《计算机图形学》12 月报告

201250182，郑义，201250182@smail.nju.edu.cn

2021 年 12 月 16 日

目录

1 每月进度

1.1 9 月进度

1. 配置了 wsl + ubuntu 20.04 + vscode 环境
2. 预习了直线绘制的 DDA 算法和 Bresenham 算法
   和中点画线算法
3. 学习了 python 的基本语法

1.2 10 月进度

1. 发现 wsl + ubuntu 20.04 + vscode 环境在使用 PyQt5 有一些不兼容问题
2. 更换了 Anaconda + PyCharm 的环境
3. 实现了直线绘制的 DDA 算法和 Bresenham 算法
4. 实现了部分 gui 的功能
5. 实现了绘制多边形的算法
6. 实现了部分的绘制圆的中点圆算法

1.3 11 月进度

1. 11 月主要将除了**裁剪**和**曲线绘制**算法之外的其他主要算法都大致实现完毕了
2. 修改了关于直线绘制的 Bresenham 算法的特殊情况问题和多边形绘制在运用到 Bresenham 算法的情况下会有的一些问题
3. 实现了基本的 cli 程序，12 月会进一步优化和进行测试
4. 实现了直线和圆的绘制的 gui 功能
5. 添加了一些单元测试功能

1.4 12 月进度

1. 完成了所有算法
2. 完成了 gui
3. 复习了之前的算法
4. 移除了之前的 test

2 综述

计算机图形学的大作业从总体来看主要是要实现一个绘图的系统构建，通过 python 语言来实现一些画图的算法和 gui。计算机图形学 (Computer Graphics) 是一门涉及多个学科交叉，通过算法来对计算机的图形进行处理的一门学科，在我看来，计算机图形学中的许多图形绘制算法都需要具有一定的数学基础，同时需要在对算法原理有很好理解的基础上来进行学习。图形在计算机上的显示是一个已经有了长久历史研究的学科，且在当今时代发展出了众多分支，比如计算机视觉等领域，计算机图形学同时也结合了建模、投影等不局限于计算机领域的问题。可以说，计算机图形学是一门基石一般的学科，为当今的许多分支发展奠定了一个良好的基础。

图形在计算机中的显示其实一直是一个让我好奇的问题，我们是如何通过机器中的硬件来实现在屏幕上绘制出与现实世界相差不大的图形呢？有了图形还不够，我们又是如何在屏幕上渲染出五颜六色的色彩呢？这都是让我选择了这门课的原因。通过对本门课程的深入学习和计算机绘制图形的不断了解，才能够真正明白其背后的奥妙之所在，也才能够感慨到先人的智慧之所在。很难想象在之前时代，没有图形用户界面的计算机是如何被人们所接受的，或许，真正驱使计算机图形学蓬勃发展的，正是为了将计算机推入到千家万户的需求所促进的吧。

在本门课上，我们不仅会学习到关于一些基本图形（包括但不限于直线、圆、曲线等等）在计算机中的绘制，同时我们也会学习到计算机图形学中最重要（个人认为）的光栅扫描，光栅扫描可以说是计算机图形学的发展历程中一个十分重要的点，也是值得我们去学习和深究背后原理的一个地方。

本门《计算机图形学》课程大作业主要通过 Python 来绘制常见几何图形来带同学们简单了解计算机图形学的一些基础内容，包括但不限于直线绘制、圆形/椭圆绘制，曲线绘制、图形的变换和裁剪 ...，在经过这次大作业之后，相信我会对计算机图形学中的一些图形基本绘制算法有一定的了解，为以后的课程打下一定的基础。同时也锻炼了我写 python 的能力，在本专业基本都是用 Java，这是第一次用 python 来实现一个完整的系统。

3 算法介绍

3.1 绘制线段算法

3.1.1 计算机绘制直线原理

在计算机中，直线的显示并不是连续的，而是离散的点，这是由光栅化的本质决定的。我们可以把屏幕理解为阴极射线管光栅显示器，这个显示器是由离散可发光的有限区域单元 (像素) 组成的矩阵。确定最佳逼近某直线的像素的过程通常叫做光栅化。对于水平线、垂直线以及 45° 线，选择哪些光栅元素是显而易见的，而对于其他方向的直线，像素的选择就很困难。因此，选择一个重要的算法来实现直线的绘制显得很有必要。计算机绘制直线的算法多种多样，这里主要介绍两种，也是上课所讲的和大作业中所要实现的两种算法

