Tina Memo No. 2005-004
Internal

TINA 5.0 Programmer's Reference

Edited by Neil Thacker
Last updated
8 / 12 / 2004

Medical Biophysics and Clinical Radiology
University of Manchester
Manchester M13 9PT
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 Tina Windows **7**

1.1 Compiling Applications with Tina Windows 7
1.2 Initialisation 7
1.3 Tools 7
1.4 Panel Layout 8
1.5 Labels 8
1.6 Buttons 8
1.7 Menu Bars 9
1.8 Check Lists 9
1.9 Choice Lists 10
1.10 Value Items 10
1.11 The Text Window 11
1.12 Subtools 11
1.13 Dialog Boxes 12
1.14 Example Program 12
1.15 Re-displaying Subtools and Dialogs 12
1.16 Call Scripts 13

2 Tina View **15**

2.1 Installing a Tv on a Tvtool 16
2.2 Output Primitives 17
2.3 Output Attributes 19
2.4 Overlay Mode 20
2.5 The 2D Graphics Camera 20
2.6 The 3D Graphics Camera 21
2.7 Geometrical Display 22
2.8 Zoom Facilities 23
2.9 Adding Redraw Facilities 23
2.10 Colour-Map Selection 24
2.11 Workings of the X11 Colour System 25

3 Generic Lists **26**
3.1 Introduction . . . 26
3.2 Specifying Types . . . 26
3.3 Lists and Ddlists . . . 27
3.4 Property Lists . . . 27
3.5 Tina Strings . . . 28
3.6 Recursive Lists . . . 28
3.7 Applying Functions . . . 29
3.8 List and Ddlist functions . . . 30
3.9 String Functions . . . 35
3.10 Property List Functions . . . 37
3.11 Recursive List Functions . . . 38

4 Serialisation **40**
4.1 Functionality . . . 40
4.2 Data Parsing . . . 40
4.3 Data Flexibility . . . 42
4.4 Utilities . . . 42
4.5 General Programming Advice . . . 43
4.6 Machine Dependance . . . 43

5 Tina Maths Library **44**
5.1 Util: Maths Utilities . . . 44
5.1.1 Complex Variables . . . 45
5.1.2 Random Variables . . . 46
5.2 2D and 3D vector geometry . . . 47
5.2.1 3D Vector Algebra Functions . . . 48
5.2.2 3D Vector Geometry . . . 52
5.2.3 3D Matrix Algebra Functions . . . 55
5.2.4 2D Vector Algebra Functions . . . 59
5.2.5 2D Vector Geometry Functions . . . 60
5.2.6 2D Matrix Algebra Functions . . . 61
5.2.7 3D Rotation Functions . . . 62
5.2.8 3D Transformation Functions . . . 63

6 Maths Utilities **66**
6.1 Simple Arrays and Vectors . . . 66
6.2 Simple Functions . . . 67
6.3 Random Number Generator . . . 68
6.4 Statistical Distributions . . . 68

6.5 Matrix Inversion and Decomposition . . . 68
6.6 Solvers and Optimisation . . . 69
6.7 Covariance Functions . . . 70

7 Image Handling **71**
7.1 Introduction . . . 71
7.2 Imrects . . . 71
7.2.1 Image Regions . . . 72
7.2.2 Image Types . . . 72
7.2.3 Image Indexing . . . 73
7.2.4 Pointer Images . . . 74
7.3 Image Functions . . . 74
7.3.1 Region of Interest Functions . . . 77

8 Image Processing **78**
8.1 Introduction . . . 78
8.2 Convolution and Filtering . . . 78
8.3 Pixel Processing . . . 79
8.4 Complex Images . . . 82
8.5 Noise Filtering . . . 83
8.6 Image Warping . . . 84
8.7 Feature Location . . . 85
8.8 Generic Manipulation of mixed image types . . . 85
8.9 Manipulating Images of Tina Data Structures . . . 86

9 Feature Representation **87**
9.1 Introduction . . . 87
9.2 Image Features . . . 87
9.2.1 Image Feature Functions . . . 90
9.3 Geometrical Primitives . . . 92
9.3.1 Scalar . . . 92
9.3.2 Point Geometry . . . 93
9.3.3 Line Geometry . . . 94
9.3.4 Curve Geometry . . . 96
9.3.5 Plane Geometry . . . 99
9.3.6 Transformations . . . 99
9.3.7 Generic Geometry Functions . . . 100

10 Camera Geometry **103**
10.1 Introduction . . . 103
10.2 Describing Cameras . . . 104

10.2.1 Parallel Stereo Camera Geometry . . . 106
10.3 Camera Functions . . . 107
10.4 Rectification Functions . . . 107
10.5 Parallel Projection . . . 108

11 Stereo matching **110**
11.1 Stereo Index . . . 110
11.2 Edges . . . 111
11.3 Matches . . . 111
11.4 Edge and Stereo Data Structures . . . 111
11.5 Edgerect . . . 112
11.6 Edge Strings . . . 113
11.7 Stereo Index . . . 113
11.8 Matching Sub Strings . . . 114
11.9 Matching Whole Strings . . . 115

A Type Definitions **116**

Preface

This document is intended as a quick reference to the data strucutres and core libraries in Tina.

Chapter 1

Tina Windows

Tina windows is a toolkit for rapid prototyping of button/menu driven applications programs. It can be combined with the Tina View graphics capabilities to make complex interactive graphics tools. It also incorporates a save facility that allows the user to save a history of the user interactions and tool placements, which can be replayed on start up.

Tools based entirely on Tw and Tv will compile with both Motif and XView and give similar user interfaces. This is achieved by offering only a subset of the capabilities of these toolkits, however this subset does include the facilities felt to be most useful and general.

1.1 Compiling Applications with Tina Windows

Tina Windows must be linked with the Tina libraries and the appropriate tinaXv or tinaMv library for XView of Motif environments respectively followed by tinaX11 , eg:

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

1.2 Initialisation

The function:

```
tw_init(&argc, argv)
```

strips off the the calling arguments used by the window system, initialising the underlying window system and Tina windows. It must always be called before any other Tina windows function.

1.3 Tools

Tina windows applications consist of one or more tools created by calling:

```
tw_tool(char *name, int posx, int posy)
```

which specifies a tool with given name positioned at (posx, posy).

Tools have a single control panel (to contain buttons, menus etc.) positioned above a single optional text window. After all entries in a tool have been specified a call to

```
tw_end_tool(void)
```

is required.

When all tools have been specified, a call to

```
tw_main_loop(void)
```

runs the application; this should be the last statement in main().

1.4 Panel Layout

Tools and sub tools have a single panel in which buttons etc. can be positioned in row order.

Panel items are positioned on the same row until

```
tw_newrow(void)
```

is called, when a new row is started.

Layout is as simple and automatic as possible, and is usually satisfactory, but fine tuning controls could be made available.

1.5 Labels

A label is created by calling the function

```
tw_label(char *name)
```

A label has no functionality - it merely provides information.

1.6 Buttons

A button is created by calling the function

```
int   tw_button(char *name, void (*func) ( /* ??? */ ), void *data)
```

This creates a button with the given name which calls the procedure argument with the given data when the button is pressed.

1.7 Menu Bars

Motif prefers all menus in a single bar at the top of the program. A menu bar controlling multiple menus can be created with a single call to

```
tw_menubar(menu_bar_name,
  menu_1_name,
      name1_1, proc1_1, data1_1,
      name1_2, proc1_2, data1_2,
      ...,
      NULL,
  menu_2_name,
      name2_1, proc2_1, data2_1,
      name2_2, proc2_2, data2_2,
      ...,
      NULL,
  ...,
  NULL);
char *menu_bar_name, *menu_1_name, ..., *name1_1, ... ;
void (*proc1_1)(), (*proc1_2)(), ... ;
void *data1_1, *data1_2, ... ;
```

This function has a variable length argument list, sensible indentation will help. A menu bar with name menu_bar_name is created, with buttons called menu_1_name, menu_2_name etc. which bring up menus when pressed.

On each menu choosing the entry with a given name calls the associated procedure with the given data as argument. Note each menu is terminated by a NULL argument and the last argument must also be NULL; no names can be NULL pointers (though data items can be).

1.8 Check Lists

Check lists allow multiple options to be selected from a permanently displayed horizontal list. A check list is created with a call to the function

```
void *tw_check(char *name,...)
```

such as

```
tw_check(name,
  func, mask,
        name_1, name_2, ... ,
        NULL);
char *name, *name_1, ..., *name_k;
void (*func)();
int mask;
```

Changing the checked entries sets a bit-mask mask (the lowest order bit represents choice of the leftmost item) and func is called with this as argument. The initial state of mask sets the display appropriately (remember to make this correspond to the initial state of the application).

1.9 Choice Lists

Choices are like checks but only one item can be selected.

```
tw_choice(menu_name,
  func, value,
        name_1, name_2, ... ,
        NULL);
char *name, *name_1, ..., *name_k;
void (*func)();
int value;
```

On changing the choice item the integer value is set to the number of the chosen item (starting at 0), and func is called with this as argument. The initial state of value sets the display appropriately (remember to make this correspond to the initial state of the application).

1.10 Value Items

These allow the input of parameter values as text items. Items are integer, floats (which means doubles), and strings.

Value items come in two styles. The first asks the user to provide a pointer to the variable whose value is to be updated. The second uses user provided get and set functions to update a hidden variable.

Any change to the text item automatically updates the variable value if it is meaningful.

Global integer, float and string value items are set up by calls to

```
void *tw_iglobal(char *name, int *ptr, int nchars)


void *tw_fglobal(char *name, double *ptr, int nchars)


void *tw_sglobal(char *name, char *ptr, int nchars)
```

Remember that the pointers must point to accessible memory (in particular p_char must point to enough memory to hold the length of strings required).

Get and set integer, float and string value items are set up by calls to

```
void *tw_ivalue(char *name, int (*get) ( /* ??? */ ),
                void (*set) ( /* ??? */ ), int nchars)

void *tw_fvalue(char *name, double (*get) ( /* ??? */ ),
                void (*set) ( /* ??? */ ), int nchars)

void *tw_svalue(char *name, char *(*get) ( /* ??? */ ),
                void (*set) ( /* ??? */ ), int nchars)
```

Any change to the items in the text item automatically calls the appropriate set function. Initial values are provided by the get function.

Initial values in value items are specified by the initial pointer or get function. Again make sure this corresponds to the state of your application. To reset parameters which calculated internally to the algorithm software use the funtions

```
void    tw_fglobal_reset(void *ptwc)

void    tw_iglobal_reset(void *ptwc)

void    tw_sglobal_reset(void *twc) ;
```

1.11 The Text Window

One text window is allowed per tool at present. It must be created last and before calling tw_end_tool by a call to

```
Widget  tw_textsw(int rows, int cols)
```

This creates a text window wih nrows rows and ncols cols. The last created textsw (most Tina applications need only one) can be written to by a call to the function

```
void    format(char *fmt,...)
```

which works exactly like printf, but sends its output to the text subwindow (if it exists, if not it goes to stdout).

If there are multiple tools with text subwindows tw_textsw returns an integer identifier which can be used together with the print functions

```
void    tw_text_add(Widget text, char *string)

void    tw_text_set(Widget text, char *string)
```

where tw_text_add adds the string to the end of the displayed text and tw_text_set replaces the displayed text by string.

1.12 Subtools

Subtools in Tina windows are also created with the same function as tools:

```
tw_tool(char *name, int x, int y);
```

Their entries are described with exactly the same syntax as tools. All entries described above are allowed; in the case of a textsw this will become the new preferred destination of output from Tina print functions such as format. After entries have been specified there must be a call to

```
tw_end_tool(void);
```

Subtools iconify separately from the top level tool.

1.13 Dialog Boxes

Sometimes a tool is required only for a limited time, for example to change parameter values, and can then be dismissed without the need for an icon. In this case a dialog box should be used rather than a tool. The are created with a call to

```
tw_dialog(char *name)
```

Dialog entries are described with exactly the same syntax as tools. They can contain anything other than text subwindows. After specifying their entries there should be a call to

```
tw_end_dialog(void)
```

The have either a pushpin (XView) or a dismiss button (Motif).

1.14 Example Program

The Tina source directories contain a standard version of the UNIX Tina Makefile, together with the source code skeleton.c which contains examples of the functionality described above. For details on how to develop your own vision algorithm software within the Tina system please read the accompanying README in the same directory.

