Abstract

A graphics metafile typically contains a series of graphic vector outline commands that allow a picture or design to be produced by executing those commands within the context of a graphics display. Unlike other graphics file formats such as bitmaps, metafiles are often more flexible in that they may be more easily scaled or manipulated since they are not tied to a discrete pixel resolution. Most often, a system will execute the specified vector commands scaled to a desired size within a raster-scan or pixel-based device context, where, for example, pixels that lie within vector outlines are set with specified color information. When a subset of vector commands overlap or otherwise intersect with previously drawn or executed commands, the pixels within the overlapped areas are simply reset to the color specified by the more recent vector commands. Thus, potential redundancies within a metafile (i.e. situations where multiple commands repeatedly “paint” within the same area) are resolved through this process of rasterization where more recent commands always take precedence over those that were previously executed.

However, for many applications, the loss of flexibility that results from rasterization (e.g. loss of detailed outline information, etc.) makes it unsuitable for developing a more usable composite representation of a metafile’s vector commands. Specifically, it may be desirable to eliminate redundancies within vector outlines by actually modifying the underlying outlines directly such that painting within any given area never occurs more than once (i.e. no overlapping occurs). The resultant composite representation may be beneficial in many respects. For example, greater compression of picture information may perhaps be achieved since many redundant or overlapping commands would have

been eliminated or at least reduced in size. Also, the resultant information may be used for other applications such as computerized embroidery imprinting where it is often undesirable to repeatedly sew or place stitches within of single area of fabric.

Thus, metafile compositing, in the manner described above, is quite similar to the problem of map overlay studied within the field of computational geometry. Solutions to this problem involve detecting and subsequently processing the intersections and unions of polygonal objects that are placed within a two-dimensional space (e.g. outlines of highways, rivers, lakes, etc.). More specifically, map overlay algorithms are used within many Geographic Informa tion Systems (GISs) which process map layers to organize geographic features. Each layer describes a certain aspect of the modeled real world (e.g. elevations, highway information, etc.). Thus, if each vector command within a graphics metafile is considered as a layer in a geometric map, any existent overlapping or redundancy problems may be detected and eliminated using similar methods.

One of the fundamental problems addressed within this work is that of implementing theoretically correct algorithms within a computer system that is not inherently capable of exact computation. Specifically, inexact floating point number representation that is commonly used during mathematical computation within a microprocessor can easily cause such algorithms to fail or be completely unreliable. As widely discussed within the academic literature, this has been a major hurdle to developing practical implementations for many algorithms within the field of computational geometry. Furthermore, moving to a model of exact computation (using additional software libraries, etc.), can cause a resulting implementation's speed performance to degrade to the point of being impractical for most engineering applications. Thus, recently published results dealing with such issues were drawn upon extensively to ultimately form the metafile compositing algorithm presented here.

Chapter I
Introduction

1.1 Motivation of Windows Metafile Compositing

A computer graphics file format is the way in which an image file is saved. There are many kinds of graphic file formats which can usually be identified by their three letter extensions at the end of the file name. Some graphic format example files are illustrated in Appendix I. This paper primarily focuses on Windows metafiles (this includes enhanced metafiles). Of course, the methodologies that are used here could easily be extended to other kinds of graphics metafiles, such as Adobe Postscript or Portable Document Format (pdf) files and others.

This work was supported by and produced in conjunction with Soft Sight, Inc., a company specializing in software for machine vision and image processing applications. The objective of our Windows Metafile (WMF) project for Soft Sight Inc. was to retrieve the Windows Metafile command records and translate them into a set of closed contours that delineate the contiguous regions that would be painted by the commands.

The initial strategy was to use a Brute Force algorithm for compositing. The Brute Force algorithm tests every line segment from one edge contour or polygon with every line segment from the other polygons for intersection. Assuming that both map layers contain n line segments, then this process has $O(n^2)$ time complexity for just 2 layers, for m layers the complexity would be $O(mn^2)$. This is prohibitively expensive in terms of the computation time required. Therefore, map overlay algorithms and sweep-line algorithms were examined and adapted to perform the compositing. These algorithms have time complexity $O(n\mathrm{Log}_2 n + K)$, where n is the number of segments, and K is the number of intersecting points.

Unfortunately, the map overlay and line sweep intersection algorithms have traditionally adopted the exact arithmetic model which is based on real numbers. This model has been extremely productive in terms of algorithmic research. However, all computer calculations have finite precision. To overcome computational limitations, we devised an algorithm that allows a computation difference epsilon while not affecting the logical results. In finding segments intersection, our algorithms also have time complexity $O(n\mathrm{Log}_2 n+K)$, but the results are not affected by the computational epsilon; this is a practical engineering approach for graphical processing.

1.2 Significance of Windows metafiles Compositing Algorithms

A. Embroidery Design Automation

Embroidery is one of the oldest and most elegant of the textile arts. It involves weaving colored thread with a needle onto a fabric background. Embroidered images are typically translated manually from every kind of graphic arts, including computer graphic files. Unfortunately, numerous graphic files are not able to be used directly for computerized embroidery due to the fact that sewing machine stitches on a garment differ from the way that GDI functions draw on the screen. One of many necessary preprocessing steps is to remove redundant records and eliminate the overlapped portions of the records; our algorithms make the translation from computer graphic files to embroidery image designs more easily feasible. Therefore, an endless variety of decoration and patterning is possible by graphic metafile compositing.

B. Other Applications

The Windows metafile compositing algorithm may also be used to improve plotting quality. If a plotting device plots each record, some records may be immediately overwritten or partially written by the next records. This not only wastes the material but also reduces the quality of the plotting result.

As for the graphic compression concern, reducing the quantity of the graphic file records and the size of the records reduces the memory size of the graphic file without losing the quality of the image. Therefore, the Windows metafile compositing algorithm is an alternative optimized method for graphic file compression.

Chapter II
Windows Metafile Compositing

2.1 What is a Windows Metafile?

Microsoft Windows Metafiles store both vector and bitmap-format graphical data. Vector graphic data is made of lines and curves defined by mathematical objects called vectors, which describe graphics according to their geometric characteristics. Vector graphic files are often more resolution-independent and may be more easily scaled or resized since actual outline information is stored rather than raw pixels, which are used by bitmap files. If a bitmap file is displayed at twice the size, we don't get twice the resolution. The following example (Figure 2-1) shows their differences:

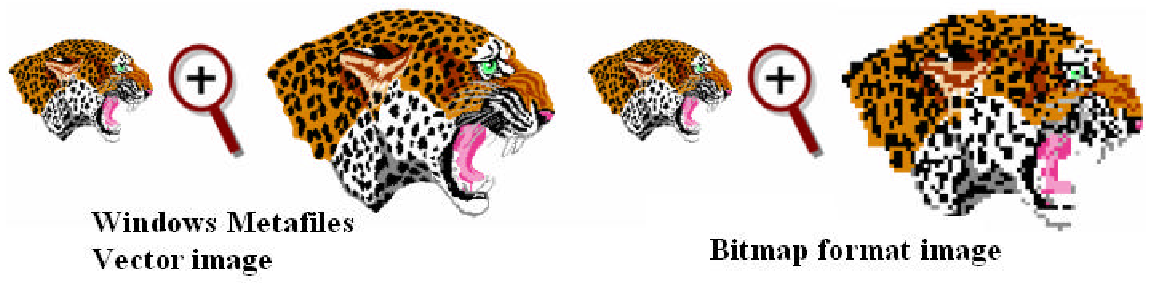


Figure 2-1 Different Format Image Resize Results

The mapping mode of a metafile can be altered during playback. Thus, the image can be scaled arbitrarily, with every component scaling separately, which minimizes the loss of information for the image as a whole. This is not characteristic of bitmaps. In addition, if the image is sparse, a metafile may use less memory than does a bitmap of the same image.

2.2 Compositing of Windows Metafiles

Internally, a Windows metafile is an array of variable-length structures called metafile records. The first records in the metafile specify general information such as the resolution of the device on which the picture was created, the dimensions of the picture, and so on. The remaining records, which constitute the bulk of any metafile, correspond to the graphics device interface (GDI) functions required to draw the picture. These records are stored in the metafile after a special metafile device context (DC) is created. This metafile device context is then used for all drawing operations required to create the picture. When the system processes a GDI function associated with a metafile DC, it converts the function into the appropriate data and stores this data into a record appended to the metafile. There are four kinds of Windows metafiles: standard, clipboard, placeable, and enhanced. Each metafile is comprised of an array of variable-length records that store GDI functions and/or information header(s).

2.2.1 Standard Metafiles

A standard Windows metafile file (WMF) has an extension of ".wmf". A standard WMF file contains a header followed by metafiles records. A WMF's header contains the description of the record data stored in the metafile. Each record represents a Microsoft Windows Graphics Device Interface (GDI) function call. The last record in the file contains information indicating that the end of the record data has been reached. See Appendix II for standard Windows metafile header structure.

2.2.2 Clipboard Metafiles

A clipboard metafile has an extension of ".clp" or ".wri". A clipboard metafile has an extra pre-pended header followed by an 8- or 16-byte header that allows the position of the metafile on the clipboard viewer to be specified. If the clipboard metafile is created using a 16-bit version of Windows, its header contains 2-byte fields; if it is created under a 32-bit Windows environment, its header contains the same fields as the

Win16 WMF header, but the fields are 32 bytes in length. See Appendix II for the header structure.

2.2.3 Placeable Metafiles

Placeable metafiles were created by the Aldus Corporation. A placeable metafile has an ".apm" as its extension. A placeable metafile has an 18-byte header pre-pended. This pre-header contains information used to describe the position of the metafile drawing on the page.

Placeable metafiles are not directly supported by the Windows API. To use these metafiles, the header must be removed from the metafile. In our project, a placeable metafile is identified by its magic number in the header, which is always *9AC6CDD7h*. Some metafiles even have different extensions (i.e. their extensions are not *.amp), but as long as they have the magic number 9AC6CDD7h in their headers, they are interpreted as placeable metafiles, and the 22 bytes of the header are removed before going into further processing. Appendix II lists the placeable metafile structure and Appendix III is the C++ code for how to convert the placeable metafile record into the enhanced metafile record.

2.2.4 Enhanced Metafiles

An enhanced metafile (EMF) has an ".emf" as its file extension. An enhanced metafile has the same basic format of a WMF file: a header followed by one or more records of drawing objects stored as GDI commands. Unlike a WMF, the header is also stored in a record, which appears as the first record in each EMF file. EMF adds the features of a file description string, a programmable color palette to the metafile format and also supports newer more robust GDI commands. EMF files have a header that is 80 bytes in length and contains the same features as found in placeable and clipboard metafiles. Similar to the placeable metafile which has a magic number in its header, an EMF has a signature ID which is always 0x464D4520 in its header. Standard, clipboard, and placeable metafiles can be converted into enhanced format files. Enhanced format files can be converted into Windows standard format metafiles, but some features which are not support by Windows standard format metafiles will be lost.

2.3 Retrieval of Windows Metafiles

As we mentioned above, metafiles can be converted from each other. If an opened metafile is not an enhanced metafile, it needs to be converted into an Enhanced Metafile first by calling the function SetWinMetaFileBits() before being further processed; if it is an enhanced metafile or after converting into enhanced metafile, the GetEnhMetaFile() function is called to create a handle that identifies the enhanced-format metafile stored in the specified file. Finally, the handle is passed into another c++ class to decompose the record data.

Chapter III
Map Overlay and its problems in Graphic Metafiles Compositing

3.1 What is a map overlay?

The input of a map overlay operation consists of two or more topologically structured layers. The output is a new layer in which the new areas are given attributes that are based on the input layers:

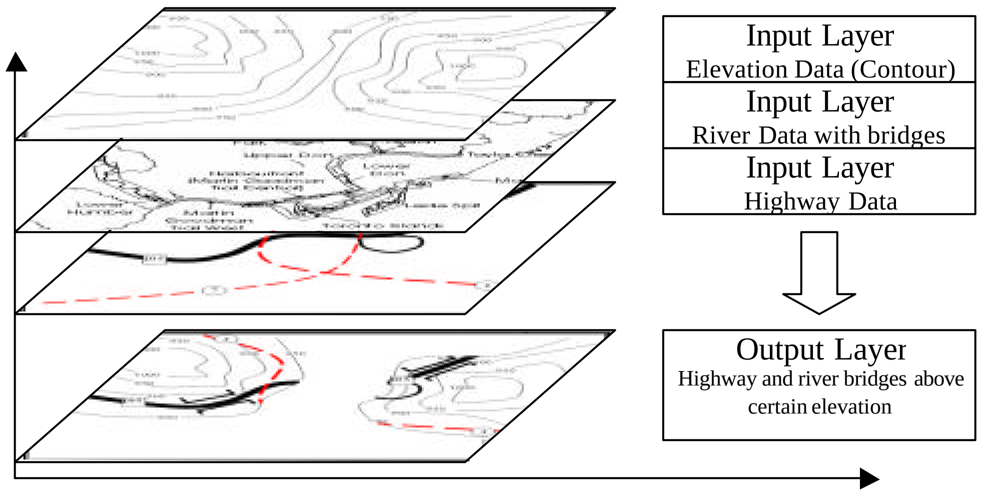


Figure 3-1 Map-Overlay
The upper layers are the input layers which contain different geometric information. The bottom layer is the output layer which contains only the area and data of interest to the user.

