}{u_{i+k+1} - u_{i+1}} B_{i+1,k}(u)$$

由于实验要求为三次（四阶）均匀 B 样条曲线，因此节点向量为 0,1,2,...,n+3，可以使用上述递推公式计算出曲线定义域：$[u_3, u_{n+1}]$ 内的值。

另外曲线取样的点数：points_num 由控制顶点之间的像素距离确定，这样可以尽可能保证曲线在绘制时不出现断裂。

算法实现

```
def deboox_cox(i, k, u):
    if k == 1:
        if i <= 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)

def draw_curve(p_list, algorithm):
    points_num = 1
    for i in range(len(p_list) - 1):
        points_num += max(abs(p_list[i][0] - p_list[i + 1][0]), abs(p_list[i][1] - p_list[i
            + 1][1]))
    if algorithm == 'Bezier':
        if len(p_list) < 2:
            return p_list
        result = [bezier_point(p_list, i / points_num) for i in range(points_num)]
        return result
    elif algorithm == 'B-spline':
        n = len(p_list)
        if n < 4:
            return p_list
        result = []
        u = 3
        du = (n - 3) / points_num
        while u < n:
            x, y = 0, 0
            for i in range(n):
                k = deboox_cox(i, 4, u)
                x0, y0 = p_list[i]
                x += k * x0
                y += k * y0
            result.append((round(x), round(y)))
            u += du
        return result
```

3.4.3 算法对比

B 样条曲线是 Bézier 曲线的拓广情况。相比于 Bézier 曲线, B 样条曲线对**局部**的控制能力较强，拟合精度更高。但是同时也有着形状构造过程复杂，初等曲线拟合能力差的缺点。

3.5 图形变换

3.5.1 平移变换

算法原理

设图形物体原坐标为 (x, y)，给定水平方向偏移量 $\mathrm{d}x$ 和垂直方向偏移量 $\mathrm{d}y$，那么平移后的位置就是 $(x+\mathrm{d}x, y+\mathrm{d}y)$。因此对图源参数中的所有点进行平移变换即可。

3.5.2 旋转变换

算法原理

设图形物体原坐标为 (x, y)，给定旋转基准点 (x_r, y_r) 及顺时针旋转角度 θ，那么这一点旋转后的位置是 $(x_r+(x-x_r)\cos\theta-(y-y_r)\sin\theta, y_r+(x-x_r)\sin\theta+(y-y_r)\cos\theta)$。在实现时由于画布的原点位于左上角，水平方向为 y 轴，垂直方向为 z 轴，因此此时画布中顺时针旋转对应于坐标系中逆时针旋转，要将公式中的 θ 转为 $-\theta$。

3.5.3 缩放变换

设图形物体原坐标为 (x, y)，给定缩放中心点 (x_f, y_f) 及缩放系数，缩放后的位置为 $(x\cdot s+x_f(1-s), y\cdot s+y_f(1-s))$。

算法实现

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

def rotate(p_list, x, y, r):
    r = math.radians(r)
    result = [(round(x + (u - x) * math.cos(r) - (v - y) * math.sin(r)),
               round(y + (u - x) * math.sin(r) + (v - y) * math.cos(r)))
              for u, v in p_list]
    return result

