《计算机图形学》系统技术报告

姓名：刘志刚 学号：141242022

(南京大学 匡亚明学院, 南京 210093)

引言

本报告主要对图形学大作业所使用的技术进行详细介绍。第一部分是使用的技术、技术难点、解决方案及工程实现说明，第二部分是使用的绘图算法。

技术及实现

使用的图形库

本图形系统在开发时主要使用的是微软.NET平台的WinForm图形界面开发框架。其中主要包括了窗体和控件。我们在窗体上绘图，使用控件、鼠标等实现控制。另外，对于3D六面体部分，使用了微软.NET平台的WPF开发框架。在开发语言的选取上，使用的是C#。使用Visual Studio进行开发，程序在Windows平台上运行。

图形绘制模式

我们使用最基础的绘制点的函数进行绘制，由于我们是使用鼠标进行操作，要实现所见即所得的效果，我们在绘制时必须要让绘制过程中的图形随鼠标的移动而改变，进行重绘。因此随着鼠标的移动，重绘是非常频繁的，因此图形绘制模式对于保证用户体验及提高性能异常关键

基于时钟的定期重绘

在一开始，我使用的是基于时钟的重绘。即设定一个周期性的时钟，以10ms为间隔。每次时钟到来，我们就将屏幕清空，并把所有图形重新绘制一遍。简单总结以下：即有两个事件处理循环，在鼠标事件处理函数中更新图形列表；在时钟事件处理函数中刷新屏幕，重绘图形。这样有个好处就是实现起来简单，只要定期重绘，不需要跟踪记录每个像素点什么颜色。而缺点就是响应比较慢，在画线时进行拖动，虽然线也能跟着鼠标走，但是总让人感觉线跟不上鼠标。还有就是，定期重绘，耗费CPU的计算资源，即使我们什么也不画，也要进行计算。第三点缺点就是，重绘速度（即响应）与屏幕上要画的点的数目有直接关系。对于普通的线、圆，重绘都异常快。但是对于填充图元，由于填充图元涉及到的点太多，会导致系统重绘时消耗时间过多，响应变慢，甚至失去响应。

流程图如下：

图形列表，记录用户画下的所有图形

鼠标事件

时钟事件

记录图形

刷新屏幕，重绘所有图形

基于动作的增量绘制

由于定期重绘存在以上问题，因此我们采用了基于动作的增量绘制，即根据用户的操作，我们每次只更新屏幕上发生改变的部分。流程图如下：

下面主要说明一下在这个模式下我遇到的主要问题及解决方法。

**如何追踪颜色？**

这种模式复杂就复杂在跟踪每个像素点的状态，这个像素点有没有颜色？是什么颜色？有多少个图形在这个点有颜色？将某个图形移除之后，这个像素点应该变成什么颜色？如果我们允许图形有不同的颜色，则这个问题提就更加复杂了。要完整地支持不同颜色图形，我们可能需要实现图层等高级概念，即我们按照图形绘制的先后顺序，后绘制的图形位于上面的图层，我们显示最上面一层图层的颜色。移走一个图形，就是改变那个图形所在的图层。只有当最上层的图层移开时，我们才需要更新这个点的颜色。但使用图层显然过于复杂。但没有颜色又是不行的，例如，我使用鼠标选中图形，如果没有颜色，我怎么知道我有没有选中图形？

对此我见到了两种解决方案：

欧先飞同学只使用一种颜色，这样就只要对每个像素点维持一个引用计数，即记录有多少个图形在这里有颜色。当且仅当计数值变为0时，我们才为这个点重新绘上背景色。对于选中，他采用了着重显示所选图形端点的方式。

我使用的是固定颜色的方案。即绘制，填充，选中颜色各不同，但是都固定。绘制过程中，颜色为蓝色。绘制完成后，颜色变为黑色，选中拖动、旋转时颜色为黄色，填充时颜色为棕色。则为每个像素点，我维护四个计数值。分别表示这四种颜色在这里的引用计数。同时，对于由于没有图层的概念，我规定黑色总是在最下层，因为那是绘制完成的图形的颜色。

**如何避免闪烁**

即使我们把窗体的属性置为doubleBuffered（双缓冲），即使用图形系统为我们提供的界面双缓冲，在画线时，拖动直线依然会出现小的闪烁问题。

我们可以简要分析一下，闪烁出现的原因。如下图所示，我们把直线的一个端点从b拖动到c。

b

c

a

在对屏幕进行增量更新时，我们需要将线段ab上的像素颜色置为背景色，将线段ac上的像素颜色设为蓝色。但是如果直接用一个循环更新每个像素，由于更新像素这个动作相对比较耗时，中间擦除点、绘新点的过程会显示出来，会有闪烁。

对此，在欧先飞的建议下，我使用了双缓冲。我将一个bmp图像作为自己的buffer，每次增量更新都是先将bmp图形进行更新，最后再将bmp内容整体的映射到屏幕上。这样子对屏幕的更新次数就明显减少，中间的擦除点，绘点过程就不会由于更新像素点的速度问题，导致闪烁。