The map overlay is usually computed in three logical phases[Fr87]. The first step is performed at the metric level and computes all *intersections* between the edges (line segments) from the different layers. This is followed by a reconstruction of the *topology* and assignment of labels or *attribute* values in the next two steps. Here are the four alternatives for the first step of the map overlay process:

?? brute force algorithm
?? uniform grid algorithm
?? z-order-based algorithm
?? plane-sweep algorithm

The brute force algorithm tests every line segment from one layer with every line segment from the other layer for intersection. Assuming that both map layers contain O(n) line segments, then this process takes $O(n^2)$, which is too computationally expensive if n is a big number.

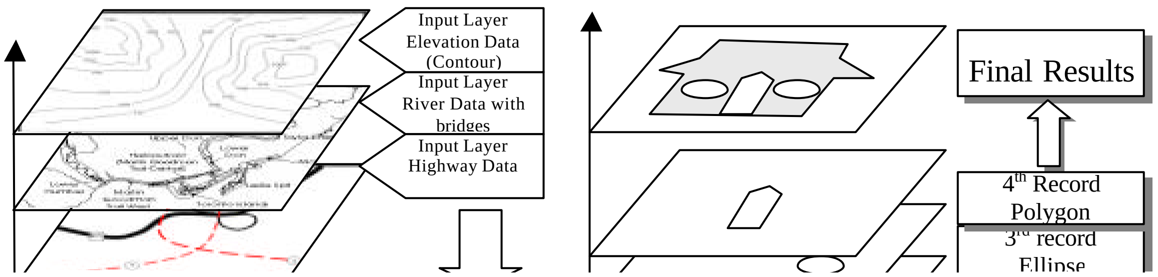
Franklin [FNS⁺89] suggested using a uniform grid to organize the lines: the *uniform grid* method. Line intersections only have to be performed for lines located in cells at the same location. The method works optimally when the lines are distributed in a uniform manner. If the distribution is not uniform, which is very common for geographic data as well as Windows metafile records, the performance degrades because one cell contains a lot of lines which all have to be tested for intersections with a local quadratic time brute force algorithm.

Orenstein [Or91] describes a map overlay algorithm for k-dimensional (k not necessarily 2) situations based on the *z-order*. The z-order is a stacking order that determine which layer appears on top of another for overlapping layers. It results from bitwise interleaving coordinates of a k-dimensional point. All objects are first approximated by boxes (which may be of different sizes: larger boxes are described by fewer bits) according to a space filling z-curve. These boxes are inserted into a one dimensional index structure sorted on the z-value. Map overlay is similar to merging sorted linear lists.

Bentley and Ottmann give one of the first descriptions of the plane-sweep algorithm [BO79] for reporting the intersections of line-segments. The idea for sweep is simple. Sweep the plane by a vertical line (the sweep line) from left to right. The sweep line halts at the segment end-points or intersection points. If there is a new segment intersecting the sweep line, this segment must be tested for intersection against its two neighbors along the sweep line. The segments that are behind the sweep line are already computed. Bentley and Ottmann algorithm [BO79] solved the segment intersection problem by sweep in time O(N*longN + S*longN), where N is the number of segments and S is the number of pairwise intersections. However, Boissonnat and Preparata say in a later paper that Bentley and Ottmann algorithm has been reported as being very sensitive to numerical errors [BP00]. We also found the same problem with Bentley and Ottmann algorithm[BO79] in out project. I Chapter IV we present a modification to Bentley and Ottmann's sweep line algorithm [BO79], which is more reliable and practical in engineering application.

3.2 How is Windows Metafile Compositing Similar to Map Overlay?

An overlay operation takes two or more data layers as input and results in an output data layer. A metafile contains many records and the final layout is one layer. Figure 3-2 indicates their similarities.



In both map overlay and metafile compositing we use Boolean operations to construct the new data layer. Figure 3-3 shows the basic Boolean operations between any two graphics.

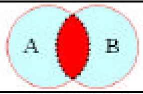
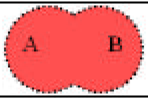
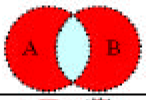
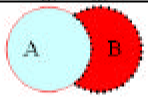
Operation	Logical Notation	Example
AND	A^B	
OR	A v B	
XOR	A⊕B	
NOT	+ A	

Figure 3-3 Basic Graphic Boolean Operation
This Figure shows the basic rule for two graphic Boolean operations. The darkly shaded parts are the results after operation.

3.3 How are They Different?

A) The order of Windows Metafile compositing is important. For the same metafile records, if we change their input orders, the output will be different. However, for map overlay, the input order is not vital. Therefore, when applying map overlay algorithms for Windows Metafile compositing, we must be aware of the time sequential features of the metafile records.

B) In map overlay algorithms, different layers have different attributes. However, in Windows metafile compositing, different records may have identical attributes, for example, the same color. Therefore, in certain situations, merging operations must be performed for the same attribute layers when constructing a new metafile layer.

C) In map overlay one region may receive attributes from many layers; in compositing metafiles, any given region only receives attributes from a single record.

3. 4 Problems in Applying Map Overlay

3. 4 .1 Computational Precision Problem

The precision problem comes from the limited space available to store the data; some data requires unlimited space. A floating-point data type is an approximate numeric value as defined by ANSI standards and the IEEE 754 standard. It stores slightly imprecise representations of real numbers as binary fractions at the hardware level. The accuracy of float data types is limited by the number of bits used to represent the mantissa and the number of bits used to represent the exponent. The mantissa of the floating-point value is a fractional value, in many cases, impossible to represent adequately with a binary number. For values that are not exact multiples of 2^n, precision can be lost. Because of the limitation of the bits used in both mantissa and exponent, the following errors happen:

1. **Representation Errors**

Some rational number, such as 1/3˜ 0.3333…, and some irrational numbers, such as p ˜ 3.1415…can't be represented exactly in a computer with limited storage. A finite decimal representation number might become an infinite repeating representation in binary such as 0.1_{10} ˜$1.1001100110011\ldots \times 2^{-4}{}_{2}$. These representation errors generate computational errors.

2. Normalization Errors

In general, a floating point number will be represented as ±d.dd…d × $ß^{e}$, where d.dd…d is called the significand and has *p* digits. Two other parameters associated with floating-point representations are the largest and smallest allowable exponents, e_{max} and e_{min} [Go91]. Consider normalized floating-point numbers with ß = 10, p=3, and e_{min}=-98. The numbers x=6.87 × $10^{-97}$and y=6.81 × 10^{-97} appear to be perfectly ordinary floating-point numbers. However x-y = 0 even though x?y. The reason is that x-y =0.6 × 10^{-97} =6.0 × 10^{-99} is too small to be represented as a normalized number and so must be flushed to zero[Go91].

3. **Different Applications Result in Different Environment**

Differences in specifications for floating-point representation may cause a loss of precision or magnitude/range when data is moved from one platform to another or when data is converted from one format to another. For example, conversion of ASCII representations of float into IEEE format is inaccurate in the last part of the number.

Because of the computational precision problem, a standard algorithm may not always work properly in practice if its design is based on pure mathematical equations. An example is illustrated in Figure 3-4. Point P is the intersection point of Line [AB] and

Line [CD], which is found based on mathematical equations. Due to the computer floating-point rounding errors, the coordinate values stored in the computer memory may be different from the mathematically calculated correct numbers.

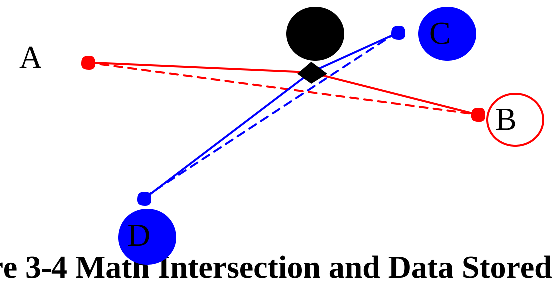


Figure 3-4 Math Intersection and Data Stored in the Computer
Mathematical intersection and actual data stored in the computer sometimes are different. Storing mathematical intersection points needs unlimited space while the space for storing data in the computer is limited. This limitation can sometimes cause some algorithms to fail.

3.4.2 Edge Matching Problems

Shared edges are edges that have the same start point and end point from different graphs or from different layers. They should remain consistent. However, because of the computational precision problems mentioned above, the edges may in some cases be the same, and in other cases they may be different.

These problems impact basic correctness of many published algorithms. Ignoring such issues has a greater impact than, for example, just causing some points to be rounded. Specifically, many pathological cases, may be formulated that cause catastrophic failures in an implementation (i.e. the program crashes).

Chapter IV
Implementations

4.1 Basic Definitions

Before describing algorithms presented here, some terms that are to be used in the following sections are first introduced. The notations in this section will be used through out this paper.

Vertex: The topological construct corresponding to a point. Its domain, if present, is a point in 2D or 3D space. Let A_i denote the i-th vertex in the plane. $A_i <_y A_j$ denotes the y-coordinate of A_i is smaller than the y-coordinate of A_j and $A_i =_y A_j$ denotes y-coordinate of A_i equals the y-coordinate of A_j, this notation is also used with relation to x-coordinates.

Event Point: While the sweep line moving downwards in the plane, it halts at particular points. These points are called event points. In this algorithm, the event points are the endpoints of the line segments, which we know beforehand, and the intersection points, which are computed on the fly.

Edge: corresponds to the connection between two vertices. Its domain is a finite, non-self-intersecting open curve. An edge has two end-points and its length is greater than zero. $E[A_iA_j]$ denotes an edge that has vertex A_i and vertex A_j as its end-points.
Segment: similar to an edge, it is also a closed line. It stores an **upper-end-point** and a **lower-end-point**. Let $S[A_iA_j]$ denote a segment that has vertex A_i and vertex A_j as its end-points. If $A_i <_y A_j$, or $A_i =_y A_j$ and $A_i <_x A_j$, in the Windows coordinates scheme, A_i is the **upper-end-point** and A_j is the **lower-end-point**. Zero-length segments are treated as vertices.

Half-edge: A half opened edge, which only includes the origin point, is called a half-edge. A half-edge stores an origin pointer, a pointer to its twin (see below), and a pointer to the face (see below) that it bounds. $E[\overset{?}{V_iV_j})$ denotes a Half-edge that has vertex V_i as its origin and vertex V_j as its destination. An example is shown in Figure 4-1. The Half-edges in each **face** are classified into two categories: **main-half-edge** and **twin-half-edge**. Take the Face-1 in Figure 4-1 for example, if one walks along its **main-half-edge** $E[\overset{?}{V_1V_2})$, $E[\overset{?}{V_2V_3})$, $E[\overset{?}{V_3V_4})$, $E[\overset{?}{V_4V_5})$, and $E[\overset{?}{V_5V_1})$ the Face-1 (Figure 4-1) will lie to the left. The **main-half-edge** $E[\overset{?}{V_1V_2})$'s **twin-half-edge** is $E[\overset{?}{V_2V_1})$ which has vertex $\mathbf{V_1}$ as an origin and vertex $\mathbf{V_2}$ as the destination. Half-edge $E[\overset{?}{V_2V_1})$ is one of the main-half-edges in Face-2 and $E[\overset{?}{V_2V_1})$'s **twin-half-edge** is $E[\overset{?}{V_1V_2})$.

