

Tina Memo No. 2005-001
Internal

TINA 5.0 Programmer's Guide

Neil Thacker
Last updated
8 / 12 / 2004

Medical Biophysics and Clinical Radiology
University of Manchester
Manchester M13 9PT
England

©University of Manchester, England.
Software contributors: Steve Pollard, John Porrill,
Neil Thacker, Stuart Cornell, Julian Briggs, Dave Prendergast,
Richard Lane, Antony Ashbrook, Tony Lacey, Beth Vokurka,
Marietta Scott, Paul Bromiley, Maja Pokric.

Contents

1	Programming with Tina	4
1.1	Introduction	4
1.2	Software Requirements Analysis	5
1.2.1	Programming Environment	6
1.2.2	Library Contents	6
1.2.3	Hardware	7
1.2.4	User Support	8
2	C Programming for Research.	9
2.1	Indentation and Commenting	10
2.2	Modularity and Functions	10
2.2.1	Data Structures	11
2.3	Programming Paradigms	11
2.4	Debugging	12
2.4.1	Programming Checklist	12
2.4.2	Recongnising Common Bugs When You See Them	13
2.5	Header Files and General Tips	14
2.6	Optimisation and Accuracy	15
2.7	Macro Definitions	15
2.8	Data Input/Output	15
3	Getting Started	16
3.1	Overview of Tina	16
3.2	Tina Libraries, Source Code and Include Files	18
3.3	Starting to Program	19
3.3.1	Linking Your Own Code	19
3.3.2	Modifying Header Files	20
3.3.3	Incorporation of Tina Sub-Tools	20
3.3.4	Putting Software Back into the Library	21
4	Programmer Graphics Support	22

4.1	Introduction	22
4.2	Infrastructure Graphics	22
4.3	Histogramming, Storage and Output	24
4.4	Histogram Manipulation	25
4.5	Graph Plotting	26
5	Examples from Edge Tool	29
5.0.1	Canny Callback	30
5.0.2	Geom2 Callback	31
5.0.3	Rectification	32
A	TINA Development Strategy	34

Chapter 1

Programming with Tina

Preface

This document is written for those intending to use the Tina libraries for software development. We cover the various reasons for the design of the libraries, the basics of software development and some of the detail of library functionality. Minimal knowledge of UNIX and C programming is assumed. Only the basic essentials of programming are described, and no attempt is made here to explain in detail the various intricacies of the higher level vision algorithms. Details of this work should be sought in the published literature and on our Tina memo web pages. For technical reference material describing data structures and core library functions see Tina memo 2005-004.

1.1 Introduction

The Tina system has been written and developed with the benefit of many years of hard learned experience in computer vision and image processing algorithm development. It is intended to provide a platform for research and development of vision and image processing research software without the application programmer having to start from scratch (see the *software requirements analysis* below). In doing so the user may inherit as much or as little of the existing Tina modules, graphics, code and programming style as they wish. Of course the more a program conforms to the Tina way of doing things and the representation of image, image feature and geometrical data then the more fully will it be able to interact with other modules (written or future developments). This chapter gives an overview of the philosophy behind Tina, following chapters cover tips on programming style, and how to begin writing applications. Though the first 4 chapters are recommended reading (prior to starting to use Tina), the later chapters are largely technical and should be regarded as reference material.

Considerable effort has been made to integrate into the environment as much general purpose flexibility as possible with regard to user programming. It is intended that a researcher using the system will expend a maximum of effort in developing the software of his algorithm, rather than spending his time developing basic support infrastructure. Developing debugging and testing new software is efficient provided that the user follows some simple rules for the set up of his programming environment. These programming tips are given below and in the basic *tinaTool* development directory. This directory also contains a Makefile and instructions on how to view and modify existing routines and some example "skeleton" software, as an example of how to include new software (add buttons and parameters and access graphics). Additionally, tools developed using Tina Windows (*Tw*) have a built in history mechanism which can be saved at the end of a processing session. This avoids the need for tedious tool manipulations that often precede useful work.

As Tina is the product of a research environment it is likely to undergo whatever changes are found

to be necessary in the light of continued research efficiency. As such it is impossible to produce and maintain a definitive set of documentation and over time much of the software is likely to change. We apologise in advance for any discrepancies that may have crept into the software since the last rewrite of this documentation.

It is recommended that users acquaint themselves with the software relating to the algorithms they intend to use. This is the only way to make sure that the data delivered is what they actually require and computed in the way that they would want. Utilities such as doxygen, lxr and simple search utilities (ntw) are provided for rapid searching of the software to facilitate this but researchers should expect to spend at least a few days on this task. The Tina environment makes vision research more efficient, but it doesn't make the research any easier.

Unfortunately, it is impossible to protect the programmer from the task of following what is often a difficult learning curve, but our belief is that access to such software is still preferable to spending three years or more re-inventing such methods once again, rather than building upon what has already been achieved. This idea also extends to coding practices and we try to avoid techniques which produce undue levels of obscurity (such as object orientation and function overloading). Our avoidance of unnecessary complexity has evolved out of experience (and code sharing) in a research environment over an extended period (15 years). We accept that sometimes this results in practice which is not entirely endorsed by trained software engineers. However, the aim is to develop software (which we know works) with a complexity level justified by the task, which permits future maintenance and modification by non-expert programmers (see below). Some parts of Tina were necessarily written by very talented programmers, non-expert programmers will probably be able to identify these sections (such as the generic graphics wrappers) very quickly, for all the wrong reasons. The hope is that most vision researcher, will not need to modify these parts of the library and will make use of the functionality already provided.

Although the exact design of the top level modules are by no means fixed the modular nature of the top level will maximise the longevity of applications designed within it and minimise the effort involved in their update to future, more powerful, versions. Furthermore the underlying image processing, maths and graphics libraries are now in a sufficiently fixed state of repair that major changes are becoming increasingly unlikely. No commitment has been made, at this time, to maintain backward compatibility as this is seen as likely to cause unwanted expansion of the libraries with old (and therefore inferior) versions of algorithms. If we make a change to the library it is because we think it is necessary and compatibility can normally be regained by a simple change to the argument list of a few functions or the list of headers.

Several years ago, this software was converted to the ANSI standard, and recently (in Tina5) we have modified the header file system in order to obtain forward compatibility with C++ standards. However, there is currently no intention to convert this software itself to C++. Contrary to naive expectation C++ is not simply an extension of the C language, it encompasses a completely different style to programming and development which we decided not to adopt for reasons of maintenance and productivity¹. C++ is perhaps better suited for people who know exactly what their program should do when it is finished and this is rarely the case for a research software engineer. Although it is true that C++ has some useful features, upon reading this manual you will discover that many of the advantages expected from programming in C++ are already obtained from the design of the Tina software. As far as object oriented programming is concerned, our preferred language is Java, and to this end we have recently investigated the use of **swig** wrappers. We would not however, consider re-coding in Java, as we believe C has more utility.

1.2 Software Requirements Analysis

Although the Tina system has been developed via a process of gradual change it can be said to have developed according to the following set of (evolving) research requirements.

¹A minority of us (experienced researchers) still believe that standard C is easier to develop and debug than C++.

1.2.1 Programming Environment

In order to support programming research efficiently the learning curve should be as short as possible and targeted at a reasonable level of software literacy amongst PhD candidates, as this group are the lowest common denominator. We can assume that most people in this group are familiar with one ‘reasonable’ programming language eg: FORTRAN, pascal, C (and ++). We should not assume however, that they are mathematical or programming experts. These skill will be developed during research, and we should encourage people to examine programming styles of the existing framework to help to facilitate this. Our software should thus be easy to interpret and require the minimum of programming sophistication (Keep It Simple!). Interfaces that are automatically constructed but generate a large quantity of unreadable source code, which cannot be easily modified without considerable effort, have been avoided (eg. Visual C++).

We cannot expect to be able to provide all programmers with the particular variants of an algorithm required for their particular project, though we can hope for and encourage a maximum reuse of existing (stable and therefore debugged) software.

