

Raster to Vector conversion in a Local, Exact and Near optimal manner.

John Andrew Carter

A dissertation submitted to the Faculty of Science, University of the Witwatersrand, Johannesburg, in partial fulfilment of the requirements for the degree of Master of Science.

Pretoria 1991

**Abstract**

Remote sensing can be used to produce maps of land-cover, but to be of use to the GIS community these maps must first be vectorized in an intelligent manner.

Existing algorithms suffer from the defects of being slow, memory intensive and producing vast quantities of very short vectors. Furthermore if these vectors are thinned via standard algorithms, errors are introduced.

The process of vectorizing raster maps is subject to major ambiguities. Thus an infinite family of vector maps corresponds to each raster map. This dissertation presents an algorithm for converting raster maps in a rapid manner to accurate vector maps with a minimum of vectors.

The algorithm converts raster maps to vector maps using local information only, (a two by two neighbourhood). The method is "exact" in the sense that rasterizing the resulting polygons would produce exactly the same raster map, pixel for pixel.

The method is "near optimal" in that it produces, in a local sense, that "exact" vector map having the least number of vectors.

The program is built around a home-grown Object Oriented Programming System (OOPS) for the C programming language. The main features of the OOPS system, (called OopCdaisy), are virtual and static methods, polymorphism, generalized containers, container indices and thorough error checking. The following general purpose objects are implemented with a large number of sophisticated methods :- Stacks, LIFO lists, scannable containers with indices, trees and 2D objects like points, lines etc.

Declaration

I declare that this dissertation is my own, unaided work. It is being submitted for the degree of Master of Science to the University of the Witwatersrand, Johannesburg. It has not been submitted before for any degree or examination in any other University.

J. A. Louw  
This the 21<sup>st</sup> day of December, 1991.

To Rachel, without whom, Life, The Universe, and Everything would have become just plain too much. I.e. > 42

## Preface

The project to create a program to perform raster to vector conversion arose for a number of reasons :-

- 1) The Department of Water Affairs and Forestry Remote Sensing Section needed to distribute the results of mapping landcover by satellite images.
- 2) The existing software was bug-ridden and unsupported.
- 3) The major client for our data was the Department's Geographic Information System (GIS). The GIS required vectorised images.
- 4) The vectorization software on the GIS took a long time and produced neither optimal nor exact vectors, resulting in resistance to the acceptance of our results in daily use.

This work was made possible by the forbearance of the Department of Water Affairs and Forestry, the requirements of the Catchment Studies section of the Hydrological Research Institute, the support / nagging / useful discussions of my colleagues and the gentle amusement of God.

Part of this work was presented at the EDIS/SAGIS conference held in Pretoria during July 1991.

My thanks also to my proofreaders who revealed curious aspects of their characters by the nature of the errors they found.

## Glossary cum List of Abbreviations.

ASCII	-	American Standard Code for Information Interchange, a machine code for the display of human readable characters.
Else When	-	Quadrants 2 and 4.
GIS	-	Geographic Information System
join-point	-	Intersection of two lines approximating successive strings.
link-point	-	Point linking two strings.
mixel	-	MIXture pixEL
OOPS	-	Object Oriented Programming System
OpCdaisy	-	Homegrown Object Oriented Programming system in C
pixel	-	PICTure ELEment
raster classification	-	A grid representing landcover classes.
raster image	-	Picture consisting of a grid of numbers.
spaghetti	-	Edible - Marco Polo's new fangled pasta
spaghetti	-	Digital - lists of directed line segments
string	-	ASCII - C style string of characters.
string	-	Vexil - list of vexils
vector map	-	Map made up of directed line segments.
vector representation	-	A vector map representing a raster classification.
vexil	-	Vector bounding a pixel.
VMT	-	Virtual Method Table - the heart of OOPS
voxel	-	VOLUME ELEment

List of Figures.

Figure 1 From satellite to GIS, the grand scheme of things.	Page 12
Figure 2 Rasterizing a triangle. a) The original scene. b) Original scene with grid overlay. c) Only those grid cells with 50% of area within the triangle. d) The result.	Page 13
Figure 3 Vectorizing a triangle. e) Raster triangle. f) There are 12 unsmoothed vectors. g) Smoothing. h) Result is NOT the same as original. Right and bottom sides now match raster grid.	Page 14
Figure 4 Using thinning to smooth a triangle. i) Raster scene. j) Outline. k) 2nd approximation not exact. l) 3rd approx. but more vectors. m) Exact Approximation. n) Uses 6 vectors instead of 3!	Page 14
Figure 5 Block types and their edges.	Page 35
Figure 6 Examples of strings.	Page 40
Figure 7a) A "complex" but exact line. b) No exact line may replace this string.	Page 41
Figure 8 A Finite Automaton to recognise the language $u^n v^n$ .	Page 42
Figure 9 The duality between lines and points.	Page 44
Figure 10 The line must thread the vertices.	Page 45
Figure 11 Constraints in line space.	Page 45
Figure 12 This string cannot be represented exactly by any one line.	Page 46

Figure 13 The line must be pinned to the end of the vexil.	Page 46
Figure 14 Link point 1 is in the "Future", Link point 2 is "Else When"	Page 49
Figure 15 Previous example after smoothing.	Page 49
Figure 16 Space-time diagram illustrating the concept of "Else When".	Page 49
Figure 17 Anti parallel lines.	Page 1
Figure 18 Result of linking algorithm.	Page 51
Figure 19 String resulting in two parallel lines.	Page 51
Figure 20 Result of link Algorithm.	Page 51
Figure 21 The result of applying the Spaghetti Machine to a 256 by 256 pixel classification. (Unsmoothed)	Page 52
Figure 22 The result of thinning the spaghetti with the largest tolerance giving "exact" results.	Page 54
Figure 23 "Exact, Near-optimal representation"	Page 55
Figure 24 An example of a free-floating node.	Page 57
Figure 25	
a) Empty Stack	
b) Stack with point object	
c) Stack with line and point object.	Page 64

## Table of Contents.

Raster to Vector conversion in a Local, Exact and Near optimal manner. . . . .	Page 1
Abstract . . . . .	Page 2
Declaration . . . . .	Page 3
Preface . . . . .	Page 5
Glossary cum List of Abbreviations. . . . .	Page 6
List of Figures. . . . .	Page 7
Table of Contents. . . . .	Page 9
1. Introduction. . . . .	Page 12
2. OopCdaisy . . . . .	Page 16
2.1 What is object oriented programming?	Page 18
2.1.1 Representation hiding, and Inheritability - an example. .	Page 18
2.1.2 Polymorphism - An illustrative example. . . . .	Page 20
2.2 How does OopCdaisy implement objects?	Page 20
2.2.1 What is OopCdaisy? . . . . .	Page 21
2.2.2 What are OopCdaisy Objects and methods? . . . . .	Page 21
2.2.3 The global conception of object methods. . . . .	Page 21
2.2.4 Dynamic vs Static instances of Objects. . . . .	Page 22
2.2.5 How is a new class of objects created? . . . . .	Page 23
2.2.6 How is a new class of objects registered with OopCdaisy? . .	Page 24
2.2.7 How is an object created? . .	Page 25

	Page 10
2.2.8 How is an object used? . . . . .	Page 27
. . . . .	Page 28
2.3 Other OopCdaisy objects. . . . .	Page 28
2.4 OOPS in the literature. . . . .	Page 29
2.4.1 Turbo Pascal 5.5 . . . . .	Page 30
2.4.2 Oberon . . . . .	Page 31
2.4.3 Smalltalk. . . . .	Page 31
2.4.4 Eiffel. . . . .	Page 31
2.5 OOPS in the future. . . . .	Page 33
3. The first phase of raster to vector conversion -	
Producing Spaghetti . . . . .	Page 34
3.1 Edges, vexils, strings, nodes, knots,	
spaghetti - some definitions . . . . .	Page 34
3.2 An overview of how the Spaghetti Machine	
works. . . . .	Page 35
3.3 Block classes. . . . .	Page 35
3.4 Starting up the Spaghetti Machine. . . . .	Page 36
3.5 The Spaghetti Machine. . . . .	Page 37
4. Smoothing. . . . .	Page 38
4.1 Exactness. . . . .	Page 39
4.2 How does one find an exact, near-optimal	
vector representation of a raster image? -	
an overview. . . . .	Page 39
4.3 Parse the Spaghetti please. . . . .	Page 40
4.3.1 From string of vexils to a character	
string. . . . .	Page 41
4.3.2 The language of worms. . . . .	Page 42
4.4 Line Space. . . . .	Page 44
4.4.1 Threading the vexils. . . . .	Page 45
4.4.2 Missing the point. . . . .	Page 46
4.4.3 Putting all the constraints	
together. . . . .	Page 46
4.4.4 Which feasible line? Global versus	
Near-optimal representations. . . . .	Page 47
4.4.5 Keeping track of the constraints. . . . .	Page 47
5. Linking the Lines. . . . .	Page 48

	Page 11
5.1 Special cases. . . . .	Page 48
5.1.1 When a Link-point is Else When. . . . .	Page 49
5.1.2 Parallel lines. . . . .	Page 51
6. Examples and Discussion. . . . .	Page 52
7. Time complexity of the Algorithm. . . . .	Page 56
8. Conclusions . . . . .	Page 57
Appendices. . . . .	Page 61
Appendix 1. An Annotated Example. . . . .	Page 61
A.1.1 Declaring the data structure. . . . .	Page 61
A.1.2 Registering the object type. . . . .	Page 62
A.1.3 Implementing the data structure. . . . .	Page 64
A.1.4 Using the data structure. . . . .	Page 68

## 1. Introduction.

At the SAGIS conference on Geographic Information Systems held in Pietermaritzburg in July 1989, it was made clear that the majority of time, effort and expense in Geographic Information Systems, (GIS), is in the data capture phase. Remotely sensed data provides, relatively cheaply and rapidly, an already digitised form of landcover information. However, remotely sensed data is usually in raster (grid) form, and most GIS's work with vector polygon data. Thus remotely sensed data must be converted to GIS format before it can be used.

Figure 1 displays a very broad overview of the production cycle of an image. Firstly the scene is captured by one of the satellites, (eg LANDSAT 5), and the scene is converted into a grid of numbers called an image. The image is classified into classes of interest to

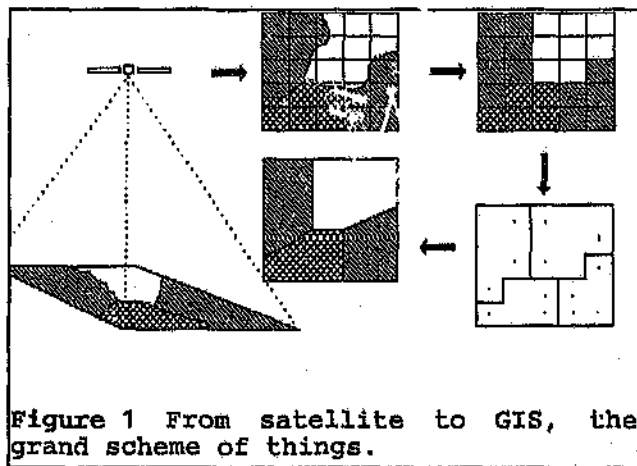


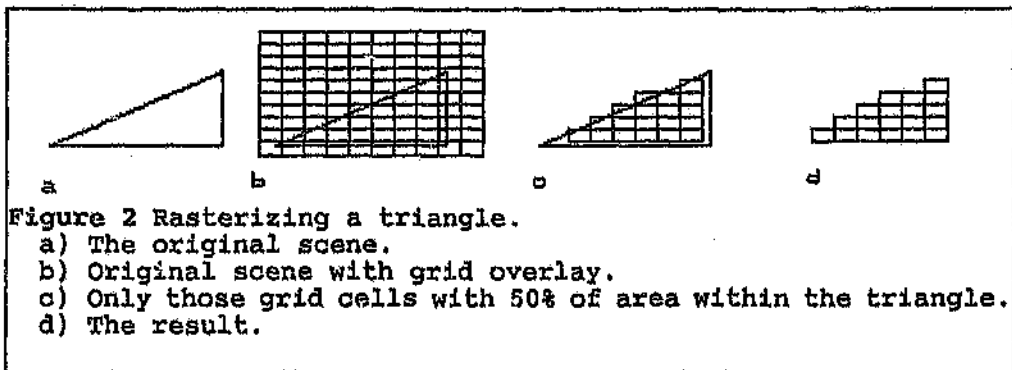
Figure 1 From satellite to GIS, the grand scheme of things.

the user. The classified image is in raster form and must be vectorized by finding a polygon that surrounds each region. These strings of vectors are then smoothed and exported to a GIS.