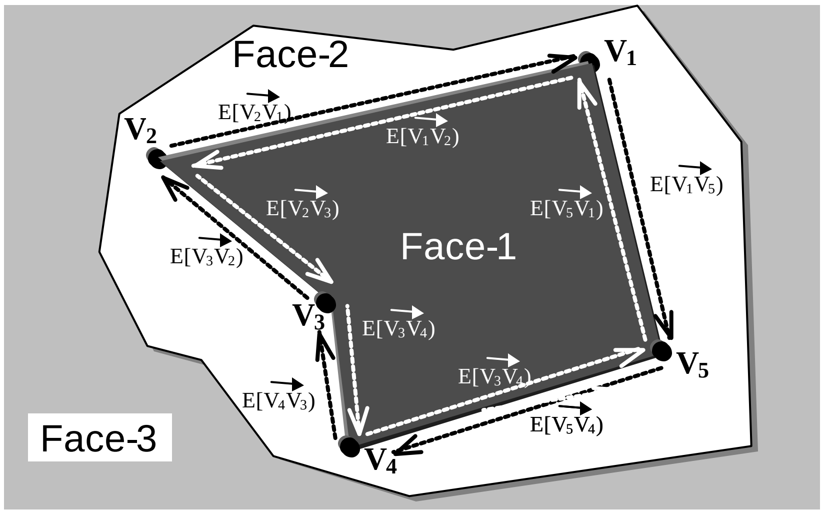


Figure 4-1 Half-Edge and Face
Above figures show half-edges and faces. The Main half-edges bound Face 1 and the Twin Half-edges bound Face 2.

Face is a topological entity of dimensionality 2 corresponding to the intuitive notion of a piece of surface bounded by its **main-half-edges**. A face stores the properties of an object, such as colors, the sequence sequential numbers, etc. In Figure 4-1, Face1 is bounded by Half **main-half-edge** $E[\overset{?}{V_1V_2})$, $E[\overset{?}{V_2V_3})$, $E[\overset{?}{V_3V_4})$, $E[\overset{?}{V_4V_5})$, and $E[\overset{?}{V_5V_1})$ while Face-2 is partially bounded by $E[\overset{?}{V_2V_1})$, $E[\overset{?}{V_1V_5})$, $E[\overset{?}{V_5V_4})$, $E[\overset{?}{V_4V_3})$, and $E[\overset{?}{V_3V_2})$. If a face has no outer main-half-edges, it is defined as the nil face. Face-3 in Figure 4-1 is a example. **Face ID** represents the relative order in which faces were created as instigated by the metafile record ordering within the input file.

Red-black Tree: The Red-black tree is a binary search tree with one extra bit of storage per node: its color, which can be either red or black. Using the x-coordinate of the intersection as a key value, a Red-black tree that stores all the segments that intersect with the Sweep-Line is called a **Status-tree**.

Sweep-line: an imaginary line that moves downwards over the plane, starting from a position above all segments to the bottom of lowest segments. At each event point, we process all the segments that intersect with the sweep line.

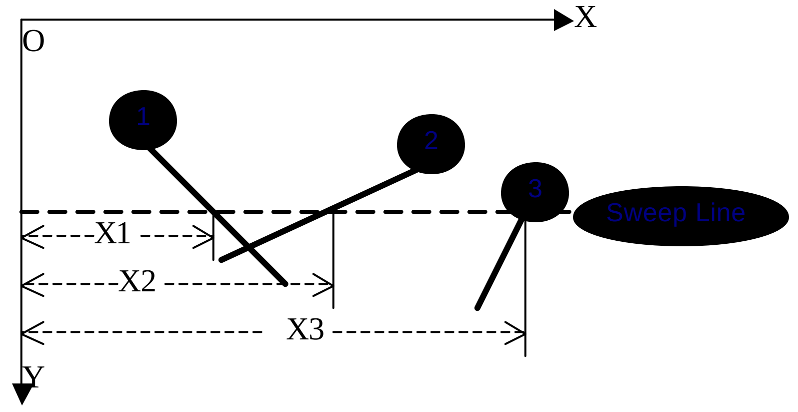


Figure 4-2 Sweep-line in Windows Coordinates
X1, X2, X3 are the keys for segment 1, 2, 3. The Keys are used to sort the segment in the status tree.

Object: An object is a distinct, named set of attributes that represents a graphic. The attributes hold data describing the graphic. Attributes of a graphic includes the color, outer-contour, inner-contours (holes), number of contour, and object ID. **Object ID** is a unique value that identifies each object in the file.

4.2 Data Structures

A Red-black tree (Figure 4-3) is a binary search tree and was introduced by R. Bayer[Ba72] in 1972. The Red-black tree is a binary search tree with one extra bit of storage per node that indicates color, which can be either RED or BLACK. By constraining the way nodes can be colored on any path from the root to a leaf, Red-black trees ensure that no path is more than twice as long as any other, so that the tree is approximately balanced. A Red-black tree has the following properties:

1. Every node is either red or black
2. Every leaf (NULL) is black
3. If a node is red, then both its children are black
4. Every simple path from a node to a descendant leaf contains the same number of black nodes
5. The height is O(lg(n)) for an n node Red-black tree

Another important property is that a node can be added to a Red-black tree and, in O(lg(n)) time. Similarly, a node can be deleted (or find) from a Red-black tree and, in O(lg(n)) time. The tree is readjusted to become a larger or a smaller Red-black tree after adding or deleting a node. Due to these properties, Red-black trees are used in this project.

Each node of the tree now contains the fields color, key, left, right, and parent. We define a node data structure of the Red-black tree as follows:

```
typedef struct tagREDBLKNODE {
    void*  pKey;
    void*  pItem;
    int    nRed; /* if red=0 then the node is black */
    struct tagREDBLKNODE* pLeft;
    struct tagREDBLKNODE* pRight;
    struct tagREDBLKNODE* pParent;
} REDBLKNODE;
```

If the child and the parent of a node do not exist, the corresponding pointer field of the node contains the value NIL. We regard these NIL's as being pointers to external nodes (leaves) of the binary search tree and the normal, key-bearing nodes as being internal nodes of the tree.

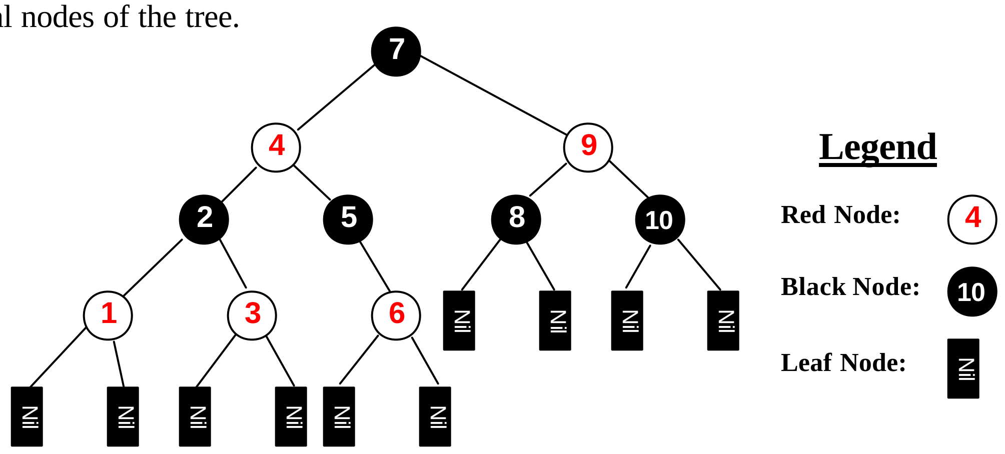


Figure 4-3 Red-black Tree
*All the data are stored in the internal nodes. An alternative way to build a Red-black tree is to store all the data in the external nodes. To store all the data in the external node may be efficient in searching, but it uses O (2*N) space. The Nil nodes which terminate the tree are considered to be the leaves and are colored black.*

During insert and delete operations, nodes may be rotated to maintain tree balance. Both average and worst-case search time is $O(\text{Log}_2 N)$. Because nil nodes don't have parents and children, one memory location can be used for all leave nodes. We also need one extra node as the header of the tree. Therefore, Storing all items uses O(N+2) spaces.

A Red-black tree class in this project has implemented in C++. Other items, such as segments, half-edges and vertices, are stored in Red-black trees class. In order to compare the items of two Red-black tree Nodes, a comparison function must be passed into the Red-black tree class. The comparison function returns a value the way strcmp does: negative if the first argument is "less" than the second, zero if they are "equal", and positive if the first argument is "greater". Here is an example of a comparison function which works with double:

```
Int compare_doubles (const void *a, const void *b)
{
  const double *da = (const double *) a;
  const double *db = (const double *) b;
  return (*da > *db) - (*da < *db);
}
```

4.3 Data Memory Managers

We use memory managers to manage data and to prevent memory leakage. A memory manager keeps track of small blocks in a table and larger blocks in a linked list. All the memory in the Red-black Tree is from the same pool and so can be allocated and freed in different threads without problem. Freed memory is reused; otherwise, the process will quickly run out of memory and slow down the application.

Note that the manager can also hold much more memory than is actually allocated. For most applications, this is an acceptable price to pay for increased speed.

4.4 The Two Segment Intersection Algorithms

4.4.1 A Simple Method to Find Two Segments Intersection

Let A,B,C,D be 2-space position vectors. Then the directed line segments AB and CD are given by the parametric equation:

AB=A+r(B-A), r ? [0,1]
CD=C+s(D-C), s ? [0,1]

Solving the above for r and s yields:

$$r = \frac{(YA-YC)(XD-XC)-(XA-XC)(YD-YC)}{(XB-XA)(YD-YC)-(YB-YA)(XD-XC)} \quad \text{(eqn 1)}$$

$$s = \frac{(YA-YC)(XB-XA)-(XA-XC)(YB-YA)}{(XB-XA)(YD-YC)-(YB-YA)(XD-XC)} \quad \text{(eqn 2)}$$

Let I be the position vector of the intersection point, then I=A+r(B-A). By examining the values of r and s, we can also determine some other limiting conditions: If 0<=r<=1 & 0<=s<=1, the intersection exists; if r<0 or r>1 or s<0 or s>1 the line segments do not intersect.

The above method is an exact arithmetic model which is based on real numbers and is very sensitive to computation errors. When two line segments are very close or when two tiny line segments are very close then the chance of errors increases. Figure 4-4 shows the relationship between the difference of two line slopes and relevant floating point rounding error.

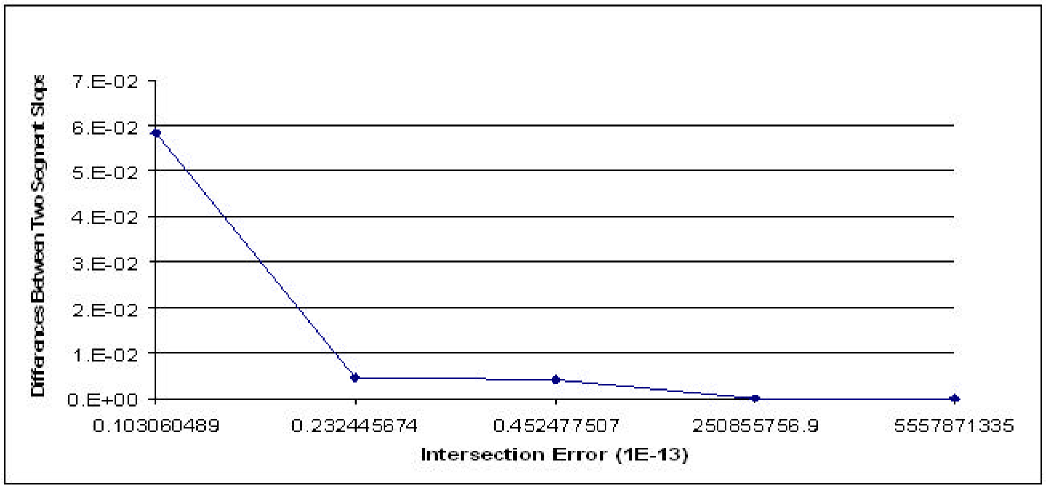


Figure 4-4 Relation between Slope Difference and Intersection Errors
The error means the difference between the values calculated by this method and the mathematically calculated value. The curve indicates that when two line's slopes get closer, the chance of rounding error increases. 1E-13 means 10^{-13}.

