Filing Algorithms

Filling closed shapes means coloring the interior area of the shape. This requires algorithms to compute all the interior shape points using the shape coordinate parameters. As an example, in filling a triangle, we start with its three vertex coordinates. We may compute its edge points by linear interpolation of its vertex point pairs. We may then draw horizontal lines between pairs of edge points having the same y-coordinates or draw vertical lines between pairs of boundary points with the same x-coordinates. Color interpolation is also possible. In fact, filling with one filling color is considered 'constant' color interpolation or flat shading. Linear color interpolation is called 'Gouraud' shading. Mapping of texture images to the interior area of the closed shape is also possible by transforming the coordinates of texture points to the interior points of the shape.

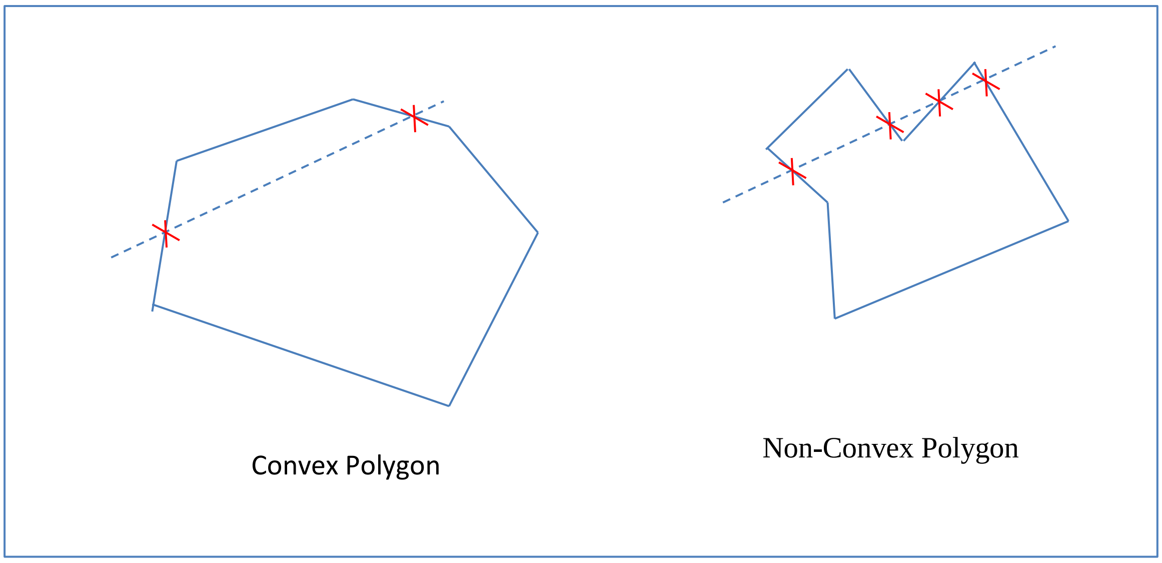
Polygon-Filling algorithms

First, we will focus on algorithms that fill polygons with constant 'filling' colors (flat shading). The algorithms are given a set of polygon vertices $P_i, i = 0, 1, \dots., n-1$ ordered in some direction (clockwise or anticlockwise) and a 'filling' color. We'll study two algorithms, one for convex polygon filling and the other for general polygon filling. Both algorithms draw horizontal scan lines between pairs of points having the same y-coordinates. The two algorithms are thus two different realization of the scan fill algorithm. In this algorithm, the horizontal scan line intersections with polygon edges are computed incrementally using a DDA approach. An edge with endpoints (x_1, y_1) and (x_2, y_2) where $y_1 < y_2$ intersects with the horizontal scan-lines $y = y_1, y_1 + 1, y_1 + 2, \dots, y_2$ at points $\left(x_1 + \frac{k}{m}, y_1 + k\right)$ where m is the slope of the line and k runs from 0 to $y_2 - y_1$. As you see, y is increased by 1 and x is increased by 1/m. This is exactly the simple DDA algorithm in which y is the independent variable. To generate these point:

```
- Start with $(x, y) = (x_1, y_1)$
- Loop while y<$y_1$
    o y=y+1
    o x=x+1/m
    o So the generated point is (x, y)
  End loop
```

Convex polygon fill

A convex polygon is a polygon that has at most two intersection points with any line. The following figure shows examples of convex and non-convex polygons.



Since, scan-lines intersect with the convex polygon in at most two points, the algorithm uses an array with an entry for each scan line. The entry contains two fields to store the x coordinates of the intersection at the scan line. One field (call it:x_{min}) contains the smaller x and the other (call it:x_{max}) contains the larger x.