1.15 Re-displaying Subtools and Dialogs

The fragment of code below will create a new version of the tool every time it is called.

```
void create_a_subtool(void)
{
    tw_tool("Sub Tool", 256, 128);

    /** specify sub tool entries here **/

    tw_end_subtool();
}
```

Sub tools invoked by the user (say by a button press) are usually meant to be unique, and only need to be re-displayed if they have been iconified. They should thus be created with a code fragment like

```
void create_a_subtool(void)
{
    static void *tool_id = NULL;
```

```
    if(tool_id != 0)
    {
  tw_show_tool(tool_id);
  return;
    }

    tool_id = tw_tool("Sub Tool", 256, 128);

    /** specify sub tool entries here **/

    tw_end_subtool();
}
```

The first time the tool is invoked it will be created in the usual way, subsequent invocations will simply re-display the tool. Similarly dialog boxes should be created with a code fragment that allows them to be re-displayed after dismissal rather than re-created, e.g.

```
void create_a_dialog()
{
    static void *dialog_id = NULL;

    if(dialog_id != 0)
    {
  tw_show(dialog_id);
  return;
    }

    dialog_id = tw_dialog("Dialog Box");

    /** specify dialog entries here **/

    tw_end_dialog();
}
```

1.16 Call Scripts

Tina Windows incorporates a facility for saving scripts of user interactions with tools and user tool placement.

If the name of the program is progname then running the program with a -s (save) option

```
progname -s
```

saves a record of user interactions with Tina Windows in a file called progname.cls. Beore exiting the program a record of tool placement can be created by pressing the Save button which is a special button created with a call to

```
tw_save_button()
```

The next time the program is run with a -r (recover) option

```
progname -r
```

it will start by running all the saved calls and placing all invoked tools. If both -r and -s are specified saved calls will be replayed, then the user can then perform further interactions which will be appended to the call file.

This facility is especially useful when there are a lot of tools to be placed and standard set up procedures to be invoked.

The present version does not store the window hierarchy - tools that were visible before the save may be obscured on replay. Also iconification is not always dealt with properly.

Chapter 2

Tina View

TinaView is an interactive graphics system aimed at vision applications.

The design aims were:

1. Window system independence . This has been achieved by separating the display facilities between Tv's, logical display devices which are window system independent, and Tvtool's, actual tools on the screen which have standard facilities, and involve only a small amount of window system dependent code.
2. Integration with the window system for example displays should automatically rescale themselves to fit a resizable windows.
3. Flexible allocation of display facilities To achieve this the user defines Tv's for various purposes, e.g. image display, and multiple copies of these can be installed on different physical Tvtools.
4. Automatic provision of standard visualisation facilities For example, arbitrary scaling of 2D images on display, zooming and rotating displayed 3D objects, and changing from orthographic to perspective viewing.
5. Facilities for programming standard interaction modes such as choosing a region of interest, applying a function to a list of objects picked from the window, or associating functions with mouse button up drag and down events.
6. Provision of standard graphics utilities such as graph plotting, wire frame display, surface rendering etc.

The tv data structure in the standard TINA system is used for displaying graphics under X windows on UNIX machines, though the approach taken makes re-implementation on other graphics platforms relatively simple (requiring new versions of the "screen" functions only). It contains sufficient information to control the construction of several types of graphical display.

1. Direct graphical output to a Tv window in direct pixel coordinates.
2. Graphical display in a prespecified 2D coordinate system such as an image.
3. Graphical display in a 3D cartesian coordinate system such as generated by a 3D vision system.
4. Graph plotting routines for 2D data display (histogramming etc.)
5. Surface ploting for data visiualisation of 3D surfaces (including hidden line removal).
6. Red/green anaglyphs for stereo data display.

The standard tv libraries are thus extensive with a wide variety of graphical applications supported. During mouse interaction the graphics display uses a different set of display funtions as specified by skeldraw functions. These will correspond to reduced plotting versions of the fulldraw for reasons of speed. For example the image is normally displayed only as a wirefram mesh during zooming. Tina uses a standard naming proceedure to produce graphics where functions ending in a "3" correspond to 3D graphics, a "2" for 2D graphics and xy for direct plotting on the graphical window. These functions are generally built up as a series of wrappers around the "xy" or IPOS graphics functions and the relavent projection routines. The loweset level functions are the "screen" functions which plot data directly onto the machine dependant bit map.

The easiest way to use tv graphics is to make use of pre defined display utilities, see the chapter on Programmer Graphics Support. It is recommended that you gain some familitarity with the existing features before attempting to write your own. The simplest way to write your own TinaView routines are as a dead graphics display system. In this mode the display will not refresh itself on damage, resizing etc. and some interaction modes are not available. This is not recommended, but gives an easy introduction to the use of TinaView. More advanced programming will use the fulldraw, skeldraw and backdraw callbacks to make the application 'live'.

2.1 Installing a Tv on a Tvtool

A Tv is a logical display device. They are created by a call to

```
Tv *tv_create(char *name)
```

for example

```
display_tv = tv_create("Display");
```

Graphics can now be draw into display_tv by calls like

```
/* draw a diagonal green line **/
tv_set_color(display_tv, green);
tv_linexy(display_tv, 0, 0, 256, 256);
```

For those wishing to generate a Tv quickly for image display and manipulation, the **simple Tv** is provided as a template. The routines

```
Tv *simple_tv_make(char *name)

Imrect *simple_image_get(Tv *tv)

void simple_image_set(Tv *tv, Imrect *im)
```

use function calls described later in this chapter to provide a Tv structure which allows an image to be registered for display.

To be useful, a Tv must be installed on a Tvtool or a Display Tool. A Tvtool provides the buttons and menus needed to manage an actual display window on the screen, it is created by a call to

```
void *tv_tool(int xpos, int ypos)
```

A Display Tool, which is a cut down version of a TvTool for image or data display without interaction, can be generated by a call to

```
void *display_tool(int x, int y, int width, int height)
```

The returned pointers to void hides window system dependent data structures. The display_tv can now be installed on either of these tools by a call to

```
void    tv_install(Tv * tv, void *tv_screen)
```

This process is not irreversible, a given Tv can be associated with a Tool as required allowing the display tools to be used a free floating resources. The TvTool has an **install** button specifically for this process, a Tv can be registered as the next one for installation on any TvTool via a call to

```
void    tv_set_next(Tv * tv)
```

2.2 Output Primitives

This section describes the lowest level of graphic output, using raw screen positions. In the examples above screen positions were described by a pair of integer values (x, y). Actually the lowest level of graphics uses an integer position structure

```
typedef struct ipos
{
    Ts_id ts_id;                /* Tina structure identifier */
    int     x;
    int     y;
}       Ipos;
```

to describe positions relative to the top left-hand corner of the display window in pixel coordinates. Such positions can be created by calling the function

```
Ipos ipos(int x, int y)
```

for example

```
Ipos centre;
centre = ipos(128, 128);
```

The primitive display functions are then

```
void tv_point(Tv * tv, Ipos p)
```

- draws a single point at the given position.

```
void tv_dot(Tv * tv, Ipos p)
```

- draws a 3 by 3 square dot centred at the given position.

```
void tv_line(Tv * tv, Ipos p1, Ipos p2)
```

- draws a line between the given positions.

```
void tv_rect(Tv * tv, Ipos p1, Ipos p2)
```

- draws a rectangle with given upper left and lower right corners.

```
void tv_fillrect(Tv * tv, Ipos p1, Ipos p2)
```

- draws a filled rectangle with given upper left and lower right corners.

```
void tv_text(Tv * tv, char *string, Ipos p)
```

- writes the text string with its top right corner at the given position.

```
void tv_circle(Tv * tv, Ipos centre, int radius)
```

- draws a circle with given centre and radius.

Two more functions are described here for completeness, they are the lowest level image display functions.

```
void tv_raster(Tv * tv, int x1, int x2, int y, char *raster)
```

- puts a whole raster into the display window, going from (x1, y) to (x2, y). The pixels values used are given by the values raster[x1], ..., raster[x2] in the array of characters. (For an explanation of pixel values vs. colors see "Using Color in Tv's").

```
void tv_image(Tv * tv, int c, int r, int w, int h, char *data)
```

- puts a whole image of given width and height into the display window with the top left corner at (x, y). The pixel values are stored by rows in a data array of length width*height. Note that this storage convention is rarely used in Tina - this is a low level display function.

These low-level functions will not often be used in practice, since there are versions which perform automatic 2D and 3D projection (see "Specifying Tv Projection"). When they are used it is often more convenient to call the xy version of the function. These have 'xy' appended to the name above and have Ipos arguments replaced by the separate x and y values, e.g.

```
void tv_textxy(Tv * tv, char *string, int x, int y)
```

```
void tv_linexy(Tv * tv, int x1, int y1, int x2, int y2)
```

Only the image display functions do not have xy versions (they are xy already).

2.3 Output Attributes

The Tv stores a current draw context. This contains information about the current draw colour, raster op, line style, line width, and text font. The specified context will be used for drawing all primitives until it is changed. Note that not all the options given here will always be available, however a sensible default will be chosen. Note that some options do not work properly until at least one tvtool has been created (since this is when the colormap, fonts, etc. are loaded).

The current draw colour is set by a call to

```
void    tv_set_color(Tv * tv, Tina_pixel color)
```

Colours are described in more detail later; they are specified by integers, and global variables with colour names such as red, green, magenta etc. refer to the standard colours. The default colour is black.

```
tv_set_rop(Tv *tv, int rop)
```

Alternative raster ops will not be useful in general. Two global variables are defined, rop_copy (which is the default, and copies the primitive onto the screen) and rop_xor (which xor's the primitive onto the screen). In practice the only use for rop's is overlaying for animation purposes, this is best done using the overlay mode (see below).

```
tv_set_linestyle(Tv *tv, int style)
```

Two line styles are available at present, specified by integer global variables line_solid and line_dotted.

```
tv_set_linewidth(Tv *tv, int width)
```

This sets the line width in pixels. Width zero is the default, this is a fast line drawing mode. Widths 1 and up are much slower.

```
tv_set_font(Tv *tv, int font)
```

Graphics is displayed using the current font. Three fixed width fonts are searched for, and identified by the global variables small_font, medium_font, and large_font. If a search fails default_font will be used, this always exists. The initial font is medium_font or default_font.

So that functions drawing to the same Tv do not interfere, two functions are provided to store and reset the draw context. After making a call to

```
tv_save_draw(Tv *tv)
```

the draw context is saved. A subsequent call to

```
tv_reset_draw(Tv *tv)
```

will restore the original draw context. Repeated calls to these functions nest properly.

2.4 Overlay Mode

In order to generate interactive moving cursor functions it is necessary to produce sprite like graphical out put. This requires an "XOR" modification of the displayed graphics. After a call to

```
tv_set_overlay(Tv *tv)
```

all drawing takes place in a colour called overlay (usually red) and uses rop_xor. This means that after drawing a sequence of primitives, the screen can be restored to its original state by repeating the draw sequence. This allows red sprites to wander over a fixed background display for example.

To return to the previous draw context call tv_reset_draw(tv) as above.

2.5 The 2D Graphics Camera

The 2D camera determines the way 2D 'world' points are mapped into pixel coordinates in the graphics window. In Tina 2D floating point positions are specified by a structure

```
typedef struct vec2
{
    Ts_id ts_id;                /* Tina structure identifier */
    float   el[2];
}       Vec2;
```

they can be created with calls to

```
Vec2 vec2(double x, double y)
```

of the form

```
Vec2 p;
p = vec2(0.0, 0.5).

(For further details on vector algebra see the Tinamath documentation).
The 2D projection function can be called explicitly as

Ipos tv_proj2(Tv *tv, Vec2 p)
```

Suppose we want to draw edge points from a 512 by 512 image, but the Tv is installed on a Tvtool which currently has size 256 by 256. The call

```
tv_camera2_image(tv, 512, 512)
```

computes a projection function which scales the entire image uniformly to fit centrally in the window. For example the centre point of the image can now be drawn with a call to

```
tv_point2(tv, vec2(256.0, 256.0));
```

rather than explicitly using

```
tv_point(tv, ipos(128, 128));
```

The available ways of setting up a 2D camera are described below:

```
void    tv_camera2_image(Tv * tv, int width, int height)
```

- scales the rectangle (0, 0) to (width, height) uniformly (that is equally in x and y) so as to fit centrally in the window. As its name suggests it is usually called in Tv's used to display images. The image y-axis is draw downwards.

```
void    tv_camera2_rect(Tv * tv, Vec2 v1, Vec2 v2)
```

- scales the rectangle with lower left at p1 and upper right at p2 in both x and y to fit exactly in the window. Note this draws the 2D source y-axis pointing upwards.

```
void    tv_camera2(Tv * tv, Vec2 centre, double radius, Vec2 down)
```

This uniformly scales the circle with given centre and radius to fit centrally in the window. There is also a 2D rotation to make the vector specified by down point vertically downwards in the window.