4.4.2 Algebraic Degree Predicates

Reporting an intersection of a pair of segments needs a predicate of algebraic degree 2 [BP00]. The main idea of the predicate is the following. A_i denotes an endpoint of a segment. $[A_iA_j]$ denotes the line segment whose upper and lower endpoints are, respectively, A_i and A_j. (A_iA_j) denote the line containing $[A_iA_j]$. $A_0 <_y (A_iA_j)$ denotes A_0 is above line (A_iA_j) (in Windows coordinates). We will use the above notations hereafter. Predicate degrees for finding segment intersections are listed below:

Predicate 1: $A_0 <_y A_1$

Predicate 2: $A_0 <_y (A_1A_2)$

Predicate 2': $[A_0A_1]$ n $[A_2A_3]$?0

Predicate 2 is equivalent to evaluating the sign of the orient as determinate:

Define: orient $(A_0A_1A_2) = \begin{vmatrix} X_0 & X_1 & X_2 \\ Y_0 & Y_1 & Y_2 \\ 1 & 1 & 1 \end{vmatrix} = \begin{vmatrix} X_1-X_0 & X_2-X_0 \\ Y_1-Y_0 & Y_2-Y_0 \end{vmatrix}$

4.4.3 Finding Two Segment Intersections with Predicates

Let $[A_0A_1]$ and $[A_2A_3]$ be the two segments whose intersection is to be determined. Define N_I = orient$(A_0A_2A_3)$; D_I = orient$(A_0A_1A_2)$ - orient$(A_0A_1A_3)$. Without loss of generality, we assume that $A_0 <_y A_2$ (otherwise just switch their order). The following is the algorithm for finding the intersection of the two segments (See Appendix III for C++ format code).

<u>Algorithm for Finding Two Line segments Intersection:</u>

```
1. If $A_1 <_y A_2$ //Predicate 1
2.       return false;
3. If $A_3 <_y A_1$
4.     If orient($A_0A_1A_2$)* orient ($A_0A_1A_3$) < 0  //predicate 2
```

```
5.          $I = A_0 + (A_1 - A_0)* N_I / D_I$ ;
6.          return true;
7.      Else return false;
8.  Else
9.      If $orient(A_0A_1A_2)* orient(A_2A_3A_1) > 0$ //predicate 2
10.         $I = A_0 + (A_1 - A_0)* N_I / D_I$ ;
11.         return true;
12.     Else return false;
```

From Line 4 and Line 9, we can see that the maximum algebraic degree is 2. Therefore Predicate 2' reduces to Predicate 2.

4.4.4 Why 4.4.3 is more accurate than 4.4.1?

The above two methods (in practice there are some other methods) logically are the same; with predicates, why do we get a more accurate result? Similar to the floating-point rounding errors we discussed in Chapter 3, without extra checking, the following error happens with the method in 4.3.2: when two lines are very close, for example, the closed points from two segments are (x1,y1) and (x2, y2), respectively. If, for example $X1=6.87 \times 10^{-97}$ and $X2 =6.81 \times 10^{-97}$ and $Y1=7.88 \times 10^{-97}$and $Y2 =7.83 \times 10^{-97}$**,** due to the floating round error, both X1-X2 and Y1-Y2 will be equal to 0. Therefore, point (X1,Y1) or point (X2,Y2) will be returned as an intersection point, which could not happen in 4.4.3.

4.5 Segment Intersection and the Line-Sweep Algorithm

We reveal missing algorithmic details of the description in [BO79] and in Computational Geometry [BKO+97]. We present detailed descriptions related to the handling of segments that are parallel and overlapping. Because the Bentley and Ottmann algorithm [BO79] is very sensitive to numerical errors [BP00], we will present a modification here.

Let S be the set of segments of all faces in the plane. Let Q be the sorted vertices (sorted by y and then x values inside the Red-black tree) in the plane, these points will be evaluated as "event points" within the algorithm. Let t be the sorted list that stores those segments that intersect with the sweep line. P is the pointer that points to the current value of the Q. Let U(P) be the set of segments which has P as its upper endpoint. Let L(P) be the subset of t which has P as its lower endpoint. Let C(P) be the subset of t which has P as its interior point, meaning P is on that segment but is not the endpoint. S_l(P) and S_r(P) denote, respectively, the left and right neighbors of P in t. Let A be the collection segments in the status-tree. Let M_l(A) be the left-most segment of A and M_r(A) be the right most segment of A. The following is the algorithm developed for FindIntersections.

FindIntersections(S)

```
1.  Initialize t
2.  while Q is not empty
3.      p=Q.value;
4.      HandleEventPoint(p);
5.      p= Q.next;
```

HandleEventPoint(p)

```
1.  if t is empty
2.          Select $U_2$ (P) from S and store them into t
3.              If $U_2$(P) is not empty
4.                      if any two segments in $U_2$ (P)has the same slope, break the
                        longer segment by the shorter segment
5.                      if segments in $U_2$ (P) are from different object, report P as
                        an intersection
6.          return
7.  UpdateStatusKey ( t )
8.  Select all C(P) from t , and break it into L(P) and $U_1$(P).
9.  Insert $U_1$(P) into S.
10. Delete L(P)and $U_1$(P) segments from t
11. Select $S_l$(p) and $S_r$(p) from t
```

```
12. Select $U_2$ (P) from S and store them into t
13. If U(P) from t is not empty
14.         if any two segments in U(P)have the same slope, break the longer
            segment by the shorter segment's lower end-point
15.         if segments in U(P) are from different object, report P as an
            intersection
16.         if segments in U (P) and L(P) are from different object, report P as
            an intersection
17. Get U(P) from t
18. Get, $M_l$(U(p)) and $M_l$(U(p)) from U(P)
19. if segments from U(P) and L(P) are not from one object
20.         report P as intersection
21. if   U(p) ==0
22.     then   FindNewEvent($S_l$(p)  , $S_r$(p), p);
23. else
24.      FindNewEvent ($S_l$(p)  , $M_l$(U(p)))
25.      FindNewEvent($M_r$(U(p)), $S_r$(p)  , p);
```

UpdateStatusKey (RedBlackTree)

```
1.   node=RedBlackTree->GetFirstNode();
2.   do{
3.       Let P the intersection of node->key;
4.       node= RedBlackTree->GetNextNode();
5.   }while(node !=RedBlackTree->GetLastNode();
```

FindNewEvent (left, right, p)

```
1.   Intersectpoint = FindIntersect(left, right);
2.   If Intersectpoint below the sweep line, or on it and to the right of the current
     event point p, insert Intersectpoint into Q;
```

In line 1 of FindIntersections(S), we initialize a status tree (t). The status tree stores those segments that intersect with the sweep-line. In line 2 to line 5, we are traversing a Red-black tree and P is the event point.

In line 1 to line 6 of HandleEventPoint (P), we deal with the situation when the status tree is empty (see Figure 4-5). We first select all the segments that have P as their upper endpoint, namely U(P). If any two segments in U(P) have the same slope, break the longer segment by the shorter segment's lower end-point and report the breaking point as an intersection. If not all the segments are from the same object set, we report P as an intersection and return and wait for the next event point.

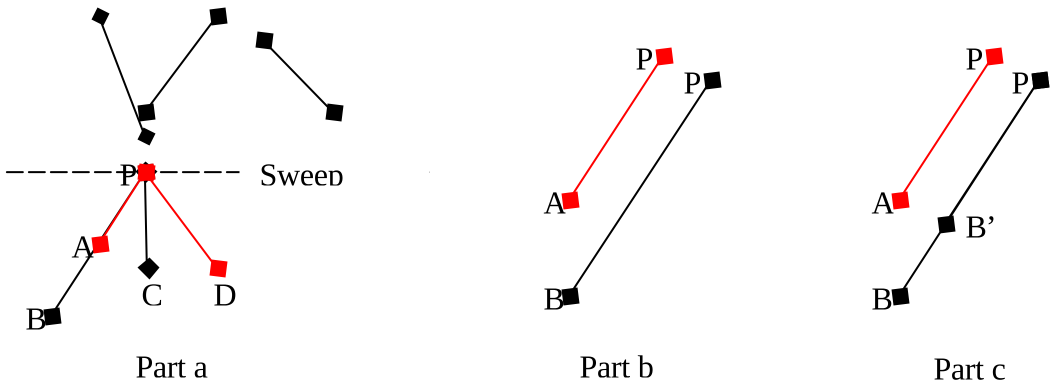


Figure 4-5 Situation When the Status Tree is Empty

Part a shows a situation in which the status tree was empty before at event point P. [PA] and [PB] are parallel and overlapping. Part b shows before breaking [PB]. Part c shows [PB] is broken in to Line [PB'] and Line [B'B]. Line [PA] and Line [PD] are from one object while Line [PB] and Line [PD] are from a different object. Therefore, point P and A are reported as intersections.

Line 7 updates all the key-values in the status tree `t`. The key-value is the x-coordinate of the intersection point of the sweep-line and the segments in the status tree `t`. At this stage, the order of the segments in `t` doesn't need to be reset, because their order will be reset at line 12 after reinserting segments. In *Computational Geometry* [BKO+97], the authors say that the order of the segments in the status-tree corresponds to the order in which they are intersected by the sweep line just **below** point E. However, this is not correct. Take Figure 4-6 as an example. According to the method in the book, the key value for [AB] cannot be found, because an intersection point below the sweep line is not present. So we propose to use two prime-keys to sort the segments in the status-tree. The first prime key we use is the X coordinate of the intersecting point intersected by the sweep line and the segment at E. The second prime key is the segment's slope.

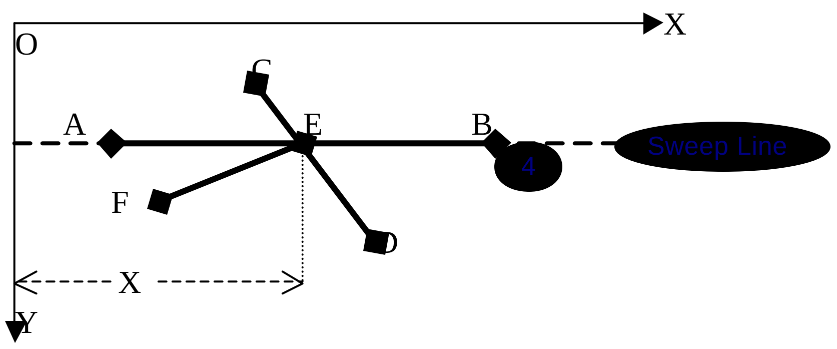


Figure 4-6 How to Update the Key

At event point E, if [AB] is horizontal, we use the x-coordinate of point E as the primary key. Therefore, [AB],[CD],[EF] have the same primary key, but they are differentiated by their slops.

In lines 8-10, we select all the C(P) segments from t and immediately break them by event P. By breaking the segment(s) at event point p, a new set of segments U_1(P), whose upper end points are P and whose lower end points are the lower points of the segments from C(P), is created. We insert U_1(P) into S and then reset the end points of the segments from C(P); as a result, C(P) becomes L(P). Remove L(P) and U_1(P) from t. Then, only reinsert U_1(P) into t, which will ensure that they are positioned in the correct order within t.

In line 11, we select the left neighbor segment(s) S_l(p) and the right neighbor segment(s) S_r(p) of event point P from t. Figure 4-7 shows the situation, S_l(p) and S_r(p) are, respectively, [AB] and [EF].

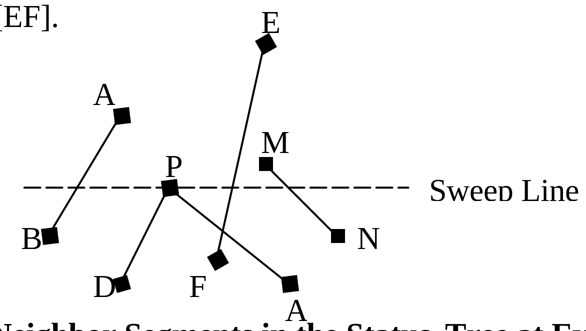


Figure 4-7 Neighbor Segments in the Status-Tree at Event P

[AB], [EF], and [MN] are the segments stored in the status tree. The S_l(p) of event point P is [AB] and the S_r(p) of event P is [EF].

Lines 12 -16 are is similar to lines 2-5. U(P) in Line 16 is equivalent to $U_1(P)n$ $U_2(P)$. $M_l(U(p))$ and $M_r(U(p))$ in line 18 are, respectively, the left-most and the right-most segments in U(P). In Figure 4-7, $M_l(U(p))$ is [PD] and $M_r(U(p))$ is [PA].

In Lines 19 -20, if L(P) and U(P) are not from the same object, we report P as an intersection. If U(P) is empty, in line 22 we will calculate the intersection between S_l(p) and S_r(p). Figure 4-8 shows such a condition.

