算法题两道 40分 和实验相关的算法(老师给的算法)

1. 实验一：扫描转换

> 图形的生成：是在指定的输出设备上，根据坐标描述构造二维几何图形。
> 图形的扫描转换：在光栅显示器等数字设备上确定一个最佳逼近于图形的象素集的过程。

1. 数值微分法DDA算法：

> $x_{i+1}=x_i+\varepsilon \cdot dx$
>
> $y_{i+1}=y_i+\varepsilon \cdot dy$
>
> $\varepsilon=1/max(|dx|,|dy|)$

2. 中点Bresenhan算法：

> $d=F(x_M,y_M)=F(x_i+1,y_i+0.5)=y_i+0.5-k(x_i+1)-b$
>
> $x_{i+1}=x_i+1$
>
> $x_{i+1}=y_i+(d<0)$

3. 改进的Bresenhan算法：

> 改进：用2d△x代替d ,令D＝2d△x
> D=2△xd=2△x(0.5-k)=△x-2△y(待补充)

4. 代码 onMidPointEllispe（中点Bresenham算法）

```
//onMidPointEllipse.h
#include <vector> // 引入标准向量模板
#include <windows.h> // 引入Windows头文件，用于OpenGL窗口操作
#include <GL/glut.h> // 引入OpenGL Utility Toolkit头文件，用于OpenGL功能
using namespace std; // 使用标准命名空间

// 定义一个结构体Point来存储点的坐标
typedef struct Point {
    int x, y;
```

```
    // 构造函数，默认x和y坐标为0
    Point(int a = 0, int b = 0) {
        x = a, y = b;
    }
} point;

// 声明函数onMidPointEllispe，它将用于绘制椭圆
void onMidPointEllispe(int a, int b);
```

```
#include "onMidPointEllispe.h"
/************************************************
* 已知椭圆的长短轴a,b
* 根据椭圆的中点Bresenham算法，扫描转换椭圆
* 以glBegin(GL_POINTS);glEnd(); 方式，绘制椭圆
************************************************/
void onMidPointEllispe(int a, int b) {
    int x, y; // 定义两个整型变量x和y，用于存储当前点的坐标
    float d1, d2; // 定义两个浮点变量d1和d2，用于存储椭圆判别式

    glBegin(GL_POINTS); // 开始绘制点模式

    x = 0; y = b; // 初始点在椭圆的上半部分最右侧点
    d1 = b * b + a * a * (-b + 0.25); // 计算判别式d1，b^2 + (0.25 - b)*a^2
    glVertex2i(x, y); // 绘制当前点
    glVertex2i(-x, -y); // 绘制当前点的对称点
    glVertex2i(-x, y); // 绘制当前点的对称点
    glVertex2i(x, -y); // 绘制当前点的对称点

    while (b * b * (x + 1) < a * a * (y - 0.5)) { // 当当前点在椭圆上半部分时
        if (d1 <= 0) { // 如果当前点在椭圆内部
            d1 += b * b * (2 * x + 3); // 更新判别式d1
            x++; // 移动到下一个x坐标
        } else { // 如果当前点在椭圆外部
            d1 += b * b * (2 * x + 3) + a * a * (-2 * y + 2); // 更新判别式d1
            x++; // 移动到下一个x坐标
            y--; // 移动到下一个y坐标
        }
        glVertex2f(x, y); // 绘制当前点
        glVertex2f(-x, -y); // 绘制当前点的对称点
        glVertex2f(-x, y); // 绘制当前点的对称点
```

```
32          glVertex2f(x, -y); // 绘制当前点的对称点
33      } /* while上半部分 */
34
35      d2 = b * b * (x + 0.5) * (x + 0.5) + a * a * (y - 1)
    * (y - 1) - a * a * b * b; // 计算判别式d2
36      while (y > 0) { // 当当前点在椭圆下半部分时
37          if (d2 ⩽ 0) { // 如果当前点在椭圆内部
38              d2 += b * b * (2 * x + 2) + a * a * (-2 * y
    + 3); // 更新判别式d2
39              x++; // 移动到下一个x坐标
40              y--; // 移动到下一个y坐标
41          } else { // 如果当前点在椭圆外部
42              d2 += a * a * (-2 * y + 3); // 更新判别式d2
43              y--; // 移动到下一个y坐标
44          }
45          glVertex2f(x, y); // 绘制当前点
46          glVertex2f(-x, -y); // 绘制当前点的对称点
47          glVertex2f(-x, y); // 绘制当前点的对称点
48          glVertex2f(x, -y); // 绘制当前点的对称点
49      }
50
51      glEnd(); // 结束绘制点模式
52  }
```