The programming development environment should thus support the following;

- a standard development directory including Makefiles and documentation on getting started.
- software templates for GUI’s and simple processing schemes for rapid code development.
- easy access to existing (understandable) source code and a scheme for local library software modification.
- the facility to update libraries to make use of successful algorithm development in a way that causes the minimum of disruption.
- rapid recompilation and execution either as an interpreted language or with the possibility to reconfigure the interface to the same location quickly.
- reliable support for debugging.
- platform independence (SUN’s, PC’s etc).
- independence from third party development (ie: we do not want to have to start again in two years because the graphical display package we have been using is no longer supported) ².

1.2.2 Library Contents

Code reuse is not automatically obtained by choice of a language, however, it does seem to be the case that unwarranted generation of data structures limits code re-use, and some languages encourage this. The software should possess a minimum set of useful data structures, so that the majority of programming tasks are supported but selected library routines can be modified without having to modify the basic structure (and so recompile the entire software library). This can be supported via the use of user extendable data structures. (eg: see property lists).

Common data structures for vision research must include;

- vectors
- arrays
- transformation matrices
- camera models
- images

²This requirement immediately take us away from the commercial route for software development on a Windows platform, once we understand the advantages provided by Unix/Linux.

- low level image features (eg: edges, corners, strings)
- models primitives (eg: lines, conics, splines, surfaces, deformable templates)
- lists

Software libraries must include basic manipulation routines for all of these.

The next single most important requirement for research is data display and interaction. The software should thus support a wide range of data visualisation and manipulation both for the purposes of debugging and (though to a lesser extent) graphical generation for publications.

Standard data types should include

- Statistical frequency plots (histograms).
- grey level and colour image display (perhaps also anaglyphs)
- 3D data display, surface and volume
- high dimensional data visualisation

Standard interfaces should provide easy access to the display and manipulation with the minimum of programming (eg. generate the standard data structure and register it for use with the interface). It will probably be necessary for the manipulation routines to be modifiable and the graphics display to be data dependant in ways specified by the user. Also a set of standard interfaces are of most value when they can freely exchange data.

From an algorithmic point of view, the software must contain a minimum mathematical library including;

- matrix algebra (SVD etc.)
- optimisation (gradient and non-gradient based, Hough spaces, Ransac, GA's)
- pattern recognition (clustering, KNN, Neural Nets)
- simple image processing
- machine vision algorithms

For more specialised algorithms, such as temporal sequence analysis we may need to interface the software to existing research tools (eg: HTK for temporal sequence analysis) ³.

1.2.3 Hardware

For more advanced use of the software and hardware support, the environment will need an asynchronous data exchange protocol. The standard approach is to use something like a unix pipe for communication and to pass an encoded (variable length) data block through this pipe for reconstruction at the other end. We can assume that hardware accelerators will not have advanced compilers or graphical display output. Debugging must often be done using print statements and a basic ASCII file data output stream becomes essential. Standard data files should thus be in human readable ASCII where possible.

Clearly the software library must be capable of supporting a basic kernel of functionality including only file i/o and algorithmic processing but excluding all graphical interface functions. The libraries will thus need to be well defined with a clear hierarchal separation. Machine vision libraries should be independent and modular, though this could be considered as standard programming practice.

Efficient use of some hardware may require a specific language support mechanism (eg: parallelism or vectorisation).

³Thus ease of integration with third party software is important and has consequences in terms of the selected programming language (see below).

1.2.4 User Support

It is natural that during the process of interface development the programmer will want to put a wide range of flexibility into algorithm control. On the other hand users may wish a single button press system. The easiest way to achieve this is to place all algorithmic options a level further down in the GUI than the display and function calls (generally hidden from the user).

It would be advantageous to be able to automatically configure the software for a particular algorithmic variant. This would be best achieved via the use of a replayed interaction sequences or macro. Editable interaction sequences (if human readable) can also provide a high level interpreted programming language for users who wish to modify algorithms without getting involved with programming. In order to use algorithms in this way a standard procedure will need to be adopted for error reporting so that any algorithmic failiures are logged in a common location easily accessible to the user.

Users will also need to make repetitive use of algorithms, applying them to many images, either as volumes or sequences. As the users will not want to sit at a terminal repetitively pressing buttons but the programmers will not want to have to write software wrappers for individual users, this too may be best handled using macro replays.

Users, more than programmers, will probably need to use several software packages. We cannot enforce a common data format for acquisition, so a wide range of common data formats will have to be supported in the libraries.

In conclusion the users will require

- simple intuitive interfaces
- useful visualisation
- data compatability with a range of other packages (Analyse, IDL, DICOM Server)
- a programmable macro language
- error reporting

This functionality must be provided in a manner that is maintainable, and that will require some element of future proofing (we must choose ways of doing things which are likely to be long lived and easily updated).

Chapter 2

C Programming for Research.

The Tina libraries are written in C. This section has been written with the view of trying to define some reasonable rules of thumb for efficient use of that language. It is primarily intended for naive C programmers, but competent programmers may find some of our opinions ‘interesting’ if they have not explicitly thought about the way they use the language previously ¹. All rules of thumb are expected to have exceptions in specific circumstances, unless preceded by “ALWAYS” or “NEVER”.

Serious software engineers may insist that software must always be formally designed before coding. However, in the area of algorithm development, the final form of a piece of software cannot be envisaged until enough research has been done to define the algorithm. This is an obvious problem for people new to the research area who will need to try a few things out before settling on an approach. However, the situation also applies equally where the research has generated an unexpected result. At any stage, deciding to redesign and rewrite the software raises the possibility of introducing new errors, abandoning the effort already expended in debugging. If we accept that algorithm testing should ALWAYS drive the design of software (not pre-conceptions), this inevitably leads to the need for an approach which supports rapid incremental modification and puts less emphasis on a-priori design.

The C programming language has some very useful properties with respect to approaches to software development which require gradual change. In particular, the language is so flexible that it is generally possible to identify a continuous development path between any one specification of function and any other. This flexibility can be directly attributed to the separation between the sequential form of functions and data structures (and can be lost once some object oriented approaches are adopted). This can be used for the purposes of either algorithmic variant evaluation or general improvement and minimises the time required for subsequent de-bugging (see Debugging below). This flexibility supports maximal reuse of existing software and easy integration of packages.

Anything which adds overhead to the development cycle has to be weighed according to its utility and impact upon research productivity. It is because of this that conventional software design methodologies (as found on large scale industrial projects), which use formalised ways for programmers to contribute towards a software product via a rigid versioning system, are not necessarily appropriate for research software development. However, formal design methods and version control of software are not necessary on research projects provided that you follow some simple rules. These rules basically involve following common sense programming practice; regular backups and testing software modifications before inclusion into common libraries (See Appendix A). It also helps to have some guidelines for good programming practice.

So what constitutes good programming? Well it certainly isn’t exploring all of the various complexities of the language, or minimising the total line count by embedding several statements within one line. This may be considered fun but is not particularly productive. It is our opinion that the main goal

¹WARNING: What we are about to suggest may cause headaches in people who believe that programming should be regarded as an art, or are paid by the KLOC.

of algorithmic reasearch is not the generation of cleverly defined software, but a more fundamental understanding of how to extract information from data. Good programming must result in **software that others can understand**, use and potentially modify. Good programming is well defined by the phrase “KEEP IT SIMPLE STUPID”, while at the same time adopting a development methodology which will ultimately eliminate the maximum number of bugs. Keeping your software simple is ALWAYS the best method of ensuring others can understand it. So don’t try to be too clever, reserve that for the algorithms not the software design.

2.1 Indentation and Commenting

Writing understandable software starts with sensible use of indentation, but try not to use tab characters as they can take different sizes depending upon your editor. This may not cause an immediate problem for you, but it will for others.

