<u>Computer Graphics</u>

Laboratory Assignments

BCSE – 3rd Year, 1st Semester

Group Members:
Soumya Kanti Naskar - 001410501044
Sahil Pandey - 001410501057

Index

```c
/*                               0.RASTER.C                                */

#ifndef magnified_raster_display
#define magnified_raster_display
#include <stdio.h>
#include <graphics.h>
#include <time.h>

struct Raster //holds configuration settings for exploded raster display
{
    int centerX, centerY, width; //origin co-ordinates and pixel width
    float dely; //delay between pixels
    clock_t pre, cur; //used to realize delays
} raster;

void fillPixel(int x, int y) //fills the specified pixel
{
    while(((raster.cur=clock())-raster.pre)<(raster.dely*CLOCKS_PER_SEC));
                                    //wait for the required delay
    int i, j;
    int bLeftX=raster.centerX+x*raster.width;
    int bLeftY=raster.centerY-y*raster.width;
    for(i=1; i<raster.width; ++i)
        for(j=0; j<raster.width; ++j)
            if(getpixel(bLeftX+i, bLeftY-j)==WHITE)
                putpixel(bLeftX+i, bLeftY-j, BLUE); //fill pixel by pixel if empty
    raster.pre=raster.cur=clock();
}

void refLine(int x1, int y1, int x2, int y2) //reference line
{
    setcolor(RED);
    line(raster.centerX+x1*raster.width,
            raster.centerY-y1*raster.width,
                raster.centerX+x2*raster.width,
                    raster.centerY-y2*raster.width);
}

void refEllipse(int xc, int yc, int a, int b) //reference ellipse
                                  //also used for circle
{
    setcolor(RED);
    ellipse(raster.centerX+xc*raster.width,
            raster.centerY-yc*raster.width,
                0, 360,
                a*raster.width,
                    b*raster.width);
}


void horiLine(int x1, int x2, int y) //trivial algorithm for horizontal case
{
    int i;
    refLine(x1, y, x2, y);
    for(i=(x1>x2?x2:x1); i<(x1>x2?x1:x2); ++i)
        fillPixel(i, y);
}

void vertLine(int x, int y1, int y2) //trivial algorithm for vertical case
{
    int i;
    refLine(x, y1, x, y2);
    for(i=(y1>y2?y2:y1); i<(y1>y2?y1:y2); ++i)
        fillPixel(x, i);
}

void diagLine(int x1, int y1, int x2, int y2) //trivial algorithm for diagonal cases
{
    int i, j, sign=(x1-x2)+(y1-y2)?1:-1;
    refLine(x1, y1, x2, y2);
    for(i=(x1>x2?x2:x1), j=(x1>x2?y2:y1); i<(x1>x2?x1:x2); ++i, j+=sign)
        fillPixel(i, j);
}

void grid() //initializes grid for graphics
{
    FILE *setting;
    int i, mx, my;
    setting=fopen("settings.kpd", "rx"); //open settings file
    if(!setting) //file not opened, use defaults
    {
        raster.centerX=320; //origin at center of screen
        raster.centerY=240;
        raster.width=10; //width 10p
        raster.dely=0.25; //delay 1/4s
        setting=fopen("settings.kpd","w"); //force create settings file...
        if(!setting)
            printf("\nLOG [ERROR] : could not create file settings.kpd\n");
        else
        {
            fwrite(&raster, sizeof(raster), 1, setting); //...and store defaults
            fclose(setting);
        }
    }
    else
    {
        fread(&raster, sizeof(raster), 1, setting); //read settings
        fclose(setting);
    }
    raster.pre=0; //set previous time to beginning of program
    setcolor(WHITE);
    bar(0, 0, mx=getmaxx(), my=getmaxy()); //white out grid
    setcolor(LIGHTGRAY);
    for(i=raster.centerY; i<=my; i+=raster.width) //draw gridlines
        line(0, i, mx, i);
    for(i=raster.centerX; i<=mx; i+=raster.width) //(begin at origin,
        line(i, 0, i, my);
    for(i=raster.centerY; i>=0; i-=raster.width) //proceed in both directions)
        line(0, i, mx, i);
    for(i=raster.centerX; i>=0; i-=raster.width)
        line(i, 0, i, my);
    setcolor(BLACK);
    line(0, raster.centerY, mx, raster.centerY);  //draw axes
    line(raster.centerX, 0, raster.centerX, my);
}

#endif
```

```c
/*                              0.SETTING.C
*
*   command line syntax (case insensitive)
*
*   -ctln = sets origin to narrow top left
*   -ctb  = sets origin to top center
*           notice the 'b'ottom is ignored
*           as it contradicts the 't'op
*   -cn   = sets origin to center of the screen
*   -c    = no change to origin
*
*   -w25  = sets raster pixel width to 25p
*   -w    = no change to width
*
*   -d0.5 = sets rasterisation delay to 1/2s
*   -d    = no change to delay
*
*/

#include <stdio.h>
#include <stdlib.h>
#include "0.raster.c"

int main(int argc, char **argv)
{
    int x, y, n, i, j, flag;
    FILE *setting;
    x=y=n=i=j=flag=0;
    if(argc>1) //arguments to parse
    {
        setting=fopen("settings.kpd", "rx"); //open settings file
        if(!setting) //file not opened, use defaults
        {
            raster.centerX=320; //origin at center of screen
            raster.centerY=240;
            raster.width=10; //width 10p
            raster.dely=0.25; //delay 1/4s
            setting=fopen("settings.kpd","w"); //force create settings file...
            if(!setting)
            {
                printf("\nFATAL ERROR : could not create file \"settings.kpd\"\n");
                return 1;
            }
            else
                fwrite(&raster, sizeof(raster), 1, setting); //...and store defaults
        }
        else
        {
            fread(&raster, sizeof(raster), 1, setting); //read settings
            fclose(setting);
        }
    }
    else return 0; //no arguments, do nothing
    while((i++)<argc-1) //parse the arguments
    {
        if(argv[i][0]=='-') //commands start with a '-'
            switch(argv[i][1])
            {
                case 'c': case 'C': //center command
                    for(j=2; argv[i][j]; ++j)
```

```
                        switch(argv[i][j])
                        {
                            case 't': case 'T': //top
                                if(!y)
                                    y=1;
                                break;
                            case 'b': case 'B': //bottom
                                if(!y)
                                    y=-1;
                                break;
                            case 'l': case 'L': //left
                                if(!x)
                                    x=-1;
                                break;
                            case 'r': case 'R': //right
                                if(!x)
                                    x=1;
                                break;
                            case 'n': case 'N': // narrow/neutral
                                if(!n)
                                    n=1;
                                break;
                        }
                    if(x || y || n) //if a change has been made
                    {
                        flag=1; //set flag
                        raster.centerX=320 + x*(160 + 140*n); //recalculate
                        raster.centerY=240 - y*(120 + 100*n);
                    }
                    break;
                case 'w': case 'W': //width command
                    raster.width=atoi(argv[i]+2);
                    flag=1; //set flag
                    break;
                case 'd': case 'D': //delay command
                    raster.dely=atof(argv[i]+2);
                    flag=1; //set flag
                    break;
                default:
                    printf("\nLOG [ERROR] : Command [ %s ] not found", argv[i]);
            }
    }
    if(flag)
    {
        setting=fopen("settings.kpd", "w");
        if(!setting)
        {
            printf("FATAL ERROR : aborted");
            return 1;
        }
        fwrite(&raster, sizeof(raster), 1, setting); //write changes to file
        fclose(setting);
    }
    return 0;
}
```

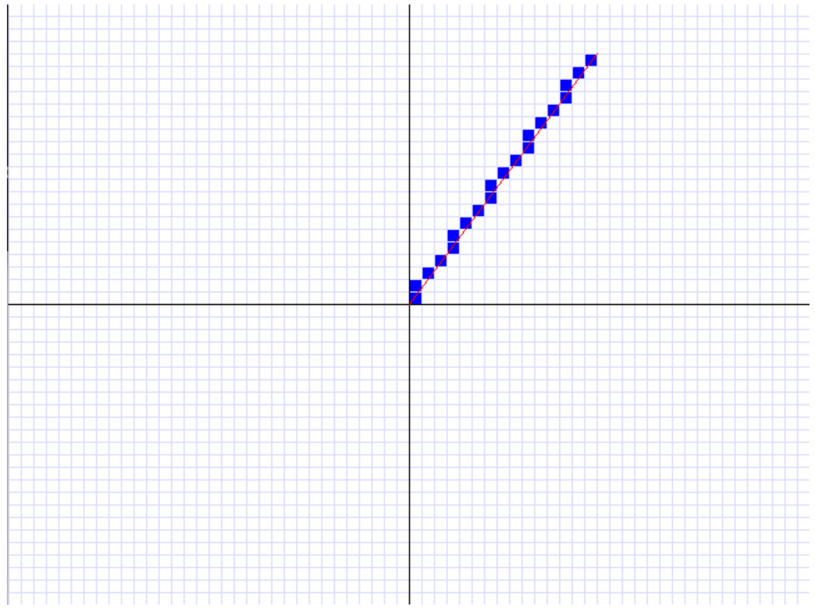
<u>Digital Differential Analyzer</u>

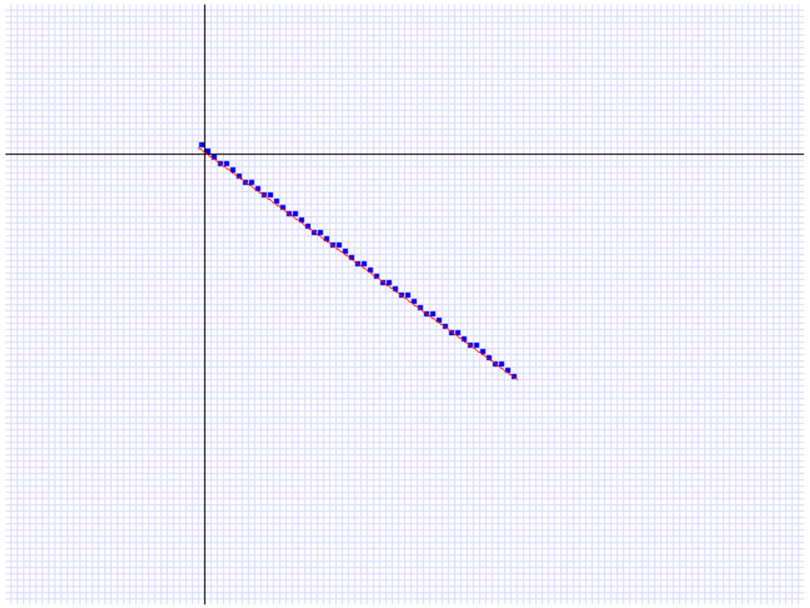
A line rasterization algorithm based on calculating the derivative of the line (i.e. the slope). The axis along which a higher change occurs is taken as the primary iterative direction. Along this direction, the variable is varied by one and the corresponding change in the other co-ordinate is tracked.

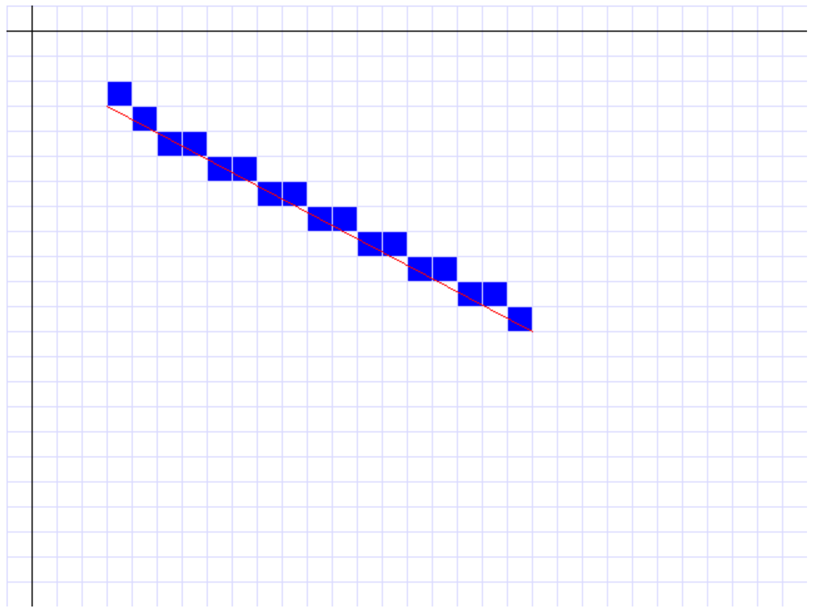
The algorithm uses floating point arithmetic; hence its performance leaves something to be desired. Further, the algorithm assumes that are floating point values are always rounded towards $-\infty$. However most computers implement a “round towards 0” approach. This necessitates the simulation of the greatest integer function in software.

```c
/*                                  1.DDA.C                                 */
#include <stdio.h>
#include <stdlib.h>
#include "0.raster.c"

int gif(double d) //greatest integer function
{
    if((int)d>d)
        return (int)d-1;
    return (int)d;
}

void ddaLine(int x1, int y1, int x2, int y2)
{
    double x, y, xinc, yinc, dx, dy;
    int step, k;
    dy = y2-y1;
    dx = x2-x1;
    step=(abs(dx)>abs(dy))?abs(dx):abs(dy); //choose the larger number of divisions
    xinc = dx/step; //change at each step
    yinc = dy/step;
    x=gif(x1+0.5); //round down
    y=gif(y1+0.5);
    refLine(x1, y1, x2, y2); //reference
    for(k=1; k<=step; ++k)
    {
        fillPixel(gif(x), gif(y)); //plot
        x+=xinc; //update
        y+=yinc;
    }
}

int main(int argc, char** argv)
{
    int x1,y1,x2,y2;
    int gd=DETECT,gm;
    if(argc<5)
    {
        printf("[ ERROR ] Usage : x1 y1 x2 y2\n");
        return 1;
    }
    x1=atoi(argv[1]); //get arguments
    y1=atoi(argv[2]);
    x2=atoi(argv[3]);
    y2=atoi(argv[4]);
    initgraph(&gd, &gm, NULL);  //initialize graphics
    grid();
    if(x1==x2) //check for trivial cases
        vertLine(x1, y1, y2);
    else if(y1==y2)
        horiLine(x1, x2, y1);
    else if(abs(x1-x2)==abs(y1-y2))
        diagLine(x1, y1, x2, y2);
    else ddaLine(x1, y1, x2, y2); //call algorithm
    getchar();
    closegraph();
    return 0;
}
```

Output





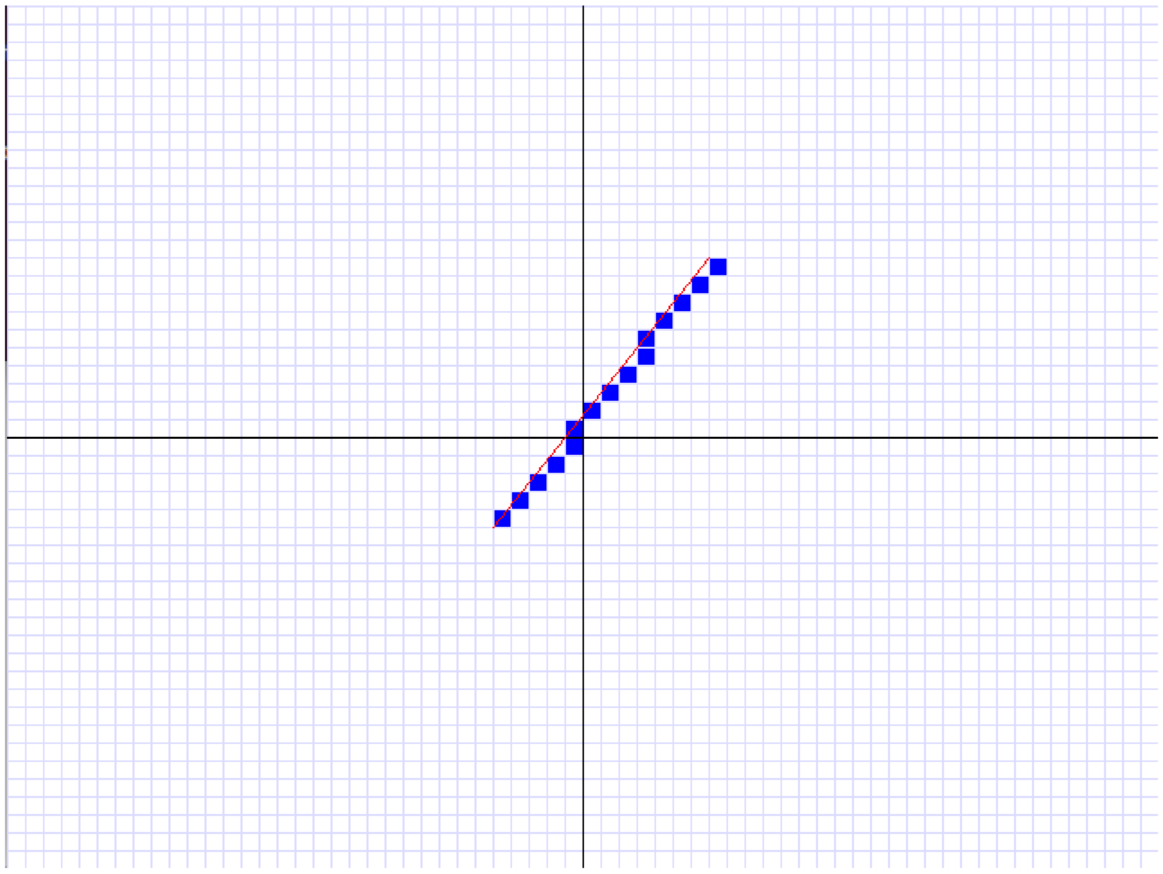


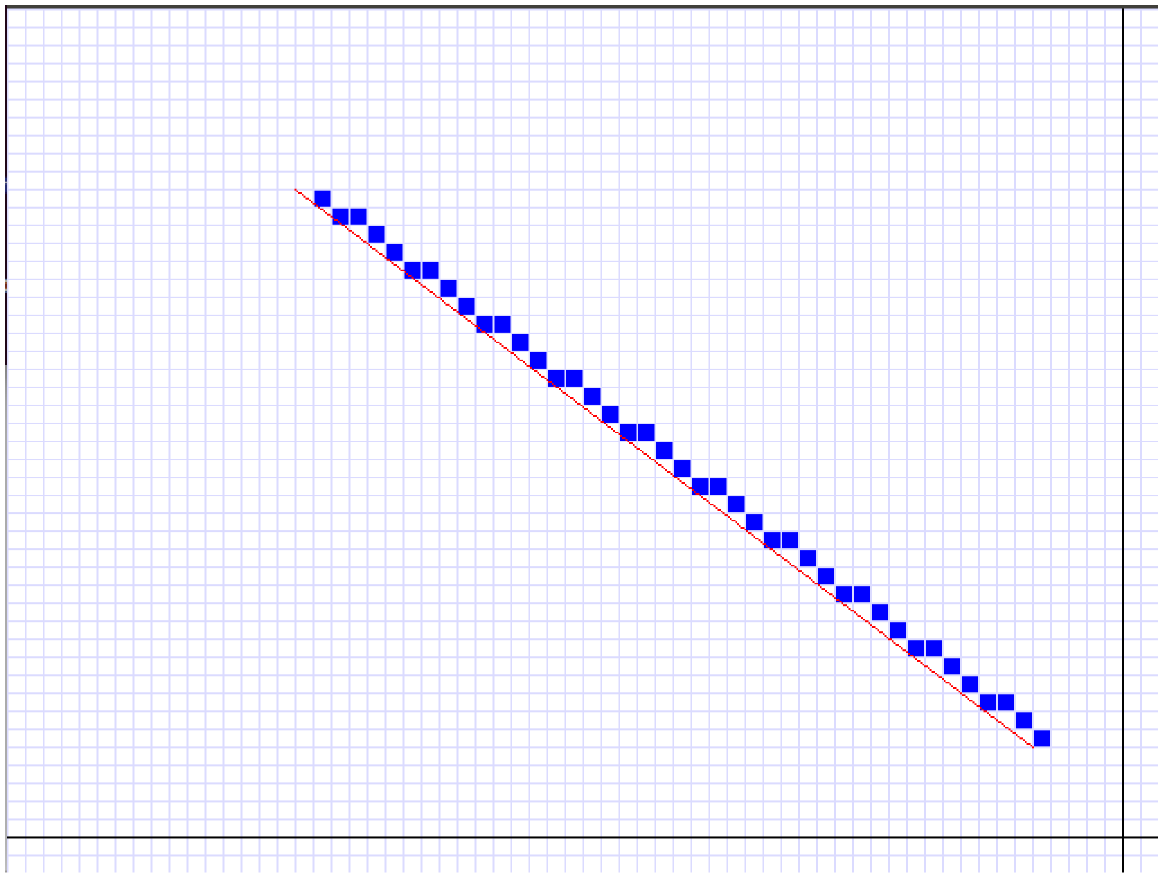
<u>Bresenham’s Line Algorithm</u>

Bresenham's line-drawing algorithm uses an iterative scheme. A pixel is plotted at the starting coordinate of the Line and each iteration of the algorithm increments the pixel one unit along the major axis. The algorithm updates and monitors the value of a decision variable, which is dependent on the slope of the line. The pixel is incremented along the minor axis only when the decision variable changes sign. A key feature of the algorithm is that it requires only integer data and simple arithmetic. This makes the algorithm highly efficient. The algorithm assumes the line has positive slope less than one, but a simple variable exchange technique can modify it for the general case.

```
/*                           2.LINE_BRESENHAM.C                              */
#include <stdio.h>
#include <stdlib.h>
#include "0.raster.c"

void bresenhamLine(int x1, int y1, int x2, int y2)
{
    int x, y, dx, dy, s1, s2, e, flag, i;
    refLine(x1, y1, x2, y2); //reference
    i=1; //initialize variables
    x=x1;
    y=y1;
    dx=x2-x1;
    dy=y2-y1;
    s1=dx>0?1:-1;
    s2=dy>0?1:-1;
    dx*=s1;
    dy*=s2;
    e=(dy<<1)-dx; //initialize e for comparison with the y=1/2 line
    if(dy>dx) { int temp=dx; dx=dy; dy=temp;
                flag=1; } //interchange dx and dy taking lesser slope
    else flag=0;
    while(i<=dx)
    {
        fillPixel(x,y); //plot
        if(e>=0) //above y=1/2 line, diagonal move
        {
            if(flag) x+=s1;
            else y+=s2;
            e-=(dx<<1);
        }
        if(flag) y+=s2; //orthogonal move
        else x+=s1; //update
        e+=(dy<<1);
        ++i;
    }
}

int main(int argc, char** argv)
{
    int x1,y1,x2,y2;
    int gd=DETECT,gm;
    if(argc<5)
    {
        printf("[ ERROR ] Usage : x1 y1 x2 y2\n");
        return 1;
    }
    x1=atoi(argv[1]); //get arguments
    y1=atoi(argv[2]);
    x2=atoi(argv[3]);
    y2=atoi(argv[4]);
    initgraph(&gd, &gm, NULL); //initialize graphics
    grid();
    if(x1==x2) //check for trivial cases
        vertLine(x1, y1, y2);
    else if(y1==y2)
        horiLine(x1, x2, y1);
    else if(abs(x1-x2)==abs(y1-y2))
        diagLine(x1, y1, x2, y2);
    else bresenhamLine(x1, y1, x2, y2); //call algorithm
    getchar();
    closegraph();
    return 0;
}
```

Output





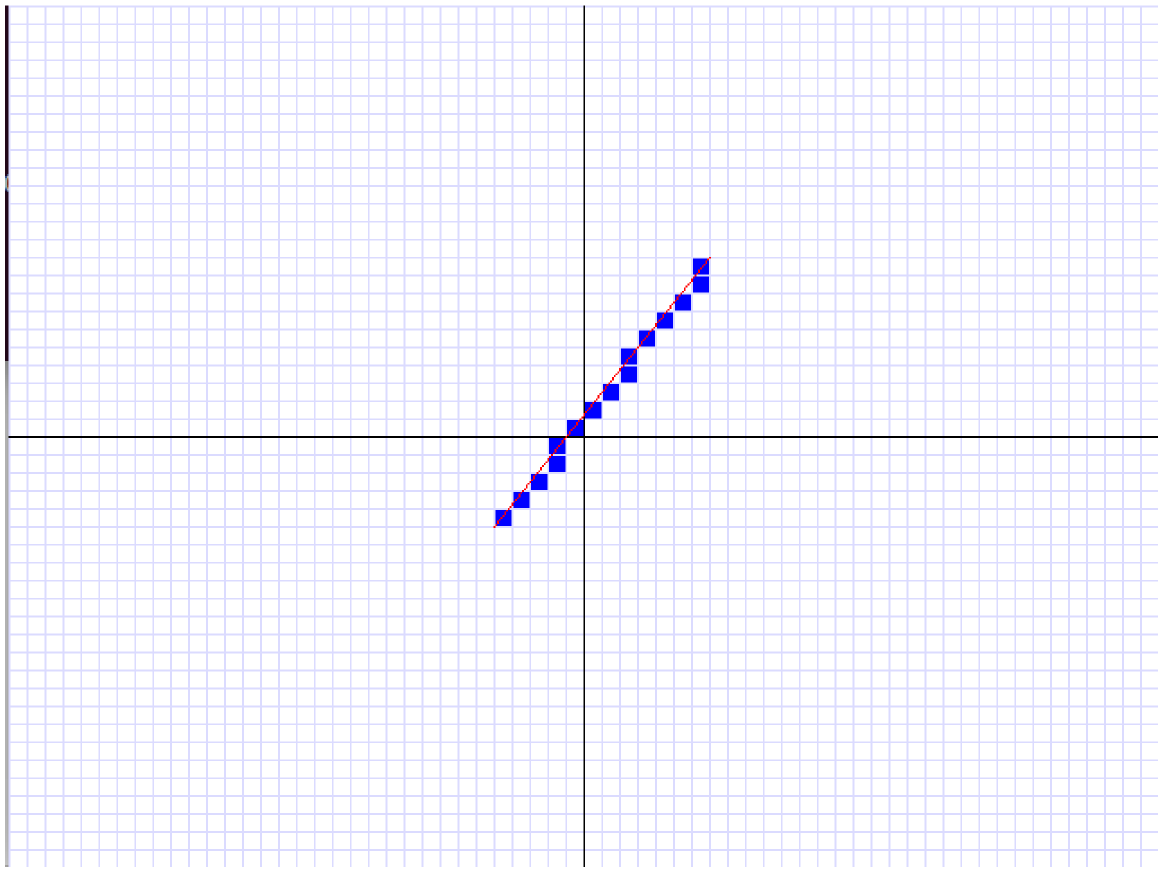
Mid-point Line Algorithm

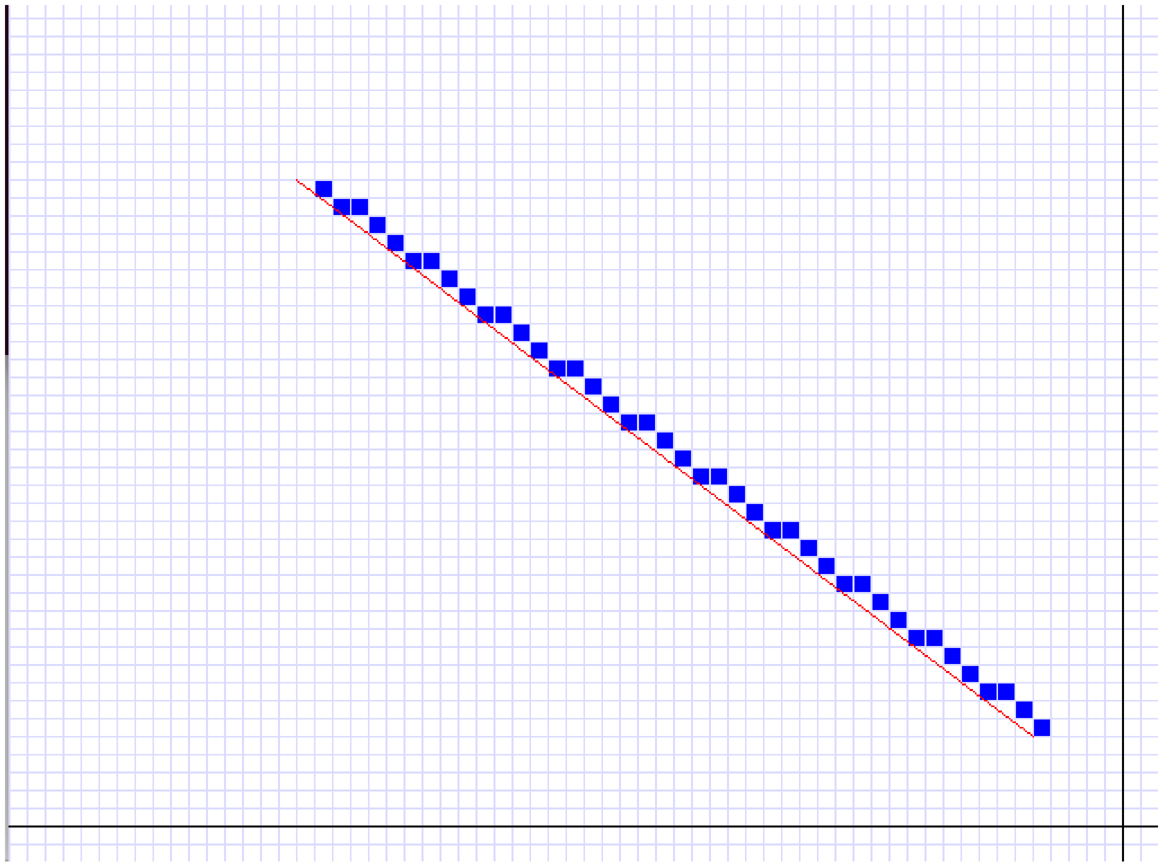
The Mid-point Line Algorithm is a simplification of Bresenham’s Line. It too is an iterative procedure that relies on simple integer arithmetic. The decision variable, in this case, can intuitively by thought of as the distance of the actual line, from the mid-point of the two candidates for the next iteration (hence the name). The choice between the two is made based on the sign of the decision variable. This algorithm too assumes the line to have positive slope less that one and it too can be generalized with a variable exchange.

The most attractive feature of this algorithm is its simplicity and the ease with which its basic idea can be extended to more complex curves.

```c
/*                                  3.MIDPNTLINE.C                          */
#include <stdio.h>
#include <stdlib.h>
#include "0.raster.c"

int gif(double d) //greatest integer function
{
    if((int)d>d)
        return (int)d-1;
    return (int)d;
}

void mpLine(int x1, int y1, int x2, int y2)
{
    int dy, dx, p, x, y, s1, s2, i, flag;
    refLine(x1, y1, x2, y2); //reference
    i=1; //initialize variables
    x=x1;
    y=y1;
    dx=x2-x1;
    dy=y2-y1;
    s1=dx>0?1:-1;
    s2=dy>0?1:-1;
    dx*=s1;
    dy*=s2;
    p=dy-(dx>>1); //initialize first error
    if(dy>dx) { int temp=dx; dx=dy; dy=temp;
                flag=1; } //interchange dx and dy taking lesser slope
    else flag=0;
    while(i<=dx)
    {
        fillPixel(x,y); //plot
        if(p>0) //above midpoint, diagonal move
        {
            p-=dx; //diagonal move
            if(flag) x+=s1; //update
            else y+=s2;
        }
        p+=dy; //orthogonal move
        if(flag) y+=s2; //update
        else x+=s1;
        ++i;
    }
}

int main(int argc, char** argv)
{
    int x1,y1,x2,y2;
    int gd=DETECT,gm;
    if(argc<5)
    {
        printf("[ ERROR ] Usage : x1 y1 x2 y2\n");
        return 1;
    }
    x1=atoi(argv[1]); //get arguments
    y1=atoi(argv[2]);
    x2=atoi(argv[3]);
    y2=atoi(argv[4]);
    initgraph(&gd, &gm, NULL); //initialize graphics
    grid();

    if(x1==x2) //check for trivial cases
        vertLine(x1, y1, y2);
    else if(y1==y2)
        horiLine(x1, x2, y1);
    else if(abs(x1-x2)==abs(y1-y2))
        diagLine(x1, y1, x2, y2);
    else mpLine(x1, y1, x2, y2); //call algorithm
    getchar();
    closegraph();
    return 0;
}
```

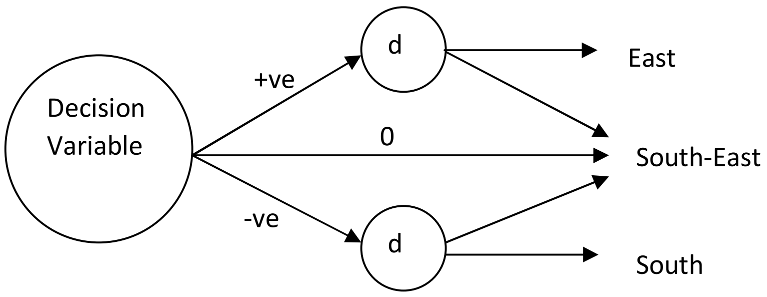
Output





Bresenham’s Circle Algorithm

Bresenham’s Circle is an iterative algorithm that rasterizes the first quadrant of a circle. The other 3 quadrants are plotted by symmetry. Here, we carry out this procedure from the point (0, r), and proceed clockwise. The algorithm uses 2 decision variables in each iteration; depending on the signs of these we get 5 possible branches: 3 “South-East Moves”, 1 “East Move” and 1 “South Move”



The first decision variable is calculated a point relative the current pixel. Based on its sign, one of two other decision variables is calculated, whose sign determines the final choice. The decision variable is then updated by a simple recursive formula.