2. 实验二：直线裁剪

   1. Cohen-Sutherland算法

      对于任一端点(x,y)，根据其坐标所在的区域，赋予一个4位的二进制码D3D2D1D0。

      （1）若x<wxl，D0=1，否则D0=0；

      （2）若x>wxr，D1=1，否则D1=0；

      （3）若y<wyb，D2=1，否则D2=0；

      （4）若y>wyt，D3=1，否则D3=0。

      a.若code1|code2=0，对直线段简取之；

      b.若code1&code2≠0，对直线段简弃之；

      c.若上述判断条件不成立，则需求出直线段与窗口边界的交点。

      两个端点进行逻辑或操作：根据其结果中1的位置来确定可能相交的窗口边

      左、右边界交点的计算：y = y1 + k(x - x1)；

      上、下边界交点的计算：x = x1 + (y-y1)/k

用编码方法实现了对完全可见和不可见直线段的快速接受和拒绝；求交过程复杂，有冗余计算，并且包含浮点运算，不利于硬件实现。

2. 中点分割算法

中点分割算法的核心思想是通过二分逼近来确定直线段与窗口的交点。

特点：主要计算过程只用到加法或位移运算，易于硬件实现，同时适合于并行计算。

3. Liang-Barsky算法

算法步骤：

(1)输入直线段的两端点坐标：(x1,y1)和(x2,y2)，以及窗口的四条边界坐标：wyt、wyb、wxl和wxr。

(2)若Δx=0，则p1=p2=0。此时进一步判断是否满足q1<0或q2<0，若满足，则该直线段不在窗口内，算法转(7)。否则，满足q1>0且q2>0，则进一步计算u1和u2。算法转(5)。

(3)若Δy=0，则p3=p4=0。此时进一步判断是否满足q3<0或q4<0，若满足，则该直线段不在窗口内，算法转(7)。否则，满足q1>0且q2>0，则进一步计算u1和u2。算法转(5)。

(4)若上述两条均不满足，则有pk≠0（k=1,2,3,4）。此时计算u1和u2。

(5)求得u1和u2后，进行判断：若u1>u2，则直线段在窗口外，算法转(7)。若u1<u2，利用直线的参数方程求得直线段在窗口内的两端点坐标。

(6)利用直线的扫描转换算法绘制在窗口内的直线段。算法结束。

4. Sutherland-Hodgeman多边形裁剪

基本思想：将多边形的边界作为一个整体，每次用窗口的一条边界对要裁剪的多边形进行裁剪，体现分而治之的思想。

算法实施策略：

为窗口各边界裁剪的多边形存储输入与输出顶点表。在窗口的一条裁剪边界处理完所有顶点后，其输出顶点表将用窗口的下一条边界继续裁剪。

窗口的一条边以及延长线构成的裁剪线把平面分为两个区域，包含窗口区域的区域称为可见侧；不包含窗口区域的域为不可见侧。

假定按顺时针方向处理顶点，且将用户多边形定义为Ps，窗口矩形为Pw。

算法从Ps的任一点出发，跟踪检测Ps的每一条边，当Ps与Pw相交时（实交点），按如下规则处理：

(1)若是由不可见侧进入可见侧，则输出可见直线段，转(3)；

```
(2)若是由可见侧进入不可见侧，则从当前交点开始，沿窗口边界顺时针检测Pw的边，即用窗口的有效边界去裁剪Ps的边，找到Ps与Pw最靠近当前交点的另一交点，输出可见直线段和由当前交点到另一交点之间窗口边界上的线段，然后返回处理的当前交点；
(3)沿着Ps处理各条边，直到处理完Ps的每一条边，回到起点为止。
```