1. DDA 算法
2. Bresenham 算法

3.1.2 DDA 算法的原理

DDA 算法的基本思想是微积分中的微分算法:

$$\frac{dy}{dx} = \frac{y_2 - y_1}{x_2 - x_1}$$

其有限差分近似解为：

$$\begin{cases} x_{i+1} = x_i + \Delta x \\ y_{i+1} = y_i + \frac{y_2 - y_1}{x_2 - x_1}\Delta x \end{cases}$$

即：通过直线的斜率来求出直线上的下一个像素点。然后通过逐渐缩短两个像素点之间的距离 (即减小 Δx 来让无数个点看起来像一条直线)，即当前我们要绘制的像素点的 y 值会等于前一个像素点的 y 值加上斜率 k（这里的设定是 $x_{i+1} = x_i + 1$）。

在实现过程中，如何判断斜率 k 是大于一还是小于一呢？其实并不是计算出斜率来比较，而是通过差值的思想，即定义:

$$dx = x_1 - x_0, dy = y_1 - y_0$$

然后通过比较 $abs(dx)$ 与 $abs(dy)$，如果前者大于后者，则说明 x 的变化量大于 y 的变化量，则斜率的绝对值就是小于 1 的，那么我们就可以通过让 x 作为自变量，下一个像素点 $x_{i+1} = x_i + 1$，然后来改变 y 的值，推导如下所示:

$$y_i = kx_i + b \tag{1}$$

$$y_{i+1} = kx_{i+1} + b \tag{2}$$

$$\because x_{i+1} = x_i \tag{3}$$

$$\therefore y_{i+1} = k(x_i + 1) + b \tag{4}$$

$$\Longrightarrow y_{i+1} = kx_i + b + k \tag{5}$$

$$\Longrightarrow y_{i+1} = y_i + k \tag{6}$$

综上，我们就可以使用 DDA 算法来绘制直线，DDA 算法可以说是在直线方程上的一种优化算法，通过迭代来消除了直线方程中自带的乘法。

但其缺点也十分明显，一个是对于计算出来的 y_{i+1}，实际情况并没有我们所想像的那么美好，在计算机中往往会对像素点进行取整处理，而在 DDA 算法的迭代过程中，这种取整所造成的误差就会不断放大，最终影响到计算机中直线的绘制会越来越偏离原本预期给定的直线。

与此同时，取整操作和浮点数运算在计算机内部的运行都比较耗时，这驱使着我们去寻找在性能上更加优秀的算法来替代 DDA 算法，这也是接下来所要介绍的 Bresenham 算法，同时也是目前较为常用的一种算法。

3.1.3 Bresenham 算法的原理

Bresenham 算法是一种不仅仅用于直线绘制的算法,在其他的曲线表示中也可以用 Bresenham 算法来表示，该算法的核心思想是通过整数增量运算来实现，是一种精确而有效的光栅设备生成算法。与下文即将要介绍的中点画线算法相同，Bresenham 算法通过在两个候选像素之间计算决策参数来决定究竟应该使用哪一个候选像素作为真正要绘制的像素，下面同样以斜率 $k \in (0,1)$ 作为例子：

因为我们此处以 $k \in (0,1)$ 作为例子，假设我们已经确定了当前位置像素点 (x_i, y_i)，则下一个候选像素的位置就是 (x_i, y_i) 和 $(x_i, y_i + 1)$ (因为斜率在 0 到 1 之间)，则我们可以通过计算 x_{i+1} 处的 y 坐标：$y_{i+1} = y_i + k$，然后通过计算其与两个候选像素点之间的距离来确定我们应该选择哪个像素点。