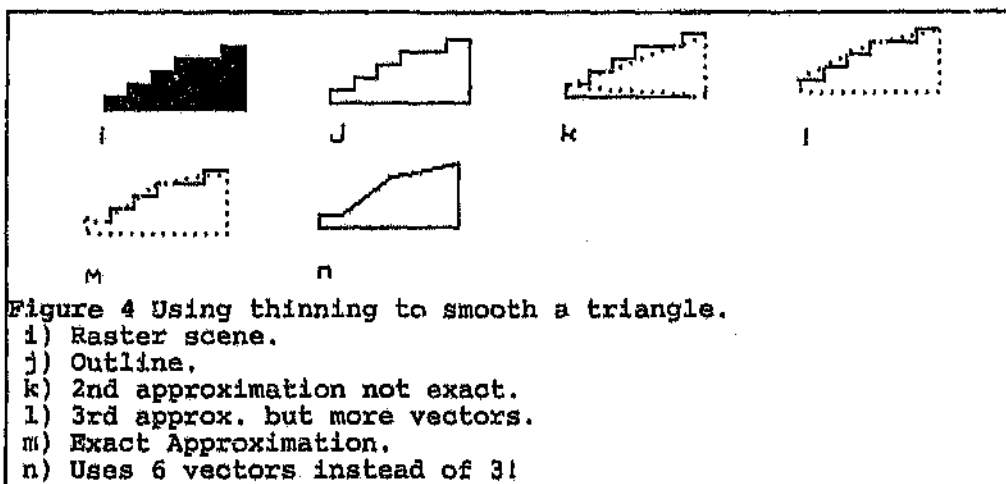
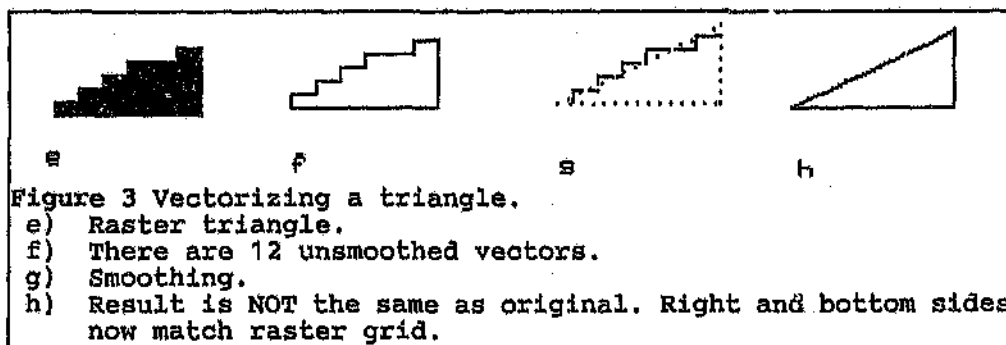
A GIS, such as ARC/INFO, provides a raster to vector conversion routine. So why write another? There are several motivations :-

- a) The image processing site is a service organization providing a service to clients using GIS's. Thus to fulfil our role we must provide data suitable for immediate consumption, and not requiring many hours of preprocessing to render it digestible to the GIS.

- b) Raster classifications can require very large amounts of storage space, especially if stored in a form easily transportable between different systems.
- c) The algorithm developed here is a local one and thus suitable for processing very large raster images, of the order of 8000 by 7000 pixels.
- d) The output of most raster to vector conversion routines is a set of vectors that outlines each pixel resulting in jagged lines. Consider the process of rasterizing a large vector triangle, as shown in Figure 2, and then vectorizing the resulting raster image, (Figure 3f). The end result may look quite like a triangle but may contain hundreds of tiny vectors instead of the original three.



- e) Smoothing or "thinning" algorithms, which are standard on most GIS packages, can be used to remove the "jaggies", but suffer from the following problems :-
  - i. Either they are not exact, i.e. a raster to vector conversion routine is exact if and only if rasterizing the resulting vector image, using the same grid, results in exactly the same raster image.
  - ii. Or they are not optimal. A raster to vector conversion routine is optimal if it results in an exact vector representation containing less than or the same number of vectors as all other exact vector representations.
  - iii. The vector end points are often constrained to lie only on pixel corners.



iv. The smoothing algorithm is general and is divorced from the vectorizing algorithm, thus it cannot make use of considerable information obtainable from contemplating the nature of the output of vectorizing procedures.

Two assumptions need to be made before continuing :-

- a) The features that make up our classes partition the earth into polygonal regions. This polygonal earth model, although it is only a poor approximation, is so universally used, (without mention), in mapping, that I will unabashedly adopt it henceforth.
- b) The rasterization / classification process works in the Westminster style. I.e. Each pixel is assigned the class

value of the class on the ground which occupies more of the pixel area than any other single class.

The raster to vector conversion process can be divided into three sections, the first of which, called the Spaghetti Machine, produces an outline of the pixels of each region. The second parses these outlines into line segments, and the third smooths these in an exact manner. But first a chapter on the programming environment.

## 2. OopCdaisy

While writing the raster to vector program, the I was concerned with several Software Engineering issues. An Object Oriented Programming System, called OopCdaisy, was built on top of the C programming language. The aims of OopCdaisy are to gain :-

### a) Reliability through :-

- i. Thorough error checking, to prevent hidden errors.
- ii. The availability of debugged sophisticated general data structures.
- iii. Strict control and accounting of objects created and deallocated.
- iv. Preventing use of deallocated objects.
- v. Ancestral type guards on parameters.

### b) Reusability through :-

- i. Polymorphism or genericity to share code.
- ii. Inheritability.

### c) Flexibility through :-

- i. Separating the data structures from the program logic.
- ii. Late binding.
- iii. A global conception of object methods according to rule :- "If two methods have the same ASCII name then they do 'similar' sorts of things."

### d) Ease of debugging through :-

- i. Comprehensive error reports including
  - \* Description of error, and parameters involved.
  - \* Whether it is a user error or an internal logic error. If an internal error it reports internal to which module.
  - \* In which routine the error occurred.
- ii. Timeous error checking, finding the error when and where it occurs, not many lines later.
- iii. Traceback of object and methods.
- iv. Polymorphism enables one to tell any object including all objects in a container to display themselves. e.g. Using the VAX/VMS debugger :-

```
call TELL(%VAL self, %VAL display)
```

## e) Robustness through :-

- i. Errors are allocated error numbers via the VMS message utility, thus making it easy to check and handle particular error classes.
- ii. Errors are signalled as a standard VMS condition, thus the VMS condition handling facilities can be used to trap and handle errors.

f) Practicality and cheapness - Ideally I would have preferred an operating system and language with all these features built in. One can buy products that partially address the above issues, but nothing that addresses all of them. To write a complete compiler and operating system was out of the question. Thus impoverishment of purse, dissatisfaction with current offerings, an overheated imagination, a bit of C code and a bit of assembler produced OopCdaisy, a object oriented program system piggy backed onto standard VAX C.

OopCdaisy is a suite of VAX C programs aimed at achieving the above ideals. As the Operating System and language on which OopCdaisy is piggy backed are fundamentally flawed in many of the respects mentioned above, only progress and not success is achieved.

The raster to vector conversion routine was built around OopCdaisy and much of its flavour has been carried through. Objects may be viewed as actors, and can be "told" to do things. This results in a slightly cute phrasing when talking about objects. For example, the phrases "Tell an object to do...", or "The object copies itself out to disk and then commits suicide."

It is well known that the debugging and maintenance phases are the most costly parts of the program development cycle. Experience with this program demonstrated that the time and effort in developing OopCdaisy was well rewarded in terms flexibility and reusability of code already developed and in terms of ease of debugging due to error checking and robustness

of the system. Further benefits from the objects developed for this program are expected in the future, as many of the objects are quite generally applicable.

### 2.1 What is object oriented programming?

I like to think of OOP in terms of an Actor language. All the computer is a stage, and all the objects are merely actors therein. Objects are independent entities with their own memory, and own characteristic way of doing things. These characteristic ways of doing things are called methods. Actors can tell another Actor to do something, (to use one of its methods.)

Objects can as Actors do, have children. And these children have similar memories and similar ways (methods). I.e. The child object inherits the memory, (but not the contents), and methods of the parent. The child can then be changed so as to be slightly different from the ancestor.

When we tell a person to do something, we are not concerned with how he does it, only with what is done.

The "Actor allegory" sets the scene, the paradigm in which we are working, but it must be admitted that the actuality doesn't fit it very well.

The actuality derives more from placing a level of indirection between the data and the use of the data. The major gains from this level of indirection are implementation hiding, Inheritability and Polymorphism. To explain what is meant by these three terms, consider the following two examples.

#### 2.1.1 Representation hiding, and Inheritability - an example.

As an illustrative example, consider points and lines in  $R^2$ . One can represent them in cartesian coordinates  $(x, y)$  or polar coordinates  $(r, \theta)$ . One can represent lines as  $y$ -intercept and slope  $(c, m)$  or parametrically as a quadruple  $(x_0, y_0, dx, dy)$  where

$x(t) = x_0 + dx * t$  and  $y(t) = y_0 + dy * t$ . But whatever the representation, we can speak about the distance between two points, the distance from the origin, the perpendicular distance between a point and a line, and the intersection point between two lines.

In a program dealing with points in  $R^2$ , such as a raster to vector conversion program, the logic of the program is concerned with points, lines, intersections and distances and could be programmed in any representation. Some representations are more convenient in some applications than others. Polar coordinates are very useful if you are mostly interested in angles between lines and distances from the origin, but add unwanted complexities when working with lines.

Central to OOPS is that the *internal representation of the data becomes separated from the use of the data*. Thus programs using an object are protected from the internal complexities of and changes in the implementation of an object.

Thus in the paradigm we would call the internally stored floating-point numbers the memory of the object, and the functions that calculate things about the object, the methods of the objects.

You may note in the discussion of representations above that in some respects the points and lines were similar. The point was represented by  $(x,y)$  and the line by  $(x_0,y_0,dx,dy)$ . Indeed  $(x_0,y_0)$  can be said to be the starting point of the line. All methods that could possibly apply to a point could equally well apply to the starting point of a line. So in this respect a line descends from a point, in that a line's "memory" is a superset of the "memory" of a point, and all the methods of the parent apply to the child. This process is called inheritance.