Note that once these functions are invoked any resizing of the Tvtool will automatically cause the Tv to adjust its projection function appropriately.

2.6 The 3D Graphics Camera

The 3D camera determines the way 3D 'world' points are mapped into pixel coordinates in the graphics window. In Tina 3D floating point positions are specified by a structure

```
typedef struct vec3
{
    Ts_id ts_id;                /* Tina structure identifier */
    float   el[3];
}       Vec3;
```

they can be created with calls to

```
Vec2 vec3(double x, double y, double z)
```

of the form

```
Vec3 p;
p = vec3(0.0, 0.5, 1.0).
```

```
(For further details on vector algebra see the Tinamath documentation).
The 3D projection function can be called explicitly with

Ipos tv_proj3(Tv *tv, Vec3 p)
```

3D cameras are usually specified initially by a call to

```
void    tv_camera3(Tv * tv, Vec3 centre, double radius, double pscale,
                   Vec3 aim, Vec3 down)
```

This sets the 3D camera used by tv so that:

1. The point centre appears in the centre of the image
2. A ball of the given radius about the central point can be seen in the tv window.
3. The camera is pointing in direction aim
4. The vector down in the world will point vertically downwards in the tv window.
5. When perspective projection mode is set the camera will be at a distance pscale*radius from centre (pscale = 3.0 usually gives obvious but not exaggerated perspective).

This 'implicit' camera specification is very convenient for visualisation of 3D objects - usually Tina provides a function which gives the centre and radius of an enclosing sphere for such objects, calling tv_camera3 above with varying aim and up directions allows the observer to move round the object, keeping it always in the field of view (see the zoom facility below).

2.7 Geometrical Display

The display of 2D and 3D gemoetrical data structures is handelled automatically in an object oriented manner with standard colours used for the display of each geometrical entity using the function

```
void    geom_col_draw(Tv * tv, void *geom, int type)
```

The colours used are

```
POINT2 == blue
LINE2  == baby_blue
CONIC2 == cyan
POINT3 == blue
LINE3  == baby_blue
CONIC3 == cyan
PLANE  == magenta
```

However, if alternative colours are required in order to display quantitative results from algorithms this default can be over-ridden by attaching a *Graphic* data structure to the geometric feature's property list.

2.8 Zoom Facilities

Several standard zoom modes are provided for mouse interaction with 2D and 3D graphical displays. These are selectable via a call to

```
int     tv_set_zoomlevel(Tv * tv, int zoomlevel)
```

where the zoomlevel takes the following defined integer variables

```
IMZOOM   == standard 2D image zoom
ZOOMAF   == constrained x and y zoom
ZOOM2    == 2D image zoom including rotation
ZOOM3    == 3D data zoom
ZOOMGR   == 3D surface viewer
```

Selection of these alternatives determines how the 2D and 3D co-ordinate frames associated with each Tv are manipulated by mouse interactions. The best way to understand the capabilities of these options is to try them.

2.9 Adding Redraw Facilities

Suppose we draw a rectangle into a Tv

```
void tv_rect2(tv, vec2(0.0, 0.0) , vec2(2.0, 1.0))
```

If the tvtool on which the tv is installed is resized the display will be damaged and the rectangle lost.

This can be rectified by defining a function

```
void fulldraw(Tv *tv)
{
    tv_rect2(tv, vec2(0.0, 0.0) , vec2(2.0, 1.0));
}
```

and setting this to be the fulldraw function for the Tv with

```
tv_set_fulldraw(tv, fulldraw)
```

whenever the display is resized or damaged the function fulldraw will be called.

During an interactive zoom the display is repeatedly redrawn in overlay (xor) mode. It is often slow to redraw everything in the display, this makes the interactive zoom difficult to use. For this reason the user must define a second function to quickly draw a skeleton version of the display

```
void skeldraw(Tv *tv)
{
    ...
}
```

and set this to be the skeleton draw function for the Tv with

```
tv_set_skeldraw(tv, skeldraw)
```

Sometimes the display has an unchanging background but a foreground which must be repeatedly re-drawn. When the background is expensive to draw it may be worthwhile to define a background draw function

```
void backdraw(Tv *tv)
{
    ...
}
```

and set this to be the background draw function for the Tv with

```
tv_set_backdraw(tv, backdraw)
```

When the background changes, for example on a re-size, this function will be called to display the new background, this background is then re-displayed in future by a fast raster-op. The fulldraw function is then responsible only for drawing the foreground features.

2.10 Colour-Map Selection

Colour-maps in Tina are defined around a standard lookup table of pixel values

```
 cmap ->std_lut[0:CMAPLOOKUPSIZE])
```

for each Tv panel. These are the pixel values neccesary to generate a particular colour from the graphics server. Tina uses a set of simple rules to locate grey scales and colours within this array, there may be some duplication of colours depending upon availability of resources. The intention is to provide a software buffer between the graphics software and the Tina Tv libraries in case of future changes in the graphics environment. Graphical colours are set via the use of the Tina_Color data structure. All X graphics calls are isolated in the src/X11 directory.

There are several possible colour-maps, two primarily for grey scale display with geometric overdraws, a purple green anaglyph, which can be viewied with red/green filters for stereo vision (but also maximises visual contrast for those who are red/green colour blind). Finally there is a false colour-map which is useful for displaying absolute values from quantitative pixel measurement processes. This attempts to map the majority of the available colour space in a way which moves from low to high intesities and also cold to hot. The grey scale and false colour-maps also have red overlay planes for temporary sprite graphics. These are stored in the upper half of the colour cells and are accessed by a non destructive XOR process.

Colour-maps are allocated to a given Tv via calls to

```
Bool    tv_grey_cmap_create(Tv * tv, int base)

Bool    tv_standard_cmap_create(Tv * tv, int base)

Bool    tv_false_cmap_create(Tv * tv, int base)

Bool    tv_anag_cmap_create(Tv * tv, int base)
```

The **base** variable specifies how much of the system colour-map should be preserved (see below). Colour-map information is stored within each Tv data structure and on a list of available maps. When a new colour-map is requested this list is searched for the corresponding map (with the same cmap_create_fn() and number of default colours) before a new one is generated. The available colour cells within each Tv can be displayed from the View Tool.

2.11 Workings of the X11 Colour System

The color-maps are perhaps the most environment sensitive parts of Tina and have had to be modified to keep pace with hardware and operating system changes. For this reason this area deserves additional explanation. Under an X11 server, Tina will attempt to request a dynamic cell colour-map (Pseudo_Color). This is done at the point where the display panel and visual are requested from the server (see tw_canvas). If succesful 256 cells will be allocated (cmap_create()) and then filled according to the required RGB values. These values will swap in and out as the active Tv panel (or other X graphics panel) changes. This is a fundamental limitation of 8 bit graphics displays. There are only 256 colour cells available for the entire system and all packages must share them. An attempt is made to copy at least some of the lower cells in the default colour-map cells to the new colour-map in order to minimise flickering. The number of default colours can be modified using the View Tool. This strategy used to be a good one, but unfortunately, many recent window managers spatter frequently accessed colours all over the cell array rather than grouping them in the lower cells, making it difficult to avoid flickering. Some of these cells may even be used to define the colours of the tools causing problems with legibility of buttons. Also some common packages seem to have the capability to permanently modify the system colour-map leading to similar problems. Packages certainly cannot be used together usefully, particularly when requiring a graphics dump. This has presumably been tollerated due to the general trend away from 8 bit graphics with recent improvements in hardware which elimiate these problems (see below). The View Tool contains a postscript dump facility so that the need to use other X based packages for graphics storage can be avoided.

If dynamic colour-map generation is unsuccesful and a static colour-map is returned (True_Color), it is assumed that the X server is running with at least a 16 bit colour-map and this default colour-map is THE system colour-map. [1]. Pixel values are then found by searching for the closest RGB values in the default colour-map. The same colour-map types are available as for the 8 bit colour space so we still use a maximum of 256 colours. No attempt to copy the default colour-map is necessary and the space saved in the standard look-up table is used for extra colours. There is no flickering, but a red overlay plane can no longer be explicitly allocated and the XOR process simply generates a pixel value for a colour cell which is very dissimilar to the original.

X is particularly sensitive to attempts to query or allocate non-existent colours. In addition the nearest pixel value returned by X server queries is not hardware independent and endian swapping may be neccesary when using 16 bit colour-maps (or greater) on remote X servers. Clearly, 16 bit data may also increase the quantity of data transmitted from the server for a given image size.

[1] The returned number of available cells (64) is ignored!

Chapter 3

Generic Lists

3.1 Introduction

This chapter describes facilities that exist within the TINA system libraries for the manipulation of generic list structures. It is strongly recommended that these structures (along with the additional data structures described in the next chapter) are used to provide the connectivity amongst other data types. The core TINA system code and most of the application are written in this way. Doing so will assist the modularity, regularity and robustness of the system.

Tina list structure definitions and typedefs (amongst other things) can be included with

```
#include <tina/sys/sysDef.h>
```

function declarations with

```
#include <tina/sys/sysPro.h>
```

programs should be compiled with the library

```
-ltinaSys
```

3.2 Specifying Types

All list, and related, structure definitions use integer **type** fields to identify referenced data structures. A current list of defined data types and their associated meanings is given in the Appendix. They fall into the following categories which are grouped in terms of their numeric values

```
1......99   used to define system data structures
100...199   used to define math types
200...299   used to define Tv types
300...999   used to define Tina data structures
1000.....   available to the user
```

Note that the NULL type 0 is not used explicitly. This allows the type NULL to be used both as a catch all and in situations where it is thought unnecessary to specify the type of the elements.

3.3 Lists and Ddlists

Two basic structures exist for the representation of lists. These are singly

```
typedef struct list  /* generic list type for all simple list structures */
{
    Ts_id ts_id;                /* Tina structure identifier */
    int type;
    struct list *next;
    void *to;
} List;
```

and doubly connected

```
typedef struct ddlist  /* generic double dircted list type */
{
    Ts_id ts_id;                /* Tina structure identifier */
    int type;
    struct ddlist *next;
    struct ddlist *last;
    void *to;
} Ddlist;
```

with obvious meanings.

The field **to** provides a pointer to the data specified by **type.** The fields **next** and **last** provide simple and doubly directed manipulation.

As far as possible the names of functions for manipulating list structures conform to the following conventions. Functions acting upon single elements of a list are prefixed either **link_** or **ref_** depending whether they act upon the list element or the item referenced through it respectively. Functions that act upon lists as a whole are prefixed **list_**. Wherever possible functions for manipulating doubly directed lists have the same names and meanings except they have a further (or alternative) prefix **dd_**. In general complementary functions exist for each type of list with as similar a functionality as possible.

3.4 Property Lists

One particularly important use of the the List data structure is the property list. It is used to associate extra specific information (relevant to individual application domains) with generic feature descriptions. The usual use is exemplified in the following generic data storage definition, which can be used to add extra information to a data structure that does not itself have a property list.

```
typedef struct generic
{
    Ts_id ts_id;                /* Tina structure identifier */
    int type;
    int label;
    void *to;
    struct list *props;
} Generic;
```

The list referenced by the field **props** is treated as a property list. List elements of the property lists are identified by the reserved system type PROP_TYPE (although this is hidden from the user and therefor

largely redundant). Through a hidden data structure the list is used to reference additional specific and shared data structures associated with the defined structure. All tina structures used to define physical data (image, scene or model based) have their own property lists. Other data structures (eg. those representing transformations or projections) must be referenced through the type **Generic** in order to associate properties with them.

As implied above, property lists are not manipulated by the user, but by a limited set of access functions that allow the creation, deletion, update, and copying of properties using the data types as a reference key. The property list structure is given here purely for completeness.

```
typedef struct prop  /* a structure used to maintain property like lists */
{
    Ts_id ts_id;                /* Tina structure identifier */
    int type;
    int count;   /* the number of references through this prop */
    void (*freefunc)();
    void *to; /* must be suitably cast */
} Prop;
```

A few functions also exist for changing the order of elements on the property list to improve the efficiency of frequently accessed members, these and other props list manipulation functions will be described later.