An interesting detail is that the change to the first decision variable due to a South-East move, is the sum of the changes due to the East and South moves. Further, all increments/decrements to the decision variable are multiples to 2. In such a case, conditionals like

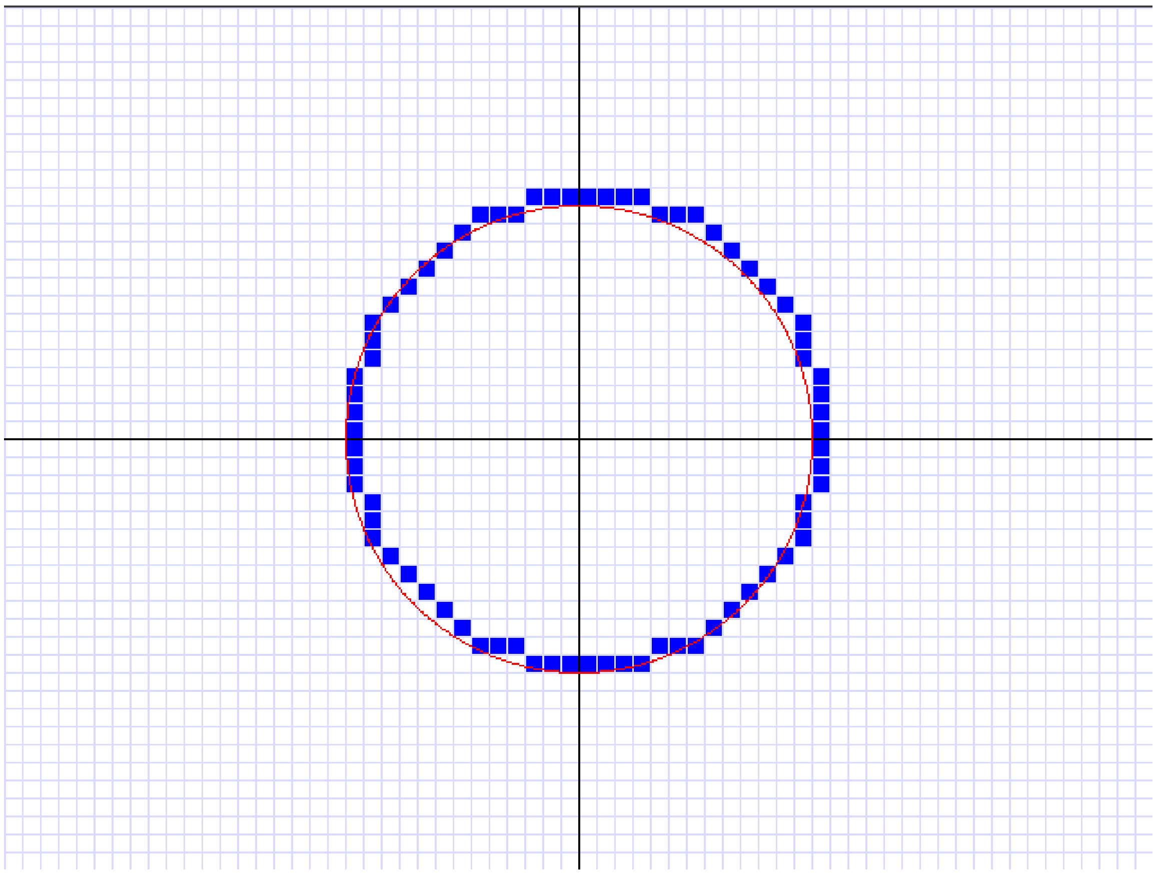
$$\boldsymbol{D - 1 > 0}$$
and
$$\boldsymbol{D > 0}$$

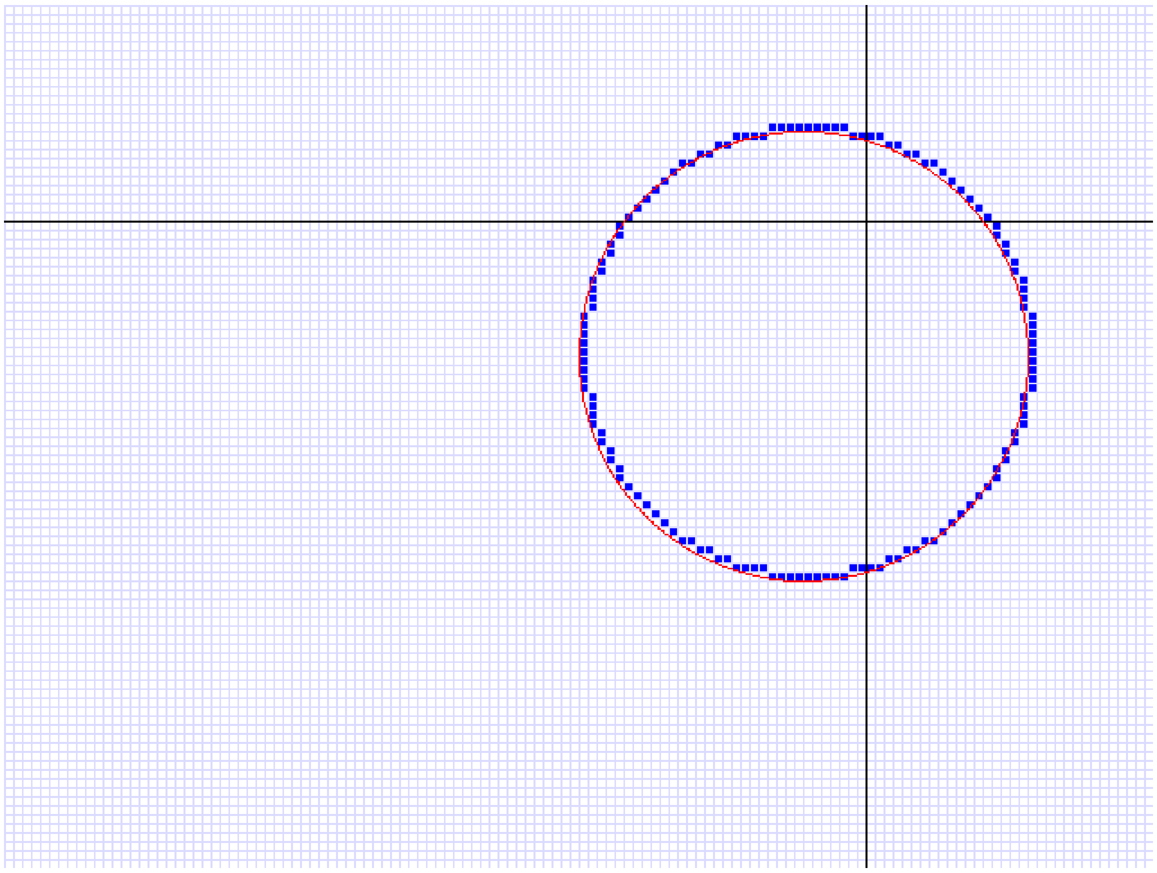
become identical. This greatly simplifies the iterative structure.

We show 3 variations of this algorithm (2 of which are commented out), all of which are mathematically identical and have the same behaviour. The first (bottom most one) is the original Bresenham’s 5 branch structure. We have taken advantage of the all changes being multiples to 2 to reduce computation and to calculate the 2nd level decision variable directly inside the conditional. We have also used auto-increment operators to show the structure concisely. The middle variant simply clubs the three south-east moves together. The final variant calculates the vertical and horizontal change to the decision variable separately and updates the variable at the end.

```
/*                        4.CIRCLE_BRESENHAM.C                              */
#include <stdio.h>
#include <stdlib.h>
#include "0.raster.c"

void bresenhamCircle(int xc, int yc, int r)
{
    int x, y, d, D;
    refEllipse(xc, yc, r, r); //reference
    x=0; //start at (0,r)
    y=r;
    D=((1-r)<<1); //initial error
    fillPixel(xc, yc+r); //plot starting points
    fillPixel(xc, yc-r);
    while(y) //rasterise 1st quadrant clockwise
    {
        d=0; //change in error
        if(D-x<=0) d+=((++x)<<1)+1; //horizontal move
        if(D+y>0) d+=1-((--y)<<1); //vertical move
        D+=d; //update error

/*    //Grouping the Diagonal Moves together
        if(D+y<=0) D+=((++x)<<1)+1; //move east
        else if(D-x>0) D+=1-((--y)<<1); //move south
        else D+=(((++x)-(--y)+1)<<1); //move south-east   */

/*   //Original structure
        //if (D<0) //move east or south-east
            if(D+y<=0) D+=((++x)<<1)+1; //move east
            //else D+=(((++x)-(--y)+1)<<1); //move south-east
        else if(D>0) //move south or south-east
            if(D-x<=0) D+=(((++x)-(--y)+1)<<1); //move south-east
            else D+=1-((--y)<<1); //move south
        else D+=(((++x)-(--y)+1)<<1); //move south-east   */

        fillPixel(xc+x, yc+y); //plot symmetric points
        fillPixel(xc+x, yc-y);
        fillPixel(xc-x, yc+y);
        fillPixel(xc-x, yc-y);
    }
}

int main(int argc, char** argv)
{
    int xc,yc,r;
    int gd=DETECT,gm;
    if(argc<4)
    {
        printf("[ ERROR ] Usage : x_center y_center radius\n");
        return 1;
    }
    xc=atoi(argv[1]); //get arguments
    yc=atoi(argv[2]);
    r=atoi(argv[3]);
    initgraph(&gd, &gm, NULL); //initialize graphics
    grid();
    bresenhamCircle(xc, yc, r); //call algorithm
    getchar();
    closegraph();
    return 0;
 }
```

Output



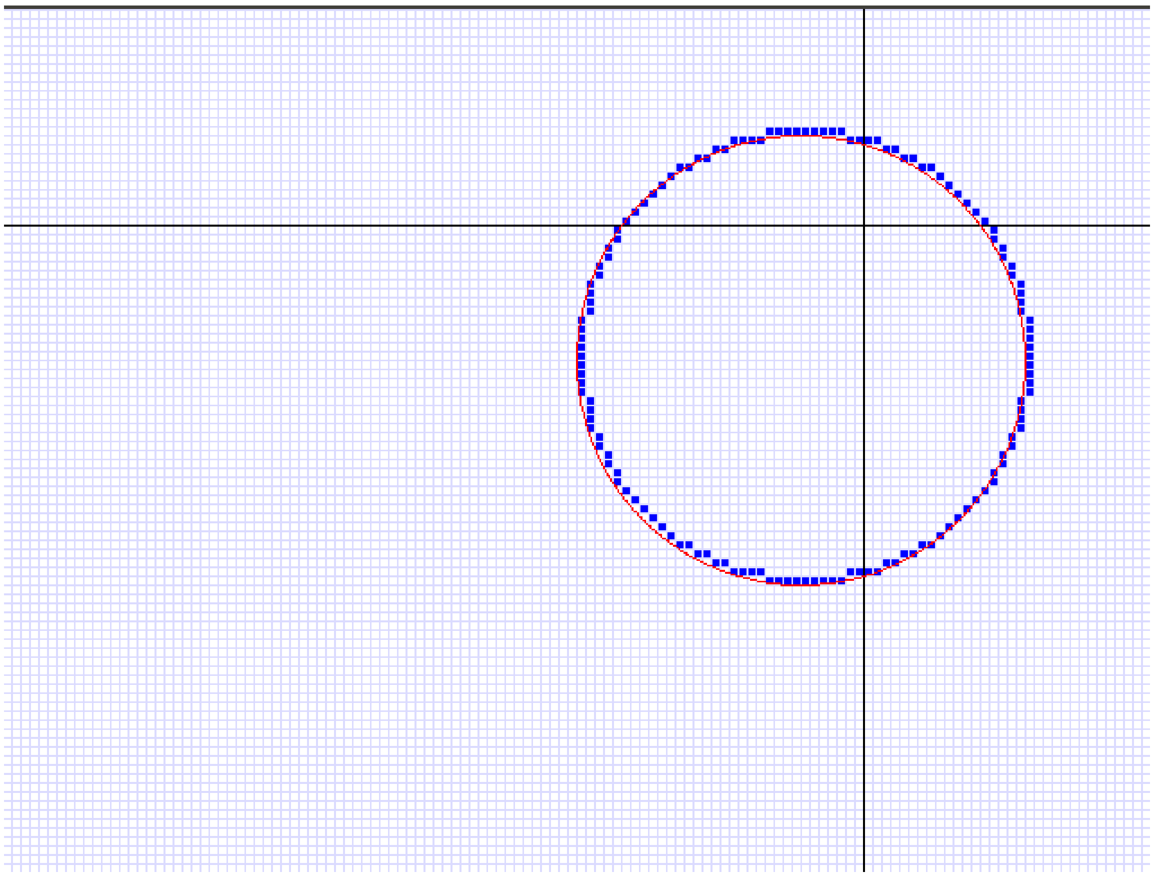


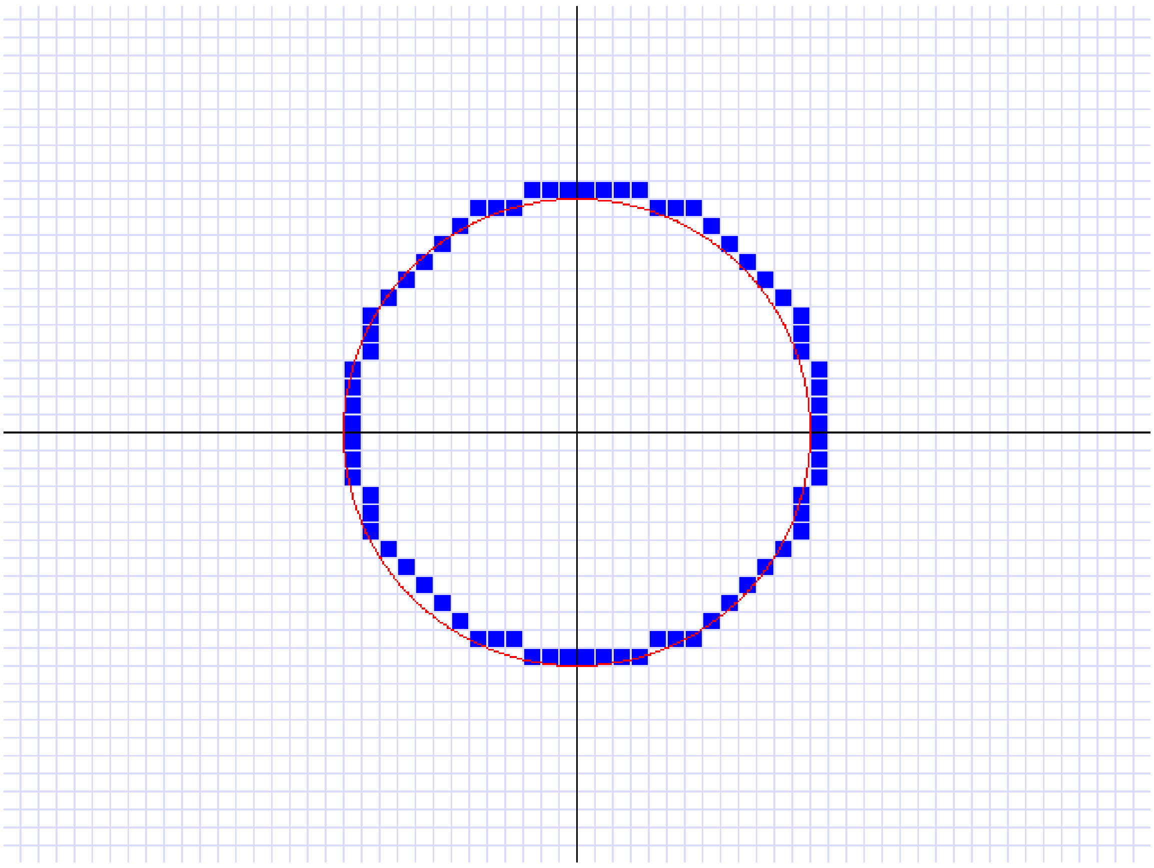
8-Way Symmetry Bresenham’s Circle

This is a variation of the Bresenham Circle Algorithm. Here, we rasterize the 2nd octant clockwise from the point (0, r). The condition y>x tells us when to stop. The remaining points are plotted by symmetry. Since we only rasterize the 2nd octant, the south move becomes impossible. Hence the first decision variable need not be checked. We must only check the 2nd level decision variable to decide whether to go east or south-east.

```c
/*                      4.CIRCLE_BRESENHAM_OCT.C                              */
#include <stdio.h>
#include <stdlib.h>
#include "0.raster.c"

void bresenhamCircle(int xc, int yc, int r)
{
    int x, y, d, D;
    refEllipse(xc, yc, r, r); //reference
    x=0; //start at (0,r)
    y=r;
    D=2-r; //initial error
    fillPixel(xc, yc+r); //plot starting points
    fillPixel(xc, yc-r);
    fillPixel(xc-r, yc);
    fillPixel(xc+r, yc);
    while(y>x) //rasterise 2nd octant clockwise
    {
        if(D>0) D-=((--y)<<1); //diagonal move
        D+=((++x)<<1)+1; //horizontal move
        fillPixel(xc+x, yc+y); //plot symmetric points
        fillPixel(xc+x, yc-y);
        fillPixel(xc-x, yc+y);
        fillPixel(xc-x, yc-y);
        fillPixel(xc+y, yc+x);
        fillPixel(xc+y, yc-x);
        fillPixel(xc-y, yc+x);
        fillPixel(xc-y, yc-x);
    }
}

int main(int argc, char** argv)
{
    int xc,yc,r;
    if(argc<4)
    {
        printf("[ ERROR ] Usage : x_center y_center radius\n");
        return 1;
    }
    xc=atoi(argv[1]); //get arguments
    yc=atoi(argv[2]);
    r=atoi(argv[3]);
    int gd=DETECT,gm; //initialize graphics
    initgraph(&gd, &gm, NULL);
    grid();
    bresenhamCircle(xc, yc, r); //call algorithm
    getchar();
    closegraph();
    return 0;
}
```

Output





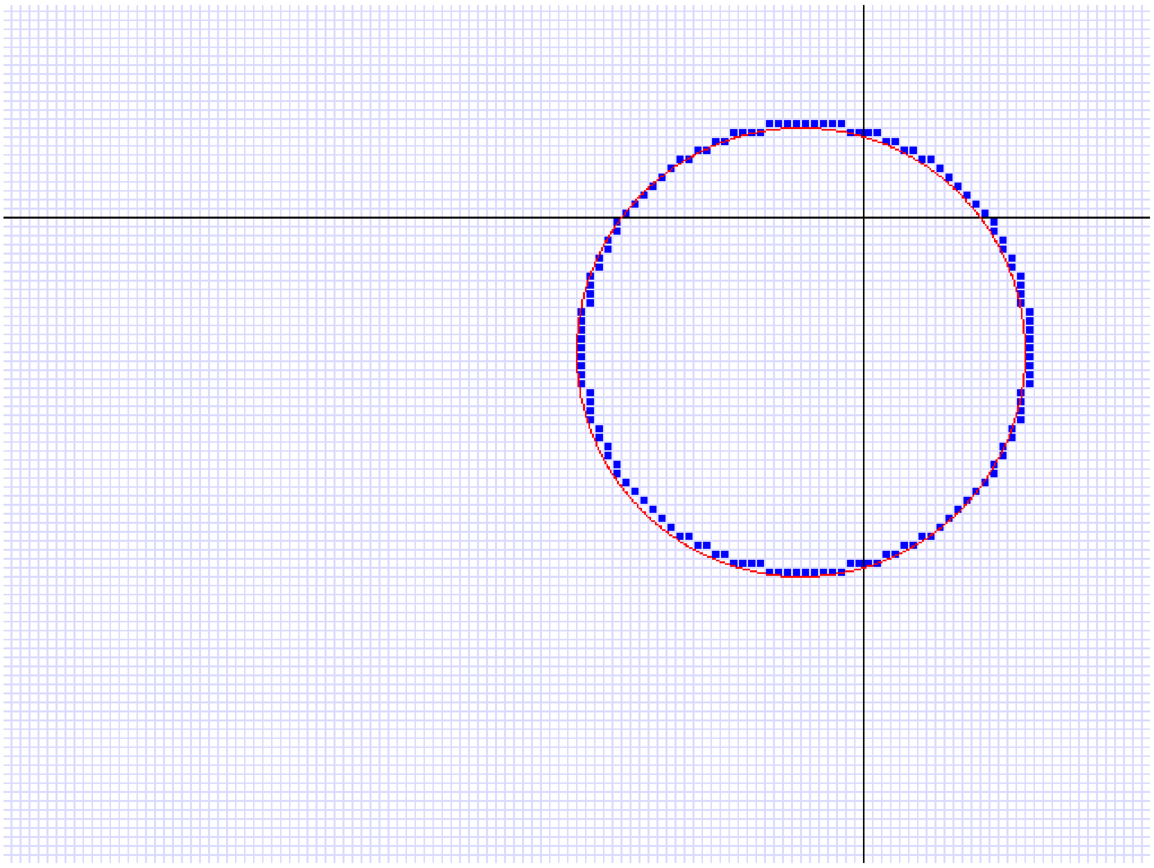
Mid-point Circle Algorithm

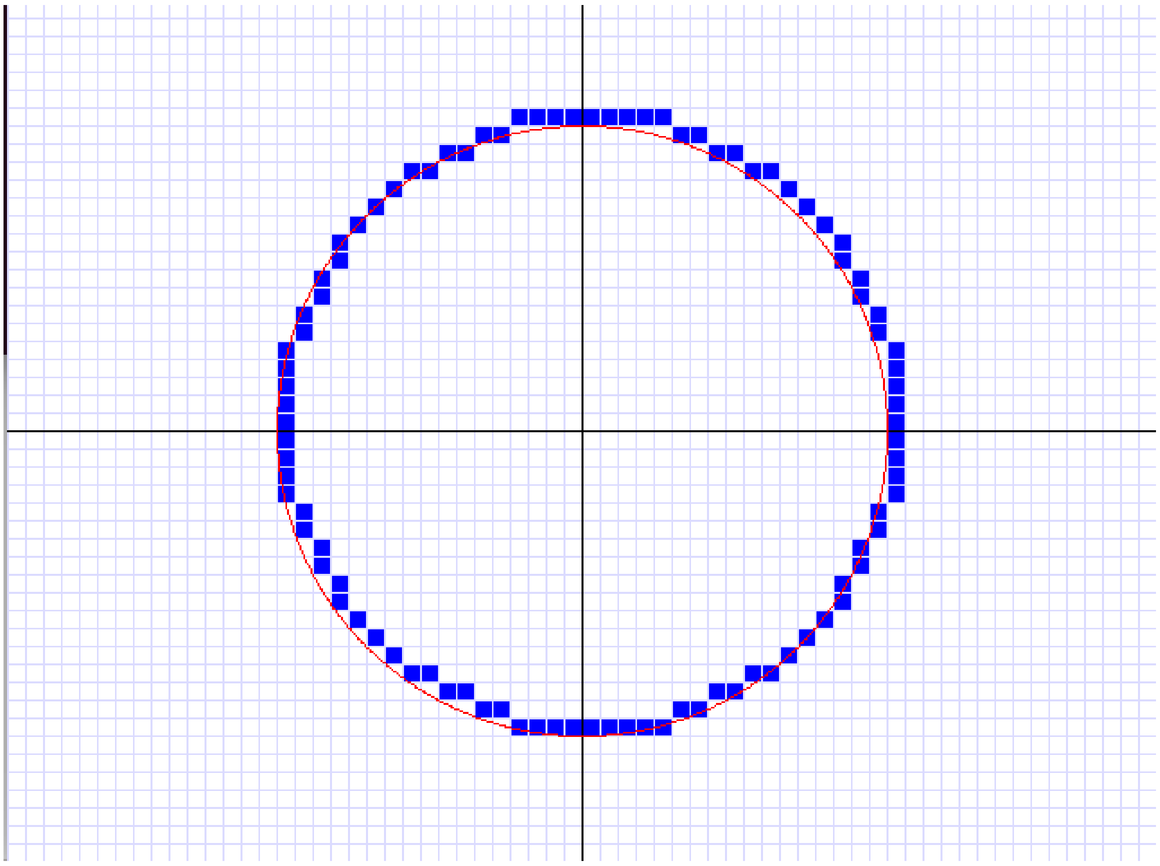
The Mid-point Circle Algorithm proceeds much like Bresenham's circle, except that we rasterize only 1 octant and plot the remaining points by symmetry. This simplifies the structure greatly as we now require only a single decision variable. At each point, we must choose between going east and going south-east. Hence our decision variable is situated at the mid-point of these two candidates. We make our decision based on the sign. The decision variable is then updated by a simple recursive procedure. However, this calculation requires the values of the co-ordinates of the current pixel.

This algorithm generalizes more easily to conic sections. Hence, it is preferred over Bresenham's for its simplicity.

```
/*                            5.MIDPNTCIRCLE.C                                  */
#include <stdio.h>
#include <stdlib.h>
#include "0.raster.c"

void mpCircle(int xc, int yc, int r)
{
    int x, y, d;
    refEllipse(xc, yc, r, r); //reference
    x=0; //start at (r,0)
    y=r;
    d=1-r; //initial error
    fillPixel(xc, yc+r); //plot starting points
    fillPixel(xc, yc-r);
    fillPixel(xc-r, yc);
    fillPixel(xc+r, yc);
    while(x<y) //rasterise 2nd octant
    {
        if(d>=0) d-=((--y)<<1); //update error diagonal move
        ++x; //rasterise clockwise
        d+=(x<<1)+1; //update error horizontal move
        fillPixel(xc+x, yc+y); //plot symmetric points
        fillPixel(xc+x, yc-y);
        fillPixel(xc-x, yc+y);
        fillPixel(xc-x, yc-y);
        fillPixel(xc+y, yc+x);
        fillPixel(xc+y, yc-x);
        fillPixel(xc-y, yc+x);
        fillPixel(xc-y, yc-x);
    }
}

int main(int argc, char** argv)
{
    int xc,yc,r;
    int gd=DETECT,gm;
    if(argc<4)
    {
        printf("[ ERROR ] Usage : x_center y_center radius\n");
        return 1;
    }
    xc=atoi(argv[1]); //get arguments
    yc=atoi(argv[2]);
    r=atoi(argv[3]);
    initgraph(&gd, &gm, NULL); //initialize graphics
    grid();
    mpCircle(xc, yc, r); //call algorithm
    getchar();
    closegraph();
    return 0;
}
```

Output



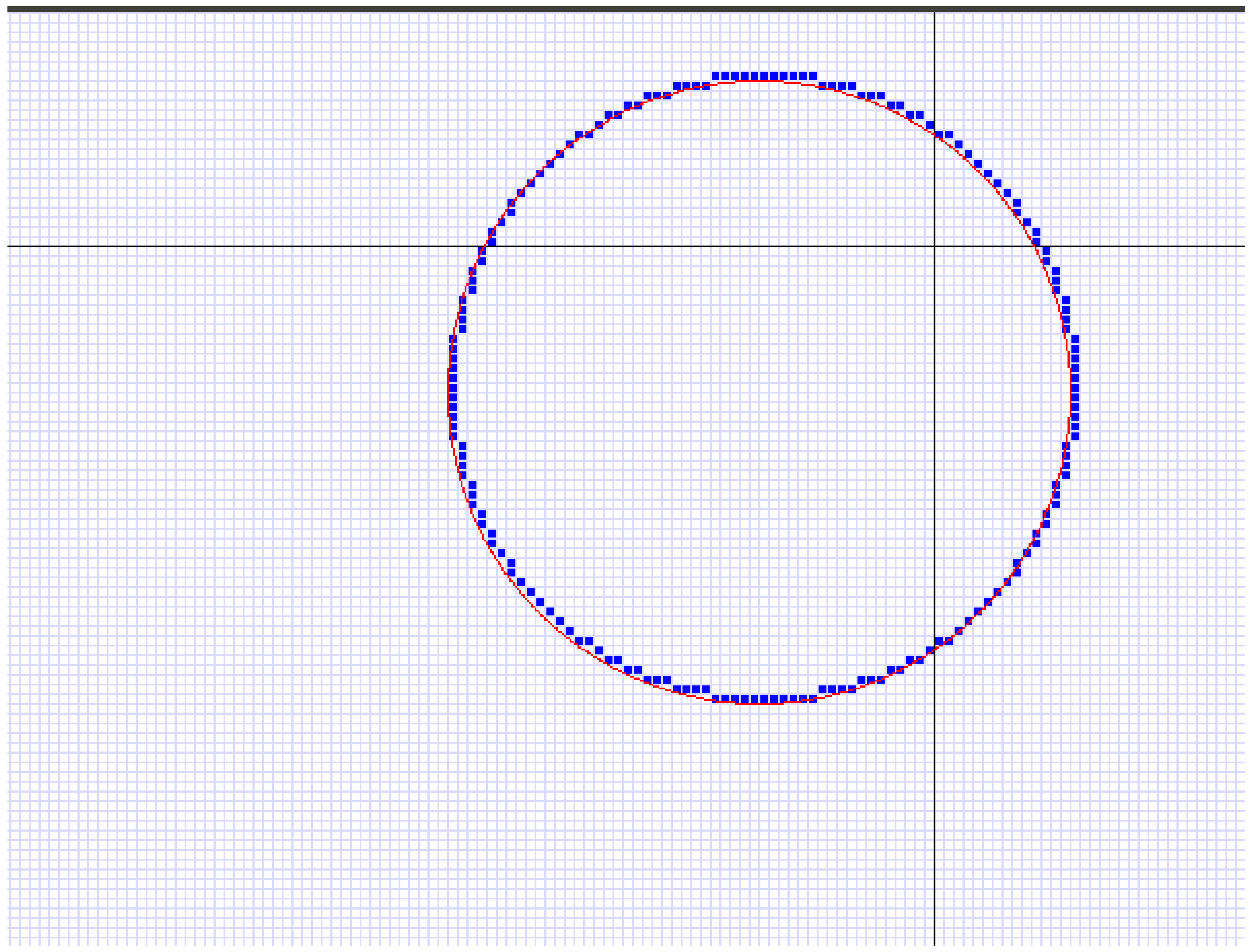


Mid-point Circle Variation

This is a variation of the Mid-point Circle Algorithm. Here, the horizontal and vertical increments are stored in separate variables and their values updated every iteration by a fixed amount. The east move causes only the horizontal increment to be added to the decision variable. The south-east move requires both these increments to be added.

```c
/*                        6.MIDPNTCIRCLE_D1.5X.C                              */
#include <stdio.h>
#include <stdlib.h>
#include "0.raster.c"

void mpCircle_d1p5x(int xc, int yc, int r)
{
    int x, y, d, dhorz, dvert;
    refEllipse(xc, yc, r, r); //reference
    x=0; //start at (r,0)
    y=r;
    d=1-r; //initial error
    dhorz=1; //initial change in error
    dvert=-(r<<1);
    fillPixel(xc, yc+r); //plot starting points
    fillPixel(xc, yc-r);
    fillPixel(xc-r, yc);
    fillPixel(xc+r, yc);
    while(x<y) //rasterise 2nd octant
    {
        if(d>=0) //diagonal move
        {
            --y;
            dvert+=2; //update change in error
            d+=dvert; //update error
        }
        ++x; //rasterise clockwise
        dhorz+=2; //update change in error
        d+=dhorz; //update error
        fillPixel(xc+x, yc+y); //plot symmetric points
        fillPixel(xc+x, yc-y);
        fillPixel(xc-x, yc+y);
        fillPixel(xc-x, yc-y);
        fillPixel(xc+y, yc+x);
        fillPixel(xc+y, yc-x);
        fillPixel(xc-y, yc+x);
        fillPixel(xc-y, yc-x);
    }
}

int main(int argc, char** argv)
{
    int xc,yc,r;
    int gd=DETECT,gm;
    if(argc<4)
    {
        printf("[ ERROR ] Usage : x_center y_center radius\n");
        return 1;
    }
    xc=atoi(argv[1]); //get arguments
    yc=atoi(argv[2]);
    r=atoi(argv[3]);
    initgraph(&gd, &gm, NULL); //initialize graphics
    grid();
    mpCircle_d1p5x(xc, yc, r); //call algorithm
    getchar();
    closegraph();
    return 0;
}
```