在 Bresenham 算法中，我们引入了决策参数 p_i 来决定要选择两个候选像素点中的哪个像素点，p_i 的计算公式如下

$$p_i = d_1 - d_2$$

其中，$d_1 = y - y_i, d_2 = y_i + 1 - y$，这里的 y 是实际直线在 x_{i+1} 处计算出的值，y_i 是我们在上一步已经选择到的像素点，在这一步其作为候选像素之一使用，另一个候选像素就是 $y_i + 1$，归纳如下：

$$候选像素点一：(x_{i+1}, y_i) \tag{7}$$

$$候选像素点二：(x_{i+1}, y_i + 1) \tag{8}$$

$$y = y_i + k(这里证明请参考\ \text{DDA, k}\ 是斜率) \tag{9}$$

$$d_1 = y - y_i, d_2 = y_{i+1} - y \tag{10}$$

$$p = d_1 - d_2 = 2k(x_i + 1) - 2y_i + 2b - 1 \tag{11}$$

实际操作中，会将 x 的变化量设为 1，此时 y 的变化量就等于斜率，与此同时，由于 p 只是一个决策参数，与上一个像素点无关的量不会改变 (下文用 c 表示)，因此，我们就可以得到下列的公式：

$$p_i = (d_1 - d_2) = 2kx_i - 2y_i + c$$

其中，$c = 2k + 2b - 1$，接下来，我们就可以通过决策参数的符号来判断我们究竟要选择候选像素点中的哪个像素了，具体的判断方法如下所示：

1. $p_i \geq 0$，则应该选择 $(x_i, y_i + 1)$ 作为下一次绘制的像素点；

2. $p_i < 0$，则应该选择 (x_i, y_i) 作为下一次绘制的像素点。

综上所述，Bresenham 算法就实现了运用决策参数来决定下一个像素点，其好处就是使用了整数增量运算，大幅提升了运算速度，相比 DDA 计算浮点数会造成硬件上的计算缓慢，Bresenham 算法则更好的利用了整数运算的优势来绘制像素点。算法的核心就是选取与实际点距离最近的那个候选像素。下面来介绍 Bresenham 算法的一种特殊版本，中点画线法

3.1.4 中点画线法的原理

中点画线算法顾名思义就是通过取中点近似原理来选取下一个像素点的。下面用直线斜率在区间 $(0, 1)$ 为例子说明：对于已经确定的某个像素点 (x_i, y_i)，我们确定下一个像素点的位置是通过取 $(x_{i+1}, y_i + 0.5)$，即取斜率范围边界的两个像素点 (x_{i+1}, y_{i+1}) 和 (x_{i+1}, y_i) 的中点。记中点为 M，然后将直线与 $x = x_{i+1}$ 的交点记作 D，我们就得到了以下的取法：

1. 如果 M 在 D 的上方，那么我们就取 (x_{i+1}, y_i) 为下一个像素点

2. 如果 M 在 D 的下方，那么我们就取 (x_{i+1}, y_{i+1}) 为下一个像素点

上述的步骤就是中点画线法的中心思想所在，在具体的实现中，我们可以通过将距离做 2 倍的扩展并加以处理，这样做的好处是因为在中点计算的过程中，我们待选的两个像素点都是整数，这就会造成中点会以浮点数来表示，我们可以通过 2 倍的扩展来保证在算法的过程中所做的都是整数的运算。

中点画线算法其实是一种较为特殊的 Bresenham 算法，其 python 代码可以用以下方式来实现

```
def mid_pointer_algorithm(x0, y0, x1, y1):
    result = []
    dx = x1 - x0
    dy = y1 - y0
    d = 2*dx + dy

    delta1 = 2 * dx
    delta2 = 2 * (dx + dy)
    x = x0
    y = y0
    result.append((x0, y0))

    while x < x1:
        if d < 0:
            x++
            y++
            d += delta2
        else:
            x++
            d += delta1

        result.append((x, y))
```

3.2 绘制多边形算法

3.2.1 绘制多边形与绘制直线

绘制多边形算法其实与绘制直线的算法是没有区别的，因为我们知道，绘制一个多边形实际上就是绘制多根直线所组成的封闭直线而已。这在程序上也是非常好实现的，对于给定的某个列表（其中的元素是多边形的顶点坐标），我们只需要将其分为多根线段，然后用绘制直线的算法来绘制就可以了，这里不过多赘述。

具体的程序实现可以如下所示：

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

3.2.2 DDA 算法

与直线绘制部分相同，不过多赘述。

3.2.3 Bresenham 算法

与直线绘制部分相同，不过多赘述。

3.3 绘制椭圆算法

3.3.1 reference

1. 中点圆算法
2. 计算机图形学–圆的扫描转换
3. how to draw elliptical sector with bresenhams algorithm
4. Bresenham’s Circle Drawing Algorithm