### 2.1.2 Polymorphism - An illustrative example.

Now consider a bag actor. A bag actor holds things. Anythings. Consider the display method. If you tell a line to display itself, it would call upon its display method to draw an appropriate line on the screen. If you tell a point to display itself it would call upon its display method to draw a dot in the right place on the screen. If you told a fluffy bunny to display itself, it would call upon its display method to draw a fluffy bunny on the screen. If you tell a bag to display itself, it would call upon its display method which would draw a picture of a bag on the screen, and then tell everything within the bag to draw itself. Thus if the bag contained a point, a line and three fluffy bunnies, you would get a bag, a point, a line and three fluffy bunnies on the screen. This process is called polymorphism, and is a marvellous labour saving device. Instead of having very intelligent plastic bags, that can draw a picture of everything, you just need ordinary dumb brown paper bags that can draw themselves and tell everything within them to draw themselves.

What if there was a bag inside the bag? No problem, the whole thing is quite recursive.

### 2.2 How does OopCdaisy implement objects?

So with the above mental picture in mind, we proceed to lose ourselves in detail.

First we must clear up two points :-

Actors do not multiprocess. In life one person can do something while another is still busy. On the stage and in OOPS, one actor holds a pose while the other plays his part.

The memory of an actor is just a group of ordinary computer variables.

Secondly we should answer the question...

### 2.2.1 What is OopCdaisy?

OopCdaisy is a set of C functions that maintain, manage and query a registry of object types and methods. The registry contains the following information on each object type :-

- a) The memory required to create an object.
- b) The ASCII name of the object type.
- c) The object's parent object type.
- d) The number objects of this type which are currently active.
- e) The number of objects of this type which have been created so far.

The registry also contains a list of all method names and a 2D array called the Virtual Method Table or VMT.

For every object type and for every method in the method name list there is a cell in the VMT that either contains nothing, or a pointer to a function that will execute the method on a object of that particular class.

### 2.2.2 What are OopCdaisy Objects and methods?

In OopCdaisy an object is a C structure, (the C equivalent of a Pascal record). The memory of the object/actor is the data stored in the structure. The methods of the objects are ordinary C functions.

### 2.2.3 The global conception of object methods.

A major point on which OopCdaisy deviates from most OOPS implementations is the idea that each method is global across the whole system. This is polymorphism taken to the extreme. Every object type has an entry in the Virtual Method Table for every method available to every object class in the entire system, whether ancestor or not. In Turbo Pascal 5.5 by Borland for instance, only those methods of a class and its direct ancestors appear in the VMT.

The rule governing OopCdaisy is :- "If two methods have the same ASCII name then they do 'similar' sorts of things." What are the implications of this?

- \* You can tell any object to do any method. If that object class does not have such a method, a non-fatal information grade signal called "NOMETHOD" is raised.
- \* You can tell a completely mixed bag of objects to do something.
- \* you are guaranteed that when this mixed bag of objects does something :-
  - the program is not going to crash because some object within the bag is does not know what to do.
  - all the objects within the bag will be doing more or less what you would expect.

The disadvantage of course is that the VMT must be large. In fact the maximum number of object classes times the maximum number of methods times the sizeof each slot.

#### 2.2.4 Dynamic vs Static instances of Objects.

One of the central activities engaged in by an OOPS programmer is to add objects into containers. This conflicts quite severely with a structured programmer's habit of declaring local variables. If you declare an object local to a procedure and then add it into a container created at a higher level, as soon as you exit the low level procedure there is a "garbage" object within your container! When the program goes into any lower level procedure, the stack grows and the "garbage" object is overwritten. If you have full control over your compiler you may conceivably implement your procedure termination code to include instructions to all local objects to remove themselves from all containers and then die tidily. (The usual termination code is just to drop the stack pointer by the size of the local variables.)

A simpler and possibly more useful approach is to place a global ban on local objects. If you want an object, you can declare a local pointer to an object, and then allocate the object dynamically on the heap. This is the approach OopCdaisy takes.

Of course it may happen that some objects created at a local level may be lost, i.e. allocated on the heap, but not accessible as there are no pointers to it. There are two views of what has happened :-

- \* Niklaus Wirth and J Gutknecht [Wirth & Gutknecht 1989] in their Oberon system would say the programmer no longer needs those objects anyway and then proceed to recover the memory automatically in a garbage collection phase. In fact the Oberon system assumes that a programmer is too error-prone and unreliable to be trusted when the programmer claims a data item is not needed.
- \* OopCdaisy agrees with Oberon that programmers are not to be trusted, but is more authoritarian and insists that the programmer has made a mistake. If the object still exists, the programmer probably still wants it, but has just forgotten where he put the thing. So to prompt the programmers faulty brain, OopCdaisy maintains an accounting system of objects created and deleted. It is the programmers responsibility to deallocate all created objects, but OopCdaisy will at least be so kind as to inform the programmer at any stage how many objects of each variety that he still needs to dispose of.

#### .5 How is a new class of objects created?

declare a class of objects one must declare a typedef of a struct, the first element being of the parent type. (If you speak Pascal, declare a record type.) For example, the type declaration for the points object is as follows :-

```
typedef
    struct
    {
        objects ancestor;
        double x, y;
    } points;
```

The objects type is defined in an OopCdaisy standard include file "oopDir:base.h" as :-

```
typedef
    struct
    {
        objectTypes type;
    } objects;
```

OopCdaisy identifies each class by a number that is an index into the class register and the Virtual Method Table. This number is stored in the class id variable, which for the point object type would be declared as follows :-

```
objectTypes pointType;
```

The person who implemented the point object would have stored these declarations in an include file e.g. "oopdir:2d.h". These declarations would be included into the users program and into the file which implements the class.

Before you can use a class of objects you must first inform OopCdaisy about the existence and nature of the class.

2.2.6 How is a new class of objects registered with OopCdaisy? Before we can use a class of objects, we have to inform the OopCdaisy system that this class exists, and what methods this class has. This is done by calling the class's registering function. The class registering function is part of the implementation of the class. For example, if you wish to use the point object class, you would call the :-

```
register_point();
```

function. The register\_point function typically would have been implemented like so :-

```
void register_point()
{
```

```

/* First tell OopCdaisy about the type we are registering,
how many bytes of memory each instance uses, which object
is its parent type. OopCdaisy finds a slot for the type
and returns the slot number in class id variable
pointType. OopCdaisy also copies all the parents methods
into the VMT for pointType. So all methods available to
the base type are also available to point!
*/
register_type( "point", &pointType, "base",
              sizeof( points));
/* Then OopCdaisy must be told all about additional methods
that are available to the pointType. The "init" methods
initialise a newly created object. Note that the init
method is already available to the base type, but by
registering init here we force OopCdaisy to use
point_init to init point objects. OopCdaisy returns the
slot number of the method in the method variables init
and distanceSq.
*/
register_method(
    pointType, "init", &init, &point_init);
register_method(
    pointType, "distanceSq", &distanceSq,
    &point_distanceSq);
.
.
.
}

```

### 2.2.7 How is an object created?

OopCdaisy provides an assembly language function called `new`, which one can call to allocate and initialise an instance of an object.

`New` is a function returning a pointer to an object. The caller must tell `new` which type of object to create by passing a class

id variable. The second parameter is a initialization method id variable, and the remaining parameters are parameters passed on to the initialization method.

```
objectPtr =
    new( objectType, method, p1, p2, p3, ..);
```

So to create an object, first declare a pointer to the object you want to create, then call new. For example, to create a point object :-

```
main()
(
    points * point;
    double x, y;
    .
    .
    .
    point = new( pointType, init, x, y);
    .
    .
)
```

What has happened here? C calls the assembly language function new(), which performs the following actions :-

- \* Looks up in the registry to find out how much space is required by a point object.
- \* Allocates that much space on the heap.
- \* Stores the object type in the first element of this space.
- \* Looks up, (in the virtual method table), the address of the point class's init method, and then calls the point class's init method with parameters x and y.

The init method stores x and y in the next two elements of the allocated space. If the chosen representation was polar coordinates, the init method could have converted the (x,y) to (r,θ) and then stored r and θ in the first two elements of the storage space.

### 2.2.8 How is an object used?

To find the distance between two objects one can either call the distance function directly :-

```
distanceSq = point_distanceSq( point1, point2);
```

This is called a static method invocation, as the method invocation is done at compile time and can not be changed thereafter.

Or :-

```
distanceSq = tell( point1, distanceSq, point2 );
```

This is called a virtual method invocation. What happens here is the assembler routine tell takes its first argument to be a pointer to an object. It then verifies that this pointer points to an accessible part of memory. And that there is a valid object at that location. From this location it takes the object type. Tell takes the second argument to be a method identifier.

This is where the flexibility arises. The decision as to which function is actually called is only made when the 'tell' function is executed, and NOT before.

Now we must be at least partially aware of several distinctions here :-

- 1) A method is a C function. The naming convention for a method is objectType\_methodName. Thus for example the stack push method is defined by a function called stack\_push.
- 2) A method id is an ordinary C variable which contains an index into the Virtual Method Table. Thus for example, tell finds the stack push method by using the contents of the variable push as an index into the Virtual Method Table. At that location tell finds the address of the function stack\_push.
- 3) The name of a method is an ASCII string, for example "push". Each slot in the VMT corresponds to a method name. Thus you can tell a Volkswagen to "ClimbTrees", but you will just get the signal NOMETHOD.

Thus tell finds the address of the function `point_distanceSq` by looking up in the Virtual Method Table at `VMT[ objectType][ method]` and passes the remaining parameters to this function.

Because I cannot build `OopCdaisy` into the compiler, these curious distinctions that are not normally forced upon ones attention, must be made. In commercial OOPS systems such as Turbo Pascal, these distinctions exist, but are hidden from the user. However, as I will see in the section on Turbo Pascal, these subtleties are not necessarily a weak point of `OopCdaisy`.

Now we have had a glimpse of `OopCdaisy` sufficient to continue the discussion. A closer look and more on the programming philosophy of `OopCdaisy` is given in the annotated example in Appendix 1.

### 2.3 Other `OopCdaisy` objects.

One of the main selling points of the OOPS approach is the ability to create truly flexible and reusable libraries of objects. So what does `OopCdaisy` offer?

Stacks with extensions to :-

- \* push and pop from the top and the bottom of the stack.
- \* Find objects on the stack.
- \* TellAll objects within the stack to do something.
- \* Rotate the stack until a chosen object is on top.
- \* Concatenate stacks.

The next major class is the direct descendant of the stack called a container. Thus all methods applicable to stacks apply to containers as well. The main feature of containers is that they can have index objects attached to them. Thus you can create an index object and attach it to a container. The index object can be marched up and down the container, objects can be lifted out of the container via the index, objects inserted into the container before or after the index, objects can be found via the index

etc. The main point about using an index object is it is safe. The programming to move up and down is done, debugged, and overruns are guarded against. Containers are aware of which indices are attached. Thus if several indices are attached to a container, and an object is removed from the container, all indices are checked to see if any index is left pointing at a non-existing object. If a container is deallocated, the container checks that all index objects are detached.

Two dimensional objects such as points, lines, arcs and circles are implemented with many useful methods such as distance between, intersection, etc. Half-planes or linear inequalities are implemented as being "to-the-left-of" directed lines.

Other object classes include LIFO lists, LISP style lists, and binary trees.

#### 2.4 OOPS in the literature.

In theory I should give a large and scholarly literature review of all sources which influenced the design of OopCdaisy. Unfortunately in practice I find this extremely difficult. Many of the ideas were accumulated during a decade of omnivorous reading, the vast majority of this time I had no intention of writing such a system. Thus I did not collect the references. OopCdaisy just "happened" because I urgently needed decent list handling facilities, and I didn't want to have to keep rewriting these facilities. While I can "after the fact" produce a list of references from the library, it is somewhat unfair in the sense that it wasn't those papers that lead to OopCdaisy.