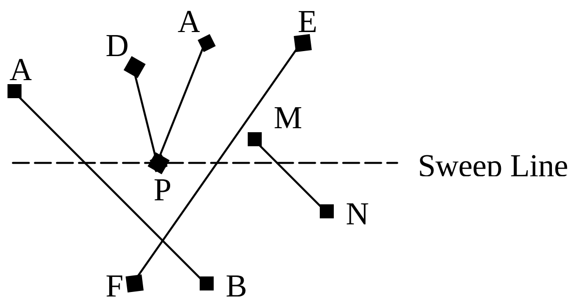


Figure 4-8 A Condition That U(P) Is Empty
[DP] and [AP] are in L(P) and were deleted from t*. Therefore, the $S_l(p)$ of event point P is [AB] and the $S_r(p)$ of event P is [EF].*

In Lines 24 -25, we calculate the intersections between $S_l(p)$ and $M_l(U(p))$, and between $M_r(U(p))$ and $S_r(p)$. For example in figure 4-7, U(P) is not empty, therefore, we need to calculate the intersection between [AB] and [PD] and between [PA] and [EF].

The FindNewEvent() function is used to find segment intersections with the methods we discussed in section 4.4. When new intersections are found, we insert them into the Red-black vertex tree. Note that finding the local intersections (intersections of the same object) and finding the global intersection (i.e. the intersection of segments that are from different objects) are different, as seen in the illustration examples in Figure 4-9. When the above algorithm is applied to find the local intersections, an intersection is only reported if two segments intersect where the intersection point is interior (not an end point) to at least one of the segments, or if there are more than 3 segments from U(P)*n* L(P). However, for finding global intersections, at event point P for example, as long as there are two segments in U(P)*n* L(P) that are from different objects, the event point P is reported as an intersection.

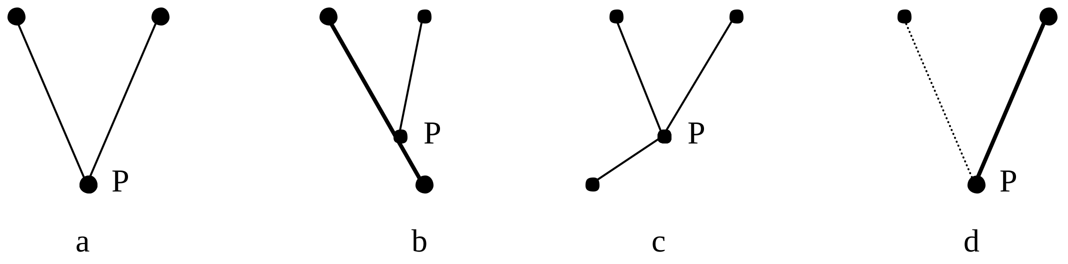


Figure 4-9 Segment Intersection

Figure a, b, and c illustrates segments from same object. Figure d shows segments from different objects. Event point P is not an intersection at a, but it IS in the situation of b and c. Event point P is an intersection in the situation of d.

4.6 Polygon Normalization

A polygon is a closed loop, represented by an ordered collection of vertices. The order in which vertices are traversed within such a loop is significant in that it may be used to indicate whether the loop represents the external edge or an internal hole of an associated contiguous region. The vertices of an external edge should be stored in a counter-clockwise order, and vertices representing inner holes should be stored in a clockwise order. This is done so that it is always know that the contiguous region delineated by a set of contours is always contained to the left of a contour edge if one images walking along a contour in the order within which the vertices are specified. However, the original Windows Metafile record order doesn't follow this rule. For example, sometimes the outer most edge of a region is specified in a clockwise order and other times it is specified in a counter-clockwise order depending on how the associated metafile record was originally created. Thus it is necessary to first detect the order in which a contour is specified and then, if necessary, alter or reverse that order to maintain the type of left-sided bounding consistency previously described. Therefore, we have to normalize the order of vertices after loading the records to make sure that the vertices of an external polygon are stored in a counter-clockwise order, and vertices representing inner holes are stored in a clockwise order.

4.6.1 Trapezium rule can not get a correct order for a random polygon

Consider a polygon made up of line segments between N vertices (x_i,y_i), i=0 to N-1. The last vertex (xN,yN) is assumed to be the same as the first, i.e. the polygon is closed. The area is given by

$$A = \frac{1}{2} \sum_{i=0}^{N-1} (x_i y_{i+1} - x_{i+1} y_i)$$

The sign of the area expression above is used to determine the ordering of the vertices of the polygon in many applications. If the sign is positive, then the polygon vertices are ordered counter-clockwise, otherwise, clockwise. The restriction is that the polygon must not be self intersecting. Therefore, the sign cannot be used to test order of vertices for self-overlapped polygons, such as those shown in Figure 4-11.

4.6.2 Test a List of Vertices' Order with a Half-edge Tree Scheme

At the time when the polygons are loaded, main half-edges are associated with the original vertices loading order and the twin half-edges with a reverse order. Let V_1, V_2, V_3, V_4, V_5, V_6, and V_7 be the loading vertices in order, for example, then $E[\overset{?}{V_1 V_2}]$, $E[\overset{?}{V_2 V_3}]$, $E[\overset{?}{V_3 V_4}]$, $E[\overset{?}{V_4 V_5}]$, $E[\overset{?}{V_5 V_6}]$, $E[\overset{?}{V_6 V_7}]$, $E[\overset{?}{V_7 V_1}]$ will be the main half-edges. A segment is defined as a ***going-down segment*** if and only if its main half-edge's origin is the segment's upper end-point. S[V_1V_2] in Figure 4-10 Part A is a **going-down segment**, because the S[V_1V_2]'s upper end-point is V_1 and S[V_1V_2]'s main half-edge's origin is also V_1. If a segment is not a going-down segment, then it is a **going-up segment**, for example, S[V_1V_2] in Part B.

Lemma 4.1: Order the points within a contour by their y coordinates, if two or more points have the same y coordinate, use the x coordinate to order them. The first point in this ordered list should be the top, left-most point within the contour. If the main half-edge vector that emanates from this vertex is a **going-down segment**, then the contour is specified in a counter-clockwise order, otherwise the contour is specified in a clockwise order.

Proof. Let V_1, V_2,... V_n be the input vertices list in order. At the time when the polygons are loaded, the main half-edges are associated with the original vertices loading order and the twin half-edges with a reverse order. We will get:

$$E[\overset{?}{V_1V_2}) \text{ is a main half-edge} \quad (1)$$

Without losing any generality, let V_1 be the top left-most point in the polygon, and V_1 is the upper end-point of S[V_1,V_2]. Assuming that S[V_1,V_2] is a ***going-down segment*** and the order of the list is clockwise order, then $E[\overset{?}{V_2V_1})$ will be the main half-edge. This contradicts (1). Therefore, Lemma 4.1 holds.

With Lemma 4.1, testing an input vertices' order only takes lgN time complexity, which is actually the cost to search for the top left-most point since the points and half-edges are already stored in Red-black trees to facilitate subsequent processing. Testing if the main half-edge emanating from that point is a going-down-segment may be done in constant time by simply comparing the x, y end points of half edge vector.

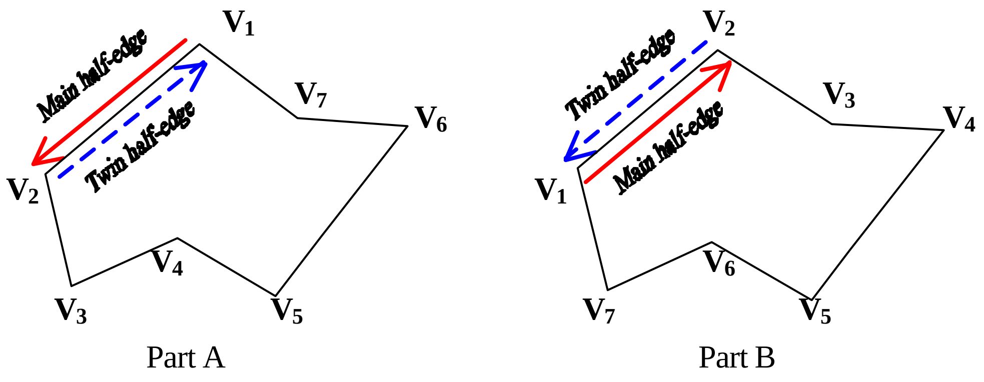


Figure 4-10 Using half-edge test polygon order

Part A is a counter clockwise ordered polygon; part B is a clockwise ordered polygon.

4.6.3 Processing Self-overlapping Polygons

"Self-overlapping" or "self-intersecting" or "degenerate" polygons are polygons that intersect themselves. Unfortunately, real Metafile records cannot be assumed to be simple polygons. Rather, real Metafile records tend to exhibit all types of deficiencies, such as self-intersections and grazing contact between polygons. Figure 4-11(A) shows an example of self-intersecting polygons and grazing contacts between polygons. Figure 4-11(B) shows the shapes after breaking self-overlapping polygons.

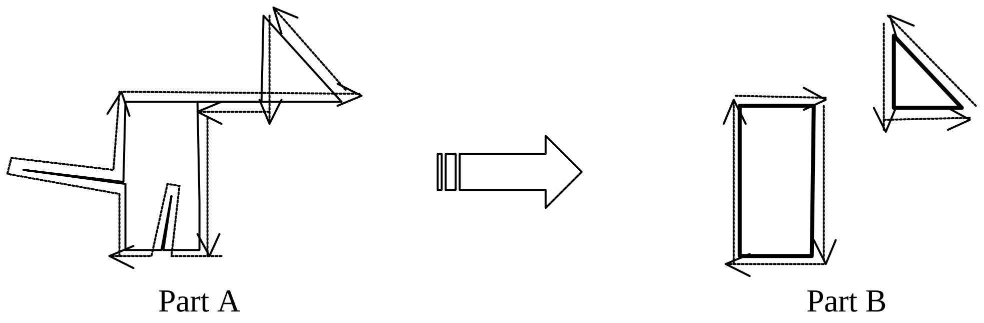


Figure 4-11 Polygon self-overlap

Solid lines represent the shapes and the broken arrow lines show the drawing direction. Part A is the original shape of the self-overlapping polygon. Part B are the normalized polygons after Part A is broken apart.

The main idea of the breaking self-overlapping algorithm is first to find self-intersections, then to separate the polygon into several sub-polygons at the self-intersection points, and delete any zero-area subset polygon(s). After polygon normalization, every polygon becomes a simple polygon. All the main half-edges go in a counter-clockwise order and all twin half-edges go in a clockwise order.

4.7 Edge Bounding Relations with Angles

4.7.1 What Is An Edge Bounding Relation?

The edge bounding relation represents the relationship between any two sets of edges at an intersection point. If an edge is inside a region which is surrounded by two other edges, the edge is called a bounded edge, and the other two edges are called bounding edges.

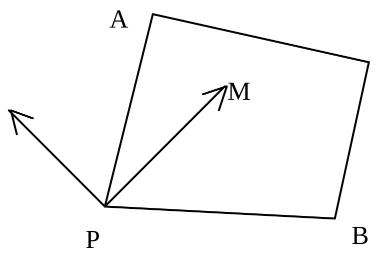


Figure 4-12 Edge Bounding Relation

In Figure 4-12, for example, edge [P,A] and edge [P,B] are boundaries of one region and edge [P,M] is a boundary of another region. Because edge [P,M] is inside the region that is surrounded by edge [P,A] and edge [P,B], edge [P,M] is bounded edge. Edge [P,A] is called the left-bounding-edge of [P,M], and [P,B] is called the right-bounding-edge of [P,M].

4.7.2 How Floating-point Rounding Errors Effect Edge Bounding Relations?

Once the segments are deleted from the event queue, these segments will not be used to find intersections again. Because of the floating-point rounding errors as stated in 3.4.2, the edges' bounding relation may change or create new intersections without being noticed with the sweep-line algorithm. An example is illustrated in Figure 4-13.

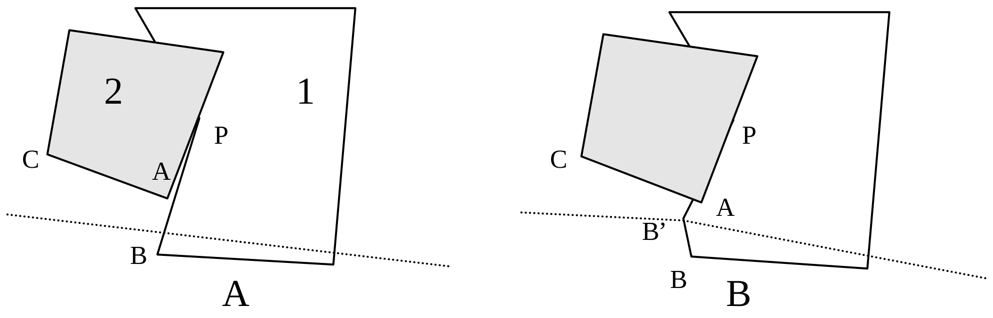


Figure 4-13 Edge Bounding Relation Problems
Part a shows a situation before inserting the intersection. Part b shows a situation after inserting intersection B'. When inserting intersection B', line [PB] will be twisted at point B'. Line [PB'] may affect previous edge bounding relations. Originally, line [PA] lies to the left of line [PB], after inserting point B', line[PA] lies to the left of line [PB']. Line [CA] intersects with Line [PB'] after inserting B'. However, it could not be reported by the sweep-line algorithm.