5. lineClipping（Liang-Barsky算法）

```
//lineClipping.h
#include <iostream>
#include <vector>
#include <windows.h>
#include <GL/glut.h>
using namespace std;
//点类型point
typedef struct Point {
    int x, y;
    Point(int a = 0, int b = 0)
    {
        x = a, y = b;
    }
} point;
//矩形类型rect
typedef struct Rectangle{
    float w_xmin,w_ymin;
    float w_xmax,w_yman;
    Rectangle(float xmin = 0.0, float ymin = 0.0,float xmax=0.0,float yman=0.0){
        w_xmin = xmin;  w_ymin = ymin;
        w_xmax = xmax;  w_yman = yman;
    }
}rect;
int Clip_Top(float p,float q,float &umax,float &umin);
void Line_Clipping(vector<point> &points,rect &winRect);
void  onMidPointEllispe(int a,int b);
```

```
void  OnMidpointellispe(int a,int b,vector<point> &points);
```

```
//lineClipping.cpp
#include <vector>
using namespace std;
#include "lineClipping.h"
/*********************************************
*如果p参数<0，计算、更新umax，保证umax是最大u值
*如果p参数>0，计算、更新umin，保证umin是最小u值
*如果umax>umin，返回0，否则返回1
*********************************************/
int Clip_Top(float p,float q,float &umax,float &umin){
    float r=0.0;
    if(p<0.0)    //线段从裁剪窗口外部延伸到内部，取最大值r并更新umax
    {
        r=q/p;
        if (r>umin) return 0;    //umax>umin的情况，弃之
        else if (r>umax)  umax=r;
    }
    else if(p>0.0)       //线段从裁剪窗口内部延伸到外部,取最小值r并更新umin
        {
            r=q/p;
            if (r<umax) return 0;    //umax>umin的情况，弃之
            else if(r<umin)  umin=r;
        }
       else          //p=0时，线段平行于裁剪窗口
          if(q<0.0) return 0;
     return 1;
}
/*********************************************************
*已知winRect：矩形对象，存放标准裁剪窗口4条边信息
*    points：点的动态数组，存放直线2个端点信息
*根据裁剪窗口的左、右边界，求umax;
*根据裁剪窗口的下、上边界，求umin
*如果umax>umin，裁剪窗口和直线无交点，否则求裁剪后直线新端点
*********************************************************
*******/
void Line_Clipping(vector<point> &points,rect & winRect)
{
```

```
    //比较左、右边界，获得最大的umax
    point &p1=points[0],&p2=points[1];
    float dx=p2.x-p1.x,dy=p2.y-p1.y,umax=0.0,umin=1.0;

    point p=p1;
    if (Clip_Top(-dx,p1.x- winRect.w_xmin,umax,umin))
 //左边界
        if (Clip_Top(dx,winRect.w_xmax-p1.x, umax,umin))
//右边界
            //比较下、上边界，获得最小的umin
            if (Clip_Top(-dy,p1.y- winRect.w_ymin,
umax,umin)) //下边界
                if (Clip_Top(dy,winRect.w_yman-p1.y,
umax,umin)) //上边界
                {//求裁剪后直线新端点
                    p1.x=(int)(p.x+umax*dx);
                    p1.y=(int)(p.y+umax*dy);
                    p2.x=(int)(p.x+umin*dx);
                    p2.y=(int)(p.y+umin*dy);
                }
}
```

3 实验三：三维投影

OpenGL中只提供了两种投影方式，一种是平行投影（正射投影），另一种是透视投影。

在投影变换之前必须指定当前处理的是投影变换矩阵：

glMAtrixMode(GL_PROJECTION)；glLoadIdentity()；

平行投影：视景体是一个矩形的平行管道，也就是一个长方体，其特点是无论物体距离相机多远，投影后的物体大小尺寸不变。

void glOrtho (GLdouble left，GLdouble right，GLdouble bottom，GLdouble top，GLdouble near，GLdouble far)；

