

《计算机图形学》系统报告

学号：181860077 姓名：余帅杰，3121416933@qq.com

2020 年 12 月 21 日

1 综述

1. 熟悉实验框架和所需的 PyQt5 知识
2. 实现了所有的绘图算法，包括直线绘制，多边形绘制，椭圆和曲线绘制
3. 实现了所有的图元操作算法，包括旋转，平移，缩放，裁剪
4. 在 GUI 里对各个事件进行处理，使用核心算法模块中的算法绘制图元或者是对图元进行操作
5. 实现了保存图元，设置画笔，重置画布等 GUI 标配功能
6. 实现了使用鼠标选中图元，删除，复制，粘贴，快捷键等等的附加功能

2 实验框架阅读和理解

在实现之前自然要对拿到手的框架代码进行阅读和分析，可以显然的看到代码主要都分布在三个文件中，分别进行阅读和分析

2.1 cg_algorithm.py

这个文件主要就是实现绘图代码的算法，每一个函数基本都代表这一个需要实现的功能或者是算法，每一个函数接受绘图所必须的点，一般都是以列表的形式给出，其他的参数一般就是一些可选参数（如：想要实现的后台算法种类）这一块结构比较简单，也是实现的核心和重点

2.2 cg_cli.py

cli 文件主要是负责解析命令行文件，并做出回应。基本的功能就是按行读取文件然后利用 split 分解命令行，并根据预设的指令的格式进行匹配并执行

整体的主要代码逻辑如下

1. 收到的是绘图的指令，那就使用特定的格式，把绘图的要求以及颜色等信息都保存在字典里，等待后续的使用。

2. 收到的是 saveCanvas 指令，就会遍历先前提到的字典，对每一项进行解析，利用存储的信息调用对应的绘图算法得到像素点的坐标列表，根据坐标点的位置信息在画布数组上填入颜色信息（坐标信息实际上就是画布数组的下标信息）
3. 收到的是 resetCanvas 指令，相对就会比较简单，就只需要重新设置画布的规格（高，宽）然后清空刚才提到的对象列表

2.3 cg_GUI.py

本模块就是对于之前的一些功能的集成，并对外提供图形化的界面进行使用。基本的模块由三块构成

1. MyItem 继承于 QGraphicsItem，本质上就是用来描述图元的类，保存了图元相关的信息，可以类比到 Cli 文件里的那个字典里的元素，只不过这里使用了一个专门的类来描述，同时由于有了图形化界面，所以对图元的描述又多了一个选中的标志位。以及图元的共性操作：上色显示。还有的就是一些不同图元的特色操作，如：选中框显示。
2. MyCanvas 继承于 QGraphicsView，类比起来就是电视机的显示屏幕，负责图元的管理和显示，简单的可以类比做 Cli 里的那个字典保管着若干图元。但是对于 GUI 而言还有鼠标等事件的 Handle，所以对应的这个模块负责了鼠标在画布上的操作的处理，以及对应的对图元进行处理，如绘制，选中等等。刚好前文说到，他保管着所有图元的信息字典。
3. MainWindow 继承于 QMainWindow，负责的区域是除了画布以外的窗口区域，也就是菜单等所以他负责构造整个菜单的结构以及按钮，还有就是绑定菜单信号和对应处理信号的槽函数，但是他是负责菜单，不直接的对画布和图元操作，因此二者时间的通信实际上是通过设置 State 来做到的。对应的 MyCanvas 在处理鼠标事件的时候也会根据状态的不同做出改变。（比如说，菜单选中了”选择图元”，则 MainWindow 把状态设置为”Selecting”，这个时候再用鼠标点击画布等操作，MyCanvas 就会根据 Selecting 这个状态对对应的鼠标事件进行处理。）

3 算法介绍

3.1 直线绘制算法

3.1.1 Naive

算法原理 Naive 算法相对简单，算法的基本思路如下：

- (1) 首先排除可能的特殊情况，即横坐标为 0 斜率不存在情况，然后直接计算对应的 (x,y) 坐标组合
- (2) 没有触发特殊情况则计算斜率，斜率即 X 轴方向增大的时候 Y 对应的增量
- (3) 然后就横坐标移动，纵坐标做对应的变化