Output

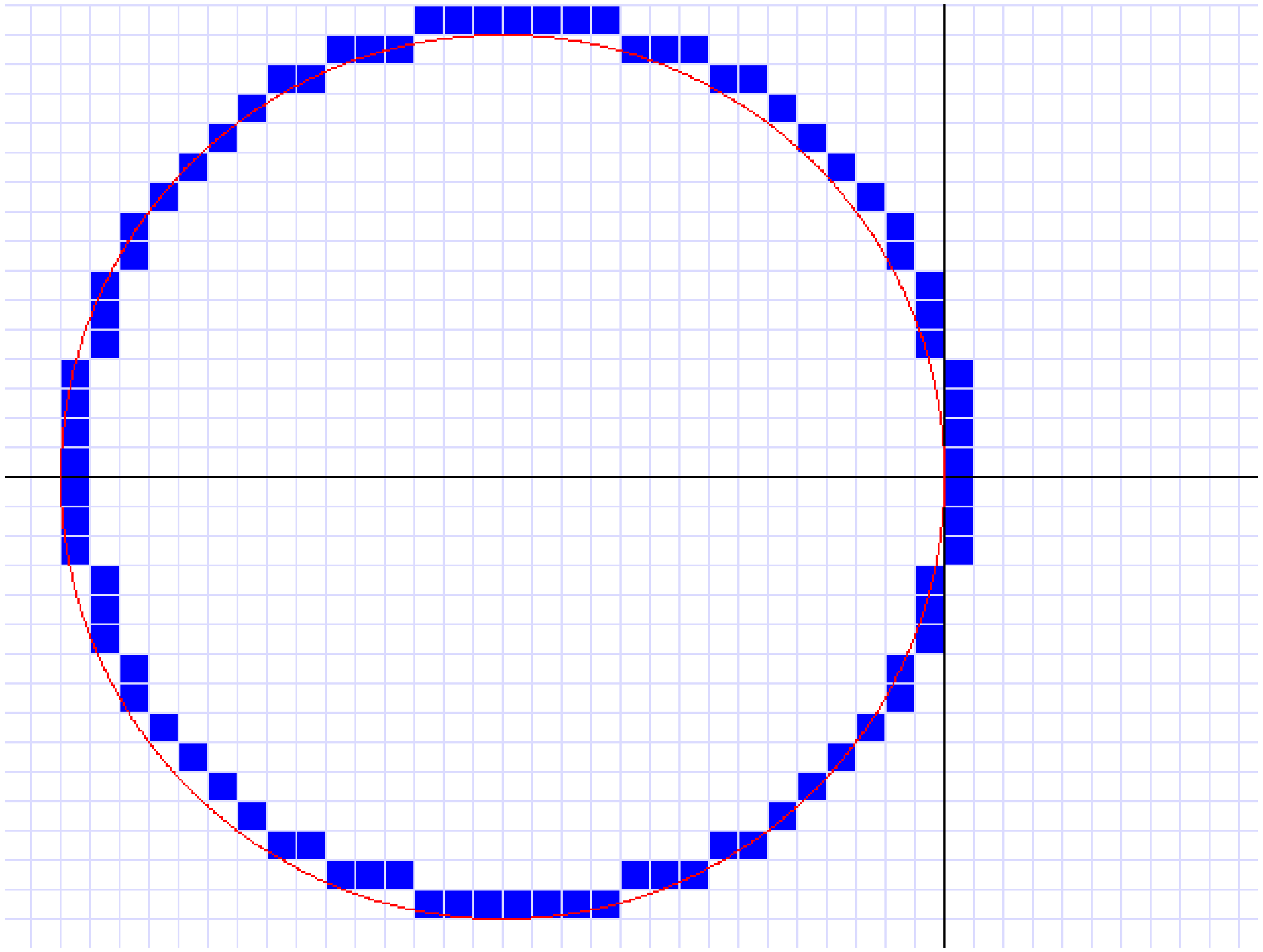


2nd Degree Mid-point Circle

This is a variation of the Mid-point Circle Algorithm. Here, the east and south increments are stored in separate variables and their values updated every iteration by a fixed amount. However this amount varies for the east and south-east moves. The decision variable is update with the values of these variables. Using this method removes the overhead on constantly finding out the co-ordinates of the current pixel and using its value in a computation.

```c
/*                           6.MIDPNTCIRCLE_D2X.C                            */
#include <stdio.h>
#include <stdlib.h>
#include "0.raster.c"

void mpCircle_d2p0x(int xc, int yc, int r)
{
    int x, y, d, dhorz, ddiag;
    refEllipse(xc, yc, r, r); //reference
    x=0; //start at (r,0)
    y=r;
    d=1-r; //initial error
    dhorz=1; //initial change in error
    ddiag=1-(r<<1);
    fillPixel(xc, yc+r); //plot starting points
    fillPixel(xc, yc-r);
    fillPixel(xc-r, yc);
    fillPixel(xc+r, yc);
    while(x<y) //rasterise 2nd octant
    {
        ++x; //rasterise clockwise
        dhorz+=2; //update change in error
        ddiag+=2;
        if(d<0) //horizontal move
            d+=dhorz; //update error
        else //diagonal move
        {
            --y;
            ddiag+=2; //update change in error
            d+=ddiag; //update error
        }
        fillPixel(xc+x, yc+y); //plot symmetric points
        fillPixel(xc+x, yc-y);
        fillPixel(xc-x, yc+y);
        fillPixel(xc-x, yc-y);
        fillPixel(xc+y, yc+x);
        fillPixel(xc+y, yc-x);
        fillPixel(xc-y, yc+x);
        fillPixel(xc-y, yc-x);
    }
}

int main(int argc, char** argv)
{
    int xc,yc,r;
    int gd=DETECT,gm;
    if(argc<4)
    {
        printf("[ ERROR ] Usage : x_center y_center radius\n");
        return 1;
    }
    xc=atoi(argv[1]); //get arguments
    yc=atoi(argv[2]);
    r=atoi(argv[3]);
    initgraph(&gd, &gm, NULL); //initialize graphics
    grid();
    mpCircle_d2p0x(xc, yc, r); //call algorithm
    getchar();
    closegraph();
    return 0;
 }
```

Output



Midpoint Ellipse Algorithm

This is an extension of the mid-point class of algorithms to a more complex conic section. The advantage of this method is that only addition operations are required in the iterative steps. This leads to simple and fast implementation on all processors.

Here, we rasterize the 1st quadrant of the ellipse clockwise from the point (0, b). In region 1, the x co-ordinate changes faster that the y co-ordinate. The opposite is true for region 2. Hence, slightly different handling is required for the two. At the beginning, we are forced to make some expensive calculations involving multiplication and division. However, this is a one-time overhead and is acceptable. The remaining calculations in region involve a decision variable at the mid-point of the two candidates for the next pixel (the east and south-east moves).

When we switch regions, the mid-point ellipse algorithm recalculates the decision variable at the border region, with multiplications and divisions. However, this can be avoided and the region change calculation reduced to a simpler addition (there are some multiplications by 2 involved, but these are highly efficient for binary representations).

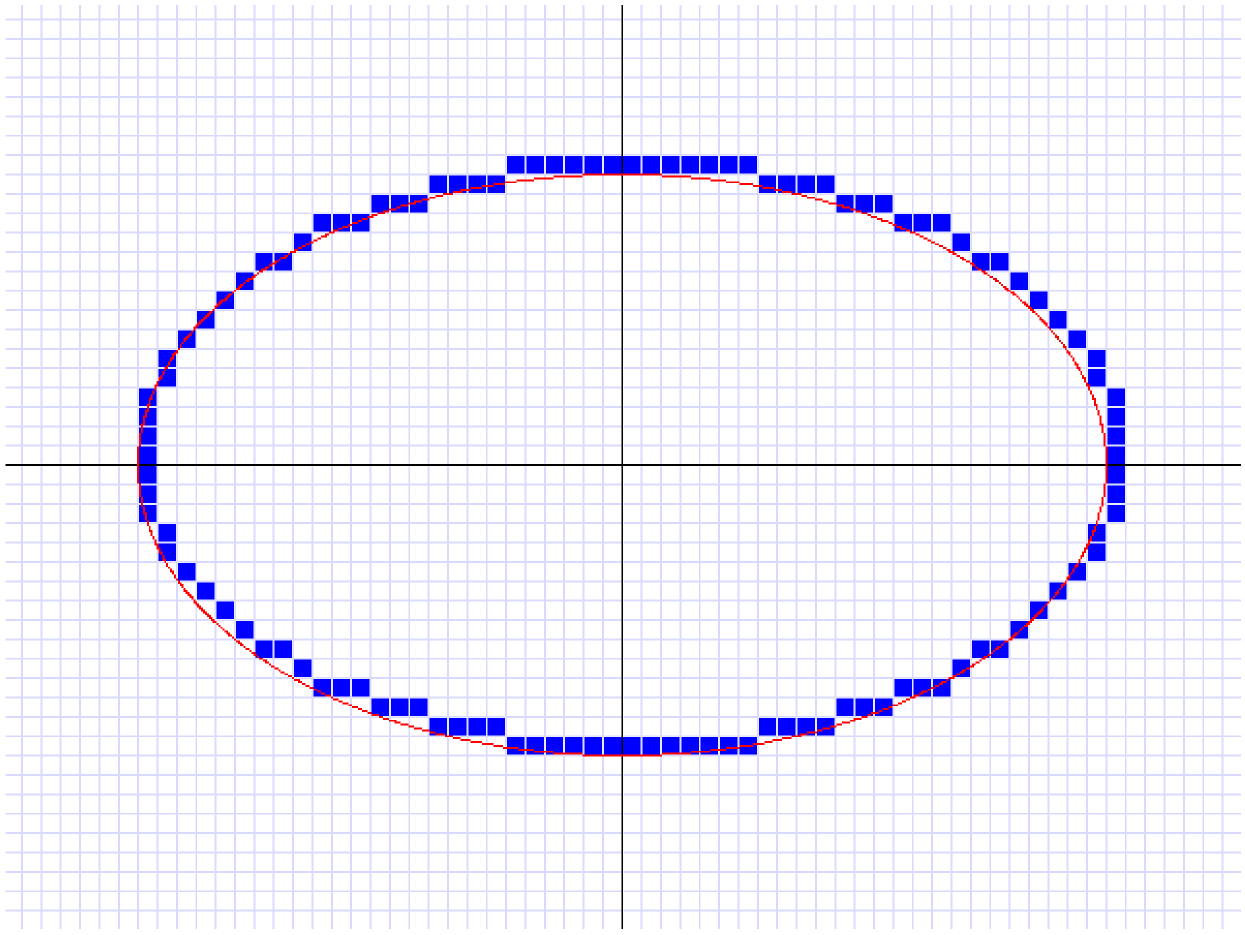
The procedure for region 2 is similar to region 1, except that our choices are now between the south and south-east moves. The recursive calculations are also different.

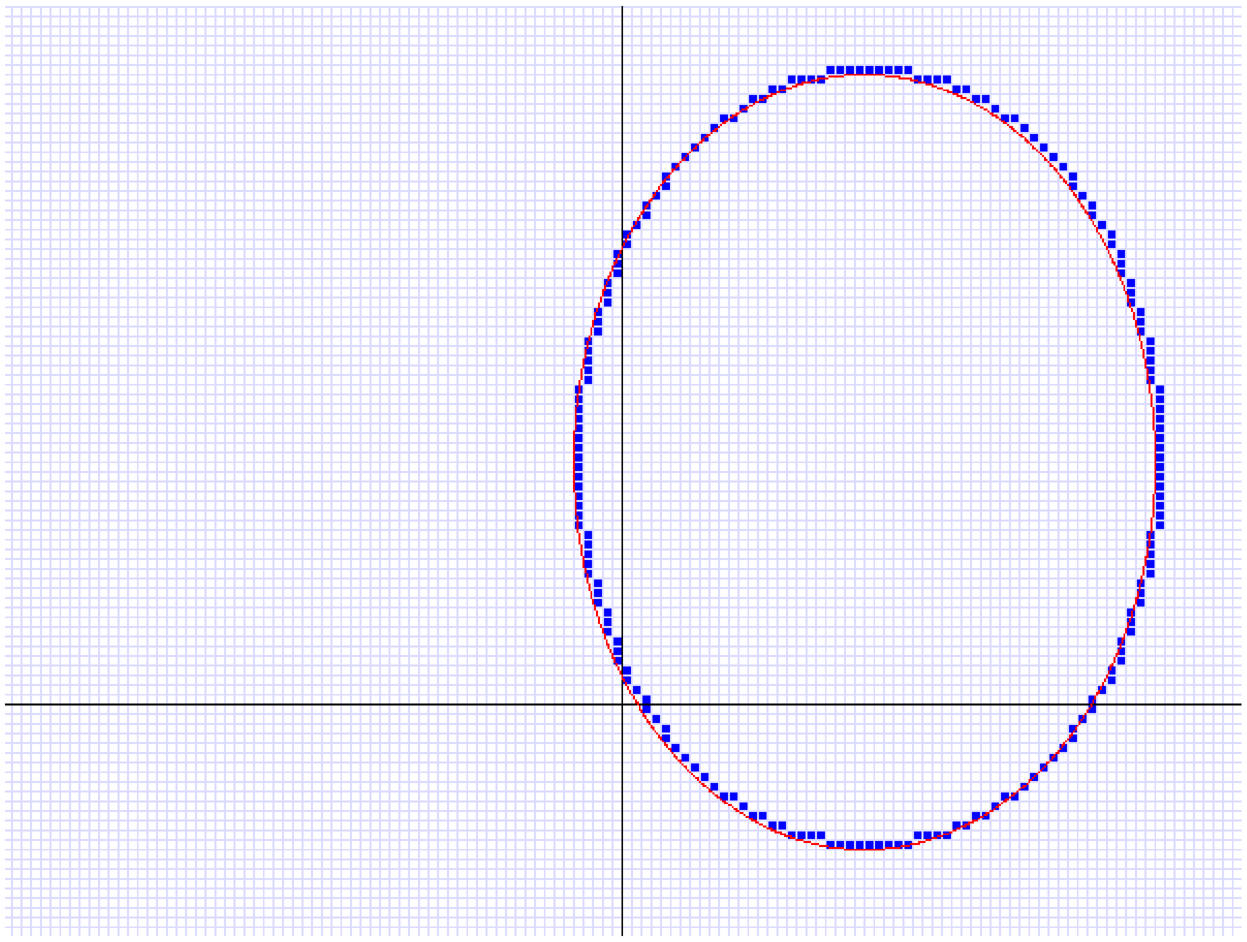
The algorithm terminates when y becomes 0. The remaining three quadrants are plotted simultaneously by symmetry.

```c
/*                            7.MIDPNTELLIPSE.C                                */
#include <stdio.h>
#include <stdlib.h>
#include "0.raster.c"

void mpEllipse(int xc, int yc, int a, int b)
{
    int x, y, dp, dhorz, dvert;
    int Asq, Bsq, fourAsq, fourBsq;
    refEllipse(xc, yc, a, b); //reference
    Asq=a*a; //calculate constants
    Bsq=b*b;
    fourAsq=(Asq<<2);
    fourBsq=(Bsq<<2);
    x=dhorz=0; //start ar (0,b)
    y=b;
    dvert=((fourAsq*b)<<1);
    fillPixel(xc, yc+b); //plot initial points
    fillPixel(xc, yc-b);
    dp=fourBsq-(fourAsq*b)+Asq; //calculate initial error
    while(dhorz<dvert) //in region 1
    {
        if(dp>=0)
        {
            --y;
            dvert-=(fourAsq<<1); //update change in error
            dp-=dvert; //update error diagonal move
        }
        ++x; //rasterise clockwise
        dhorz+=(fourBsq<<1); //update change in error
        dp+=fourBsq+dhorz; //update error horizontal move
        fillPixel(xc+x, yc+y); //plot symmetric points
        fillPixel(xc+x, yc-y);
        fillPixel(xc-x, yc+y);
        fillPixel(xc-x, yc-y);
    }
    dp+=Bsq-(fourBsq*x)+Asq-(fourAsq*y); //update error for region 2
    while(y>0) //in region 2
    {
        if(dp<=0)
        {
            ++x;
            dhorz+=(fourBsq<<1); //update change in error
            dp+=dhorz; //update error diagonal move
        }
        --y;
        dvert-=(fourAsq<<1); //update change in error
        dp+=fourAsq-dvert; //update error vertical move
        fillPixel(xc+x, yc+y); //plot symmetric points
        fillPixel(xc+x, yc-y);
        fillPixel(xc-x, yc+y);
        fillPixel(xc-x, yc-y);
    }
}
```

```c
int main(int argc, char** argv)
{
    int xc, yc, a, b;
    int gd=DETECT,gm;
    if(argc<5)
    {
        printf("[ ERROR ] Usage : x_center y_center a b\n");
        return 1;
    }
    xc=atoi(argv[1]); //get arguments
    yc=atoi(argv[2]);
    a=atoi(argv[3]);
    b=atoi(argv[4]);
    initgraph(&gd, &gm, NULL); //initialize graphics
    grid();
    mpEllipse(xc, yc, a, b); //call algorithm
    getchar();
    closegraph();
    return 0;
}
```

Output





```c
/*                           0.KPGPARSE.C
*
* FORMAT FOR .kpg (KANTI PANDEY GRAPHICS) FILES
*
* FORMAT IS NOT CASE SENSITIVE
*
* # ...                        => COMMENT (BEGINNING OF LINE)
* L X1 Y1 X2 Y2                => DEFINES A LINE
* P N X1 Y1 X2 Y2 ...          => DEFINES A POLYGON WITH N VERTICES (INPUT IN CLOCKWISE ORDE
* S X Y                        => DEFINES A SEED POINT
* C X Y R                      => DEFINES A CIRCLE
* E X Y A B                    => DEFINES AN ORTHOGONAL ELLIPSE
*
* G N X1 Y1 Z1                 => DEFINES A 3D OBJECT GRAPH
*     X2 Y2 Z2                    (INPUT AS VERTICES AND ADJACENCY MATRIX)
*     .
*     .
*     .
*     A11 A12 A13 ...
*     A21 A22 A23 ...
*     .
*     .
*     .
*
*/
#ifndef Soumya_Kanti_Naskar_001410501044_and_Sahil_Pandey_001410501057_Graphics_Format
#define Soumya_Kanti_Naskar_001410501044_and_Sahil_Pandey_001410501057_Graphics_Format
#include <stdio.h>
#include <stdlib.h>

struct point
{
    int x, y;
    struct point *next;
};

typedef struct point *SHAPE;

void erase(SHAPE shape)
{
    SHAPE temp;
    while(shape)
    {
        temp=shape;
        shape=shape->next;
        free(temp);
    }
}

SHAPE parseComment(FILE* kpg)
{
    char c;
    while((c=fgetc(kpg))!='\n');
    return NULL;
}

SHAPE parseLine(FILE *kpg)
{
    SHAPE v1=(SHAPE)malloc(sizeof(struct point));
    SHAPE v2=(SHAPE)malloc(sizeof(struct point));
    if(fscanf(kpg, "%d %d %d %d ", &(v1->x), &(v1->y), &(v2->x), &(v2->y))!=4)
    {
        free(v1);
        free(v2);
        return NULL;
    }
    v1->next=v2;
    v2->next=NULL;
    return v1;
}
```

```
SHAPE parsePoly(FILE *kpg)
{
    int n, i;
    SHAPE pre=NULL;
    fscanf(kpg, "%d ", &n);
    for(i=0; i<n; ++i)
    {
        SHAPE v=(SHAPE)malloc(sizeof(struct point));
        if(fscanf(kpg, "%d %d ", &(v->x), &(v->y))!=2)
        {
            erase(pre);
            free(v);
            return NULL;
        }
        v->next=pre;
        pre=v;
    }
    return pre;
}

SHAPE parseSeed(FILE *kpg)
{
    SHAPE v=(SHAPE)malloc(sizeof(struct point));
    if(fscanf(kpg, "%d %d ", &(v->x), &(v->y))!=2)
    {
        free(v);
        return NULL;
    }
    v->next=NULL;
    return v;
}

SHAPE parseCircle(FILE *kpg)
{
    SHAPE c=(SHAPE)malloc(sizeof(struct point));
    SHAPE r=(SHAPE)malloc(sizeof(struct point));
    if(fscanf(kpg, "%d %d %d ", &(c->x), &(c->y), &(r->x))!=3)
    {
        free(c);
        free(r);
        return NULL;
    }
    r->y=r->x;
    c->next=r;
    r->next=NULL;
    return c;
}

SHAPE parseEllipse(FILE *kpg)
{
    SHAPE c=(SHAPE)malloc(sizeof(struct point));
    SHAPE r=(SHAPE)malloc(sizeof(struct point));
    if(fscanf(kpg, "%d %d %d %d ", &(c->x), &(c->y), &(r->x), &(r->y))!=4)
    {
        free(c);
        free(r);
        return NULL;
    }
    c->next=r;
    r->next=NULL;
    return c;
}
```

```
SHAPE expectLine(FILE *kpg)
{
    char c;
    fscanf(kpg, "%c ", &c);
    if(c=='l' || c=='L')
        return parseLine(kpg);
    else if(c=='#')
    {
        parseComment(kpg);
        return expectLine(kpg);
    }
    else return NULL;
}

SHAPE expectPoly(FILE *kpg)
{
    char c;
    fscanf(kpg, "%c ", &c);
    if(c=='p' || c=='P')
        return parsePoly(kpg);
    else if(c=='#')
    {
        parseComment(kpg);
        return expectPoly(kpg);
    }
    else return NULL;
}

SHAPE expectSeed(FILE *kpg)
{
    char c;
    fscanf(kpg, "%c ", &c);
    if(c=='s' || c=='S')
        return parseSeed(kpg);
    else if(c=='#')
    {
        parseComment(kpg);
        return expectSeed(kpg);
    }
    else return NULL;
}

SHAPE expectCircle(FILE *kpg)
{
    char c;
    fscanf(kpg, "%c ", &c);
    if(c=='c' || c=='C')
        return parseCircle(kpg);
    else if(c=='#')
    {
        parseComment(kpg);
        return expectCircle(kpg);
    }
    else return NULL;
}

SHAPE expectEllipse(FILE *kpg)
{
    char c;
    fscanf(kpg, "%c ", &c);
    if(c=='e' || c=='E')
        return parseEllipse(kpg);
    else if(c=='#')
    {
        parseComment(kpg);
        return expectEllipse(kpg);
    }
    else return NULL;
}
```

```c
struct vpoint //2d points
{
    double x, y;
    struct vpoint *next;
};

typedef struct vpoint *VSHAPE;

VSHAPE toVector(SHAPE s)
{
    SHAPE t;
    VSHAPE v=NULL;
    for(t=s; t; t=t->next)
    {
        VSHAPE m = (VSHAPE)malloc(sizeof(struct vpoint));
        m->x=t->x;
        m->y=t->y;
        m->next=v;
        v=m;
    }
    erase(t);
    return v;
}

void eraseVector(VSHAPE shape)
{
    VSHAPE temp;
    while(shape)
    {
        temp=shape;
        shape=shape->next;
        free(temp);
    }
}

typedef struct vgraph //3d object
{
    int n, **adj;
    double **vertices;
}VGRAPH;

void eraseGraph(VGRAPH shape)
{
    free(shape.vertices[0]);
    free(shape.vertices);
    free(shape.adj[0]);
    free(shape.adj);
}
```

```
VGRAPH parseGraph(FILE *kpg)
{
    VGRAPH graph;
    int i, j;
    if(fscanf(kpg, "%d ", &graph.n)!=1)
    {
        graph.n=0;
        return graph;
    }
    graph.vertices = (double**)malloc(graph.n*sizeof(double*));
    graph.vertices[0] = (double*)malloc(graph.n*3*sizeof(double));
    for(i=0; i<graph.n; ++i)
    {
        graph.vertices[i] = graph.vertices[0] + 3*i;
        if(fscanf(kpg, "%lf %lf %lf", &graph.vertices[i][0],
                                      &graph.vertices[i][1],
                                      &graph.vertices[i][2] ) !=3)));
        {
            free(graph.vertices[0]);
            free(graph.vertices);
            graph.n=0;
            return graph;
        }
    }
    graph.adj = (int**)malloc(graph.n*sizeof(int*));
    graph.adj[0] = (int*)malloc(graph.n*graph.n*sizeof(int
    for(i=0; i<graph.n; ++i)
    {
        graph.adj[i] = graph.adj[0] + graph.n*i;
        for(j=0; j<graph.n; ++j)
            if(fscanf(kpg, "%d ", &graph.adj[i][j])!=1)
            {
                free(graph.vertices[0]);
                free(graph.vertices);
                free(graph.adj[0]);
                free(graph.adj);
                graph.n=0;
                return graph;
            }
    }
    return graph;
}

VGRAPH expectGraph(FILE *kpg)
{
    char c;
    fscanf(kpg, "%c ", &c);
    if(c=='g' || c=='G')
        return parseGraph(kpg);
    else if(c=='#')
    {
        parseComment(kpg);
        return expectGraph(kpg);
    }
    else
    {
        VGRAPH g;
        g.n=0;
        return g;
    }
}

#endif
```

Scan Line Conversion

A linked list based Active Edge List Scan Line Conversion program. The lines are input through a file.

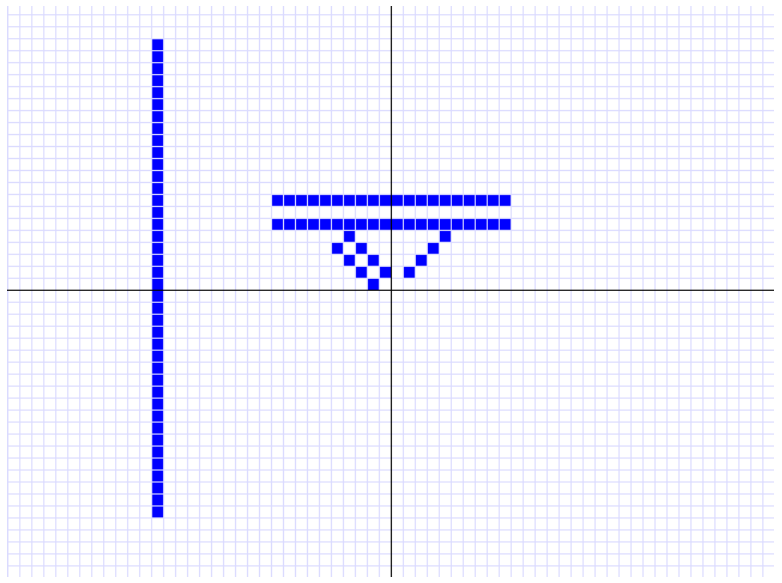
The case of horizontal lines is handled by modifying the y-bucket entry suitably. The case of near horizontal lines is handled by iterating a number of time in each scan line.

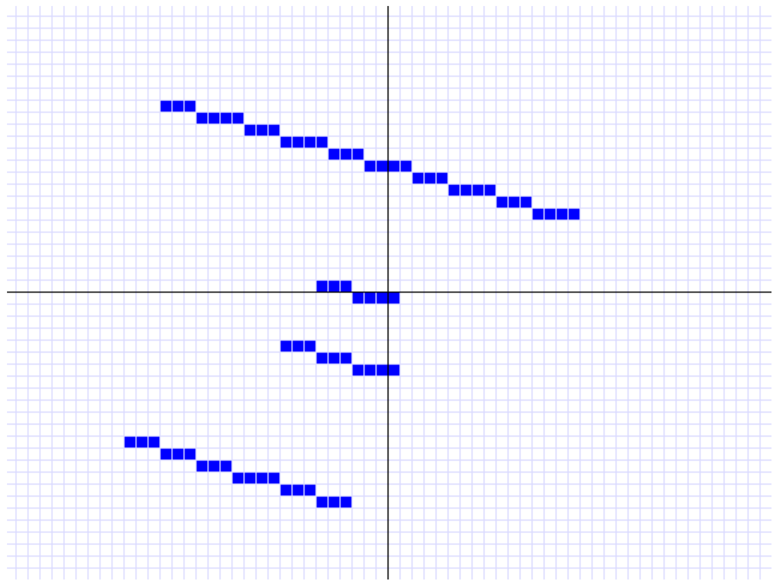
```c
/*                              8.SCANLINES.C                              */
#include <stdio.h>
#include <graphics.h>
#include <stdlib.h>
#include "0.kpgparse.c"
#include "0.raster.c"

int gif(double d) //greatest integer function
{
    if(d<0 && (int)d>d)
        return (int)d-1;
    return (int)d;
}

int numCol, numSL, top, left;
int *SL=NULL; //single scan line

typedef struct edge
{
    double x, dx;
    int dy;
    struct edge *next;
}EDGE;

EDGE **buckets=NULL, *ael=NULL;

void fillBucket(SHAPE shape)
{
    int dy;
    double dx;
    EDGE *add;
    int x1=shape->x;
    int y1=shape->y;
    int x2=shape->next->x;
    int y2=shape->next->y;
    erase(shape);
    if(y2>y1) { int t=x1; x1=x2; x2=t;
                t=y1; y1=y2; y2=t; } //P1 is the upper point
    dy = y1-y2;
    if(dy) dx=(x2-x1)*1.0/dy;
    else //horizontal line
    {
        if(x1>x2) { int t=x1; x1=x2; x2=t; } //P1 is the leftmost point
        dx=x2-x1;
    }
    if(y1<top-numSL || y2>top) return; //totally above or below
    if(y1>top) //clipping needed
    {
        x1 += dx*(y1-top); //calculate
        dy = top-y2;
        y1 = top;
    }
    add = (EDGE*)malloc(sizeof(EDGE));
    add->x=x1;
    add->dx=dx;
    add->dy=dy;
    add->next=buckets[top-gif(y1)]; //add to the bucket
    buckets[top-gif(y1)]=add;
}
```

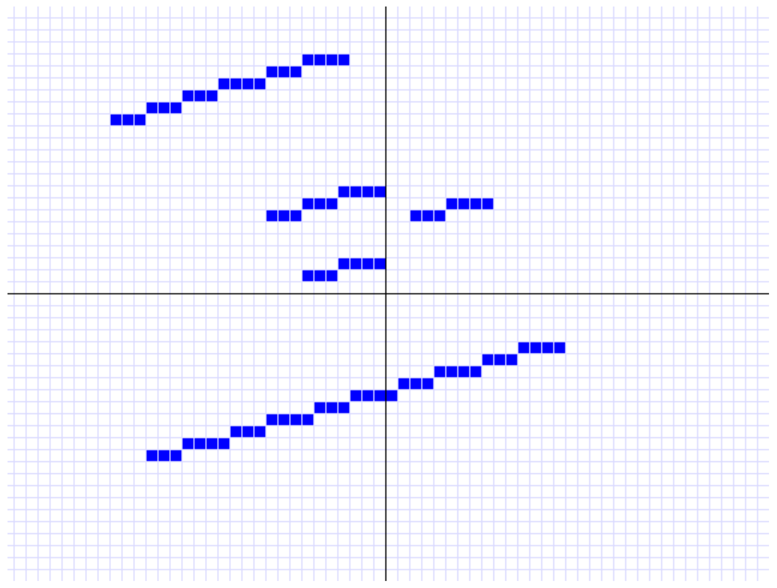
```
void scanConvert()
{
    EDGE *cur;
    int i, j, var;
    for(i=0; i<numSL; ++i)
    {
        for(j=0; j<numCol; ++j) SL[j]=0; //clear the Scan Line
        for(cur=buckets[i]; cur; cur=cur->next)
            if(cur->dy==0) //horizontal line
            {
                for(j=0; j<cur->dx; ++j)
                    SL[gif(cur->x+j)+left]=1;
            }
            else //add to Active Edge List
            {
                EDGE *temp = (EDGE*)malloc(sizeof(EDGE));
                temp->x=cur->x;
                temp->dx=cur->dx;
                temp->dy=cur->dy;
                temp->next=ael;
                ael=temp;
            }
        for(cur=ael; cur; cur=cur->next)
            if(cur->dy > 0)
            {
                if(cur->dx > 1) //not steep slope, multiple points
                    for(var=gif(cur->x+cur->dx); var > gif(cur->x); --var)
                        SL[var+left]=1;
                else if(cur->dx < -1) //not steep slope, multiple points
                    for(var=gif(cur->x+cur->dx); var < gif(cur->x); ++var)
                        SL[var+left]=1;
                else SL[gif(cur->x)+left]=1; //steep slope, single point
                --cur->dy; //update
                cur->x+=cur->dx;
            }
        for(j=0; j<numCol; ++j)
            if(SL[j]) fillPixel(j-left, top-i); //display Scan Line
    }
}
```

```c
int main(int argc, char **argv)
{
    SHAPE T;
    int i;
    int gd=DETECT,gm;
    FILE *kpg;
    if(argc<2)
    {
        printf("[ ERROR ] Usage : <.kpg>_format_input_file\n");
        return 1;
    }
    kpg = fopen(argv[1], "rx");
    if(!kpg)
    {
        printf("[ ERROR ] Usage : <.kpg>_format_input_file\n");
        return 1;
    }
    initgraph(&gd, &gm, NULL); //initialize graphics
    grid();
    numSL = (top=raster.centerY/raster.width)
          + (480-raster.centerY)/raster.width; //number of Scan Lines
    SL = (int*)malloc(sizeof(int)*(numCol=(left=raster.centerX/raster.width)
       +(640-raster.centerX)/raster.width)); //Number of Pixels per Line
    --top; --left;
    buckets = (EDGE**)malloc(sizeof(EDGE*)*numSL); //initialize buckets
    for(i=0; i<numSL; ++i) buckets[i]=NULL;
    while(T=expectLine(kpg)) fillBucket(T); //add lines to respective buckets
    fclose(kpg);
    scanConvert(); //call algorithm
    getchar();
    closegraph();
    for(i=0; i<numSL; ++i) //free memory
    {
        EDGE *E=buckets[i];
        while(E)
        {
            buckets[i]=E;
            E=E->next;
            free(buckets[i]);
        }
    }
    free(buckets);
    free(ael);
    free(SL);
    return 0;
}
```

