CG 实验报告

胡俊豪 181240020@smail.nju.edu.cn

1 准备工作

1.1 Ubuntu 18.04 虚拟机

我们在 VirtualBox 上安装了 Ubuntu 18.04，64 位系统，50GB 内存，4 个核心。

1.2 Anaconda

我们在安装好的 Ubuntu18.04 上安装了 Anaconda3，并创建了名为 graphics 的虚拟环境，使用的 python 版本和各种包的版本与实验要求一致，分别为：python 3.7.4，numpy 1.18.1，pillow 7.0.0，pyqt 5.9.2。

我们选择的 IDE 是 Anaconda 自带的 Spyder IDE。

1.3 初试代码

我们将实验网站上的示例代码下载下来，放入自己搭建好的环境，并成功运行。在 gui 和 cli 两个界面下进行了探索，对整个实验的要求和最后的结果有了一个更加深刻的理解。

2 cg_algorithm.py

在这个章节，我们将对 cg_algorithm.py 文件中涉及到的每一个函数，所用到的每一种算法的原理进行分析和对比。

2.1 draw_line

2.1.1 DDA

DDA 算法又称数值差分分析，它在一个坐标轴上以单位增量对线段离散取样（$\Delta x = 1$ 或 $\Delta y = 1$），确定在另一个坐标轴上最靠近线段路径的对应整数值。取样方向决定于直线斜率绝对值的大小：斜率绝对值小于 1 时，在 x 方向取样，计算 y 方向的位置坐标；斜率

绝对值大于 1 时，在 y 方向取样，计算 x 方向的位置坐标。增量取决于线段生成方向与坐标轴方向的关系：二者相同取 $+1$，相反取 -1。

DDA 算法比直接使用直线方程快，但是求解每个像素点位置的时候都必须进行取整操作和浮点运算，这会导致误差的累计，以及消耗大量运算时间。

要优化 DDA，可以将增量乘以一个常数，把所有运算变成整数操作。

2.1.2 Bresenham

Bresenham 提出一种算法。根据光栅扫描原理，线段离散取样过程中的每一个放置位置上只可能有两个像素接近于线路经。考虑两候选像素于实际路劲间的偏移关系，选择更近的一个像素作为线的一个离散点。

需要注意的是，水平线，垂直线以及对角线可以直接装入帧缓冲器，无需进行画线算法处理。

Bresenham 算法相比于 DDA 的优点是：它同样可以用作除了线段以外的其他图形的绘制。

2.1.3 关于进一步优化的思考

在使用 DDA 或者 Bresenham 算法进行线段绘制的时候，免不了会讨论各种各样繁琐冗余的情况。比如斜率大于一，斜率小于一，斜率为正，斜率为负，从左到右画，还是从右到左画等等。为了统一画线方式，我们试图采取以下方法进行代码优化。

首先，对于任何线段，我们都从左往右画；其次，对于任何线段，我们首先把它的左端点移到原点，并带着右端点做整体平移；接下来，我们试图把这根直线对称到第一象限，斜率小于一的八分之一部分，在那里进行描点；最后，把所描的点，对称回原来直线所在的八分之一部分，并按照原来的轨迹，把所有点平移回原来的位置。

这样一来，无论是哪种直线，我们只关心如何画好第一象限斜率小于一的那八分之一部分的直线，就可以了。

但这样的代码优化会导致性能下降。画线不像画椭圆：画椭圆的时候，只需要画一个象限的点，其他象限的点只需要通过对称得到，每个点都被 “用上了”。但用上述方法画直线，会有很多为了对称而造出来的点，这些点 “不能被用上”，用于计算它们的时间就浪费了。

2.2 draw_polygon

在生成多边形的时候，我们只需要记录多边形的所有顶点，然后按照描述顺序，将这些顶点用直线连起来就可以了。这个时候，只需要不断调用上一小节的 draw_line。

2.3 draw_ellipse

生成椭圆的时候，我们首先生成在原点处的标准椭圆。然后在根据平移规则，将在原点处的椭圆平移到指定位置上去。在画标准椭圆的时候，我们只需要画出第一象限的所有点，然后按照对称的原则，找到其他三个象限的点集即可。在这个过程中，我们将椭圆函数当作决策参数，进行增量式的描点。并按照椭圆切线斜率，将描点过程细分为两个部分。每一部分按照切线斜率，决定逐渐增加哪一个轴。

以上方法成为中点画椭圆法，已将其运用在任何常见曲线的光栅化中。常见曲线包括圆锥曲线、三角和指数函数、概率分布、多项式和样条函数。

2.4 draw_curve

2.4.1 Bezier

贝塞尔曲线通过几个控制顶点，勾勒出由此定义好的一条曲线，优点是形式简单，计算比较简便。但缺点也是非常的明显，即是不能实现局部控制。一旦改变了其中一个顶点，整条曲线都会受到影响。