An edge bounding relation is used to construct new faces, which is similar to the Weiler-Atherton clipping algorithm [WA77]. Figure 4-14 illustrates the logical result of constructing a new face for object 1 if there no floating-point rounding errors.

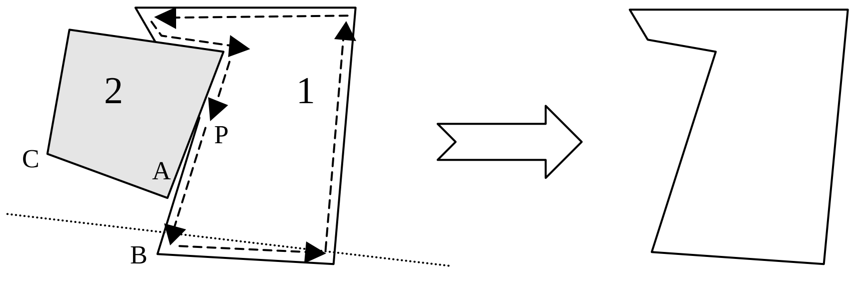


Figure 4-14 Logical Result

However, due to the computational error, some line segments may create new intersection points. Figure 4-13 (B) illustrates such an example. When the same algorithm is applied, infinite loops or incorrect results will be generated. In Figure 4-15, for example, p' is the intersection point created by inserting point B' , but missed being reported by the sweep-line algorithm. It is impossible to insert p' because when p' is inserted, this insertion could possibly produce another intersection. However, line [PB] is twisted by B'. By logical result, line [PB] should be outside of object 2. But the computed result makes line [PB] partially fall inside Object 2. Because line [PB'] partially falls inside object 2, but point P' was not reported as an intersection, when a new face is constructed starting at vertex V on object 1, the traversal algorithm degenerates into an infinite is shown in figure 4-16.

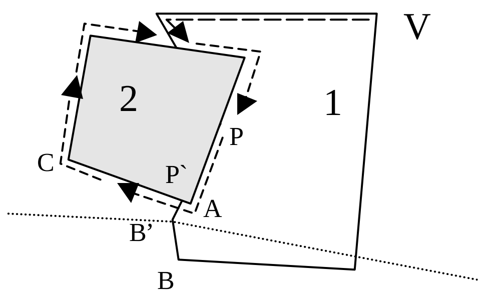


Figure 4-15 Infinity Loops during Face traversal

4.7.3 A Lazy Updating Angles Method

The angle of the original segment (i.e. the segment before being broken apart by an intersection point) is stored with the upper segment (i.e. the part that lies above the intersection point) and used for testing edge bounding relations; the angle of lower segment (i.e. the part that lies below the intersection), used for testing edge bounding relations, is updated with a newly computed angle (using the intersection point and the lower end point). For example, even line [PB'] in Figure 4-15 falls into the boundary of object 2 by floating-point rounding errors and they are labeled outside of object 2 because the original angle is used to test those edges relations at point P. This careful updating prevents erroneous arrangements or relationships among segments that have common end points.

4.8 Constructing New Faces

Before constructing a face, the pre-processing procedure is needed to eliminate generating islands when we construct new faces. Pre-processing consists of labeling the hidden object segments which are occluded within or beneath a more recently drawn object or record, for these hidden segments will not be used to construct new faces.

Faces can be directly constructed if a polygon is a non-intersected polygon. To construct the outer most edge of new face from intersected polygons, we start to traverse a non-hidden main half-edge at an intersection point and continue to its next half-edge until we come back to the starting point. We have to do the same traversal at all of the intersection points.

During traversal, when a new intersection point (not the start vertex), say P is found, a choice for the next half-edge must be made. Let E_{in} be the traversing half-edge that has P as its destination. Let E_{out} be a main half-edge that has P as an origin and its face incident is the face that was traversed initially (at the starting intersection point). Let E_t be the collection of twin half-edges that pass through the intersection point P. We select half-edges from E_t such that their twin-> face_incident ID's are greater than the incident face ID of the starting half-edge, denoted by $E'_t = \{ E_{t1}, E_{t2}, \ldots, E_{tn}\}$. Let E_m be the other main-half-edges that have P as their origin (excluding E_{out}). Figure 4-16 shows the different situations in which we choose the next half-edge at an intersection point.

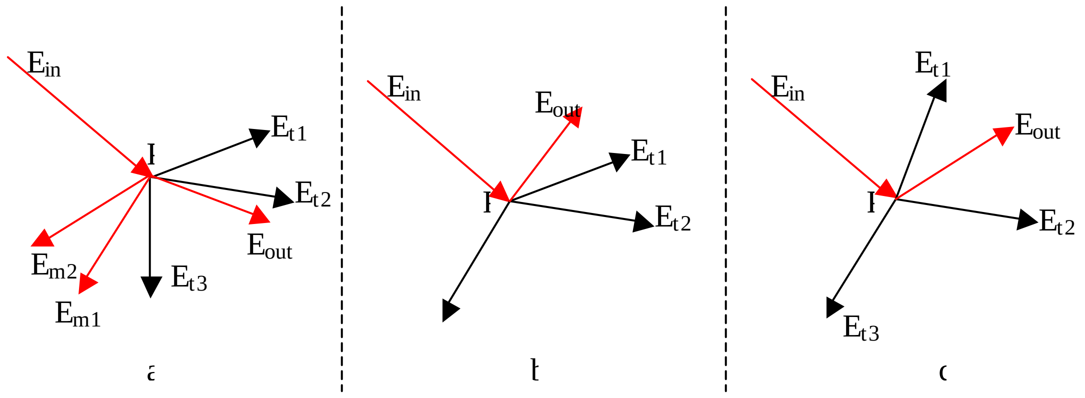


Figure 4-16 Selecting Next Half-edge During Traversing

In Figure 4-16 (a), if E_m's incident face, the color attribute is the same as $E`_{in}$ s, $E_m$ will be chosen as $E`_{in}$s next half-edge. If there are more than one E_m's, the one that next to right of right-most E_t , for example E_{m1} in Figure 4-16 (a), will be chosen as $E`_{in}$s next half-edge.

If there is no E_m next to right of right-most E_t, the half-edge to the left of E_{in} from $E`_t \cup E_{out}$ will chosen as the next half-edge to traverse. Figure 4-16 (b) and (c) illustration such situation.

4.9 Reload Data

A reloading procedure is needed after constructing new faces. Constructing new faces only constructs those polygons that intersect. In order to report the spatial relations between non-intersected faces and new constructed faces, we have to reload the faces. The reason to reload is that the current data is mixed with data that is never being used or data that has been checked as invalid data. Deleting the data costs more than just reloading the valid data.

4.10 Adding a Hole to a Face

4.10.1 Why Add Holes?

When two or more polygons overlap (but do not intersect), the new object hides or partially hides the old object(s). In order to embed the new object into the old object, we have to cut a hole from the old object. An example is shown in below:

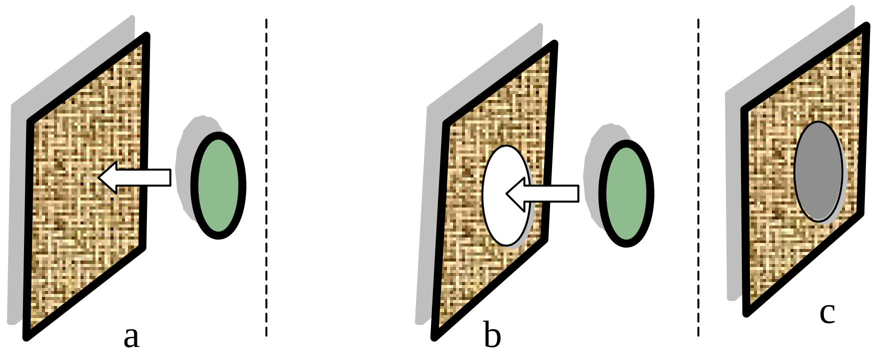


Figure 4-17 Two Objects Overlapping

a. Adding a new object onto another object. b. generating a hole in the old object to the same size of the new object. c. Final results.

4.10.2 Testing the Relationships between Separate Contours

In order to generate holes on one object, we need to find a way to test the regional relationships of any two given polygons.

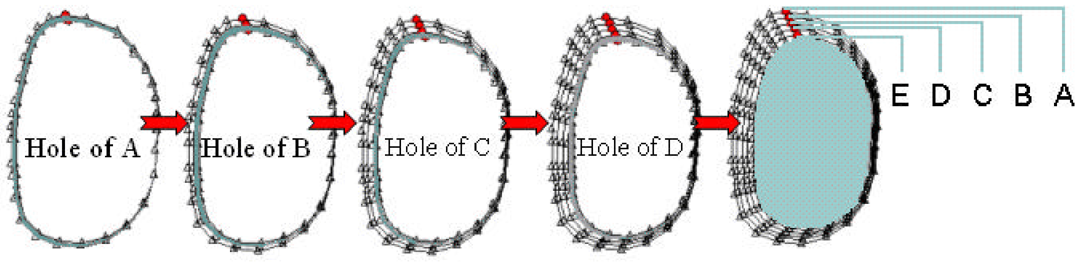


Figure 4-18 Relations between a set of polygons

Figure 4-18 shows that polygon E is inside polygon D, D is inside C, C is inside B and polygon B is inside polygon A. How do we know polygon E is inside D and not A, B, or C? Similarly, how do we know D is inside C and not A or B? The Brute Force method has to test the point with each edge to see whether the point is inside a polygon. Even with the scan-line method, to find an edge still takes time O(n), where n is the number of polygons. Based on fact that after reloading the data the relation between any two polygons is that either one is inside another or both are outside each other, in other words, no two polygons are partially overlapping, due to the fact that intersections have already been accounted for and the external contour of each new face has already been extracted (via the traversal algorithm presented earlier). Thus, any point (i.e. a seed point) may be taken from a polygon and used to find the nearest segment/main-half-edge to its left that may be classified as a going-down-segment (i.e. the origin point of the segment's main-half-edge lies above its destination point) and it is said that this point is inside the incident face of the associated going-down-segment/main-half-edge. If this seed point, say P, is on a polygon, say polygon1 and if P is inside another polygon, say polygon2, then polygon1 is inside polygon2. The following is our algorithm to find the outer polygon. Let P be the seed point from a polygon (inside polygon) to be tested. Let S be the segments/half-edges from the plane.

Finding outside face algorithms:

```
1.  Select all the segments from S whose upper end points are above P, and store
    them into $S_{up}$.
2.  Select those segments from $S_{up}$ such that their lower points are lower than P.
    Store them into Red-black tree Q.
3.  Search the segment from Q that lies to the nearest left of P.
4.  if the nearest left segment is not found,
5.      return NULL.
6.  else
7.      Store the segment to $S_1$, and store the incident face of $S_1$ to F,
8.      if $S_1$ is going down,
9.          return F;
10.     else
```

```
11.    Find the segment, say $S_2$, that is a going-up segment from F that lies to
       the left of $S_1$ in the Q.
12.    Delete $S_1$ and $S_2$ from Q.
13.    Repeat 3-12;
```

In line 1, all the segments from S whose upper end points are above P are seleted, and stored into a Red-black tree $S_{up.}$ In line 2, those segments from S_{up} whose low end-point is lower than P are selected and stored into Red-black tree Q. In line 3, a segment to the left of P needs to be found. First compute the x coordinate for each segment in Q using the y coordinate of P, then get the segment whose x-coordinates is the largest(in Windows coordinate system). If the segment is not found, a Null will be returned in line 5; otherwise, the face information of the segment left to P (let it be denoted by S_1) is stored into F. If S_1 is going down, (i.e. the origin point of the segment's main-half-edge lies above its destination point), F is returned in line 9. If the S_1 is going up, its twin (denoted by S_2) needs to be found in line 11. Note, in this case, S_2 can always be found in Q because all of the faces are closed polygons. This is the situation where P is outside and on the left of the face of the associated S1 and $S_{2.}$ Both S_1 and S_2 are deleted from Q in line 12, and repeat line 3-12.