在自己维护的缓冲中更新完成后, 整体更新

b

c

a

b

c

a

如何仅使用鼠标来输入这些图形

由于我使用的是图形界面输入，因此绘制图形所需的全部信息必须从鼠标获得。具体说来，winform对于鼠标的动作提供了多个事件，我们要通过这些事件来获得用户的输入。我用到事件有MouseDown（按下鼠标键）、MouseUp（松开鼠标键）、MouseMove（鼠标移动）、MouseClick（鼠标单击）、MouseDoubleClick（鼠标双击）。

如何提升代码的可维护性

由于窗体的鼠标事件处理默认只有一个回调函数，这意味着我们不管画什么图形，都要写在这一个回调函数里。这让维护变得异常复杂。

下面的表格展示了各个图形的绘图操作模式，以及涉及到的事件：

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 绘图 | 操作 | MouseDown | MouseUp | MouseMove | MouseClick | DoubleClick |
| 线/圆/椭圆/变换/裁剪/编辑 | 左键按下，移动鼠标，最后抬起 | ☑ | ☑ | ☑ |  |  |
| 多边形/样条曲线 | 单击确定一个点，最后一个点双击结束 |  |  |  | ☑ | ☑ |
| 扫描填充/泛滥填充 | 单击 |  |  |  | ☑ |  |

可能这看起来还不算复杂，但是考虑到不同的事件处理函数之间还需要共享变量。同时，单击会先后触发MouseDown、MouseClick、MouseUp事件，而一次双击会先后触发MouseDown、MouseClick、MouseUp、MouseDown、DoubleClick、MouseUp事件。最后的结果就是每一个回调函数都巨长无比，里面充满了if else。同时还要小心地初始化及设置共享变量，否则你画线时的结果可能会对你画多边形造成影响。

最终，我决定换一种模式，即按照上面的操作模式分类，不同的操作模式采用不同的鼠标事件回调函数。同时，不同模式的操作事件处理函数之间坚决不共享变量。我们在切换绘图模式之后，第一件事情就是切换鼠标事件处理函数，及初始化好相关变量。

最终在提升了一定量代码冗余的情况下，可维护性极大提高，一些奇奇怪怪的操作上的bug也一一解决。

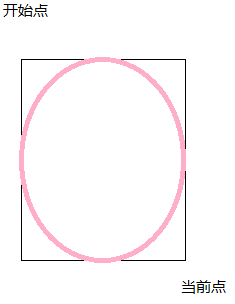
下面，我们描述几个操作在实现中，是怎么做的。

圆

对于圆，我们以鼠标左键按下去的点为圆心，以鼠标当前位置到圆心的距离作为半径，这样随着鼠标的移动，圆自然就变大变小。

椭圆

在绘制椭圆时，我们至少需要三个信息，即椭圆圆心，椭圆的长轴和短轴的长度。经过欧先飞的提示，我采用了根据鼠标的开始位置和当前位置确定一个矩形，再绘制这个矩形的内切椭圆的方式。如下图所示：



我们最开始点下去的点即开始点，鼠标当前位置即当前点，以这两点做矩形，再绘制出其内切椭圆即可。

选中

对于选中，我采取了计算离鼠标单击点最近的图形的方式。同时，该距离必须小于一定值，图形才能被选中。

旋转

在旋转时，最主要的问题是：如何通过鼠标计算出旋转角度。对此，我采取这样的计算方式。鼠标最开始单击的位置为a，之后连续两次鼠标的位置分别为b和c。c为当前当前位置，b为鼠标上一次的位置。则在这期间，旋转角度表示为∠cab即可。使用简单的向量运算即可算出夹角。实践表明，采用这种计算方式还是很容易控制图形旋转角度的。

示意图如下：

a

b

c

如何求出角cab的值呢？不妨称此角为alpha。

利用内积，我们可以算出此角的余弦值：

由余弦值，我们只能知道此角的大小，但不能知道转动的方向，根据算法导论中的相关内容，叉乘的符号，说明了方向。不妨令,。则有：

同时，有结论，若叉乘结果为正，则从ac转到ab是顺时针。由此，我们既可以算出转动的方向。

值得注意的是，我们接下来应该直接用算出来的正弦、余弦值去填旋转矩阵即可，并不需要再计算反三角函数求出角度。这不仅有先反三角再三角的精度损失原因，更重要的是，计算反三角之后再计算三角，会容易算出NaN等浮点异常，使处理更加复杂。

3D六面体

由于winform框架并没有对3D提供很好的支持，因此我使用了WPF框架来显示3D六面体。WPF对3D提供了很完备的封装。因此我们只需要几行代码，即可显示一个3D六面体。