Output





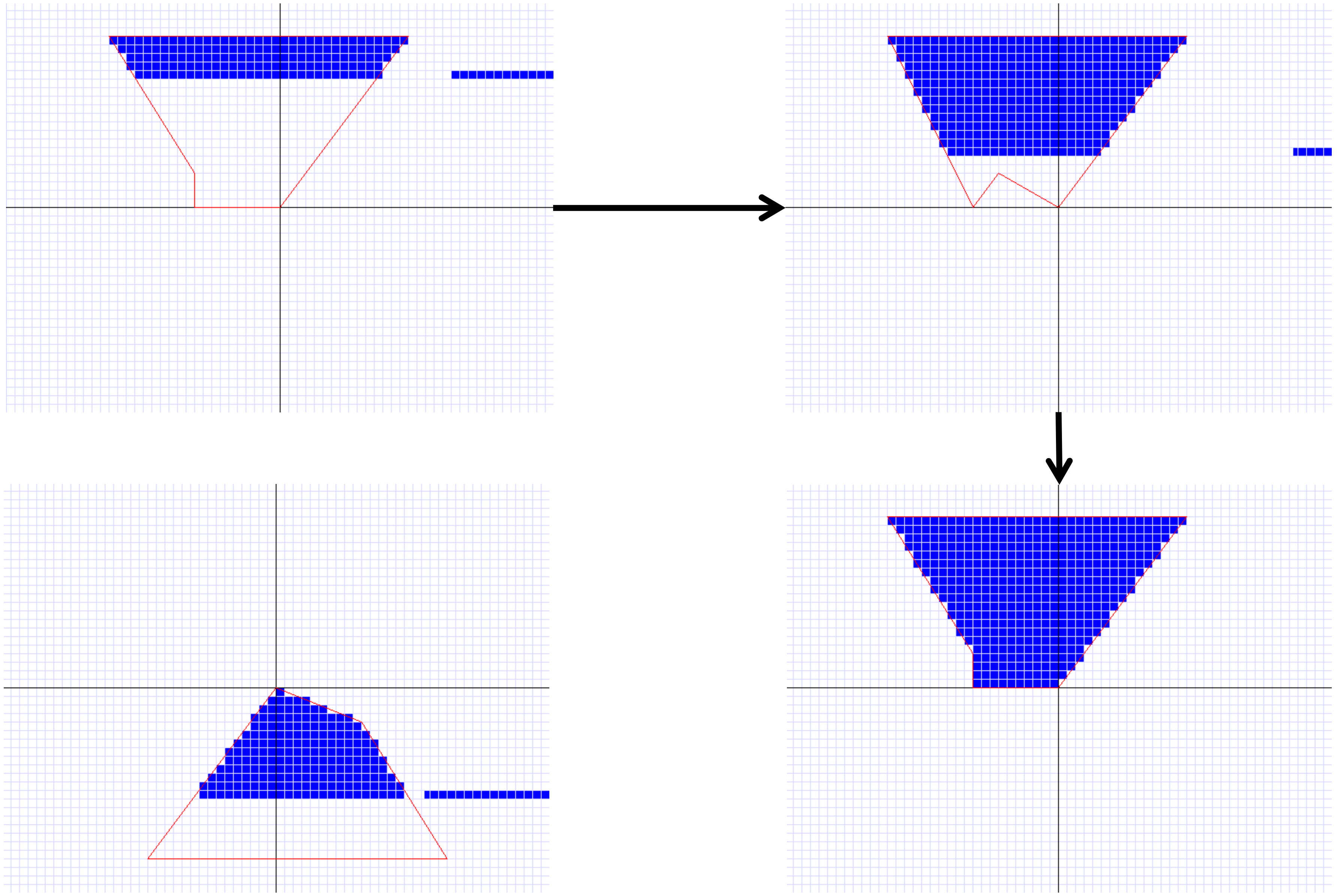


Edge Fill Polygon Filling

In polygon filling, any pixel whose center lies within the actual polygon must be highlighted. The Edge Fill Algorithm accomplishes this by repeatedly toggling all pixels to the right of each edge. This algorithm has a simple structure, but due to the repeated iterations is very inefficient. This is a scan line filling technique.

```
/*                        9.EDGEFILL.C                                      */
#include <stdio.h>
#include <graphics.h>
#include <stdlib.h>
#include "0.kpgparse.c"
#include "0.raster.c"

int gif(double d) //greatest integer function
{
    if(d<0 && (int)d>d)
        return (int)d-1;
    return (int)d;
}

int numCol, numSL, top, left;
int *SL=NULL; //single scan line

typedef struct edge
{
    double x, dx;
    int dy;
    struct edge *next;
}EDGE;

EDGE **buckets=NULL, *ael=NULL;

void fillBucket(SHAPE shape)
{
    int dy;
    double dx;
    EDGE *add;
    int x1=shape->x;
    int y1=shape->y;
    int x2=shape->next->x;
    int y2=shape->next->y;
    refLine(x1, y1, x2, y2); //polygon boundaries
    if(y2==y1) return; //horizontal lines need not be added
    if(y2>y1) { int t=x1; x1=x2; x2=t;
                t=y1; y1=y2; y2=t; } //P1 is the upper point
    dy = y1-y2;
    dx=(x2-x1)*1.0/dy;
    if(y1<top-numSL || y2>top) return; //totally above or below
    if(y1>top) //clipping needed
    {
        x1 += dx*(y1-top); //calculate
        dy = top-y2;
        y1 = top;
    }
    add = (EDGE*)malloc(sizeof(EDGE));
    add->x=x1+dx/2;
    add->dx=dx;
    add->dy=dy;
    add->next=buckets[top-gif(y1-0.5)]; //add to the bucket using half interval scan lines
    buckets[top-gif(y1-0.5)]=add;
}

void clearPixel(int x, int y) //clears the specified pixel
{
    while(((raster.cur=clock())-raster.pre)<(raster.dely*CLOCKS_PER_SEC));
                                    //wait for the required delay
    int i, j;
    int bLeftX=raster.centerX+x*raster.width;
    int bLeftY=raster.centerY-y*raster.width;
    for(i=1; i<raster.width; ++i)
        for(j=0; j<raster.width; ++j)
            if(getpixel(bLeftX+i, bLeftY-j)==BLUE)
                putpixel(bLeftX+i, bLeftY-j, WHITE); //clear pixel by pixel if filled
    raster.pre=raster.cur=clock();
}


void edgeFill()
{
    EDGE *cur;
    int i, j;
    for(i=0; i<numSL; ++i)
    {
        for(j=0; j<numCol; ++j) SL[j]=0; //clear the Scan Line
        for(cur=buckets[i]; cur; cur=cur->next)
            if(cur->dy!=0) //add to active edge list
            {
                EDGE *temp = (EDGE*)malloc(sizeof(EDGE));
                temp->x=cur->x;
                temp->dx=cur->dx;
                temp->dy=cur->dy;
                temp->next=ael;
                ael=temp;
            }
        for(cur=ael; cur; cur=cur->next)
            if(cur->dy > 0)
            {
                for(j=0; j<numCol; ++j)
                {
                    if(j-left+0.5>cur->x) SL[j]=!SL[j];
                                          //invert pixels to right of intersection
                    if(SL[j]) fillPixel(j-left, top-i);
                                    //display (for animation purposes only)
                    else clearPixel(j-left, top-i);
                }
                --cur->dy; //update
                cur->x+=cur->dx;
            }
        /*for(j=0; j<numCol; ++j)
            if(SL[j]) fillPixel(j-left, top-i);
                      //normally display would only be done here
                                 however for animation purposes, display is handled abo
    }
}

int main(int argc, char **argv)
{
    SHAPE T, cur;
    int i;
    int gd=DETECT,gm;
    FILE *kpg;
    if(argc<2)
    {
        printf("[ ERROR ] Usage : <.kpg>_format_input_file\n");
        return 1;
    }
    kpg = fopen(argv[1], "rx");
    if(!kpg)
    {
        printf("[ ERROR ] Usage : <.kpg>_format_input_file\n");
        return 1;
    }
    initgraph(&gd, &gm, NULL); //initialize graphics
    grid();
    numSL = (top=raster.centerY/raster.width)
          + (480-raster.centerY)/raster.width; //number of Scan Lines
    SL = (int*)malloc(sizeof(int)*(numCol=(left=raster.centerX/raster.width)
       +(640-raster.centerX)/raster.width)); //Number of Pixels per Line
    --top; --left; --numCol;
    buckets = (EDGE**)malloc(sizeof(EDGE*)*numSL); //initialize buckets
    for(i=0; i<numSL; ++i) buckets[i]=NULL;
    if(!(T=expectPoly(kpg)))
    {
        printf("\nInvalid File Format\n");
        fclose(kpg);
        return 1;
    }
    for(cur=T; cur->next; cur=cur->next)
        fillBucket(cur); //add lines to respective buckets
    cur->next=T;
    fillBucket(cur);
    cur->next=NULL;
    erase(T);
    fclose(kpg);
    edgeFill(); //call algorithm
    getchar();
    closegraph();
    for(i=0; i<numSL; ++i) //free memory
    {
        EDGE *E=buckets[i];
        while(E)
        {
            buckets[i]=E;
            E=E->next;
            free(buckets[i]);
        }
    }
    free(buckets);
    free(ael);
    free(SL);
    return 0;
}
```

Output



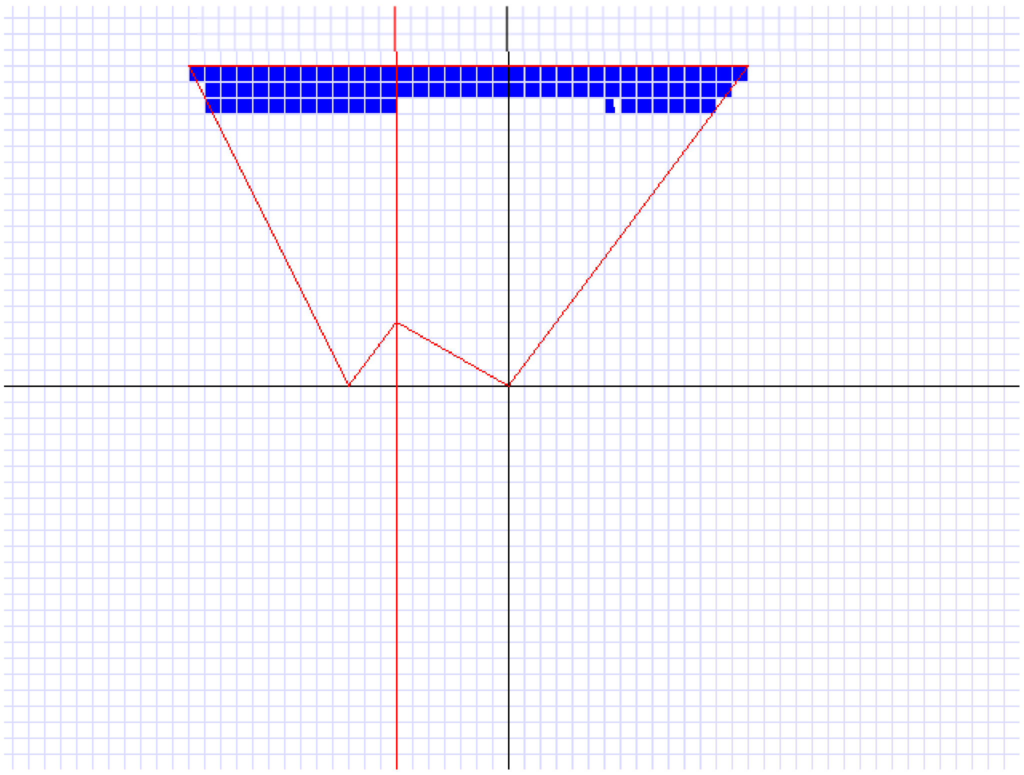
Fence Fill Polygon Filling

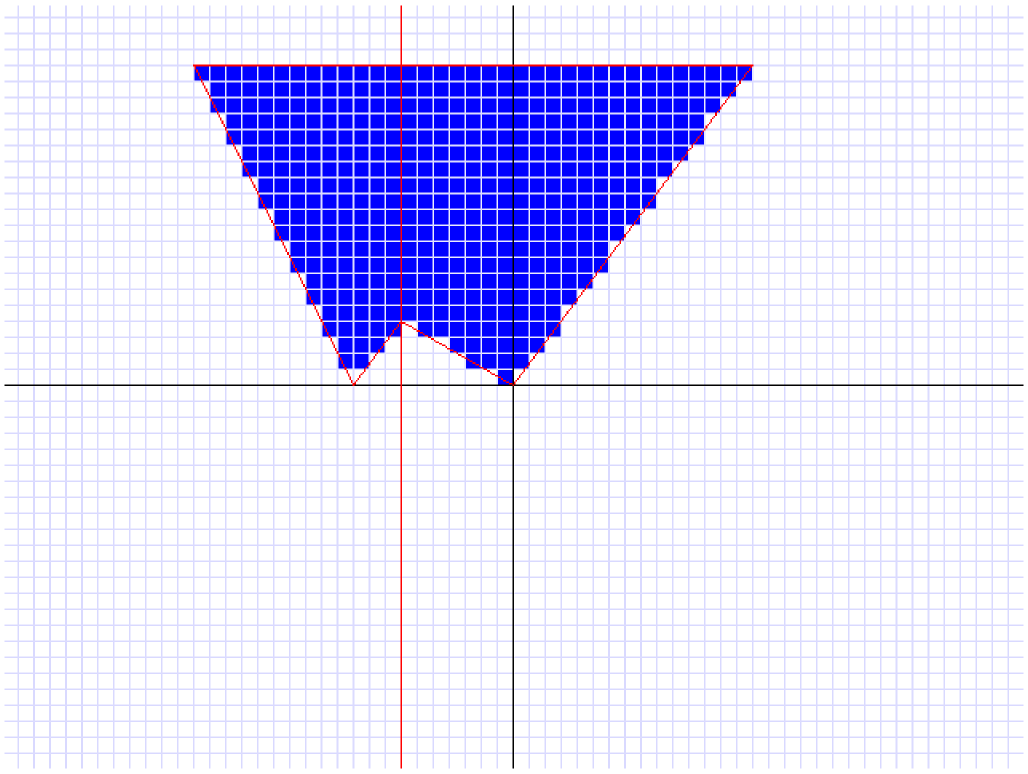
An improvement on the Edge Fill algorithm, Fence Fill passes an imaginary line through the middle of the polygon (usually through a vertex). Now, each half is Edge Filled separately. However, the filling is always done towards the Fence. By passing the Fence through a vertex of the polygon, we reduce the number of intersections between the scanlines and the polygon. Also, on average only half the pixels are toggled per edge intersection. Choosing the Fence location is an important consideration here.

```c
/*                            10.FENCEFILL.C                                    */
#include <stdio.h>
#include <graphics.h>
#include <stdlib.h>
#include "0.kpgparse.c"
#include "0.raster.c"

int gif(double d) //greatest integer function
{
    if(d<0 && (int)d>d)
        return (int)d-1;
    return (int)d;
}

int numCol, numSL, top, left;
int *SL=NULL; //single scan line

typedef struct edge
{
    double x, dx;
    int dy;
    struct edge *next;
}EDGE;

EDGE **buckets=NULL, *ael=NULL;

void fillBucket(SHAPE shape)
{
    int dy;
    double dx;
    EDGE *add;
    int x1=shape->x;
    int y1=shape->y;
    int x2=shape->next->x;
    int y2=shape->next->y;
    refLine(x1, y1, x2, y2); //polygon boundaries
    if(y2==y1) return; //horizontal lines need not be added
    if(y2>y1) { int t=x1; x1=x2; x2=t;
                t=y1; y1=y2; y2=t; } //P1 is the upper point
    dy = y1-y2;
    dx=(x2-x1)*1.0/dy;
    if(y1<top-numSL || y2>top) return; //totally above or below
    if(y1>top) //clipping needed
    {
        x1 += dx*(y1-top); //calculate
        dy = top-y2;
        y1 = top;
    }
    add = (EDGE*)malloc(sizeof(EDGE));
    add->x=x1+dx/2;
    add->dx=dx;
    add->dy=dy;
    add->next=buckets[top-gif(y1-0.5)]; //add to the bucket using half interval scan lines
    buckets[top-gif(y1-0.5)]=add;
}

void clearPixel(int x, int y) //clears the specified pixel
{
    while(((raster.cur=clock())-raster.pre)<(raster.dely*CLOCKS_PER_SEC));
                                            //wait for the required delay
    int i, j;
    int bLeftX=raster.centerX+x*raster.width;
    int bLeftY=raster.centerY-y*raster.width;
    for(i=1; i<raster.width; ++i)
        for(j=0; j<raster.width; ++j)
            if(getpixel(bLeftX+i, bLeftY-j)==BLUE)
                putpixel(bLeftX+i, bLeftY-j, WHITE); //clear pixel by pixel if filled
    raster.pre=raster.cur=clock();
}


void fenceFill(int fence)
{
    EDGE *cur;
    int i, j;
    refLine(fence, 24, fence, -24); //show fence
    fence+=left;
    for(i=0; i<numSL; ++i)
    {
        for(j=0; j<numCol; ++j) SL[j]=0; //clear the Scan Line
        for(cur=buckets[i]; cur; cur=cur->next)
            if(cur->dy!=0) //add to active edge list
            {
                EDGE *temp = (EDGE*)malloc(sizeof(EDGE));
                temp->x=cur->x;
                temp->dx=cur->dx;
                temp->dy=cur->dy;
                temp->next=ael;
                ael=temp;
            }
        for(cur=ael; cur; cur=cur->next)
            if(cur->dy > 0)
            {
                for(j=0; j<fence; ++j) //left of fence
                {
                    if(j-left+0.5>cur->x) SL[j]=!SL[j];
                                            //invert pixels to right of intersection
                    if(SL[j]) fillPixel(j-left, top-i);
                                //display (for animation purposes only)
                    else clearPixel(j-left, top-i);
                }
                for(j=numCol; j>fence; --j) //right of fence
                {
                    if(j-left-0.5<cur->x) SL[j-1]=!SL[j-1];
                                            //invert pixels to left of intersection
                    if(SL[j-1]) fillPixel(j-left-1, top-i);
                                  //display (for animation purposes only)
                    else clearPixel(j-left-1, top-i);
                }
                --cur->dy; //update
                cur->x+=cur->dx;
            }
        /*for(j=0; j<numCol; ++j)
            if(SL[j]) fillPixel(j-left, top-i);
                        //normally display would only be done here
                                    however for animation purposes, display is handled abo
    }
}
```

```c
int main(int argc, char **argv)
{
    SHAPE T, cur;
    int i, fence;
    int gd=DETECT,gm;
    FILE *kpg;
    if(argc<2)
    {
        printf("[ ERROR ] Usage : <.kpg>_format_input_file\n");
        return 1;
    }
    kpg = fopen(argv[1], "rx");
    if(!kpg)
    {
        printf("[ ERROR ] Usage : <.kpg>_format_input_file\n");
        return 1;
    }
    initgraph(&gd, &gm, NULL); //initialize graphics
    grid();
    numSL = (top=raster.centerY/raster.width)
          + (480-raster.centerY)/raster.width; //number of Scan Lines
    SL = (int*)malloc(sizeof(int)*(numCol=(left=raster.centerX/raster.width)
       +(640-raster.centerX)/raster.width)); //Number of Pixels per Line
    --top; --left; --numCol;
    buckets = (EDGE**)malloc(sizeof(EDGE*)*numSL); //initialize buckets
    for(i=0; i<numSL; ++i) buckets[i]=NULL;
    if(!((T=expectPoly(kpg)) && (cur=expectSeed(kpg))))
    {
        printf("\nInvalid File Format\n");
        fclose(kpg);
        return 1;
    }
    fence = cur->x;
    for(cur=T; cur->next; cur=cur->next)
        fillBucket(cur); //add lines to respective buckets
    cur->next=T;
    fillBucket(cur);
    cur->next=NULL;
    erase(T);
    fclose(kpg);
    fenceFill(fence); //call algorithm
    getchar();
    closegraph();
    for(i=0; i<numSL; ++i) //free memory
    {
        EDGE *E=buckets[i];
        while(E)
        {
            buckets[i]=E;
            E=E->next;
            free(buckets[i]);
        }
    }
    free(buckets);
    free(ael);
    free(SL);
    return 0;
}
```

Output





Edge Flag Polygon Filling

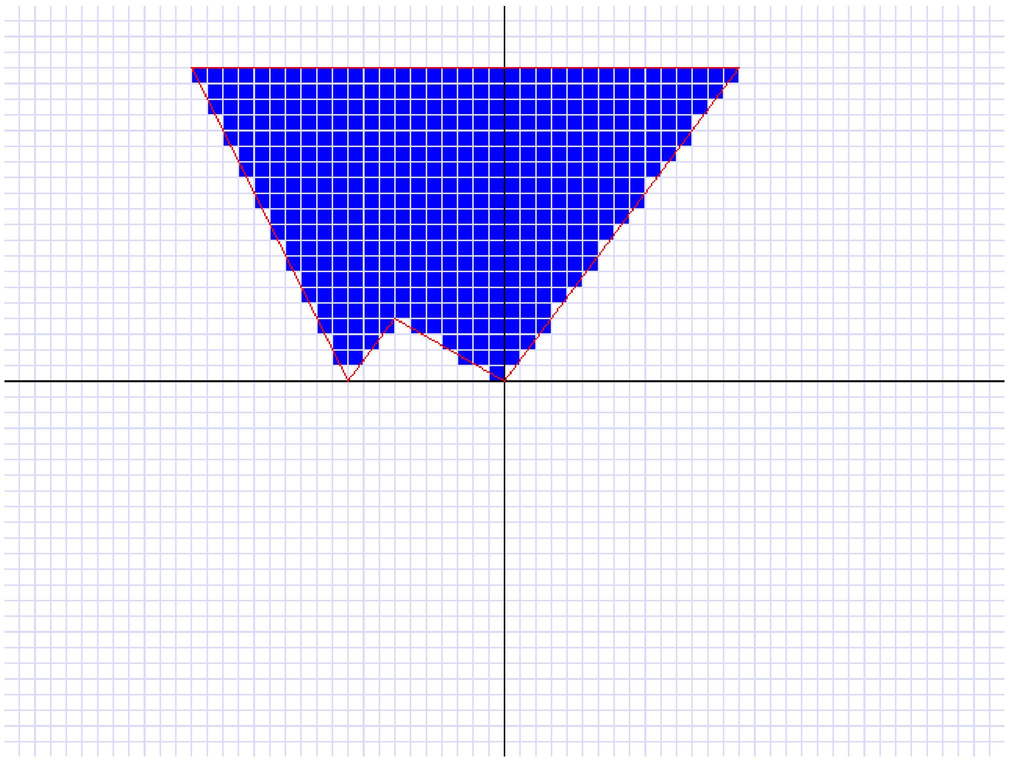
Here, we mark the first pixel whose center lies to the right of each edge intersection with an edge flag. Then we fill portions of the scan line that lies between pairs of edge flags. The flag while entering the polygon is filled; the flag while exiting is not.

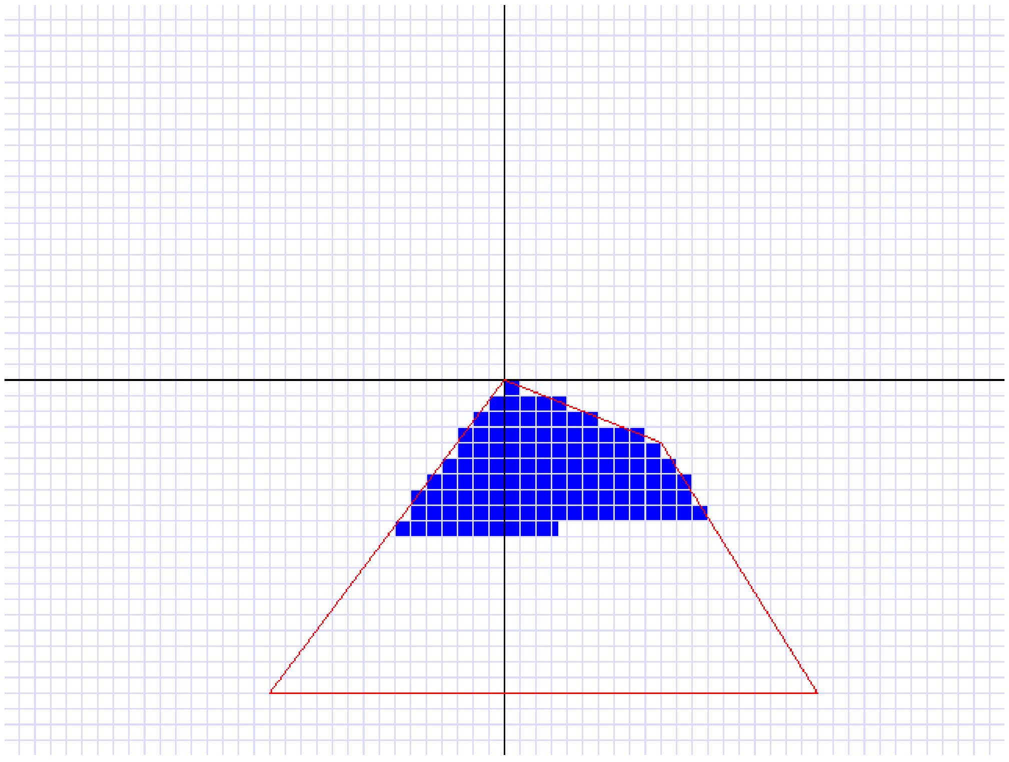
```
/*                           11.EDGEFLAG.C                                  */
#include <stdio.h>
#include <graphics.h>
#include <stdlib.h>
#include "0.kpgparse.c"
#include "0.raster.c"

int gif(double d) //greatest integer function
{
    if(d<0 && (int)d>d)
        return (int)d-1;
    return (int)d;
}

int numCol, numSL, top, left;
int *SL=NULL; //single scan line

typedef struct edge
{
    double x, dx;
    int dy;
    struct edge *next;
}EDGE;

EDGE **buckets=NULL, *ael=NULL;

void fillBucket(SHAPE shape)
{
    int dy;
    double dx;
    EDGE *add;
    int x1=shape->x;
    int y1=shape->y;
    int x2=shape->next->x;
    int y2=shape->next->y;
    refLine(x1, y1, x2, y2); //polygon boundaries
    if(y2==y1) return; //horizontal lines need not be added
    if(y2>y1) { int t=x1; x1=x2; x2=t;
                t=y1; y1=y2; y2=t; } //P1 is the upper point
    dy = y1-y2;
    dx=(x2-x1)*1.0/dy;
    if(y1<top-numSL || y2>top) return; //totally above or below
    if(y1>top) //clipping needed
    {
        x1 += dx*(y1-top); //calculate
        dy = top-y2;
        y1 = top;
    }
    add = (EDGE*)malloc(sizeof(EDGE));
    add->x=x1+dx/2;
    add->dx=dx;
    add->dy=dy;
    add->next=buckets[top-gif(y1-0.5)]; //add to the bucket using half interval scan lines
    buckets[top-gif(y1-0.5)]=add;
}
```

```c
void edgeFlag()
{
    EDGE *cur;
    int i, j, flag;
    for(i=0; i<numSL; ++i)
    {
        for(j=0; j<numCol; ++j) SL[j]=0; //clear the Scan Line
        for(cur=buckets[i]; cur; cur=cur->next)
            if(cur->dy!=0) //add to active edge list
            {
                EDGE *temp = (EDGE*)malloc(sizeof(EDGE));
                temp->x=cur->x;
                temp->dx=cur->dx;
                temp->dy=cur->dy;
                temp->next=ael;
                ael=temp;
            }
        for(cur=ael; cur; cur=cur->next)
            if(cur->dy > 0)
            {
                flag = gif(cur->x+0.5)+left;
                SL[flag]=!SL[flag]; //edge flag
                --cur->dy; //update
                cur->x+=cur->dx;
            }
        flag=0;
        for(j=0; j<numCol; ++j)
        {
            if(SL[j]) flag=!flag;
            if(flag) fillPixel(j-left, top-i); //display Scan Line
        }
    }
}
```

```c
int main(int argc, char **argv)
{
    SHAPE T, cur;
    int i;
    int gd=DETECT,gm;
    FILE *kpg;
    if(argc<2)
    {
        printf("[ ERROR ] Usage : <.kpg>_format_input_file\n");
        return 1;
    }
    kpg = fopen(argv[1], "rx");
    if(!kpg)
    {
        printf("[ ERROR ] Usage : <.kpg>_format_input_file\n");
        return 1;
    }
    initgraph(&gd, &gm, NULL); //initialize graphics
    grid();
    numSL = (top=raster.centerY/raster.width)
          + (480-raster.centerY)/raster.width; //number of Scan Lines
    SL = (int*)malloc(sizeof(int)*(numCol=(left=raster.centerX/raster.width)
       +(640-raster.centerX)/raster.width)); //Number of Pixels per Line
    --top; --left; --numCol;
    buckets = (EDGE**)malloc(sizeof(EDGE*)*numSL); //initialize buckets
    for(i=0; i<numSL; ++i) buckets[i]=NULL;
    if(!(T=expectPoly(kpg)))
    {
        printf("\nInvalid File Format\n");
        fclose(kpg);
        return 1;
    }
    for(cur=T; cur->next; cur=cur->next)
        fillBucket(cur); //add lines to respective buckets
    cur->next=T;
    fillBucket(cur);
    cur->next=NULL;
    erase(T);
    fclose(kpg);
    edgeFlag(); //call algorithm
    getchar();
    closegraph();
    for(i=0; i<numSL; ++i) //free memory
    {
        EDGE *E=buckets[i];
        while(E)
        {
            buckets[i]=E;
            E=E->next;
            free(buckets[i]);
        }
    }
    free(buckets);
    free(ael);
    free(SL);
    return 0;
}
```

Output





Seed Fill Polygon Filling

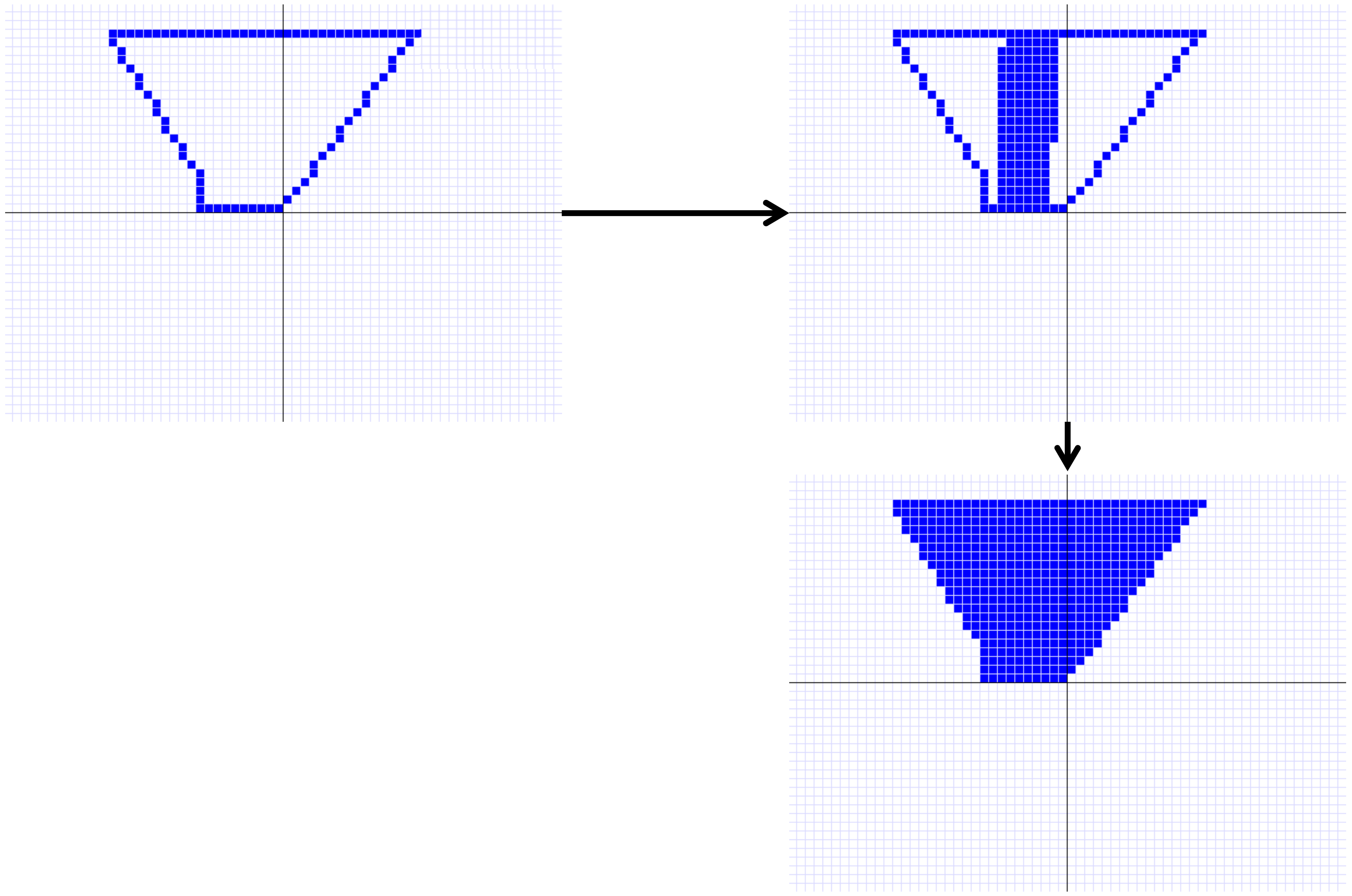
This is a very simple and highly inefficient algorithm for polygon filling. It requires a full frame buffer in memory and the program stack can grow uncontrollably. Once a seed point is provided inside the polygon, the seed fill algorithm is recursively called on all its neighbours as long as the boundary color is not encountered. Due to this, the same point is often pushed onto the stack several times.

