《CG2018》系统技术报告

郑来栋

(南京大学 计算机科学与技术系, 南京 210093)

摘 要: 系统使用C++语言，基于QT和OpenGL，实现了一个交互式的绘图系统。实现了图形的绘制，填充，旋转，平移，缩放以及裁剪，三维模型的显示，实现了将图形保存为BMP格式的文件。

# 引言

引言

# 算法描述

## 图形绘制

### 直线绘制-Bresenham画线算法

假设线段l = , 绘制从P1(x1, y1)到P2(x2, y2), l直线方程记为

先考虑0＜ m ＜ 1， 且x1 < x2的情况，其余情况可根据对称性做一些提前处理得到。在Δx >= Δy的情况下，应在x方向进行取样，并在y方向上最接近线段的像素上绘出一点。

A

B

C

如上图所示，假设在绘制一系列点之后，在A()绘出一点，下一步需要确定在时绘制在

B()还是C(), 也就是决定B和C点哪一点离直线更近。

#### 推到决策参数,确定离线段最近的候选点

设线段l与BC连线交于D, 记，下面将用坐标表示出来。

D点的y坐标可由直线方程得到

故，

因此B和C离线段的竖直距离差|BD| - |DC|

=

记

由于, 化简可得

由于dx > 0, 所以与同号，也就是

所以，把上面的记为决策参数，用来决定每一步中离线段更近的候选点。

#### 增量式计算,消除代价高昂的乘法运算

当时，同样可得

则

而, 所以

而计算开始时，初始的决策参数, 这样可以只通过加减运算高效地计算决策参数

#### 处理所有情况

1. |m| ≤1时，在x方向取样，|m|＞1时，在y方向取样
2. 为了使的符号与相同，需保证dx(或dy)为正值
3. 使用一个变量来记录取样方向是正方向还是负方向

dx =2( x2 – x1), dy = 2(y2-y1)

stepx = stepy = 1 //采样方向的增量

if(dx < 0) {dx = -dx, stepx = -1}

if(dy < 0) {dy = -dy, stepy = -1}

plot(x0, y0)

if(dx > dy){

p = dy – dx >> 1

while(x0 != x1){

x0 += stepx

if(p >= 0){y0+=stepy; p -= dx}

p += dy

plot(x0, y0)

}

}else{

p = dx – dy >> 1

while(y0 != y1){

if(p>=0){x0 += stepx; p-=dy}

y0+=stepy

p+=dx

plot(x0, y0)

}

}

### 圆绘制-中点圆算法

对于给定半径r和圆心(xc, yc)，可以先计算出圆心在原点(0, 0)的圆的像素位置，然后通过平移得到最终的像素位置。

考虑到圆的对称性，我们只需计算出在x = 0和x = y两条直线之间的坐标(一个八分圆)，通过对称性就可以得到所有的像素位置。

类似于直线绘制过程中，我们使用决策参数来确定候选点()和中，哪一点离圆最近，决策参数p的递推公式为

初始决策参数

这样，可以根据的值决定选择还是

绘制过程伪代码如下

(1)计算p0 = 1-r

(2)plot(x0, y0)

(3)在x方向进行取样，对每个，若, 则下一个点为

且

否则下一个点是

且

plot()

其中，子程序plot用于根据对称性，对每个八分圆上的点进行绘制，且进行平移得到最终像素坐标。

plot(x, y)

{

draw(xc+x, yc+y) draw(xc+x,yc-y)

draw(xc-x, yc+y) draw(xc-x,yc-y)

draw(xc+y,yc+x) draw(xc-y, yc+x)

draw(xc+y, yc-x) draw(xc-y, yc-x)

}

### 椭圆绘制-中点椭圆算法

类似于中点圆算法，将椭圆平移到原点(0，0)，计算出像素坐标，然后再平移回去。且考虑对称性，只计算第一象限的像素即可。

依据椭圆切线斜率对第一象限进行划分，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区域在y方向取单位步长。

1

2

k=-1

在区域1，决策参数的递推式如下

初始值

且根据的符号，可以确定下一个y的值

在区域2，决策参数的递推式如下

初始值

且根据的符号，可以确定下一个y的值

且为了加快计算速度，应提前计算平方项，作为常数使用。绘制过程类似于中点圆算法，分开对区域1和区域2进行处理.

### 矩形和多边形

矩形多多边形的绘制是利用直线绘制完成的，系统存储其顶点，绘制时，在顶点之间按序绘制线段即可。不同点是它们的表示不同，矩形采用左上角和右下角表示(这就造成了系统只能表示标准矩形，旋转时只能旋转90度的倍数)，而多边形保存顶点表。

### Bezier曲线

给定n+1个控制点，, 0≤k≤n，输出n阶贝塞尔曲线上的点。

曲线上点的坐标可以表示为

其中，, 如果直接计算该式子，计算量将相当大，我们可以采用迭代式地计算

## 图形填充

### 圆的填充

采用扫描填充，直接处理，对每条平行x轴的扫描线，计算其与圆两侧的交点，填充交点之间的线段即可。扫描线y与圆两侧的交点可用勾股定理求出。

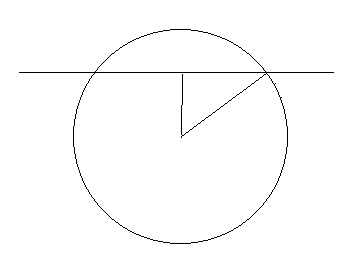
B

A

C()