Rather than treating the software as a piece of literature (where each variable or function name can easily become a sentence), it actually helps not to use un-necessarily long variable or function names. Preferably, common variables within functions should take common names which other programmers may conventionally use for the same task. Otherwise, others reading your software will need to keep refering to the top of the file or function to remember how variables were defined. It is quite appropriate to define locations in a 2D co-ordinate system as x or y and loop integers as i,j or k. There is no need to name them in a more complicated way which may be more meaningful to you, it’s what others will see that is important and if people are to read your software and understand it keeping things simple helps with the flow.

Put comments in your software but do not overuse them. Complex functions probably need some comment on the definition of parameters and expected returned values. These should be placed immediately after the function definition. However, comments like

```
/* a for loop over i */  
for(i=0;i<num_int;i++) {...}
```

are completely unnecessary. In general, comments which merely describe the meaning of existing C syntax should be avoided. A reasonably competent programmer will understand the software more quickly if he does not need to read around such comments.

Take the time to write the comments: when you feel they are unlikely to need changing; when you expect the software to soon be made available to others; or when you think you may not be developing the software for some time.

Don’t use C++ style comment markers “//” unless they are expected to be temporary, as they do not compile properly on all platforms.

2.2 Modularity and Functions

Modularity is important, code should be procedurised by splitting complex algorithms into self contained parts which have **conceptual relevance**. This aids with program understanding. Also, write functions for pieces which can be re-used in several places, this will reduce both the total quantity of software and the possibility of bugs (LESS CODE > LESS BUGS). This (when done properly) should result in small numbers of function parameters. If a large number seem to be required consider whether they are really necessary or if they would be sensibly encapsulated in a data structure (see below).

Always name functions so that their names bear more than just a passing resemblance to their task, but avoid long function names. Function names should follow a naming convention where possible, in C this can be used as a primitive form of “namespacing”.

Declare functions as static if they are not intended to be accessed by other parts of the system.

Functions should not be overlong. A good (understandable) length for a function is around 40 to 80 lines. Any longer than this and it becomes difficult for other programmers to understand exactly what it does (simply limitations of human short term memory) any shorter than this and you may be needlessly introducing levels of indirection into program flow which is difficult to track.

Associated functions should be stored in the same C file. Hierarchical splitting of functions is considered good programming style but it is useful for other programmers to be able to follow the flow of a program around the minimum number of files. Up to 10 associated function declarations within a single file is acceptable. An acceptable length for a C file is 250-500 lines, any longer than this and you should seriously start considering common data flow and organisation to see if the routines would not be better split up. The main reason for having separate files is development flexibility. If two programmers need to simultaneously modify the same file then this will make integrating their efforts more complicated. You can thus make decisions on which functions to group together on the basis of considering which functions are likely to require joint modification.

2.2.1 Data Structures

Before defining your own structure, check that there isn't already an equivalent one available within your existing environment. If there is you should use it as this will directly add to the resources available to other people. Defining your own structure will actually prevent code re-use.

All new data structures should have an associated memory allocation and free function which should be carefully tested to avoid error leaks (put them in the same C file so that you can compare and modify them consistently). Once written this function should ALWAYS be used to generate and free the structure.

Integration with other software libraries is generally simple and can be done at a number of levels. The simplest (though slowest and most clumsy) involves writing wrapper functions to convert data between data structures in order to allow use of externally defined functions. A better method involves modifying equivalent data structures to achieve a common definition. The clean separation of data structures and algorithms inherent in C programming makes this possible by gradual re-definition of structures using automated search and replace². A test program should be used at all stages, though the compiler will generally spot most errors.

2.3 Programming Paradigms

Recursion is very useful as a compact and powerful solution to difficult algorithmic tasks, however, it is generally difficult to understand and should be avoided if there is an alternative.

Design your software according to what you know is required now for your current research but bear in mind potential development routes for the future. Don't make the mistake of writing software now for what you think other people may use in the future, you will rarely be correct and you may just be wasting time which could be better spent on research. Doing this generally results in software which suffers from what is referred to as the second system effect, the software will be too complex for the required task and other people will rarely invest the effort needed to use it. In addition, if the additional complexity is not tested then you may just be introducing bugs which others will have to fix. The software can always be generalised later once the true nature of the required extensions are better understood and tested. If you write your software carefully and clearly others will be able to modify your software to do what they need it to do. Your main goal when considering programming style should be to support people in doing this.

Introduce object oriented mechanisms only when extending existing software to handle your particular problem. ALWAYS check that previous functionality is correctly supported. Do not rush to include

²This points up one of the main differences between a C style data structure and C++ style class.

object oriented methods into your software too quickly. This will proliferate the location of bugs and make them more difficult to track down. It will also mean that more software has to be modified in order to put the problem right. Only use these methods when the basic framework of the program has already been computationally tested.

Undue use of object oriented mechanisms should be avoided, particularly use of variable function pointers as this makes software difficult for others to understand (take a look at the graphics wrapper functions for the Tv's if you don't believe this).

Do not use global variables directly as a mechanism for exchanging data around C files. It is much better to define access functions to 'set' and 'get' the data and then to declare them only in the files which require data access. This provides better encapsulation and ensures easier integration with other software. Even this process should be only used when it is absolutely necessary, as generally is is better practice to pass all data as arguments to functions as this approach permits the possibility of parallel threading.

Avoid multiple levels of de-referencing within a single statement, when for example; extracting structures or pointer variables from other structures. Instead try to explicitly extract each structure or function variable before use. This makes the software much more readable and easier to debug.

2.4 Debugging

NEVER ignore a bug, they don't just go away and it may come back to haunt you.

The ultimate aim of good programming is to reduce the bug count to an absolute minimum. Unfortunately the definition of a bug is not particularly simple. A bug may cause a program to crash, generate a few unwanted results, or compute completely the wrong thing. The worst form of bug is the conceptual bug, where the programmer misunderstood what was needed. The functional requirement may even develop over time during the research process. Effective algorithmic testing is the only way to ensure that conceptual bugs are found and eliminated. Good performance evaluations rarely lie.

For complicated software try to test with data for which you know the exact solution.

Write groups of 10-20 lines at a time. All new software should be interactively stepped through in a debugger, as it is written, to check that each line behaves as expected. Variables should be examined to check that they are within sensible bounds and loop variables should be tested to ensure they do not pass outside the valid data domain. Attempting to avoid this process will result in software which is ultimately more difficult to debug.

When finished any associated test data sets should be preserved so that further modifications can be tested. Such test data sets are generally best kept as software research demonstrations. This ensures that the software will get tested at regular intervals during the normal course of activities.

Software should NEVER be modified without assessing the effect on an associated test data set. This is the only way to reduce the average bug count, you can't rely on version control to do this for you. Modifying software without testing is probably the best definition there is of 'hacking'.

Avoid completely rewriting software. This immediately increases the total bug count in established code. It is far better to identify a continuous development path and modify one feature of the program at a time and test each step. This is particularly important for mature software which may have been debugged over a period of years by a group of programmers and for which you may not understand all of the complexities (OLD CODE IS BETTER THAN NEW CODE).

2.4.1 Programming Checklist

There are a standard set of things which must be done when defining a new function, and it is very easy to forget one or two in the rush to get the software running. When you feel that you have finished a new function try attended to the following checklst.

- Always check parameters passed to a function. Returning a NULL pointer is a sufficient and standard mechanism to signal inappropriate parameters or incorrect function termination provided that you also register a warning (generally to stdout or equivalent).
- Do not leave untested functions in a file without a warning comment.
- Do not leave the old (discredited) algorithms or functions in a file, but don't delete old versions of functions until you are sure that the new one works.
- Always check that all data structures which have been explicitly allocated within the function (either by calls to structure allocation routines or other functions), except those which are to be returned, are correctly 'free'd. Remember to set any associated static pointers to NULL so that other routines will not attempt to use them. The apparent inconvenience of needing to attend to this yourself within C programs is also one of the languages virtues when it comes to debugging. This is an example of the (once respected) "NOTHING HIDDEN" philosophy.
- Always check that you are not attempting to access a data structure once it has been 'free'd.