However, some sources such as Turbo Pascal 5.5 from Borland [Borland 1989], Smalltalk from Xerox [Mevel & Gueguen 1987], [Goldberg 1984] and the Oberon system [Wirth & Gutknecht 1989] can be identified and discussed, and comparisons can be made with other OOPS languages such as Eiffel [Meyer 1990].

### 2.4.1 Turbo Pascal 5.5

A major prompt in starting work on OOPS was the arrival of Turbo Pascal 5.5 from Borland. Borland had grafted OOPS fairly neatly onto their Pascal compiler. However I consider Turbo Pascal 5.5 to be severely limited in the scope of its OOPS on the following grounds :-

- a) You cannot "tell" a container full of objects to "do" a certain "method", as Turbo Pascal 5.5 does not have any "method" variables.
- b) Turbo Pascal strict typing can interfere with object inheritance. Suppose you had a "line" object which was a descendant of a graphical "figures" object. A common method would be "intersection" which would calculate the intersection point between the "self" object and some "other" figure object passed as a parameter. The intersection method would decide what type the "other" figure was and uses the appropriate method, (say `lineIntersectsLine` or `lineIntersectsCircle`), to calculate the intersection point. Although one can use the `typeof` function, (basically the Virtual Method Table pointer), to decide which descendant of "figures" "other" is, Turbo pascal strict typing implies that there is no way of upgrading the "other" variable to the appropriate descendant type. In `OopCdaisy` the problem doesn't arise since C allows type conversions between pointers to incompatible types. (Although not type conversion between incompatible types themselves.) `OopCdaisy` safeguards the whole process by placing a type guard check at the start of every low-level method.
- c) Neither Turbo Pascal nor `OopCdaisy` allows multiple inheritance. The absolute simplicity of single inheritance storage schemes militates against multiple inheritance.
- d) Neither Turbo Pascal nor `OopCdaisy` go far enough. For improved reliability, all data types should be objects. For example one can put an integer into a container, but as the

integer is not an object, the polymorphism is totally lost and a program crash is likely to occur.

#### 2.4.2 Oberon

The other source of ideas for OopCdaisy was Niklaus Wirth's Oberon system. From Oberon came the idea that the system should have the duty of memory policing. Oberon does garbage collection based on information in the object registry. OopCdaisy merely does memory accounting, thus enabling the programmer to find out which objects have not been deallocated.

#### 2.4.3 Smalltalk.

To quote the BYTE may 1985 issue [Webster 1985], 'the influence of SmallTalk-80...has become just about legendary.' Indeed I regret that most of the influence of Smalltalk on OopCdaisy was via the BYTE August 1981 issue on Smalltalk, written some 9 years earlier than OopCdaisy. Thus Smalltalk's influence was more on the basis of half-remembered legend than on fact. The 'actor' paradigm came from Smalltalk. Had I remembered more I would surely have made OopCdaisy self-consistent in the sense of making the object types Smalltalk-like metaclass objects. Indeed the implementation of OopCdaisy registry is entirely in the classical pre-OOPS C programming style.

#### 2.4.4 Eiffel.

OopCdaisy would never have occurred if Eiffel and the Eiffel libraries had been available at the time of writing OopCdaisy. Eiffel as described in [Meyer 1990], achieves all the goals I set out for OopCdaisy. Particular points that are in Eiffels favour are :-

- \* Eiffel is a pure OOP system. All data types are objects. [Korson & Mcgregor 1990]
- \* Encapsulation of data is enforced. Access to an objects private variables can only be made via the objects methods.
- \* Eiffel is strongly typed, (statically typed), to aid the production of correct systems.

- \* Generic classes can be declared. I.e. class definitions can be parametrized. For example the generic class `list[ T]` are lists of type T. One can then declare a `list[ integer]` or a `list[ POPUP_MENU]` or whatever.
- \* Infix and prefix operator functions for easy reading of expressions.
- \* Abstract classes can be declared in the form of deferred classes. (OopCdaisy has this facility as a common ancestor type with unimplemented methods)
- \* Reliable Software components using a contractual model. The Eiffel libraries are a collection of software components that provide a contractually bound service to client routines. If the client undertakes to meet certain preconditions as stated in and automatically checked via assertion statements, the component undertakes to perform a specified task resulting in post-conditions again automatically checked by assertion statements.
- \* Classes may be grouped into clusters or frames. For example there is a suite of objects designed to perform windowing and menu manipulations that can be specialised via the inheritance mechanism for individual applications.
- \* Reliability through :-
  - Persistence of objects "beyond the programmatic grave", removing I/O related problems.
  - Automatic garbage collection for complex dynamic data structures.
  - Disciplined exception handling providing graceful termination and recovery.
- \* Availability of CASE tools.

With reference to the problem with strictly typed OOP languages that was mentioned in the discussion of Turbo Pascal, [Meyer 1990] is inconclusive. His discussion on the subject of static versus dynamic typing favours static typing "whenever possible". However the rest of the text makes it unclear whether there exists a means of guardedly performing a type conversion, or a

bug in the current compiler makes it is possible to bypass static typing!

## 2.5 OOPS in the future.

The rapid growth of OOPS is not only due to the buzz word driven nature of the computer industry, but due to the virtues of the OOP listed at the start of the chapter.

However, I feel OopCdaisy has some valid points to make about OOPS in the future.

- 1) Programs based on OOP systems like OopCdaisy, Oberon and Eiffel are intrinsically more reliable than standard programming systems.
- 2) OopCdaisy in its implementation makes it clear that an OOPS system is in fact a Data-Base Management System. The database being managed is that of data fields and methods. Current OOPS offerings, (OopCdaisy included), are in the same infantile stage of development as the early hierarchical or network database systems. I think one can say that the dust has settled over the database conflict and relational databases have emerged as the victor. I would strongly advocate that someone, somewhere initiates a research project into implementing an OOPS system which is based on a true relational database. Indeed partial approaches have already been made in the limited sense of a Data Dictionary. For example the DEC Common Data Dictionary. [DEC CDD]
- 3) Experience gained on the OopCdaisy project would suggest the Eiffel technology is fundamentally sound.

### 3. The first phase of raster to vector conversion - Producing Spaghetti.

The design of this algorithm was constrained by the need to vectorize large images, typically 7000 by 8000 pixel images, on a computer with a relatively small amount of memory, (4 megabytes).

#### 3.1 Edges, vexils, strings, nodes, knots, spaghetti - some definitions

The real world is "imaged" by a scanning device. The output of the scanning device is an image. The image is classified into different classes, resulting in a classification. A classification is made up of regions. The regions are bordered by edges. Each region is made up of pixels. The edges run between pixels. Each pixel is square. The border of a square pixel is made up by four vexils.

(The name "pixel" is commonly used and derives from "PIcture X ELeMent", and other common terms such as "voxel" from "VOLUME X ELeMent", and "nixel" from "MIXture pixEL" are well known. However, "vexil" is, admittedly, my own term arising from "VEctor X picture eLeMent".

A string is a list of vexils. Hence an edge of a region is a string of vexils.

Sometimes three or four regions meet at the same point. This point is called a node. All strings start at a node and end at a node. No string goes through a node.

If a region is surrounded by one and only one other class, (think of an island in an ocean), then the string of vexils bounding this region must meet itself. This meeting point also called a node, but a special kind of node called a knot.

In GIS parlance, vector strings without the associated topological information are termed "spaghetti".

3.2 An overview of how the Spaghetti Machine works.

The program shifts a 2 by 2 window from left to right over the image looking for edges. Edges found are joined into strings. A container called *dangleList* holds all incomplete strings which are dangling down from north to south. An index attached to the *dangleList* moves with the window from one dangling string to the next.

3.3 Block classes.

Spaghetti inspects each 2 by 2 block of pixels, separates the blocks into classes based on the edges within the block and not on the actual pixel values.

Thus for example, the block:-

12 12

12 13

contains one 'Southward' vertical edge and one 'Eastward' horizontal edge and is handled in exactly the same way, by the same bit of code, as the block :-

98 98

98 21

as the edges are the same even if the pixel values are different.

Whereas the block :-

13 12

12 12

is handled in a different manner to both blocks above.

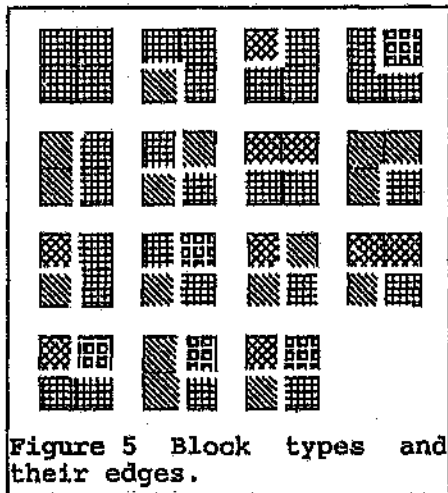


Figure 5 Block types and their edges.

Here is a complete list of all possible block varieties...

```
xx xx yx yy zx zz xw zk zz xw yw zw zw
xx yx yx yx xx xx xx yx yx yx yx xx yx
```

### 3.4 Starting up the Spaghetti Machine.

A buffer is created in memory that is at least 2 lines long, and the lines are 2 pixels longer than the image lines.

The first few lines of the image are read into the buffer starting at the second line of the buffer, with the first pixel of image data starting at the second pixel of the buffer. The first line of the buffer are set to theWorld value. The first and last pixels of each line in the buffer is set to theWorld. ("theWorld" is a nominal class value for the scene beyond the image)

A container object called the dangleList is created and initialised, and an index object called the dangleDex is attached to the container. An Index object acts as an index into the container it is attached to, and can be moved around the container. Objects within the container can be accessed via the index.

The buffer and the container is passed to the Spaghetti Machine which vector'es each buffer.

The last line of the buffer is copied to the first line of the buffer, and the next group of lines are read from the image into the buffer starting at the second line second pixel. If this is the last bufferfull to be processed the last line of the buffer is set to theWorld. The buffer is fed to the Spaghetti Machine and this process is repeated until the image has been completely processed.

### 3.5 The Spaghetti Machine.

Starting at the second line, second pixel of the buffer, scanning as one reads, from west to east then north to south.

For each line in the buffer :-

Set the `dangleDex` to point to the front of the `dangleList`.  
For each pixel in the line, (starting at the second pixel), check each 2 by 2 window for edges. (Place the bottom left hand corner of the window over the pixel.)

The very first edge scanned has to be in a

`yy`

`yx`, class block. Thus a new string is created. The horizontal and vertical vexil is placed in the string. As the string dangles southwards, the string is pushed onto the `dangleList` just before the point indexed by the `dangleDex`. As the string dangles eastwards, a pointer to the string is placed in the variable `eastString`.

If the next edge scanned was of the form

`zz`

`xx`, then the horizontal vexil would be attached to the string pointed to by `eastString`.

If a window of the form

`xx`

`yx` is scanned, then the horizontal and vertical edges are attached to the string pointed to by `eastString`, and the string is pushed onto the `dangleList` just before `dangleDex`.

If a window of the form

`yx`

`yx` is scanned, the `dangleDex` must be pointing to a string dangling from the north. So the vertical edge is placed on the string, and `dangleDex` is pushed forward.

If a window of the form

zx

xx is scanned, then the string dangling from the north is removed from the dangleList, and is knotted to the string coming from the west, which is pointed to by eastString.

If a window of the form

xy

yx is scanned, then if  $x > y$  this is regarded as a corridor of x's in a field of y's. (Otherwise the other way round)

If a window of the form

zx

yx is scanned, then a node is created, the string dangling from the north, and the string coming from the west, and a newly created string dangling south is attached to the node. The string from the north is replaced on the dangleList by the newly created string going south. The dangleDex is advanced.