扫描线y

则 , 因此，扫描线y上地填充范围为[]



center: P(cx, cy)

radius: r

for y from cy to cy + r – 1:

dy = y – cy

dx = sqrt()

left = cx – dx

right = cx + dx

for x from left + 1 to right – 1:

fillPoint(x, y)

fillPoint(x, 2 \* cy - y)

### 矩形填充

矩形填充比较简单，直接对四条边内部区域进行填充即可

for x from xmin to xmax:

for y from ymin to ymax:

fillPoint(x, y)

### 一般多边形的填充(扫描填充算法)

通过构建有序边表和活化边表，求出各条扫描线上的填充范围，实现了多边形的扫描填充算法。

边表的数据结构如下：

struct Edge

{

double x; //起始点横坐标

double dx; //y每增加1，x地增加量

int y\_upper; //该边地最高点

}

主要步骤如下，

1. 生成有序边表

扫描多边形中所有边，若是非水平边，计算相应的x，dx和y\_upper填入有序边表中

1. 用有序边表构造活化边表

扫描当前扫描线及以下的有序边表，将y\_upper大于等于当前y值的Edge加入到本扫描线对应的活化边表中，其中起始点横坐标需要重新计算，然后根据x对活化边表从小到大排序

1. 根据活化边表计算各扫描线填充范围

遍历活化边表中的Edge项，两两配对，对中间区域进行填充

## 几何变换

实现基本的几何变换，包括平移、旋转和缩放，只需实现对点的变换，其余几何图形可根据几何特征基于点进行变换。

设点P(x, y) ,

### 平移

设平移向量为t = (), 则P平移后的坐标为

基于点的平移，实现基本几何形状的平移

|  |  |
| --- | --- |
| 几何图形 | 平移方式 |
| 直线 | 分别平移线段端点 |
| 圆 | 平移圆心 |
| 椭圆 | 平移椭圆外接矩形左上角和右下角端点 |
| 矩形 | 平移左上角和右下角 |
| 多边形 | 分别平移每个顶点 |
| 曲线 | 分别平移每个控制点 |

### 旋转

设旋转基准点为C = (), 旋转角为α，则P旋转后的坐标可表示为

基于点的旋转，实现基本几何形状的旋转

|  |  |  |
| --- | --- | --- |
| 几何图形 | 旋转中心 | 旋转方式 |
| 直线 | 线段中点 | 分别旋转线段端点 |
| 圆 | -- | do nothing |
| 椭圆 | 椭圆中心 | 旋转椭圆外接矩形的顶点 |
| 矩形 | 矩形中心 | 旋转左上角和右下角的顶点 |
| 多边形 | 所有顶点的中心(取平均) | 分别旋转每个顶点 |
| 曲线 | 所有控制点的中心(取平均) | 分别旋转每个控制点 |

### 缩放

设缩放基准点为C = (), 缩放系数大小为s，则P旋转后的坐标可表示为

基于控制点的缩放也可以实现基本几何形状的缩放，其基准点的选取和旋转一样，都是选择“中心”

## 裁剪

设裁剪窗口是一个标准位置的矩形

W = {(x, y) | xmin ≤ x ≤ xmax, ymin ≤ y ≤ ymax}, 且舍弃裁剪窗口外的图形部分。

### 线段裁剪（梁友栋-Barsky算法）

设线段P = P1P2 ，端点分别是P1(x1, y1), P2(x2, y2)，其参数方程可以写成如下形式：

其中

如果点P(x, y)位于W内，则应满足

,

将该公式变形做统一之后记为

可得

我们可将这四种情况对应到裁剪窗口的四条边，对于pk和qk, k = 1, 2, 3, 4分别对应W的左、右、下、上边界, 从上式可以有如下观察

1. 线段平行于裁剪边界k(实际上是两条)

* 若, 线段在裁剪窗口外
* 若, 线段平行于裁剪边界且在窗口内
* 线段从外到内穿过裁剪边界第k侧延长线
* 线段从内到外穿过裁剪边界第k侧延长线

裁剪过程的伪代码如下

计算pk, qk

u0 = 0, u1 = 1

for i from 1 to 4:

if pi == 0:

if qi < 0:

return //在裁剪窗口外，直接舍弃

else

r = qi / pi //r表示直接与第i边的交点的参数值

if pi < 0

u1 = max(u1, r)

else

u2 = min(u2, r)

if u1 > u2:

return

update:

//保留参数在[u1, u2]内的部分

P1 = (x1 + u1\* Δx, y1 + u1\*Δy)

P2 = (x1 + u2\* Δx, y1 + u2\*Δy)