2.4.2 Recongnising Common Bugs When You See Them

Having difficulty finding why your software doesn't work? Here are a list of bugs which regularly crop up during software development, taking the time to remember them now may save you time later. Many of them are not specific to C and are likely to be the main cause of problems even when writing with other languages or using complex software development tools.

Silent but Deadly

Many bugs produce no obvious problems at the site of the cause, instead the effects become visible later. This makes them difficult to track down. Take the time to find the cause and then fix the cause (not the symptom!).

- *Accessing data arrays outside of allowed bounds.* This will result in either a program crash or unexpected (often very small) data values or NaN variables within the output. Overwriting will often corrupt data in the program which has been defined immediately after the offending data structure in the program.

eg:

```
/* top of program */
float a[10],b;
```

where 'b' takes on an unexpected value is probably due to overwriting within the 'a' array.

This is such a difficult problem to spot that Tina uses a technique known as "picket fencing" to identify overwriting at the end of arrays. If it has occurred you will get the message "attempt to redefine array" in the main tinaTool dialog window. Stopping the program at the location of this error message in the debugger will allow you to see what has been overwritten. You can then work backwards to find where it happened and correct to problem.

- *Freeing a data structure which is invalid or has already been 'free'd.* This will cause the program to have an exception violation in one of the system memory management routines (malloc, calloc or free). Step up the processing stack to investigate how this has occurred. Tina provides some protection for this, by automatically resetting vector pointers to NULL (within the macro construct) whenever they are 'free'd.
- *An extra semi-colon in a one line "for" loop.* This is particularly difficult to spot.

```

for (i=0;i<my_int;i++) ; /* here it is !!! */
    x[i] = y[i]*y[i];      /* will only be executed for x[my_int] */

```

- *Casting to integer for values around zero.*

```

float x;
(int)(x);

```

Where most people use this they often should have used `floor(x)`. For historical reasons, Tina also provides the function `tina_int()`, for the same purpose.

- *Loss of scope for non ‘static’ variables.* Variables lose their value outside the scope of the C file in which they are declared and will take on random values the next time they are needed.

Monsters from the If

‘If statements’ are nearly always the cause of most development bugs. There is a multitude of ways they can be incorrectly used. In general you should attempt to use them with care using ‘if .. then .. else ’ constructs helps.

Try to minimise the total number, every new statement generates a new branch of execution possibilities increasing the overall number of ways that the program can execute by another factor of two (LESS PATHS = LESS TESTING).

Algorithms should never be written so that they are sensitive to arbitrary thresholds, they should be based on ‘soft’ statistical methods if possible. This generally follows directly if an algorithm is designed using a statistical methodology (see Tina memo 2002-005) which takes due account of accuracy of data.

Here are a few cases of potential error to look out for;

- Non initialised accumulated variables or test variables (maximum or minimum).
- Watch out for

```

if (i=value) {...} /* ie: not i==value */

```

which actually assigns value to x and is always true.

- NaN variables will always cause if statements to evaluate as false. Given that NaN are generally generated due to division by zero you can attempt to write software so that this operates in your favour. Always consider potential over-flow and under-flow conditions and take them into account.
- Never compare floating point or double variables for equivalence

```

if (x==value){...}

```

As numerical in-precision will probably ensure unreliable testing.

- Watch out for pointer comparison of char strings as an incorrect substitute for `strcmp()`.

2.5 Header Files and General Tips

Always remember to include the “math.h” file at the top of any C file which includes transcendental functions (sin, cos, sqrt etc.). Otherwise the compiler will (rather uselessly) assume that these return integers.

Try to avoid multiple level header files, they make software interpretation difficult.

When modifying library .c files, do not attempt to re-define functions away from the file they were originally defined in. Most C compilers and linkers will complain about this. If you must create a new .c file then change the function name.

Most linkers check that function prototypes agree with the function call, but do not check the function declaration itself³. All function prototypes for functions within a c file (mycode.c) should be placed in a header file (mycode.h) which should be included at the top of the c file. The compiler will then check that the function prototypes are consistent with the function definition.

2.6 Optimisation and Accuracy

Don't set software optimisation as a goal too early in software development, instead implement simple slow techniques which are easy to check (and understand) and only develop complicated (fast) approaches when you have an answer against which to compare it. A good example is writing differential functions for minimisation routines, which are very difficult to write correctly the first time due to mathematical complexity, they should be compared against methods based on finite differences.

Since the advent of floating point co-processors integer arithmetic is now actually slower on most machines than floating point!

Condensed C source code does not run any faster once it is compiled, it's just harder to read.

Good algorithms should be bounded by statistical rather than numerical stability. It is therefore generally a good idea to perform all intermediate calculations in double precision but it is generally sufficient to represent all input and output data as floating point.

Watch out for bad random number generators, they will produce peculiar systematic effects in Monte-Carlo studies.

2.7 Macro Definitions

Do not embed double, float or int control variables directly in software eg:

```
my_func(array, vectory, 0.0001);
```

Declare such variable at the top of the C function or file as a macro definition together with a comment explaining how (or at least who) defined the variable.

Do not include complicated macro definitions in software which is still to be debugged as debuggers cannot expand macro's in order to show you where the problem lies.

2.8 Data Input/Output

Simple data sets should be output from programs as ASCII and not in machine binary format unless strictly necessary. ASCII files are generally human readable and make it far easier for you and others to interpret the output of a program. In binary format it becomes necessary to write a specific binary file reader in order to access the data. Loss of precision during ASCII conversion is generally not a problem but should be expected.

³A bit of a waste of time if this was the reason you switched from K and R to ANSI

Chapter 3

Getting Started

3.1 Overview of Tina

Tina provides a hierarchy of image processing, display and interactive manipulation modules specifically designed for the recovery and representation of the 2D and 3D geometrical primitives required for the development and evaluation of computer vision systems. Beneath this is an extensive mathematics library for the manipulation of image and geometric data structures.

Clearly, graphical interaction is an important component in the rapid development of vision software and reuse of previously developed algorithms is crucial in the evaluation and development of new techniques. For this reason, care has been taken to separate functions into two libraries (tina-tools and tina-libs) of 3 basic types.

- Window System Specific Graphics (tina-tools)
- Window System Independent Graphics (tina-tools)
- No Graphics at all (tina-libs)

The hierarchy is actually in a different order, as follows:

1. window system independent
 - (a) Tw User Interface Tools
 - (b) Tv Interactive Graphics
 - (c) Tina Infrastructure
2. window system specific
 - (a) Tw tool widget wrappers
 - (b) Tvtool graphics wrappers
3. graphics independant
 - (a) File i/o
 - (b) Image/geometry processing libraries
 - (c) Supporting maths libraries
 - (d) Basic system libraries (eg: lists and memory allocation)

such that high level tools can be written in a way that the specific (window system specific) widgets and graphics libraries can be interchanged. The intention here is to minimise reliance on any one windowing system. Isolating window system specific code allows rapid reimplementations with different user interfaces. This should also simplify the development of versions of Tina for various different window systems, and allow rapid migration to new windowing environments as existing ones inevitably become outdated. This is particularly important for low level graphical display where the cost of conversion is very high. The Tvtool (Tina View Tool) device allows the concentration of all window system specific graphics into a few hundred lines of code ("screen" functions). Similarly the Tw (TinaWindows) tool prototyping libraries provide a generic subset of standard interactive tools with appropriate callback functions. Equally important, ensuring that all the low level software libraries (where all the real work is done) call no graphics routines at all, allows applications to be written that have no requirement for interactive graphical displays.

The consequence of these decisions is that we can provide only a subset of the functionality available within any one window system such that it can be implemented (and maintained) in all others. This combined with the need to be able to generate largely equivalent interfaces across platforms results in a relatively simplistic interface design which does not necessarily exploit all of the available ergonomic features and does not have the slickness of commercial packages.