In a similar manner, every block class is handled. Note :- when the strings are told to attach themselves to a node, they check if both ends are now attached. If so, they smooth themselves, then move themselves out of memory onto the disk.

#### 4. Smoothing.

The output of most raster to vector conversion routines are strings of voxels outlining the pixels. Unfortunately the corners of the pixels create jagged edges to lines that are not horizontal or vertical. (See Figure 4 for example.) These jagged artifacts are unwanted for two reasons,

- 1) they produce extra and unnecessary vectors that require extra storage manipulation etc.

- 2) one knows that the natural scenes that were imaged did not contain jagged lines.

Why do we not simply recover the original picture? The rasterizing process loses information, thus there is not sufficient information within the raster picture to recover the original picture. The best we can get is an approximation.

Vectorization is thus an underdetermined problem. The raster image only provides constraints on which vector images are possible.

#### 4.1 Exactness.

It would be reasonable to expect the result of a vectorization procedure to have the property that if the polygons were rasterized using the same grid as the original image, exactly the same pixels would result. I define vectorization procedures which have this property as being exact.

However, it should be noted that the requirement that vectorization must be exact does not determine the result.

#### 4.2 How does one find an exact, near-optimal vector representation of a raster image? - an overview.

It turns out that the boundaries of the edge pixels of each region contain all the information on the exactness constraint. So the first step is to outline the boundary pixels of each region.

This step is performed by "The Spaghetti Machine" described earlier. As each string of voxils is completed it is handed over to be smoothed. These strings of voxils are the boundary vectors of pixels so can only run north, south, east or west.

How can one find the least number of vectors to replace this list of voxils? The easiest way is as follows :-

- a) Start at the beginning of such a string. One can always fit an exact line through a single vexil.
- b) Scan down the string while you can still fit an exact line through all the vexils scanned so far.
- c) Shift one vexil back.
- d) Output any exact line that fits through the vexils scanned.
- e) Delete all but one of the vexils scanned.
- f) Repeat until finished string.
- g) Join the lines at the points of intersection.

Note that in step d) any exact line can be output. This is the point where the algorithm is only near-optimal not optimal. It is locally optimal, but to be globally optimal the choice of exact line to use would be influenced by the requirements of other lines. More on this aspect later.

So the second step is to parse these lists of "vexils" into possible lines. For example a string of vexils going north, east, north, east, south, east, cannot possibly be part of a straight line. The parsing is performed by a finite automaton that recognises longest possible lines. The possible lines are then checked for exactness, and a string of the longest possible exact lines is output.

#### 4.3 Parse the Spaghetti please.

The input to the smoothing algorithm is a list of vexils. Each vexil is essentially a direction, (one of north, south, east or west), and a positive integer distance, (1 pixel, 2 pixels, 3 pixels,...). Through some strings of vexils it is possible to plot an "exact" line and through other strings it is not.

For example, a vector moving north-east could replace the string in Figure 6a. On the other hand, no single line could represent the string in

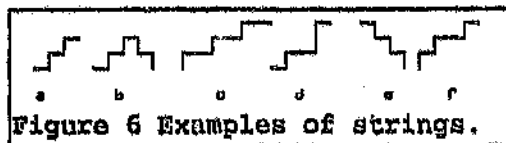


Figure 6 Examples of strings.

Figure 6b. If you did use a single line in such a case it would not be an exact representation in the sense that rasterizing the output would result in a different image. Figure 6c is an example of a valid line going east of north-east. Figure 6d is an example of what would appear to be a line going north-east, yet the exactness constraint cannot be satisfied by any one line. Thus two lines are needed here. Note that this case can be recognised as a line that started going east of north-east and then changed to north of north-east. Figure 6e is an example of a simple line going south-east. Simple lines are strings which move purely north-east or south-east or south-west or north-west, and need not be checked further for exactness.

Complex lines such as Figure 6f, unlike simple lines, must be checked more closely for exactness. Figure 7b gives an example of a string whose validity cannot be determined by parsing alone.

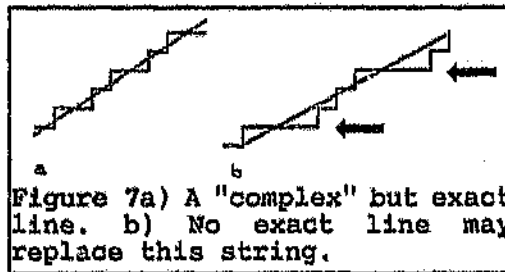


Figure 7a) A "complex" but exact line. b) No exact line may replace this string.

#### 4.3.1 From string of vexils to a character string.

For the purposes of parsing the string of vexils into possible lines, it is convenient to convert the string of vexils into a string of characters. As each vexil can only go in one of four directions, and vexils of length one are very common and important, each vexil is mapped onto the character set {'n', 'e', 's', 'w', 'N', 'E', 'S', 'W'}. Where a vexil of length one will be represented by a lowercase character, {'n', 'e', 's', 'w'} and a vexil of length two or greater will be represented by an uppercase {'N', 'E', 'S', 'W'}.

Thus for example, the string in Figure 7b can be converted into the string of characters "enEnenenEnen".

Of course this translation depends on which way down the string we travel, travelling the opposite way along Figure 7b gives us "SWSvibSWSWSw".

#### 4.3.2 The language of worms.

The process of converting strings of vexils into strings of characters has converted the problem of recognising lines into recognising a formal language.

The alphabet of the language is the set {'n', 'e', 's', 'w', 'N', 'E', 'S', 'W'}. Naturally the range of lines that can be identified would be larger if the alphabet was enlarged. For example, let 'U'

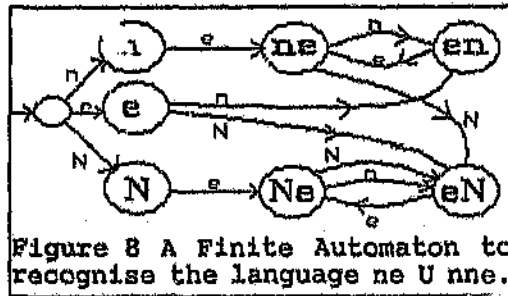


Figure 8 A Finite Automaton to recognise the language  $ne \cup ne$ .

be for upward, (northward), vexils of length 2 and 'N' be for vexils with length 3 or greater. Similarly let 'L' correspond to westward vexils and 'R' to eastwards vexils and 'D' to southward vexils of length 2. However the language becomes correspondingly more complex, so such languages won't be discussed here.

The language of lines can expressed as a Regular Expression. The definition of a Regular Expression (RE) is :-

- 1) Any finite set of finite length strings of characters from the alphabet.
- 2) If L is an RE, then  $L^*$ , the set consisting of the empty string and all finite-length strings formed by concatenating words in L, is also an RE.
- 3) If A and B are RE's then A union B is also an RE.
- 4) If A and B are RE's then  $AB = \{x \mid x = uv, u \in A \text{ and } v \in B\}$ , is also an RE.

It can be shown [Hopcroft & Ullman 1969], that any RE can be recognised by a Finite Automata.

Denote  $A^* U A$  by  $A^+$

The language of simple lines is defined :-

The set of all ne lines  $ne = \{\epsilon, 'e'\} \{ 'ne' \}^* \{\epsilon, 'n'\}$

The set of all se lines  $se = \{\epsilon, 'e'\} \{ 'se' \}^* \{\epsilon, 's'\}$

The set of all nw lines  $nw = \{\epsilon, 'w'\} \{ 'nw' \}^* \{\epsilon, 'n'\}$

The set of all sw lines  $sw = \{\epsilon, 'w'\} \{ 'sw' \}^* \{\epsilon, 's'\}$

The set of all simple lines is  $ne U se U nw U sw$ . Simple lines are easy to handle as all simple strings can be exactly represented by a single line. Only the end two vexils need be considered when calculating the representative line.

The language of complex lines are defined :-

The set of all nne lines =  $(\{\epsilon, 'e'\} \{ 'Ne' \}^* \{\epsilon, 'n', 'N'\} U (\{\epsilon, 'n', 'N'\} \{ 'eN' \}^* \{\epsilon, 'e'\}))$

The set of all ene lines =  $(\{\epsilon, 'e', 'E'\} \{ 'nE' \}^* \{\epsilon, 'n'\} U (\{\epsilon, 'n'\} \{ 'En' \}^* \{\epsilon, 'e', 'E'\}))$

The set of all nnw lines =  $(\{\epsilon, 'w'\} \{ 'Nw' \}^* \{\epsilon, 'n', 'N'\} U (\{\epsilon, 'n', 'N'\} \{ 'wN' \}^* \{\epsilon, 'w'\}))$

The set of all wnw lines =  $(\{\epsilon, 'w', 'W'\} \{ 'nW' \}^* \{\epsilon, 'n'\} U (\{\epsilon, 'n'\} \{ 'Wn' \}^* \{\epsilon, 'w', 'W'\}))$

The set of all sse lines =  $(\{\epsilon, 'e'\} \{ 'Se' \}^* \{\epsilon, 's', 'S'\} U (\{\epsilon, 's', 'S'\} \{ 'eS' \}^* \{\epsilon, 'e'\}))$

The set of all ese lines =  $(\{\epsilon, 'e', 'E'\} \{ 'sE' \}^* \{\epsilon, 's'\} U (\{\epsilon, 's'\} \{ 'Es' \}^* \{\epsilon, 'e', 'E'\}))$

The set of all ssw lines =  $(\{\epsilon, 'w'\} \{ 'Sw' \}^* \{\epsilon, 's', 'S'\} U (\{\epsilon, 's', 'S'\} \{ 'wS' \}^* \{\epsilon, 'w'\}))$

The set of all wsw lines =  $(\{\epsilon, 'w', 'W'\} \{ 'sW' \}^* \{\epsilon, 's'\} U (\{\epsilon, 's'\} \{ 'Ws' \}^* \{\epsilon, 'w', 'W'\}))$

The set of all complex lines =

$nne U ene U ese U sse U ssw U wsw U wnw U nnw$

Not all complex strings in the complex language can be represented by exactly by a single line. For example the string

'nenenenenEnEnEnEnEnEnEnenenene' is fairly obviously three strings, a simple ne string followed by a ene string followed by another simple ne string. Thus a more subtle algorithm is needed to find whether a string can be exactly represented by a line. However, the efforts in parsing have not been wasted, as the next algorithm requires other information that only parsing can give it. Furthermore, all simple lines are identified by parsing and thus do not need the computationally more expensive algorithm.

4.4 Line Space.

As with many mathematical problems, the problem of finding an exact line is made easier by looking at the dual space.

Consider a line approximating the boundary between two regions of pixels with different classes. Suppose the class on the left of the line is A and on the right B. To be an exact representation, less than 50% of each class B pixel must be on the left of the line and less than 50% of each class A pixel must be on the right of the line.

The dual space of the problem is the space of all lines. If a line is represented by the equation  $y=mx+c$  then, for each point in  $R^2$   $(c,m)$  there is a corresponding line  $y=mx+c$ .

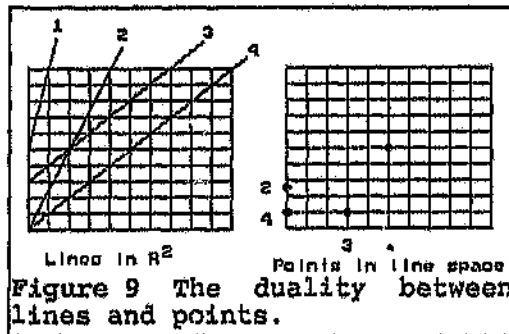


Figure 9 The duality between lines and points.