算法 梁友栋-Barsky

### 矩形裁剪

矩形裁剪较简单，我们只需要求出待裁剪矩形与裁剪窗口W的重叠部分即可

设待裁剪矩形左上角坐标为P1(x1, y1), 右下角坐标为P2(x2, y2)

先假设两个矩形相交，相交后的矩形为C, 设C的左上角坐标为(xc1, yc1), 右下角坐标为(xc2, yc2), 则其顶点坐标可通过如下式子计算出来

最后判断不等式 是否满足即可判断出是否有交集，如果相交，则重设带裁剪矩形左上角和右下角的坐标。

### 一般多边形裁剪(Sutherland-Hodgeman算法)

依次用裁剪窗口的四条边进行裁剪，每一次裁剪过程中，依次检查多边形的每条边，根据边与裁剪窗口对应当前用于裁剪的边界的位置关系，决定将哪些点加入到新顶点集中。

设当前用裁剪窗口的边Ei (0≤i≤3分别对应上、右、下和左边界), 当前检查边E = P1P2, P1为起点，P2为终点(因为检查过程是有序的，所以边必须有方向性)。设Ei和E的交点为V(若有)

1. P1在Ei外侧空间，P2在Ei内侧空间

依次添加V和P2

1. P1和P2都在内侧空间

添加P2

1. P1在E内侧空间，P2在Ei外侧空间

添加V

1. P1和P2都在外侧空间

不添加

输入: vertex //多边形顶点

W(xmin, ymin, xmax, ymax) // 裁剪窗口

initialise cut\_edges // cut\_edges是裁剪窗口的上，右，下，左四条边

for i in 0 to 3 do:

vertex\_temp = {}

Ei = cut\_edges[]

for j in 0 to vertex.size – 2 do:

start = vertex[j], end = vertex[j+1]

E = Line(start, end)

if outside(start, Ei)

if !outside(end, Ei)

V = intersect(E, Ei) //求交点坐标

vertex\_temp.add(V)

vertex\_temp.add(end)

else

if outside(end, Ei)

V = intersect(E, Ei) //求交点坐标

vertex\_temp.add(V)

else

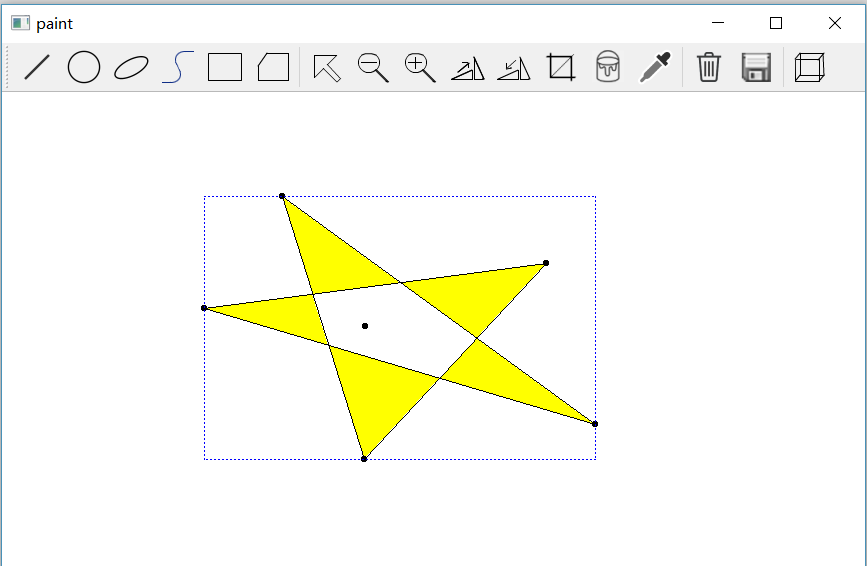
vertex\_temp.add(end)

vertex = vertex\_temp //作为下一轮的输入

vertex\_temp = {}

# 系统框架

## 用户界面



用户界面主要包括两大部分，一是顶部的TooBar，用于实现图形输入，编辑等菜单选项，改变系统状态。二是中间的绘图区，继承自QOpenGLWidget，用于处理绘制图形，处理鼠标事件等。

## 模块划分及类层次结构

### 图形类

IShape

Poly

Curve

Oval

Rect

Circle

Line

CutBox

1. 所有的图形类都继承自IShape，IShape实现了一些基本方法，且定义了一套接口，如draw()，translate()，rotate()等。
2. 利用C++的多态性，绘制系统不需要知道图形的种类然后分情况绘制。
3. CutBox类继承自Rect类，用于裁剪窗口的绘制
4. Curve类实现的是Brazier曲线