The Tina Application Program refers to the top level tinaTool program usually called tinaTool.c. This is a user refinable version of a standard tinaTool interface. It is responsible for handling command line arguments (for example in X based systems it passes on the standard X arguments), generating new Tvtool display tools, displaying diagnostics, instantiating other Tina Interface Tools (see the tinaTool user guide for more details of the functionality of the tinaTool program and other modules in the Tina interface). The user may, if so desired, create a completely different top level program and still utilise other aspects of the Tina system. The TinaWindows tool prototyping libraries have been designed to make such a process rapid. The resulting tools don't always look elegant ¹. but they provide the required level of functionality for a minimum of programming effort, and of course anyone who really wants a pretty interface can always call the relevant system graphics routines directly if he wishes (if they really want to spend time reading graphics display manuals and shuffling buttons around).

The Tina Interface Tools are responsible for managing resources in the Tina Infrastructure and controlling the application of various Tina processing modules. The interface tools are responsible for handling input from the keyboard and the mouse as well as directing graphics to Tvtool display devices and handling input from them. Separating the interface tools from the infrastructure has the effect of making simple and obvious the available resources and minimising dependence upon the graphical user interface. It is possible, for instance, to develop a command line driven version of tinaTool that utilised the infrastructure modules in the absence of the interface modules (see the Tcl libraries).

The infrastructure makes available a number of cliché resources and some high level routines for acting upon them (see the chapter on Programmer Graphics Support, below). For example the module "left" makes available routines for handling images, edges, poly strings, camera geometry and Tv virtual graphic output devices. The module "left" (in conjunction with the module right) is intended for use by applications performing various forms of stereo matching. This is not to say that other applications could not use different images/edges/etc for stereo matching and display them on different TVs. In short infrastructure modules like "left" are provided for convenience.

The application program and graphics are but the tip of the iceberg, over 70% of the Tina exists at or below the image/geometry processing library level. This code is generic, independent of the user interface, and to a large extent reusable by the application programmer. The contents of the **Tina** libraries for image processing, model matching, maths and underlying list and image manipulations, etc, can be found in the Tina system documentation chapters. A fuller description of how to program with the Tv and Tvtool can be found in the chapter on Tina View programming .

¹We hope that serious researchers will judge the software based upon its contents, in particular the rather unique combination of integrated vision algorithms, rather than what it looks like.

Functions	Definitions	Prototypes	Library Archive
system	tina/sys/sysDef.h	tina/sys/sysPro.h	ltinaSys
maths	tina/math/mathDef.h	tina/math/mathPro.h	ltinaMath
image	tina/image/imgDef.h	tina/image/imgPro.h	ltinaImage
geometry	tina/geometry/geomDef.h	tina/geometry/geomPro.h	ltinaGeom
vision	tina/vision/visDef.h	tina/vision/visPro.h	ltinaVision
medical	tina/medical/medDef.h	tina/medical/medPro.h	ltinaMedical
file	tina/file/fileDef.h	tina/file/filePro.h	ltinaFile
tools	tinatool/tlbase/tlbaseDef.h	tinatool/tlbase/tlbasePro.h	ltinatoolTlBase
draw	tinatool/draw/drawDef.h	tinatool/draw/drawPro.h	ltinatoolDraw
tw wdgts	tinatool/wdgts/wdgtsDef.h	tinatool/wdgts/wdgtsPro.h	ltinatoolWdgts
X wdgts	tinatool/wdgts/ X /wdgts_ X Def.h	tinatool/wdgts/ X /wdgts_ X Pro.h	ltinatoolWdgts X
X graphics	tinatool/gphx/ X /gphx_ X Def.h	tinatool/gphx/ X /gphx_ X Pro.h	ltinatoolGphx X

3.2 Tina Libraries, Source Code and Include Files

The Tina library structure and include files reflect the hierarchy described in the previous section. For each layer there exist a corresponding library archive, a data structures, data types and macro definitions header file, and function definitions header file. Furthermore the source code for the libraries is also arranged to reflect this structure, in that each level in the hierarchy is a sub-directory of either the tina or tinatool **src** directories. Further sub-directories, below this level, reflect further degrees of specialisation, which is reflected by the availability of more specific definition header files (with names that mirror the directory structure). For simplicity, in general the application programmer need only concern themselves with the top level include files (these recursively include the rest).

The naming conventions are as defined in the table above, where **X** allows for window system specific variation. For instance for the Xview version substitute "xv", while for motif "xm" and Gtk, Gdk "gtk", "gdk".

The order of includes and/or the linking of libraries must conform to the hierarchy as specified in the table. That is the full list of includes would need to look like

```
#include <tina/sys/sysPro.h>
#include <tina/sys/sysDef.h>
#include <tina/math/mathDef.h>
#include <tina/math/mathPro.h>
#include <tina/image/imgDef.h>
#include <tina/image/img_EMDef.h>
#include <tina/file/fileDef.h>
#include <tina/file/filePro.h>
#include <tina/geometry/geomDef.h>
#include <tina/geometry/geomPro.h>
#include <tina/vision/visDef.h>
#include <tina/vision/visPro.h>

#include <tinatool/wdgts/wdgtsDef.h>
#include <tinatool/wdgts/wdgtsPro.h>
#include <tinatool/draw/drawDef.h>
#include <tinatool/draw/drawPro.h>
#include <tinatool/tlbase/tlbaseDef.h>
#include <tinatool/tlbase/tlbasePro.h>
```

although of course it should never be necessary to do so.

And the library load (for Xv) would be:

```
-ltinatoolTlMed -ltinatoolTlvision -ltinatoolTlBase -ltinatoolDraw  
-ltinatoolWdgtXv -ltinatoolGphxX11 -ltinaMedical -ltinaVision -ltinaImage  
-ltinaFile -ltinaGeom -ltinaMath -ltinaSys
```

quite a mouthful as they are all required to compile a useful application program (plus extra system, user interface, and graphics libraries).

3.3 Starting to Program

Having installed the *tina-libs* and *tina-tools* libraries, the easiest way to get familiar with Tina is by downloading several of the demonstrations and following the instructions. This will give you an idea of what is possible. The User's Guide (Tina memo 2005-002) explains most of the algorithms and projects which have already been undertaken. All of the integrated source code for these projects is available, together with algorithmic documentation and test data sets. This body of work represents well in excess of 50 man years of co-ordinated development based upon a statistical approach to system construction (see Tina memo 2001-007).

Most project directories contain a standard **Makefile** and top level C files for executable construction. The standard tinaTool download also contains additional material and instructions in order to get you started. Your personalised executable can be built simply by typing 'make' in the tinaTool directory, provided that the appropriate system environment variables have been set up as specified in the installation instructions so that the appropriate libraries and header files are correctly identified.

Keep your software up to date with the latest revision of the libraries (for example using cvs update). When your project is finished it will then be a simple matter to include generally useful elements of the software in the libraries. Software which is not directly included in the libraries should be kept in a project directory with a README file and a test data set for future extension and testing. This is the mechanism by which most of the libraries have been developed and maintained.

To aid the identification and location of functions a number of code browsers are provided on our web pages. If you need a local version you may wish to build your own (see Tina memo 2005-003).

3.3.1 Linking Your Own Code

A basic skeleton windows interface is provided written using tw_windows. This is called skeleton.c and included in the provided Makefile with the line

```
OFILES = skeleton.o
```

Your own routines can be immediately added by re-writing the example algorithms provided. Do not waste time trying to understand the user interface wrappers and graphics. The whole point of Tina is that you don't need to do this. Concentrate instead on the process of code modification, debugging (eg: using DDD) and location of algorithmic sources in the library. Once you are familiar with this process you may wish to extend your scope and try adding your own files or modifying library routines.