在WPF 3D中，存在如下概念：

DirectionalLight：光源

MeshGeometry3D：构成曲面的一系列三角形，例如六面体有六个面，每个面由两个三角形组成

DiffuseMaterial：曲面材质

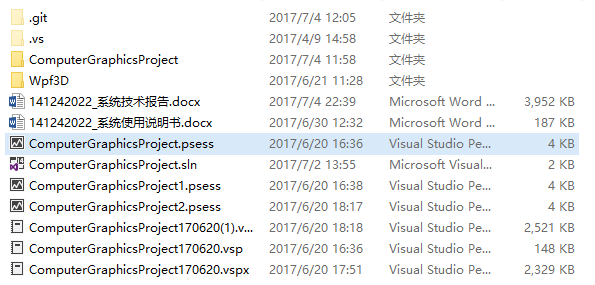
Camera:即观察点及观察方向

因此，我们在编程时只需指定上述元素即可。

工程实现

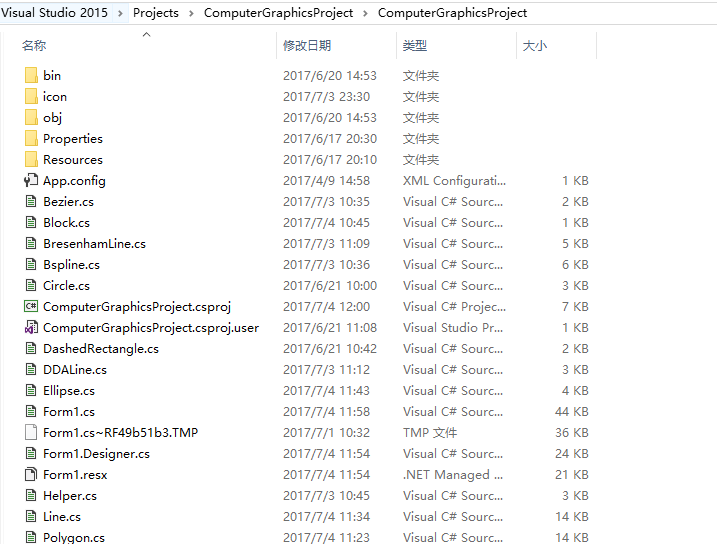
目录结构

如下图所示，在工程目录下包括两个文件夹ComputerGraphicsProject及Wpf3D，这两者分别对应于此工程中的两个项目，前者为绘图，是主窗口所在的项目，使用winform开发；后者为3D六面体窗口，使用WPF开发，被主窗口调用。

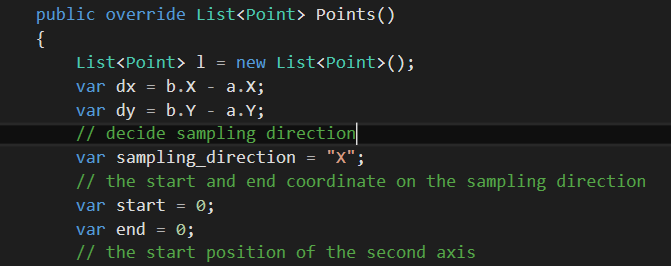


如下图所示，对于ComputerGraphicsProject，它的源文件主要在From1.cs中，里面主要是负责处理鼠标响应、双缓冲管理、文件保存等功能的函数。而其他一些\*.cs源文件，如Line.cs，DDALine.cs则如文件名所表述的一样，是在实现对应的算法。在每个文件中，都有一个函数Points()，是用来计算这个图形上的所有点的。这即是绘图算法的实现。

实现3D六面体的相关代码在Wpf3D项目文件夹下的MainWindow.xaml文件中。



下图展示的DDALine.cs中的Points()函数的实现：



以上介绍的是大致的目录结构，以及文件中大致的内容是什么，具体的代码细节还需要看代码才能知道。

类结构层次

本项目使用了面向对象的范式，其类的大致结构层次如下。

Primitive

Line

Polygon

Spline

Circle

Ellipse

Blockk

DDALine

Bresenham

Bezier

Bspline

基类是Primitive，主要实现了通用的图形绘制（Draw）/擦除函数（UnDraw）。以下有各个派生类，绝大多数图形学算法，如计算曲线上的点、裁剪、填充、平移、旋转、缩放等都是在派生类中实现的。

其中值得注意的是Block这个类，这个类不是什么图形，它是填充色块。它的主要组成部分就是填充点集合。添加这个类之后，便于我们对填充色块进行管理。

另外，由于泛滥填充与任何一个图形都无关，只与屏幕上有没有像素有关，因此泛滥填充是在Form1.cs中实现的。

绘图算法

接下来是绘图算法的描述，这部分内容主要是抄的书和PPT上的，到后来，我懒得打公式了，就直接截图了。您看看就好。公式如有疏漏或错误，还望见谅。

DDA（数值差分分析算法）

数字差分分析(DDA： Digital Differential Analyzer)方法是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换算法。也就是说，通过在一个坐标轴上以单位间隔对线段取样(取△x=1 或△y=1)，计算△y 或△x 决定另一个坐标轴上最靠近线段路径的对应整数值。

先考虑具有正斜率，从左端点到右端点进行处理的线段。若斜率 m≤1，则在单位 x

间隔(△x=1)取样并计算每个顺序的 y 值：

其中，下标 k 取整数值从第一个点 1 开始递增，直至最后端点。由于 m 可以是 0 与 1 之间的任意实数，所以计算出的 y 值必须取整。

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：

假如起始点在右侧，则取△x = -1，并且：

或者(当斜率大于 1 时)，△y=-1，且：

计算具有负斜率线段的像素位置也可用上述方程进行。假如斜率的绝对值小于 1，  
并且起始端点在左侧，可设置△x=1 并用方程计算 y 值；当起始端点在右侧(具有相  
同斜率)，可设置△x=-1 并且方程得到 y 位置。同样，负斜率的绝对值大于 1 时，用  
△y=1 和方程，或者△y=－1 和方程。