### 绘制类

**FigureGenerator**

**LineGenerator**

**PolygonGenerator**

**CurveGenerator**

**OvalGenerator**

**RectangleGenerator**

**CircleGenerator**

1. 绘制类均继承自FigureGenerator，用于实现图形的绘制功能
2. 图形输入功能来自于鼠标事件，依据当前状态和鼠标事件进行不同的动作, 下表是对不同图形，不同的鼠标事件应当采取的动作

|  |  |  |  |
| --- | --- | --- | --- |
|  | MousePress | MouseMove | MouseRealse |
| 直线 | 以点击点为起点和终点创建一条直线 | 更新终点 |  |
| 圆 | 以点击点为圆心，半径为1创建一个圆 | 重新计算并设置半径 |  |
| 椭圆 | 创建一个椭圆，以点击点的坐标初始化其外接矩形 | 重新设置外接矩形的一个顶点 | 绘制完成 |
| 矩形 | 以点击点为起点和终点创建一个矩形 | 重新设置一个顶点 |  |
| 多边形 | 以点击坐标添加一个顶点，如果多边形已封闭则结束绘制 | 更新最后一个顶点坐标 | 更新最后一个顶点坐标(跟踪鼠标) |
| 曲线 | 添加一个控制点 | 不做操作 | 不做操作 |

其中，

1. 对于多边形的绘制，由于不确定用户要输入多少顶点，因此当用户点击时就需要添加顶点，当多边形封闭时标志绘制结束(最后一个点和第一个点重合)
2. 对于曲线的绘制，同样需要点击时添加控制点，当用户再次点击ToolBar的曲线绘制按钮时，一条曲线绘制结束。
3. 如果这些功能在一个类里实现，将有很大的复杂度，难以控制，所以将每个图形的绘制抽象出一个类来实现，它们都能处理鼠标事件，这样设计每一个类结构简单容易控制。

### 图形编辑类

编辑类主要用于处理图形选中情况下的图形编辑，如修改顶点位置，平移，拖动矩形边长等等。

**Editor**

**LineEditor**

**PolyEditor**

**CurveEditor**

**OvalEditor**

**RectEditor**

**CircleEditor**

系统在鼠标点击时，或是在图形选择模式下，会首先检测点击点是否在已选中图形上(如果有)，如果没有，再次检测是否选中了新图形(点击了另一个图形上的点)。这些类重载了关于鼠标的一些事件函数，每个类又分别管理一类图形的编辑。这里的编辑主要事件如下表

|  |  |
| --- | --- |
| 图形 | 事件 |
| 直线 | 修改端点 |
| 圆 | 通过拖动圆上的点修改其半径 |
| 矩形 | 修改四个顶点的坐标 |
| 多边形 | 修改顶点的位置 |
| 椭圆 | 通过修改其外接矩形的四个顶点修改其大小 |
| 曲线 | 修改控制点的位置 |

### 工具栏

工具栏包含系统的所有功能，包括绘图，编辑，选中，旋转，填充，保存，打开三维模型等



1

2

3

4

5

6

7

8

9

10

11

12

13

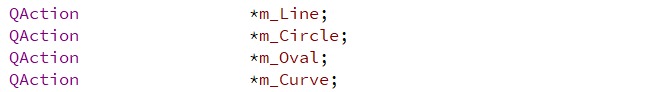
14

15

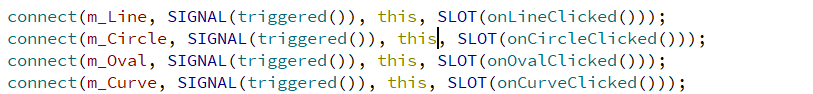
16

17

ToolBar是一个自定义的工具栏类，它继承自QToolBar类，其中每个工具对应一个QAction成员：



每个QAction通过信号槽机制连接到一个响应函数：



这些响应函数是在ToolBar类中定义的槽函数：



在这些响应函数中做相应动作即可。

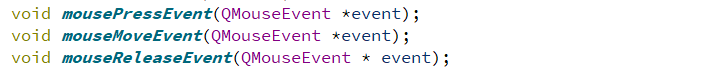
### **PaintWidget和GL3Dwidget**

我们的绘图任务是在这两个类中完成的，分别负责2D和3D绘图，它们均继承自QOpenGLWidget和QOpenGLFunctions，重载了***initializeGL、paintGL和resizeGL***三个函数用来实现绘制，每次需要重绘的时候，调用它们的update函数即可重新绘制。

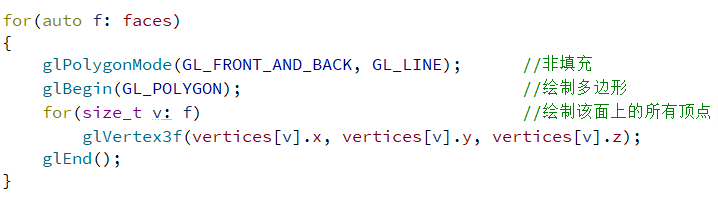
1. PaintWidget用来绘制2D图形，所有的绘制最终都调用点的绘制函数来实现



1. PaintWidget重载了一些鼠标事件函数，并负责将这些事件分发到对应的处理者，比如负责绘图的，编辑的等。



1. **GL3Dwidget**中绘制三维模型时，各个面分别绘制，每个面由一个多边形围成



### 图形保存功能

系统实现了将图形保存为BMP文件格式，其中有两个主要任务，一是填充文件头部，二是将图形像素填充到文件对应的位置。如果当前有图形处于选中状态，只保存被选中的图形，否则，保存画板上所有的图形。