When writing a new routine it is recommended that you start by copying an existing function from the library and using it as a template (See Tina memo 2005-004, the Programmer Reference, for a quick overview of functions). For example, if you wish to write a simple image processing routine you may wish to start by looking at `imf_add()`. Suitable templates can often be located by chasing down through the

function calls in the user interface. Simply add the new function in the same file as the chosen template until you have a better idea of what the completed function will look like and what header files you need. Tina software contains useful approaches to memory management and data processing and as the software has already been debugged, you will be less likely to forget to do things (such as testing the validity of input data and freeing up memory). You won't find out about these or the other functionality of the libraries unless you look.

You can modify any of the software already in the libraries simply by copying the C file (and if needed the accompanying .h file) to your project directory and including the object file in the Makefile. To do this you will need to include the object file (eg: `imgPrc_add.o`) in the list of `OFILES` specified in the Makefile.

```
OFILES = skeleton.o imgPrc_add.o
```

This file, when compiled and linked, will supercede the library version.

Once a function has been written you may wish to separate it into a new file. This will require you to duplicate the structure of the original file (including headers), and at this point you should also generate a .h file for the function prototypes in the new file. To compile with your own files simply include your source code filename in the Makefile provided.

```
OFILES = skeleton.o yourownprog.o imgPrc_add.o
```

3.3.2 Modifying Header Files

If you need to modify header files copy the header to the project directory and run the *inctw* script provided. This will produce a hierachal set of directories which mimic the library structure with a soft link to any header in your working directory which is also found in the library. Any modifications to this file will automatically get included in your compiled code. Note however, modifications to structures will not apply to .o files which have not been recompiled by your Makefile (ie: those still in the library).

3.3.3 Incorporation of Tina Sub-Tools

The default `tinaTool.c` does not contain every possible sub-tool and at some point you may wish to get access to additional ones. This can be done by duplicating the process already used for all other sub-tools, define a button in the "main" tool definition at the bottom of the file, eg:

```
tw_button("RawInput", raw_input_tool_proc, NULL);
```

and provide the wrapper for the tool call at the top of the file, eg:

```
static void    raw_input_tool_proc(void)
{
    raw_input_tool(50, 50);
}
```

Now rebuild the executable, the new button (in this case "RawInput") should now be visible in the main tool, executing the function *raw_input_tool()* from the library when pressed.

3.3.4 Putting Software Back into the Library

Modifying the libraries carries with it some level of responsibility. It is generally expected that you will not wish to do this until you have quite a lot of experience (see Appendix A). However, just as a guide we will provide here a quick summary of what would be involved.

Putting modifications to existing files back into the library simply involves copying them into the appropriate locations and recompiling the library. The main issue to be aware of is that you have not redefined the functions so that they no-longer work as intended in other parts of the library.

If you have new C files and headers then you will need to make sure that these files use naming conventions which are appropriate to the part of the library you wish to place them. Header files need to be indexed using the file locations as they will exist in the library. All function calls need to use the most appropriate library routines, rather than any home grown ones that you may have used during development (this requires some familiarity with the existing libraries as there is no simple way of telling people what they should have used). The last thing to do is to add the required software licence or copyright, depending upon the required level of protection.

Once the files have been added to the library you will need to update the CVS repository to maintain them and also modify the library Makefiles to include them. All modifications should be stored in the appropriate library Change Log and then the whole lot committed back to the CVS repository. Finally, the repository should be checked out (from scratch), in order to test the build process and the new functions tested on example data.

Chapter 4

Programmer Graphics Support

4.1 Introduction

Much of the value of the Tina system comes from the ability to reuse existing data display and manipulation routines that are already supported by the software infrastructure. The user programmers can thus spend most of their time developing algorithms rather than having to develop windows interfaces. Several globally accessible data structures exist in Tina which can be manipulated by the infrastructure sub-tools.

A substantial part of algorithmic development and evaluation involves the generation of data distributions. To facilitate this process under Tina there are two levels of graph and histogramming analysis support. The first is a graphics independant library of histogram manipulation routines which can be used to provide file based output on platforms without graphical user interfaces. These can be used to generate ASCII data or text files of the histogram data and derived quantities. The other is a set of graphical plot and histogram display facilities.

4.2 Infrastructure Graphics

These are;

```
stack          : List data structure visualised in the imcalc Tv
left image     : Imrect data structure visualised in the left Tv
left edges    : "
left geometry  : List of geometrical features visualised in the left Tv
right image    : Imrect data structure visualised in the right Tv
right edges   : "
right geometry : List of geometrical features visualised in the right Tv
threed geometry : List of geometrical features visualised in the threed Tv
```

The routines for accessing and manipulating the stack are as follows:

```
void stack_push(void *val, int type, void (*freefunc) ( ))
void *stack_pop(int *type)
void stack_flip(void)
```

```
void stack_clear(void)
```

Routines for accessing the other data structures atr as follows:

```
void left_image_set(Imrect * im)
```

```
void mono_image_set(Imrect * im)
```

```
void right_image_set(Imrect * im)
```

```
void left_edges_set(Imrect * er)
```

```
void mono_edges_set(Imrect * er)
```

```
void right_edges_set(Imrect * er)
```

```
List *left_geom_get(void)
```

```
List *mono_geom_get(void)
```

```
List *right_geom_get(void)
```

```
List *threed_geom_get(void)
```

```
void mono_geom_set(List * geom)
```

```
void left_geom_set(List * geom)
```

```
void right_geom_set(List * geom)
```

```
void threed_geom_set(List * newgeom)
```

The programmer can thus obtain and modify these stanadrd structures and benefit directly from the existing user interface. For the more ambitious programmer access to the sub-tool *Tv*'s can be gained via the following functions:

```
Tv *imcalc_tv_get(void)
```

```
Tv *imcal2_tv_get(void)
```

```
Tv *imcalc_graph_tv_get(void)
```

```
Tv *imcmem_tv_get(void)
```

```
Tv *mono_tv_get(void)
```

```
Tv *left_tv_get(void)
```

```
Tv *right_tv_get(void)
```

```
Tv *threed_tv_get(void)
```

4.3 Histogramming, Storage and Output

The following routines exist for 1D and 2D histogram allocation and for resetting entries.

```
shistogram *hbook1(int id, char *title,float xmin,float xmax,int xbins)

shistogram *hbook2(int id,char *title,float xmin,float xmax,int xbins,
                  float ymin,float ymax,int ybins)

void hreset(shistogram *ph)
```

The data structure *shistogram* has the following structure

```
typedef struct shistogram
{
    Ts_id;
    int id;
    char *title;
    int type;
    double (*super)(int, double*, float);
    double *par;
    int npar;
    float xmin;
    float xmax;
    float ymin;
    float ymax;
    double mean;
    double mean2;
    int xbins;
    int ybins;
    float xincr;
    float yincr;
    int entries;
    float contents;
    float under;
    float over;
    float above;
    float below;
    float **array;
} shistogram;
```

The following routines exist for filling of 1D and 2D histograms.

```
float hfill1(shistogram *ph,float x,float w)

float hfill2(shistogram *ph,float x,float y,float w)
```

Where x y w are the ordinates and entry weights respectively and the returned value is the current contents of the selected bin.