这个算法将两个端点的像素位置作为输入，其过程可概括为：计算端点位置间的水  
平和垂直差值，两者绝对值大者决定算法的循环次数；从某一端点像素位置开始，确定  
沿线段生成下一个像素位置每一步的所需偏移量，计算下一个点的某一坐标值并取整；

按循环次数循环上述过程。

假如 x 方向差值的绝对值大于 y 方向差值的绝对值，且起始端点位于直线左侧(即起  
始端点 x 方向的坐标值小于另一端点的同方向坐标值)，那么 x 和 y 方向的增量值分别为1 和 m。假如 x 方向变化大，但起始端点位于直线右侧(即起始端点 x 方向的坐标值大于另一端点的同方向坐标值)，那么就采用减量-1 和-m 来生成线上的每个新点。其它情况下，在 y 方向使用单位增量/减量， x 方向使用 1/m 的增量/减量。

Bresenham算法

原理

Bresenham 画线算法是一种精确而有效的光栅线段生成算法，它可用于圆和其它曲线显示的整数增量运算。图 2-21(a)示出了直线段的局部显示，垂直轴线表示扫描线位置，水平轴线标识像素列数，在此例中，在某个方向以单位间距取样。根据扫描转换原理，在每一个取样位置处，需确定哪个像素位置更接近于线段路径。

先考虑具有小于1的斜率线的扫描转换过程。线路径上的离散像素位置可通过在x方向以单位间距对线路径取样来确定：从给定线段的左端点所在像素位置开始，以单位间隔依次处理每个后继像素列x位置，在所处理像素列选择 y 值最接近线段的像素，并逐次绘出。

为简化像素的选择，Bresenham 算法通过引入整型参量定义来衡量两候选像素与线路径上实际(数学)点间在某方向上的相对偏移，并利用对整型参量符号的检测来确定最接近于实际线路径的像素。在这个过程的第 k 步，假如已经确定了这一步所显示的像素在位置处，那么，下一步需要确定在列上绘制哪个像素：是在位置还是在位置？在取样位置处，用d1和d2来标识两个候选像素与线段数学路径的垂直偏移，在像素列位置处数学线段上的y坐标可计算为：

注意：屏幕坐标为单位网格，所以有： ；。

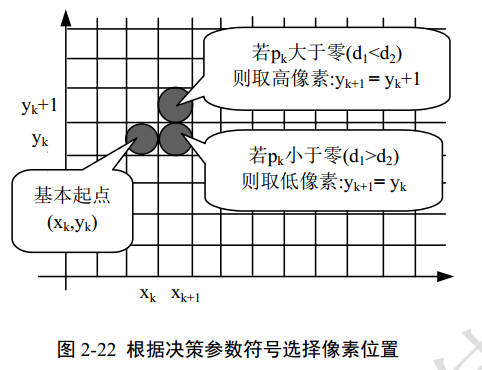
那么，两个候选像素与线段数学路径的垂直偏移分别为：

这两个分离点的距离差分为：

设：△y 和△x 分别为端点的垂直和水平偏离量，令：，代入上述方程可以得到：

式中的称为画线算法中第 k 步的决策参数。由于屏幕网格坐标系为整数坐标系，因此，方程仅包含整数运算，且参数：是一常量，与像素位置无关，且会在循环计算时被消除。

由于x方向单位增量，因此，的符号与的符号相同。假如处的像素比的像素更接近于线段(即)，那么，参数是负的，此时，选择绘制处的像素；反之，处的像素比的像素更接近于线段(即)，那么，参数是正的，此时，选择绘制处的像素。也就是说，可以根据的符号来决定第 k+1 步所需要选择的像素。



生成过程

每一单位步长都会引起沿线段x和y方向的坐标变化。因此，可利用递增整数运算得到后继的决策参数值。在k+1步，决策参数可从上述方程中计算出为：

将上述方程减去方程，可得到：

但，因而得到：

其中：的取值取决于参数的符号，或者说，取决于像素的选择。规则如下：

，当，

，当，

即当时，取高像素，，此时，决策参数为：

反之，当时，取低像素，，此时，决策参数为：

从线段某个坐标端点开始，在每个离散整数x位置，反复进行决策参数的这种递归运算，就可得到整个线段上的所有离散点集。而在起始像素位置 的第一个参数可从方程(2-12)及计算出：

对于斜率绝对值小于 1 的直线，其生成算法过程如下：

输入线的两个端点，并将左端点存贮在中；

将装入帧缓冲器，画第一个点；

计算常量：△x、△y、 2△y 和 2△y-2△x，按公式(2-19)并得到决策参数的第一

个值： p0；

(4). 从 k=0 开始，在沿线的每个 xk 处，进行下列检测：假如，画下一点，且按公式计算下一个决策参数值： ；否则，画下一点，且按公式(2-18)计算下一个决策参数值：。