3.5 Tina Strings

A doubly connected list indexed by its **start** and **end** is referred to as a **Tstring** (tina string).

```
typedef struct tstring /* defining arbitary strings */
{
    Ts_id ts_id;                /* Tina structure identifier */
    int type;
    struct ddlist *start;
    struct ddlist *end;
    int count;
    struct list *props; /* user definable properties */
} Tstring;
```

The property list field **props** is included in the string as they are often used to represent physical events (notably edge and poly strings used to represent image features and geometry recovered from them in both 2D and 3D). The string is defined between **start** and **end** inclusive (hence neither start nor end can be defined as NULL). The integer field **count** is the number of elements in the string. The **type** field refers to the general nature of string in question rather than to the individual idems referenced by it whose type may vary (as indicated by the type fields of the individual **Ddlist** elements).

Generic string manipulation functions are prefixed **str_** and additional **Ddlist** functions that take both the start and end of a string as parameters are prefixed **ddstr_.**

Note that a string can be defined within a more extensively connected section of **Ddlist** structures, hence the fields **last** of start and **next** of end need not be NULL.

3.6 Recursive Lists

Functions exist to allow generic lists and strings to be used recursively to define reclists. That is lists and strings whose elements may point to lists and/or strings or other data. The scope is endless and in

practice only structures a few recursions deep prove to be of much use.

All such functions are prefixed in one of three ways **reclist_list_**, **reclist_string_** or simply **reclist_** if the structure type is to be determined from a type argument. The reserved type values LIST and STRING are used to indicate extra levels of recursion from the current **list** or **ddlist** element.

3.7 Applying Functions

It is often convenient to apply a function to all members of a list (or through some other data structure used in Tina). To facilitate this a standard form of indirect function calling has been adopted with the following arguments

```
void indirect_function(void *ptr, int type, void* data)
```

The function call for applying this function to all the members of a list is

```
void    list_apply_func(List * list, void (*func) ( /* ??? */ ), void *data)
{
    List   *lptr;

    for (lptr = list; lptr != NULL; lptr = lptr->next)
        func(lptr->to, lptr->type, data);
}
```

The type field supplied to the indirectly called function is obtained from the list element. Note that for some Tina structures the associated function apply procedure provides additional arguments. These follow the standard ones to allow indirect function calls of the general type as well as more specific variants.

For example the function

```
void    er_apply_to_all_edges(Imrect * edgerect,
                              void (*edge_save_pos_prop),
                              (void *)IMPOS)
```

applies the following function call to all edges in the edgerect

```
void    edge_save_pos_prop(Edgel * edge, int type, int prop_type)
{
    Vec2   *p;

    if (type != EDGE || edge == NULL)
        return;

    p = vec2_alloc();
    *p = edge->pos;
    edge->props = proplist_addifnp(edge->props, (void *) p, prop_type, vec2_free
, true);
}
```

Where $IMPOS$ is the proplist type for the structure to be added to each edgel proplist.

Two other forms of standard function call exist in Tina, these are for copying and updating elementary data referenced by various Tina connector structures (including lists).

```
void *copy_func(void *ptr, int type, void *data)
```

```
void *update_func(void* ptr, int *type_ptr, void *data)
```

Each is very similar except that the update version is also able to change the type field by which the data is referenced.

3.8 List and Ddlist functions

This section provides prototypes and short descriptions of the useful functions available for **List** and **Ddlist** manipulation.

```
List   *link_alloc(void *ptr, int type)

Ddlist *dd_link_alloc(void *ptr, int type)
```

Allocate link of given **type** and set to point at **ptr.**

```
void    ref_set(List * el, void *ptr, int type)

void    dd_ref_set(Ddlist * el, void *ptr, int type)
```

Set list element **el** to reference **ptr** of given **type.**

```
List   *list_make(int n, int type)

Ddlist *dd_list_make(int n, int type)
```

Make and a list of length **n** of given **type** with NULL references.

```
List   *list_append(List * l1, List * l2)

Ddlist *dd_append(Ddlist * l1, Ddlist * l2)
```

Append list **l2** on to end of **l1** and return combined list.

```
List   *link_addtostart(List * list, List * el)

Ddlist *dd_link_addtostart(Ddlist * list, Ddlist * el)
```

Add new element to the start of the list returning new start of the list. If the element is NULL the old start of list will be returned.

```
List   *ref_addtostart(List * list, void *ptr, int type)

Ddlist *dd_ref_addtostart(Ddlist * list, void *ptr, int type)
```

Create a new element of given **type** referencing **ptr** and add it to the start of the list returning the new start of the list.

```
List   *link_addtoend(List * end, List * el)

Ddlist *dd_link_addtoend(Ddlist * end, Ddlist * el)
```

Add new element to the end of the list returning new end of the list.

```
List   *ref_addtoend(List * end, void *ptr, int type)

Ddlist *dd_ref_addtoend(Ddlist * end, void *ptr, int type)
```

Create a new element of given **type** referencing **ptr** and add it to the end of the list returning the new end of the list.

```
List   *list_addtoend(List * list, List * el)

Ddlist *dd_list_addtoend(Ddlist * list, Ddlist * el)
```

Add new element to back of list returning the front of the new list. The list will actually only change at the initialisation of the list. These functions are easier to use but less efficient than the **link_addtoend** functions above.

```
void    link_addafter(List * at, List * el)

void    dd_link_addafter(Ddlist * at, Ddlist * el)
```

Add new element **el** to the list after **at.**

```
List   *list_get_end(List * list)

Ddlist *dd_get_end(Ddlist * list)

Ddlist *dd_get_start(Ddlist * list)
```

Get start and end of a list from element **el.**

```
List   *link_get_by_ref(List * list, void *ptr)

List   *link_get_by_type(List * list, int type)

Ddlist *dd_link_get_by_ref(Ddlist * list, void *ptr)

Ddlist *dd_link_get_by_type(Ddlist * list, int type)
```

Get the first element in **list** referencing a given **ptr** or of a given **type.**

```
void    ref_free(List * at, void (*freefunc) ( /* ??? */ ))

void    dd_ref_free(Ddlist * at, void (*freefunc) ( /* ??? */ ))
```

Free the reference of the element **el** using **freefunc** and set the reference to NULL. The function **freefunc** is of the form

```
void freefunc(void *ptr, int type)
```

```
void    link_rm_next_el(List * at)

List   *link_rm_el(List * at)

void    dd_link_rm_next_el(Ddlist * at)

Ddlist *dd_link_rm_el(Ddlist * at)
```

Functions for removing and freeing individual list elements. Note that **link_rm_el** results in a broken list which must be corrected (if required) by the user.

```
void    link_rm_next(List * at, void (*freefunc) ( /* ??? */ ))

List   *link_rm(List * at, void (*freefunc) ( /* ??? */ ))

void    dd_link_rm_next(Ddlist * at, void (*freefunc) ( /* ??? */ ))

Ddlist *dd_link_rm(Ddlist * at, void (*freefunc) ( /* ??? */ ))
```

Functions to both remove a link and free up the reference. Again **link_rm** will result in a broken list. The function **freefunc** is of the form

```
void freefunc(void *ptr, int type)
```

```
void    list_rm_links(List * list)
```

```
List   *list_rm_links_on_type(List * list, int type)

List   *list_rm_links_on_func(List * list, Bool(*func) ( /* ??? */ ),
                              void *data)

void    list_free_refs(List * list, void (*freefunc) ( /* ??? */ ))

void    list_rm(List * list, void (*freefunc) ())

void    dd_list_rm_links(Ddlist * list)

Ddlist *dd_list_rm_links_on_type(Ddlist * list, int type)

Ddlist *dd_list_rm_links_on_func(Ddlist * list, Bool(*func) ( /* ??? */ ))

void    dd_list_free_refs(Ddlist * list, void (*freefunc) ( /* ??? */ ))

void    dd_list_rm(Ddlist * list, void (*freefunc) ( /* ??? */ ))
```

Various functions for freeing lists as a whole (predicated by type or a boolean function). The function **freefunc** is of the form

```
void freefunc(void *ptr, int type)
```

The function call in **list_rm_links_on_func** and **dd_list_rm_links_on_func** is of the form

```
Bool bool_func(void *ptr, int type)
```

```
List   *list_rm_el(List * list, List * el, void (*freefunc) ( /* ??? */ ))

List   *list_rm_ref(List * list, void *ptr, void (*freefunc) ( /* ??? */ ))

Ddlist *dd_list_rm_el(Ddlist * list, Ddlist * el, void (*freefunc) ( /* ??? */ ))

Ddlist *dd_list_rm_ref(Ddlist * list, void *ptr, void (*freefunc) ( /* ??? */ ))
```

These functions simplify the situation (at some extra cost for singly directed lists), each removes an element from a list, repairs the list and returns it. For example, the function **list_rm_el** finds the element **el** in **list** and removes it and the object it references using **freefunc.** Similarly the function **list_rm_ref** is identical except that it locates the item to be deleted by the object it references rather than explicitly. The function **freefunc** is of the form

```
void freefunc(void *ptr, int type)
```

```
List   *link_copy(List * el, void *(*cpfunc) ( /* ??? */ ), void *data)

List   *list_copy(List * list, void *(*cpfunc) ( /* ??? */ ), void *data)
```

```
List   *list_reverse(List * list)

List   *list_reversecopy(List * list, void *(*cpfunc) ( /* ??? */ ), void *data)

Ddlist *dd_link_copy(Ddlist * el, void *(*cpfunc) ( /* ??? */ ), void *data)

Ddlist *dd_list_copy(Ddlist * list, void *(*cpfunc) ( /* ??? */ ), void *data)

Ddlist *dd_list_reverse(Ddlist * list)
```

For list copying and/or reversing. Where **copy_func** is of the form

```
void *copy_func(void *ptr, int type, void *data)
```

and a NULL copy function results in the generation of new link elements and/or lists which share reference data items with their originals.

```
void    list_apply_func(List * list, void (*func) ( /* ??? */ ), void *data)

void    dd_apply_func(Ddlist * list, void (*func) ( /* ??? */ ), void *data)
```

Apply function to elements of a list. Where **func** is of the form

```
void func(void *ptr, int type, void *data)
```

```
int list_length(List *list)

int dd_list_length(Ddlist *list)
```

Return the number of elements in a list.

```
List   *sort_list(List * list, double (*valfunc) ( /* ??? */ ), void *data)

Ddlist *sort_ddlist(Ddlist * list, double (*valfunc) ( /* ??? */ ), void *data)
```

Sort list into ascending order on the basis of applying **valfunc** to each data item, where **valfunc** is of the form

```
double valfunc(void* ptr, int type, void *data)
```

3.9 String Functions

This section provides prototypes and short descriptions of the useful functions available for tina string manipulation.

```
Tstring *str_alloc()

Tstring *str_make(int type, Ddlist * start, Ddlist * end)
```

Allocate NULL and filled string structures respectively.

```
void str_rm_links(Tstring * str)

void str_free(Tstring * str, void (*freefunc) ( /* ??? */ ))

void str_rm(Tstring * str, void (*freefunc) ( /* ??? */ ))

void    ddstr_rm_links(Ddlist * start, Ddlist * end)

void    ddstr_free(Ddlist * start, Ddlist * end, void (*freefunc) ( /* ??? */ ))

void    ddstr_rm(Ddlist * start, Ddlist * end, void (*freefunc) ( /* ??? */ ))
```

Respectively these functions: remove string elements alone; free up string references and set them to NULL; both free up references and remove elements. The **str_** versions also free the **Tstring** data structure and its associated property list.

```
void str_rm_only_str(Tstring *string)
```

Frees the **Tstring** data structure and associated property list but nothing more. The function **freefunc** is of the form

```
void freefunc(void *ptr, int type)
```

```
void str_reverse(Tstring * str)

Tstring *str_copy(Tstring * str, void *(*copyfunc) ( /* ??? */ ), void *data)

Tstring *str_clone(Tstring * str)

void ddstr_reverse(Ddlist **startp,Ddlist *endp)

void    ddstr_copy(Ddlist ** startp, Ddlist ** endp,
                   void *(*copyfunc) ( /* ??? */ ), void *data)
```

Functions to copy and reverse string structures. The copy has standard tina arguments

```
void *copy_func(void *ptr, int type, void *data)
```

A NULL copy function results in the generation of new link elements and/or lists which share reference data items with their originals. Likewise **str_clone** makes a new string structure (**Tstring** and **Ddlist)** that points to the original data.