The algorithm starts by setting x_{min} to very large value and x_{max} to a very small value. The algorithm then applies the DDA for each polygon edge. Every generated point (x, y) is represented in the array as an entry at index y with $x_{min} = x$ if x is smaller than the current x_{min} and $x_{max} = x$ if x is larger than current x_{max}. The algorithm finally traverses all the entries of the array. If $x_{min} < x_{max}$ at some entry with index y, the algorithm draws a horizontal line from (x_{min}, y) to (x_{max}, y).

Implementation of the algorithm

First, start with defining the constants and data structures. Note that the constant `MAXENTRIES represents the maximum number of scan lines which equals the height of the screen viewport.`

```
#define MAXENTRIES 600;
struct Entry
{
        int xmin,xmax;
};
```

Second, define a utility to initialize the entry table (array)

```
void InitEntries(Entry table[])
{
        for(int i=0;i<MAXENTRIES;i++)
        {
                        table[i].xmin=MAXINT;
                        table[i].xmax=-MAXINT;
        }
}
```

Third, define the scan edge utility that takes the end points of the edge, generates all points of intersection with scan lines and update the entry table as discussed before

```
Void ScanEdge(POINT v1,POINT v2,Entry table[])
{
	if(v1.y==v2.y)return;
	if(v1.y>v2.y)swap(v1,v2);
	double minv=(double)(v2.x-v1.x)/(v2.y-v1.y);
	double x=v1.x;
	int y=v1.y;
	while(y<v2.y)
	{
		if(x<table[y].xmin)table[y].xmin=(int)ceil(x);
		if(x>table[y].xmax)table[y].xmax=(int)floor(x);
		y++;
		x+=minv;
	}
}
```

Fourth, define a utility to draw scan lines stored in the entry table.

```
void DrawSanLines(HDC hdc,Entry table[],COLORREF color)
{
	for(int y=0;y<MAXENTRIES;y++)
		if(table[y].xmin<table[y].xmax)
			for(int x=table[y].xmin;x<=table[y].xmax;x++)
				SetPixel(hdc,x,y,color);
}
```

Last, write the main function of the algorithm that contains the overall logic of the algorithm.

```
Void ConvexFill(HDC hdc,POINT p[],int n,COLORREF color)
{
	Entry *table=new Entry[MAXENTRIES];
	InitEntries(table);
	POINT v1=p[n-1];
	for(int i=0;i<n;i++)
	{
		POINT v2=p[i];
		ScanEdge(v1,v2,table);
		v1=p[i];
	}
	DrawSanLines(hdc,table,color);
	delete table;
}
```

Implementation of Gouraud shading for convex polygons

The following modifications can be done to the above algorithm to realize Gouraud shading:

1. A color structure is used with color components red, green and blue taking scaled color values from 0 to 1. A set of utilities is defined on the color structure to permit addition and subtraction of two colors and multiplication of colors by constants. These utilities are used by the color interpolation operations.

```
struct GColor
{
	double red,green,blue;
	GColor(double r=0, double g=0,double b=0)
	{
		red=r; green=g; blue=b;
	}
	GColor(COLORREF c)
	{
		red=GetRValue(c)/255.0;
```

```
                green=GetGValue(c)/255.0;
                blue=GetBValue(c)/255.0;
        }
        COLORREF ToColor()
        {
                int r=(int)(255*red+0.5);
                if(r>255)r=255;
                if(r<0)r=0;
                int g=(int)(255*green+0.5);
                if(g>255)g=255;
                if(g<0)g=0;
                int b=(int)(255*blue+0.5);
                if(b>255)b=255;
                if(b<0)b=0;
                return RGB(r,g,b);
        }
};
GColor operator-(GColor& c1,GColor& c2)
{
        return GColor(c1.red-c2.red,c1.green-c2.green,c1.blue-c2.blue);
}
GColor operator*(GColor& c1,double a)
{
        return GColor(c1.red*a,c1.green*a,c1.blue*a);
}
GColor operator+=(GColor& c1,GColor& c2)
{
        return c1=GColor(c1.red+c2.red,c1.green+c2.green,c1.blue+c2.blue);
}
```

2. Every vertex of the polygon has a color field in addition to its coordinates (x, y). The Vertex structure is defined as follows:

```
struct Vertex
{
        int x,y;
        GColor color;
        Vertex(int x=0,int y=0,COLORREF c=0):x(x),y(y),color(c)
        {
        }
};
```

3. The entries of the table (array) is used to store the colors ***(c_{min}, c_{max})*** of the two end points ***(x_{min}, y)*** , ***(y_{min}, y)*** of the scan line y. The definition of the structure is as follows:

```
struct GEntry
{
        int xmin,xmax;
        GColor cmin,cmax;
};
```