There are two variants, 4-connected, where the recursion is carried out only in the horizontal and vertical directions, and 8-connected, where the diagonals are considered as well. Here, we have used the 4-connected system.

```c
/*                              12.STACKSEED.C                              */
#include <stdio.h>
#include <graphics.h>
#include <stdlib.h>
#include "0.kpgparse.c"
#include "0.raster.c"

int gif(double d) //greatest integer function
{
    if(d<0 && (int)d>d)
        return (int)d-1;
    return (int)d;
}

int numCol, numSL, top, left;
int **SL=NULL; //single scan line

typedef struct edge
{
    double x, dx;
    int dy;
    struct edge *next;
}EDGE;

EDGE **buckets=NULL, *ael=NULL;

void fillBucket(SHAPE shape)
{
    int dy;
    double dx;
    EDGE *add;
    int x1=shape->x;
    int y1=shape->y;
    int x2=shape->next->x;
    int y2=shape->next->y;
    if(y2>y1) { int t=x1; x1=x2; x2=t;
                t=y1; y1=y2; y2=t; } //P1 is the upper point
    dy = y1-y2;
    if(dy) dx=(x2-x1)*1.0/dy;
    else //horizontal line
    {
        if(x1>x2) { int t=x1; x1=x2; x2=t; } //P1 is the leftmost point
        dx=x2-x1;
    }
    if(y1<top-numSL || y2>top) return; //totally above or below
    if(y1>top) //clipping needed
    {
        x1 += dx*(y1-top); //calculate
        dy = top-y2;
        y1 = top;
    }
    add = (EDGE*)malloc(sizeof(EDGE));
    add->x=x1;
    add->dx=dx;
    add->dy=dy;
    add->next=buckets[top-gif(y1)]; //add to the bucket
    buckets[top-gif(y1)]=add;
}
```

```c
void scanConvert()
{
    EDGE *cur;
    int i, j, var;
    for(i=0; i<numSL; ++i)
    {
        for(j=0; j<numCol; ++j) SL[i][j]=0; //clear the Scan Line
        for(cur=buckets[i]; cur; cur=cur->next)
            if(cur->dy==0) //horizontal line
            {
                for(j=0; j<cur->dx; ++j)
                    SL[i][gif(cur->x+j)+left]=1;
            }
            else //add to Active Edge List
            {
                EDGE *temp = (EDGE*)malloc(sizeof(EDGE));
                temp->x=cur->x;
                temp->dx=cur->dx;
                temp->dy=cur->dy;
                temp->next=ael;
                ael=temp;
            }
        for(cur=ael; cur; cur=cur->next)
            if(cur->dy > 0)
            {
                if(cur->dx > 1) //not steep slope, multiple points
                    for(var=gif(cur->x+cur->dx); var > gif(cur->x); --var)
                        SL[i][var+left]=1;
                else if(cur->dx < -1) //not steep slope, multiple points
                    for(var=gif(cur->x+cur->dx); var < gif(cur->x); ++var)
                        SL[i][var+left]=1;
                else SL[i][gif(cur->x)+left]=1; //steep slope, single point
                --cur->dy; //update
                cur->x+=cur->dx;
            }
        for(j=0; j<numCol; ++j)
            if(SL[i][j]) fillPixel(j-left, top-i); //clear the Scan Line
    }
}

void seedFill(int x, int y)
{
    if(x<0 || y<0 || x>numCol || y>numSL) return; //out of bounds
    if(SL[y][x]>0) return; //already processed
    SL[y][x] = 1;
    fillPixel(x-left, top-y);
    seedFill(x, y-1); //four connected
    seedFill(x, y+1);
    seedFill(x-1, y);
    seedFill(x+1, y);
}

int main(int argc, char **argv)
{
    SHAPE T, cur;
    int i;
    int gd=DETECT,gm;
    FILE *kpg;
    if(argc<2)
    {
        printf("[ ERROR ] Usage : <.kpg>_format_input_file\n");
        return 1;
    }
    kpg = fopen(argv[1], "rx");
    if(!kpg)
    {
        printf("[ ERROR ] Usage : <.kpg>_format_input_file\n");
        return 1;
    }
    initgraph(&gd, &gm, NULL); //initialize graphics
    grid();
    numSL = (top=raster.centerY/raster.width)
          + (480-raster.centerY)/raster.width; //number of Scan Lines
    numCol = (left=raster.centerX/raster.width)
           +(640-raster.centerX)/raster.width; //number of columns
    SL = (int**)malloc(sizeof(int*)*numSL);
    SL[0] = (int*)malloc(sizeof(int)*numCol*numSL); //Number of Pixels per Line
    --top; --left;
    buckets = (EDGE**)malloc(sizeof(EDGE*)*numSL); //initialize buckets
    for(i=0; i<numSL; ++i)
    {
        buckets[i]=NULL;
        SL[i] = SL[0] + numCol*i; //array manipulation
    }
    if(!(T=expectPoly(kpg))) //input boundary
    {
        printf("\nInvalid File Format\n");
        fclose(kpg);
        return 1;
    }
    for(cur=T; cur->next; cur=cur->next)
        fillBucket(cur); //add lines to respective buckets
    cur->next=T;
    fillBucket(cur);
    cur->next=NULL;
    erase(T);
    scanConvert(); //draw boundary
    getchar();
    if(!(T=expectSeed(kpg)))
    {
        printf("\nInvalid File Format\n");
        fclose(kpg);
        return 1;
    }
    seedFill(T->x+left, top-T->y); //call algorithm
    erase(T);
    fclose(kpg);
    getchar();
    closegraph();
    for(i=0; i<numSL; ++i) //free memory
    {
        EDGE *E=buckets[i];
        while(E)
        {
            buckets[i]=E;
            E=E->next;
            free(buckets[i]);
        }
    }
    free(buckets);
    free(ael);
    free(SL[0]);
    free(SL);
    return 0;
}
```

Output



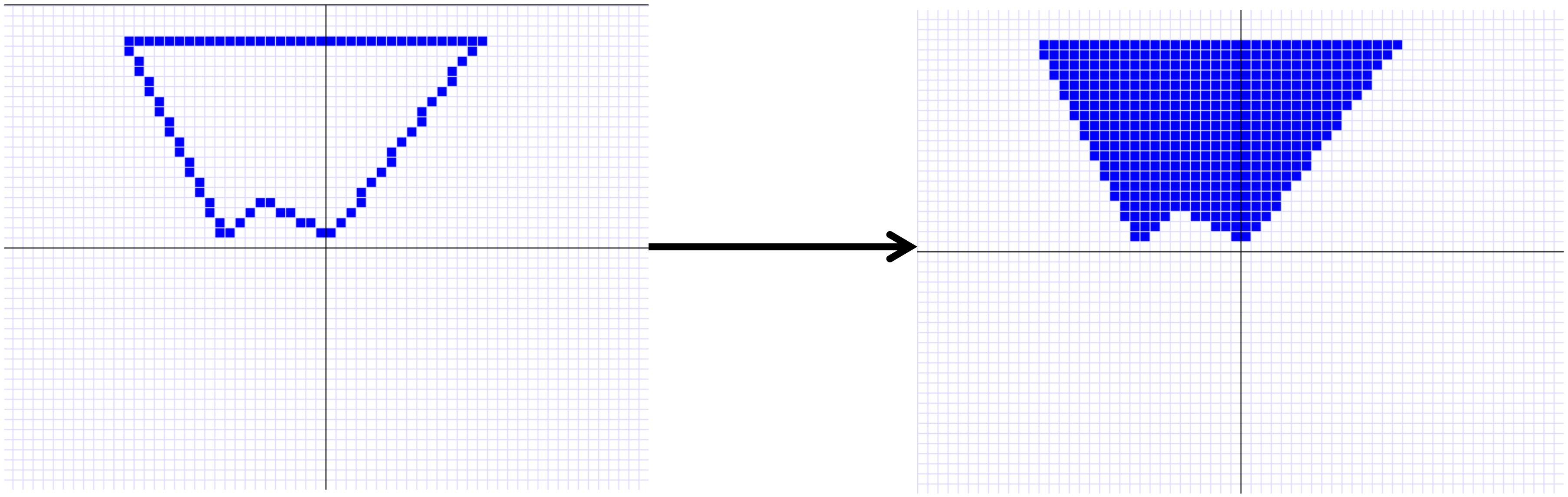
Scan Line Seed Fill

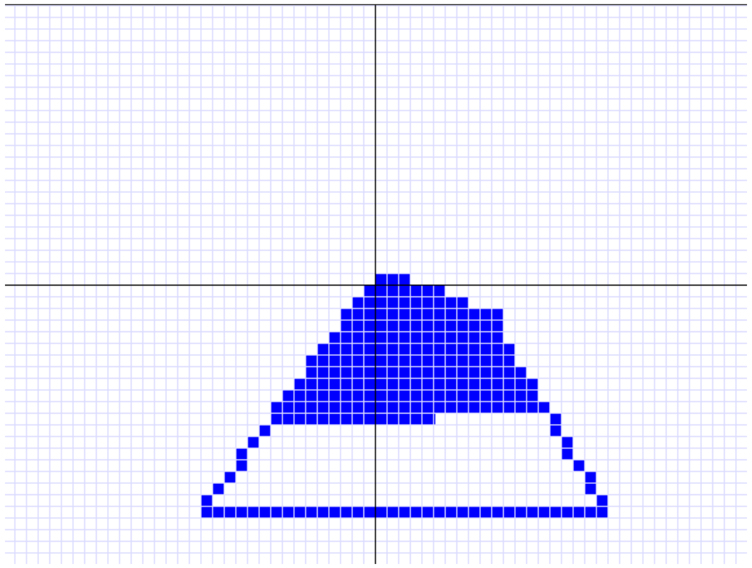
An improvement on the Seed Fill Algorithm, this method fills continuous stretches of the interior region of the polygon on the same scan line. It then recursively searches the scan line directly above and below it for un-filled interior pixels. Care must be taken while searching above and below boundary points in case of 4-connected regions.

```
/*                       13.SEEDSCANLINE.C                                  */
#include <stdio.h>
#include <graphics.h>
#include <stdlib.h>
#include "0.kpgparse.c"
#include "0.raster.c"

int gif(double d) //greatest integer function
{
    if(d<0 && (int)d>d)
        return (int)d-1;
    return (int)d;
}

int numCol, numSL, top, left;
int **SL=NULL; //single scan line

typedef struct edge
{
    double x, dx;
    int dy;
    struct edge *next;
}EDGE;

EDGE **buckets=NULL, *ael=NULL;

void fillBucket(SHAPE shape)
{
    int dy;
    double dx;
    EDGE *add;
    int x1=shape->x;
    int y1=shape->y;
    int x2=shape->next->x;
    int y2=shape->next->y;
    if(y2>y1) { int t=x1; x1=x2; x2=t;
                t=y1; y1=y2; y2=t; } //P1 is the upper point
    dy = y1-y2;
    if(dy) dx=(x2-x1)*1.0/dy;
    else //horizontal line
    {
        if(x1>x2) { int t=x1; x1=x2; x2=t; } //P1 is the leftmost point
        dx=x2-x1;
    }
    if(y1<top-numSL || y2>top) return; //totally above or below
    if(y1>top) //clipping needed
    {
        x1 += dx*(y1-top); //calculate
        dy = top-y2;
        y1 = top;
    }
    add = (EDGE*)malloc(sizeof(EDGE));
    add->x=x1;
    add->dx=dx;
    add->dy=dy;
    add->next=buckets[top-gif(y1)]; //add to the bucket
    buckets[top-gif(y1)]=add;
}
```

```c
void scanConvert()
{
    EDGE *cur;
    int i, j, var;
    for(i=0; i<numSL; ++i)
    {
        for(j=0; j<numCol; ++j) SL[i][j]=0; //clear the Scan Line
        for(cur=buckets[i]; cur; cur=cur->next)
            if(cur->dy==0) //horizontal line
            {
                for(j=0; j<cur->dx; ++j)
                    SL[i][gif(cur->x+j)+left]=1;
            }
            else //add to Active Edge List
            {
                EDGE *temp = (EDGE*)malloc(sizeof(EDGE));
                temp->x=cur->x;
                temp->dx=cur->dx;
                temp->dy=cur->dy;
                temp->next=ael;
                ael=temp;
            }
        for(cur=ael; cur; cur=cur->next)
            if(cur->dy > 0)
            {
                if(cur->dx > 1) //not steep slope, multiple points
                    for(var=gif(cur->x+cur->dx); var > gif(cur->x); --var)
                        SL[i][var+left]=1;
                else if(cur->dx < -1) //not steep slope, multiple points
                    for(var=gif(cur->x+cur->dx); var < gif(cur->x); ++var)
                        SL[i][var+left]=1;
                else SL[i][gif(cur->x)+left]=1; //steep slope, single point
                --cur->dy; //update
                cur->x+=cur->dx;
            }
        for(j=0; j<numCol; ++j)
            if(SL[i][j]) fillPixel(j-left, top-i); //clear the Scan Line
    }
}

void seedSL(int x, int y)
{
    int l, r, i, pair;
    if(x<0 || y<0 || x>numCol || y>numSL) return; //out of bounds
    for(l=x; l>0 && !SL[y][l]; --l); //get left most point
    for(r=x; l<numCol && !SL[y][r]; ++r); //get right most point
    for(i=l+1; i<r; ++i)
    {
        SL[y][i] = 1;
        fillPixel(i-left, top-y); //fill
    }
    pair=1;
    if(y) //check upper line
    {
        for(i=l+1; i<r; ++i)
        {
            if(!SL[y-1][i] && pair) //if internal empty zone found
                seedSL(i, y-1);
            else if(i && SL[y-1][i]!=SL[y-1][i-1])
                pair = !pair;
        }
    }
    pair=1;
    if(y<numSL-1) //check lower line
    {
        for(i=l+1; i<r; ++i)
        {
            if(!SL[y+1][i] && pair) //if internal empty zone found
                seedSL(i, y+1);
            else if(i && SL[y+1][i]!=SL[y+1][i-1])
                pair = !pair;
        }
    }
}

int main(int argc, char **argv)
{
    SHAPE T, cur;
    int i;
    int gd=DETECT,gm;
    FILE *kpg;
    if(argc<2)
    {
        printf("[ ERROR ] Usage : <.kpg>_format_input_file\n");
        return 1;
    }
    kpg = fopen(argv[1], "rx");
    if(!kpg)
    {
        printf("[ ERROR ] Usage : <.kpg>_format_input_file\n");
        return 1;
    }
    initgraph(&gd, &gm, NULL); //initialize graphics
    grid();
    numSL = (top=raster.centerY/raster.width)
          + (480-raster.centerY)/raster.width; //number of Scan Lines
    numCol = (left=raster.centerX/raster.width)
           +(640-raster.centerX)/raster.width; //number of columns
    SL = (int**)malloc(sizeof(int*)*numSL);
    SL[0] = (int*)malloc(sizeof(int)*numCol*numSL); //Number of Pixels per Line
    --top; --left;
    buckets = (EDGE**)malloc(sizeof(EDGE*)*numSL); //initialize buckets
    for(i=0; i<numSL; ++i)
    {
        buckets[i]=NULL;
        SL[i] = SL[0] + numCol*i; //array manipulation
    }
    if(!(T=expectPoly(kpg))) //input boundary
    {
        printf("\nInvalid File Format\n");
        fclose(kpg);
        return 1;
    }
    for(cur=T; cur->next; cur=cur->next)
        fillBucket(cur); //add lines to respective buckets
    cur->next=T;
    fillBucket(cur);
    cur->next=NULL;
    erase(T);
    scanConvert(); //draw boundary
    getchar();
    if(!(T=expectSeed(kpg)))
    {
        printf("\nInvalid File Format\n");
        fclose(kpg);
        return 1;
    }
    seedSL(T->x+left, top-T->y); //call algorithm
    erase(T);
    fclose(kpg);
    getchar();
    closegraph();
    for(i=0; i<numSL; ++i) //free memory
    {
        EDGE *E=buckets[i];
        while(E)
        {
            buckets[i]=E;
            E=E->next;
            free(buckets[i]);
        }
    }
    free(buckets);
    free(ael);
    free(SL[0]);
    free(SL);
    return 0;
}
```

Output





```
/*                        0.CLIPPING.C                                        */
#ifndef clipping_graphics
#define clipping_graphics
#include <graphics.h>
#include "0.kpgparse.c"

void clippingWindow(SHAPE window) //draws the clipping window polygon
{
    setcolor(WHITE);
    bar(0, 0, 640, 480);
    setcolor(LIGHTGRAY);
    line(320, 0, 320, 480);
    line(0, 240, 640, 240);
    setcolor(BLACK);
    SHAPE vertex1=window;
    do
    {
        line(320+(window->x), 240-(window->y),
             320+(window->next->x), 240-(window->next->y));
        window=window->next;
    }while(window->next);
    line(320+(window->x), 240-(window->y),
         320+(vertex1->x), 240-(vertex1->y)); //last edge
}

void refLine(x1, y1, x2, y2) //original line
{
    setcolor(RED);
    line(320+x1, 240-y1, 320+x2, 240-y2);
    ellipse(320+x1, 240-y1, 0, 360, 2, 2);
    ellipse(320+x2, 240-y2, 0, 360, 2, 2);
}

void clippedLine(x1, y1, x2, y2) //clipped lines
{
    setcolor(BLUE);
    line(320+x1, 240-y1, 320+x2, 240-y2);
    ellipse(320+x1, 240-y1, 0, 360, 3, 3);
    ellipse(320+x2, 240-y2, 0, 360, 3, 3);
}

#endif
```

Sutherland-Cohen Clipping

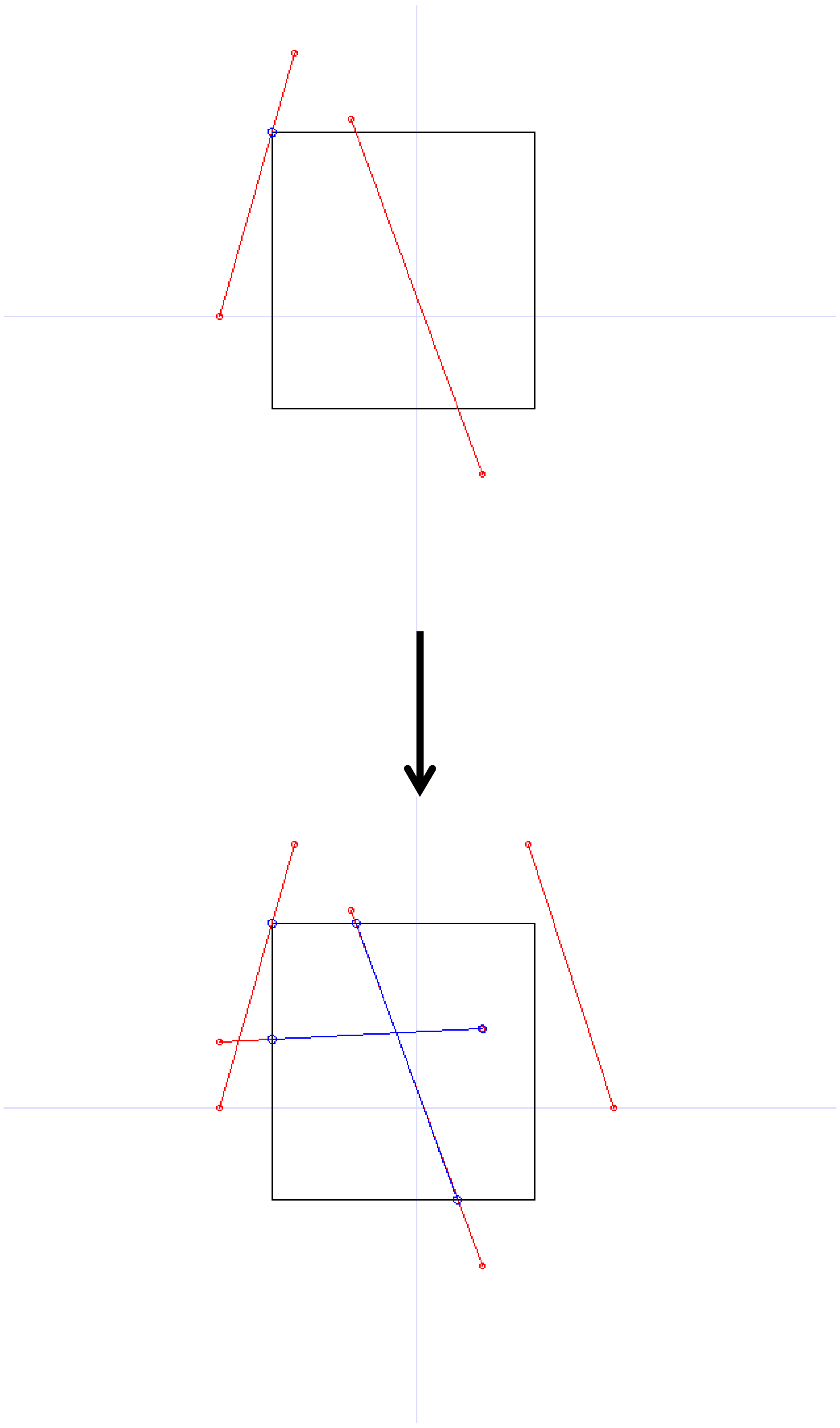
The algorithm divides a two-dimensional space into 9 regions and determines the lines and portions of lines that are visible in the region of interest (the viewport/window). This is accomplished by using End Point Codes, calculated based on whether the end-points are inside or outside the viewport relative to each window edge. The Sutherland-Cohen algorithm is only applicable for regular windows (rectangular windows with sides parallel to the co-ordinate axes). The algorithm includes, excludes or partially includes the lines based on whether:

- Both endpoints are in the viewport – trivial acceptance.
- Both endpoints are external relative to the same line –trivial rejection.
- The endpoints are in different regions: The algorithm finds one point that is outside the viewport (at least one must exist). The intersection of the external point and extended viewport border is then calculated by algebraic means. The intersection replaces the external point. The algorithm repeats until an accept or a reject occurs.

This algorithm preforms well when most of the lines are either entirely inside or entirely outside the window. The drawback is that it uses multiplications and divisions to determine the intersection. For a large number of lines, this is an unacceptable overhead. Also, this method does not generalize easily to arbitrary convex viewports.

```c
/*                        14.COHEN.C                                    */
#include <stdio.h>
#include <graphics.h>
#include "0.kpgparse.c"
#include "0.clipping.c"

int window[4]; //regular window [l, r, b, t]

unsigned char calcEPC(int x, int y) //calculate End Point Code
{
    unsigned char c=0;
    if(y>window[3]) c+=8; //above t
    if(y<window[2]) c+=4; //below b
    if(x>window[1]) c+=2; //right of r
    if(x<window[0]) c+=1; //left of l
    return c;
}

void cohenClip(SHAPE lin)
{
    int i;
    unsigned char code;
    double slope;
    int y1=lin->y;       //get co-ordinates
    int y2=lin->next->y;
    int x1=lin->x;
    int x2=lin->next->x;
    erase(lin);
    refLine(x1, y1, x2, y2); //original line
    getchar();
    code=(calcEPC(x1, y1)<<4) + calcEPC(x2, y2); //calculate end point codes
    if(!code) //trivially visible
    {
        clippedLine(x1, y1, x2, y2); //output
        return;
    }
    if((code>>4)&code) return; //trivially invisible
    slope=(x1==x2)?0:(double)(y2-y1)/(x2-x1); //calculate slope
    while(code) //while partially visible
    {
        for(i=0; i<4; ++i) //each window edge
        {
            if(((code>>(4+i))&1)==((code>>i)&1)) continue; //does not cross the edge
                                                   //(x1,y2) is inside relative to edge
            if(!((code>>(4+i))&1)) { int t=x1; x1=x2; x2=t;
                                      t=y1; y1=y2; y2=t;
                                      code=((code&15)<<4)+(code>>4); } //so swap
            if(x1!=x2 && i<2) //left and right edges
            {
                y1+=slope*(window[i]-x1); //find intersection
                x1=window[i];
            }
            else if(y1!=y2 && i>1) //top and bottom edges
            {
                x1+=slope?(window[i]-y1)/slope:0; //find intersection
                y1=window[i];
            }
            code=(calcEPC(x1, y1)<<4)+(code&15); //update end point codes
            if(!code) break; //totally visible
            if((code>>4)&code) return; //totally invisible
        }
    }
    clippedLine(x1, y1, x2, y2); //output
}

int main(int argc, char **argv)
{
    int gd=DETECT,gm;
    FILE *kpg;
    SHAPE wndw, lin;
    if(argc<2)
    {
        printf("[ ERROR ] Usage : <.kpg>_format_input_file\n");
        return 1;
    }
    kpg = fopen(argv[1], "rx"); //open input file
    if(!kpg)
    {
        printf("[ ERROR ] Usage : <.kpg>_format_input_file\n");
        return 1;
    }
    if(!(wndw=expectPoly(kpg))) //input regular window and line
    {
        printf("\nInvalid Format\n");
        fclose(kpg);
        return 1;
    }
    window[3]=wndw->y;       //get co-ordinates of window
    window[0]=wndw->x;
    window[2]=wndw->next->next->y;
    window[1]=wndw->next->next->x;
    if(window[3]<window[2]) { int t=window[3]; window[3]=window[2]; window[2]=t; }
    if(window[0]>window[1]) { int t=window[0]; window[0]=window[1]; window[1]=t; }
    initgraph(&gd, &gm, NULL); //initialize graphics
    clippingWindow(wndw); //draw window
    erase(wndw);
    while(lin=expectLine(kpg))
        cohenClip(lin); //call algorithm
    getchar();
    closegraph();
    if(!feof(kpg))
    {
        printf("\nInvalid File Format\n");
        fclose(kpg);
        return 1;
    }
    fclose(kpg);
    return 0;
}
```