Note that string property lists are also copied.

```
Ddlist *str_link_get_by_ref(Tstring * str, void *ptr)

Tstring *str_list_get_by_ref(List * strings, void *ptr)

Ddlist *ddstr_link_get_by_ref(Ddlist * start, Ddlist * end, void *ptr)
```

Functions to identify the list element or string, from a list of strings, that references **ptr.**

```
void str_apply_func(Tstring * str, void (*func) ( /* ??? */ ), void *data)

void    ddstr_apply_func(Ddlist * start, Ddlist * end,
                         void (*func) ( /* ??? */ ), void *data)
```

Apply functions to elements of a string. Where **func** is of the form

```
void func(void *ptr, int type, void *data)
```

```
Ddlist *ddstr_mid_point(Ddlist * start, Ddlist * end)

Ddlist *ddstr_mid_point(Ddlist * start, Ddlist * end)
```

Appproximate middle point of the string.

```
Tstring *str_segment(Tstring * str, Ddlist * at)

Tstring *str_combine(Tstring * s1, Tstring * s2)
```

String segmentation and combination.

str_segment forms 2 strings from a single string. The original **string** is broken to make its end **at.** A new string is created starting at **at->next.** String segmentations that could create zero length strings return NULL and leave the original string unaffected.

str_combine creates a new string from pair of strings. The new string has **s1->end** adjacent to **s2->start.** The original strings structures are unaffected but list elements are shared with the combined version. The property list of the new string is NULL.

```
int str_length(Tstring * str)

int     ddstr_count(Ddlist * start, Ddlist * end)
```

Return the number of elements in a string.

3.10 Property List Functions

```
List   *proplist_add(List * proplist, void *ptr, int type,
                     void (*freefunc) ( /* ??? */ ))

List   *proplist_addifnp(List * proplist, void *ptr, int type,
                         void (*freefunc) ( /* ??? */ ), Bool dofree)

int     prop_set(List * proplist, void *ptr, int type, Bool dofree)
```

For adding new properties and/or replacing existing ones. The **type** field is used as a key to the data referenced by **ptr.** Property lists are able to destroy the data they refer to using **freefunc.** The function **freefunc** is of the form

```
void freefunc(void *ptr, int type)
```

Copied data can be protected with a NULL **freefunc.** When setting a property the boolean **dofree** determines whether **freefunc** should be run on the existing property. This is important when augmenting or updating an existing property.

proplist_addifnp (property list add if not present) performs **proplist_add** or **prop_set** depending on whether a property keyed by **type** exist or not. New properties are always added to the front of the list. Multiple calls to **proplist_add** with the same **type** argument result in shadowing of previous property settings.

```
void   *prop_get(List * proplist, int type)

int     prop_present(List * proplist, int type)

int     prop_ref_present(List * proplist, void *ptr)
```

Property location and retrieval functions. Each **prop_get** function finds the first property in to match the **type** or **ptr** argument.

```
List   *proplist_rm(List * proplist, int type)

List   *proplist_rm_by_ref(List * proplist, void *ptr)

List   *proplist_free(List * proplist, int type)

List   *proplist_free_by_ref(List * proplist, void *ptr)
```

Functions to remove properties (**_rm** initial postfix) or both remove properties and free references (**_free** initial postfix) of individual items in a property list and return the amended property list. Versions with the additional **_by_ref** postfix, use the value referenced by the property (if available) rather than the property itself as a key to the property list.

```
void    proplist_rmlist(List * proplist)

void    proplist_freelist(List * proplist)
```

Functions to remove properties (**_rmlist** postfix) or both remove properties and free references (**_freelist** postfix) of an entire property list.

```
List   *proplist_copy(List * proplist)
```

Make a copy of a property list. It actually references the same items through the same property structure (hence a change to a shared property will change all occurrences of it). Subsequent **rm** and **free** requests made on the property will only remove the property structure and/or free the referenced data when made by the last remaining share holder. Other requests will only update the list through which properties are accessed.

3.11 Recursive List Functions

```
void    reclist_apply(void *ptr, int ptype, void (*func) ( /* ??? */ ), int type,
                      void *data)

void    reclist_string_apply(Tstring * str, void (*func) ( /* ??? */ ), int type,
                             void *data)
```

Apply a function to all members of a recursive list that match the given **type.** The type NULL is wild and applies the function to all members of the recursive list. **reclist_list_apply** and **reclist_string_apply** use top level list and string structures respectively. Where **func** is of the form

```
void func(void *ptr, int type, void *data)
```

```
List   *reclist_list_update(List * list, void *(*func) ( /* ??? */ ), int type,
                            void *data)

Tstring *reclist_string_update(Tstring * str, void *(*func) ( /* ??? */ ),
                               int type, void *data)
```

Update all members of a recursive list that match the given **type** creating a new recursive list structure. The type NULL is wild and applies the function to all members of the recursive list. **reclist_list_update** and **reclist_string_update** use top level list and string structures respectively. Where **func** is of the form

```
void *func(void *ptr, int *type_ptr, void *data)
```

To allow the type fields of string and list structures to be updated. If **func** is NULL then elements are copied.

```
List   *reclist_list_flat(List * list, void *(*func) ( /* ??? */ ), int type,
                          void *data)

List   *reclist_string_flat(Tstring * str, void *(*func) ( /* ??? */ ), int type,
                            void *data)
```

Update all members of a recursive list that match the given **type** creating a simple (flat) list structure of type **List.** The type NULL is wild and applies the function to all members of the recursive list. **reclist_list_update** and **reclist_string_update** use top level list and string structures respectively. Where **func** is of the form

```
void *func(void *ptr, int *type_ptr, void *data)
```

To allow the type fields of string and list structures to be updated. If **func** is NULL then elements are copied.

```
List   *reclist_list_free(List * list, void (*freefunc) ( /* ??? */ ), int type,
                          void *data)

Tstring *reclist_string_free(Tstring * str, void (*freefunc) ( /* ??? */ ),
                             int type, void *data)
```

Free recursive lists using the function **freefunc** to free elements. The function **freefunc** is of the form

```
void freefunc(void *ptr, int type)
```

Chapter 4

Serialisation

4.1 Functionality

The serialisation process within TINA allows arbitrarily complex data structures to be packed into a linear array of data, transmitted to another processor and reconstructed for continued processing. The mechanism handles selected pointers for data structures which may be embedded in the data and can eliminate repeated data referencing (see **repeat_checker()**). This latter facility can be turned off via a call to **repeatp_checking_set()**.

The information needed to serialise/de-serialise a TINA data structure is held in the global array variable tina_data_info_array. During use this array is indexed according to the TINA data type to recover serialisation/deserialisation function pointers. This array is defined in the file tstruct_info.h and the default elements of the array are defined in tstruct_info.c . Modifications to these default settings is made by the routine **define_all_ts_handlers()**. This is useful for modifying the basic functionality without having to recompile the entire TINA library. The tina_data_info_array has slots allocated for additional user defined structures (User_type_0_id - User_type_9_id) which can be used during algorithm development.

The ts_id flag determines which routines will be called during the serialisation and deserialisation process.

```
tina_data_info_array[*(int *) data - ts_base_id].serialise()
tina_data_info_array[*(int *) data - ts_base_id].deserialise()
```

The default version of these routines are **ts_serialise()** and **ts_deserialise()** respectively.

Serialisation proceeds by first processing the specified data structure to produce a linked list of all tagged TINA data structure. The type field of the List data structure is used to encode either the size of the data structure (if it is stored with this entry) or a negative offset to a previous entry in the list if it is repeated. This list is then written out as a sequential series of numbers specifying all of the data types present (or offsets) followed by the data contents of these structures. The whole lot is prefixed by the total number of data structures serialised and a magic number to encode byte reversal. During deserialisation the index list is read back and memory allocated for each data structure. The routines which handle this bottom level data transfer are fread_ts() and fwrite_ts().

4.2 Data Parsing

The serialisation mechanism relies upon a parse string to control the function **seialise_parse()** to process the data from each data structure. This is stored in

```
tina_data_info_array[*(int *) data - ts_base_id].swap_string
```

where ts_bas_id is the base address of the TINA structure ts_id tags and defined by ts_base_id = TS_ID_BASE.

The basic data types are as follows:

```
k = unsigned short
b = unsigned char
u = unsigned int
h = short
c = char
i = int
f = float
g = double
. = pointer (generally not usable after transfer).
w = function pointers ( generally not usable after transfer).
, = filler (see below)
o = unknown (but 4 bytes long)
l = unknown (but 4 bytes long)
e = unknown (but 8 bytes long)
```

These leave the transferred data unmodified on reconstruction.

The most important (tagged) data types are:

```
t = Previously defined TINA data structure
p = pointer to TINA data structure
s = pointer to string ( char *)
```

as these specify a recursion down the structure to serialise and transfer further TINA data.

The swap-string variable is defined starting from the first variable defined in a data structure following the Ts_id.

This string can contain real numbers (0-10) which encode the number of repeated data types. The repeated data type may be parenthasized (with either "(" or "[" bracket types) so that complicated structures can be represented concisely. It is perhaps inevitable that some data may be redundant when transmitted to another processor, a repeat number of "0" signals the serialisation code to ignore this item. Trailing items can also be ignored by prematurely truncating the parse string. The variables which are communicated during the serialisation process are thus completely specifiable by the user.

eg:

```
typedef struct conic
{
    Ts_id ts_id;                /* Tina structure identifier */
    unsigned int type;
    unsigned int label;
    int filler1;
```

```
    double a, b, c, d, e, f;   /* algebraic formula */
    double theta, alpha, beta;
    struct vec2 center;
    int filler2;
    double t1, t2;       /* conic params of p1 and p2 */
    int branch;          /* for hyperbola only */
    List *props;         /* property list */
} Conic;

#define CONIC_SWAP_STRING "uuiggggggggtiggip"
```

The string allows the use of comments delimited by "/" eg:

```
TS_SWAP_STRING_SET(Line3,"/type/u/label/u/p1/t/p2/t/p/t/v/t/length/f/props/p"
```

4.3 Data Flexibility

As all data structures must have a ts_id, as defined in the TINA enumerated type, in order to be processed. A potential problem may arise when trying to construct flexible data structures for base C defined variables. For this reason the basic structure types "ts_int", "ts_float", "ts_double" and "ts_string" have been defined and should be used whenever a data structure needs to point to a simple C variable. The best example of this is the List data structure.

```
typedef struct list  /* generic list type for all simple list structures */
{
    Ts_id ts_id;               /* Tina structure identifier */
    int type;
    struct list *next;
    void *to;
} List;
#define LIST_SWAP_STRING "ipp"
```

If the list− >to field is required to point to float variables the "ts_float" structure must be used.

Also the mechanism cannot handle function pointers (as this must be processor dependent data) and cannot handle variable length structures such as arrays directly. These have to be handled by purpose written routines (eg: nvector_ts_serialise()). Potentially this problem could be rectified relatively simply with a slight modification to the parsing algorithms but this will not be necessary if programmers stick with the TINA List data structure for constructing variable length data sets.

4.4 Utilities

Several utilities have been provided with which to convert serialised data to and from TINA data files. These can be found in the top level "serialise" directory within TINA. They include routines for visualising serialised data directly for example:

```
tsgeom2tv (TINA geometry)
tsimseq2tv (grey-level image sequences)
```

```
tscimseq2tv (for colour image sequences)
```

These routines can be found in the top level "serialise" directory in Tina and provide a useful means for testing some aspects of modified algorithms.

4.5 General Programming Advice

In order to extend the existing software to deal with new algorithms the serialisation mechanism needs to be extended to parse any new data structures. This can be done by following the definitions and examples given above. The new SWAP_STRING definition statement should be included underneath the structure definitions in the relavant header file. In addition the serialisation routines and swap string variables need to be stored, in the tina_data_info_array by the function **define_all_ts_handlers()**, in the user defined slots (tset.c). When porting the serialisation code to previously unsupported processors slightly more care will need to be taken (see machine dependence below).

4.6 Machine Dependance

In principle the serialisation process can transfer between different machine architectures. It automatically stores and interprets a "magic" number in order to cope with byte reversal between (for example) Transputers and SUN workstations. This is handled by the function **word_swap()** which is controlled by a global variable within the serialisation code, at the beginning of the deserialisation process (the serialisation process does not need to be concerned with byte ordering of course.

The standard way of communicating between different machine architectures is via conversion to an intermediate machine independent format, such as ANSI (for example uuencode on UNIX systems). However, such a process is unwieldy and not appropriate for real-time processing systems. In the interest of speed the serialisation process performs a low-level write operation directly on the data. This places a limitation on the data structures which can be serialised and communicated between architectures reliably. For example some SUN machines will not permit a "double" variable to be specified on an odd machine address. This problem can often be rectified by careful arrangement of variables within a structure but on some occasions this cannot be achieved. This is handled by inserting an extra "int filler" variable to realign the structure (see Conic above, the smart user will notice that in this particular case the filler variables could actually have been avoided!).