(5) k=k+1；

(6) 回到步骤(4)，重复进行△x 次。

中点圆算法

原理

圆心在半径为 r 的圆可以用标准方程表示为：

也可以用极坐标方程表示为：

生成圆的直接方法是利用圆方程进行离散化。利用圆标准方程，可沿 x 轴以单位步长计算对应的y值来得到圆周上每个点的位置。这个方法的每一步包含很大的计算量，而且，所画像素位置间的间距是不一致的。利用圆参数极坐标方程可以消除上述不等间距现象，以固定角度为步长生成显示时，圆就可沿圆周等距点绘制出来；在沿圆周使用较大的角度作为步长时，可用直线段连接从而逼近圆的路径；为在光栅系统上得到更连续的边界，可将步长设为，这样绘制的像素位置大约为一个单位间隔；此外，考虑圆的对称性可以减少计算量：由于平面内的圆是关于八分之一象限对称的，因此，圆的八分之一圆弧上的一点可映射到xy平面的圆的其它七个八分之一圆弧上的点，也就是说，仅需计算从 x = 0 到 x = y 段内的点就可得到整个圆的所有像素位置。采用上述两种方法决定圆周上的像素位置都仍需要大量的计算时间。笛卡尔方程包括乘法和平方根运算，而参数方程中包含乘法和三角运算。更有效的圆算法是如Bresenham 画线算法一样以决策参数的增量计算为基础，通过设定在每一取样步寻找最接近圆周像素的决策参数而移植为画圆算法。然而，圆方程是非线性的，因此，计算像素与圆的距离必须进行平方根运算。可以通过比较像素与圆的距离的平方而避免了平方根运算，而更有效的方法是检验两像素间的中间位置，以确定这个中点是在圆边界内或外的方式来确定圆周上的像素，如下图所示。这种方法更易应用于其它圆锥曲线，且对于整数圆半径，中点方法生成与 Bresenham 圆算法相同的像素位置，而且使用中点检验时沿任何圆锥曲线段所确定的像素位置的误差限制在像素分段的 1/2 以内。像在 Bresenham 画线算法中一样，以单位间隔取样并在每个步长中确定离指定圆最近的像素位置。对于给定半径 r 和圆心在(xc, yc)处的圆，可先给出圆心在坐标原点（0，0）圆的像素位置的算法，然后通过将xc 加到每个像素位置的 x 坐标、将 yc 加到每个像素位置的 y 坐标就可得圆心在、半径为 r 的圆周上的屏幕像素位置(x, y)。  
在第一象限中，圆弧段从x=0到x=y，曲线的斜率从0变化到-1。因此，可以这八分  
圆上的正x方向取单位步长，并使用决策参数来确定每一步中两个可能的y位置中更接  
近于圆的位置，其它七个八分圆中的位置可由对称性得到。

为了应用中点法，定义圆函数：  
那么，任何点(x,y)的相对位置可由对圆函数符号的检测来决定：位于圆周边界外。  
, (x,y)位于圆周边界内

, (x,y)位于圆周边界上

, (x,y)位于圆周边界外

式中圆函数的检测在每个取样步上对接近圆周的两个像素的中点进行。因此，在中点算法中圆函数是决策参数，且可以像在画线算法中一样为这个函数设置增量运算。下图示出了取样位置 xk+1 处两候选像素间的中点，假设(xk , yk )为前一像素，下一步需要决定是像素位置，还是像素位置更接近于圆（注意：由于圆在第一象限，△y<0，所以 y 方向应取减量，即：。决策参数是圆函数在这两个候选像素的中点处求值：

假如 pk < 0 ，这个中点在圆内，扫描线 yk 上的像素（高像素）接近于圆边界；反之，假如 pk ≥ 0 ，中点位于圆外或在圆周边界上，选择扫描线 yk−1 的像素（低像素）。图 2-26 示出了取样位置 xk+2 处两候选像素间的中点。如果 pk < 0 ，取样位置 xk+1决定的像素位置是(xk+1, yk )，那么在取样位置 xk+2 处的两个候选像素分别是和，其中点位置是；如果 pk ≥ 0 ，取样位置 xk+1 决定的像素位置  
是， 那 么 在 取 样 位 置 xk+2 处 的 两 个 候 选 像 素 分 别 是和，其中点位置是。因此，确定第k+2步候选象素的决策参数可计算为：

,if pk < 0

,if pk >= 0

if pk < 0

if pk >= 0

在 坐 标 轴 上 圆 周 上 的 点 的 决 策 参 数 的 初 始 值 为 ：

生成算法过程如下

1. 输入圆半径r和圆心。
2. 确定圆心在原点的圆周上的第一点为：
3. 利用式计算决策参数的初始值 p0 ；
4. 在每个xk位置处，从k = 0开始，按公式确定第k步的候选像素，并按公式完成增量计算。
5. 确定在其它七个八分圆中的对称点。
6. 将每个像素位置移动到中心在的圆路径上，并画坐标值：
7. 重复步骤(4)到(6)，直止 x ≥ y 。