1 setLookAt

void gluLookAt（GLdouble eyex，GLdouble eyey，GLdouble eyez，GLdouble centerx，GLdouble centery，GLdouble centerz，GLdouble upx，GLdouble upy，GLdouble upz）；//视图变换函数（定义观察坐标系）

```
//setLookAt.h
#include <gl/glut.h>
```

```
#include <math.h>
//点类型point
typedef struct Point3D {
    double x, y,z;
    Point3D(double a = 0.0,double b = 0.0,double c = 0.0)
    {
        x = a, y = b;z=c;
    }
} point3D;
 float PI=3.14159f;
 //根据菜单menuValue选择的三视图，建立视角视觉效果
void setLookAt(int menuValue,point3D eye,point3D at){
    switch(menuValue)//根据菜单项值
    {
        case 1://正视图XOZ(V)
            gluLookAt(0,eye.y,0,at.x,at.y,at.z,0,0,1);//视点在y轴上
            break;
        case 2://俯视图XOY(H)

gluLookAt(0,0,eye.z,at.x,at.y,at.z,0,-1,0);//视点在z轴上
            break;
        case 3://侧视图YOZ(W)
            gluLookAt(eye.x,0,0,at.x,at.y,at.z,0,0,1);//视点在x轴上
            break;
    }
}
//根据菜单menuValue选择的三视图，根据右手原则，围绕某坐标轴旋转angle，建立视角视觉效果
void setLookAt(int menuValue,float angle,point3D eye,point3D at){
    GLfloat squareRootOf2=1.4142;  //√2= 1.4142136023731
    GLdouble dcos,dsin,radius;
    float Rot =float(PI*angle/180.0f);
    dcos=cos(Rot);
    dsin=sin(Rot);
    switch(menuValue)//根据菜单项值
    {
        case 1://正视图XOZ(V)，视点绕y轴旋转
            radius=sqrt((eye.x*eye.x+eye.z*eye.z));
            gluLookAt(dcos* radius,eye.y,dsin*radius,at.x,at.y,at.z,0,1,0);
            break;
```

```
        case 2://俯视图XOY(H)，视点绕z轴旋转
            radius=sqrt((eye.x*eye.x+eye.y*eye.y));
            gluLookAt(dsin* radius,dcos
*radius,eye.z,at.x,at.y,at.z,0,0,1);
            break;
        case 3://侧视图YOZ(W)，视点绕x轴旋转
            radius=sqrt((eye.y*eye.y+eye.z*eye.z));
            gluLookAt(eye.x,dsin *radius,dcos
*radius,at.x,at.y,at.z,1,0,0);
            break;
    }
}

//控制照相机的三维旋转角度，将场景坐标系映射到eye的观察坐标系，建
立第一人称视角视觉效果
void setLookAt(const GLfloat x_angle,const GLfloat
y_angle,const GLfloat z_angle,point3D eye,point3D at){
    GLdouble dxc,dxs,dyc,dys,dzc,dzs;
    dxc=cos(PI*x_angle/180);
    dxs=sin(PI*x_angle/180);
    dyc=cos(PI*y_angle/180);
    dys=sin(PI*y_angle/180);
    dzc=cos(PI*z_angle/180);
    dzs=sin(PI*z_angle/180);
    GLdouble ux,uy,uz,vx,vy,vz,nx,ny,nz;
    ux=dyc*dzc;
    uy=dyc*dzs;
    uz=-dys;
    vx=dxs*dys*dzc-dxc*dzs;
    vy=dxs*dys*dzs+dxc*dzc;
    vz=dxs*dyc;
    nx=dxc*dys*dzc+dxs*dzs;
    ny=dxc*dys*dzs-dxs*dzc;
    nz=dxc*dyc;
    point3D camera;
    camera.x=eye.x *ux+eye.y*vx+eye.z*nx;
    camera.y=eye.x *uy+eye.y*vy+eye.z*ny;
    camera.z=eye.x *uz+eye.y*vz+eye.z*nz;

gluLookAt(camera.x,camera.y,camera.z,at.x,at.y,at.z,vx,v
y,vz);
}
```