Output

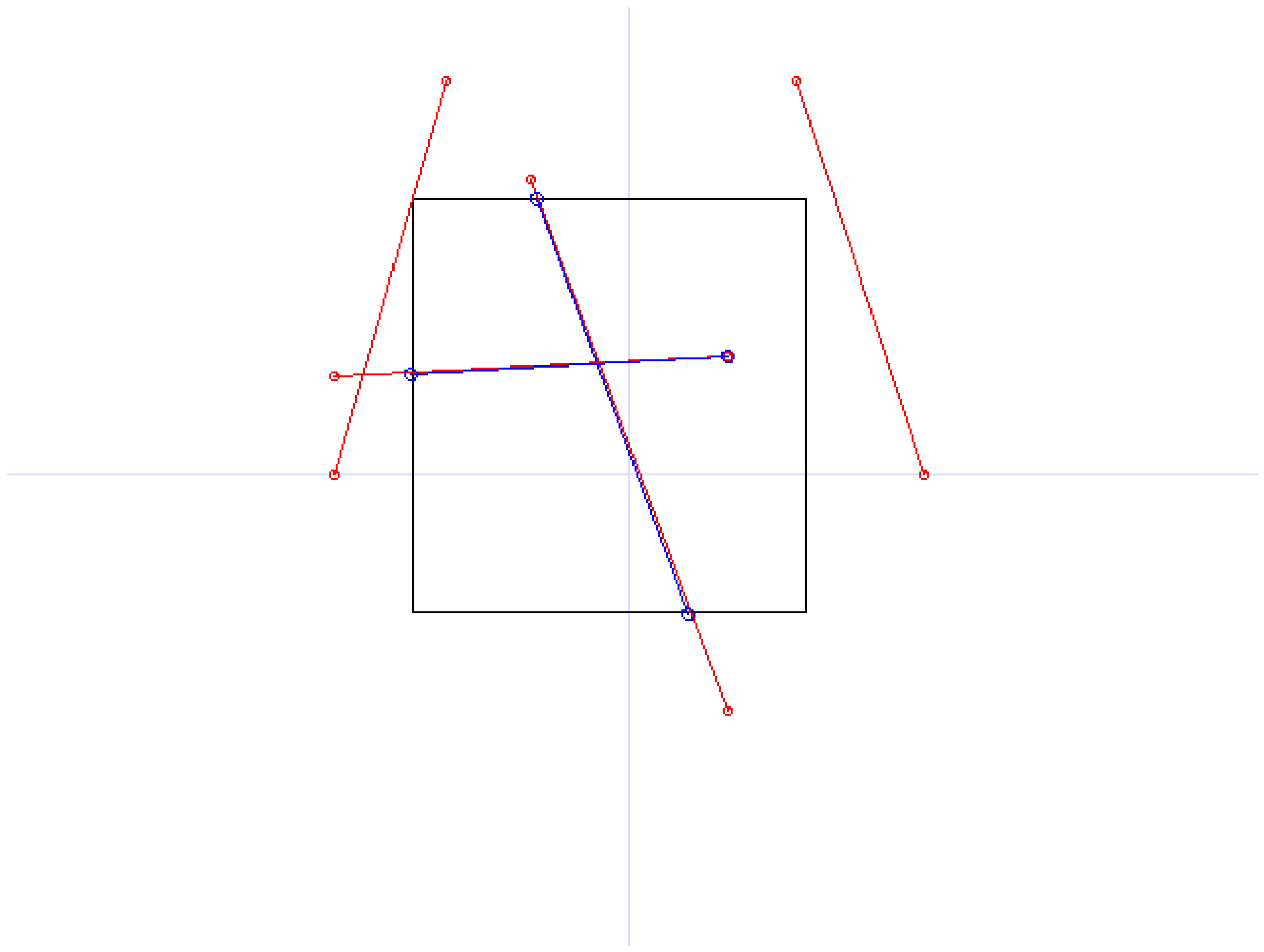


Mid-point Subdivision Clipping

This is an improvement of the Sutherland-Cohen algorithm. Here, the final intersection point calculation is done in a manner similar to a binary search. The lines that are neither trivially accepted nor rejected are broken in to two halves which are then evaluated. However, recursively calling the algorithm this way leads to large overhead in adding lines to lists. Instead, the concept of furthest visible point is used. For a line P1→P2, if P1 is an external point, we find the furthest visible point from P1 and replace P1 by it.

```
/*                            15.MIDPNTCLIP.C                                 */
#include <stdio.h>
#include <stdlib.h>
#include <graphics.h>
#include "0.kpgparse.c"
#include "0.clipping.c"

#define MAXERROR 1 //how to make this 0

int gif(int a1, int a2) //greatest integer function
{
    double d=(a1+a2)*0.5;
    if(d<(int)d)
        return (int)d-1;
    return (int)d;
}

int window[4]; //regular window [l, r, b, t]

unsigned char calcEPC(int x, int y) //calculate End Point Code
{
    unsigned char c=0;
    if(y>window[3]) c+=8; //above t
    if(y<window[2]) c+=4; //below b
    if(x>window[1]) c+=2; //right of r
    if(x<window[0]) c+=1; //left of l
    return c;
}

void getFar(int *x1, int *y1, int x2, int y2) //get furthest visible point from P2
{
    int midx, midy;
    while(abs(*x1-x2)>MAXERROR || abs(y2-*y1)>MAXERROR)
    {
        midx=gif(*x1, x2); //midpoints
        midy=gif(*y1, y2);
        if(calcEPC(*x1, *y1)&calcEPC(midx, midy))
                        { *x1=midx; *y1=midy; } //check visibility
        else { x2=midx; y2=midy; }
    }
    *x1=gif(*x1, x2); //update P1
    *y1=gif(*y1, y2);
}

void midpntClip(SHAPE lin)
{
    int i;
    unsigned char code;
    int y1=lin->y;       //get co-ordinates
    int y2=lin->next->y;
    int x1=lin->x;
    int x2=lin->next->x;
    erase(lin);
    refLine(x1, y1, x2, y2); //original line
    getchar();
    code=(calcEPC(x1, y1)<<4) + calcEPC(x2, y2); //calculate end point codes
    if(!code) //trivially visible
    {
        clippedLine(x1, y1, x2, y2); //output
        return;
    }
    if((code>>4)&code) return; //trivially invisible
    if(code>>4)
        getFar(&x1, &y1, x2, y2); //get furthest visible point from P2
    if(code&15)
        getFar(&x2, &y2, x1, y1); //get furthest visible point from P1
    if(!(calcEPC(x1, y1)&calcEPC(x2, y2)))
        clippedLine(x1, y1, x2, y2); //output
}

int main(int argc, char **argv)
{
    int gd=DETECT,gm;
    FILE *kpg;
    SHAPE wndw, lin;
    if(argc<2)
    {
        printf("[ ERROR ] Usage : <.kpg>_format_input_file\n");
        return 1;
    }
    kpg = fopen(argv[1], "rx"); //open input file
    if(!kpg)
    {
        printf("[ ERROR ] Usage : <.kpg>_format_input_file\n");
        return 1;
    }
    if(!(wndw=expectPoly(kpg))) //input regular window and line
    {
        printf("\nInvalid File Format\n");
        fclose(kpg);
        return 1;
    }
    window[3]=wndw->y;       //get co-ordinates of window
    window[0]=wndw->x;
    window[2]=wndw->next->next->y;
    window[1]=wndw->next->next->x;
    if(window[3]<window[2]) { int t=window[3]; window[3]=window[2]; window[2]=t; }
    if(window[0]>window[1]) { int t=window[0]; window[0]=window[1]; window[1]=t; }
    initgraph(&gd, &gm, NULL); //initialize graphics
    clippingWindow(wndw); //draw window
    erase(wndw);
    while(lin=expectLine(kpg))
        midpntClip(lin); //call algorithm
    getchar();
    closegraph();
    if(!feof(kpg))
    {
        printf("\nInvalid File Format\n");
        fclose(kpg);
        return 1;
    }
    fclose(kpg);
    return 0;
}
```

Output



Cyrus–Beck Algorithm

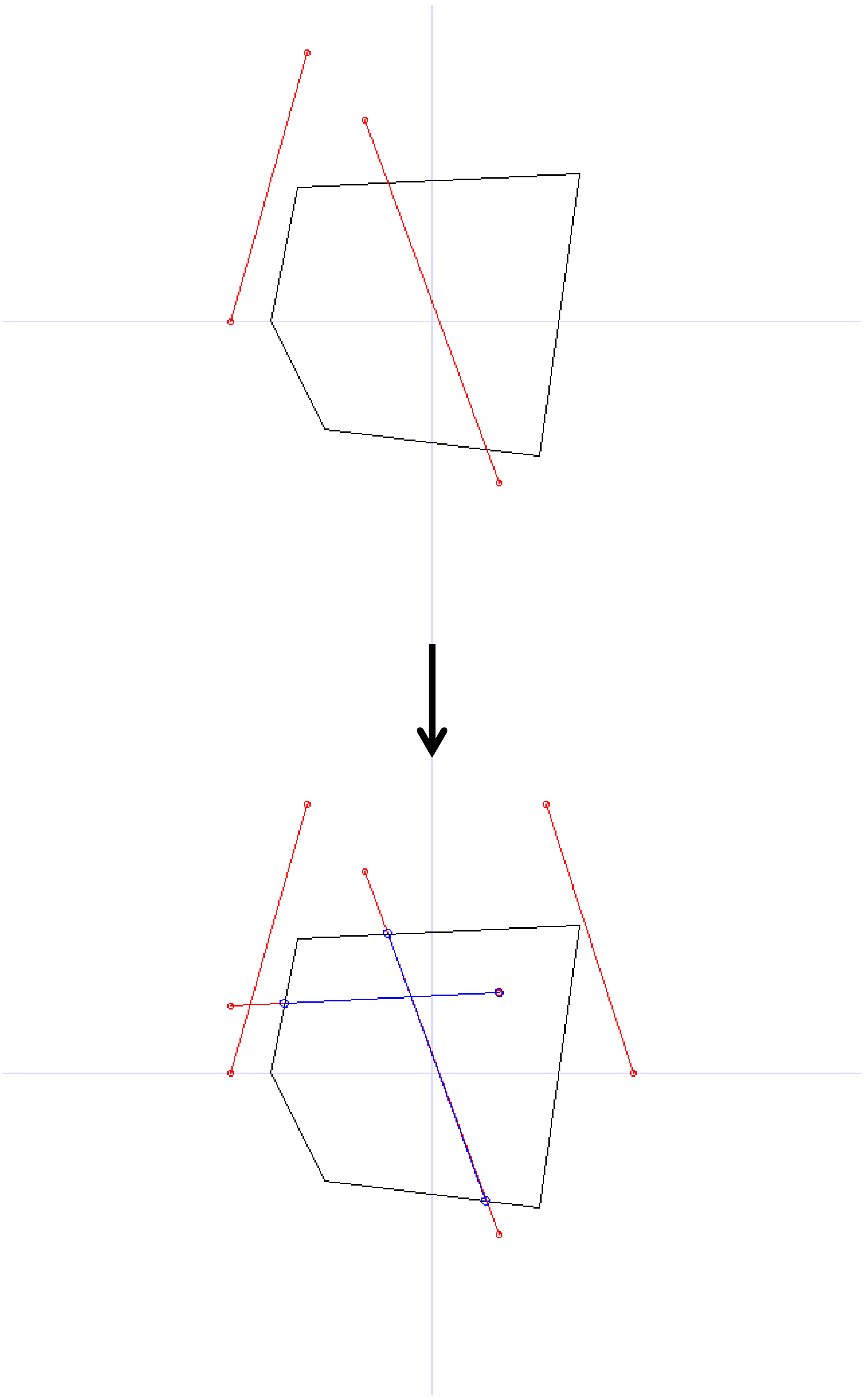
The Cyrus–Beck Algorithm is a generalized Line Clipping algorithm. It was designed to be more efficient than the Sutherland–Cohen algorithm which uses repetitive clipping. Cyrus–Beck can be used with any convex polygon clipping window unlike Sutherland-Cohen that can be used only on a rectangular clipping area.

The Cyrus-Beck method uses the parametric equation of a line to find out the intersection points. We use vector dot products involving the normal to each window edge to determine whether a line intersects an edge. If so, we calculate the parameter *t* of the point of intersection rather than the exact co-ordinates. This parameter may be recalculated several times, so the overhead reduction here is significant. Further, since parametric equations are valid for higher dimensions, this algorithm readily generalizes to 3D.

The exact co-ordinates of the clipped line are calculated once at the end.

```c
/*                              16.CYRUSBECK.C                              */
#include <stdio.h>
#include <stdlib.h>
#include <graphics.h>
#include "0.kpgparse.c"
#include "0.clipping.c"

SHAPE window;

void beck(SHAPE lin)
{
    double t, tu=1.0, tl=0.0;
    int y1=lin->y;      //get co-ordinates
    int y2=lin->next->y;
    int x1=lin->x;
    int x2=lin->next->x;
    SHAPE cur = window; //vertex_1
    erase(lin);
    refLine(x1, y1, x2, y2); //original line
    getchar();
    do
    {
        SHAPE nxt = cur->next?cur->next:window; //closed polygon, vertex_1 follows vertex_N
        int Dn = (nxt->x - cur->x)*(y2-y1) - (nxt->y - cur->y)*(x2-x1);
                         //Dot product of Direction of Line and Inward Normal
        int Wn = (nxt->x - cur->x)*(y1 - cur->y) - (nxt->y - cur->y)*(x1 - cur->x);
                         //Dot product of Weight Vector and Inward Normal
        if(!Dn) //If parallel
            if(Wn<0) return; //totally outside
            else continue; //totally inside
        t=Wn*(-1.0)/Dn; //calculate parameter
        if(Dn>0) //entering
        {
            if(t>1) return; //totally outside
            else if(t>tl) tl=t; //maximum of lower limits
        }
        else if(t<0) return; //totally outside
        else if(t<tu) tu=t; //exiting, minimum of upper limits
        if(tl>tu) return; //exits before entering, totally outside
    }while(cur=cur->next); //loop over window edges
    clippedLine(x1+(int)((x2-x1)*tl), y1+(int)((y2-y1)*tl),
                x1+(int)((x2-x1)*tu), y1+(int)((y2-y1)*tu)); //output
}

int main(int argc, char **argv)
{
    int gd=DETECT,gm;
    SHAPE lin;
    FILE *kpg;
    if(argc<2)
    {
        printf("[ ERROR ] Usage : <.kpg>_format_input_file\n");
        return 1;
    }
    kpg = fopen(argv[1], "rx"); //open input file
    if(!kpg)
    {
        printf("[ ERROR ] Usage : <.kpg>_format_input_file\n");
        return 1;
    }
    if(!(window=expectPoly(kpg))) //input regular window and line
    {
        printf("\nInvalid File Format\n");
        fclose(kpg);
        return 1;
    }
    initgraph(&gd, &gm, NULL); //initialize graphics
    clippingWindow(window); //draw window
    while(lin=expectLine(kpg))
        beck(lin); //call algorithm
    getchar();
    closegraph();
    erase(window);
    if(!feof(kpg))
    {
        printf("\nInvalid File Format\n");
        fclose(kpg);
        return 1;
    }
    fclose(kpg);
    return 0;
}
```

Output

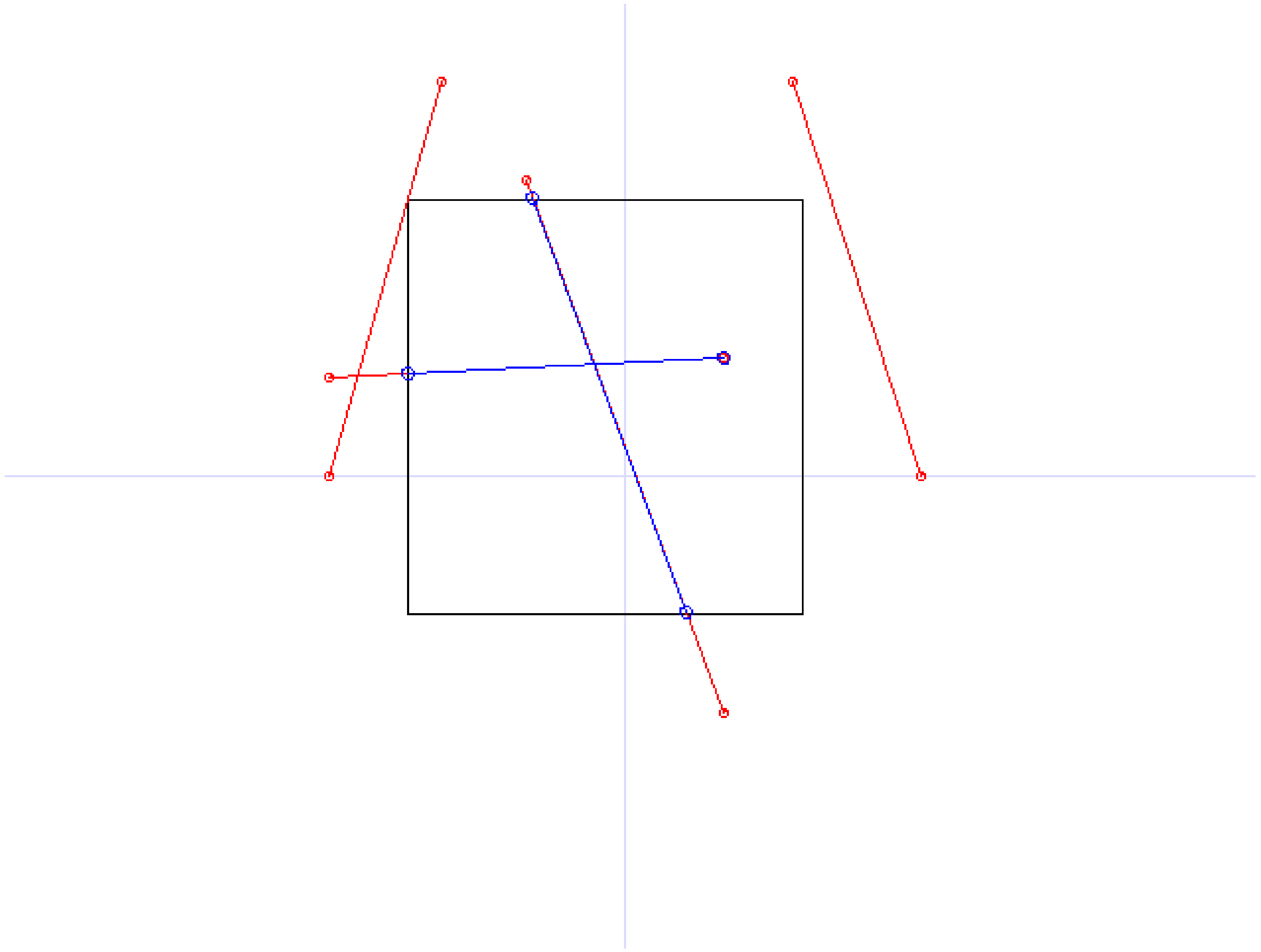


Liang-Barsky Algorithm

Liang-Barsky's Line Clipping Algorithm uses the parametric equation of a line and solves 4 inequalities to find the range of the parameter for which the line is within the viewport. Liang-Barsky is a special case of the Cyrus-Beck Algorithm, only for regular windows.

```
/*                        17.LIANGBARSKY.C                                  */
#include <stdio.h>
#include <graphics.h>
#include "0.kpgparse.c"
#include "0.clipping.c"

int window[4]; //regular window [l, r, b, t]
double tl, tu;

int isIn(int d, int q)
{
    if(!d) //parallel
        if(q<0) return 0; //totally outside
        else return 1; //totally inside
    double t=q*(-1.0)/d;//calculate parameter
    if(d>0) //entering
    {
        if(t>1) return 0; //totally outside
        else if(t>tl) tl=t; //maximum of lower limits
    }
    else if(t<0) return 0; //totally outside
    else if(t<tu) tu=t; //exiting, minimum of upper limits
    if(tl>tu) return 0; //exits before entering, totally outside
    return 1;
}

void liangBarsky(SHAPE lin)
{
    int y1=lin->y;       //get co-ordinates
    int y2=lin->next->y;
    int x1=lin->x;
    int x2=lin->next->x;
    int del = x2-x1; //direction of line
    erase(lin);
    refLine(x1, y1, x2, y2); //original line
    getchar();
    tl=0.0; tu=1.0;
    if( !isIn(del, x1-window[0]) || !isIn(-del, window[1]-x1) ) return; //clip along x-axis
    del = y2-y1; //direction of line
    if( !isIn(del, y1-window[2]) || !isIn(-del, window[3]-y1) ) return; //clip along y-axis
    clippedLine(x1+(int)((x2-x1)*tl), y1+(int)((y2-y1)*tl),
                x1+(int)((x2-x1)*tu), y1+(int)((y2-y1)*tu)); //output
}

int main(int argc, char **argv)
{
    int gd=DETECT,gm;
    FILE *kpg;
    SHAPE wndw, lin;
    if(argc<2)
    {
        printf("[ ERROR ] Usage : <.kpg>_format_input_file\n");
        return 1;
    }
    kpg = fopen(argv[1], "rx"); //open input file
    if(!kpg)
    {
        printf("[ ERROR ] Usage : <.kpg>_format_input_file\n");
        return 1;
    }
    if(!(wndw=expectPoly(kpg))) //input regular window and line
    {
        printf("\nInvalid Format\n");
        fclose(kpg);
        return 1;
    }
    window[3]=wndw->y;      //get co-ordinates of window
    window[0]=wndw->x;
    window[2]=wndw->next->next->y;
    window[1]=wndw->next->next->x;
    if(window[3]<window[2]) { int t=window[3]; window[3]=window[2]; window[2]=t; }
    if(window[1]<window[0]) { int t=window[0]; window[0]=window[1]; window[1]=t; }
                                           //force ordering of edges
    initgraph(&gd, &gm, NULL); //initialize graphics
    clippingWindow(wndw); //draw window
    erase(wndw);
    while(lin=expectLine(kpg))
        liangBarsky(lin); //call algorithm
    getchar();
    closegraph();
    if(!feof(kpg))
    {
        printf("\nInvalid File Format\n");
        fclose(kpg);
        return 1;
    }
    fclose(kpg);
    return 0;
}
```

Output



Sutherland–Hodgeman Algorithm

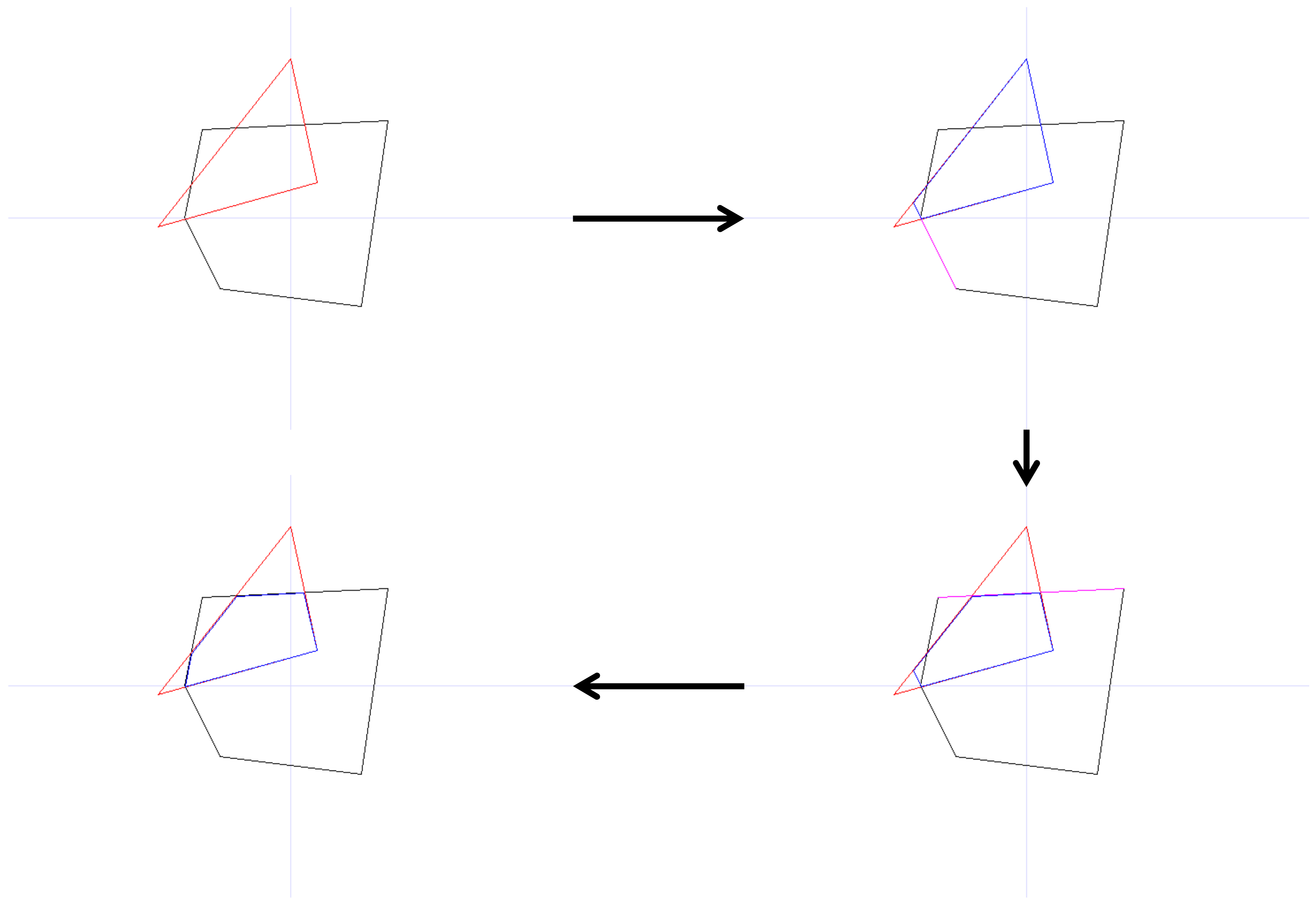
The Sutherland–Hodgeman Algorithm is a Polygon clipping algorithm. It proceeds by iterating over the window edges and creating a new output polygon for each such edge. It does this by iterating over the polygon vertices and determining which of these lie within the viewport relative to the current window edge. Vertices lying outside are clipped. In case of an intersection, the position is calculated by some suitable line clipping algorithm and the intersection point added to the output polygon.

The output list from one stage is used as the input list for the next. Once all sides of the clip polygon have been processed, the final generated list of vertices defines a new single polygon that is entirely visible. If the subject polygon was concave at vertices outside the clipping polygon, the new polygon may have overlapping edges. Such artifacts are generally acceptable for rendering, but not for other applications such as computing shadows.

```
/*                              18.HODGEMAN.C                                   */
#include <stdio.h>
#include <stdlib.h>
#include <graphics.h>
#include "0.kpgparse.c"
#include "0.clipping.c"

SHAPE window;

void DrawPoly(SHAPE gon, int color)
{
    setcolor(color);
    SHAPE vertex1=gon;
    do
    {
        line(320+(gon->x), 240-(gon->y), 320+(gon->next->x), 240-(gon->next->y));
        gon=gon->next;
    }while(gon->next);
    line(320+(gon->x), 240-(gon->y), 320+(vertex1->x), 240-(vertex1->y));
}

int isVis(SHAPE pnt, SHAPE w1, SHAPE w2) // outside=-1, inside=1, on_the_boundary=0
{
    long t= (pnt->y - w1->y)*(w2->x - w1->x) - (pnt->x - w1->x)*(w2->y - w1->y);
    return t>0?1:t<0?-1:0;
}

void hodgeman(SHAPE oriGon)
{
    SHAPE gon=oriGon;
    SHAPE wcur = window; //window_vertex_1
    int vc, vn;
    DrawPoly(gon, RED); //reference
    do
    {
        SHAPE out=NULL; //temporary polygon
        SHAPE wnxt = wcur->next?wcur->next:window;
                     //closed polygon, vertex_1 follows vertex_N
        SHAPE pcur = gon; //polygon_vertex_1
        vn=isVis(pcur, wcur, wnxt);
                       //check visibility of object_vertex_1 w.r.t. window edge
        do
        {
            SHAPE pnxt = pcur->next?pcur->next:gon;
                       //closed polygon, vertex_1 follows vertex_N
            vc=vn;
            vn=isVis(pnxt, wcur, wnxt); //check visibility w.r.t. window edge
            if( (vc<0 && vn>0) || (vc>0 && vn<0) ) //intersection
            {
                SHAPE tmp = (SHAPE)malloc(sizeof(struct point));

                /*int x1=pcur->x;      //calculate values
                int y1=pcur->y;        //original formula
                int x2=pnxt->x;
                int y2=pnxt->y;
                int x3=wcur->x;
                int y3=wcur->y;
                int x4=wnxt->x;
                int y4=wnxt->y;
                tmp->x = ( (x1*y2-y1*x2)*(x3-x4) - (x3*y4-y3*x4)*(x1-x2) )
                       / ( (x1-x2)*(y3-y4) - (y1-y2)*(x3-x4) );
                tmp->y = ( (x1*y2-y1*x2)*(y3-y4) - (x3*y4-y3*x4)*(y1-y2) )
                       / ( (x1-x2)*(y3-y4) - (y1-y2)*(x3-x4) );*/
                int T1, T2, T3, T4; //calculate intersection point by algebraic method
                int A = pcur->x*pnxt->y-pcur->y*pnxt->x;
                int B = wcur->x*wnxt->y-wcur->y*wnxt->x;
                int D = (T1=pcur->x-pnxt->x)*(T2=wcur->y-wnxt->y)
                      - (T3=pcur->y-pnxt->y)*(T4=wcur->x-wnxt->x);
                tmp->x = ( A*T4 - B*T1 ) / D;
                tmp->y = ( A*T2 - B*T3 ) / D;

                tmp->next=out;         //add intersection point to temporary polygon
                out=tmp;
            }
            if(vn>=0) //add visible points to temporary polygon
            {
                SHAPE tmp = (SHAPE)malloc(sizeof(struct point));
                tmp->x=pnxt->x;
                tmp->y=pnxt->y;
                tmp->next=out;
                out=tmp;
            }
        }while(pcur=pcur->next); //loop over polygon vertices/edges
        if(gon!=oriGon)
            erase(gon);
        if(!out) return;
        gon=out; //update polygon
        getchar(); //show progress based clipping against current line
        clippingWindow(window);
        DrawPoly(oriGon, RED);
        setcolor(MAGENTA);
        line(320+wcur->x, 240-wcur->y, 320+wnxt->x, 240-wnxt->y);
        DrawPoly(gon, BLUE);
    }while(wcur=wcur->next); //loop over window edges
    clippingWindow(window);
    DrawPoly(oriGon, RED);
    DrawPoly(gon, BLUE); //output
    erase(gon);
    erase(oriGon);
}
```

```
int main(int argc, char **argv)
{
    int gd=DETECT,gm;
    FILE *kpg;
    SHAPE gon;
    if(argc<2)
    {
        printf("[ ERROR ] Usage : <.kpg>_format_input_file\n");
        return 1;
    }
    kpg = fopen(argv[1], "rx"); //open input file
    if(!kpg)
    {
        printf("[ ERROR ] Usage : <.kpg>_format_input_file\n");
        return 1;
    }
    if(!(window=expectPoly(kpg))) //input regular window and line
    {
         printf("\nInvalid File Format\n");
         fclose(kpg);
         return 1;
     }
     initgraph(&gd, &gm, NULL); //initialize graphics
     clippingWindow(window); //draw window
     while(gon=expectPoly(kpg))
         hodgeman(gon); //call algorithm
     getchar();
     closegraph();
     erase(window);
     if(!feof(kpg))
     {
         printf("\nInvalid File Format\n");
         fclose(kpg);
         return 1;
     }
     fclose(kpg);
     return 0;
 }
```

Output



2D Transformations

The original object is to be provided in a file. The file contains lines and polygons. The transformations are accomplished using 3x3 matrices.

Using homogeneous co-ordinates, we are able to perform the following tranformations:

- Translation – Along x-axis and y-axis
- Scaling – Selectively along the axes or as a whole (homogeneously)
- Rotation – About any arbitrary point
- Reflection – About any arbitrary line
- Shearing – Shear on x co-ordinate due to y co-ordinate and vice versa

A powerful command line interface provides the ability to chain these transformations along with full control over the animation speed. As an alternative, a menu driven interface is also provided. However, each instance of the menu can only perform a single transformation, which it performs in 10 steps, each taking 0.1 seconds.

The animation system breaks the transformation matrix into smaller matrices. This is equivalent to determining the nth root of the matrix, which has no general solution. For some cases, e.g. translation, rotation, we are able to do this intuitively. For others, like reflection, the animation occurs in a single step.