def scale(p_list, x, y, s):
    result = [(round(u * s + x * (1 - s)), round(v * s + y * (1 - s))) for u, v in p_list]
    return result
```

3.6 图形裁剪算法

3.6.1 Cohen-Sutherland 算法

算法原理

Cohen-Sutherland 算法的思想是对裁剪窗口的不同位置进行编码，根据线段端点的位置编码，得到线段与裁剪窗口的相对位置关系。首先可以使用位置码快速判断出两种情况：

1. 两端点区域码均为 0000，表明线段完全在裁剪窗口内，无需裁剪，直接返回即可。
2. 两端点区域码逻辑与操作, 结果不为 0000，表明线段完全在裁剪窗口外，可以直接丢弃。

剩下的情况不能直接判断，需要按 “左-右-上-下” 顺序与边界进行位置关系的判断，裁剪掉边界之外的部分。直到线段完全在区域外被舍弃，或找到区域内的一段线段。

在实现时为了方便在每次裁剪后进行检查，只需维护一个循环：在每次循环开始时检查是否是可以直接判断的两种情况，并且保存一个 code_out 变量表示当前在裁剪窗口之外的端点，接下来对端点进行裁剪，之后重复这个循环。

算法实现

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

def encode(x, y):
    code = INSIDE
    if x < x_min: # to the left of rectangle
        code |= LEFT
    elif x > x_max: # to the right of rectangle
        code |= RIGHT
    if y < y_min: # below the rectangle
        code |= BOTTOM
    elif y > y_max: # above the rectangle
        code |= TOP
    return code

code0 = encode(x0, y0)
code1 = encode(x1, y1)

while True:
    if code0 == 0 and code1 == 0:
        return [(round(x0), round(y0)), (round(x1), round(y1))]
    elif code0 & code1:
        return []

    if code0:
        code_out = code0
    else:
        code_out = code1

    if code_out & LEFT:
        x = x_min
        y = y0 + (y1 - y0) / (x1 - x0) * (x_min - x0)
    elif code_out & RIGHT:
        x = x_max
        y = y0 + (y1 - y0) / (x1 - x0) * (x_max - x0)
    elif code_out & TOP:
        x = x0 + (x1 - x0) / (y1 - y0) * (y_max - y0)
        y = y_max
    elif code_out & BOTTOM:
        x = x0 + (x1 - x0) / (y1 - y0) * (y_min - y0)
        y = y_min
    if code_out == code0:
        x0, y0 = x, y
        code0 = encode(x0, y0)
    else:
        x1, y1 = x, y
        code1 = encode(x1, y1)
```

3.6.2 Liang-Barsky

算法原理

Liang-Barsky 算法的基本思想是将 2 维的裁剪问题转化为 1 维裁剪问题，通过参数化将二维坐标变成一维的参数坐标。线段所在直线 L 与裁剪窗口的 2 交点为 Q_1Q_2, 称之为诱导窗口。将裁剪窗口的左、右、上、下边界与直线的交点参数化得到 $p_1 - p_4$ 以及 $q_1 - q_4$。

若 $p_k == 0$ 则，说明线段平行于裁剪边界之一，可以快速处理。否则可以计算出边界参数值 $u = q_k/p_k$。具体实现上首先令 $u_1 = 0, u_2 = 1$ 表示未裁剪的线段的参数坐标，再使用每个裁剪边界的参数 u_k 更新线段端点的参数坐标。若更新后 $u_1 > u_2$ 那么表示线段完全在窗口外，直接舍弃，否则根据参数计算出裁剪后端点的坐标。

算法实现

```
dx, dy = x1 - x0, y1 - y0
p1, p2 = -dx, dx
p3, p4 = -dy, dy
q1 = x0 - x_min
q2 = x_max - x0
q3 = y0 - y_min
q4 = y_max - y0

u1, u2 = 0, 1

if ((p1 == 0 and q1 < 0) or (p2 == 0 and q2 < 0) or (p3 == 0 and q3 < 0) or (p4 == 0 and q4
    < 0)):
    return []
if p1 != 0:
    r1, r2 = q1 / p1, q2 / p2
    if p1 < 0:
        u1 = max(u1, r1)
        u2 = min(u2, r2)
    else:
        u1 = max(u1, r2)
        u2 = min(u2, r1)
if p3 != 0:
    r3, r4 = q3 / p3, q4 / p4
    if p3 < 0:
        u1 = max(u1, r3)
        u2 = min(u2, r4)
    else:
        u1 = max(u1, r4)
        u2 = min(u2, r3)

if u1 > u2:
    return []
x_0 = round(x0 + u1 * dx)
y_0 = round(y0 + u1 * dy)
x_1 = round(x0 + u2 * dx)
y_1 = round(y0 + u2 * dy)
return [(x_0, y_0), (x_1, y_1)]
```

3.6.3 算法对比

Liang-Barsky 算法与 Cohen-Sutherland 算法相比：

1. 避免了重复迭代裁剪、测试的求交次数，仅需要一次计算就可以得到 u_1, u_2 的最终值。算法运行效率高。

2. Liang-Barsky 算法可以很容易地直接拓展为三位裁剪算法，而 Cohen-Sutherland 则要引入更多的区域码和求交测试才能实现三维裁剪。

4 系统介绍

4.1 系统框架

本实验实现的绘图系统由三部分组成:

- 核心图形学算法模块: cg_algorithm.py
- 命令行界面程序: cg_cli.py
- 用户交互界面程序: cg_gui.py

在本进度报告第3小节已经重点介绍过了图形学核心算法的原理及实现，接下来主要介绍 CLI 程序和 GUI 程序的实现思路和内容。

4.2 命令行界面 CLI

CLI 程序接受两个外部参数: 指令文件的路径和图像保存目录。程序读取包含了图元绘制指令序列的文本文件，依据指令调用核心算法模块中的算法绘制图形以及保存图像。

程序中存在一个主循环: 不断读取一行指令序列，使用 split 函数分解指令，根据不同的指令命令进行相应的处理。

主要可分为绘制命令、编辑命令和画布命令三种指令。

4.2.1 绘制命令

对于绘制命令，会得到待绘制图元的 id，控制点和要使用的绘制算法。对于不定个数控制点的绘制任务，则要特别将控制点使用 for 循环取出。需要注意的是，执行完绘制命令，图元并没有真正的调用图形学算法、绘制到画布上，而是先将所有的图元绘制参数存入 item_dict 字典中。等到执行画布保存命令时再统一从字典中取出绘制。

4.2.2 编辑命令

编译命令主要执行图元的平移、旋转、缩放、裁剪等变换。主要流程:

- 读入图元 id 和变换参数。
- 从 item_dict 中读取图元参数 p_list，传入相应的图形学算法进行编辑。
- 变换的返回值是新的 p_list, 以此替代图元原有的 p_list。

4.2.3 画布命令

画布相关命令可以改变画布的参数（长、宽、画笔颜色），当新建画布时会将原有的图元字典清空，保存画布时统一从字典中取出图元，调用相关图形学算法进行绘制。

4.3 用户交互界面 GUI

通过使用 PyQt 提供的槽函数机制，将主窗口类 MainWindow 中菜单栏的选项与画布窗体类 MyCanvas 建立连接，从而能向画布中添加图元类 MyItem，并调用相应的 paint 函数进行绘制。

4.3.1 画布窗体类

画布窗体类 MyCanvas 继承自 QGraphicsView，采用 QGraphicsView、QGraphicsScene、QGraphicsItem 的绘图框架。在 MyCanvas 类中主要通过定义鼠标控制事件，捕捉用户的鼠标点击以及释放的坐标，建立图元的信息：

- mousePressEvent：鼠标点击事件，主要用于创建新的图元、添加图元中的控制点（p_list）或者结束多控制点图元的定义（例如：多边形绘制和曲线绘制）
- mouseMoveEvent：鼠标移动事件，主要用于调整定义的控制点的位置
- mouseReleaseEvent：鼠标释放事件：对于只有 2 个控制点的图元或编辑操作，会结束操作，对操作结果进行保存、更新；对于多控制点图元一般不设响应。

4.3.2 自定义图元类

自定义图元类 MyItem 继承自 QGraphicsItem。可以通过 MyCanvas 类创建 MyItem 元素，创建函数保存了图元的各种信息，包括：图元 ID、图元类型、图元参数、图元颜色和绘制算法。

在绘制图元的过程中，通过调用图形学算法函数，获得绘制结果的像素点坐标列表，最终将获得的像素坐标绘制到画布上即可得到最终结果。

同时提供了一个 boundingRect() 函数，用以绘制每一个图元的边框，便于在选择图元时获得更直观的效果。

4.3.3 主窗口类

主窗口类 MainWindow 继承自 QMainWindow。使用 QListWidget 来记录已有的图元，和 QGraphicsView 作为画布。同时将菜单中的每一个操作定义的连接信号通过槽函数调用执行 MyCanvas 类中的图元操作函数。

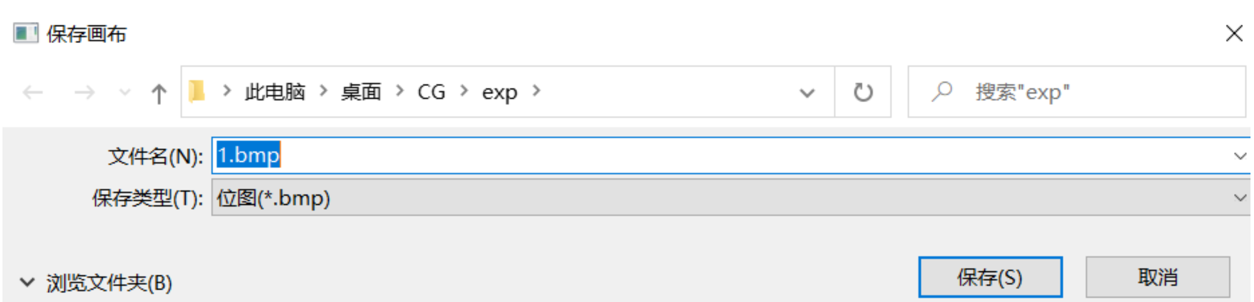
我在菜单中增加了一个保存画布的菜单，类似于 cg_cli.py 中的 saveCanvas 指令，可以将当前的画布保存到用户定义的 bmp 文件中。