2. getTetrahedron (四面体图元)

```
//getTetrahedron.h
#include <gl/glut.h>
GLuint getTetrahedron(point3D pA,point3D pB,point3D pC,point3D pD){//4点ABCD定义四面体图元
    GLuint tetrahedronID = glGenLists(1);//获得一个有效的无符号整数作为列表ID
    glNewList(tetrahedronID, GL_COMPILE);//通过显示列表定义四面体
        glBegin(GL_TRIANGLES);//通过4个三角形定义四面体图元
            /*三角形ABD*/
            glColor4f(0.0f,0.6f,1.0f,0.5f);//设置笔的颜色为黑
            glVertex3f(pA.x, pA.y, pA.z);    // 绘制A顶点
            glVertex3f(pB.x, pB.y, pB.z);    // 绘制B顶点
            glVertex3f(pD.x, pD.y, pD.z);    // 绘制D顶点
            /*三角形ABC*/
            glColor4f(0.6f,0.0f,1.0f,0.5f);//设置笔的颜色为黑
            glVertex3f(pA.x, pA.y, pA.z);    // 绘制A顶点
            glVertex3f(pB.x, pB.y, pB.z);    // 绘制B顶点
            glVertex3f(pC.x, pC.y, pC.z);    // 绘制C顶点
            /*三角形ACD*/
            glColor4f(1.0f,0.6f,0.0f,0.5f);//设置笔的颜色为黑
            glVertex3f(2.0f, 0.0f, 0.0);     // 绘制A顶点
            glVertex3f(pC.x, pC.y, pC.z);    // 绘制C顶点
            glVertex3f(pD.x, pD.y, pD.z);    // 绘制D顶点
            /*三角形BCD*/
            glColor4f(0.6f,1.0f,0.6f,0.5f);//设置笔的颜色为黑
            glVertex3f(pB.x, pB.y, pB.z);    // 绘制B顶点
            glVertex3f(pC.x, pC.y, pC.z);    // 绘制C顶点
            glVertex3f(pD.x, pD.y, pD.z);    // 绘制D顶点
        glEnd();
    glEndList();//显示列表定义结束
    return tetrahedronID;//返回四面体图元ID
}

```

3. getAxis

```
//getAxis.h
#include <windows.h>
#include <gl/glut.h>
#define MAX_CHAR 128
```