算法评估 对 Naive 算法性质简单评估

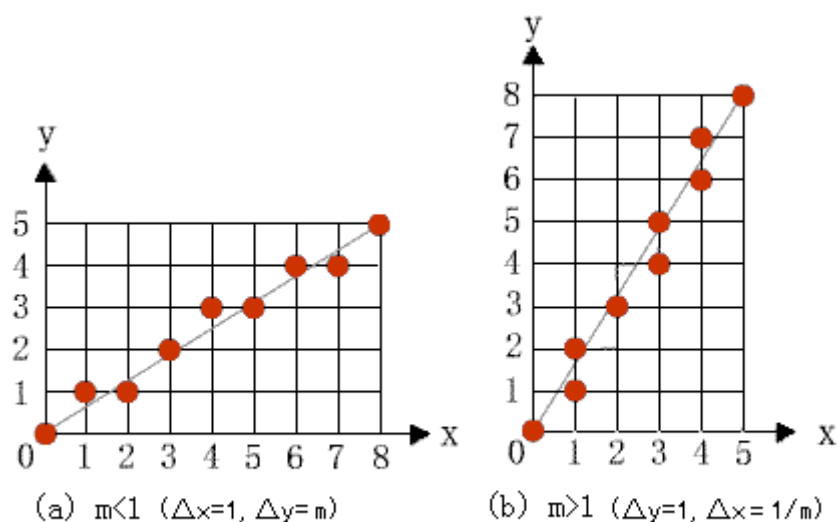
1. 优点：简单易懂便于理解，符合数学公式
2. 缺点：算法每生成一个点都需要做一次乘法和若干次的加法和减法，复杂度较高

3.1.2 DDA

算法原理 DDA 算法

DDA 算法，也就是数字差分分析方法，利用计算两个坐标方向的差分来确定线段显示的屏幕像素位置的线段扫描转化算法，也就是说对一个坐标轴上以单位时间间隔对线段采样 ($\Delta x = 1$ 或 $\Delta y = 1$)，计算 Δy 或 Δx 决定另一个坐标轴上最靠近线段的路径的对应整数值 [1]

DDA 算法实际上也可以看作是在沿着某一条坐标轴做扫描，只不过基于斜率的不同导致了扫描的坐标轴不同原因：如果都是按照 Naive 那样沿着 X 轴进行扫描进而得到所谓的 Y 轴的位置，那么对于斜率大的直线而言，由于斜率大导致 Y 轴步长远大于 X 轴的步长，进而导致了线上的点非常的稀疏，所以 DDA 采用的是根据斜率的值来动态的选择沿着哪一条轴扫描。需要注意的是，由于沿着 Y 轴扫描的时候，X 轴上的增量就不再是斜率，而是斜率的倒数



亮度均匀的考虑

图 1: DDA 算法示意图

如图1, 有如下公式

$$\begin{cases} y_{k+1} = y_k + m & m \leq 1 \\ x_{k+1} = x_k + \frac{1}{m} & m > 1 \end{cases}$$