在我们的实现中，我们采用了 de Casteljau 递推算法，按照计算组合数的递推算法的思路，依次计算。

2.4.2 B-spline

B-spline 曲线是 Bezier 曲线的一种推广，其继承了贝塞尔曲线的一系列优良性质，并且可以实现局部控制。缺点是实现和计算比较困难。

在我们的实现中，我们利用了 deBoor-Cox 递推对每一个 B-spline 基函数进行计算，然后对所有控制点进行加权求和。

2.5 clip

2.5.1 Cohen-Sutherland

在这个部分，我们利用 Cohen-Sutherland 算法对线段进行裁剪。需要注意的是，该算法只适合用来裁剪线段，不能用来裁剪多边形或者椭圆等其他形状的图元。

Cohen-Sutherland 算法的基本思路是，把线段的定点与裁剪窗口的四条边进行位置比较——点在边的左边还是右边还是上边还是下边。然后据此算出一个区域编码。决定一条线段需要两个顶点，分别算出这两个顶点相对于裁剪窗口四条边的区域编码，然后根据两个顶点的区域编码判断如何对线段进行裁剪。

如果两个顶点的区域编码二进制与起来不等于零，说明两个顶点同时在裁剪窗口某一条边的外侧，这时可以判断直接丢弃线段，因为他不可能出现在裁剪窗口内部。如果两个顶点区域编码的二进制或等于零，说明这两个顶点都位于裁剪相框的内部，不用裁剪。

以上讨论的是两种简单的特殊情况，接下来是除此之外的其他一般情况。这个时候，我们先判断线段的两个端点是否分别位于裁剪线框上边界的两侧（如果不是的话那就都位于上边界的内侧，因为我们已经排除了特殊情况），如果是的话，就需要将上边界上面的部分裁剪掉。这只需要一次求交运算加上一些判断就可以了。剩下来的线段，作为一个新的线段，重新计算两个端点的区域编码，判断是否完全在裁剪线框内部或者完全在裁剪线框外部（之前讨论的两种特殊情况），如果是，则结束讨论输出裁剪结果；如果不是的话，再按照同样的思路，将新线段分别与裁剪线框的右边界，下边界，左边界进行讨论。

2.5.2 Liang-Barsky

Liang-Barsky 算法将二维问题一维化，将线段与裁剪线框之间的求交问题一维化，减少了计算复杂度。在实际操作过程中，我们只需要时刻牢记，我们的计算任务是，从线段的两个端点以及线段与裁剪线框（及其边长延长线）的四个交点一共六个点中，选出两个点作为求交的结果。如果这两个点满足一前一后的关系，就说明线段确实有一部分落在裁剪线框内部，否则应该舍弃整条线段。

从六个点中选出两个点的要点就是要搞清楚，按照直线的方向探索过去，是从哪些边“进入”矩形线框，又是从哪些边“离开”矩形线框。判断这个的标准时利用书上给出的式子里面的 p，然后就能根据这个已知条件进行点的选择。

2.6 symmetry

为了实现上述画椭圆过程中的对称操作，我们新增了一个函数叫“def symmetry(p_list, axis) -> result”，接受一系列点，将他们按照 axis（x 或者 y）进行对称之后，返回对称得到的一系列新点。

2.7 translate

上述画椭圆操作中还有平移操作。平移操作非常简单，接受一系列点，将这些点的横纵坐标都加上一个要平移的常量就可以了。

2.8 scale

缩放改变物体的尺寸。相对于原点的缩放，只需要对每个顶点坐标乘以一个缩放系数 (s_x, s_y)。如果 $s_x = s_y$，则是一致缩放，如果 $s_x \neq s_y$ 则是差值缩放。我们统一针对固定点进行缩放，这样，对于原点的缩放也能被包括进来。

2.9 rotate

根据书上的对于任意顶点进行旋转的公式。

3 cg_cli.py

在命令行界面，我们按要求完成了实验手册中所提到的所有指令，并将所有已经画出来的图元存储起来，在 saveCanvas 的时候统一画出。在这个过程中，我们合并了许多冗余代码。

4 cg_gui.py

在这个小节中，我们主要讨论，为了设计一个简洁而美观交互式图形界面，我们所做的工作。

4.1 文件

4.1.1 设置画笔

在点击菜单中的“设置画笔”的时候，我们先在窗口左下方显示出“设置画笔”的状态提示，然后清除之前被选中的图元，最后利用组件 QColorDialog 与用户交互，让用户选择喜欢的画图颜色，之后所有的图元都是以这种颜色呈现，除非重新选择颜色。

为了实现这一目的，在 MyWindow 类里面记录了用户所选择的颜色，然后通过 color 变量，传递给 MyCanvas 类。在 MyCanvas 类中创建任何新的图元时，按照变量 color 设置新图元的 color 属性，这样一来，这个图元在今后的只会以这种颜色呈现出来。在没有设置画笔的时候，默认使用黑色。

需要注意的是，选中图元时显示的现况永远默认使用红色。另一个注意事项是，在 paint 任何一个图元或者矩形线框之前，一定要先设置画笔颜色 painter.setPen。