### 图形删除

删除当前处于选中状态的图形。

# 软件测试

## 绘制

### 直线

在各种斜率情况下，都能正确绘制直线，效果如下

|  |  |  |
| --- | --- | --- |
| 0 < k < 1 | k >= 1 | k 不存在 |
| -1 < k < 0 | k <= -1 | k = 0 |

### 圆

|  |  |  |
| --- | --- | --- |
| r较小 | r较大 | 圆一部分超出屏幕范围 |

### 椭圆

|  |  |  |
| --- | --- | --- |
| rx > ry | rx 接近 ry | rx < ry |

### 曲线

对不同的控制点个数

|  |  |  |
| --- | --- | --- |
| 2 | 3 | 4 |
| 4 | 4 | 5 |

### 矩形

各种形状的矩形

|  |  |  |
| --- | --- | --- |
|  |  |  |

### 多边形

绘制三个顶点及以上的各种多边形

|  |  |  |
| --- | --- | --- |
| 凸多边形 | 凹多边形 | 五角星 |

## 填充

### 圆的填充

|  |  |  |
| --- | --- | --- |
| 填充前 | 填充后 | 更改填充色 |

### 矩形的填充

|  |  |
| --- | --- |
| 填充前 | 填充后 |

### 多边形填充

|  |  |  |
| --- | --- | --- |
| 描述 | 填充前 | 填充后 |
| 凸多边形填充 |  |  |
| 凹多边形填充 |  |  |
| 任意多边形（没有进行内外测试） |  |  |

## 裁剪(内裁剪)

### 线段裁剪

根据线段与裁剪窗口的位置关系进行测试

|  |  |  |
| --- | --- | --- |
| 描述 | 裁剪前 | 裁剪后 |
| 线段穿过裁剪窗口，两个顶点均不在裁剪窗口内部 |  |  |
| 线段一点在裁剪窗口内 |  |  |
| 线段完全在裁剪窗口内 |  |  |
| 线段在裁剪窗口外，这种情况下，虽为外裁剪，但作为交互性的应用，可以保留不与裁剪窗口相交的图形 |  |  |

### 矩形裁剪

|  |  |  |
| --- | --- | --- |
| 描述 | 裁剪前 | 裁剪后 |
| 矩形与裁剪窗口部分相交 |  |  |
| 裁剪窗口在矩形内部 |  |  |

矩形完全在裁剪窗口内部以及外部的情况，和线段裁剪类似

### 多边形裁剪

|  |  |  |
| --- | --- | --- |
| 描述 | 裁剪前 | 裁剪后 |
| 凸多边形裁剪 |  |  |
| 凹多边形裁剪 |  |  |
| 凹多边形  (裁剪后有多余边) |  |  |
| 复杂多边形 |  |  |

### 填充区域

填充区域只支持多边形区域以及矩形区域的裁剪，是在对多边形裁剪后重新填充实现的。

## 平移（填充区域平移之后重新填充）

|  |  |  |
| --- | --- | --- |
| 直线 |  |  |
| 圆 |  |  |
| 椭圆 |  |  |
| 曲线 |  |  |
| 多边形 |  |  |
| 矩形 |  |  |

## 旋转（以图形中心为旋转中心）

|  |  |  |
| --- | --- | --- |
| 直线 |  |  |
| 圆 |  | 无需操作 |
| 椭圆 |  |  |
| 曲线 |  |  |
| 多边形 |  |  |
| 矩形 |  |  |

## 缩放(填充区域的缩放：缩放之后重新填充)

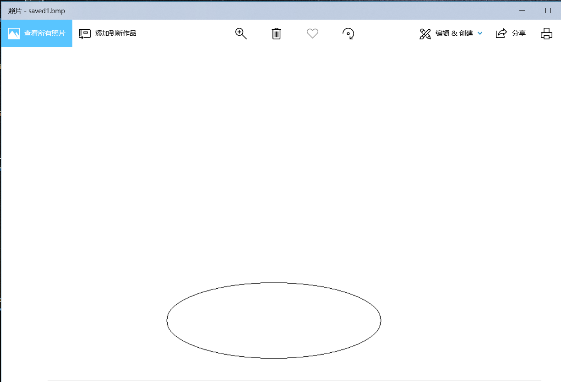
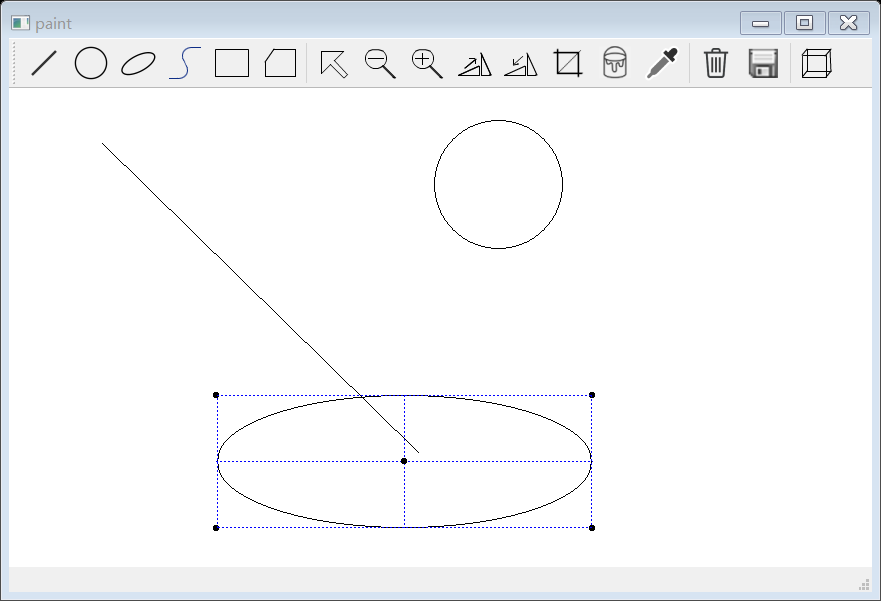
|  |  |  |
| --- | --- | --- |
| 直线（缩小） |  |  |
| 圆 |  |  |
| 椭圆 |  |  |
| 曲线 |  |  |
| 多边形 |  |  |
| 矩形 |  |  |