5 总结

我顺利完成了图形学实验所要求的全部任务，实现了所有要求的基本功能，并具有以下特色：

5.1 额外的功能

可以保存当前的画布，参照 CLI 中保存画布的功能。



5.2 易用的交互

- 对于两个控制点的图元，一个拖拽就能绘制完成；多个控制点的图元，可以通过单击鼠标右键完成绘制。

- 连续绘制：当选择菜单栏中某种绘制方法后，可以连续在画布上绘制同类型的图元。
- 平移：实现了拖拽的交互式操作方法，并且为了契合拖拽的显示感受，只有当鼠标位于图元边框内时，点击鼠标才可以拖拽。
- 旋转：鼠标点击时确定旋转中心，通过鼠标移动控制旋转角；
- 裁剪：特别设置了裁剪框，使得用户可以直接看到**裁剪窗口的范围**；并且当图元完全被裁剪掉时，图元的 ID 会自动从边栏中**去除**掉。

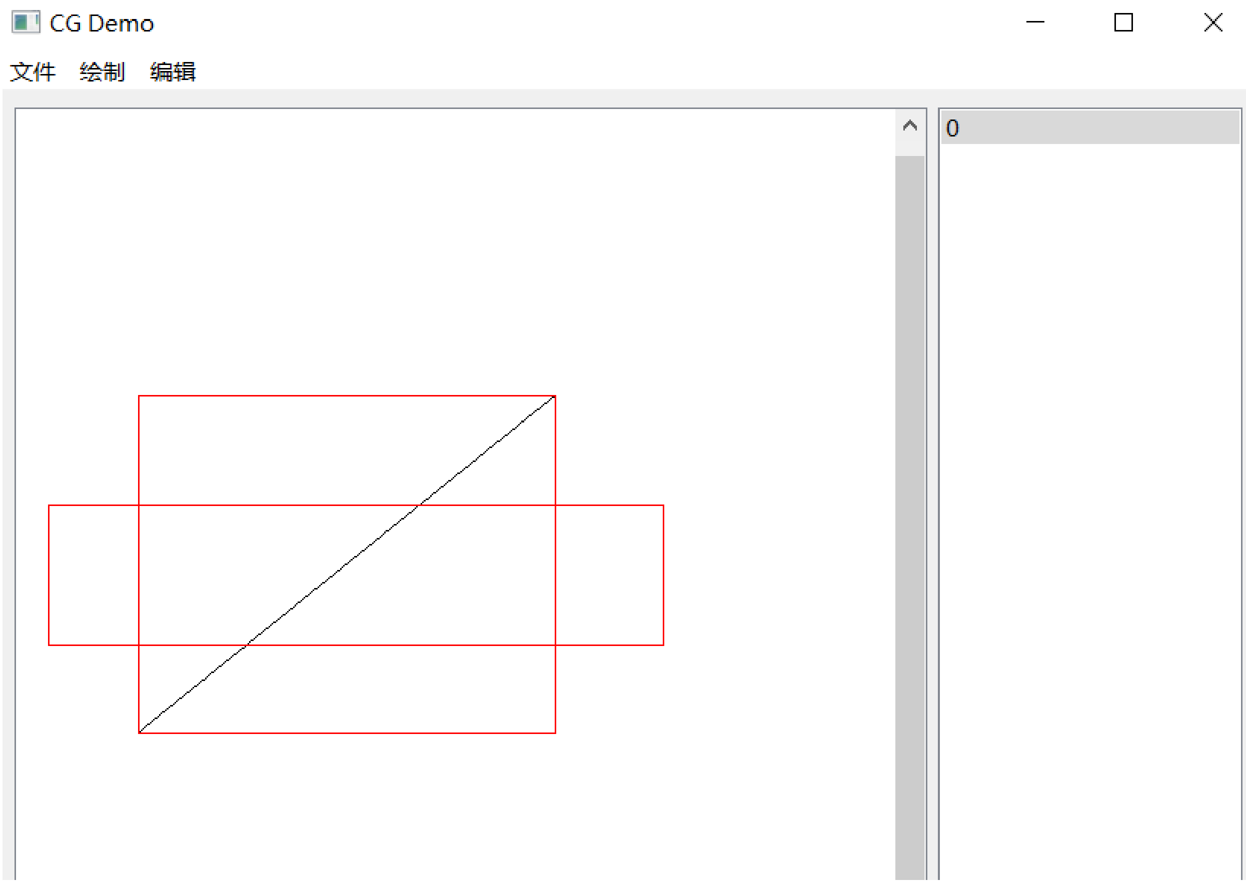