4. Edge pixel colors are computed for a polygon edge $v_1 - v_2$ as the interpolation between $v_1.color$ and $v_2.color$ with respect to y in the ScanEdge function. The incremental change in color is $(v_2.color - v_1.color)/\ (v_2.y - v_1.y)$ with initial value $v_1.color$. Following is the modified ScanEdge function:

```
void ScanEdge(Vertex v1,Vertex v2,GEntry table[])
{
        if(v1.y==v2.y)return;
        if(v1.y>v2.y)swap(v1,v2);
        double idy=1.0/(v2.y-v1.y);
        double minv=(v2.x-v1.x)*idy;
        GColor cslope=(v2.color-v1.color)*idy;
        double x=v1.x;
        int y=v1.y;
        GColor color=v1.color;
        while(y<v2.y)
        {
                        if(x<table[y].xmin)
                        {
                                table[y].xmin=(int)ceil(x);
                                table[y].cmin=color;
                        }
                        if(x>table[y].xmax)
                        {
                                table[y].xmax=(int)floor(x);
                                table[y].cmax=color;
                        }
                        y++;
                        x+=minv;
                        color+=cslope;
                }
  }
```

5. In the DrawScanLines function, color is interpolated between c_{min} and c_{max} of the array entry of each scan line with respect to x. The incremental change in color is $(c_{max} - c_{min})/(x_{max} - x_{min})$ with initial value c_{min}. The modified function is as follows:

```
void DrawSanLines(HDC hdc,GEntry table[])
{
        for(int y=0;y<MAXENTRIES;y++)
                if(table[y].xmin<table[y].xmax)
                {
                        double idx=1.0/(table[y].xmax-table[y].xmin);
                        GColor color=table[y].cmin;
                        GColor cslope=(table[y].cmax-table[y].cmin)*idx;
                        for(int x=table[y].xmin;x<=table[y].xmax;x++)
                        {
                                SetPixel(hdc,x,y,color.ToColor());
                                color+=cslope;
                        }
                }
}
```

6. The main function of the algorithm is as follows:

```
void ConvexShade(HDC hdc,Vertex p[],int n)
{
        GEntry *table=new GEntry[MAXENTRIES];
        InitEntries(table);
        Vertex v1=p[n-1];
        for(int i=0;i<n;i++)
        {
                Vertex v2=p[i];
                ScanEdge(v1,v2,table);
                v1=p[i];
        }
        DrawSanLines(hdc,table);
}
```

General Polygon Filling

The previous algorithm can only be used for convex or horizontally convex polygons. For general polygon filling, every edge is represented by a record called 'Edge record'. For an edge $v_1 - v_2$ with $v_1.y < v_2.y$, the edge record is stored in the edge table (array) at position: $v_1.y$ and contains the following information about the edge:

x: initially contains the x-coordinate of the vertex v_1 (i.e. $v_1.x$)

m_{inv}: the inverse of the slope of the edge ($m_{inv} = \frac{1}{m} = \frac{v_2.x - v_1.x}{v_2.y - v_1.y}$)

y_{max}: contains the y-coordinate of vertex v_2(i.e. $v_2.y$)

Clearly the edge record data together with the index of its position in the edge table contains full information about the edge (start and end points and slope inverse). Note that the field x takes as an initial value $v_1.x$ and will be incremented by m_{inv} through the algorithm when moving from one scan line to another (i.e. increasing y) as known in the simple DDA algorithm.

Each entry in the edge table is a list of edge records having the same $v_1.y$ and maintains an ascending sorting order with x. The algorithm thus has the following data structures:

```
#include <list>
using namespace std;
#define MAXENTRIES 600
struct EdgeRec
{
        double x;
        double minv;
        int ymax;
        bool operator<(EdgeRec r)
        {
                return x<r.x;
        }
};
typedef list<EdgeRec> EdgeList;
```

The operator overloading of the '<' operator is used by the sorting function as will be shown to sort edge records in ascending order with x.

The algorithm has a utility to construct an edge record for a given edge's end points as shown below. Note that this utility has a side effect of swapping the points when `v1.y>v2.y`

```
EdgeRec InitEdgeRec(POINT& v1,POINT& v2)
{
        if(v1.y>v2.y)swap(v1,v2);
        EdgeRec rec;
        rec.x=v1.x;
        rec.ymax=v2.y;
        rec.minv=(double)(v2.x-v1.x)/(v2.y-v1.y);
        return rec;
}
```

The 'InitEdgeTable' is used to initialize the edge table (an array of edge lists) from the edges of the polygon by calling `InitEdgeRec` for each edge as shown below.

```
void InitEdgeTable(POINT *polygon,int n,EdgeList table[])
{
POINT v1=polygon[n-1];
for(int i=0;i<n;i++)
{
        POINT v2=polygon[i];
        if(v1.y==v2.y){v1=v2;continue;}
        EdgeRec rec=InitEdgeRec(v1, v2);
        table[v1.y].push_back(rec);
        v1=polygon[i];
}
}
```