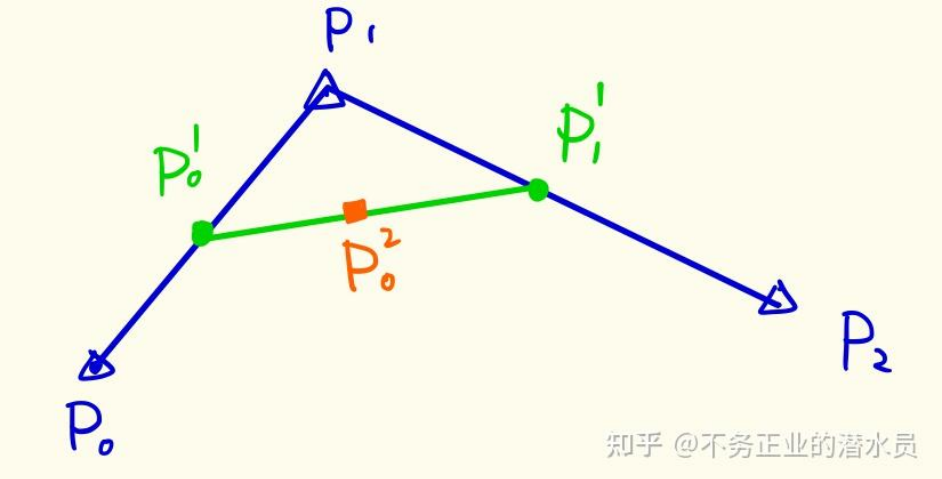
```
GLuint getAxis(float axisLength){//根据轴长axisLength定义XYZ三维坐标系
    GLuint axisID = glGenLists(1);//获得一个有效的无符号整数作为列表ID
    static int isFirstCall=1;
    static GLuint lists;
    if(isFirstCall){//如果第一次调用，为每一个ASCII转换产生一个显示列表
        lists = glGenLists(MAX_CHAR); //申请MAX_CHAR个连续的显示列表ID

wglUseFontBitmaps(wglGetCurrentDC(),0,MAX_CHAR,lists);//将每个ASCII字符的绘制命令封装到对应的显示列表中
    }
    glListBase(lists);//设置显示列表ID的起始位置
    glNewList(axisID, GL_COMPILE);//通过显示列表定义坐标系
        glColor4f(1.0f,0.0f,0.0f,1.0f);//x轴，设置笔的颜色为red
        glBegin(GL_LINES);
            glVertex3f(-axisLength,0.0,0.0);
            glVertex3f(axisLength,0.0,0.0);
        glEnd();
        glRasterPos3f(axisLength,0,0);//设置字符串"x"的起始位置
        glCallLists(strlen("x"),GL_UNSIGNED_BYTE,"x");//一次调用多个显示列表，显示字符串"x"
        glColor4f(0.0f,1.0f,0.0f,1.0f);//y轴，设置笔的颜色为Greed
        glBegin(GL_LINES);
            glVertex3f(0.0,-axisLength,0.0);
            glVertex3f(0.0,axisLength,0.0);
        glEnd();
        glRasterPos3f(0,axisLength,0);//设置字符串"y"的起始位置
        glCallLists(strlen("y"),GL_UNSIGNED_BYTE,"y");//一次调用多个显示列表，显示字符串"y"
        glColor4f(0.0f,0.0f,1.0f,1.0f);//z轴，设置笔的颜色为blue
        glBegin(GL_LINES);
            glVertex3f(0.0,0.0,-axisLength);
            glVertex3f(0.0,0.0,axisLength);
        glEnd();
        glRasterPos3f(0,0,3);//设置字符串"z"的起始位置
        glCallLists(strlen("z"),GL_UNSIGNED_BYTE,"z");//一次调用多个显示列表，显示字符串"z"
```

```
36        glEndList();
37        return axisID;
38    }
```

4. 实验四：样条曲线
    1. 曲线常用的定义方法

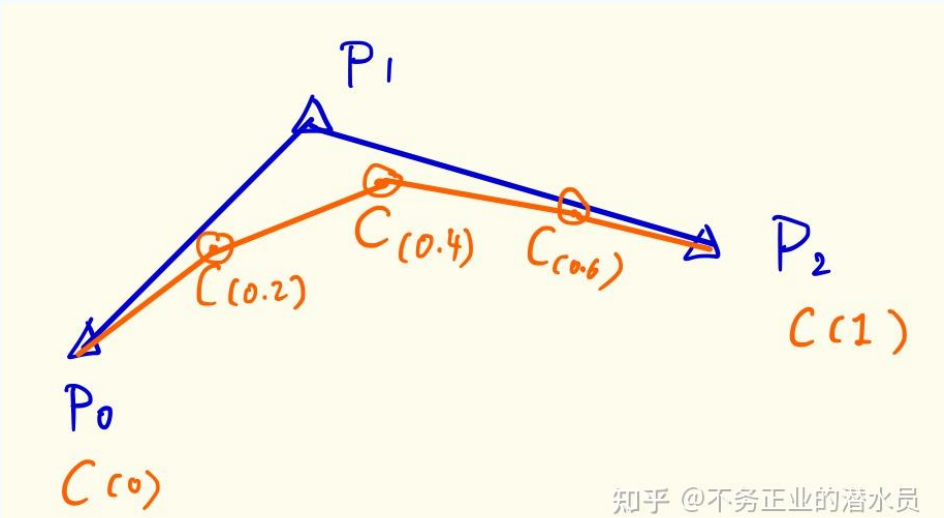
    > 插值：给定一组有序的数据点Pi，i=0, 1, …, n，构造一条曲线顺序通过这些数据点，称为对这些数据点进行插值，所构造的曲线称为插值曲线。常用插值方法有线性插值、抛物线插值等(Interpolation)。型值点.
    > 逼近：构造一条曲线使之在某种意义下最接近给定的数据点，称为对这些数据点进行逼近，所构造的曲线为逼近曲线(Approximation)。控制点
    > 拟合：插值和逼近则统称为拟合（fitting）.