If a face (inner face) is inside another face (outer face) and is not bordering on other faces, the inner face's twin-half-edges will be used to create a hole for the outer face. However, if the inner face is bordering on other faces, outlines of these bordering inner faces have to be created with their twin-half-edges. If the outline of these inside faces is in a clockwise order, the outline will be inserted into the outer face as a hole; if the outline of these inside faces is in a counter-clockwise order, a new face will be created.

The color property of the new face is the same as the property of the outer face. Figure 4-19 illustrates an example.

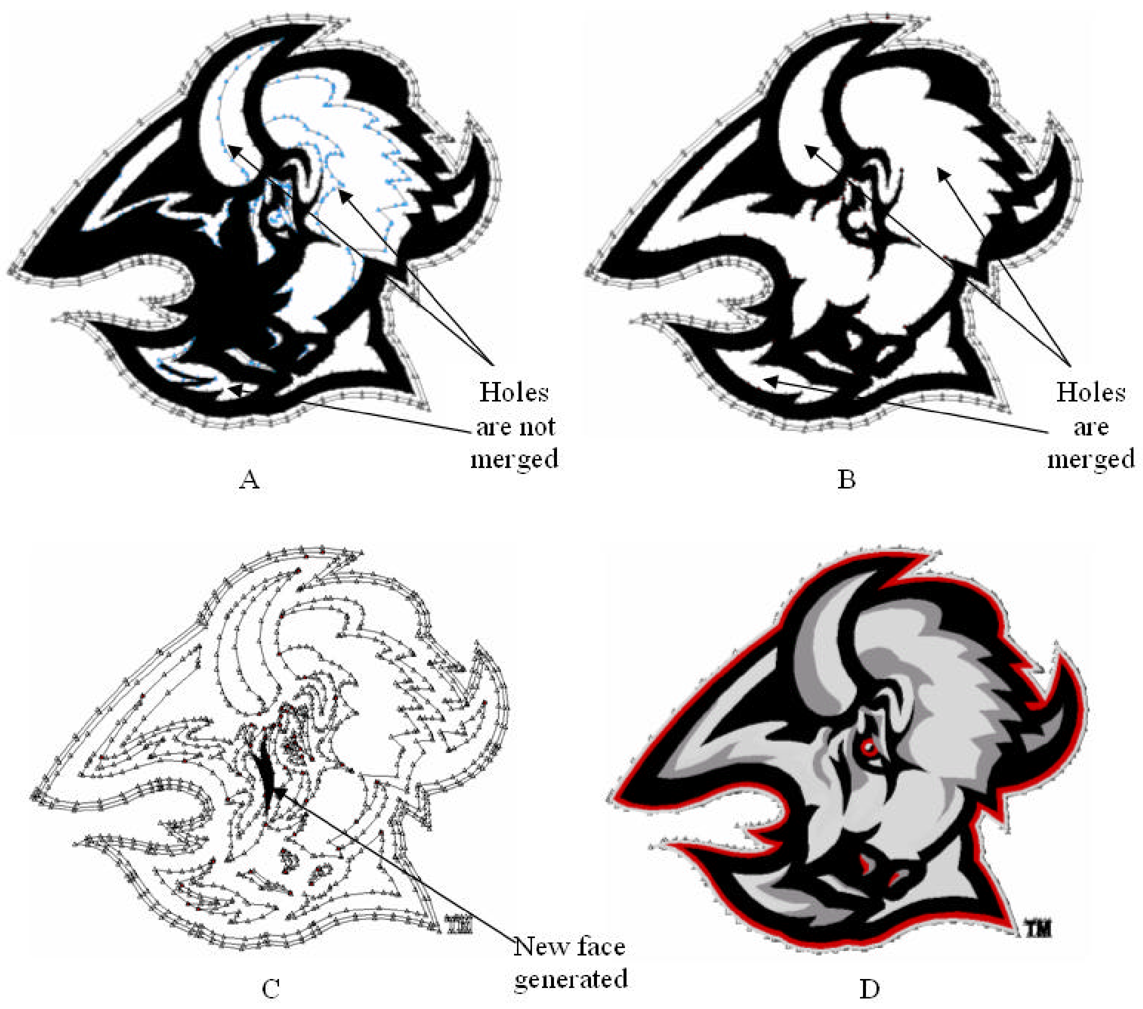


Figure 4-19 Faces and Holes

Part A shows holes are not merged; part B shows that holes are merged correctly; part C show a new face generated if the inside faces' outline is a counter-clockwise order and part D shows the final result of a Windows metafile compositing

Chapter V
Testing and Performance

We have spent a significant amount of effort tuning the code and paying attention to "hotspots" that slow down the implementation. Storage management is improved by memory managers which pre-allocate and reuse memory efficiently. Besides being efficient, we expect the implementation of the algorithm to be correct in handling all cases (including degenerate ones), to be stable in spite of numerical errors that occur in floating-point arithmetic, and to be dependably accurate in the final result.

Redundancy is avoided and even very large data sets may be effectively processed due in part to the utilization of Red-black trees and associated algorithms that bound most operations within O(lgn) time. The code has been tested successfully with a large variety of input files including many that contain sophisticated degenerate cases that would cause previously studied algorithms to fail.

Table 5-1 indicates the performance of the Windows metafile compositing. The computer used is a Dell laptop Inspiron 8100 with 1.0 GHz CPU and 256 megabytes of main memory. Runtime includes fixing self-overlap, finding intersections, constructing new faces, testing face special relations, and adding border information. From our test results, we can see that runtime increases gracefully as the number of input segments increases. In contrast, the number of intersections detected has a relatively small impact on runtime since constructing new faces and computing border information dominate runtime computation and these operations are proportional to the number of segments being processed. From Figure 5-3, we can see that the run time has a linear growth with N*lgN +C where N is the number of segments and C is the number of intersections.

Table 5-1 Input Data and Running Time

Layers (records)	Segments	Intersections	Time (Milliseconds)
12	4062	1	4557
25	4052	3	2814
41	1603	348	2163
43	2103	78	2013
50	2764	139	2574
54	2897	227	3095
55	4329	669	5518
55	5276	1345	5189
56	7909	98	6119
56	9406	948	9054
59	3836	299	2525
61	6203	1410	6670
64	6818	187	6800
85	4660	939	2200
103	4822	2830	8300
119	5363	2969	9600
119	10112	4	8582
144	6902	371	6239
148	52604	248	52185
151	14879	2049	15000
194	12132	161	12568
307	20631	3066	30654

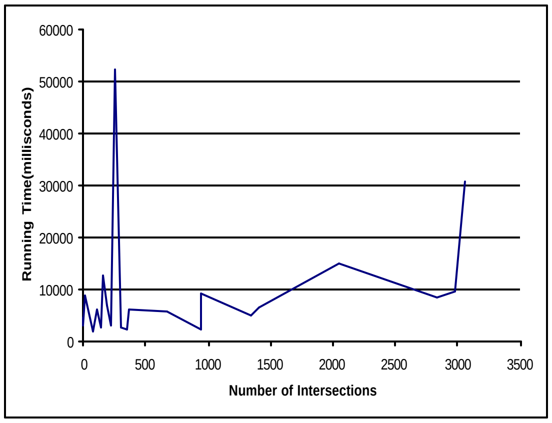


Figure 5-1
Running Time vs. Number of Intersections
Running time appears not grow linearly with number of intersections

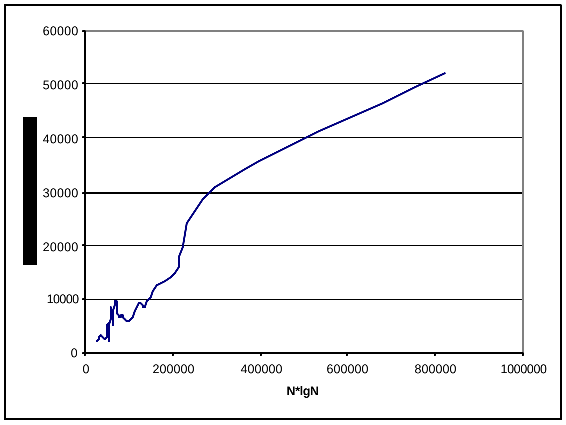


Figure 5-2
Running Time and Segments
*Running time appears to grow linearly N*lgN, where N is the number of segments*

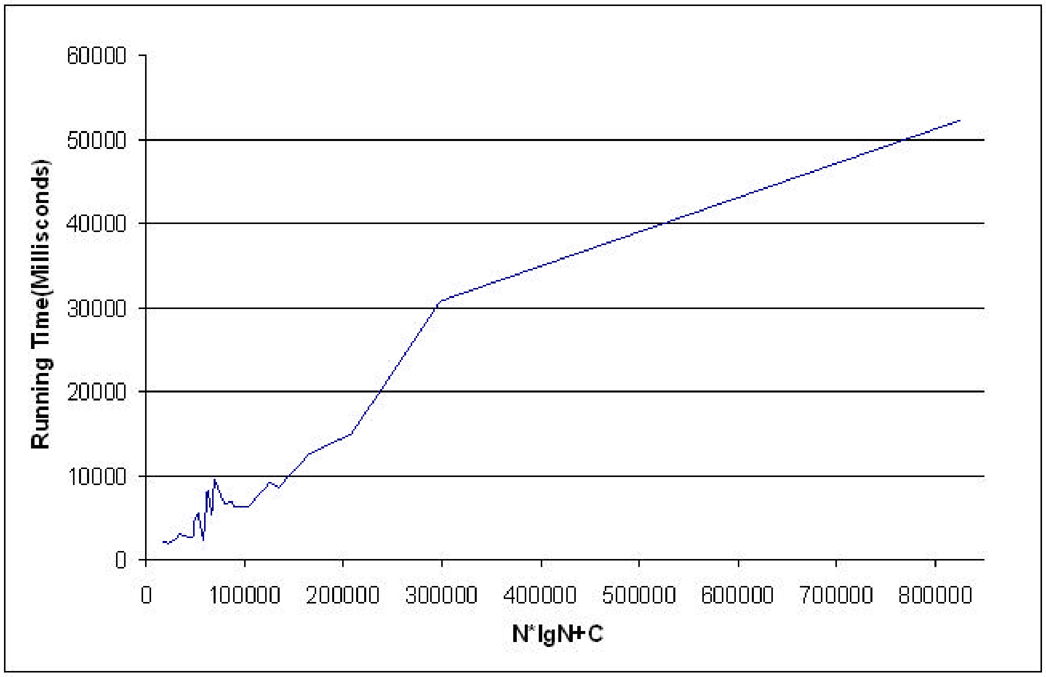


Figure 5-3 Running Time and Segments
*Running time appears to grow linearly with C+ N*lgN, where C is the number of intersections and the N is the number of segments.*

Using the Red-black tree data structure to test the order of the list of vertices takes time complexity O(1), while the conventional method takes time complexity O(n). Usually testing a point inside a polygon takes time O($\sum_{i=1}^{N} Ei$) with the Brute Force method, where E_i is the total segments of the i^{th} polygon object. Our method to test a point inside a polygon takes time O(NlgE), where N is the number of the polygons and E is the total number of edges.

Chapter VI
Conclusions

Initially, Brute Force algorithms were used for graphic file compositing: every line segment from one polygon set had to be tested for intersection with all line segments from the rest of the polygons. Constructing a new face had a limit of two faces at each step. And constructing all of the new faces in a plane increased the computation time significantly. It was found that compositing even a small metafile with the Brute Force method took almost 20 minutes of computation time on a 1GHz Pentium III; hours were needed with larger files. Map overlay and Sweep-Line algorithms shed new light on the possibility of developing a more practical and robust algorithm for metafile compositing. With these algorithms, all of the intersections of segments in a plane are found within a single sweep pass. New faces are constructed by tracing half-edges starting at a randomly–selected intersection and all new faces are constructed after all intersections have been traced. Non-intersecting faces are labeled and can be exported directly. Unfortunately, these methods have traditionally depended on using an exact arithmetic model which is based on real numbers. All computer calculations have finite precision and the precision problems may, in certain situations, cause these algorithms to fail catastrophically. In the investigation of robust graphic compositing algorithms presented in this paper, algebraic predicates have been utilized to allow the reliable and predictable detection of intersecting line segments permitting a correct and dependable implementation of Bently and Ottmann's sweep-line algorithm [BO79]. The way in which intersections are computed and way of testing edge bounding relations in this technique (referred to as a lazy update of segment angles) prevents erroneous arrangements or relationships among segments that have common end points. Without this precaution, errors in arrangements stemming from floating point computation errors would cause subsequently executed traversal algorithms to fail (e.g. enter infinite loops, etc.). Attention to details such as these and others discussed previously have allowed a comprehensive and robust solution to be formulated for the problem of metafile compositing that permits a practical real-world implementation. The key significance of our algorithm is that this robust Windows metafile compositing algorithm produces results similar to that of the theoretical algorithms presented in computational geometry literature that may be obtained without requiring expensive and impractical models of exact computation to be implemented or restricting input to exclude many degenerate, but common, input cases. This makes it very appropriate for engineering applications such as circuit design, vector file compression or embroidery imprint automation.