The serialise code also assumes that all variables are stored in the same position relative to the start of the structure and in the same machine representation (floats and doubles etc.). The size of data structure written out to file is specified by a "sizeof" declaration and the same value is read back from the file and used to allocate the memory for the data structure later. It is clear that once again this assumption was made on the basis of speed considerations (it would have been safer to copy the data from a temporary stored file into a separately allocated TINA data structure which was allocated in the usual manner.) This of course may not always be the case. This problem will of course need to be reconsidered when porting to any new architectures. A simple first step check would be to compare the size of all TINA data structures calculated by **sizeof()** on different machines.

Chapter 5

Tina Maths Library

Tinamath comprises routines for 2D and 3D vector geometry, matrix algebra, common numerical methods, utilities such as random number generation etc. Many of these are very similar to those described in "Numerical Recipies", except for a slight re-write to make them compatable with Tina data structures and memory management. Guaranteeing numerical stability is a difficult process and these routines should always be tested and an attempt made to understand their strengths and weaknesses before use. These standard packages are described briefly below.

Tinamath structure definitions and typedefs can be included with

```
#include <tina/math/mathDef.h>
```

function declarations with

```
#include <tina/math/mathPro.h>
```

programs should be compiled with (at least)

```
-lm -ltinaSys -ltinaMath
```

5.1 Util: Maths Utilities

Util is a collection of miscellaneous utility functions, only the complex variable and random variabler utilities are described here.

Complex numbers are represented as structures

```
typedef struct tcomplex
  {
      Ts_id   ts_id;            /* Tina structure identifier */
      double   x;
      double   y;
  }
Complex;
```

The internal structure should not be used. The real and imaginary parts of a complex should be got and set using statements like:

```
cmplx_re(z) = x;
y = cmplx_im(z);
```

(this is allowed since cmplx_re() and cmplx_im() are macros).

A complex number with given components can be constructed using

```
i = cmplx(0.0, 1.0);
```

and operations on complex numbers are implemented as functions, e.g.

```
w = z/(z+2+3i)
```

translates as

```
w = cmplx_div(z, cmplx_sum(z, cmplx(2.0, 3.0)));
```

The random number generators all call the C-library function random() and can be seeded with

```
srandom(int seed)
```

if desired. Functions are provided for returning random bits, uniform random integers, and uniform and gaussian real variables. A function for return confidences in chi square variables is included.

5.1.1 Complex Variables

```
Complex cmplx(double x, double y)
```

Returns the complex number x+yi with real part x and imaginary part y.

```
Macro cmplx_re(Complex z);
Macro cmplx_im(Complex z);
```

Return the real and imaginary parts of z respectively. Implemented as macros, so can be used on both sides of assignment statements.

```
Complex cmplx_sum(Complex a, Complex b)
```

Returns the complex sum a+b of two complex numbers.

```
Complex cmplx_diff(Complex a, Complex b)
```

Returns the complex difference a-b of two complex numbers.

```
Complex cmplx_prod(Complex a, Complex b)
```

Returns the complex product ab of two complex numbers.

```
Complex cmplx_times(double a, Complex b)
```

Returns the complex product ab of a real and a complex number.

```
Complex cmplx_zero()
```

Returns the complex zero (0+0i).

```
Complex cmplx_unit()
```

Returns the complex unit (1+0i).

```
Complex cmplx_conj(Complex z)
```

Returns the complex conjugate (x-yi) of x+yi.

```
Complex cmplx_div(Complex a, Complex b)
```

Returns the complex quotient a/b of a and b.

```
double cmplx_mod(Complex z)
```

Returns the real modulus |z| of complex z.

```
double cmplx_mod2(Complex z)
```

Returns the real squared modulus of complex z.

```
Complex cmplx_sqrt(Complex z)
```

Returns the principal value of the complex square root of complex z.

5.1.2 Random Variables

Random variables generated to conform to a specific probability distribution are fundamental to the monte-carlo process of algorithm testing and calibration. Tina provides a set of routines to support this approach. These are :

```
int rand_bit()
```

Returns a random bit (i.e. 0 or 1 with probabilities 0.5).

```
int rand_int(int a, int b)
```

Returns a uniformly distributed random integer x with a <= x < b, (i.e. x never takes the value b).

```
double rand_1()
```

Returns a uniformly distributed random double between 0.0 and 1.0.

```
double rand_unif(double x, double y)
```

Returns a uniformly distributed random double between x and y.

```
double rand_normal(double mu, double sigma)
```

Returns a normally distributed random double with mean mu and standard deviation sigma.

```
double chisq(double x, int n)
```

Returns the confidence that a chi squared variable with n degrees of freedom is less than or equal to x.

5.2 2D and 3D vector geometry

2D and 3D geometry is quite fundamental to the model based approach to computer vision and even when algorithms are not specifically model based they are still often a compact way to construct and implement an algorithm. In Tina vectors are represented as structures, for example a 3-vector is defined as:

```
typedef struct vec3
{
    Ts_id ts_id;                /* Tina structure identifier */
    float   el[3];
}       Vec3;
```

The internal structure should not be used. The x-component of a 3-vector should be got and set using statements like:

```
vx = vec3_x(v);
```

and

```
vec3_x(v) = vx;
```

(this is allowed since vec3_x() is a macro).

A 3-vector with given components can be constructed using the function vec3(), e.g.

```
v = vec3(1.0, 2.0, 3.0);
```

A fairly complete set of vector algebra and geometry functions is provided, look before you write your own. Most of these functions pass arguments by value and return a structure. Though there is a slight overhead in passing a structure rather than a pointer the advantage is that translating vector algebra into code can be almost transparent. For example a function to find the component of a vector u perpendicular to a unit vector v

```
p = u-(u.v)v
```

can be defined by

```
Vec3    vec3_projperp(Vec3 u, Vec3 v)  /**component of  u  perpendicular
                                                          to unit  v**/
{
    return (vec3_diff(u, vec3_times(vec3_dot(u, v), v)));
}
```

Occasionally 2-vectors with integer components are needed (e.g. to represent positions on a graphics window). A structure Ipos (integer position) is provided to deal with this, together with a limited set of functions to manipulate them.

5.2.1 3D Vector Algebra Functions

```
Vec3 *vec3_alloc()
```

Return a pointer to a new (zero) Vec3

```
Vec3 *vec3_make(Vec3 u)
```

Return a pointer to a copy of u.

```
void vec3_free(Vec3 *v)
```

For completeness, just calls rfree.

The above functions are generally used to create lists of vectors, there is seldom any other reason to use alloc-ed vectors.

```
Vec3 vec3(double x, double y, double z)
```

Returns vector with components (x, y, z).

```
Vec3 vec3_zero()
Returns zero vector.
```

```
Vec3 vec3_ex()

Vec3 vec3_ey()

Vec3 vec3_ez()
```

Return unit vectors in x, y or z directions.

```
Macro vec3_x(Vec3 v)

Macro vec3_y(Vec3 v)

Macro vec3_z(Vec3 v)
```

Macros to get x, y, z components of vector (can be used on both sides of assignment statements).

```
double vec3_get_x(double v)

double vec3_get_x(double v)

double vec3_get_x(double v)
```

Function versions of above macros useful for passing as arguments to functions.

```
void vec3_comps(Vec3 v, float *x, float *y, float *z)
```

Recover x, y and z components from a vector v all at once.

```
Vec3 vec3_sum(Vec3 v, Vec3 w)

Vec3 vec3_sum3(Vec3 u, Vec3 v, Vec3 w)

Vec3 vec3_sum4(Vec3 u, Vec3 v, Vec3 w, Vec3 x)
```

Returns sum of two three or four vectors.

```
Vec3 vec3_minus(Vec3 v)
```

Returns negative of v.

```
Vec3 vec3_diff(Vec3 v, Vec3 w)
```

Returns difference v-w of two vectors.

```
Vec3 vec3_times(double k, Vec3 v)
```

Returns product of a scalar and a vector kv.

```
double vec3_dot(Vec3 v, Vec3 w)
```

Returns dot (scalar) product of two vectors.

```
Vec3 vec3_cross(Vec3 v, Vec3 w)
```

Returns cross (vector) product of two vectors.

```
Vec3 vec3_unitcross(Vec3 v, Vec3 w)
```

Returns unit vector in direction of cross product of two vectors.

```
double vec3_mod(Vec3 v)
```

Returns modulus (length) of a vector.

```
double vec3_sqrmod(Vec3 v)
```

Returns squared modulus of a vector.

```
Vec3 vec3_unit(Vec3 v)
```

Returns unit vector in direction of a vector.

```
double vec3_modunit(Vec3 v, Vec3 *e)
```

Returns modulus of a vector and sets *e to unit vector in direction of vector, useful for checking if v is zero before using *e.

```
double vec3_dist(Vec3 v, Vec3 w)
```

Returns distance between two 3D points.

```
double vec3_sqrdist(Vec3 v, Vec3 w)
```

Returns squared distance between two 3D points.

```
double vec3_angle(Vec3 v, Vec3 w)
```

Returns angle between two vectors.

```
Vec3 vec3_perp(Vec3 v)
```

Returns a vector (there are many!) perpendicular to v.

```
void vec3_basis(Vec3 aim, Vec3 down, Vec3 *ex, Vec3 *ey, Vec3 *ez)
```

Sets up a unit basis with z-axis along aim and y-axis in plane of vectors aim and down (we usually think of z-axes as line of sight and y-axes point down, as in images). If this fails (aim = down) a sensible y-axis is chosen (this can lead to discontinuous changes in basis as down changes, this is unavoidable).

```
Vec3 vec3_read(File *fp)
```

Return next vector read in from ascii file, on error warns and returns zero vector.

```
void vec3_print(File *fp, Vec3 v)
```

Writes out vector to ascii file.

```
void vec3_pprint(File *fp, char *msg, Vec3 v)
```

Prints message and vector to ascii file.

```
void vec3_format(Vec3 v)
```

Writes out vector to Tina format destination (stdout or main textsw).

5.2.2 3D Vector Geometry

Functions for doing 3D geometry of points lines, planes and circles using the following vector descriptions:

```
typedef struct vec3
{
    Ts_id ts_id;                /* Tina structure identifier */
    float   el[3];              /* point in 3D coordinates */
} Vec3;

typedef struct line3
{
    Ts_id ts_id;                /* Tina structure identifier */
    unsigned int type;
    unsigned int label;
    struct vec3 p1,p2;         /* end points */
    struct vec3 p;             /* point on the line */
    struct vec3 v;             /* direction vector */
    float length;
    struct list *props;
} Line3;

typedef struct plane
{
    Ts_id ts_id;                /* Tina structure identifier */
    unsigned int type;
    unsigned int label;
    struct vec3 p;             /* a point on the plane */
    struct vec3 n;             /* normal to plane */
    struct list *props;
} Plane;

typedef struct conic
{
    Ts_id ts_id;                /* Tina structure identifier */
    unsigned int type;
    unsigned int label;
    int filler1;
    double a, b, c, d, e, f;   /* algebraic formula */
    double theta, alpha, beta;
    struct vec2 center;
    int filler2;
    double t1, t2;       /* conic params of p1 and p2 */
    int branch;          /* for hyperbola only */
    List *props;         /* property list */
} Conic;
```

```
Vec3 vec3_midpoint(Vec3 q1, Vec3 q2)
```

Returns midpoint of line from q1 to q2.

```
Vec3 vec3_projperp(Vec3 u, Vec3 v)
```

Returns projection of u perpendicular to unit vector v.

```
Vec3 vec3_projpar(Vec3 u, Vec3 v)
```

Returns projection of u parallel to unit vector v.

```
Vec3 vec3_proj_on_line(Vec3 q, Vec3 l, Vec3 v)
```

Returns perpendicular projection of point q onto line (l, v) (i.e. closest point on line).

```
Vec3 vec3_proj_on_plane(Vec3 q, Vec3 p, Vec3 n)
```

Returns perpendicular projection of point q onto plane (p, n) (i.e. closest point on plane).

```
Vec3 vec3_proj_line_on_plane(Vec3 l1, Vec3 v1, Vec3 p, Vec3 n, Vec3 *l2,
                             Vec3 *v2)
```

Sets line (*l2, *v2) which is perpendicular projection of line (l1, v1) onto plane (p, n).

```
Vec3 vec3_closest_lines(Vec3 l1, Vec3 v1, Vec3 l2, Vec3 v2)
```