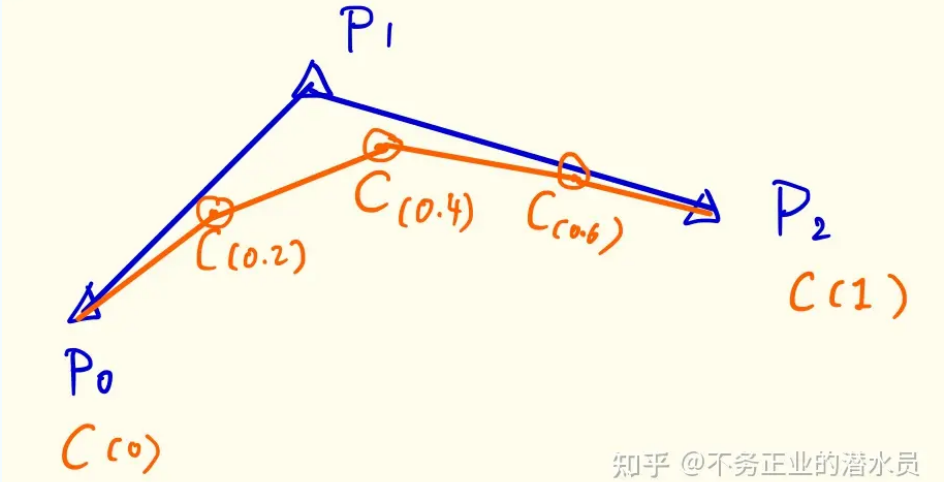
    2. 参数连续性和几何连续性
        1. n阶参数连续性:相邻点处1到n阶导数都相等 C^x连续性
        2. n阶几何连续性:相邻点处的1到n阶导数成比例 (方向相同,大小可以不同) G^x连续性
    3. Hermite三次样条
    4. Benzier曲线和B样条曲线

    > 先来一个简单的例子，理解如何用递归的方法拟合一条曲线。
    >
    > 
    >
    >
    > 先求出绿色的点,再求出橘色的点,依次递归.
    > 第n次递归之后，可以得到一条相对平滑的曲线。
    > **把前面这个例子的递归方法整理成公式，就是Bezier曲线。**
    >
    > Bezier曲线

公式：$C(t)=\Sigma B_{i,n}(t)P_i$, $0<=t<=1$

$B_{i,n}(t)=[n!/(i!(n-i)!)]*(1-t)^{n-i}t^i$ i为控制点号,n为多项式的次数





但是，Bezier曲线存在许多弊端：

1. 复杂情况时计算难度大：对于拥有很多控制点的复杂曲线，Bezier曲线需要更高阶的多项式
2. 难以修改：一个控制点的变化牵动整条曲线
3. 合并困难：合并两条曲线很难取得平滑的新曲线

这些缺点都可以运用B-spline曲线加以改善。

B-spline曲线

公式：$C(t)=\Sigma N_{i,p}(x)P_i$ $N_{i,p}(x)=N_{i,p-1}(x)(x-t_i)/(t_{i+p}-t_i)+N_{i+1,p-1}(x)(t_{i+p+1}-x)/(t_{i+p+1}-t_{i+1})$

- 和Bezier曲线的公式相似，曲线上点的坐标是控制点的加权平均值 p为多项式的次数
- 此时的x等同于Bezier曲线中的t，对应新生成的曲线点所在位置， $1\leq x\leq 11\leq x \leq 11\leq x \leq 1$