4.1.2 重置画布

在点击菜单中的“重置画布”的时候，我们先在窗口左下方显示出“重置画布”的状态提示，“一切重新开始”。

为了实现“一切重新开始”，我们干脆直接重新申明一个 MyCanvas 类型的实例，真正做到“一切重新开始”，而不是通过对现有的画布进行“删除干净”的操作。于是在重置画布的动作中，我们重复了 MyWindow 类中 ___init___ 函数中的部分操作（除了设置菜单的一些操作，因为菜单已经设置好了），换了一块完全崭新的画布上去。

4.1.3 保存画布

在点击菜单中的“保存画布”的时候，我们先在窗口左下方显示出“保存画布”的状态提示，然后弹出一个文件浏览的窗口，选择要保存的文件路径以及文件名。需要注意的是，文件名必须以“.jpg”“.png”或者“.bmp”结尾，否则保存失败。

保存画布的操作非常简单，我们只需要早用户点击“保存画布”的时候，创建实例，调用相应 API 即可。需要注意的是，我们这里保存画布之前清楚了选中图元的红色矩形线框。（也可以将选中图元的红色矩形线框一起保存下来）

4.1.4 退出

在点击菜单中的“退出”的时候，我们先在窗口左下方显示出“退出”的状态提示（尽管非常短暂），然后会弹出一个询问窗口，询问是否要保存当前画布。如果选择“否”，那么直接退出。如果选择“是”，则调用保存画布相关的 API，输入保存文件名字保存后再退出。

4.2 编辑

在进行编辑操作的时候，我们要做的第一件事情是先选中某一个图元，如果没有选中某一个图元，直接点击编辑的相关选项，则会弹出提示框，提示用户先选择一个图元作为编辑的对象。

选中图元后，图元会被一个红色的矩形线框包围，以提示用户。需要注意的是，无论是平移还是旋转还是缩放操作，鼠标都不是必须要点击在红色线框内部才能操纵相应的图元。只要在编辑状态，点击任何地方（包括不在红色线框内部的地方）都会对图元产生影响，具体可一件下面几个小节更详细的说明。

4.2.1 平移

选中图元后，点击“编辑-> 平移”，左下角的状态栏就会显示现在处于“平移”状态。点击鼠标左键并按住不动，向各个方向平移，会导致选中的图元，按照鼠标移动方向相同的方向，平移相同的距离。

在实现过程中，我们只需要记录鼠标左键按下后的初始位置，以及相对于这个初始位置，鼠标左键按住不动时的平移量，然后在图元原来的坐标中加上这个平移量就可以了。

在实现的过程中，我们需要将平移前的图元坐标信息记录下来（进行深拷贝），这样一来，随着鼠标移动的时候，我们可以用于动态更新图元的位置。

4.2.2 旋转

选中图元后，点击“编辑-> 旋转”，左下角的状态栏就会显示现在处于“旋转”状态。点击鼠标左键确定旋转中心，并按住不动，向某一个方向拖动一下确定旋转起始向量，按住鼠标左键，相对于起始向量向哪个地方转动，图元就以刚才确定的旋转中心向哪边转动。

在实现过程中，我们只需要记录鼠标左键按下后的初始位置，以及相对于这个初始位置，鼠标左键按住不动时向某一个方向拖动所确定的起始向量，在接下来的鼠标拖动中，鼠标经过之处与初始位置相连形成另外一个向量，只需要计算这个向量和起始向量之间的夹角就可以确定旋转的角度和方向。

需要注意的是，cos 函数只能计算 180 度角以内的旋转角度，为了旋转 360 度角，我们需要进一步确定两个响亮的相对位置关系。在这里，我们通过计算叉乘来确定两个向量之间的夹角是小于 180 度还是大于 180 度。

4.2.3 缩放

选中图元后，点击“编辑-> 缩放”，左下角的状态栏就会显示现在处于“缩放”状态。点击鼠标左键并按住不动，向右平移，会导致选中的图元，以鼠标点击位置为缩放中心，进行放大操作；点击鼠标左键并按住不动，向左平移，会导致选中的图元，以鼠标点击位置为缩放中心，进行缩小操作。

在实现过程中，我们只需要记录鼠标左键按下后的初始位置，以及相对于这个初始位置，鼠标左键按住不动时 x 方向的平移量，然后根据这个平移量与图元原来大小的比例，对图元进行放缩即可。如果往右平移就是放大，往左平移就是缩小。往右平移的距离除以图元原来的宽度再加上 1 就是图元的放缩系数；从中 1 减去往左平移的距离除以图元原来的宽度，也可以得到缩小的放缩系数。需要注意的是，我们规定单次缩小放缩系数不小于 0.1。

4.2.4 裁剪

选中图元后，点击“编辑-> 裁剪”，左下角的状态栏就会显示现在处于“裁剪”状态。点击鼠标左键并按住不动，向右下平移，会画出一个矩形线框。松开鼠标之后，矩形框框住的部分为裁剪后留下的部分。

在裁剪时出现的矩形线框，是通过 QRubberBand 类创造出来的。最后需要注意的是，裁剪时，也可以向左上角拖动鼠标，会得到相同的结果。

5 引用

[1] 南京大学计算机图形学幻灯片