3.3.2 计算机绘制圆

计算机中绘制圆其实在我看来是一个十分无法想象的过程，我们都知道计算机中是以像素点来绘制出图形的（这里准确的指代应该是用“图像”两个字），而如果我们要十分精确的绘制出一个圆其实是十分困难的，我们只能够在一定误差范围内做到不断的逼近，然而如果仅仅是追求精确度的话，我们大可以通过一些复杂的计算来获得一个十分高精度的圆，但这其实并不符合我们的需求，我们需要的是在保证一定精度的同时，又能够让圆的绘制在计算机中能够快速且不占用过多的计算资源，这就需要我们用算法来减少计算的次数和计算的复杂度，但又得保证圆绘制的一定精度。以下两种算法是最基础的解决了绘制圆过程中计算复杂的问题的算法：

1. Bresenham 算法（没错就是绘制直线的那个算法
2. 中点圆算法（Bresenham 算法的一个改进版）

从算法的名字我们也可以发现，其实圆绘制的算法和直线绘制的算法是有异曲同工之妙的，这其实是因为圆的绘制本质上也是通过决策参数来决定下一个像素点的绘制情况，我们将曲线泛化成了离散的像素点，通过缩小单位像素点的大小（即提高分辨率）来不断地拟合曲线，已达到我们肉眼看不出这段曲线与真正曲线的差别的一种过程。

实际上，在绘制圆的过程中，我们运用到了圆的对称性，对于一个圆来说，我们只需要绘制其的八分之一就好了，同时我们为了方便，常常会将圆平移到原点来进行绘制，这样能够方便我们的计算与对称，计算好后再平移回去即可。对于椭圆也是同理的，但是要注意椭圆的对称性较弱，我们需要画出四分之一的椭圆，这里就又涉及到了比圆稍微复杂一点的多情况的讨论，我们会在后面的小节里进一步阐述。

3.3.3 圆与椭圆

我们知道，圆是一种特殊的椭圆，本小节主要介绍的是在处理计算机绘制圆和椭圆的过程中的一些差异和相似点。我将其分成了以下的几点。（注：下面的圆与椭圆都是以原点为对称中心绘制的，且我们认为椭圆的两个参数 a, b 分别对应椭圆的长短轴，圆的参数 r 为圆的半径）首先让我们来看看差异：

1. 圆的对称性不仅包括了关于 x 轴对称，关于 y 轴对称，还包括了关于 $y = x$ 和 $y = -x$ 这两条对称，而椭圆则没有后面两条直线的对称性，因此对于计算机绘制圆的过程，我们只需要画出八分之一的圆，然后根据对称性就可以得到一个完整的圆了。但对于椭圆来说，我们需要画出四分之一的椭圆，然后才能够使用对称性。

2. 其次就是由于上面说的对称性问题，因此在圆的绘制中实际上我们一直都是位于 $y = x$ 和 $x = 0$ 这两条直线间绘制圆（即从 $(0, r)$ 绘制到 $(\sqrt{2}r, \sqrt{2}r)$）即可，也就是说我们一直是在圆的切线小于 1 的情况下来绘制圆的。考虑一下在直线的绘制当中，我们也是将情况分成了 $k > 1$ 和 $k < 1$ 来考虑的，由于对称性，我们在圆的绘制中就不需要考虑切线斜率的变化情况，只要将 x 作为自变量，然后决策 y 值即可，这对我们来说减少了情况的讨论。但对于椭圆来说，椭圆没办法避免这种情况的讨论，我们只能够将情况分成切线斜率大于 1 和切线斜率小于 1 来进行讨论，分情况是为了将决策的对象的变化率限制在 $(0, 1)$ 之间，以方便我们的计算和调整。

从上述的讨论来看，圆和椭圆在计算机中的绘制还是有点区别的，当然了，我们完全可以将圆看做特殊的椭圆，然后用中点椭圆绘制算法来绘制圆，这样更具有普适性，在程序上也没有必要分那么多种情况来讨论。但有利有弊，方便自己还是方便机器，看个人的抉择。