- 此时的t是新引入的概念，节点值，通过节点，我们可以控制控制点在局部曲线的影响。

进一步理解B-spline曲线的节点概念：

当x不在两个节点间时，对应控制点的权重为零。也就是说，控制点只对临近的曲线点有影响，对较远的曲线点没有影响，因此解决了Bezier曲线‘牵一发而动全身’的问题。

5. graphicType

```
//graphicType.h
//三维点类型point
typedef struct Point3D {
    double x, y,z;
    Point3D(double a = 0.0,double b = 0.0,double c = 0.0)
    {
        x = a, y = b;z=c;
    }
} point3D;
//二维点类型point
typedef struct Point {
    int x, y;
    Point(int a = 0, int b = 0) {
        x = a;
        y = b;
    }
} point;
//矩形类型rect
typedef struct Rectangle{
    float w_xmin,w_ymin;
    float w_xmax,w_yman;
    Rectangle(float xmin = 0.0, float ymin = 0.0,float xmax=0.0,float yman=0.0){
        w_xmin = xmin;  w_ymin = ymin;
        w_xmax = xmax;  w_yman = yman;
    }
}rect;
//颜色类型color
typedef struct Color{
    int r,g,b;
    Color(int red= 0,int green = 0,int blue=0){
        r=red;
        g=green;
```

```
        b=blue;
    }
}color;
```

6 drawBezier（Bezier曲线算法） PPT上的例题不会在考试中出现

```
//drawBezier.h
#include <vector>
using namespace std;
//根据点序列向量数组points，绘制折线
void drawPolygonalLine(vector<point> &points,color &c );
//根据点序列向量数组points，绘制折线
void drawPolygonalLine(vector<point> &points ,point &p,color &c);
//如果控制点数n<4，绘制一段n-1次Bezier曲线段，否则绘制一段三次Bezier曲线段
void drawBezierCurve(vector<Point> &points,color &c);
//通过（m×3+1）控制点，绘制m段n-1次ezier曲线段
void drawBezierCurve(vector<Point> &points,int n,color &c);
```

```
//drawBezier.cpp
#include <vector>
using namespace std;
#include <GL/glut.h>
#include "graphicType.h"
//绘制折线
void drawPolygonalLine(vector<point> &points,color &c ){//根据点序列向量数组points，绘制折线
    glColor3f(c.r, c.g, c.b);
    glBegin (GL_LINE_STRIP);
        for(int i=0;i<points.size();i++){
            glVertex2i(points[i].x, points[i].y);
        }
    glEnd();
}
//绘制折线
void drawPolygonalLine(vector<point> &points ,point &p,color &c){//根据点序列向量数组points，绘制折线
    glColor3f(c.r, c.g, c.b);
    glBegin (GL_LINE_STRIP);
        for(int i=0;i<points.size();i++){
            glVertex2i(points[i].x, points[i].y);
        }
```

```
        glVertex2i(p.x, p.y);
    glEnd();
}
//如果控制点数n<4，绘制一段n-1次Bezier曲线段，否则绘制一段三次Bezier曲线
void drawBezierCurve(vector<Point> &points,color &c){
    GLfloat ControlP[4][3];
    int iPointNum=points.size();
    int i;
    if(iPointNum<4){//控制点数<4，绘制一段n-1次Bezier曲线段
        for(i=0;i<iPointNum;i++){
            ControlP[i][0]=points[i].x;
            ControlP[i][1]=points[i].y;
            ControlP[i][2]=0.0;
        }
    }
    else{
        for(i=0;i<4;i++){//控制点数≥4，绘制一段三次Bezier曲线段
            ControlP[i][0]=points[i].x;
            ControlP[i][1]=points[i].y;
            ControlP[i][2]=0.0;
        }
    }
    glColor3f(c.r, c.g, c.b);
    glPointSize(2);
    if(iPointNum>4)

glMap1f(GL_MAP1_VERTEX_3,0.0,1.0,3,4,*ControlP);//定义一维取值器
    else

glMap1f(GL_MAP1_VERTEX_3,0.0,1.0,3,iPointNum,*ControlP);//定义一维取值器
    glEnable(GL_MAP1_VERTEX_3);
    glMapGrid1f(100,0.0,1.0);//生成均匀分布的一维网格参数值
    glEvalMesh1(GL_LINE, 0, 100);//绘制Bezier曲线
}
//通过 (m×3+1) 控制点，绘制m段n-1次ezier曲线段
void drawBezierCurve(vector<Point> &points,int n,color &c){
}
```

5 实验五：光线追踪