## 图形编辑

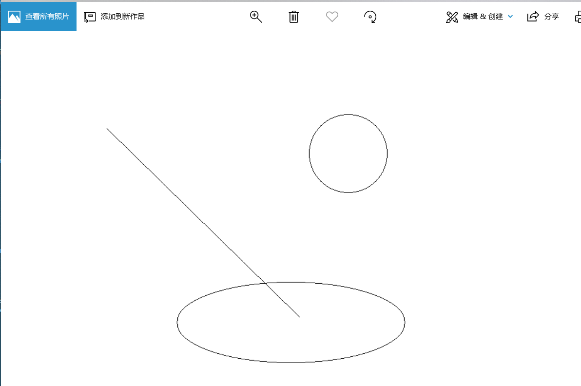
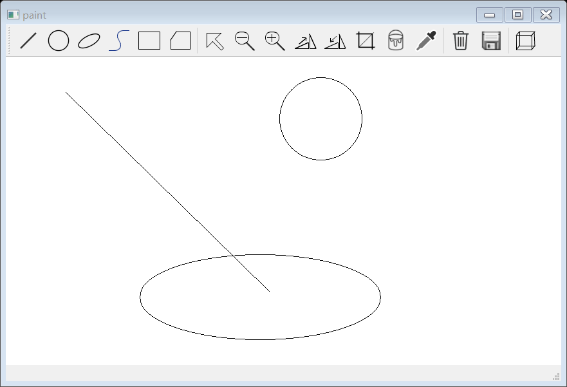
|  |  |  |  |
| --- | --- | --- | --- |
| 图形 | 编辑前 | 操作 | 编辑后 |
| 直线 |  | 拖动右下端点 |  |
| 圆 |  | 拖动圆上的点 |  |
| 椭圆 |  | 拖动编辑框左上角（四个角可编辑） |  |
| 矩形 |  | 拖动右下角（可编辑四个顶点） |  |
| 曲线 |  | 拖动其中一个控制点 |  |
| 多边形 |  | 拖动其中一个顶点 |  |