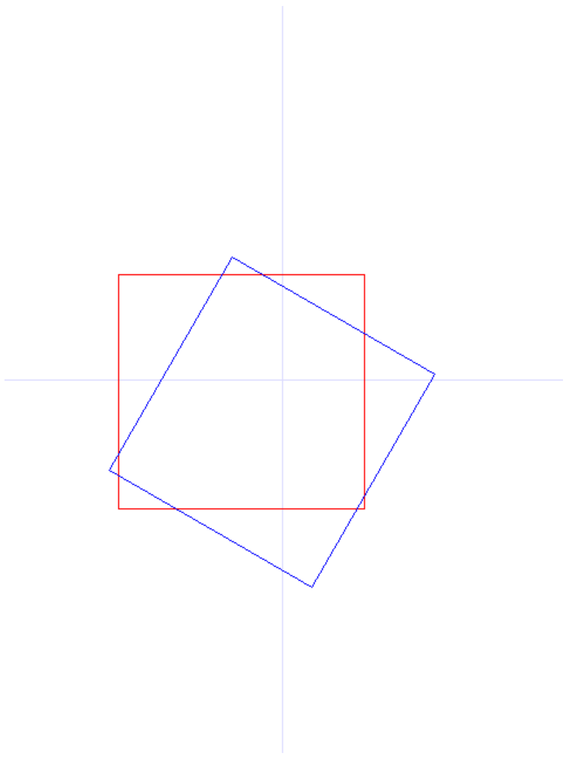
```c
/*
*   2D TRANSFORMATIONS
*   provide a .kpg file as input
*
*   translation -t:x,y
*   rotation    -r:x,y,0
*   shear       -f:x,y
*   reflection  -m:a,b,c
*   scaling     -s:x,y
*   zoom        -z:h
*   custom      -c:a,b,p,c,d,q,m,n,s
*   animate     -a:n,d
*
*/

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include <string.h>
#include <graphics.h>
#include "0.kpgparse.c"

typedef struct
{
    VSHAPE data;
    char type;
}OBJECT;

OBJECT *objects=NULL;
int aniFlag, num_objs=0;
clock_t last=0, now;
double dely=0;

typedef struct
{
    double v[3][3];
}TRANSFORMATION;

TRANSFORMATION multiply(TRANSFORMATION R, TRANSFORMATION T)
{
    int i, j, k;
    TRANSFORMATION F;
    for(i=0; i<3; ++i)
        for(j=0; j<3; ++j)
        {
            F.v[i][j]=0;
            for(k=0; k<3; ++k)
                F.v[i][j] += R.v[i][k]*T.v[k][j];
        }
    return F;
}

void loadObjects(FILE *kpg)
{
    int gd=DETECT,gm; //initialize graphics
    initgraph(&gd, &gm, NULL);
    setcolor(WHITE);
    bar(0, 0, 640, 480);
    setcolor(LIGHTGRAY); //draw axes
    line(320, 0, 320, 480);
    line(0, 240, 640, 240);
    setcolor(BLACK);
```

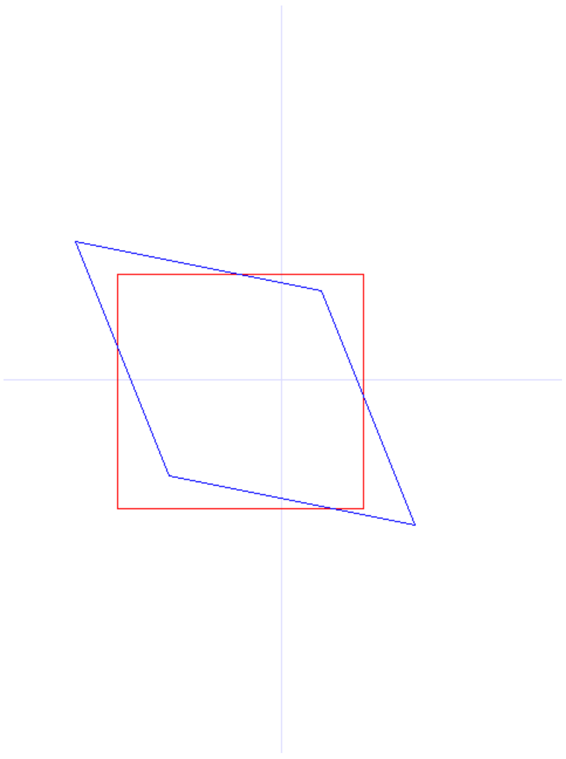
```
        do
        {
            char tp;
            fscanf(kpg, "%c ", &tp);
            if(tp=='l' || tp=='L') //line
            {
                objects = (OBJECT*)realloc(objects, sizeof(OBJECT)*(num_objs+1));
                objects[num_objs].data = toVector(parseLine(kpg)); //read line from file
                objects[num_objs].type = tp;
                if(objects[num_objs].data) ++num_objs;
                else parseComment(kpg);
            }
            else if(tp=='p' || tp=='P') //polygon
            {
                objects = (OBJECT*)realloc(objects, sizeof(OBJECT)*(num_objs+1));
                objects[num_objs].data = toVector(parsePoly(kpg)); //read polygon from file
                objects[num_objs].type = tp;
                if(objects[num_objs].data) ++num_objs;
                else parseComment(kpg);
            }
            else parseComment(kpg);
        }while(!feof(kpg));
        if(kpg) fclose(kpg);
    }
    void display(TRANSFORMATION T)
    {
        int i;
        VSHAPE cur;
        int j;
        while( ((now=clock()) - last)*1.0/CLOCKS_PER_SEC < dely);
                                        //animate output with required delay
        last=now;
        if(aniFlag>0) //animate
        {
            setcolor(WHITE);
            bar(0, 0, 640, 480);
            setcolor(LIGHTGRAY); //draw axes
            line(320, 0, 320, 480);
            line(0, 240, 640, 240);
            setcolor(BLACK);
        }
        else if(!aniFlag) //no animation, show original
        {
            setcolor(RED);
            aniFlag=-1;
        }
        else setcolor(BLUE); //no animation, show transformed object
        for(i=0; i<num_objs; ++i)
        {
            for(cur=objects[i].data; cur!=NULL; cur=cur->next)
            {
                TRANSFORMATION X = {{ {cur->x, cur->y, 1},      //multiply homogeneous
                                      {     0,      0, 0},
                                      {     0,      0, 0}  }};
                X = multiply(X, T);
                cur->x = X.v[0][0]/X.v[0][2];
                cur->y = X.v[0][1]/X.v[0][2];
            }
            if(objects[i].type=='l') //display line
                line(320+objects[i].data->x,
                         240-objects[i].data->y,
                             320+objects[i].data->next->x,
                                 240-objects[i].data->next->y);
            else
            {
                VSHAPE pcur, pnxt;      //display polygon
                pcur = objects[i].data;
                do
                {
                    VSHAPE pnxt = pcur->next?pcur->next:objects[i].data;
                                //closed polygon, vertex_1 follows vertex_N
                    line(320+pcur->x, 240-pcur->y, 320+pnxt->x, 240-pnxt->y);
                }while(pcur=pcur->next);
            }
        }
    }

    void menu(int *argc, char ***argv)
    {
        int choice;
        double x, y, other;
        char **temp;
        printf("Enter your choice:\n1)Scaling\n2)Shearing\n3)Rotation");
        printf("\n4)Reflection\n5)Translation\n6)Exit : ");
        scanf("%d", &choice);
        *argc=4;
        temp = (char**)malloc(4*sizeof(char*));
        temp[0]=(*argv)[0];
        temp[1]=(*argv)[1];
        *argv=temp;
        (*argv)[2] = (char*)malloc(sizeof(char)*20);
        strcpy((*argv)[2], "-a:10,0.1");
        (*argv)[3] = (char*)malloc(sizeof(char)*100);
        switch(choice)
        {
            case 1:
                printf("Enter X Scale Factor : "); scanf("%lf", &x);
                printf("Enter Y Scale Factor : "); scanf("%lf", &y);
                sprintf((*argv)[3], "-s:%lf,%lf", x, y);
                break;
            case 2:
                printf("Enter X Shear Factor : "); scanf("%lf", &x);
                printf("Enter Y Shear Factor : "); scanf("%lf", &y);
                sprintf((*argv)[3], "-f:%lf,%lf", x, y);
                break;
            case 3:
                printf("Enter X Center Location : "); scanf("%lf", &x);
                printf("Enter Y Center Location : "); scanf("%lf", &y);
                printf("Enter Rotation Angle : "); scanf("%lf", &other);
                sprintf((*argv)[3], "-r:%lf,%lf,%lf", x, y, other);
                break;
            case 4:
                printf("Reflect About Line...\nEnter Slope : "); scanf("%lf", &x);
                printf("Enter Y Intercept : "); scanf("%lf", &y);
                sprintf((*argv)[3], "-m:%lf,%lf", x, y);
                break;
            case 5:
                printf("Translate Along X Axis : "); scanf("%lf", &x);
                printf("Translate Along Y Axis: "); scanf("%lf", &y);
                sprintf((*argv)[3], "-t:%lf,%lf", x, y);
                break;
            default:
                strcpy((*argv)[3], "");
        }
    }
```

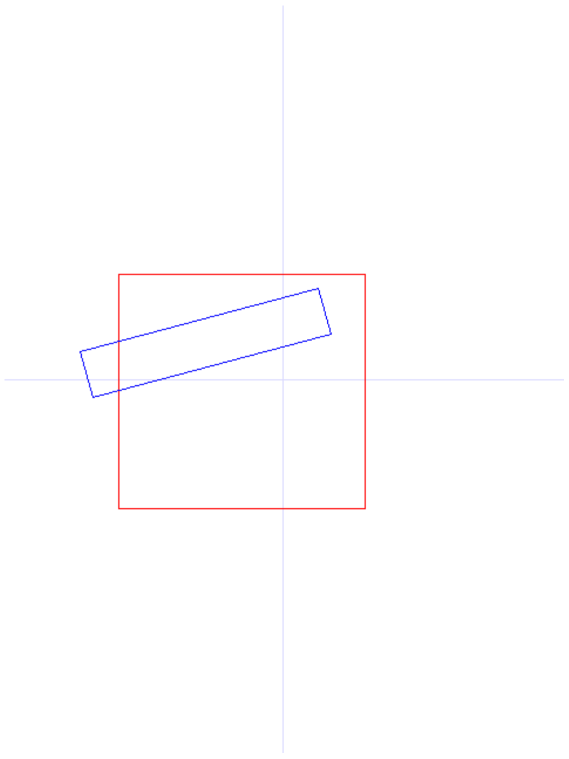
```c
int main(int argc, char **argv)
{
    int steps;
    int cnt, i=1;
    int gd=DETECT,gm;
    FILE *kpg;
    double x, y, angle;
    double a, b, p,
           c, d, q,
           m, n, s;
    if(argc<2)
    {
        printf("[ ERROR ] Usage : <.kpg>_format_input_file [options...]\n");
        return 1;
    }
    kpg = fopen(argv[1], "rx"); //open input file
    if(!kpg)
    {
        printf("[ ERROR ] Usage : <.kpg>_format_input_file [options...]\n");
        return 1;
    }
    TRANSFORMATION T = {{ {1, 0, 0},
                          {0, 1, 0},
                          {0, 0, 1} }};
    TRANSFORMATION I = T;
    aniFlag=0;
    if(argc==2) menu(&argc, &argv); //default to menu driven
    while((i++)<argc-1) //parse the arguments
    {
        if(argv[i][0]=='-') //commands start with a '-'
            switch(argv[i][1])
            {
                case 'a': case 'A': //animation
                    if(sscanf(argv[i]+2, ":%d,%lf", &steps, &dely)==2)
                    //number of steps per operation, time delay per step
                    {
                        aniFlag = 1;
                        loadObjects(kpg); //Load from file
                        display(I); //display original
                        getchar();
                        display(T); //display current
                    }
                    else fprintf(stderr,
   "LOG [ERROR] : INVALID SYNTAX. PLEASE USE WITH NO SPACES : -A:FRAMES/OP,DELAY\n");
                break;

                case 't': case 'T': //translation
                    if(sscanf(argv[i]+2, ":%lf,%lf", &x, &y)==2) //along x-axis, along y-axis
                    {
                        TRANSFORMATION R = {{ {1, 0, 0}, //translate
                                              {0, 1, 0},
                                              {x, y, 1}  }};
                        if(aniFlag)
                        {
                            R.v[2][0]/=steps; //break into n steps
                            R.v[2][1]/=steps;
                            for(cnt=0; cnt<steps; ++cnt)
                                display(R); //animate
                        }
                        else T = multiply(T, R); //update transformation matrix
                    }
                    else fprintf(stderr,
   "LOG [ERROR] : INVALID SYNTAX. PLEASE USE WITH NO SPACES : -T:X_DISP,Y_DISP\n");
                break;
            case 's': case 'S': //scaling
                if(sscanf(argv[i]+2, ":%lf,%lf", &x, &y)==2) //scale x, scale y
                {
                    x=x>0?x:-x;
                    y=y>0?y:-y;
                    TRANSFORMATION R = {{ {x, 0, 0}, //scale
                                          {0, y, 0},
                                          {0, 0, 1}  }};
                    if(aniFlag)
                    {
                        R.v[0][0] = pow(R.v[0][0], 1.0/steps); //break into n steps
                        R.v[1][1] = pow(R.v[1][1], 1.0/steps);
                        for(cnt=0; cnt<steps; ++cnt)
                            display(R); //animate
                    }
                    else T = multiply(T, R); //update transformation matrix
                }
                else fprintf(stderr,
        "LOG [ERROR] : INVALID SYNTAX. PLEASE USE WITH NO SPACES : -S:X_SCALE,Y_SCALE\n");
            break;

            case 'r': case 'R': //rotation
                if(sscanf(argv[i]+2, ":%lf,%lf,%lf", &x, &y, &angle)==3)
                                     //center of rotation x, y, angle of rotation
                {
                    angle*=3.1415926/180;
                    TRANSFORMATION R = {{ { 1,  0, 0},
                                          { 0,  1, 0},
                                          {-x, -y, 1}  }};
                                          //pass axis of rotation through origin
                    TRANSFORMATION R_INV = {{ {1, 0, 0},
                                              {0, 1, 0},
                                              {x, y, 1}  }};
                    if(aniFlag) angle/=steps; //break into n steps
                    c=cos(angle);
                    s=sin(angle);
                    TRANSFORMATION W = {{ { c, s, 0}, //rotate
                                          {-s, c, 0},
                                          { 0, 0, 1}  }};
                    R = multiply(R, multiply(W, R_INV));
                                 //calculate effective transformation matrix
                    if(aniFlag)
                        for(cnt=0; cnt<steps; ++cnt)
                            display(R); //animate
                    else T = multiply(T, R); //update transformation matrix
                }
                else fprintf(stderr,
        "LOG [ERROR] : INVALID SYNTAX. PLEASE USE WITH NO SPACES : -R:X_CEN,Y_CEN,ANGLE\n");
            break;
            case 'c': case 'C': //custom homogeneous matrix
                if(sscanf(argv[i]+2, ":%lf,%lf,%lf,%lf,%lf,%lf,%lf,%lf,%lf",
                                     &a, &b, &p, &c, &d, &q, &m, &n, &s)==9)
                {
                    TRANSFORMATION R = {{ {a, b, p},
                                          {c, d, q},
                                          {m, n, s}  }};
                    if(aniFlag) display(R); //cannot animate, show immediately
                    else T = multiply(T, R); //update transformation matrix
                }
                else fprintf(stderr,
            "LOG [ERROR] : INVALID SYNTAX. PLEASE USE WITH NO SPACES : -C:\n");
            break;
            case 'f': case 'F': //shearing
                if(sscanf(argv[i]+2, ":%lf,%lf", &x, &y)==2) //x shear, y shear
                {
                    TRANSFORMATION R = {{ {1, y, 0}, //shear
                                          {x, 1, 0},
                                          {0, 0, 1}  }};
                    if(aniFlag)
                    {
                        R.v[0][1] = y/steps;
                        R.v[1][0] = 0;
                        for(cnt=0; cnt<steps; ++cnt)
                            display(R); //animate shear along y-axis
                        R.v[1][0] = x/steps;
                        R.v[0][1] = 0;
                        for(cnt=0; cnt<steps; ++cnt)
                            display(R); //animate shear along x-axis
                        R.v[1][0]=x*x*y;
                        R.v[0][1]=0;
                        R.v[0][0]=1-x*y;
                        display(R); //correction term
                    }
                    else T = multiply(T, R); //update transformation matrix
                }
                else fprintf(stderr,
   "LOG [ERROR] : INVALID SYNTAX. PLEASE USE WITH NO SPACES : -F:X_SHEAR,Y_SHEAR\n");
            break;

            case 'z': case 'Z': //zooming
                if(sscanf(argv[i]+2, ":%lf", &s)==1) //homogeneous zoom factor
                {
                    if(s==0.0)
                    {
                        fprintf(stderr, "LOG [ERROR] : CANNOT DELETE PICTURE!");
                        break;
                    }
                    TRANSFORMATION R = {{ {1, 0,   0}, //homogeneous scaling
                                          {0, 1,   0},
                                          {0, 0, 1/s}  }};
                    if(s<0) //cannot animate reflection, correction term
                    {
                        R.v[0][0]=-R.v[0][0];
                        R.v[1][1]=-R.v[1][1];
                        R.v[2][2]=-R.v[2][2];
                    }
                    if(aniFlag)
                    {
                        R.v[2][2] = pow(R.v[2][2], 1.0/steps); //break into n steps
                        for(cnt=0; cnt<steps; ++cnt)
                            display(R); //animate
                    }
                    else T = multiply(T, R); //update transformation matrix
                }
                else fprintf(stderr,
       "LOG [ERROR] : INVALID SYNTAX. PLEASE USE WITH NO SPACES : -Z:ZOOM\n");
            break;

            default:
                fprintf(stderr,
            "LOG [ERROR] : INVALID OPERATION NOT RECOGINZED : %s\n", argv[i]);
            break;
```

```
                case 'm': case 'M': //reflection
                    if(sscanf(argv[i]+2, ":%lf,%lf,%lf", &a, &b, &c)==3)
                                                             //equation of mirror line
                    {
                        if(b)
                        {
                            x = -a/b; //slope
                            y = -c/b; //intercept
                        }
                        c = pow(1/(x*x + 1), 0.5); //calculate cos and sin from tan
                        s = (x>0?1:-1)*pow(1-c*c, 0.5);
                        TRANSFORMATION R = {{ {1, 0, 0}, //pass mirror line through origin
                                              {0, 1, 0},
                                              {0, y, 1}  }};

                        TRANSFORMATION R_INV = {{ {1,  0, 0},
                                                  {0,  1, 0},
                                                  {0, -y, 1}  }};

                        TRANSFORMATION H = {{ { c, -s, 0}, //remove y component
                                              { s,  c, 0},
                                              { 0,  0, 1}  }};

                        TRANSFORMATION H_INV = {{ { c, s, 0},
                                                  {-s, c, 0},
                                                  { 0, 0, 1}  }};

                        TRANSFORMATION M = {{ {1,  0, 0},
                                              {0, -1, 0},
                                              {0,  0, 1}  }};
                        if(!b) //special case mirror is vertical
                        {
                            M.v[0][0]=-1;
                            M.v[1][1]=1;
                        }
                        R = multiply( b?multiply(R, H):I,
                            multiply(M, b?multiply(H_INV, R_INV):I ) );
                                    //calculate effective transformation matrix
                        if(aniFlag) display(R); //cannot animate reflection, show immediately
                        else T = multiply(T, R); //update transformation matrix
                    }
                    else fprintf(stderr,
        "LOG [ERROR] : INVALID LINE ax+by+c=0. PLEASE USE WITH NO SPACES : -M:a,b,c\n");
                break;
            }
    }

    if(!aniFlag)
    {
        loadObjects(kpg); //load from file
        display(I); //display original object
        getchar();
        display(T); //display transformed object
    }
    for(i=0; i<num_objs; ++i) eraseVector(objects[i].data); //clear memory
    getchar();
    closegraph();
    return 0;
}
```

Output







3D Projections and Transformations

The original object is to be provided in a file. The file contains a graph, represented as co-ordinates of vertices and an adjacency matrix. The transformations are accomplished using 4x4 matrices.

Using homogeneous co-ordinates, we are able to perform the following tranformations:

- Perspective Projection – From an arbitrary center of projection, to an arbitrary plane of projection
- Axonometric Projection - From an arbitrary center of projection, in an arbitrary direction.
- Translation – Along x-axis, y-axis, and z-axis
- Scaling – Selectively along the axes or as a whole (homogeneously)
- Rotation – About any arbitrary axis
- Reflection – About any arbitrary plane

A powerful command line interface provides the ability to chain these transformations along with full control over the animation speed and freedom to change the camera angle/projection style. As an alternative, a menu driven interface is also provided. However, each instance of the menu can only perform a single transformation, which it performs in 10 steps, each taking 0.1 seconds.

The animation system breaks the transformation matrix into smaller matrices. This is equivalent to determining the nth root of the matrix, which has no general solution. For some cases, e.g. translation, rotation, we are able to do this intuitively. For others, like reflection, the animation occurs in a single step.

Internally, the projection matrix is stored separately from the graph. The object is transformed by normal affine transformations. Before displaying, the projection matrix (either perspective or axonometric) is applied. Note that with suitable camera position and projecting plane, we can obtain 1-point, 2-point and 3-point perspectives. The axonometric can also reduce to an orthographic projection in case the viewing direction is along one of the co-ordinate axes.

```c
/*
*   3D TRANSFORMATIONS AND PROJECTIONS
*   provide a .kpg file as input
*
*   translation -t:x,y,z
*   rotation    -r:x,y,z,a,b,c,θ
*   reflection  -m:x,y,z,a,b,c
*   scaling     -s:x,y,z
*   zoom        -z:h
*   perspective -p:x,y,z,a,b,c
*   axonometric -x:x,y,z,a,b,c
*   animate     -a:n,d
*
*/

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include <string.h>
#include <graphics.h>
#include "0.kpgparse.c"

VGRAPH objects;
int aniFlag, num_objs=0;
clock_t last=0, now;
double dely=0;

typedef struct
{
    double v[4][4];
}TRANSFORMATION;

TRANSFORMATION multiply(TRANSFORMATION R, TRANSFORMATION T)
{
    int i, j, k;
    TRANSFORMATION F;
    for(i=0; i<4; ++i)
        for(j=0; j<4; ++j)
        {
            F.v[i][j]=0;
            for(k=0; k<4; ++k)
                F.v[i][j] += R.v[i][k]*T.v[k][j];
        }
    return F;
}

TRANSFORMATION axes[6] = {{{ {  1,  0,  0, 0}, //x-axis
                             { -1,  0,  0, 0},
                             {  0,  0,  0, 0},
                             {  0,  0,  0, 0}, }},
                          {{ {  0,  1,  0, 0}, //y-axis
                             {  0, -1,  0, 0},
                             {  0,  0,  0, 0},
                             {  0,  0,  0, 0}, }},
                          {{ {  0,  0,  1, 0}, //z-axis
                             {  0,  0, -1, 0},
                             {  0,  0,  0, 0},
                             {  0,  0,  0, 0}, }}};
```

```
void xclip(double x1, double y1, double x2, double y2) //sutherland-cohen clipping
{
    unsigned char code;
    double slope;
    int i, window[4]={-320, 320, -240, 240};
    unsigned char calcEPC(double x, double y) //calculate End Point Code
    {
        unsigned char c=0;
        if(y>240) c+=8; //above t
        if(y<-240) c+=4; //below b
        if(x>320) c+=2; //right of r
        if(x<-320) c+=1; //left of l
        return c;
    }
    code=(calcEPC(x1, y1)<<4) + calcEPC(x2, y2); //calculate end point codes
    if(!code) //trivially visible
    {
        line(320+(int)x1, 240-(int)y1, 320+(int)x2,240-(int)y2);
        return;
    }
    if((code>>4)&code) return; //trivially invisible
    slope=(x1==x2)?0:(double)(y2-y1)/(x2-x1); //calculate slope
    while(code) //while partially visible
    {
        for(i=0; i<4; ++i) //each window edge
        {
            if(((code>>(4+i))&1)==((code>>i)&1)) continue; //does not cross the edge
                                                     //(x1,y2) is inside relative to edge
            if(!((code>>(4+i))&1)) { int t=x1; x1=x2; x2=t;
                                       t=y1; y1=y2; y2=t;
                                       code=((code&15)<<4)+(code>>4); } //so swap
            if(x1!=x2 && i<2) //left and right edges
            {
                y1+=slope*(window[i]-x1); //find intersection
                x1=window[i];
            }
            else if(y1!=y2 && i>1) //top and bottom edges
            {
                x1+=slope?(window[i]-y1)/slope:0; //find intersection
                y1=window[i];
            }
            code=(calcEPC(x1, y1)<<4)+(code&15); //update end point codes
            if(!code) break; //totally visible
            if((code>>4)&code) return; //totally invisible
        }
    }
    line(320+(int)x1, 240-(int)y1, 320+(int)x2,240-(int)y2);
}

void drawAxes(TRANSFORMATION P)
{
    int i;
    setcolor(WHITE);
    bar(0, 0, 640, 480);
    setcolor(LIGHTGRAY);
    for(i=0; i<3; ++i)
    {
        axes[3+i] = multiply(axes[i], P);
        axes[3+i].v[0][0]/=axes[3+i].v[0][3];
        axes[3+i].v[0][1]/=axes[3+i].v[0][3];
        axes[3+i].v[1][0]/=axes[3+i].v[1][3];
        axes[3+i].v[1][1]/=axes[3+i].v[1][3];
        xclip( axes[3+i].v[0][0], axes[3+i].v[0][1],
               axes[3+i].v[1][0], axes[3+i].v[1][1]  );
    }
    setcolor(BLACK);
}

void loadObjects(FILE *kpg, TRANSFORMATION P)
{
    int gd=DETECT,gm; //initialize graphics
    initgraph(&gd, &gm, NULL);
    objects=expectGraph(kpg); //load picture from file
    fclose(kpg);
    drawAxes(P);
}

void display(TRANSFORMATION T, TRANSFORMATION P)
{
    int i, j;
    VGRAPH temp;
    temp.n=objects.n;
    temp.vertices = (double**)malloc(temp.n*sizeof(double*));
    temp.vertices[0] = (double*)malloc(temp.n*3*sizeof(double));
    while( ((now=clock()) - last)*1.0/CLOCKS_PER_SEC < dely); //animate with time delay
    last=now;
    if(aniFlag>0) drawAxes(P); //animate
    else if(!aniFlag) //no animation, show original
    {
        setcolor(RED);
        aniFlag=-1;
    }
    else setcolor(BLUE); //no animation, show transformed object
    for(i=0; i< objects.n; ++i)
    {
        int ii, jj;
        temp.vertices[i] = temp.vertices[0] + 3*i;
        TRANSFORMATION X = {{ {objects.vertices[i][0], objects.vertices[i][1],
                               objects.vertices[i][2], 1},
                              { 0, 0, 0, 0},
                              { 0, 0, 0, 0},
                              { 0, 0, 0, 0}  }};
        X = multiply(X, T);
        objects.vertices[i][0] = X.v[0][0]/X.v[0][3]; //update picture
        objects.vertices[i][1] = X.v[0][1]/X.v[0][3];
        objects.vertices[i][2] = X.v[0][2]/X.v[0][3];
        X = multiply(X, P);
        temp.vertices[i][0] = X.v[0][0]/X.v[0][3]; //project picture
        temp.vertices[i][1] = X.v[0][1]/X.v[0][3];
        temp.vertices[i][2] = X.v[0][2]/X.v[0][3];
    }
    for(i=0; i<objects.n; ++i)
        for(j=0; j<i; ++j)
        {
            if(objects.adj[i][j]==1) //output
                xclip(temp.vertices[i][0], temp.vertices[i][1],
                      temp.vertices[j][0], temp.vertices[j][1]);
        }
    free(temp.vertices[0]); //free memory
    free(temp.vertices);
}
```