3.3.4 中点圆生成算法

这一节我们来讨论中点圆生成算法的一些详细步骤，中点这种思想其实是在 Bresenham 算法的基础上改进而来的，这在直线的绘制中我们已经感受到了。对于椭圆的情况，只是在圆的基础上增加了斜率大于 1 和斜率小于 1 的讨论而已，实际上基本思想是一样的。

思考一下如果是用最直接的绘制圆方法来绘制圆，我们会怎么做呢？或许很多人会想到直接用计算距离来决定下一个点，或许这样做能够足够精确，但是带来的是开根号的计算，中点圆算法的思想其实也是直接计算距离，只不过计算的是距离的平方，这样做避免了开根号的运算，通过计算候选像素点的中点和圆心的距离关系，来决策下一个像素点究竟应该是哪个。我们来看一下中点圆算法的决策参数:

$$f_{circle}(x, y) = x^2 + y^2 - r^2$$

实际上这就是一个对点和圆心距离计算的公式，如果是椭圆，我们只需要稍加更改即可，更具一般性的椭圆决策参数如下：

$$f_{ellipse}(x, y) = b^2x^2 + a^2y^2 - a^2b^2$$

这里 b 是椭圆短半轴的长度，a 是椭圆长半轴的长度，其实从椭圆的标准方程我们可以很容易得到这一决策参数。$\frac{x^2}{a^2} + \frac{y^2}{b^2} = 1$，然后进行一定的变换就可以得到我们的决策参数。

下面我们来看一下有了决策参数之后，绘制圆的一个具体过程，由于上面说的我们只需要绘制八分之一的圆，我们将我们绘制的圆定于点 $(0, r)$ 和 $(\sqrt{2}r, \sqrt{2}r)$ 之间，根据常识我们可以知道这一段圆弧的切线斜率是逐渐增加的，且圆弧上的点的 y 值是随着 x 值的增加而不断减少的，所以我们的候选像素就十分清晰明了了，假设当前已经选中的像素点是 (x_i, y_i)，那么由于切线斜率 $k \in (-1, 0)$，所以我们的候选像素点实际上就是 $(x_i + 1, y_i)$ 和 $(x_i + 1, y_i - 1)$ 两点，即 y 值不变或 y 值减一两种情况，那么接下来要做的步骤就十分清晰了。

两候选像素点的中点 $p_k = (x_i + 1, y_i - \frac{1}{2})$，将其代入决策参数方程 $f_{circle}(x, y)$，然后通过决策参数的符号来判断下一个像素点应该选择哪一个。

1. $p \geq 0$: 此时中点位于圆弧的外围，我们选择点 $(x_i + 1, y_i - 1)$ 作为像素点，因为此时该点是更加接近圆弧的实际位置。

2. $p < 0$: 此时中点位于圆弧的内部，我们选择点 $(x_i + 1, y_i)$ 作为像素点，因为此时该点是更加接近圆弧的实际位置。

在解决了如何根据决策参数来选择像素点之后，我们所要做的就是来计算决策参数的增量了，增量计算能够减少很大一部分的重复计算，减少计算的时间，加快图形绘制的速度。我们来看一下不同情况下增量公式推导的过程:

对 $p \geq 0$ 的情况，我们知道 p_i 选择的是 $(x_i + 1, y_i - 1)$ 的点，所以：

$$p_i = (x_i + 1)^2 + (y_i - \frac{1}{2})^2 - r^2 \tag{12}$$

$$p_{i+1} = (x_i + 2)^2 + (y_i - \frac{3}{2})^2 - r^2 \tag{13}$$

$$\Longrightarrow p_{i+1} = (x_i + 1)^2 + (y_i - \frac{1}{2})^2 - r^2 + 2x_i - 2y_i + 5 \tag{14}$$

$$\Longrightarrow p_{i+1} = p_i + 2x_i - 2y_i + 5 \tag{15}$$

对 $p < 0$ 的情况下，我们知道 p_i 选择的是 $(x_i + 1, y_i)$ 的点，所以：

$$p_i = (x_i + 1)^2 + (y_i - \frac{1}{2})^2 - r^2 \tag{16}$$

$$p_{i+1} = (x_i + 2)^2 + (y_i - \frac{1}{2})^2 - r^2 \tag{17}$$

$$\Longrightarrow p_{i+1} = (x_i + 1)^2 + (y_i - \frac{1}{2})^2 - r^2 + 2x_i + 3 \tag{18}$$

$$\Longrightarrow p_{i+1} = p_i + 2x_i + 3 \tag{19}$$

通过增量公式我们就可以得出各个像素点的决策参数从而绘制出一个圆，在圆中我们只需要画到 $x == y$ 的情况下就可以结束，然后对每一点通过对称性来对其他七个点进行绘制即可。