The algorithm described in this thesis has been implemented as an application in C++ and is also currently being utilized within Soft Sight, Inc.'s IntelliStitch software product, a leading embroidery imprint design automation package used within the textile industry. Future work may include exploring vector file compression benefits that can be achieved using metafile compositing and extending some of the solutions developed here to other problems within computational geometry such as the computation of vornoi diagrams

Acknowledgements

I am pleased to thank Dr. David Goldman for helpful discussion and for introducing Computational Geometry algorithms to this project.

This paper was supported by Dr. Richard Eckert since the beginning of this year and he has spent a lot of time helping me revise it. Without their help, this project would not have been possible.

Reference:

[Bur96] C. BURNIKEL, K. MEHLHORN, AND S. SCHIRRA. *On degeneracy in geometric computations*, in Proceedings of the 5th ACM-SIAM Symposium on Discrete Algorithms, Arlington, VA, 1994, pp. 16-23.

[HO93] S. Hoop, V. OOSTEROM, AND M. MOLENAAR. *Topological querying of multiple map layers*, in COSIT, (1993), Elba Island, Italy, pp. 139-157.

[Fr87] A. U. FRANK. *Overlay processing in spatial information systems*, in *AutoCarto 8*, (1987), pp. 12-31.

[FNS+89] W. R. FRANKLIN, C. NARAYANASWAMI, M. K. SUN, M.C. ZHOU, P.Y.P WU. *Uniform grids: a technique for intersection detection on serial and parallel machines*, in AutoCarto 9, (1989), pp. 100-109.

[Or91] J. ORENSTEIN. *An algorithm for computing the overlay of k-dimensional spaces*, in Advances on Spatial Databases, 2nd Symposium, SSD,(1991), Zurich, pp. 381-400.

[Ba72] R. Bayer. *Symmetric Binary B-Trees: Data Structure and Maintenance Algorithms*. Acta Informatica, Vol. 1 (1972), pp. 290-306.

[BO79] J. L. BENTLEY AND T.A. OTTMANN. *Algorithms for reporting and counting geometric intersections*, IEEE Trans. Comput., C-28 (1979), pp. 643-647.

[BP00] J. BOISSONNAT AND F. P, PREPARATA. *Robust Plane Sweep for Intersecting Segments*, SIAM Journal on Computing, Volume 29(2000), Number 5 pp. 1401-1421.

[BKO+97] M. DE BERY, M. VAN KREVELD, M. OVERMARS, O. SCHWARZKOPT. Computational Geometric. Springer-Verlag Berlin Heidelberg, Germany, 1997.

[Go91] D. GOLDBERG. *What Every computer Scientist Should Know About Floating-Point Arithmetic*. ACM Computer Surveys, Vol 23 (1991), No 1, pp.5-48.

[GHL+95] K. GOLDBERT, D. HALPERIN, J.C. LATOMBE, AND R.WILSON, Algorithmic foundations of Robotics. A.K. Perters, Wellesley, MA, 1995.

[WA77] K. WEILER AND P. ATHERTON. *"Hidden Surface Removal Using Polygon Area Sorting,"* Proc. Of the 4th ann. Conf. on Computer Graphics and Interactive Techniques, 1977, pp.214-222.

[BDS+92] J. D. BOISSONNAT, O. DEVILLERS, R. SCHOTT, M. TEILLAUD, AND M. YVINC, *Applications of random sampling to on-line algorithms in computational geometry*, Discrete Comput. Geom., 8(1992), pp. 51-57.

[BR81] W.S.BROWN. *A simple but realistic model of floating-point computation*, ACM Trans. Math. Softw. (1981), pp. 445-480.

[De96] M. DeBerg. *Computing half-plane and strip discrepancy of planar point sets.* Comput. Geom. Theory Appl., 6 (1996). pp. 69-83.

Appendix I

Computer Graphic Format Files

Aldus Placeable Metafile	*.amp
Adobe Illlustrator	*.al
Adobe Photoshop	*.psd
Adobe Portable Document Format	*.pdf
AutoCAD	*.dxf

clipboard metafile	*.clp
Computer Graphics Metafile	*.cgm
CorelDRAW	*.cdr
Encapsulated PostScript Format	*.eps
Extended Metafile	*.emf
Graphics Interchange Format	*.gif
Hewlett-Packard Graphics Language	*.plt
Joint Photographic Experts Group	*.jpg
Macromedia Flash	*.swf
Micrografx	*.drw
PICT – Mac graphics metafile	*.pct
Portable Network Graphics	*.png
Tag Image File Format	*.tif
Windows Bitmap	*.bmp
Windows Metafile	*.wmf
Windows Write	*.wri

Appendix II

1. The standard Windows metafile header:

```
typedef struct _WindowsMetaHeader
{
          WORD  FileType;      /* Type of metafile (0=memory, 1=disk) */
          WORD  HeaderSize;    /* Size of header in WORDS (always 9) */
          WORD  Version;       /* Version of Microsoft Windows used */
          DWORD FileSize;      /* Total size of the metafile in WORDs */
          WORD  NumOfObjects;  /* Number of objects in the file */
          DWORD MaxRecordSize; /* The size of largest record in WORDs */
          WORD  NumOfParams;   /* Not Used (always 0) */
} WMFHEAD;
```

2. The Clipboard metafile header:

```
typedef struct _Clipboard16MetaHeader
{
    SHORT MappingMode; /* Units used to playback metafile */
    SHORT Width;       /* Width of the metafile */
    SHORT Height;      /* Height of the metafile */
    WORD  Handle;      /* Handle to the metafile in memory */
} CLIPBOARD16METAHEADER;
```

```
typedef struct _Clipboard32MetaHeader
{
  LONG  MappingMode; /* Units used to playback metafile */
  LONG  Width;       /* Width of the metafile */
  LONG  Height;      /* Height of the metafile */
  DWORD Handle;      /* Handle to the metafile in memory */
} CLIPBOARD32METAHEADER;
```

3. The placeable metafile header:

```
typedef struct _PlaceableMetaHeader
{
    DWORD Key;         /* Magic number (always 9AC6CDD7h) */
    WORD  Handle;      /* Metafile HANDLE number (always 0) */
    SHORT Left;        /* Left coordinate in metafile units */
    SHORT Top;         /* Top coordinate in metafile units */
    SHORT Right;       /* Right coordinate in metafile units */
    SHORT Bottom;      /* Bottom coordinate in metafile units */
    WORD  Inch;        /* Number of metafile units per inch */
    DWORD Reserved;    /* Reserved (always 0) */
    WORD  Checksum;    /* Checksum value for previous 10 WORDs */

} PLACEABLEMETAHEADER;
```

3. The Windows enhanced metafile header:

```
typedef struct _EnhancedMetaHeader
{
   DWORD RecordType;       /* Record type */
   DWORD RecordSize;       /* Size of the record in bytes */
   LONG  BoundsLeft;       /* Left inclusive bounds */
   LONG  BoundsRight;      /* Right inclusive bounds */
   LONG  BoundsTop;        /* Top inclusive bounds */
   LONG  BoundsBottom;     /* Bottom inclusive bounds */
   LONG  FrameLeft;        /* Left side of inclusive picture frame */
   LONG  FrameRight;       /* Right side of inclusive picture frame */
   LONG  FrameTop;         /* Top side of inclusive picture frame */
   LONG  FrameBottom;      /* Bottom side of inclusive picture frame */
   DWORD Signature;        /* Signature ID (always 0x464D4520) */
   DWORD Version;          /* Version of the metafile */
   DWORD Size;             /* Size of the metafile in bytes */
   DWORD NumOfRecords;     /* Number of records in the metafile */
   WORD  NumOfHandles;     /* Number of handles in the handle table */
   WORD  Reserved;         /* Not used (always 0) */
   DWORD SizeOfDescrip;    /* Size of description string in WORDs */
   DWORD OffsOfDescrip;    /* Offset of description string in metafile */
   DWORD NumPalEntries;    /* Number of color palette entries */
   LONG  WidthDevPixels;   /* Width of reference device in pixels */
   LONG  HeightDevPixels;  /* Height of reference device in pixels */
   LONG  WidthDevMM;       /* Width of reference device in millimeters */
   LONG  HeightDevMM;      /* Height of reference device in millimeters */
} ENHANCEDMETAHEADER;
```

Appendix III

Convert replaceable metafile records to enhanced metafile records, and return Handle to an enhanced metafile records.

```
HENHMETAFILE  LoadMetaFile(CString strFilename)
{
          FILE*   fp;
          fpos_t  n64FSize;
          int     nFSize;
          unsigned char* pBuf;
          DWORD   dwErr;
          DWORD   dwKey;
          WMFSPECIAL  wmfsHdr;
          HENHMETAFILE    hEnMeta

          if ((strFilename.Find(".wmf",0)>0) ||(strFilename.Find(".apm",0)>0) ){
            //CreateMetaFile(strFilename);
            fp = fopen(strFilename,"rb");
            fseek(fp,0,SEEK_END);
            fgetpos(fp,&n64FSize);
            nFSize = (int)n64FSize;
            fseek(fp,0,SEEK_SET);
            fread(&dwKey,4,1,fp);
            if (dwKey==0x9AC6CDD7) { //if it is replaceable metafile
               fseek(fp,0,SEEK_SET);
               fread(&wmfsHdr,22,1,fp);
               nFSize -= 22;
            }
            pBuf = new unsigned char[nFSize];
            fread(pBuf,1,nFSize,fp);
            hEnMeta = SetWinMetaFileBits(nFSize,pBuf,NULL,NULL); //convert
            dwErr = GetLastError();
            delete[] pBuf;
            fclose(fp);
            //EMR_CREATEPEN
          } else
            hEnMeta = ::GetEnhMetaFile(strFilename);

          if (hEnMeta == NULL) {
            //AfxMessageBox("Error: unable to Open Meta file");
            MessageBox("Error: unable to Open Meta file");
            return NULL;
          } else
        return hEnMeta;
}//end of LoadMetaFile
```

Appendix IV

```
BOOL FindSegmentIntsctn(LPSEGMENT pSegA, LPSEGMENT pSegB, LPVERTEX pXPt)
{
            LPSEGMENT pSeg1, pSeg2;
            if(pSegA->pmvUpper->dY<pSegB->pmvUpper->dY){
               pSeg1=pSegA;
               pSeg2=pSegB;
            }else{
               pSeg1=pSegB;
               pSeg2=pSegA;
            }
            if(pSeg1->pmvLower->dY < pSeg2->pmvUpper->dY)
               return false;
            double dDi,dNi;
            double X0,X1,X2,X3;
            double Y0,Y1,Y2,Y3;
            double dOrient012, dOrient013,dOrient231, dOrient023;

            X0=pSeg1->pmvUpper->dX; X1=pSeg1->pmvLower->dX;
            X2=pSeg2->pmvUpper->dX; X3=pSeg2->pmvLower->dX;
            Y0=pSeg1->pmvUpper->dY; Y1=pSeg1->pmvLower->dY;
            Y2=pSeg2->pmvUpper->dY; Y3=pSeg2->pmvLower->dY;

            dOrient012 = (X1-X0)*(Y2-Y0)-(X2-X0)*(Y1-Y0);
            dOrient013 = (X1-X0)*(Y3-Y0)-(X3-X0)*(Y1-Y0);
            dOrient231 = (X3-X2)*(Y1-Y2)-(X1-X2)*(Y3-Y2);

            if(Y3<Y1){
               if(dOrient012*dOrient013>=0)
                  return false;
            }else{
               if(dOrient012*dOrient231<=0)
                  return false;
            }
            dNi=dOrient023 = (X2-X0)*(Y3-Y0)-(X3-X0)*(Y2-Y0);
            dDi=dOrient012-dOrient013;
            pmvXPt->dX=X0-(X1-X0)*dNi/dDi;
            pmvXPt->dY=Y0-(Y1-Y0)*dNi/dDi;
            return true;
}//end of FindSegmentIntsctn
```