## 图形保存

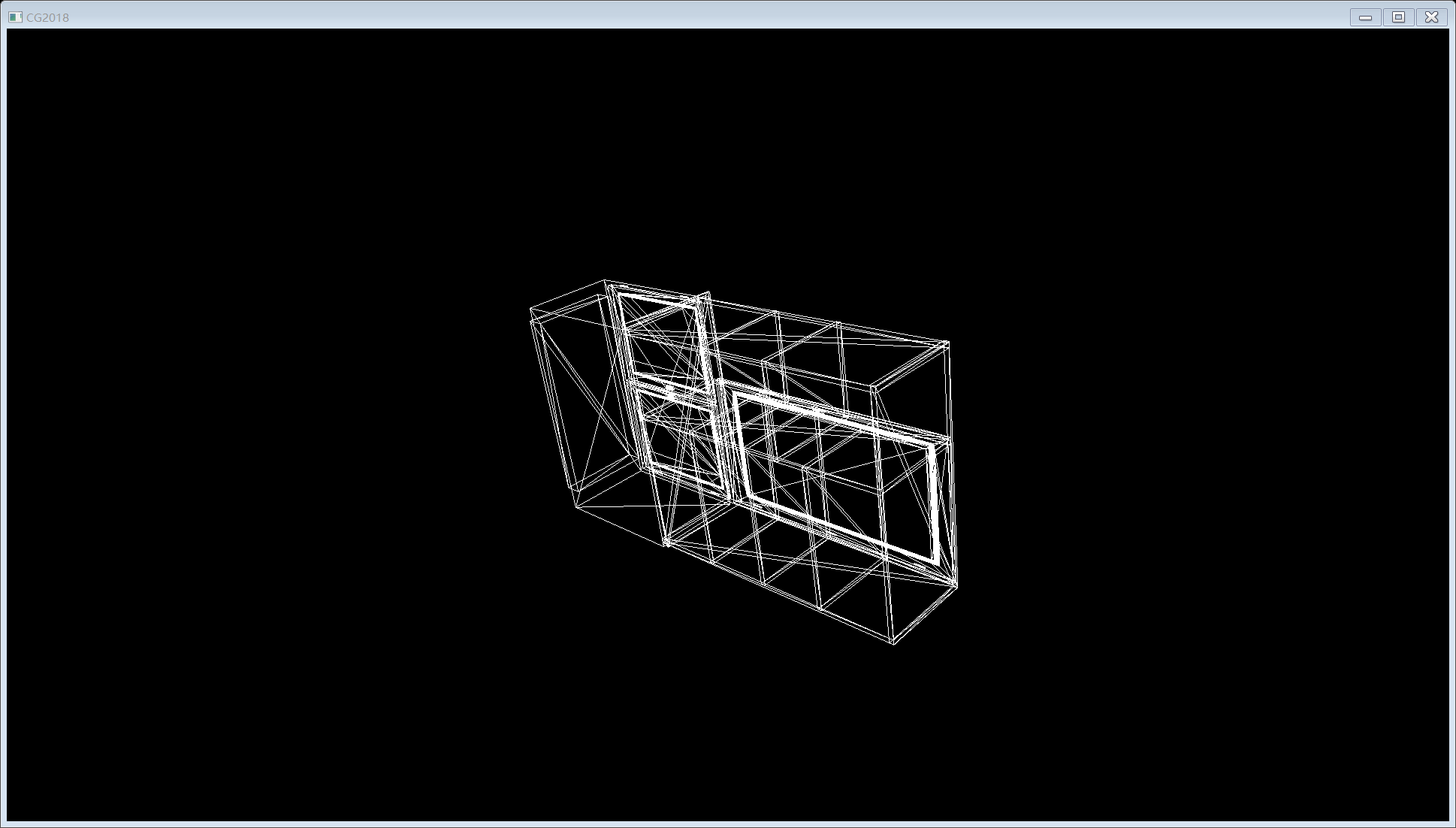
### 有图形选中的情况下，只保存选中的图形

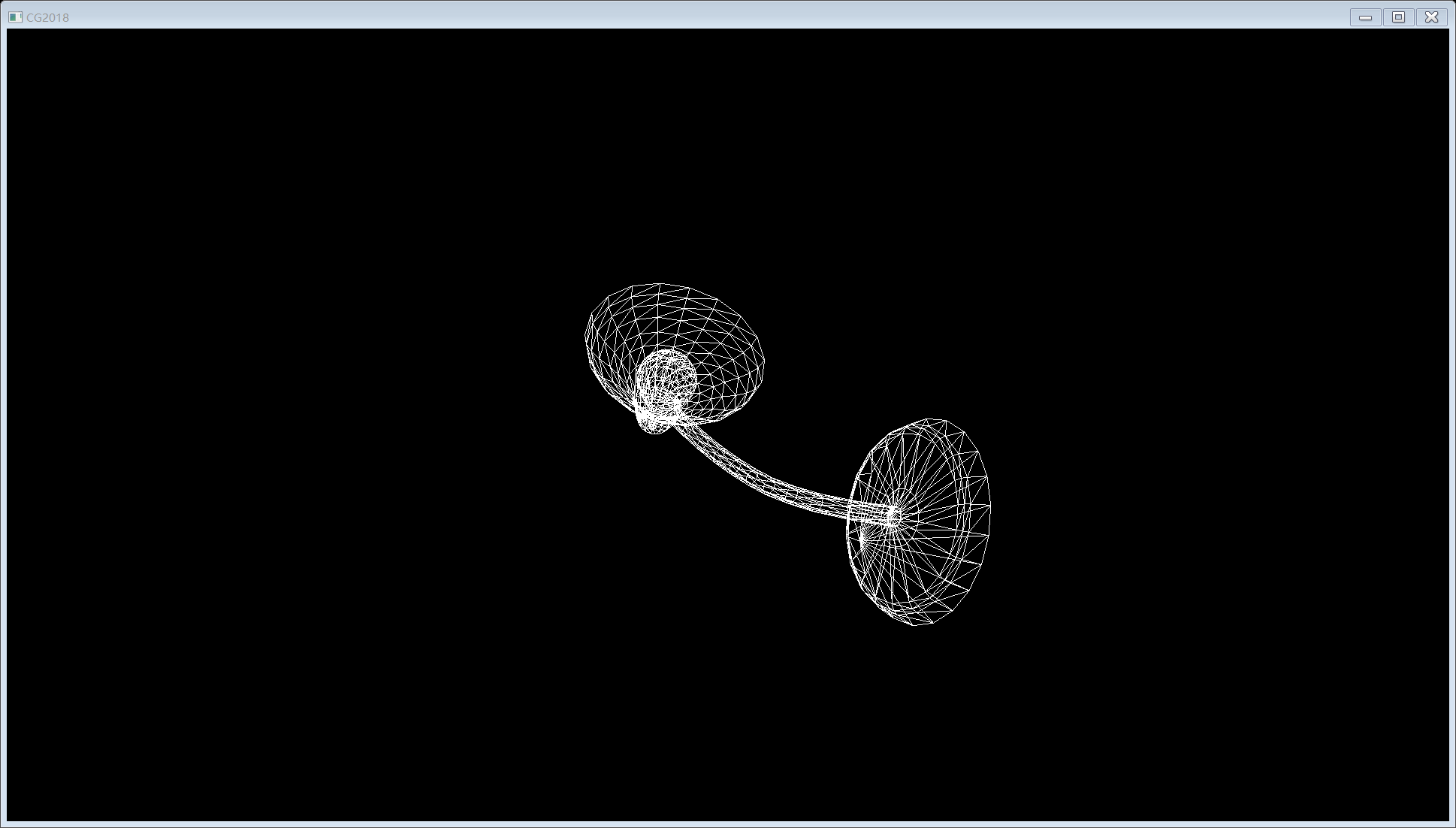


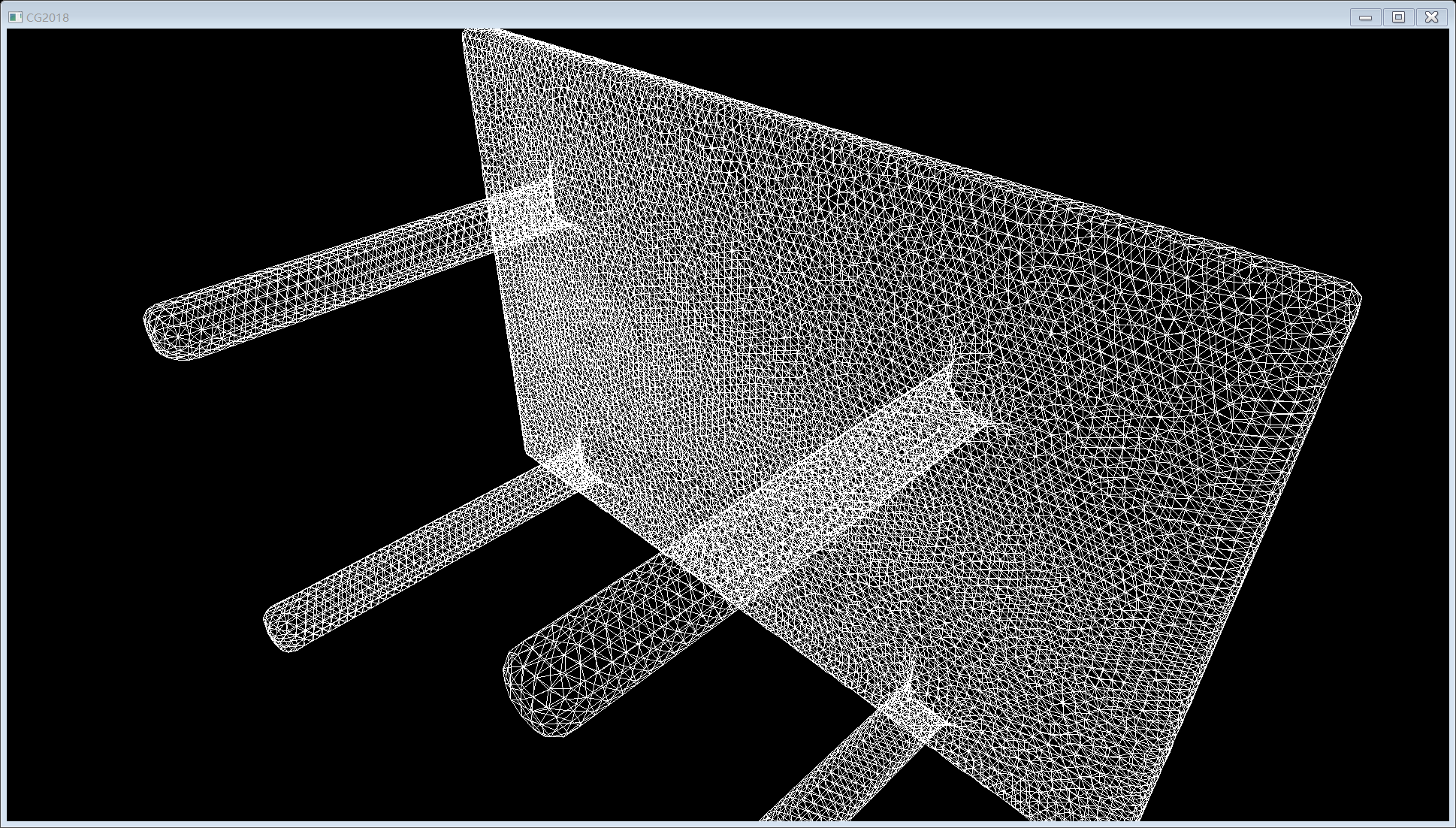
### 若没有图形选中，保存所有图形



## 三维模型显示







References:

1. Donald Hearn，M.Pauline Baker 《Computer Graphics》
2. QT documentation https://doc.qt.io/