3.3.5 中点椭圆生成算法

在 3.3.3 圆与椭圆中，我们已经简要介绍了计算机绘制圆与椭圆的一些差别，但其实两者的中心思想都是一样的，通过距离来决策候选像素，并且通过图形本身的对称性来减少计

算量，从而达到快速绘制的一个目的。我们来看一下椭圆的决策参数，实际上就是椭圆标准方程的一个转换：

$$f_{ellipse} = a^2y^2 + b^2x^2 - a^2b^2$$

与圆的决策参数相同，椭圆的决策参数的符号也是与点和椭圆的位置有关系的，如果决策参数 $p = f_{ellipse} < 0$，那么点就在椭圆内，若 $f_{ellipse} > 0$，那么点就在椭圆外，如果 $f_{ellipse} = 0$，那么点就在椭圆上，这实际上和圆是一样的。

椭圆与圆区别较大的一点可能就是对称性上面了，椭圆我们需要分成两块区域来进行处理，对于切线斜率的绝对值小于 1 的部分，我们以 x 作为自变量，来决策 y 的位置，对于切线斜率的绝对值大于 1 的部分，我们以 y 作为自变量，来决策 x 的位置，同样我们以 $(0, b)$ 作为绘制的起点, $(a, 0)$ 作为绘制的终点。然后分区域，分选择像素点不同的增量公式不同来推导即可，实际上思想与圆并无差异，只能说圆是一种特殊的椭圆了。

3.4 绘制曲线算法

3.4.1 计算机绘制曲线

...

3.4.2 Bezier 算法

...

3.4.3 B-spline 算法

...

3.5 平移/缩放/旋转

3.5.1 平移

平移变换是图形的变化中比较简单的一种，在这里就用比较简单的语言来待过。以线段为例，我们只需要知道某条线段要平移的变化量 $\Delta x, \Delta y$，然后对线段上的所有点都加上变化量即可。

不只是线段，对所有的图形的平移变换我们只需要对图形的点集 $[x_i.y_i]$ 里的所有点都加上变化量即可，即平移后的点集应该变化为 $[x_i + \Delta x, y_i + \Delta y]$, 就是我们新得到的图形。

我们以大作业里的平移代码为例：可以看到思想其实十分的简单

```
def translate(p_list, dx, dy):
result = []

for i in range(len(p_list)):
    xi = p_list[i][0]
    yi = p_list[i][1]
    result.append((xi + dx, yi + dy))
return result
```

总结：平移变换是图形学所有变换中最为简单的一种变化，但在我们绘制图形的过程中有着十分重要的作用，比如在上文我们使用中点圆算法绘制圆形的时候，我们所绘制的圆形其实都是以原点为圆心的，能够这样做的依据就是我们可以在将圆心在原点的圆绘制出来之后，再通过平移将其移动到实际要绘制的位置即可，平移让我们能够最大限度的利用到某些图形的对称性，从而简化我们绘制图形的过程。

3.5.2 缩放

对于缩放变换，首先我们需要选择一个不动点来作为我们变化的参照点，图形的点集都以这个不动点为参照来进行变换，变换的公式如下 (这里以原点为参照点)

$$x_i^{'} = x_i \times s_x,\ y_i^{'} = y_i \times s_y$$

这里的 $[x_i, y_i]$ 就是原本图形的点的坐标，s 是我们缩放的倍数，要注意，这里的 s 必须是一个正数，我们容易知道，$s > 1$ 则为放大图形，$s < 1$ 则为缩小模型，$s = 1$ 则为保持不变。