算法的测试如下：在 GUI 中沿着各种方向画线并观察

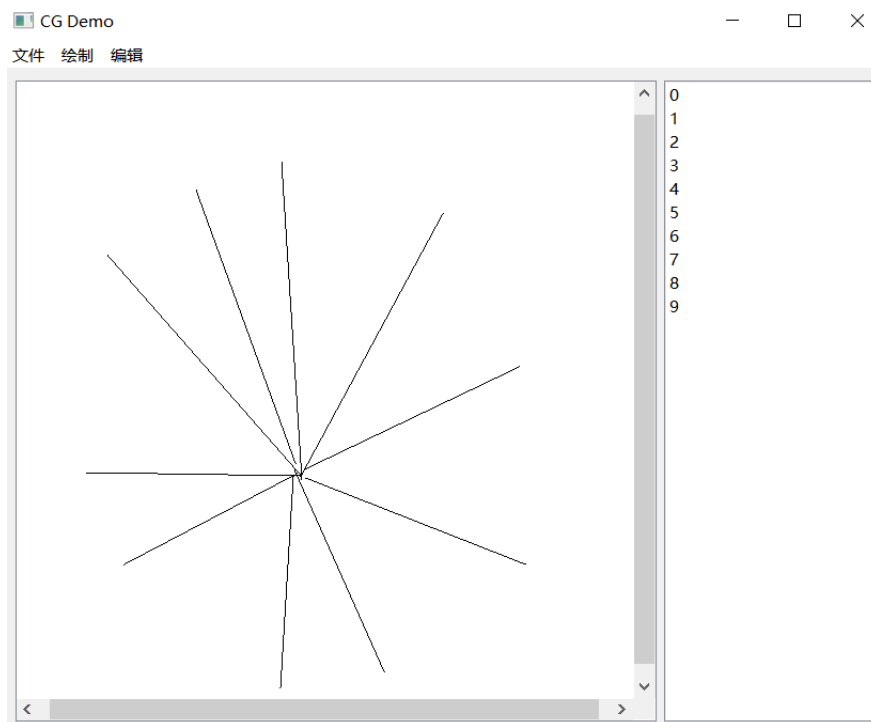


图 2: DDA_GUI 测试示意图

算法评估 基于上述的原理对 DDA 算法性质简单评估

1. 优点: DDA 算法的计算是增量计算 (充分的利用了光栅的特性), 每一次计算都是充分的利用了上一次的计算结果, 所以 DDA 算法的复杂度小于之前的 Naive 算法 (基于直线方程的)
2. 缺点: 浮点数操作会导致误差的积累, 使得结果偏离路线
3. 缺点: 取整和浮点数操作仍然耗时, 可以分解增量 $\frac{1}{m}$ 和 m 为整数和小数, 使得计算为整数间的计算而提高效率

3.1.3 Bresenham

算法原理 Bresenham 算法

Bresenham 算法是一种精准而有高效的光栅线段生成算法, 他可用于圆和其他曲线显示的整数增量运算, 简单的说就是仍然是选定了一个扫描的方向, 然后沿着方向进行扫描, 但是在扫描的过程中对另一个坐标的像素进行选择

为了简化像素的选择, Bresenham 算法通过引入整形变量去衡量候选像素和实际像素的偏移并利用对整形变量符号的检测来决定最接近真实路径的像素。

真实位置 $y = mx_{k+1} + b = m(x_k + 1) + b$

候选像素也就是 y_{k+1} 和 y_k

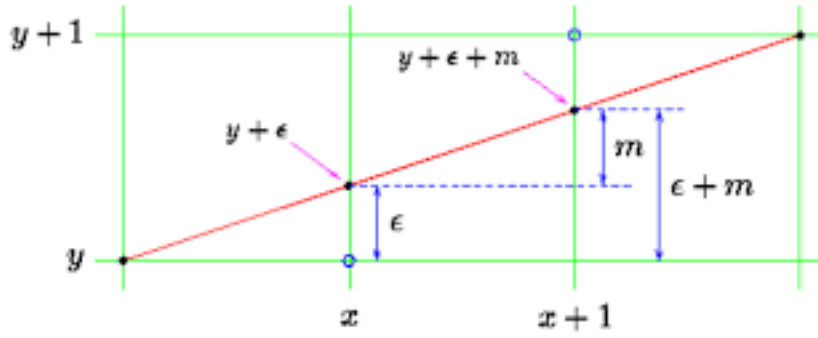


图 3: Bresenham 算法示意图

那么就计算二者的误差（距离差）得到如下的公式

$$\begin{cases} d1 = y - y_k = m(x_k + 1) + b - y_k \\ d2 = y_{k+1} - y = y_{k+1} - m(x_k + 1) + b \end{cases}$$

由于需要比较二者的大小，于是很自然的对二者进行做差即如下公式

$$d1 - d2 = 2m(x_k + 1) - 2y_k + 2b - 1$$

基于上述的公式得到本算法的核心：决策变量

$$p_k = \Delta x(d1 - d2) = 2\Delta y x_k - 2\Delta x y_k + c$$

本公式的核心还是在于两个候选像素的误差大小比较（作差）。前面乘上的 Δx 主要还是为了优化计算为整数之间的计算而提高效率。同时 $c = 2\Delta y + \Delta x(2b - 1)$ 是一个常量，并会在增量计算中被忽略很显然在绘图的时候应该选择误差小的，也就是 p_k 的正负性可以很清楚的作为像素决策的依据

目前问题就被转化为如何去计算每一步的 p_k 作选择的依据

$$p_{k+1} - p_k = 2\Delta y x_{k+1}(x_{k+1} - x_k) - 2\Delta x(y_{k+1} - y_k)$$

经过化简之后就可以得到

$$p_{k+1} = p_k + 2\Delta y - 2\Delta x(y_{k+1} - y_k)$$

$$\begin{cases} y_{k+1} - y_k = 1 & p_k > 0 \\ y_{k+1} - y_k = 0 & p_k < 0 \end{cases}$$

个人的理解就是如3中所示，就是误差的积累当误差达到一定的大小的时候就只能够去选择下一步的 y_k ，也就是在选择之后需要重新的计算误差，类似于加法加到一定大小之后取模的操作