中点椭圆算法

原理

椭圆的曲线的生成可通过考虑椭圆沿长轴和短轴尺寸不同而修改画圆程序来实现。

椭圆被定义为到两个定点(焦点)的距离之和等于常数的点的集合。在任意方向指定一个椭圆的交互方法是输入两个焦点和一个椭圆边界上的点，利用这三个坐标位置，就可求出  
显式方程中的常数，而后就可求出隐式方程中的系数，并用来生成沿椭圆路径的像素。假如短轴和长轴与坐标轴方向平行，那么椭圆方程就可大大简化。一个“标准位置”椭圆是指其长轴和短轴平行于x和y轴，参数 rx 标出长半轴，参数 ry 标识短半轴。标准位置的椭圆在四分象限中是对称的，利用对称性可减少计算量，只需计算一个四分象限中椭圆曲线的像素位置，再由对称性得到其它三个象限中的像素位置。标准椭圆方程可借助于椭圆中心坐标(xc, yc )和参数 rx 和 ry 写为

也可以参数方程来描述标准位置的椭圆：

给定参数rx 和 ry 及椭圆中心(xc , yc )，先确定中心在原点的标准位置椭圆点(x,y)，  
然后将点变换为圆心在的点。

中点椭圆方法依据椭圆斜率( ry < rx )将第一象限的椭圆分成两部分，即区域 1 和区  
域 2，如下图。 两个区域的分割条件可通过检测曲线的斜率值而得到。椭圆的斜率可从椭圆方程中计算出：

在区域 1 和区域 2 的交界区：，由此可得，区域 1 和区域 2 的分割条件  
为：。或者说，移出区域 1 进入区域 2 的条件是：。

通过在斜率绝对值小于1的区域在x方向取单位步长，在斜率绝对值大于1的区域在y方向取单位步长来处理这个象限。区域1和区域2可以多种方式来处理：可以从位置(0,ry )开始，在第一象限内沿椭圆路径顺时针步进，当斜率变为小于-1 时从x方向的单位步长转化为y方向的单位步长；也可以从开始，以逆时针方式选取点，并当斜率为小于-1 时将 y 方向的单位步长改为 x 方向单位步长。

给定参数ry, rx和，先确定中心在原点的标准位置的椭圆点，然后将点变换为中心在的点。

假设，取，定义椭圆函数为：

该函数具有下列特性：

,(x,y)位于椭圆周边内

,(x,y)位于椭圆周边上

,(x,y)位于椭圆周边外

椭圆函数作为中点椭圆生成算法的决策参数。在每个取样位置，按照椭圆函数在沿椭圆轨迹两个候选像素间中点求值的符号选择下一个像素。

假设，从开始，在x方向取单位步长步进到区域1和区域2之间的边界，而后转换成y方向的单位步长通过第一象限中剩余的曲线段。

1. 区域 1 中(|切线斜率|≤1)

假如在前一步中选择了位置，将第一象限内取样位置处两个候选像素间中点对决策参数(即椭圆函数)求值：

假如 p1k < 0 ，中点位于椭圆内，扫描线yk上的像素更接近于椭圆边界；否则，中点在椭圆之外，或在椭圆边界上，所选的像素应在扫描线上。

, if pk < 0;

, if pk >= 0;

1. 区域 2 中(|切线斜率|＞1)

当进入区域 2 时，其初始点取区域1中选择的最后位置。在区域2中，在负y方向以单位步长取样，在每一步中取水平像素间的中点对决策参数求值为：

假如 p2k > 0 ，中点位于椭圆边界之外，选择像素xk；假如p2k ≤ 0，中点位于椭圆边界之内或之上，选择像素xk+1。

, if pk <= 0;

, if pk > 0;

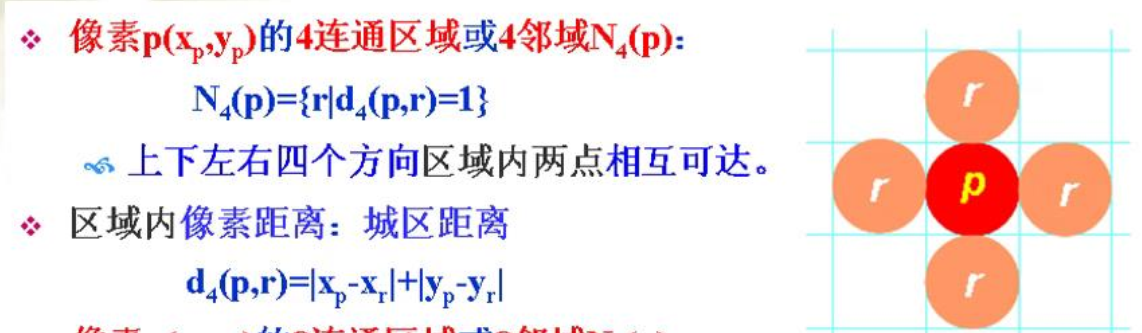
概括起来，对于中心在、长短轴径分别为rx和ry的椭圆，其生成过程如下：

1. 输入中心和长短轴径为 rx 和 ry ；
2. 计算得到中心在原点的椭圆上的第一个点：；
3. 按公式计算区域1中决策参数的初值为：p10 ；
4. 在区域1中每个xk位置处，从k = 0开始，反复按照公式确定候选像素，并进行决策参数增量计算；循环至：；
5. 使用区域1中最后点作为区域2的起始点(x0, y0)来计算区域 2 中参数初值
6. 在区域2的每个yk位置处，从 k = 0开始，反复按照公式确定候选像素，并进行决策参数增量计算；循环至。
7. 确定其它三个像限中对称的点；
8. 将每个计算出的像素位置(x, y)平移到中心在(xc , yc )的椭圆轨迹上，并按平移  
   坐标值画点。