大作业的 python 代码表示的以任意点为中心点来进行缩放的代码如下所示:

```
def scale(p_list, x, y, s):
result = []
for i in range(len(p_list)):
    xi = p_list[i][0]
    yi = p_list[i][1]
    change_x = x + (xi - x) * s
    change_y = y + (yi - y) * s
    result.append((change_x, change_y))

return result
```

总结：缩放变换的算法或许也十分简单，但是我认为还是有存在着问题的，或许会存在这样一种 $s < 1$ 的情况，使得缩放之后图形的像素点过于密集，原先图形的像素点都是绘制在整数点上的，但在缩放之后由于我们只能够在屏幕上绘制整数点，导致了 $s < 1$ 的缩放之后，再次对图形的点取整会导致一定的误差，当然，这种误差或许十分的小且不可见，毕竟在绘制图形的过程中，我们所使用的像素点随着分辨率的增加而增加，当分辨率增加到一定时，或许单个像素点的偏差已经不足以影响我们的视觉体验。

缩放变换也是图形绘制中的一种基本变换，同时也是在构建更复杂的图形中需要用到的必不可少的技巧，在缩放的过程中我们需要注意的点就是对缩放中心点的使用。

3.5.3 旋转

旋转变换是另一种基础的图形变换，采用的算法也是十分直接的计算过程。这里我们假定给定的旋转角是顺时针的弧度制表示 θ，便于后面的流程表示。旋转变换和缩放变换相同，都需要以一个“基准点”来作为我们变换的中心，一种方便的方法是通过平移变换将基准点和图形都以相同变化量移动到坐标原点，这样做的目的一方面是方便计算机内部的硬

件进行计算，另一方面也是使得我们的变换更加直观，便于我们理解。（这里的两次平移可能并不会让这种取巧的算法在时间上占据更多的优势，但也算是方便了我们的代码编写）

旋转变换与上面的变换是同理的，我们只要对某个图形的点的集合进行操作即可，具体的变换公式如下所示 (我们以原点为基准点):

$$x^{'} = x\cos\theta - y\sin\theta \tag{20}$$

$$y^{'} = x\sin\theta + y\cos\theta \tag{21}$$

可以看到在变换过程中，角度的确定就确定了我们变换之后的图形。倘若我们的基准点不在原点的话，我们也只需要加上一个“偏置”即可，即如下所示:

$$x^{'} = (x - x_0)\cos\theta - (y - y_0)\sin\theta \tag{22}$$

$$y^{'} = (x - x_0)\sin\theta + (y - y_0)\cos\theta \tag{23}$$

这里 (x_0, y_0) 是我们的基准点。我们用大作业中的旋转变换来做一个代码的实例：

```
def rotate(p_list, x, y, r):
result = []
# 将角度转为弧度，用于后面的三角函数计算
angle = math.radians(360 + r)
for i in range(len(p_list)):
    xi = p_list[i][0]
    yi = p_list[i][1]
    change_x = x + (xi - x) * math.cos(angle) -\
            (yi - y) * math.sin(angle)
    change_y = y + (xi - x) * math.sin(angle) +\
            (yi - y) * math.cos(angle)
    result.append([int(change_x), int(change_y)])

return result
```

总结：旋转变换也是一种很基础的变换，在我们图形绘制的过程中有着十分重要的作用，合理结合上述的三种变换，就可以在基础图形的绘制上进行各种变形从而达到各种绘制的效果。

3.6 裁剪算法

3.6.1 Cohen-Sutherland 算法

Cohen-Sutherland 算法是一种最早使用但是十分流行的算法,在我们进行线段裁剪的过程中，我们很自然的想到的办法是通过求线与边界的交点来进行裁剪，而 Cohen-Sutherland 算法的核心思想就是基于这个基础之上来进行更进一步的改进，通过编码测试来减少要计算的交点的次数。具体做法表述如下：