经过上述的分析之后，很容易就可以得到此算法的工作流程

- (1) 计算常量值 $\Delta x, \Delta y, 2\Delta y, 2\Delta y - 2\Delta x$
- (2) 循环扫描，对 p_k 的值进行判断

(3) 若 p_k 的值为正，则 y 加 1，同时 $p_k = 2\Delta y - 2\Delta x$

(3) 反之若 p_k 的值为负，则 y 不变，同时 $p_k = 2\Delta y$

在 GUI 中测试：

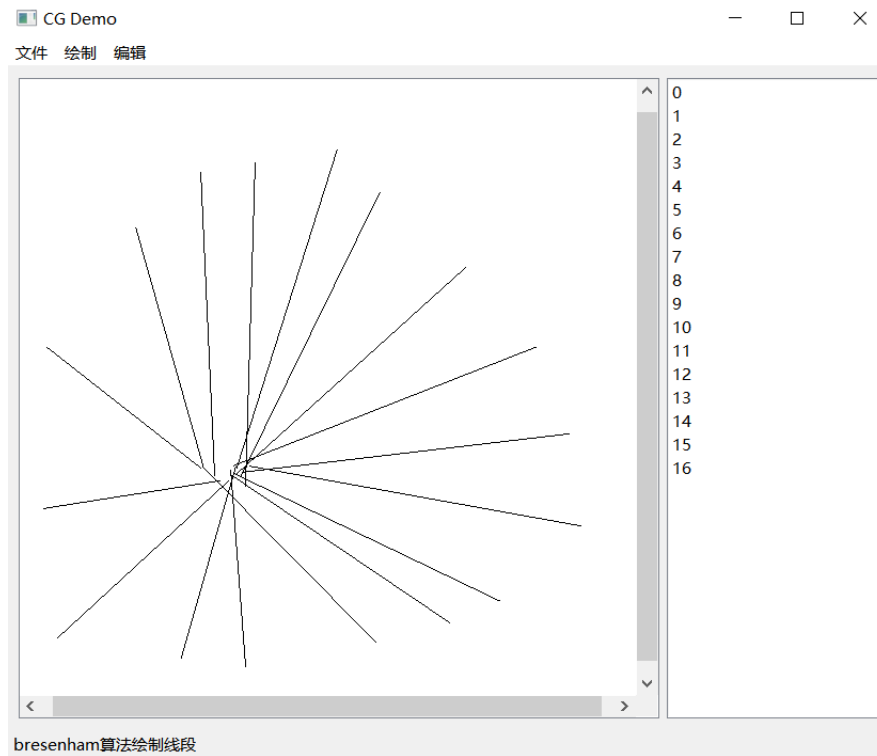


图 4: Bre_GUI 测试示意图

算法评估 基于上述的原理对 Bresenham 算法性质简单评估

1. 优点：计算量进一步的减小，每一次的计算都只发生的整数之间，并且都是复杂度相对较低的加减法计算，同时只需要判断决策参数的符号，算法效率得到提高
2. 优点：可以利用并行计算加快图像生成速度
3. 缺点：稍微复杂，直接看公式可能不能很好的理解

上述的算法大多都经过了在 GUI 环境下的测试，于是选择在命令行环境下进行统一测试，分析结果。整合测试：对上述的三个直线生成代码进行测试，为了方便直接在 cli 中使用文件测试：

测试语句如下

```
resetCanvas 600 500  
setColor 0 0 255
```

```

drawLine line1 195 363 316 50 Naive
setColor 0 255 0
drawLine line3 190 343 311 30 DDA
setColor 255 0 0
drawLine line2 185 323 305 10 bresenham
saveCanvas 3

```

测试结果如下

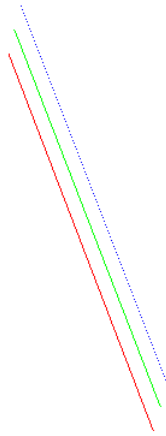


图 5: 直线绘制算法测试示意图

3.2 多边形绘制算法

对于多边形的绘制，本质上就是绘制多条直线，所需要做的就是顶点之间绘制直线，进而形成图形的边界。所以本身并不涉及过多的算法实现，只需要在命令行和 GUI 功能中加上代码，链接上对应的功能实现就可以分别对 Cli 和 GUI 文件进行更改

测试输入：

```

resetCanvas 600 500
setColor 0 0 255
drawPolygon line5 215 353 365 353 365 453 215 453 bresenham
drawPolygon line6 235 373 385 373 385 473 235 473 DDA
saveCanvas 3