图 1：裁剪前

这里的横向较长的红框就是裁剪窗口。裁剪结果如下图所示：

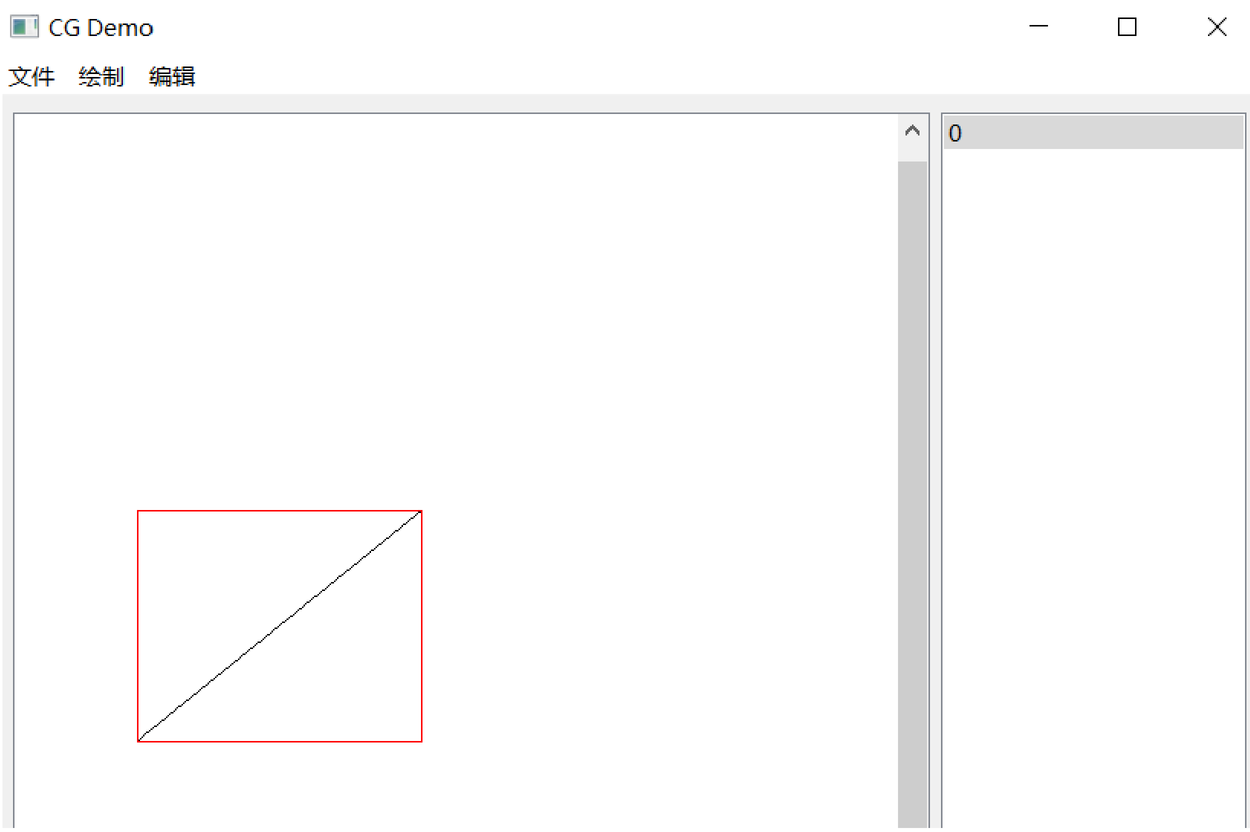
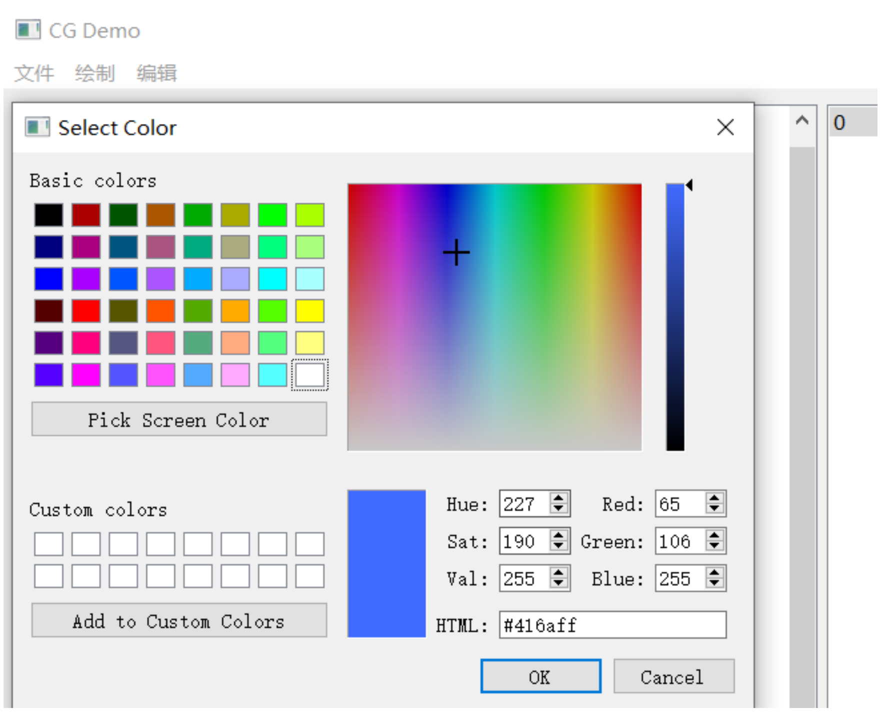
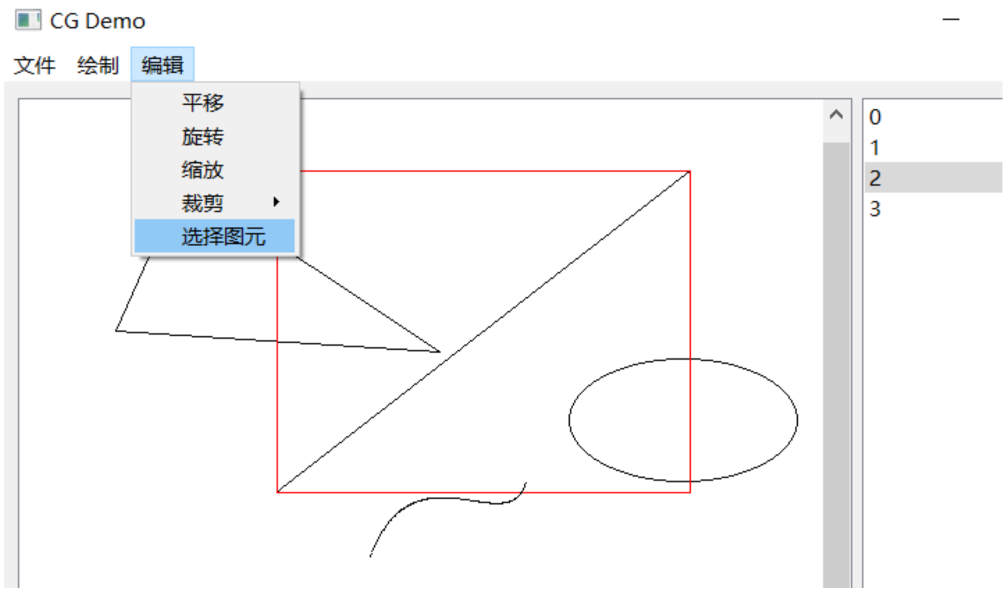


图 2: 裁剪后

- 画笔颜色选择使用了 QColorDialog 中的 getColor 控件，用户可以直接从调色板中选择所需的颜色：



- 我实现了使用鼠标**选择图元**的功能：在菜单栏编辑菜单下选择 “选择图元”，接下来直接在画布中用鼠标点击图元，即可选中该图元（当有多个图元位置重叠时，默认选择最上层的图元）。



5.3 优雅的代码

在本次实验中由于要对点绘制点进行处理，很多地方需要用到 for 循环来生成或处理 list 列表，我在绝大多数场景都使用了列表**生成式**的代码编写方式，使得代码风格简洁优雅。

例如图元的平移操作只需要一行代码就可以得到处理结果：

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

以及获取不定长参数列表：

```
x_list = [int(x) for x in line[2:-1:2]]
y_list = [int(y) for y in line[3:-1:2]]
p_list = list(zip(x_list, y_list))
```

6 参考资料

1. A simple implementation of Bresenham's line drawing algorithm：https://github.com/encukou/bresenham

2. 2021《计算机图形学》课件