Returns point on line (l1, v2) closest to line (l2, v2).

```
Vec3 vec3_inter_lines(Vec3 l1, Vec3 v1, Vec3 l2, Vec3 v2)
```

Returns point closest to two lines (i.e. their intersection if it exists).

```
Vec3 vec3_inter_line_plane(Vec3 l, Vec3 v, Vec3 p, Vec3 n)
```

Returns point of intersection of line (l, v) with plane (p, n).

```
void vec3_inter_planes(Vec3 p1, Vec3 n1, Vec3 p2, Vec3 n2, Vec3 *l, Vec3 *v)
```

Sets line (*l, *v) of intersection of two planes (p1, n1), (p2, n2).

```
void    vec3_join_2_points(Vec3 q1, Vec3 q2, Vec3 * l, Vec3 * v)
```

Sets line (*l, *v) joining two points q1, q2, *l is midpoint, *v points from q1 to q2.

```
void    vec3_join_3_points(Vec3 q1, Vec3 q2, Vec3 q3, Vec3 * p, Vec3 * n)
```

Sets plane (*p, *n) though 3 points q1, q2, q3. *p is centroid of points.

```
void    vec3_join_point_line(Vec3 q, Vec3 l, Vec3 v, Vec3 * p, Vec3 * n)
```

Sets plane (*p, *n) though points q, and line (l, v). *p is at midpoint of p and l.

```
void    vec3_join_lines(Vec3 l1, Vec3 v1, Vec3 l2, Vec3 v2, Vec3 * p, Vec3 * n)
```

Sets plane (*p, *n) though two lines (l1, v1) and (l2, v2). If they are not coplanar, plane is parallel to both lines, and passes through point of closest approach (*p is not chosen as closest approach).

```
double  vec3_dist_point_plane(Vec3 q, Vec3 p, Vec3 n)
```

Returns perpendicular distance of point q from plane (p, n).

```
double  vec3_dist_point_line(Vec3 q, Vec3 l, Vec3 v)
```

Returns perpendicular distance of point q from line (l, v).

```
double  vec3_dist_lines(Vec3 l1, Vec3 v1, Vec3 l2, Vec3 v2)
```

Returns perpendicular distance between lines (l1, v1), (l2, v2).

```
void    vec3_circ_3_points(Vec3 p1, Vec3 p2, Vec3 p3, Vec3 * centre,
                           Vec3 * normal, double *radius)
```

Sets circle (*c, *n, *r) through three points p1, p2, p3, (normal gives correct ordering for right handed rotations).

```
Bool    vec3_collinear(Vec3 p1, Vec3 p2, Vec3 q1, Vec3 q2,
                       double dotth1, double dotth2)
```

Checks whether line segments with endpoints (p1, p2) and (q1, q2) are collinear. This routine is a useful starting point but makes no attempt to model any inderlying statistics properly and should only be used with caution or preferably re-written for your own statistical model.

```
double  vec3_closest_app(Vec3 p1, Vec3 v1, Vec3 p2, Vec3 v2,
                         Vec3 * c1, Vec3 * c2)
```

Sets point *q1 on (l1, v1) closest to (l2, v2) and point *q2 on (l2, v2) closest to (l1, v1).

```
Bool    vec3_parallel(Vec3 v1, Vec3 v2, double dotthres)
```

Checks for parallelism of unit vectors using a threshold on their dot product. (So dotthres = cosine of angle thresh, e.g. it will be close to 1.0).

5.2.3 3D Matrix Algebra Functions

Note: this is not to be confused with the general matrix algebra package.

3D Matrices are structures containing a 3 by 3 float array (try not to use this). Functions usually pass matrices by value and return matrices. Again the convenience is worth the overhead.

```
void *mat3_alloc()
```

Returns pointer to a newly allocated zero matrix.

```
void *mat3_make(Mat3 m)
```

Returns pointer to a newly allocated copy of m.

```
void mat3_free(Mat3 *m)
```

Frees an allocated matrix (for completeness, just calls rfree).

```
Mat3    mat3(double mxx, double mxy, double mxz,
             double myx, double myy, double myz,
             double mzx, double mzy, double mzz)
```

Returns matrix with given components (arguments in row order).

```
Mat3 mat3_unit()
```

Returns unit matrix.

```
Mat3 mat3_zero()
```

Returns zero matrix.

```
Macro mat3_xx()

Macro mat3_xy()

Macro mat3_zz()
```

Macros to return components of matrix (can be used on left and right hand sides of assignment statements).

```
double mat3_get_xx()

double mat3_get_xy()

...

double mat3_get_zz()
```

Return components of matrix (function versions of above macros).

```
void    mat3_comps(Mat3 m, float *mxx, float *mxy, float *mxz,
                           float *myx, float *myy, float *myz,
                           float *mzx, float *mzy, float *mzz)
```

Sets *mxx etc. to components of matrix m.

```
Vec3 mat3_rowx(Mat3 m)

Vec3 mat3_rowy(Mat3 m)

Vec3 mat3_rowz(Mat3 m)
```

Return rows of matrix as Vec3's.

```
Vec3 mat3_colx(Mat3 m)

Vec3 mat3_coly(Mat3 m)

Vec3 mat3_colz(Mat3 m)
```

Return columns of matrix as Vec3's.

```
Mat3 mat3_of_cols(Vec3 cx, Vec3 cy, Vec3 cz)
```

Returns matrix with given column vectors.

```
Mat3 mat3_of_rows(Vec3 rx, Vec3 ry, Vec3 rz)
```

Returns matrix with given row vectors.

```
Mat3 mat3_sum(Mat3 m, Mat3 n)
```

Returns sum m+n of two matrices.

```
Mat3 mat3_diff(Mat3 m, Mat3 n)
```

Returns difference m-n of two matrices.

```
Mat3 mat3_minus(Mat3 m)
```

Returns negative -m of matrix.

```
Mat3 mat3_times(double k, Mat3 m)
```

Returns (matrix) product km of a scalar and a matrix.

```
Mat3 mat3_prod(Mat3 m, Mat3 n)
```

Returns matrix product mn of two matrices.

```
Mat3 mat3_inverse(Mat3 m)
```

Returns inverse of a matrix (algorithm is usually plenty good enough for 3 by 3's).

```
Mat3 mat3_transpose(Mat3 m)
```

Returns transpose of a matrix.

```
double mat3_trace(Mat3 m)
```

Returns trace (sum of diagonal elements).

```
double mat3_det(Mat3 m)
```

Returns determinant of matrix.

```
Bool mat3_posdef(Mat3 m)
```

```
Returns true if symmetric matrix is positive definite.
```

```
Vec3 mat3_vprod(Mat3 m, Vec3 v)
```

Returns (vector) product mv of matrix and vector.

```
Vec3 mat3_transpose_vprod(Mat3 m, Vec3 v)
```

Returns (vector) product m'v of transpose of matrix and vector.

```
double mat3_sprod(Vec3 v, Mat3 m, Vec3 w)
```

Returns (scalar) contraction v'mw of matrix with two vectors.

```
Mat3    mat3_tensor(Vec3 v, Vec3 w)
```

Returns (matrix) tensor product mij = vi*vj of two vectors.

```
Mat3    mat3_read(FILE *fp)
```

Returns matrix read from ascii file.

```
void    mat3_print(Mat3 fp, FILE *m)
```

Prints matrix to ascii file.

```
void    mat3_pprint(FILE * fp, char *msg, Mat3 m)
```

Prints matrix (with introductory message) to ascii file.

```
void    mat3_format(Mat3 m)
```

Prints matrix to Tina format output (stdio or main textsw).

```
void    mat3_eigen(Mat3 m, double *eval, Vec3 * evec)
```

Returns the eigen values and eigen vectors of the matrix.

5.2.4 2D Vector Algebra Functions

These have similar functionality to the corresponding 3D functions. Only those which are different are commented on here.

```
void *vec2_alloc()

void *vec2_make(Vec2 u)

void vec2_free(void *v)

Vec2 vec2(double x, double y)

Vec2 vec2_zero()

Vec2 vec2_ex()

Vec2 vec2_ey()

void vec2_comps(Vec2 v, double *x, double *y)

Macro vec2_x(Vec2 v)

Macro vec2_y(Vec2 v)

double vec2_get_x(Vec2 v)

double vec2_get_y(Vec2 v)

Vec2 vec2_sum(Vec2 v, Vec2 w)

Vec2 vec2_sum3(Vec2 u, Vec2 v, Vec2 w)

Vec2 vec2_sum4(Vec2 u, Vec2 v, Vec2 w, Vec2 x)

Vec2 vec2_minus(Vec2 v)

Vec2 vec2_diff(Vec2 v, Vec2 w)

Vec2 vec2_times(double k, Vec2 v)

double vec2_dot(Vec2 v, Vec2 w)

double vec2_mod(Vec2 v)

double vec2_sqrmod(Vec2 v)

double vec2_modunit(Vec2 v, Vec2 *e)

Vec2 vec2_unit(Vec2 v)

double vec2_dist(Vec2 v, Vec2 w)

double vec2_sqrdist(Vec2 v, Vec2 w)
```

```
Vec2 vec2_read(FILE *fp)

void vec2_print(FILE *fp, Vec2 v)

void vec2_pprint(FILE *fp, char *msg, Vec2 v)

void vec2_format(Vec2 v)
```

In addition the function

```
void vec2_basis(Vec 2 up, Vec2 *ex, Vec2 *ey)
```

sets *ex and *ey to basis vectors with the y-vector in the direction up and the x-vector rotated 90 degrees clockwise from this.

```
Vec2 vec2_perp(Vec2 v)
```

Returns a unit vector perpendicular to v rotated 90 degrees anticlockwise from v.

```
double vec2_cross(Vec2 v, Vec2 w)
```

Returns the scalar cross product vx*wy-vy*wx of v and w. (This is the signed are of the parallelogram they span).

```
double vec2_angle(Vec2 v, Vec2 w)
```

Returns the anti-clockwise angle from v to w (lies between -pi and pi).

5.2.5 2D Vector Geometry Functions

These have similar functionality to the corresponding 3D functions. Only those which are different are commented on here.

```
Vec2 vec2_midpoint(Vec2 q1, Vec2 q2)

Vec2 vec2_projperp(Vec2 u, Vec2 v)

Vec2 vec2_projpar(Vec2 u, Vec2 v)

Vec2 vec2_proj_on_line(Vec2 q, Vec2 l, Vec2 v)

Vec2 vec2_inter_lines(Vec2 l1, Vec2 v1, Vec2 l2, Vec2 v2)

void vec2_join_2_points(Vec2 q1, Vec2 q2, Vec2 *l, Vec2 *v)

double vec2_dist_point_line(Vec2 q, Vec2 l, Vec2 v)

void vec2_circ_3_points(Vec2 p1, Vec2 p2, Vec2 p3, Vec2 *centre, double radius)
```

5.2.6 2D Matrix Algebra Functions

These have similar functionality to the corresponding 3D functions. Only those which are different are commented on here.

```
void *mat2_alloc(void)

void *mat2_make(Mat2 n)

void mat2_free(void *m)

Mat2 mat2(double mxx, double mxy, double myx, double myy)

Mat2 mat2_unit(void)

Mat2 mat2_zero(void)

void mat2_comps(Mat2 m, float *mxx, float *mxy, float *myx, float *myy)

Vec2 mat2_rowx(Mat2 m)

Vec2 mat2_rowy(Mat2 m)

Vec2 mat2_colx(Mat2 m)

Vec2 mat2_coly(Mat2 m)

Mat2 mat2_of_rows(Vec2 rx, Vec2 ry)

Mat2 mat2_of_cols(Vec2 cx, Vec2 cy)

Mat2 mat2_sum(Mat2 m, Mat2 n)

Mat2 mat2_diff(Mat2 m, Mat2 n)

Mat2 mat2_prod(Mat2 m, Mat2 n)

Mat2 mat2_times(double k, Mat2 m)


Mat2 mat2_minus(Mat2 m)

Mat2 mat2_inverse(Mat2 m)

Mat2 mat2_transpose(Mat2 m)

double mat2_trace(Mat2 m)

double mat2_det(Mat2 m)

Bool mat2_posdef(Mat2 m)

Vec2 mat2_vprod(Mat2 m, Vec2 v)
```

```
double mat2_sprod(Vec2 v, Mat2 m, Vec2 w)

Mat2 mat2_tensor(Vec2 v, Vec2 w)

Mat2 mat2_read(FILE *fp)

void mat2_print(FILE *fp, Mat2 m)

void mat2_pprint(FILE *fp, char *msg, Mat2 m)

void mat2_format(Mat2 m)
```

Transformations (Rotations and Translations) in 3D

Rotations are represented by 3-Matrices (Mat3's, see above) of the active rotation i.e. the rotation represented by a matrix R takes the position x to the position Rx.