```

测试结果如下



图 6: 多边形绘制算法测试示意图

3.3 中点圆绘制算法

算法原理 中点圆算法

首先，椭圆（或者说圆）有若干的性质：

- 高度的对称性，也就是只需要生成一部分就可以得到整体
- 斜率不断的动态变化，斜率不唯一

中点圆算法的原理实际上和 Bresenham 算法的原理相似，都是通过某一个方向的扫面，在另一个方向上的候选里去评估错误率，然后进一步的选择哪一个像素点不同之处就在于所谓的椭圆方程。针对先前已经实现的算法不难发现，这种针对一个方向扫描的算法的问题就在于用哪个方向，最后的答案往往是根据斜率来决定，同样的在椭圆这里，由上述的性质知道椭圆绘制需要分段

进而计算得到了分段点：

$$2r_y^2 \geq 2r_x^2 y$$

即这里作为分界分段求解同时对于段内，需要同样的要引入一个决策变量

$$p_k = F(x_{k+1}, y - \frac{1}{2}) = (x_k + 1)^2 + (y_k - \frac{1}{2})^2 - r^2$$

同理的利用决策变量的符号来决定另一个坐标轴的移动，这个变量表示着候选像素中点的是否在圆内，也就是标识了所谓的候选点的质量。推导得到了这个决策变量的增量计算公式如下：

区域一：

$$\begin{cases} p_{k+1} = p_k + 2r_y^2 x_k + 3r_y^2 & p < 0 \\ p_{k+1} = p_k + 2r_y^2 x_k + 3r_y^2 - 2r_x^2 y_k + 2r_x^2 & p \geq 0 \end{cases}$$

区域二：

$$\begin{cases} p_{k+1} = p_k - 2r_x^2 y_k + 3r_x^2 & p \leq 0 \\ p_{k+1} = p_k + 2r_y^2 x_k + 3r_x^2 - 2r_x^2 y_k + 2r_y^2 & p > 0 \end{cases}$$

并根据公式计算决策变量的初始值

$$p_0 = r_y^2 - r_x^2 r_y + \frac{r_x^2}{4}$$

算法的整体流程即如下

- (1) 计算决策变量的初始值，并选择椭圆最上方的点
- (2) 沿着 X 轴循环扫描，根据公式更新决策变量和 y 值，直到结束区域一
- (3) 记忆区域一的最后一个位置，根据公式计算新的决策变量初始值
- (4) 沿着 Y 轴循环扫描，根据对应公式计算决策变量和 x 值，直到结束区域二
- (5) 由先前得到的第一象限的全部图像做图形变化，得到完全椭圆
- (6) 根据真实图像所在中心，平移现图像至真实位置结束算法

测试输入：

```
resetCanvas 600 500
setColor 0 0 255
drawEllipse line7 0 0 200 200
drawEllipse line8 100 100 300 300
saveCanvas 3
```

测试结果如下 (见图7)

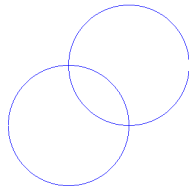


图 7: 中点圆绘制算法测试示意图

3.4 算法正确性检查模块

3.4.1 图片读取

基本步骤如下（详细可见代码:Score_Test.py）

- (1) 利用 os 读取到对应图片目录下的所有的文件名字
- (2) 拼接得到最终的路径名
- (3) 使用 PIL 库对图片进行读取 RGB
- (4) 使用 Numpy 库对数据统一转化为 Array

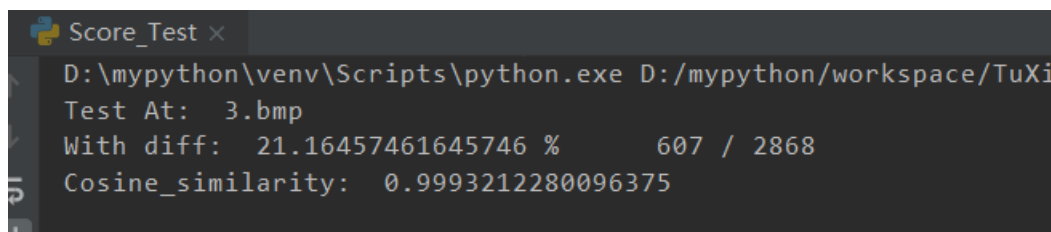
3.4.2 校验算法

使用多种方法对多种算法实现的最后结果进行对比和分析

（注：仅在本文件为了实现的方便调用了 sklearn 用于计算余弦相似度。整个工程的其他位置的代码没有使用任何未经允许的库，如果这也不合规则也可删除 sklearn 的调用）