The following routines exist for histogram output

```

void hprint(FILE *fp,shistogram *ph)

void hpxprint(FILE *fp, shistogram *ph)

void hpyprint(FILE *fp, shistogram *ph)

void histdo(FILE *fp)

void hstore(FILE *fp,shistogram *ph)

void hfetch(FILE *fp)

void hsuper(shistogram *ph,int n,double (*func)(int, double*, float),double *a)

```

Standard 1D histogram output (with superimposed fit) looks like the following:

```

h id 0 left stereo

0.00 xxxxxxxx*
1.00 xxxxxxxx*
2.00 xxxxxxxx *
3.00 xxxxxxxxxx*
4.00 xxxxxxxxxxxx*
5.00 xxxxxxxxxxxxxx*
6.00 xxxxxxxxxxxxxx*
7.00 xxxxxxxxxxxxxx*
8.00 xxxxxxxxxxxxxx*
9.00 xxxxxxxxxxxxxx*
10.00 xxxxxxxxxxxxxx*
11.00 xxxxxxxxxxxxxx*
12.00 xxxxxxxxxxxxxx*
13.00 xxxxxxxxxxxxxx*
14.00 xxxxxxxxxxxxxx*

h id 0 entries 15 contents 528.049 underflows 0.000 overflows 0.000
h id 0 mean value 9.368 variance(**0.5) 3.908 scale factor 2.000

```

Standard 2D histogram output looks like the following:

```

h id 25 KNN classification matrix

14.00      1 1 4 3      5      D
13.00 8 1 1 1      1      2 F
12.00 2      2 1 1 C 1 1
11.00
10.00
9.00 1      9
8.00
7.00      1
6.00      2 4      1
5.00
4.00      3 A 1      1 2 2
3.00      1
2.00
1.00      1 3
0.00 A 1 2      1      2 1
0 1 2 3 4 5 6 7 8 9 1 1 1 1 1
. . . . . 0 1 2 3 4
0 0 0 0 0 0 0 0 0 . . . . .
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

Key: .<1.0, A<15.0, B<20.0, C<30.0, D<45.0,
E<65.0 F<100.0, G<150.0, H<200.0, *<Infty.
entries 224 contents 224.00
under 0.00 over 0.00 above 0.00 below 0.00

```

4.4 Histogram Manipulation

In addition to the basic histogram generation and printing options some useful manipulation routines are also provided for diagonal summation, forward and backwards integration, smoothing, fitting and fit

parameter covariance estimation.

```
double hdiag(shistogram *ph)

void hintegf(shistogram *ph)

void hintegb(shistogram *ph)

void hsmoof(shistogram *ph)

void hfit(FILE *fp,shistogram *ph,int n,double *a,double *w,
          double (*fitfunc)(int, double *,float),
          double (*derfunc)(double *,float,int),
          double (*errfunc)(shistogram *,float))

double **herror_analysis(FILE *fp,shistogram *ph,int n,double *a,
                          double (*fitfunc)(int, double *,float),
                          double (*derfunc)(double *,float,int),
                          double(*errfunc)(shistogram *,float),
                          int exclude)

void hfit_gauss(FILE *fp,shistogram *ph)
```

4.5 Graph Plotting

The Tina system supports graphical display of data under a reduced version of the *TvTool* structure. Display of data on this device is supported from the function *plot(int case_flag, ...)* which takes a variable argument list of flags followed by associated data. These flags comprise:

Control Flag	Parameters	Description
PL_INIT	No params.	: initialise graphics
PL_FREE	No params.	: free graphics structures
PL_TV	Tv *tv	: set up display Tv
PL_COLOR	int color	: plot colour
PL_AXIS_COLOR	int color	: axis colour
PL_STYLE	int style	: plot style
PL_TITLE	char *title	: title
PL_LEGEND	char *legend	: legend
PL_AXES	Bool set	: set both axes
PL_TICS	Bool set	: display with axis ticks
PL_TEXT	Bool set	: overlay text string
PL_X_AXIS	Bool set	: set x axis
PL_X_TICS	Bool set	: display with x axis ticks
PL_X_TEXT	Bool set	: x axis text
PL_X_RANGE	double xmin double xmax double xinc	: x range
PL_Y_AXIS	Bool set	: set y axis
PL_Y_TICS	Bool set	: display with y axis ticks
PL_Y_TEXT	Bool set	: y axis text
PL_Y_RANGE	double ymin	

```

double ymax
double yinc      : y range
PL_XFORMAT      char *fm      : set x axis display format
PL_YFORMAT      char *fm      : set y axis display format
PL_GRAPH_DATA   int n
float *z         : data as a graph
PL_SCATTER_DATA int n
float *x
float *y         : data as a scatter plot
PL_GRAPH_FUNC   int n
double a
double b
double (*f) ()  : data from a function
PL_HIST_RANGE   double zmin
double zmax
double zinc     : set histogram range
PL_HIST_DATA    int n
float *z        : data as a histogram
PL_CUMHIST_DATA int n
float *z        : data as a cumulative histogram
PL_CTR_RANGE    double cmin
double cmax
double cin      : set contour plot range
PL_CTR_DATA     int nx
float *x
int ny
float *y
float **z       : data as a contour surface
PL_CTR_FUNC     int nx
double ax
double bx
int ny
double ay
double by
double (*f) ()  : data as a contour function
PL_PLOT         No params.    : execute plotting

```

This functionality can easily be controlled using the data stored in the histogramming structures. Typical use of this display software would look something like the following example, taken from the *Calib Tool*.

```

static Tv *graph_tv = NULL;
static double accuracy = 1.0;
.
.
.
static void graph_tool_proc(void)
{
    extern void *display_tool();

    graph_tool = (void *) display_tool(100, 200, 400, 256);
    graph_tv = tv_create("graph");
    tv_install(graph_tv, graph_tool);
}

```

```

.
.
.
static void epierr_proc(void)
{
    double *epi_err;
    float  lx[512];
    int    i, n_data = 1024;
    Vec2   *leftpix_get();
    Vec2   *rightpix_get();

    if (leftcam == NULL || rightcam == NULL)
        return;
    if (graph_tool == NULL)
    {
        error("no graph tv_screen", warning);
        return;
    }
    tv_erase(graph_tv);
    epi_err = (double *) ralloc(sizeof(double) * (unsigned int) n_data);
    (void) triv_camerror(&n_data, epi_err, leftcam, rightcam,
                        world3d, leftpix_get, rightpix_get, accuracy);
    if (n_data == 1024)
    {
        format("warning only 512 data points displayed\n");
    }
    for (i = 1; i < n_data; i += 2)
    {
        lx[i / 2] = (float) epi_err[i];
        if (lx[i / 2] >= 3.0 * accuracy)
            lx[i / 2] = 3.0 * accuracy;
        if (lx[i / 2] < -3.0 * accuracy)
            lx[i / 2] = -3.0 * accuracy;
    }
    n_data /= 2;
    plot(PL_INIT,

        PL_TV, graph_tv,
        PL_AXIS_COLOR, black,
        PL_TITLE, "epi-polar error",
        PL_HIST_RANGE, -3.0*accuracy, 3.0*accuracy, 3.0*accuracy / 30.0,

        PL_COLOR, red,
        PL_HIST_DATA, n_data, lx,

        PL_PLOT,
        NULL);

    rfree((void *) epi_err);
}

```

Chapter 5

Examples from Edge Tool

To illustrate the design of the user interface modules and their interaction with infrastructure and Tina libraries the remainder of this chapter describes elements of the **edge tool** used to perform monocular and binocular edge and feature recovery in 2D and 3D. This text assumes that the reader is familiar with the use of this tool (see user guide, Tina memo 2005-002, for details).

A full listing of the edge tool code is given at the end of this chapter. The **Edge Geom Tool** module has the following form

- #include files .
- static parameter variables .
- extern parameter access functions .
- static callback functions .
- static mouse and pick callback functions .
- static parameter dialog tools .
- extern edge tool

Note that to encourage good design only the **Edge Geom Tool** and parameter access functions (of which there is only one) are available externally from the **Edge Geom Tool** module.

The **Edge Geom Tool** has the following design

```
void edge_tool(int x, int y)
{
    static void *tool = NULL;

    if (tool)
    {
        tw_show_tool(tool);
        return;
    }
    tool = tw_tool("Edge Tool", x, y);

    tw_menubar("Pick",.....
              .
    tw_choice("Image Select:", update_pcam, NULL);
              .
}
```

```

tw_button("new pcam",.....
        .
        .
tw_button("Edge Params", edge_param_dialog, NULL);
        .
tw_end_tool();
}

```

If the tool has been called before it is simply reshown otherwise it is created. The **Edge Geom Tool** has 4 parts: menubars for mouse selection for the various Tv devices associated with it (ie. mono, left, right and threed); choice lists for "mode of operation" selections; buttons to perform edge, stereo, and description operations; buttons to instantiate dialog boxes to manage parameters of the module.