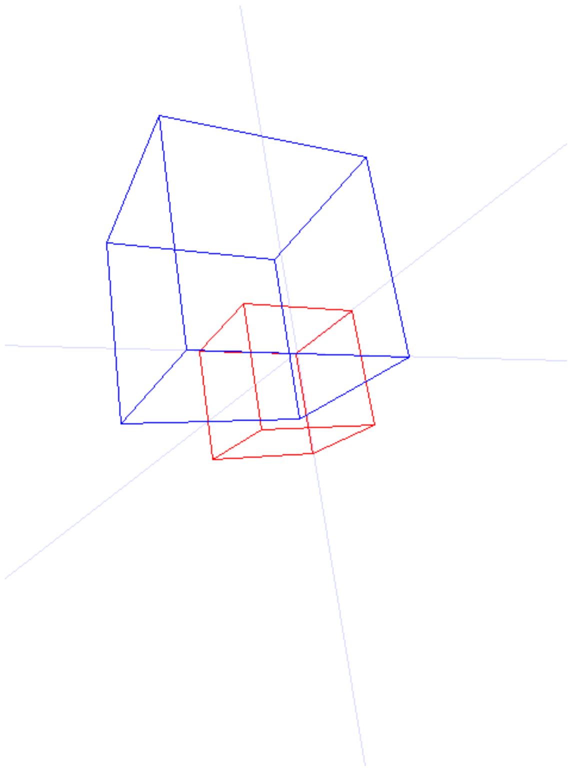
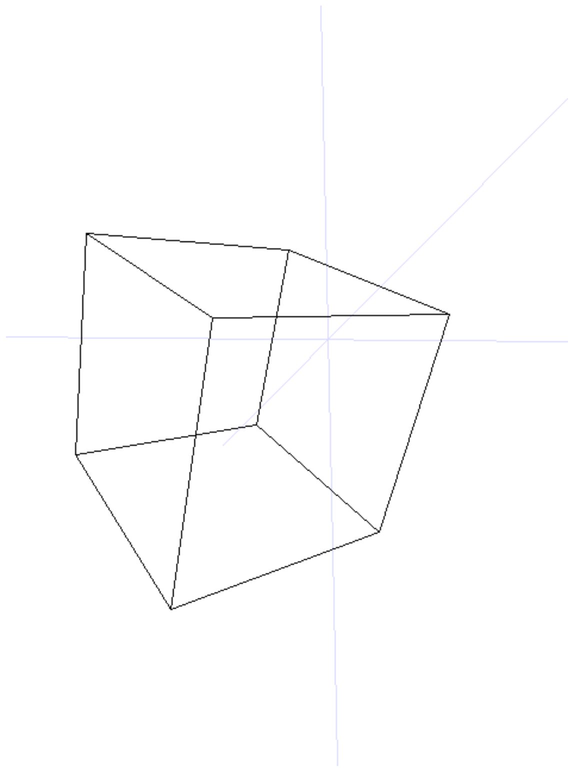
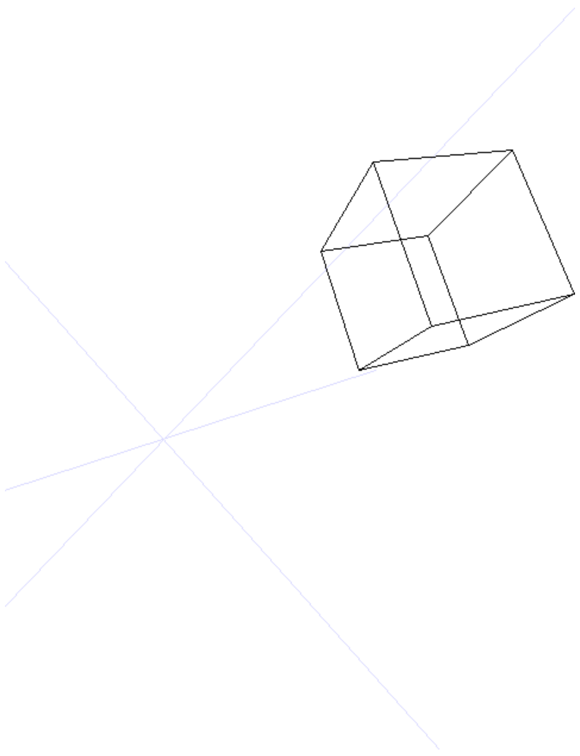
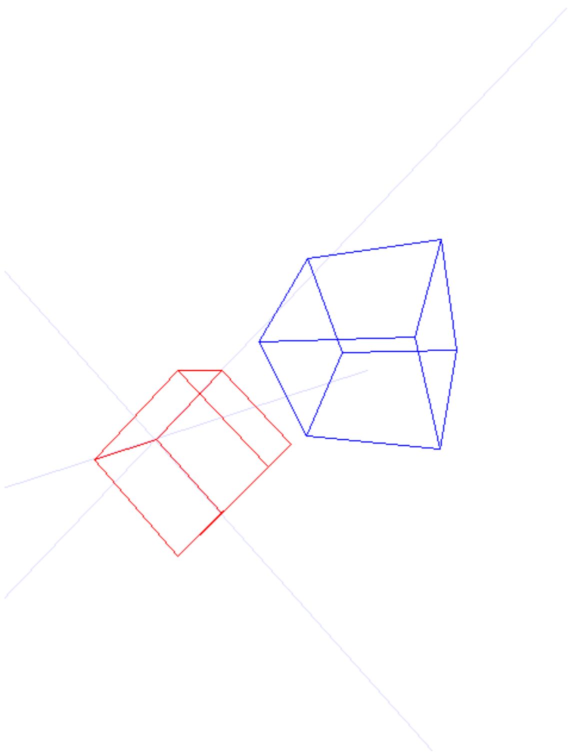
```c
void menu(int *argc, char ***argv)
{
    int choice;
    double x, y, z, other;
    char **temp;
    printf("Enter your choice:\n1)Scaling\n2)Shearing\n3)Rotation");
    printf("\n4)Reflection\n5)Translation\n6)Exit : ");
    scanf("%d", &choice);
    *argc=4;
    temp = (char**)malloc(4*sizeof(char*));
    temp[0]=(*argv)[0];
    temp[1]=(*argv)[1];
    *argv=temp;
    (*argv)[2] = (char*)malloc(sizeof(char)*20);
    strcpy((*argv)[2], "-a:10,0.1");
    (*argv)[3] = (char*)malloc(sizeof(char)*100);
    switch(choice)
    {
        case 1:
            printf("Enter X Scale Factor : "); scanf("%lf", &x);
            printf("Enter Y Scale Factor : "); scanf("%lf", &y);
            printf("Enter Z Scale Factor : "); scanf("%lf", &z);
            sprintf((*argv)[3], "-s:%lf,%lf,%lf", x, y, z);
            break;
        case 2:
            printf("Enter X Shear Factor : "); scanf("%lf", &x);
            printf("Enter Y Shear Factor : "); scanf("%lf", &y);
            printf("Enter Z Shear Factor : "); scanf("%lf", &z);
            sprintf((*argv)[3], "-f:%lf,%lf,%lf", x, y, z);
            break;
        case 3:
            printf("Enter X Center Location : "); scanf("%lf", &x);
            printf("Enter Y Center Location : "); scanf("%lf", &y);
            printf("Enter Rotation Angle : "); scanf("%lf", &other);
            sprintf((*argv)[3], "-r:%lf,%lf,%lf", x, y, other);
            break;
        case 4:
            printf("Reflect About Line...\nEnter Slope : "); scanf("%lf", &x);
            printf("Enter Y Intercept : "); scanf("%lf", &y);
            sprintf((*argv)[3], "-m:%lf,%lf", x, y);
            break;
        case 5:
            printf("Translate Along X Axis : "); scanf("%lf", &x);
            printf("Translate Along Y Axis: "); scanf("%lf", &y);
            printf("Translate Along Z Axis: "); scanf("%lf", &z);
            sprintf((*argv)[3], "-t:%lf,%lf,%lf", x, y, z);
            break;
        default:
            strcpy((*argv)[3], "");
    }
}

int main(int argc, char **argv)
{
    int steps;
    int cnt, i=1;
    FILE *kpg;
    if(argc<2)
    {
        printf("[ ERROR ] Usage : <.kpg>_format_input_file [options...]\n");
        return 1;
    }
    kpg = fopen(argv[1], "rx"); //open input file
    if(!kpg)
    {
        printf("[ ERROR ] Usage : <.kpg>_format_input_file [options...]\n");
        return 1;
    }
```

```
TRANSFORMATION T = {{ {1, 0, 0, 0},
                      {0, 1, 0, 0},
                      {0, 0, 1, 0},
                      {0, 0, 0, 1} }};
TRANSFORMATION I = T;
TRANSFORMATION P = I;
aniFlag=0;
if(argc==2) menu(&argc, &argv); //default to menu
double x, y, z, angle;
double a, b, c, d, p, q, s;
while((i++)<argc-1) //parse the arguments
{
    if(argv[i][0]=='-') //commands start with a '-'
        switch(argv[i][1])
        {
            case 'a': case 'A': //animation
                if(sscanf(argv[i]+2, ":%d,%lf", &steps, &dely)==2)
                    //number of steps per operation, time delay per step
                {
                    aniFlag = 1;
                    loadObjects(kpg, P);
                    display(I, P);
                    getchar();
                    display(T, P);
                }
                else printf("LOG [ERROR] : INVALID SYNTAX. "
                "PLEASE USE WITH NO SPACES : -A:FRAMES/OP,DELAY\n");
                break;
            case 't': case 'T': //translation
                if(sscanf(argv[i]+2, ":%lf,%lf,%lf", &x, &y, &z)==3)
                                     //along x-axis, along y-axis, along z-axis
                {
                    TRANSFORMATION R = {{ {1, 0, 0, 0}, //translate
                                          {0, 1, 0, 0},
                                          {0, 0, 1, 0},
                                          {x, y, z, 1}  }};
                    if(aniFlag)
                    {
                        R.v[3][0]/=steps; //break into n steps
                        R.v[3][1]/=steps;
                        R.v[3][2]/=steps;
                        for(cnt=0; cnt<steps; ++cnt)
                            display(R, P); //animate
                    }
                    else T = multiply(T, R); //update transformation matrix
                }
                else printf("LOG [ERROR] : INVALID SYNTAX. "
                "PLEASE USE WITH NO SPACES : -T:X_DISP,Y_DISP,Z_DISP\n");
                break;
            case 's': case 'S': //scaling
                if(sscanf(argv[i]+2, ":%lf,%lf,%lf", &x, &y, &z)==3)
                                            //scale x, scale y, scale z
                {
                    x=x>0?x:-x;
                    y=y>0?y:-y;
                    z=z>0?z:-z;
                    TRANSFORMATION R = {{ { x, 0, 0, 0}, //scale
                                          { 0, y, 0, 0},
                                          { 0, 0, z, 0},
                                          { 0, 0, 0, 1}  }};
                    if(aniFlag)
                    {
                        R.v[0][0] = pow(R.v[0][0], 1.0/steps); //break into n steps
                        R.v[1][1] = pow(R.v[1][1], 1.0/steps);
                        R.v[2][2] = pow(R.v[2][2], 1.0/steps);
                        for(cnt=0; cnt<steps; ++cnt)
                            display(R, P); //animate
                    }
                    else T = multiply(T, R); //update transformation matrix
                }
                else printf("LOG [ERROR] : INVALID SYNTAX. "
                "PLEASE USE WITH NO SPACES : -S:X_SCALE,Y_SCALE,Z_SCALE\n");
                break;
            case 'r': case 'R': //rotation
                if(sscanf(argv[i]+2, ":%lf,%lf,%lf,%lf,%lf,%lf,%lf",
                                          &x, &y, &z, &a, &b, &c, &angle)==7)
//axis of rotation - (position vector x, y, z,
//                    direction vector a, b, c), angle of rotation
                {
                    angle*=3.1415926/180;
                    q = sqrt(a*a+c*c);
                    d = sqrt(a*a+b*b+c*c);
                    TRANSFORMATION T1 = {{ {  1,  0,  0, 0},
                                           {  0,  1,  0, 0},
                                           {  0,  0,  1, 0},
                                           { -x, -y, -z, 1}  }};
                       //pass axis of rotation through origin

                    TRANSFORMATION T1_INV = {{ { 1, 0, 0, 0},
                                               { 0, 1, 0, 0},
                                               { 0, 0, 1, 0},
                                               { x, y, z, 1}  }};

                    TRANSFORMATION Ry = {{ {  c, 0, a, 0}, //remove x component
                                           {  0, q, 0, 0},
                                           { -a, 0, c, 0},
                                           {  0, 0, 0, q}  }};

                    TRANSFORMATION Ry_INV = {{ { c, 0, -a, 0},
                                               { 0, q,  0, 0},
                                               { a, 0,  c, 0},
                                               { 0, 0,  0, q}  }};

                    TRANSFORMATION Rx = {{ { d,  0, 0, 0}, //remove y component
                                           { 0,  q, b, 0},
                                           { 0, -b, q, 0},
                                           { 0,  0, 0, d}  }};

                    TRANSFORMATION Rx_INV = {{ { d, 0,  0, 0},
                                               { 0, q, -b, 0},
                                               { 0, b,  q, 0},
                                               { 0, 0,  0, d}  }};
                    if(aniFlag) angle/=steps; //break into n steps
                    b = cos(angle);
                    s = sin(angle);
                    TRANSFORMATION R = {{ {  b, s, 0, 0}, //rotate
                                          { -s, b, 0, 0},
                                          {  0, 0, 1, 0},
                                          {  0, 0, 0, 1}  }};
                    R = multiply(multiply(T1, (a||c)?Ry:I), multiply(Rx, R));
                    R = multiply(multiply(R, Rx_INV),
                        multiply((a||c)?Ry_INV:I, T1_INV));
                    if(aniFlag)
                        for(cnt=0; cnt<steps; ++cnt)
                            display(R, P); //animate
                    else T = multiply(T, R); //update transformation matrix
                }
                else printf("LOG [ERROR] : INVALID SYNTAX. "
                "PLEASE USE WITH NO SPACES : -R:Cx,Cy,Cz,Nx,Ny,Nz,ANGLE\n");
                break;
```

```
case 'z': case 'Z': //zooming
    if(sscanf(argv[i]+2, ":%lf", &s)==1)//homogeneous zoom factor
    {
        if(s==0.0)
        {
            fprintf(stderr, "LOG [ERROR] : CANNOT DELETE PICTURE!");
            break;
        }
        TRANSFORMATION R = {{ { 1, 0, 0,   0}, //homogeneous scaling
                              { 0, 1, 0,   0},
                              { 0, 0, 1,   0},
                              { 0, 0, 0, 1/s}  }};
        if(s<0)
        {
            R.v[0][0]=-R.v[0][0];
            R.v[1][1]=-R.v[1][1];
            R.v[2][2]=-R.v[2][2];
            R.v[3][3]=-R.v[3][3];
        }
        if(aniFlag)
        {
            R.v[3][3] = pow(R.v[3][3], 1.0/steps); //break into n steps
            for(cnt=0; cnt<steps; ++cnt)
                display(R, P); //animate
        }
        else T = multiply(T, R); //update transformation matrix
    }
    else printf("LOG [ERROR] : INVALID SYNTAX. "
    "PLEASE USE WITH NO SPACES : -Z:ZOOM\n");
    break;

case 'm': case 'M': //reflection
    if(sscanf(argv[i]+2, ":%lf,%lf,%lf,%lf,%lf,%lf",
                           &x, &y, &z, &a, &b, &c)==6)
                 //mirror plane - (position vector x, y, z,
                 //                normal vector a, b, c)
    {
        q = sqrt(a*a+c*c);
        d = sqrt(a*a+b*b+c*c);
        TRANSFORMATION T1 = {{ {  1,  0,  0, 0},
                               {  0,  1,  0, 0},
                               {  0,  0,  1, 0},
                               { -x, -y, -z, 1}  }};
                     //pass mirror plane through origin

        TRANSFORMATION T1_INV = {{ { 1, 0, 0, 0},
                                   { 0, 1, 0, 0},
                                   { 0, 0, 1, 0},
                                   { x, y, z, 1}  }};

        TRANSFORMATION Ry = {{ {  c, 0, a, 0}, //remove x component
                               {  0, q, 0, 0},
                               { -a, 0, c, 0},
                               {  0, 0, 0, q}  }};

        TRANSFORMATION Ry_INV = {{ { c, 0, -a, 0},
                                   { 0, q,  0, 0},
                                   { a, 0,  c, 0},
                                   { 0, 0,  0, q}  }};

        TRANSFORMATION Rx = {{ { d,  0, 0, 0}, //remove y component
                               { 0,  q, b, 0},
                               { 0, -b, q, 0},
                               { 0,  0, 0, d}  }};
        TRANSFORMATION Rx_INV = {{ { d, 0,  0, 0},
                                   { 0, q, -b, 0},
                                   { 0, b,  q, 0},
                                   { 0, 0,  0, d}  }};

        TRANSFORMATION R = {{ { 1, 0,  0, 0}, //reflect
                              { 0, 1,  0, 0},
                              { 0, 0, -1, 0},
                              { 0, 0,  0, 1}  }};
        R = multiply(multiply(T1, (a||c)?Ry:I), multiply(Rx, R));
        R = multiply(multiply(R, Rx_INV),
            multiply((a||c)?Ry_INV:I, T1_INV));
        if(aniFlag) display(R, P); //animate
        else T = multiply(T, R); //update transformation matrix
    }
    else printf("LOG [ERROR] : INVALID SYNTAX. "
    "PLEASE USE WITH NO SPACES : -M:Px,Py,Pz,Nx,Ny,Nz\n");
    break;

case 'x': case 'X': //axonometric projection
case 'p': case 'P': //perspective projection
    if(sscanf(argv[i]+2, ":%lf,%lf,%lf,%lf,%lf,%lf",
                           &x, &y, &z, &a, &b, &c)==6)
                         //camera position vector x, y, z,
                         //focus point on projection plane a, b, c
    {

        TRANSFORMATION T1 = {{ {  1,  0,  0, 0},
                               {  0,  1,  0, 0},
                               {  0,  0,  1, 0},
                               { -a, -b, -c, 1}  }};
                       //pass projection plane through origin
        a=x-a; b=y-b; c=z-c; //get normal vector
        q = sqrt(a*a+c*c); d = sqrt(a*a+b*b+c*c);
        p = (argv[i][1]=='p' || argv[i][1]=='P') ? (d ? -1/d : 0) : 0;
        s = a*x+b*y+c*z; //clip co-ordinate axes
        if(x && s/x>0) //x-axis +infinity behind camera
        {
            axes[0].v[0][0]=s/x;
            axes[0].v[0][3]=1;
            axes[0].v[1][0]=-1;
            axes[0].v[1][3]=0;
        }
        else if(x && s/x<0) //x-axis -infinity behind camera
        {
            axes[0].v[0][0]=1;
            axes[0].v[0][3]=0;
            axes[0].v[1][0]=s/x;
            axes[0].v[1][3]=1;
        }
        if(y && s/y>0) //y-axis +infinity behind camera
        {
            axes[1].v[0][1]=s/y;
            axes[1].v[0][3]=1;
            axes[1].v[1][1]=-1;
            axes[1].v[1][3]=0;
        }
        else if(y && s/y<0) //y-axis -infinity behind camera
        {
            axes[1].v[0][1]=1;
            axes[1].v[0][3]=0;
            axes[1].v[1][1]=s/y;
            axes[1].v[1][3]=1;
        }
```

```
                                    if(z && s/z>0) //z-axis +infinity behind camera
                                    {
                                        axes[2].v[0][2]=s/z;
                                        axes[2].v[0][3]=1;
                                        axes[2].v[1][2]=-1;
                                        axes[2].v[1][3]=0;
                                    }
                                    else if(z && s/z<0) //z-axis -infinity behind camera
                                    {
                                        axes[2].v[0][2]=1;
                                        axes[2].v[0][3]=0;
                                        axes[2].v[1][2]=s/z;
                                        axes[2].v[1][3]=1;
                                    }
                                    TRANSFORMATION Ry = {{ {  c, 0, a, 0}, //remove x-component
                                                           {  0, q, 0, 0},
                                                           { -a, 0, c, 0},
                                                           {  0, 0, 0, q}  }};

                                    TRANSFORMATION Rx = {{ { d,  0, 0, 0}, //remove y-component
                                                           { 0,  q, b, 0},
                                                           { 0, -b, q, 0},
                                                           { 0,  0, 0, d}  }};

                                    TRANSFORMATION Pj = {{ { 1, 0, 0, 0}, //project
                                                           { 0, 1, 0, 0},
                                                           { 0, 0, 0, p},
                                                           { 0, 0, 0, 1}  }};
                                    P = multiply(multiply(T1, (a||c)?Ry:I), multiply(Rx, Pj));
                                                          //update projection vector
                                    if(aniFlag) display(T, P); //animate
                                }
                                else printf("LOG [ERROR] : INVALID SYNTAX. "
                      "PLEASE USE WITH NO SPACES : -P/X:Eye_X,Eye_Y,Eye_Z,Ref_X,Ref_Y,Ref_Z\n");
                                break;
                            default:
                                printf("LOG [ERROR] : INVALID OPERATION NOT RECOGINZED : %s\n", argv[i]);
                                break;
                        }
                }
                if(!aniFlag)
                {
                    loadObjects(kpg, P); //load object from file
                    display(I, P); //display original object
                    getchar();
                    display(T, P); //display transformed object
                }
                eraseGraph(objects);
                getchar();
                closegraph();
                return 0;
            }
```

Output

Linear Approximations

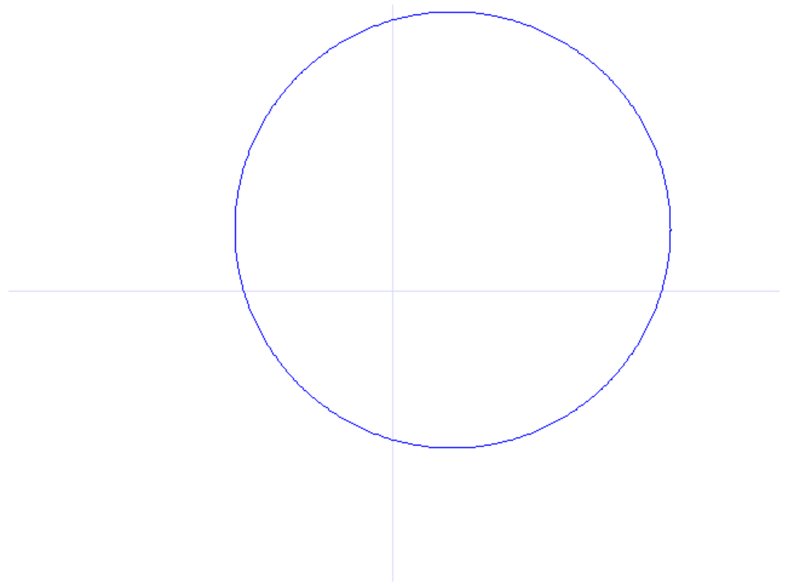
Linear Approximations of the conic sections, by incremental iterative procedures:

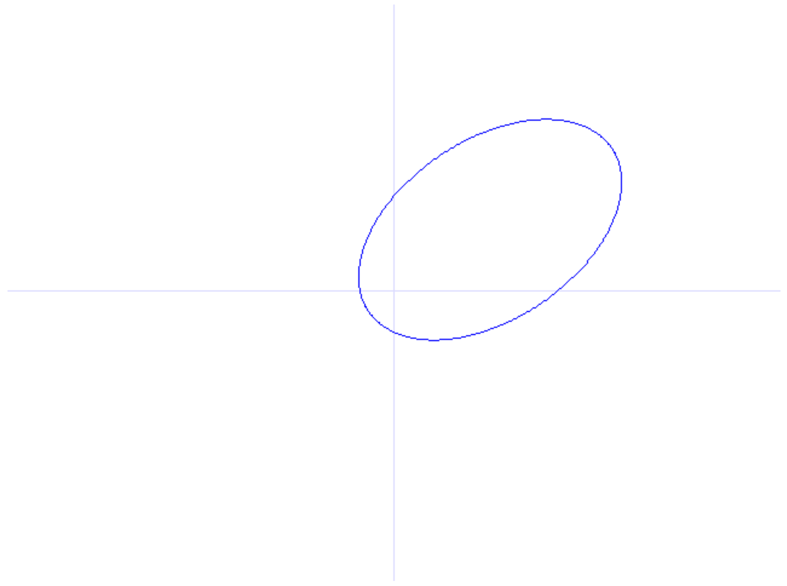
- Straight Lines – Trivial case, no approximation
- Circles – Using the parametric form (R*cos*θ, R*sin*θ). The incremental iterative procedure used is analogous to rotating the current point about the center of the circle by dθ to obtain the next point.
- Ellipses – Using the parametric form (A*cos*θ, B*sin*θ).
- Parabolas - Using the parametric form (at^2, 2at). Limits of parameter *t* are input from the user.
- Hyperbolas - Using the parametric form (A*cosh*θ, B*sinh*θ). Limits of the parameter θ are calculated based on display area size.

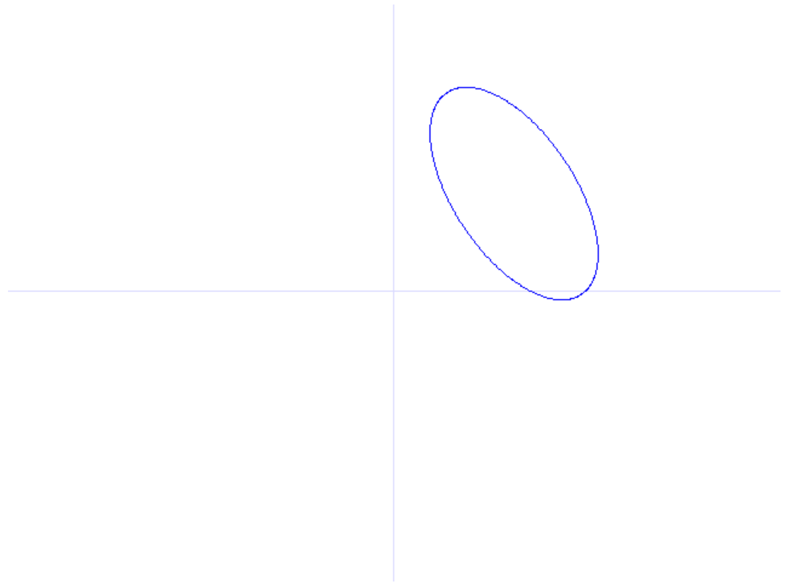
```c
/*                              21.LINEARAPPROX.C                              */
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <graphics.h>
#include <math.h>
#define PI 3.1415926

void la_circle(double *x, double *y, int n)
{
    double cx, cy, r, cosd, sind;
    int i;
    printf("\nEnter center of circle : ");
    scanf("%lf %lf", &cx, &cy);
    printf("\nEnter radius of circle : ");
    scanf("%lf", &r);
    x[0]=r; //starting point
    y[0]=0;
    cosd = cos(2*PI/n); //cos(dt)
    sind = sin(2*PI/n); //sin(dt)
    for(i=1; i<n+1; ++i)
    {
        x[i] = x[i-1]*cosd - y[i-1]*sind; //calculate r*cos(t+dt)
        y[i] = x[i-1]*sind + y[i-1]*cosd; //calculate r*sin(t+dt)
        x[i-1] += cx; //move center
        y[i-1] += cy;
    }
    x[n] += cx;
    y[n] += cy;
}

void la_ellipse(double *x, double *y, int n)
{
    double t, cx,   cy,
               a,    b,
              mx,   my,
            cosd, sind;
    int i;
    printf("\nEnter center of ellipse : ");
    scanf("%lf %lf", &cx, &cy);
    printf("\nEnter semi-major axis and semi-minor axis of ellipse : ");
    scanf("%lf %lf", &a, &b);
    printf("\nEnter direction ratio of major axis : ");
    scanf("%lf %lf", &mx, &my);
    x[0]=a; //starting point
    y[0]=0;
    t = pow(mx*mx+my*my, 0.5);
    mx /= t; //normalize to direction cosines
    my /= t;
    cosd = cos(2*PI/n); //cos(dt)
    sind = sin(2*PI/n); //sin(dt)
    for(i=1; i<n+1; ++i)
    {
        x[i] = x[i-1]*cosd - a*y[i-1]*sind/b; //calculate a*cos(t+dt)
        y[i] = b*x[i-1]*sind/a + y[i-1]*cosd; //calculate b*sin(t+dt)
        t = x[i-1]*mx - y[i-1]*my; //rotate to match axis
        y[i-1] = x[i-1]*my + y[i-1]*mx;
        x[i-1] = t;
        x[i-1] += cx; //move center
        y[i-1] += cy;
    }
    t = x[n]*mx - y[n]*my;
    y[n] = x[n]*my + y[n]*mx;
    x[n] = t;
    x[n] += cx;
    y[n] += cy;
}
```

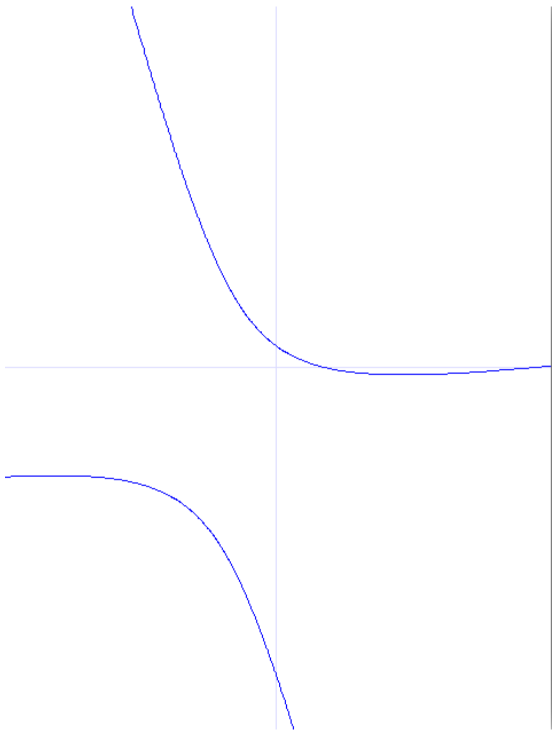
```c
void la_parabola(double *x, double *y, int n)
{
    double t, cx, cy,
              mx, my,
              t1, t2,
               a, dt;
    int i;
    printf("\nEnter vertex of parabola : ");
    scanf("%lf %lf", &cx, &cy);
    printf("\nEnter focal length of parabola : ");
    scanf("%lf", &a);
    printf("\nEnter direction ratio of axis : ");
    scanf("%lf %lf", &mx, &my);
    printf("\nEnter range of parameter t : ");
    scanf("%lf %lf", &t1, &t2);
    x[0]=a*t1*t1; //starting point
    y[0]=2*a*t1;
    t = pow(mx*mx+my*my, 0.5);
    mx /= t; //normalize to direction cosines
    my /= t;
    dt = (t2-t1)/n; //dt
    for(i=1; i<n+1; ++i)
    {
        x[i] = x[i-1] + y[i-1]*dt + a*dt*dt; //calculate a(t+dt)^2
        y[i] = y[i-1] + 2*a*dt; //calculate 2a(t+dt)
        t = x[i-1]*mx - y[i-1]*my; //rotate to match axis
        y[i-1] = x[i-1]*my + y[i-1]*mx;
        x[i-1] = t;
        x[i-1] += cx; //move vertex
        y[i-1] += cy;
    }
    t = x[n]*mx - y[n]*my;
    y[n] = x[n]*my + y[n]*mx;
    x[n] = t;
    x[n] += cx;
    y[n] += cy;
}

int xc, yc, mirrorFlag=0;

void la_hyperbola(double *x, double *y, int n)
{
    double t, cx,   cy,
               a,    b,
              mx,   my,
            cosd, sind;
    int i;
    printf("\nEnter center of hyperbola : ");
    scanf("%lf %lf", &cx, &cy);
    xc = cx + cx;
    yc = cy + cy;
    mirrorFlag = 1; //hyperbola requires 2 curves to be drawn
    printf("\nEnter transverse axis and conjugate axis of hyperbola : ")
    scanf("%lf %lf", &a, &b);
    printf("\nEnter direction ratio of transverse axis : ");
    scanf("%lf %lf", &mx, &my);
    x[0] = a*cosh(acosh(320/a)); //adaptive starting point, to fit screen
    y[0] = b*sinh(acosh(320/a));
    t = pow(mx*mx+my*my, 0.5);
    mx /= t; //normalize to direction cosines
    my /= t;
    cosd = cosh(2*acosh(320/a)/n); //cosh(dt)
    sind = sinh(2*acosh(320/a)/n); //sinh(dt)
    for(i=1; i<n+1; ++i)
    {
        x[i] = x[i-1]*cosd - a*y[i-1]*sind/b; //calculate a*cosh(t+dt)
        y[i] = y[i-1]*cosd - b*x[i-1]*sind/a; //calculate b*sinh(t+dt)
        t = x[i-1]*mx - y[i-1]*my; //rotate to match axis
        y[i-1] = x[i-1]*my + y[i-1]*mx;                          ;
        x[i-1] = t;
        x[i-1] += cx; //move center
        y[i-1] += cy;
    }
    t = x[n]*mx - y[n]*my;
    y[n] = x[n]*my + y[n]*mx;
    x[n] = t;
    x[n] += cx;
    y[n] += cy;
}

int main(int argc, char **argv)
{
    clock_t last=0, now;
    double dely;
    int n, i, ch;
    double  *x, *y;
    int gd=DETECT,gm;
    printf("Linear Approximations:\n1. Circle\n2. Ellipse"
           "\n3. Parabola\n4. Hyperbola\n : ");
    scanf("%d", &ch);
    printf("Number of Segments? "); scanf("%d", &n);
    printf("Delay time (in seconds)? "); scanf("%lf", &dely);
    x = (double*)malloc((n+1)*sizeof(double));
    y = (double*)malloc((n+1)*sizeof(double));
    switch(ch) //generate points
    {
        case 1: la_circle(x, y, n); break;
        case 2: la_ellipse(x, y, n); break;
        case 3: la_parabola(x, y, n); break;
        case 4: la_hyperbola(x, y, n); break;
        default: printf("Invalid!"); return 1; break;
    }
    initgraph(&gd, &gm, NULL); //initialize graphics
    setcolor(WHITE); //draw axes
    bar(0, 0, 640, 480);
    setcolor(LIGHTGRAY);
    line(320, 0, 320, 480);
    line(0, 240, 640, 240);
    setcolor(BLUE);
    for(i=0; i<n; ++i)
    {
        while( ((now=clock()) - last)*1.0/CLOCKS_PER_SEC < dely );
                                    //animate output with delay
        last=now;
        line(320+x[i], 240-y[i], 320+x[i+1], 240-y[i+1]);
        if(mirrorFlag) line(320+xc-x[i], 240-yc+y[i], 320+xc-x[i+1], 240-yc+y[i+1]);
                                 //special case for hyperbola only
    }
    getchar();
    closegraph();
    free(x); //free memory
    free(y);
    return 0;
}
```

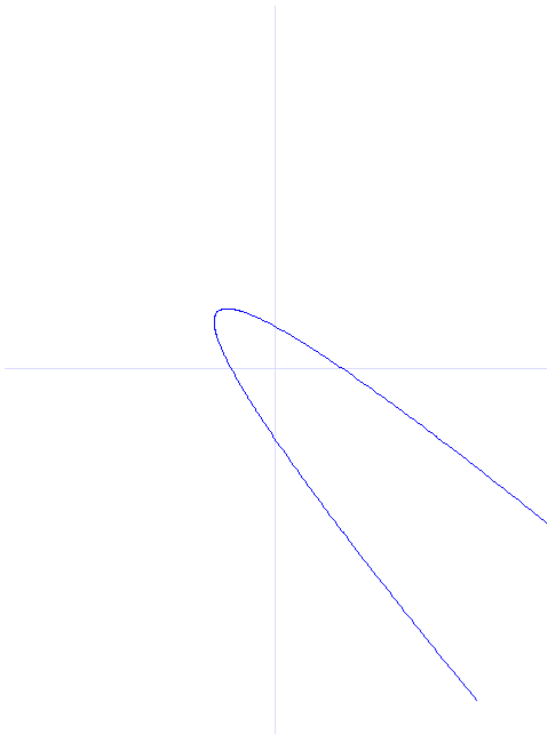
Output

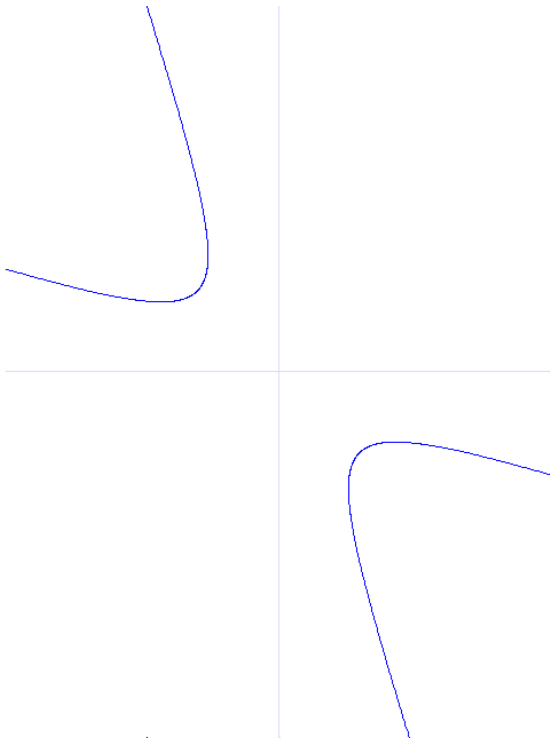












Bezier Curves

The ordered set of control points must be input as a polygon in a file. The algorithm recursively creates a smaller set of control points based on a parameter t that varies from 0 to 1 is small increments. When this set degenerates to a point, it is added to a list of vertices. Finally the vertices are plotted and joined by short line segments to display the Bezier Curve.

Bezier Curves are mathematically defined using the Binomial Expansion of

$$\big(t + (1 - t)\big)^n$$

and the co-ordinates of the control points as weights. For this reason, Bezier curves are readily generalized to 3 dimensions. This recursive implementation intuitively generates the same curve, without explicit calculation of the Binomial co-effiecients.

```
/*                            22.BEZIER.C                                   */
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <graphics.h>
#include "0.kpgparse.c"

VSHAPE reverse(VSHAPE s)
{
    VSHAPE t;
    if(s==NULL || s->next==NULL)
        return s;
    t=reverse(s->next);
    s->next->next=s;
    s->next=NULL;
    return t;
}

VSHAPE bezier(VSHAPE s, double f)
{
    VSHAPE t;
    VSHAPE out=NULL;
    for(t=s; t->next!=NULL; t=t->next)
    {
        VSHAPE v = (VSHAPE)malloc(sizeof(struct vpoint));
        v->x=t->x+(t->next->x-t->x)*f;
        v->y=t->y+(t->next->y-t->y)*f;
        v->next=out;
        out=v;
    }
    if(out->next!=NULL)
    {
        out=reverse(out);
        t=bezier(out, f);
        eraseVector(out);
        return t;
    }
    else return out;
}
```

```c
int main(int argc, char **argv)
{
    clock_t last=0, now;
    VSHAPE s, out=NULL;
    int n, i;
    int gd=DETECT,gm;
    float d;
    FILE *kpg;
    if(argc<2)
    {
        printf("[ ERROR ] Usage : <.kpg>_format_input_file"
               " [number_of_animation_frames] [frame_delay]\n");
        return 1;
    }
    n = (argc>2) ? atoi(argv[2]) :  10;
    d = (argc>3) ? atof(argv[3]) : 0.1;
    kpg=fopen(argv[1], "rx");
    if(!kpg)
    {
        printf("[ ERROR ] Usage : <.kpg>_format_input_file "
               "[number_of_animation_frames] [frame_delay]\n");
        return 1;
    }
    s=toVector(expectPoly(kpg));
    fclose(kpg);
    initgraph(&gd, &gm, NULL); //initialize graphics
    setcolor(WHITE); //draw axes
    bar(0, 0, 640, 480);
    setcolor(LIGHTGRAY);
    line(320, 0, 320, 480);
    line(0, 240, 640, 240);
    setcolor(BLUE);
    VSHAPE cur;
    for(cur=s; cur->next; cur=cur->next) //draw bounding polygon
        line(320+cur->x, 240-cur->y, 320+cur->next->x, 240-cur->next->y);
    for(i=0; i<=n; ++i)
    {
        VSHAPE t=bezier(s, i*1.0/n); //call algorithm with parameter
        t->next=out;
        out=t;
    }
    eraseVector(s);
    setcolor(RED);
    for(cur=out; cur->next; cur=cur->next)
    {
        while( ((now=clock()) - last)*1.0/CLOCKS_PER_SEC < d ); //animate output with delay
        last=now;
        line(320+cur->x, 240-cur->y, 320+cur->next->x, 240-cur->next->y);
    }
    eraseVector(out);
    getchar();
    closegraph();
    return 0;
}
```

Output