- 像素级别对比和差异统计：使用 numpy 特性扫描图片，计算有效区域（即作图区域，可以避免由于作图部分过于稀疏导致的结果差异）以及差异区域，计算差异度
- 图片整体的相似度计算，利用 RGB 值对整个图片建模表示，然后计算余弦相似度以统计计算图片之间的差异，设置一定的阈值进行警示可能的作图错误

如下为一次测试的样例



```
Score_Test x
D:\mypython\venv\Scripts\python.exe D:/mypython/workspace/TuXi
Test At: 3.bmp
With diff: 21.16457461645746 %      607 / 2868
Cosine_similarity: 0.9993212280096375
```

图 8: Score 校验算法

如图8, 差异度的计算方式为差异度 = 差异区域像素点数目/着色像素点数目
同时计算两个图片向量的余弦相似度作为参考
(简单的经验: 99.9+% 代表良好, 低于 99.9% 则可能存在算法绘图错误)

3.5 GUI 功能加强

在 GUI 方面, 为了方便使用加入了使用鼠标直接选中图元的操作由原框架代码可以知道, 选中图元可以通过界面右侧的列表, 根据编号选中图元, 即选中图元的基本操作已经被实现, 但是由于图元和编号并不显式绑定, 所以考虑使用鼠标更加便捷直观的选中图元,

对应的就是需要在鼠标按下和释放的时候，根据当前的状态判断是不是在选中图元的状态。如果是则根据当前的鼠标位置计算对应选中的图元，然后执行切换选中的过程，后半部分可以通过调用原框架代码的时候达到代码统一化简洁化的目的

基本的实现流程如下

- (1) 利用菜单键进入”Selecting” 状态
- (2) 在画布界面中点击所需的图元
- (3) 鼠标的点击和释放触发信号，并发现当前处于”Selecting” 状态，计算得到当前的图元
- (4) 如果先前选中图元，则清除先前的选中
- (4) 设置当前的选中状态，刷新画布状态，此时后续的处理则会使得图元被红色矩阵包围，显示为选中状态

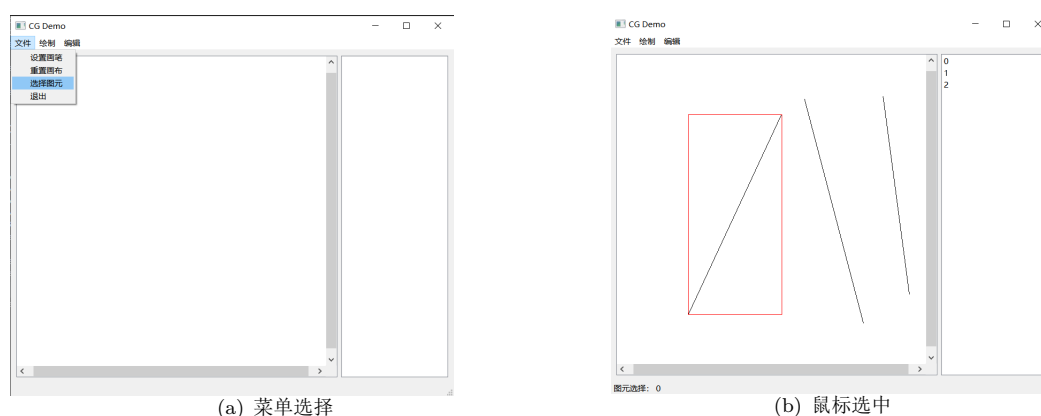


图 9: 鼠标选中图元测试实例

4 系统介绍

目前的 CG2020 图形学系统已经支持

- 命令行和 GUI 界面调用直线生成算法
- 命令行调用多边形生成算法和中点圆生成算法
- 在 GUI 界面中使用鼠标直接的选中所需图元

5 总结

本次实验基本还是以摸清框架代码结构以及了解 QT 特性为主，实现的代码量并不大主要是实现了相对简单明了的直线生成算法以及相对复杂的多边形和中点圆生成算法，同时实现了图片检验模块，为后续的复杂图像生成的正确性检验做一些准备，最后实现的鼠标选中图元的操作，即是方便操作同时也是方便于后续复杂图元生成的时候测试和调试工作

参考文献

- [1] 《计算机图形学教程》 孙正兴编
- [2] [Google 图片](#)