Thus the exactness requirement places a set of constraints on which lines are acceptable, ie defines a region within the line space.

#### 4.4.1 Threading the vexils.

The first constraint that exactness places is that the vexils must be threaded. That is to say, the line must cross each vexil.

Consider the case in Figure 10, it is clear that there cannot exist a line which does not thread the vexils and is exact. If more than 50% of pixel 2 is below the line, then less than 50% of pixel 3 is below the line, unless the line threads the vexils.

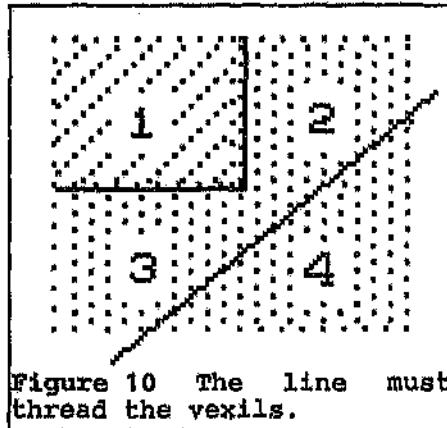


Figure 10 The line must thread the vexils.

Let the coordinates of the string separating the two regions in Figure 10 be  $((1,1), (2,1), (2,2))$ .

If the line that must represent this boundary is described by the equation  $y=mx+c$ , then the constraint that the line threads the vexils can be stated as :-

$$\begin{aligned} 1 &> m*1+c \\ 1 &< m*2+c \\ 2 &< m*2+c \end{aligned}$$

Or with a little algebra :-

$$\begin{aligned} m &< -c + 1 \\ m &> -c/2 + 1/2 \\ m &< -c/2 + 1 \end{aligned}$$

These constraints are illustrated in Figure 11. It should be noted that the feasible region, the unshaded area, is unbounded.

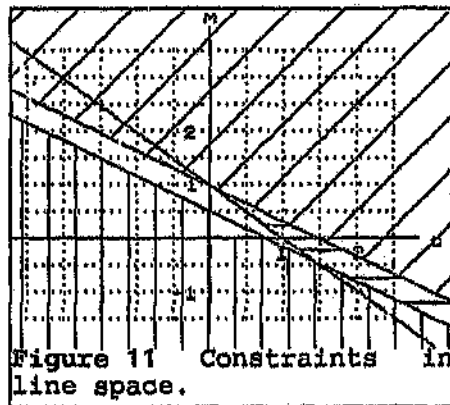


Figure 11 Constraints in line space.

4.4.2 Missing the point.

Consider the string, ((1,1), (1,2), (2,2), (2,3), (3,3), (3,4), (8,4)), shown in Figure 12. Clearly the line in Figure 12 threads the vexils, but the last pixel is entirely on the wrong side of the line.

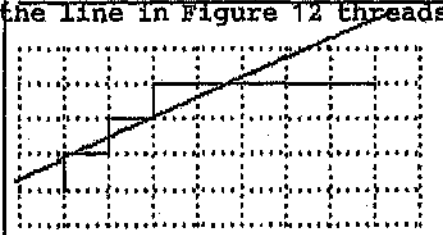


Figure 12 This string cannot be represented exactly by any one line.

Thus a further constraint is required to guarantee that pixels at the points of vexils are not on the wrong side of the line.

Figure 13 is a enlarged view of Figure 12 about the end point (8,4). The constraint that at least 50% of the area of the pixel at (7.5,4.5) is above the line can be formulated as follows.

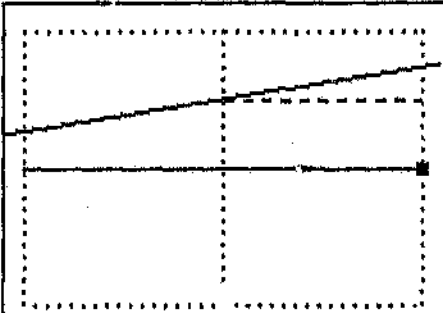


Figure 13 The line must be pinned to the end of the vexil.

The area of the rectangle below the dashed line is  $m(x-1)+c-y$  and the area of the triangle between the line and the dashed line is  $\frac{1}{2}m$ , thus the constraint is at point (8,4) is :-

$$7m+c-y+\frac{1}{2}m < 0.5$$

4.4.3 Putting all the constraints together.

The constraints define a feasible region in line space. If the constraints of threading the vexils and pinning the line to the points are applied at every vexil as we scan down the string, then as we scan down the string the feasible region shrinks. If at some point the feasible region disappears, then the string up to but not including the last vexil scanned can be represented exactly by a line. Indeed, any point within the feasible region represents an exact line.

#### 4.4.4 Which feasible line? Global versus Near-optimal representations.

As there are an infinite number of exact lines, we are free to choose. This however is the point at which this program becomes near-optimal, not optimal. An optimal program would choose each line so that the least number of lines would be used. This would require a global knowledge of the string, and considerably more time and effort.

A simple, easy to calculate, but reasonably effective choice is the centroid of the feasible region. As the feasible region is convex, the centroid of the feasible region always lies within the region.

#### 4.4.5 Keeping track of the constraints.

The crux of implementing this smoothing algorithm lies in evaluating and tracking the constraints set up by each vexil in an efficient manner, or this entire approach would become computationally untenable.

All the constraints are linear inequalities. These inequalities thus describe half-planes in line space. For programming purposes a linear inequality may be viewed as a directed line, with the feasible region being on the left of the line. This enables one to use the standard line object.

After two vexils the feasible region is a bounded polygonal region. Thus all that is needed is to maintain a list of the vertices of the polygon.

A region is convex if it is the intersection of half-planes. Clearly the feasible region is convex. This greatly speeds the chopping off of unfeasible vertices. Indeed, if any constraint is introduced that removes the convexity property, (e.g. pinning constraints to improve the optimality), the problem becomes unmanageable.

4.5 Why was parsing necessary if line space constraints give you everything?

For two reasons. Simple lines are quite common, and the parsing process is a lot more rapid than evaluating the line space constraints. More importantly, one cannot write down the constraints without information such as the sign of the slope of the line, and whether the line starts above or below the first point in the string. This information is readily obtained from the parser.

Not all lines are worth smoothing. For example, a box is never worth smoothing. Short lines of 4 vertices can be smoothed easily.

### 5. Linking the Lines.

As they are found, the lines are placed in a container. When lines have been found, the first and the last vertices are placed in at the front and the back of the container. These are 'ties' to the node points. The container is then scanned, and the start point, every intersection point between successive lines, and the end point is output to disk.

#### 5.1 Special cases.

The exactness constraints guarantee that lines are exact as far as the string goes. If the line projects beyond the end of both strings before it intersects the next line, then exactness is not guaranteed.

To handle this problem a simple test is needed to identify such cases.

5.1.1 When a Link-point is Else When.

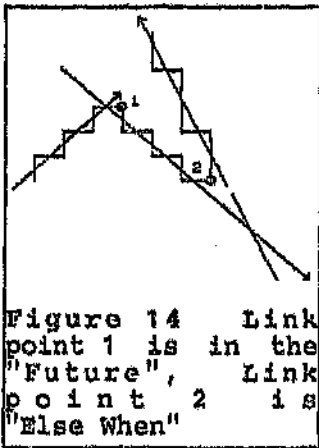


Figure 14 Link point 1 is in the "Future", Link point 2 is "Else When"

The heading of this section is the only simple and intuitive description of this test I can think of, hence I will have to settle for a technical description. Indeed the test itself was derived from inspection of all possible cases.

The link-point is the point which joins the two strings being approximated.

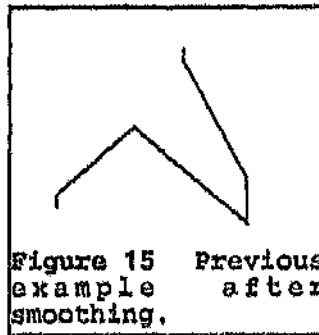


Figure 15 Previous example after smoothing.

The join-point is the point where the two approximating lines intersect. (Normally this point would be output from the program).

Make an affine transformation of the coordinate system to a new non-orthogonal coordinate system which has the join-point as its origin and the two approximating lines as its axes.

Calculate the coordinates of the link-point in the new coordinate system. If the link-point is in quadrant 2 or 4 of the new coordinate system then it is else when, and the lines extend beyond the end of both strings.

In the unlikely event you should wonder where the name "Else When" comes from, it comes from Special Relativity, where space-time is divided into three regions :-

Past :- Those points in space-time that could causally affect a particle at the origin.

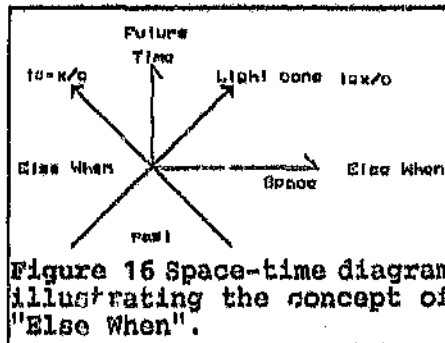


Figure 16 Space-time diagram illustrating the concept of "Else When".

- Future :- Those points in space-time that could be causally affected by a particle at the origin.
- Else When :- Those points of space-time that cannot causally affect or be affected by a particle at the origin.

The exactness constraint can only be violated in the "Else When" case, but to actually calculate when the exactness constraint is violated is tedious in the extreme, but does not pose any technical or computational difficulties.

If the link-point is Else When then the following algorithm is used :-

If exactness is not violated Then  
    output the join point

Else

    If the join point is closer to the first vexil of the second string than the last vexil of the first string Then  
        output the intersection of the first line and the last vexil of the first string.

        output the intersection of the second line and the last vexil of the first string.

    Else

        output the intersection second line and the first vexil of the second string.

        output the intersection of the first line and the first vexil of the second string.

5.1.2 Parallel lines.

As an extreme case of Else When, the two lines may in fact be parallel, thus there is no join point. In this event, the lines can either be parallel or anti-parallel.

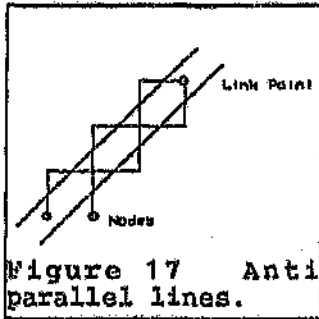


Figure 17 Anti parallel lines.

If as in Figure 17, the lines are anti-parallel then the two lines are joined via three points. The intersection of the first line with the last vexil of its corresponding

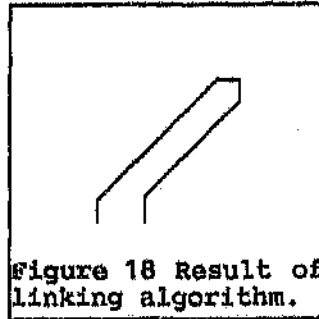


Figure 18 Result of linking algorithm.

string, the link-point that joined the original two strings, and the intersection of the first vexil of the second string with its corresponding line. The link-point is added to prevent too much of the pixel on the end being cut off.

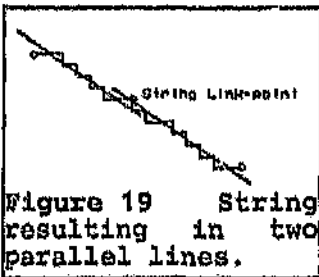


Figure 19 String resulting in two parallel lines.