The tool has 2 basic modes of operation, either mono or stereo. Furthermore the method used in polygonal approximation can also be selected.

5.0.1 Canny Callback

The basic canny callback procedure is argument free and is as follows

```

static int stereo_images;

static void canny_proc()
{
    if (stereo_images)
        stereo_canny_proc();
    else
        mono_canny_proc():
}

```

The variable **stereo_images** is used to select the appropriate Tina interface module procedure. We shall consider the case of monocular image processing

```

static void mono_canny_proc(void)
{
    Tv      *tv;
    Imrect *im;

    tv = mono_tv_get();
    im = mono_image_get();
    if (tv != NULL && tv->tv_screen != NULL) /* get roi from tv */
        im = im_subim(im, tv_get_im_roi(tv));
    mono_edges_set(canny(im, sigma, precision, low_edge_thres,
                        up_edge_thres, len_thres));
    if (tv != NULL && tv->tv_screen != NULL)
        im_free(im);
    tv_edges_conn(tv, mono_edges_get());
    tv_flush(tv);
    mono_geom_set((List *) NULL); /* now out of date */
}

```

This function involves a call to the canny function (resident in -ltina library) on the current mono image and display (volatile) on the current mono tv (if any). As well as identifying edges the canny function

also determines putative connectivity and forms a list of edge strings which is stored on the property list of the resulting `Imrect`. The region of interest of the `mono` (if in use) is used to define the image region to be processed. The display function `tv_edges_conn` uses the connectivity of an edge to determine the colour of the pixel.

5.0.2 Geom2 Callback

The basic geom callback is argument free and is as follows

```
static int stereo_images;

static void geom2_proc(void)
{
    if (stereo_images)
        stereo_geom2_proc();
    else
        mono_geom2_proc();
}
```

The variable `stereo_images` is used to select the appropriate Tina interface module procedures. We shall consider the case of monocular image processing

```
static int linear_approx;

static void mono_geom2_proc(void)
{
    Imrect *er;
    Tv      *tv;
    List    *strings;
    double  low_th, up_th;

    if ((er = mono_edges_get()) == NULL)
        return;

    low_th = low_fit_thres;
    up_th = up_fit_thres;

    strings = (List *) prop_get(er->props, STRING);

    switch (approx_type)
    {
    case POLY_PROX:
        strings = poly_strings(strings, low_th);
        break;
    case LINEAR_PROX:
        strings = linear_strings(strings, low_th);
        break;
    case CONIC_PROX:
        conic_filter_set(conic_bckf); /* set bias corrected filter */
        strings = conic_strings(strings, 12, low_th, up_th, max_div);
        break;
    case LINEAR_CONIC_PROX:
```

```

        conic_filter_set(conic_bckf); /* set bias corrected filter */
        strings = linear_strings(strings, low_th);
        poly_strings_find_conics(strings, sample, low_th, up_th, max_div);
        break;
    }

    if (options & JOIN)
        strings = conic_join(strings, er->region, conf_thres,
                             low_th, up_th, max_div);

    mono_geom_set(strings);
    tv = mono_tv_get();
    tv_repaint(tv);
    reclist_list_draw(tv, strings, NULL, geom_col_draw, NULL);
}

```

First the list of edge strings are recovered from the property list of the mono edges imrect (also called an edirect) using the type definition **ER_STRINGS** (EdgeRect STRINGS). These use the standard **Tstring** (Tina string) data structures to index connected edges. The list of **Line2** strings produced by either **linear_strings**, **poly_strings** or **conic_strings** reflect the structure of the original list of edge strings. Each poly string is recovered from a separate edge string and the order of **Line2** elements (possibly a singleton) along the string reflects the order of edges along the original string.

Note that on completion each **Line2** element has on its property list, indexed by the definition **STRING**, an index into the edge sub string to which it corresponds.

The new list data structure is finally registered in the mono geometry global variable for external access by other tool utilities with **mono_geom_set**.

5.0.3 Rectification

The rectify callback function has the following form

```

Static void rectify_edges_proc(void)
{
    Imrect *left_er;
    Imrect *right_er;
    Camera *lcam;
    Camera *rcam;
    Parcam *pcam;
    Vec2    cleft = {Vec2_id};
    Vec2    cright = {Vec2_id};

    if (!stereo_images)
    {
        error("edge tool: not in stereo mode", warning);
        return;
    }
    if (edges_rectified == true)
    {
        error("edge tool: edges already rectified", warning);
        return;
    }
}

```

```

left_er = left_edges_get();
right_er = right_edges_get();
lcam = left_camera();
rcam = right_camera();

if (left_er == NULL || right_er == NULL)
{
    error("no edges", warning);
    return;
}
if (lcam == NULL || rcam == NULL)
{
    error("no camera data", warning);
    return;
}
update_pcam();
pcam = pcam_get();
(void) ralloc_start_blocked(block_label);
er_apply_to_all_edges(left_er, edge_save_pos_prop, (void *) IMPOS);
er_apply_to_all_edges(right_er, edge_save_pos_prop, (void *) IMPOS);
if (lcam->correct_func != NULL)
    er_correct(left_er, lcam);
if (rcam->correct_func != NULL)
    er_correct(right_er, rcam);
er_rectify(left_er, pcam->rect1);
er_rectify(right_er, pcam->rect2);
(void) ralloc_end_blocked();
cleft = rectify_pos(pcam->rect1, vec2(lcam->cx, lcam->cy));
cright = rectify_pos(pcam->rect2, vec2(rcam->cx, rcam->cy));
Disp = vec2_x(cright) - vec2_x(cleft);
edges_rectified = true;
}

```

After obtaining the required pointers to global data structures the existence of left and right edges and cameras is verified. A new parallel camera geometry derived from left and right cameras using **update_pcam()** and **pcam = pcam_get()**. Then begins the process of rectification. First the current edge location is saved on the property list of the edge (useful for display purposes) using the type definition **IMPOS** (Image POSition) by applying the function **edge_save_pos_prop()** to all edges in the left and right images using **er_apply_to_all_edges()**. The function **er_rectify** applies the rectification matrix (projective transform) to all edges of the edge set and changes their type to indicate rectified locations. Finally the gross disparity that results from rectification of the left and right image centres is computed and saved in **Disp**. This value is used in conjunction with disparity ranges supplied by the user to provide initial ranges of disparity to the stereo process.

The process of drectification is similar to rectification except that rectified edge locations are not saved, and the function **er_drectify** is used to perform the rectification process. This is identical to **er_rectify** except for the effect on the rectification component of the type field of the edges.

Appendix A

TINA Development Strategy

Tina is developed during the course of several concurrent research programmes and is never entirely static, though the majority of changes to existing routines are generally minor.

Each researcher is provided with a project directory in which they can compile their own version of a Tina utility (eg: Tinatool) including their own source code and any of the library sources they wish to modify.

Upon completion of the project and successful publication, the software is installed in a project area alongside the Tina src directory together with example data sets and an appropriate README. This software is then evaluated and any parts which can be replaced by existing Tina routines are replaced. The remaining software is tidied and functions re-named to match the Tina naming conventions.

Any functions deemed to be of generic value are included in Tina src directories in the appropriate place. Functions may be included later if a need arises for them on another project.

Once included in the main library, software is distributed as copy-left open source via our WWW pages for the scrutiny and use of others.

The key steps which make this process efficient and manageable in a large research group are;

- Unrestricted access by programmers to source code.
- Transparent programming.
- The freedom to evaluate the effect of any modification.
- Regular testing on example data sets.
- Keeping to the defined Tina library structure.
- The inclusion of new routines only on the basis of proven need.
- Modification of the library only by a limited number of experienced programmers.
- All software bugs are fixed immediately they are understood.

As a consequence, any researcher can contribute to the library if they develop something of value and are of course free to take it with them when they leave (though not for commercial gain).