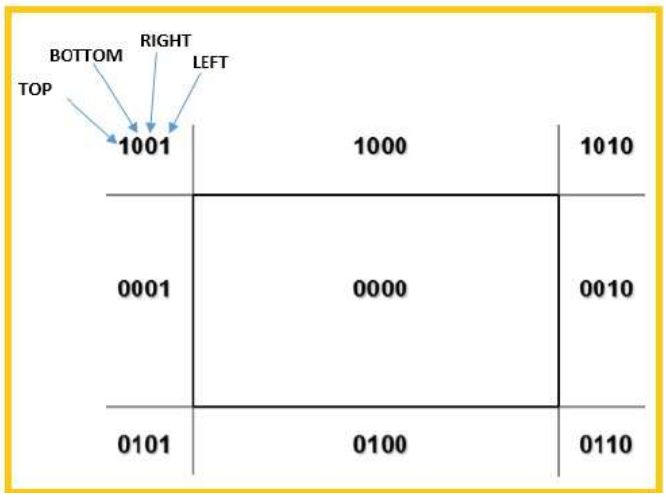


图 1: Cohen-Sutherland 算法编码

通过将裁剪窗口 (这里我们假设裁剪窗口是一个方形的窗口) 的四条边延展，我们可以将整个计算机屏幕分成 9 块，根据上述的 Cohen-Sutherland 算法的核心思想，我们需要对这九块区域进行编码，考虑到计算机内部都是用二进制来表示各种数据，很自然的我们就会联想到用二进制位来表示这几个区域，为了表示 9 个区域，我们就需要 4 位数据来进行表示，这就是我们算法的核心，该算法规定了以下的编码标准 (当然你也可以有自己的一套编码规则)

1. 如果在裁剪窗口上方，就将第一位 (高位) 标为 1
2. 如果在裁剪窗口下方，就将第二位标为 1
3. 如果在裁剪窗口右边，就将第三位标为 1
4. 如果在裁剪窗口左边，就将第四位标为 1

通过这样的编码，我们能够很轻松的确定端点的位置，有了编码的基础，在裁剪的过程中，我们要做的就是计算端点的编码，比如我们现在知道了裁剪窗口的左上角和右下角的两个点 (这里我们假设坐标系的 x 轴和 y 轴与平时数学计算时的坐标系是相同的，而不是写代码时建立坐标系的方式)。那么我们只需要通过比较端点和这两个点的 x 的大小以及 y 的大小就可以很轻松的对两个端点进行编码。

当然这个算法只是让我们求交的过程稍微简单了一点，在确定了端点的编码之后，如果其不在裁剪框内，我们要做的就是依次与各个边界进行求交操作，确定在该边界之内的线段，注意这里我们必须要与各个边界都需要计算一遍，直到端点的编码变成了全 0(全 0 代表着该端点在裁剪框之内)。只有在与所有边界都计算了一遍之后，我们才能够真正确定我们要裁剪的部分。

该算法用一个十分简单的思想来简化了我们对线段进行裁剪的方式。其逐渐被 Liang-Barsky 算法取代主要是因为线段在我们看来还是一个连续的表示形式，用编码来进行测试能够很好的确定线段是不是在裁剪框内，但对于不在裁剪框的部分，我们仍然需要通过求交来确定线段与边界的交点，这实际上还是给计算机内部的计算带来了一定的复杂度，而接下来要介绍的一种算法就更加简化了求交方面的问题。

3.6.2 Liang-Barsky 算法

...

4 系统介绍

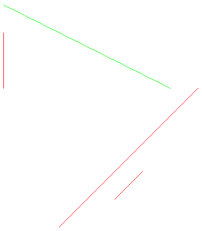
gui 系统方面是在助教给的代码基础上进行了一定的拓展与修改，使用了 PyQt5

5 总结

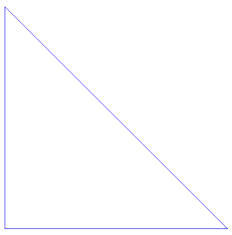
...

6 成果展示

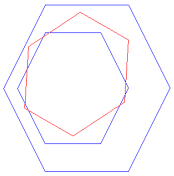
以下内容主要是对大作业中实现的各种算法进行一个简单的展示，通过助教给出的 input.txt 文件所绘制出的 jpg 文件如下所示



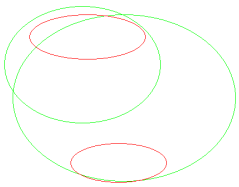
(a) 直线绘制与裁剪



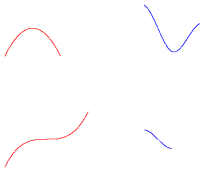
(b) 多边形绘制



(c) 多边形绘制与变换



(d) 椭圆绘制与变换



(e) 曲线绘制