多边形填充算法

在进行多边形填充时，我采用泛滥填充算法。

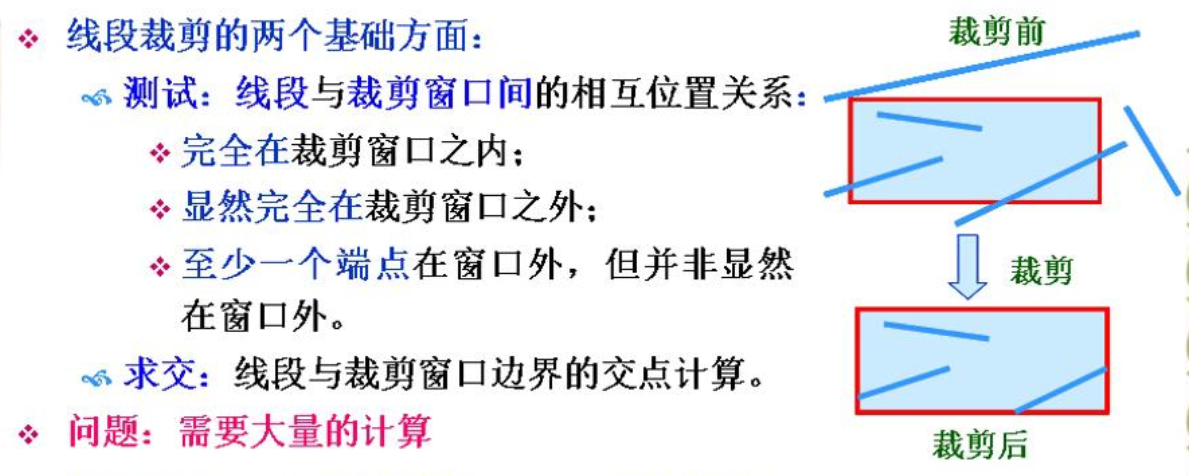
以当前选中点为种子点，并按照4邻域进行扩展。采用BFS的方法进行填充。4邻域含义的示意图如下：



裁剪算法

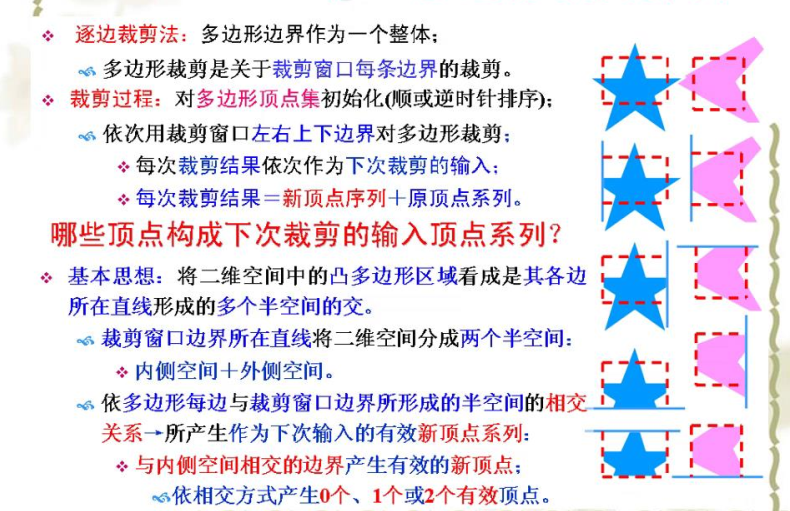
线段裁剪

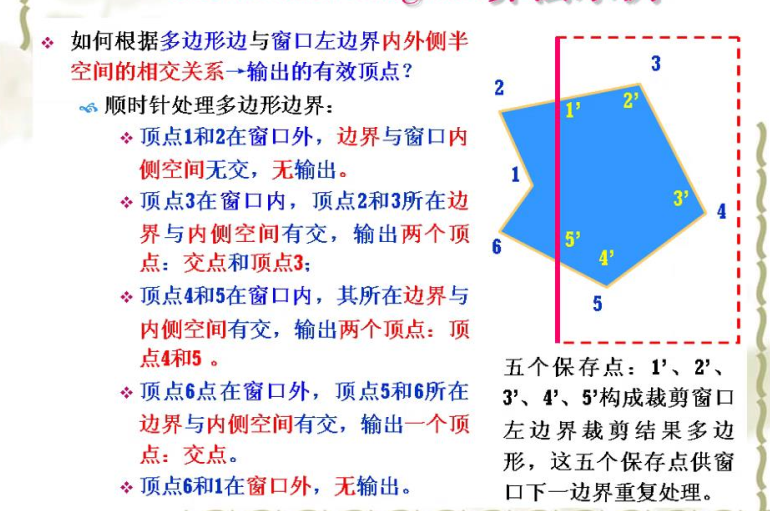
对于线段裁剪，我选择了最简单的测试求交算法，即先通过测试，判断线段与矩形裁剪框的关系，再求交计算线段与矩形裁剪框的交点。