If as in Figure 19, the lines are parallel, then the link-point joining the two original strings is not added as the intermediate pixel is safe in this case.

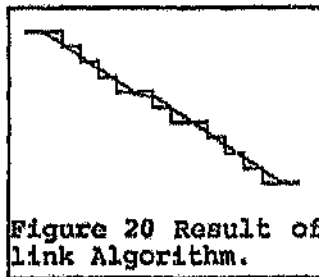
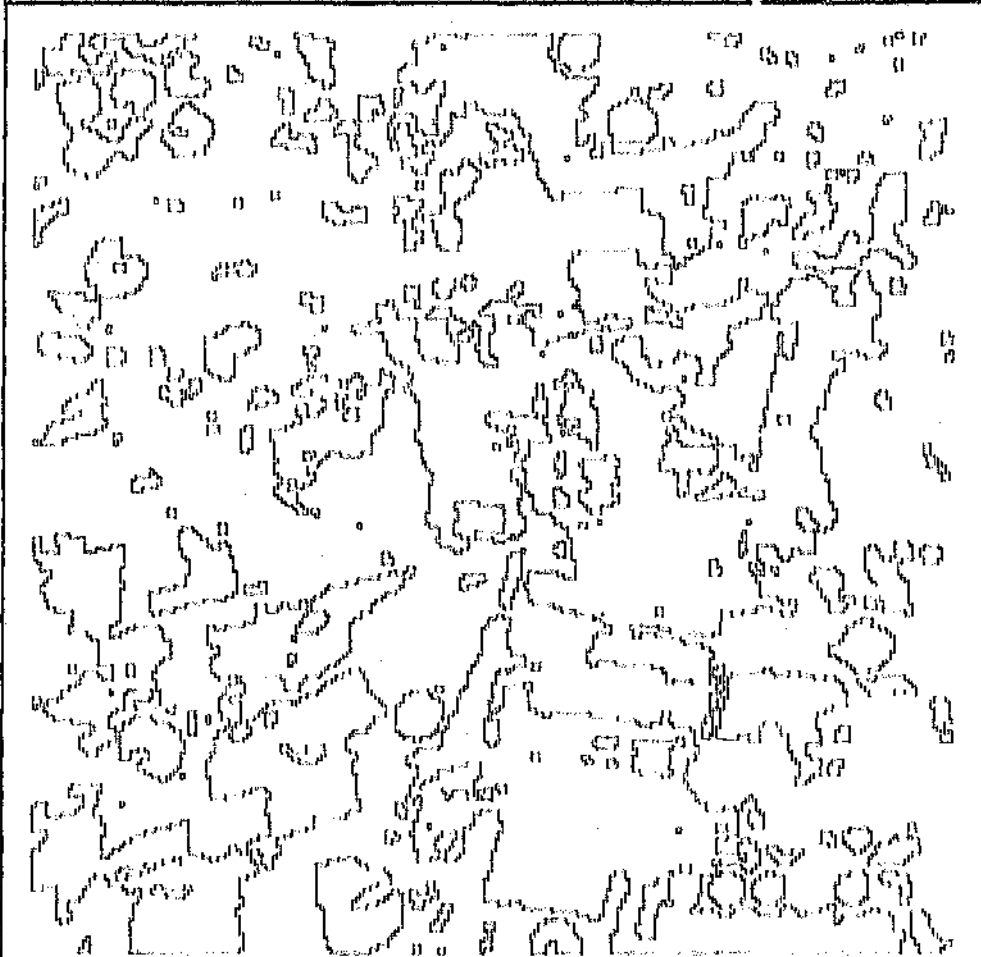


Figure 20 Result of link Algorithm.

this case.

**6. Examples and Discussion.**

How does this program work in practice? Figure 21 is the result of applying the Spaghetti Machine to a 256 by 256 pixel classified scene, and not smoothing. There are 204 strings containing a total of 5125 vertices.



**Figure 21** The result of applying the Spaghetti Machine to a 256 by 256 pixel classification. (Unsmoothed)

The alternate approach to the algorithm developed here is the thinning algorithm used by ARC/INFO GIS system. Here follows a pseudo-code description of this algorithm :-

```

const
    tolerance = 0.707;
procedure thin( startPoint, endPoint)
begin
    construct a line from the startPoint to the endPoint;
    Find the point on the string between startPoint and
    endPoint that is furthest from the constructed line;
    if distance from line to furthest point < tolerance
    then
        output startPoint;
    else
        begin
            thin( startPoint, furthestPoint);
            thin( furthestPoint, endPoint);
        end;
    end;
end;
begin {main program}
    while more strings do
        begin
            read strin
            thin( startPoint, lastPoint);
            output lastPoint;
        end;
end.

```

Figure 22 is the result of applying the standard ARC/INFO thinning algorithm, with the tolerance set to 0.707. There are 204 strings containing a total of 3401 vertices.

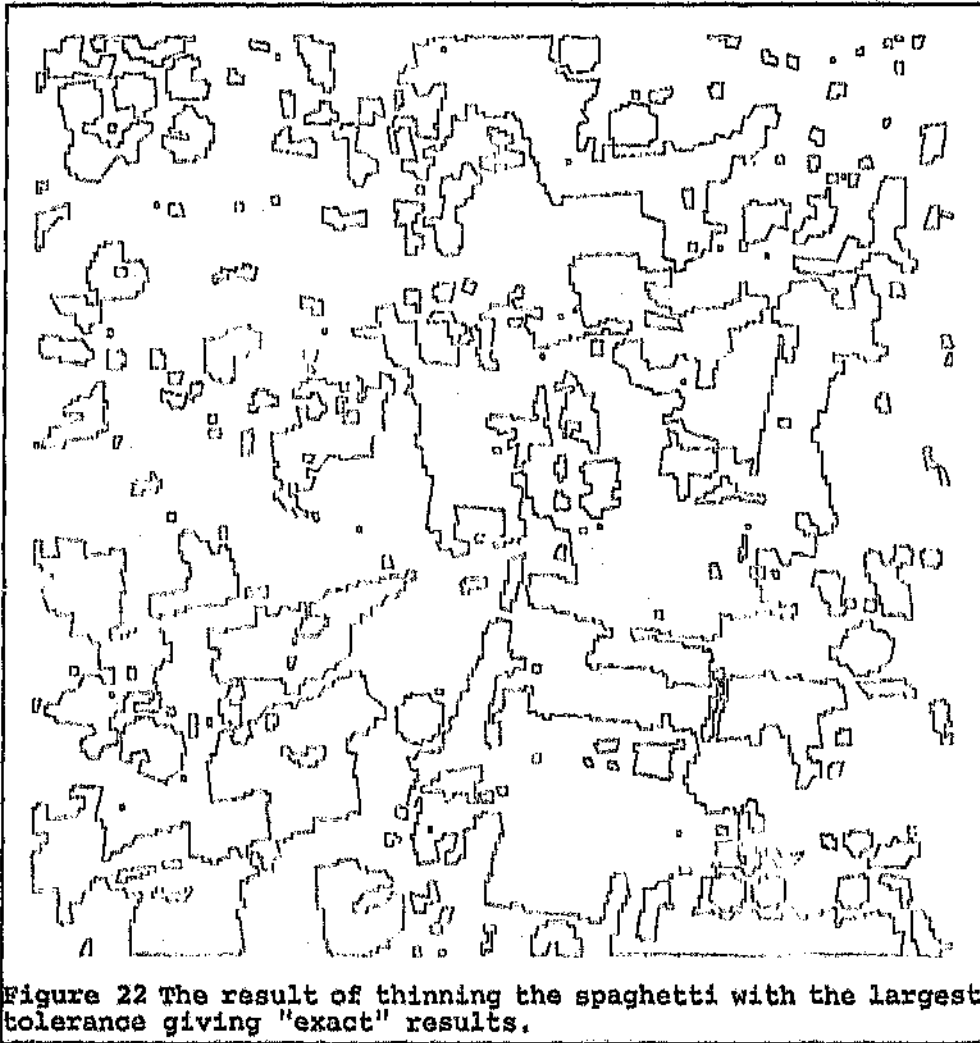


Figure 22 The result of thinning the spaghetti with the largest tolerance giving "exact" results.

Figure 23 is the result of using the smoothing algorithm described in Chapters 4 and 5. There are 204 strings containing a total of 2766 vertices.

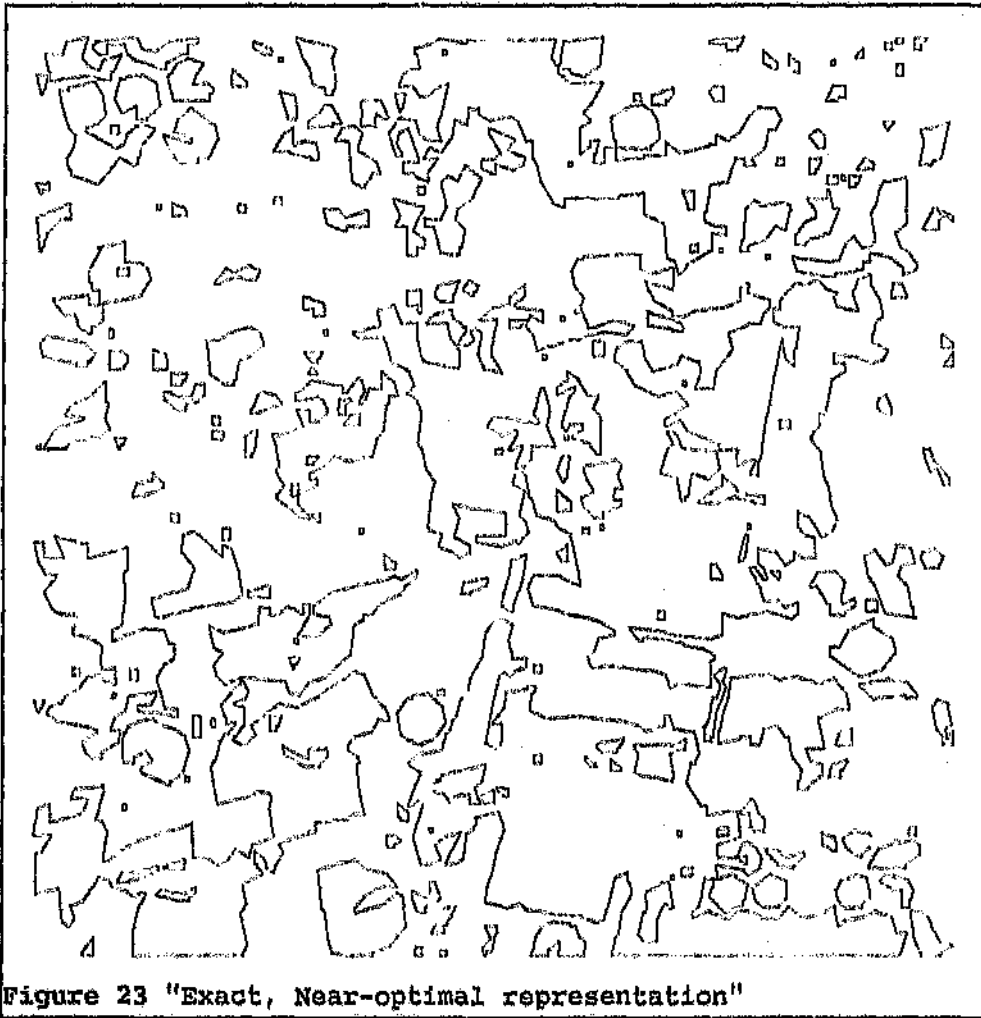


Figure 23 "Exact, Near-optimal representation"

### 7. Time complexity of the Algorithm.

The Spaghetti Machine must check each pixel in a 2 by 2 window for every pixel in the image. For every edge found, it must be placed at the top or the bottom of the string. The time complexity of this operation depends entirely on the implementation of the string object. (Whether simple stack, or list with front and end pointers whatever.) Once the string is complete, it is output, requiring a further operation for every edge vexil. Note that edge vexils pointing in the same direction as the vexil on the end of the string do not grow the string, but merely extend the last vexil.