The main algorithm does the following:

```
1.  Initialize the edge list array 'table' with edge information
2.  Start with y = the first index in table with non-empty edge list entry
3.  Let ActiveList=table[y]
4.  While ActiveList is not empty
    4.1.  Sort ActiveList nodes in an ascending order with x
    4.2.  Draw horizontal lines between points represented by successive pairs of nodes in ActiveList
    4.3.  Increment y by 1 (to go to a new scan line)
    4.4.  Delete from ActiveList those nodes with $y_{max}$=y
    4.5.  Append to ActiveList the new nodes at table[y] (if any)
    End while
```

The following is the Visual C++ implementation of the main function of the algorithm:

```
void GeneralPolygonFill(HDC hdc,POINT *polygon,int n,COLORREF c)
{
        EdgeList *table=new EdgeList [MAXENTRIES];
        InitEdgeTable(polygon,n,table);
        int y=0;
        while(y<MAXENTRIES && table[y].size()==0)y++;
        if(y==MAXENTRIES)return;
        EdgeList ActiveList=table[y];
        while (ActiveList.size()>0)
        {
                ActiveList.sort();
                for(EdgeList::iterator it=ActiveList.begin();it!=ActiveList.end();it++)
                {
                        int x1=(int)ceil(it->x);
                        it++;
                        int x2=(int)floor(it->x);
                        for(int x=x1;x<=x2;x++)SetPixel(hdc,x,y,c);
                }
                y++;
                EdgeList::iterator it=ActiveList.begin();
                while(it!=ActiveList.end())
                        if(y==it->ymax) it=ActiveList.erase(it); else it++;
                for(EdgeList::iterator it=ActiveList.begin();it!=ActiveList.end();it++)
                        it->x+=it->minv;
                ActiveList.insert(ActiveList.end(),table[y].begin(),table[y].end());
        }
        delete[] table;
}
```

As done in convex polygon filling, we can extend the algorithm to implement Gouraud shading.

Flood Fill Algorithm

Flood Fill is a recursive algorithm used to fill a closed shape with a given boundary color C_b with some filling color C_f. The algorithm starts filling from some interior point (x, y) supplied by the user as a parameter and floods in the four (or eight) directions of the neighbors of (x, y) by recursion. An outline of the algorithm is given below:

```
Algorithm FloodFill( x, y, cb, cf)
Let c=GetPixelColorAt(x,y)
If c==cb or c==cf then
 Return
End if
DrawPixel(x, y)
FloodFill(x+1,y,cb,cf)
FloodFill(x-1,y,cb,cf)
FloodFill(x,y+1,cb,cf)
FloodFill(x,y-1,cb,cf)
End Algorithm
```

A Visual C++ implementation is given below:

```
void FloodFill(HDC hdc,int x,int y,COLORREF Cb,COLORREF Cf)
{
        COLORREF C=GetPixel(hdc,x,y);
        if(C==Cb || C==Cf)return;
        SetPixel(hdc,x,y,cf);
        FloodFill(hdc,x+1,y,Cb,Cf);
        FloodFill(hdc,x-1,y,Cb,Cf);
        FloodFill(hdc,x,y+1,Cb,Cf);
        FloodFill(hdc,x,y-1,Cb,Cf);
}
```

Note that the algorithm needs a very large stack size because the program stores the next instruction address and the function parameters of function calls in the stack. Recursion has very large number of recursive calls (exponential stack space complexity) and hence the stack may overflow. One solution of this problem is to increase the stack size from the program options. Alternatively, one might rewrite the recursive algorithm using dynamic stack or queue data structures as follows:

```
#include <stack>
using namespace std;
struct Vertex
{
        int x,y;
        Vertex(int x,int y):x(x),y(y)
        {
        }
};
struct Vertex
{
        int x,y;
        Vertex(int x,int y):x(x),y(y)
        {
        }
};
void NRFloodFill(HDC hdc,int x,int y,COLORREF Cb,COLORREF Cf)
{
        stack<Vertex> S;
        S.push(Vertex(x,y));
        while(!S.empty())
        {
                Vertex v=S.top();
                S.pop();
                COLORREF c=GetPixel(v.x,v.y);
                if(c==Cb || c==Cf)continue;
                SetPixel(hdc,v.x,v.y,Cf);
                S.push(Vertex(v.x+1,v.y));
                S.push(Vertex(v.x-1,v.y));
                S.push(Vertex(v.x,v.y+1));
                S.push(Vertex(v.x,v.y-1));
        }
}
```