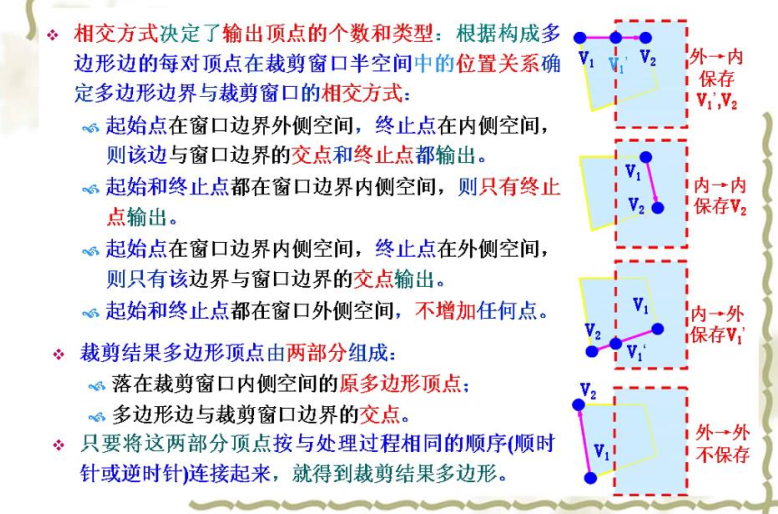


多边形裁剪算法

对于多边形裁剪，我选择了Sutherland-Hodgman算法。算法原理如下：

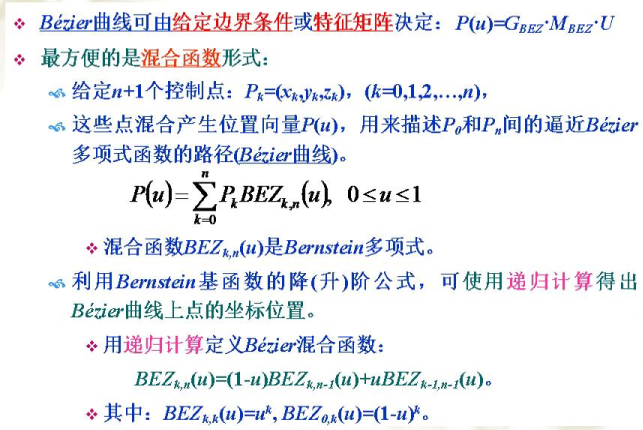




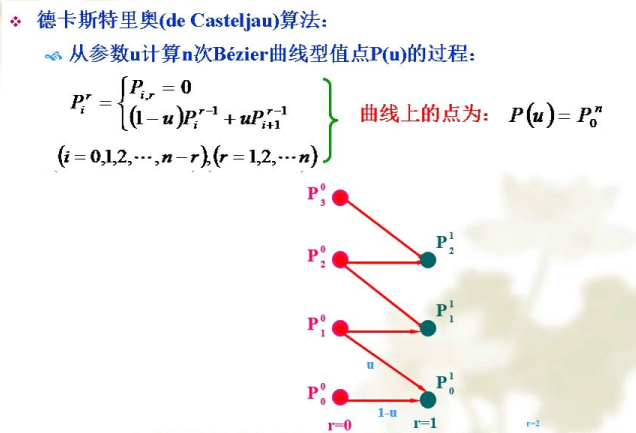


Bezier曲线

Bezier曲线的定义如下：

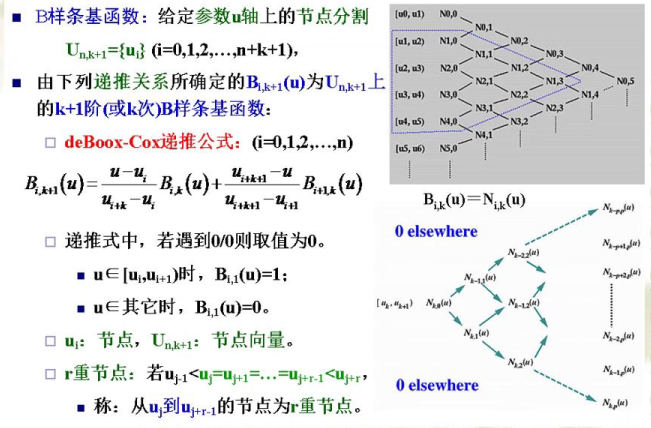


对于Bezier曲线离散点生成，我选用的是de Casteljau算法。算法描述如下：



B样条曲线

B样条基函数的定义：



在绘制过程中，我选用了三次均匀B样条曲线，说明如下：