The parsing operation performs one operation per vexil on the string.

Smoothing North-East, North-West, South-East, South-West strings needs only operate on the first two and last two vertices.

Smoothing complex strings, (nne, ene, ese, sse, etc), requires one operation per vertex but each operation requires one operation per point in the line space simplex. Usually there are around four to six points in the simplex, although for a string with irrational slope, the number of points in the simplex can grow with every point on the string.

Outputting the smoothed string requires one operation per vertex on the smoothed string.

Thus adopting the philosophy of the "Fig-leaf" notation, the number of operations the Spaghetti Machine requires to scan the image is of the order of the number of pixels in the image. The number of operations required to smooth the vexils is of the order of the number of vexils output from the Spaghetti Machine.

Thus the algorithm is approximately of linear order. (I say approximate, in that no images that this program was designed to

deal with have lines of any significant length with irrational slope. The number of classes within the image does not affect the speed of the algorithm.

However for ease of programming, and I wasn't expecting very long strings, I implemented the string object as a descendant of the stack type. Thus to scan to the bottom takes of the order of the number of elements on the stack. Thus making the time complexity of the current implementation quadratic. This has shown up as a problem when dealing with images of large rivers. However for small strings the simple stack is possibly faster. One of the virtues of the OOPS approach is that I can trivially change the string object to inherit from an array based stack or stack with front and back pointers.

### 8. Conclusions

Raster to vector conversion can be done efficiently in an exact and near optimal manner. Several aspects remain to be attended to. In this program all nodes are treated as fixed, thus to join the free floating lines to the nodes short join vectors are added. Considerable savings in the number of vectors could be achieved by constraining the first vector to go through the start node and the last vector to go through the end node. This should be a simple extension to the program involving merely some extra programming effort. Ideally the nodes themselves should be free-floating, as in Figure 24. However, if the nodes were free-floating for some pathological cases all nodes and strings would have to be in memory simultaneously.

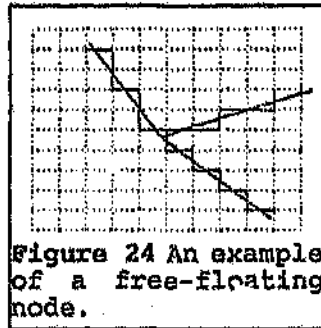


Figure 24 An example of a free-floating node.

The methods of parsing and line space constraints could be used on more general thinning algorithms.

Finally, the implementation of this project demonstrated that

- \* Object Oriented Programming will be an important component of future software systems.
- \* Current offerings in OOP field are inadequate in terms of error detection.

References

[Borland 1989]

"Turbo Pascal 5.5 Object-Oriented Programming Guide"  
Borland International Inc. 1989

[DEC CDD]

Digital Electronics Corporation  
VMS Common Data Dictionary.

[Hopcroft and Ullman 1969]

Hopcroft JE, Ullman JD  
Formal Languages and their Relation to Automata.  
Addison-Wesley Publishing Company  
ISBN 0-201-02983-9

[Korson & McGregor 1990]

Korson T, McGregor JD  
"Understanding object-oriented: A unifying paradigm"  
Communications of the Association for Computing Machinery  
Vol 33 No 9 September (1990)

[Meyel & Guegen 1987]

Meyel A, Guegen T  
"Smalltalk-80"  
Macmillan Education Ltd  
ISBN 0-333-44514-7  
005.133 QA76.8.S635

[Meyer 1990]

Meyer B  
"Tools for the new culture: Lessons from the design of the  
Eiffel Libraries"  
Communications of the Association for Computing Machinery  
Vol 33 No 9 September (1990)

[Reiser 1991]

Reiser M

"The Oberon system: user guide and programmer's manual."

ACM Press, Addison-Wesley Publishing Company

ISBN 0-201-54422-9 005.43

[Sagis 89]

Proceedings of International Conference on Geographic  
Information Systems in Southern Africa. Univ. of Natal,  
Pietermaritzburg, July 1989

[Webster 1985]

Webster B

"Methods: A preliminary look"

BYTE may 1985 Vol. 10 no 5 pp 152-154

[Wirth & Gutknecht 1989]

Wirth N, Gutknecht J

"The Oberon System"

Software - Practice and Experience 19(9) Sept 89 pp.857-893

## Appendices.

### Appendix 1. An Annotated Example.

A common data structure is the stack. Think of a stack as a bag. You can place things in a bag, you can take things out of a bag. What sort of things? Anything. What you put in last, you can take out first. This is commonly abbreviated as LIFO.

In this example and in *OopCdaisy*, a stack is implemented as a linked list. For the simplicity of it, instead of the usual link-node view, I will take a more recursive perspective.

#### A.1.1 Declaring the data structure.

A stack is a bag that can contain 2 objects. An arbitrary object, and a stack. In fact in C we have the structure :-...

```
/* the minimal container. Stacks */
typedef
    struct
    {
        objects ancestor;
        objects * object;
        struct stacks * stack;
    } stacks;
```

Thus the first field of a stack structure are the fields inherited from the base object data type, the second a pointer to an object, and the third is a pointer to a stack.

We then declare the *stackType* variable which holds the index into the registry for the stack object :-

```
objectTypes stackType;
```

And then we must declare the various method variables which hold the index into the VMT.

```
methods push, pop, isEmpty;
```

In this example I will only contemplate a very abbreviated list of methods, *OopCdaisy* itself comes with

```
lastObject, lastNode, secondLastNode, matchAddress,
howMany, pushBottom, popBottom, getBottom, putBottom,
search, searchPop, tellAll, tellUntil, tellMin,
rotate, concatenate;
```

in addition to the methods of the base object type.

The external functions corresponding to the methods must also be declared.

```
stacks * stack_init(    stacks * )          ;
void      stack_done(    stacks * )          ;
stacks * stack_display( stacks * )          ;
stacks * stack_push(    stacks * , objects * );
objects * stack_pop(    stacks * )          ;
int       stack_isEmpty( stacks * )          ;
```

Note that *stack\_init*, *stack\_done* and *stack\_display* replace the methods inherited from the parent base object type. As a matter of habit, future proofing, and often from necessity, it is good policy for a descendants method to statically invoke the parent method that it is replacing. Clearly it is also good policy to invoke the parents "init" method before the descendants initialization starts, and invoke the parents "done" method at the end of the descendants clean-up code.

The declarations for the object are placed in an include file, so they may be included into the user program and into the file that implements the stack object.

#### A.1.2 Registering the object type.

In the file which implements the stack object we must have procedure that registers the stack object in the object registry :-...

```
void register_stack()
{
```

```
/* Register the object type stack, informing OopCdaisy of
the object types existence, ASCII name, parent's name
and the amount of memory each instance uses.
```

In return OopCdaisy allocates a slot in the VMT for this object type, and returns an index into the VM in the variable stackType. OopCdaisy also zero's the count of stack instances.

```
*/
```

```
register_type(
    "stack", &stackType, "base",
    sizeof( stacks));
```

```
/*
```

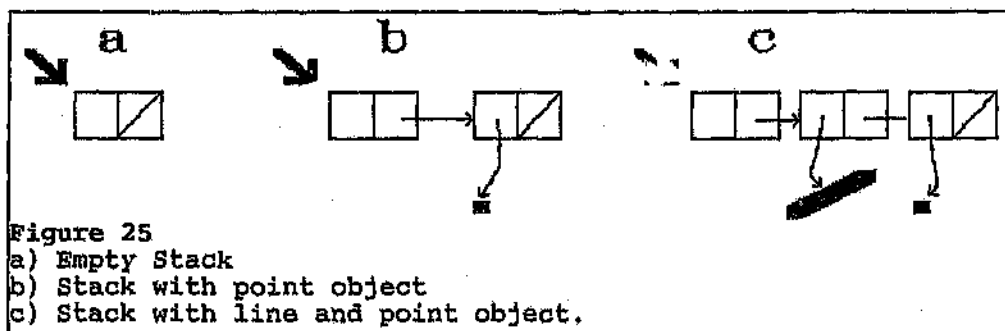
Register each method, informing OopCdaisy of the existence of the method, and its ASCII name. If a method already in the registry has the same ASCII name, any such method, whether an a method of an ancestor or not, then OopCdaisy uses the same index into the VMT and places it into the variable init. OopCdaisy then places the address of the function stack\_init into the VMT at location VMT[ stackType][ init].

```
*/
```

```
register_method(
    stackType, "init", &init,
    &stack_init);
register_method(
    stackType, "done", &done,
    &stack_done);
register_method(
    stackType, "display", &display,
    &stack_display);
register_method(
    stackType, "push", &push, &stack_push);
register_method(
    stackType, "pop", &pop, &stack_pop);
```

### A.1.3 Implementing the data structure.

To simplify life, (this is the standard dummy end node trick), the object in the topmost bag is never there. Its merely for convenience. The result is that stacks / single-link nodes etc are all the same beast, there is no special case for push and popping the last object, the user's pointer doesn't have to be modified, and a nice uniform recursive view of stacks within stacks can be maintained.



In the file that implements the stack object type the methods are defined :-...

```
stacks * stack_init( stacks * self)
(
/* stackCheck is a macro that checks that the object
pointed to by self exists, is writable, and that its
type is, or is a descendant of, the type stack. If not
it signals that the typeGuard has failed and in which
routine.
*/
stackCheck( self, "stack_init" );

/* Initialize the stack object. */
self->object = 0;
self->stack = 0;

return self;
```







## A.1.4 Using the data structure.

Here follows an exceedingly simple and useless program that creates two objects, stuffs them into a bag, displays them, and then destroys the objects in sundry manners :-

```

#include oopdir:oops
#include oopdir:stack
#include oopdir:2d
main()
{
    points * point;
    lines * line;
    stacks * bag;

    register_init();
    register_stack();
    register_point();
    /* Note : Parent types, (such as point), must be
       registered before descendant types such as line. */
    register_line();

    point = new( pointType, init, 1.0, 2.0);
    line = new( lineType, init, 1.0, 1.0, 3.0, 4.0);

    bag = new( stackType, init);
    tell( bag, push, point);
    tell( bag, push, line);
    tell( bag, display);
    /* Remove and destroy the line. */
    tell( tell( bag, pop), done);
    /* Destroys all objects in the bag and then destroys the
       bag. */
    tell( bag, destroy);
    registry_done();
}

```









**Author: Carter John Andrew.**

**Name of thesis: Raster To Vector Conversion In A Local, Exact And Near Optimal Manner.**

***PUBLISHER:***

University of the Witwatersrand, Johannesburg

©2015

***LEGALNOTICES:***

**Copyright Notice:** All materials on the University of the Witwatersrand, Johannesburg Library website are protected by South African copyright law and may not be distributed, transmitted, displayed or otherwise published in any format, without the prior written permission of the copyright owner.

**Disclaimer and Terms of Use:** Provided that you maintain all copyright and other notices contained therein, you may download material (one machine readable copy and one print copy per page) for your personal and/or educational non-commercial use only.

The University of the Witwatersrand, Johannesburg, is not responsible for any errors or omissions and excludes any and all liability for any errors in or omissions from the information on the Library website.