Functions have been supplied to return the matrix of a rotation with given angle and axis, and to recover the angle and axis from a rotation.

Rigid motions are represented by a Transform3 structure

```
typedef struct transf3
{
    Ts_id ts_id;                /* Tina structure identifier */
    unsigned int type;
    unsigned int label;
    Transform3 T;
    struct list *props;
} Transf3;
```

which represents a rotation R followed by a translation t

```
    x -> Rx+t .
```

Functions for applying a transform or its inverse to a vector have been supplied. As well as functions for representing the action in terms of coordinate frames.

Alternative representations, such as quaternions for rotations or 4x4 matrices for transformations were considered and rejected. Those adopted seem to offer the correct combination of convenience and simplicity.

5.2.7 3D Rotation Functions

```
Mat3    rot3(double theta, Vec3 axis)
```

Returns the matrix representing a rotation by an angle theta (in radians) about a (not necessarily unit) vector axis.

```
Mat3    rot3_1(Vec3 axis)
```

An alternative version of the above function in which the angle of roation theta is encoded in the length of the vector axis.

```
void    rot3_angle_axis(Mat3 m, double *theta, Vec3 * axis)
```

Sets *theta and *axis to be the angle of rotation and unit vector axis of rotation repectively of the rotation matrix r.

5.2.8 3D Transformation Functions

```
Transform3 *trans3_alloc(void)
```

Returns a pointer to a new (unit) Transform3.

```
Transform3 *trans3_make(Mat3 R, Vec3 t)
```

Returns a pointer to a new (unit) Transform3 with given rotation and translation.

```
Transform3 *trans3_copy(Transform3 * transf)
```

Returns a pointer to a new copy of *transf.

```
void    trans3_free(void *transf)
```

Frees the Transform3 pointed to by transf (provided for completeness, it just calls rfree).

```
Transform3 trans3_unit(void)
```

Returns a unit transformation (i.e. one that leaves vectors unchanged).

```
Transform3 trans3_prod(Transform3 transf2, Transform3 transf1)
```

Returns the composition of the transformations transf2 and transf1 (i.e. transf1 first then transf2).

```
Transform3 trans3_inverse(Transform3 transf)
```

Returns the inverse of the transformation transf.

```
Vec3    trans3_pos(Transform3 transf, Vec3 v)
```

Returns the transform by transf of the position v, i.e R*v+t.

```
Vec3    trans3_inverse_pos(Transform3 transf, Vec3 v)
```

Returns the inverse transform by transf of the position v.

```
Vec3    trans3_vec(Transform3 transf, Vec3 v)
```

Returns the transform by transf of the direction v, i.e R*v, (translation does not affect directions).

```
Vec3    trans3_inverse_vec(Transform3 transf, Vec3 v)
```

Returns the inverse transform by transf of the direction v.

```
void    trans3_get_frame1in2(Transform3 transf, Vec3 * p,
                             Vec3 * ex, Vec3 * ey, Vec3 * ez)
```

Sets the position *p of the origin and the directions *ei of the axes of coordinate frame1 in frame2.

```
void    trans3_get_frame2in1(Transform3 transf, Vec3 * p,
                             Vec3 * ex, Vec3 * ey, Vec3 * ez)
```

Sets the position *p of the origin and the directions *ei of the axes of coordinate frame2 in frame1.

```
Vec3    trans3_get_origin2in1(Transform3 transf)
```

Returns the position of the origin of frame2 in frame1. (trans3_get_origin1in2 is not supplied, it would just return transf.t).

```
Transform3 trans3_from_frame(Vec3 p, Vec3 ex, Vec3 ey, Vec3 ez)
```

Returns the transformation for which the position of the origin and the axes of frame1 in frame2 are given by p, ex, ey, ez).

```
Transform3 trans3_to_frame(Vec3 p, Vec3 ex, Vec3 ey, Vec3 ez)
```

Returns the transformation for which the position of the origin and the axes of frame2 in frame1 are given by p, ex, ey, ez).

```
Transform3 trans3_from_frame_to_frame(Vec3 p1, Vec3 ex1, Vec3 ey1, Vec3 ez1,
                                      Vec3 p2, Vec3 ex2, Vec3 ey2, Vec3 ez2)
```

Returns the transformation for which the position of the origin and axes of frame1 and frame2 are given by p1, ex1, ey1, ez1 and p2, ex2, ey2, ez2 respectively in some third frame.

```
void trans3_print(FILE *fp, Transform3 trans)
```

Prints out the transformation transf in ascii to a file pointer, first the rows of the rotation matrix, then the translation vector using %f and no newlines.

```
void trans3_format(Transform3 trans)
```

Writes out the transformation transf as above to the Tina textsw or standard output.

Chapter 6

Maths Utilities

Many machine vision algorithms require a number of frequently used numerical calculations. We have defined many such functions, the more complex of which are often based upon the methods described in Numerical Recipies in C. We make no effort to expand upon that excellent text here but recommmend than effort is made to understand these methods and test them before using them in your own calculations.

6.1 Simple Arrays and Vectors

Many computations require simple one dimesional or two dimensional arrays for temporary variables. These can be generated via standard calls to malloc and free, but as this can easiy lead to memory allocation errors Tina provides a set of simple C macro wrappers to handle this. These have been written to centralise allocation so that the generation of variable length resources has less potential for bugs and can be easily debugged. It is therefore recommended that they are used. The macros, which are wrappers around **nvector_alloc()** are all defined in dynamic.h and include ;

```
(_type **) tarray_alloc(m1, n1, m2, n2, _type)
(_type *)  tvector_alloc(n1, n2, _type)

(void ***) parray_alloc(m1, n1, m2, n2)
(void **)  pvector_alloc(n1, n2)

(char **) carray_alloc(m1, n1, m2, n2)
(char *)  cvector_alloc(n1, n2)

(short **) sarray_alloc(m1, n1, m2, n2)
(short *)  svector_alloc(n1, n2)

(int **) iarray_alloc(m1, n1, m2, n2)
(int *)  ivector_alloc(n1, n2)

(float **) farray_alloc(m1, n1, m2, n2)
(float *)  fvector_alloc(n1, n2)

(double **) darray_alloc(m1, n1, m2, n2)
(double *)  dvector_alloc(n1, n2)

(Complex **) zarray_alloc(_m1, _n1, _m2, _n2)
(Complex *)  zvector_alloc(_n1, _n2)
```

There are also corresponding copy, data zero and free functions. The free function macros are written not only to free the structure but also set the corresponding pointer to NULL so that any future attempt to free the same vector or array can be trapped and reported via a call to **error()**, with the message

```
warning : attempt to free NULL pointer in nvector_free()
```

This simple expedient will trap half the causes of potential memory overwrites during software development [1]. Attempts to free vectors defined with an inconsistent starting location are also trapped,, with the message

```
non_fatal : attempt to free re-defined vector in nvector_free()
```

This can be caused either by overwriting or simply specifying the wrong starting location.

Two points of note however, data structures generated with these macros must be freed with the corresponding free macros and **not free()** so that any specified offsets in the data are handelled correctly. Also, these dynamic structures should not be used as a component in a new data structure or hung off property lists if you intend to make direct use of the **serialise** process. The **Matrix** and **Vector** data structures should be used for this purpose.

6.2 Simple Functions

Basic mathematical functions include the following self explanatory routines

```
extern double  asinh(double x);

extern double  tina_acos(double c);

extern double  sqr(double x);

extern double  dist2(double x1, double y1, double x2, double y2);

extern int     imin(int x, int y);

extern int     imax(int x, int y);

extern double  fmin(double x, double y);

extern double  fmax(double x, double y);

extern int     imin3(int x, int y, int z);

extern int     imax3(int x, int y, int z);

extern double  fmin3(double x, double y, double z);

extern double  fmax3(double x, double y, double z);
```

[1] The other half being due to exceeding array bounds

```
extern void    fft_cmplx_inplace(Complex * z, int n);

extern void    fft_inverse_cmplx_inplace(Complex * z, int n);

extern void    fourier(double *data, int nn, int isign);
```

6.3 Random Number Generator

There are also a set of centralised random number routines for Monte-Carlo work.

```
extern double  rand_1(void);

extern int     rand_bit(void);

extern int     rand_int(int a, int b);

extern double  rand_unif(double x, double y);

extern double  rand_normal(double mu, double sigma);

extern void    rand_time_seed(void);
```

6.4 Statistical Distributions

Basic statistical measures based on gamma functions and chi-square distributions.

```
extern double  gammln(double x);

extern double  gammp(double a, double x);

extern double  gammq(double a, double x);

extern double  chisq(double x, int n);
```

6.5 Matrix Inversion and Decomposition

The majority of the following are based on the algorithms described in "Numerical Recipies in C" and are generally used as the basis for solutions to closed form least square systems. These need to be used very carefully in vision algorithms due to statistical difficulties arising from outliers.

```
extern void    ludcmp(double **a, int n, int *indx, double *d);

extern void    lubksb(double **a, int n, int *indx, double *b);

extern void    svd_err_func(void (*text) (   ));
```

```
extern void    SVDcmp(double **a, int m, int n, double *w, double **v);

extern void    SVDbksb(double **u, double *w, double **v, int m, int n,
                       double *b, double *x);

extern void    pentadiag(int n, double *a0, double *b0, double *c0,
                         double *d0, double *e0, double *s0);

extern void    pentaprod(int n, double *a, double *b, double *c,
                         double *d, double *e, double *x, double *y);

extern void    triadiag(int n, double *b0, double *c0, double *d0,
                        double *s0);

extern Matrix *matrix_cholesky_decomp(Matrix * A);

extern Vector *matrix_cholesky_back_sub(Matrix * A, Vector * b);

extern Vector *matrix_cholesky_sol(Matrix * A, Vector * b);

extern Vector *matrix_cholesky_weighted_least_square(Matrix * A,
                                                     Matrix * W, Vector * b);

extern Vector *matrix_cholesky_least_square(Matrix * A, Vector * b);

extern Vector *matrix_solve(Matrix * mat, Vector * y);


extern int     matrix_eigen_sym(Matrix * A, Vector * Eval, Matrix * Evec);

extern int     matrix_eigen(Matrix * A, Vector * Eval, Vector * Evec);
```

6.6 Solvers and Optimisation

```
extern Bool    quadratic_roots(double a, double b, double c, double *x1,
                               double *x2);

extern Bool    cubic_roots(double a, double b, double c, double d,
                           double *x1, double *x2, double *x3);

extern double  newton_raphson(double (*f) (), double (*df) (),
                              double x, double eps);

extern double  golden_section(double (*f) (), double a, double b,
                              int (*tol) (), void *fdata, void *tdata);

extern int    *dprog(int lx, int ly, int ux, int uy, double (*cost1) (),
                     double (*cost2) (   ));

extern double  simplexmin(int n, double *x, double lambda, double (*funk) (),
                          double ftol, void (*text) (   ));
```

```
extern double  simplexmin2(int n, double *x, double *lambda, double (*funk) (),
                           double ftol, void (*text) (   ));

extern void    amoeba(double **p, double *y, int ndim, double ftol,
                      double (*funk) (), int *nfunk);

extern double  amotry(double **p, double *y, double *psum, int ndim,
                      double (*funk) (), int ihi, int *nfunk, double fac);
```

6.7 Covariance Functions

Machine vision algorithms should not only solve numerical problems but also be capable of estimating accuracy in order to validate the results. Functions for covariance estimation and manipulation to be used in conjunction with optimisation routines include the following.

```
extern void    jacob(double *a, double *da, Matrix * jf, double (*func) ());

extern Covar  *invcov(int m, double *a, double (*func) (), int n);

extern Covar  *covar(int m, double *a, double (*func) (), int n,
                     double condition);

extern Covar  *covar_update(Covar * inv_acov, Covar * inv_bcov,
                            double condition);

extern Covar  *covar_add(Covar * inv_acov, Covar * inv_bcov);

extern Covar  *covar_alloc(int n);

extern void    covar_free(Covar * cov);
```

Chapter 7

Image Handling

7.1 Introduction

Tina supports a number of different image types. From 8 bit unsigned characters to images of type **Complex** (a C structure with a pair of floats representing real and imaginary parts). Edges and other types of feature can be indexed through a generic pointer image.

Tina image structure definitions and typedefs (amongst other things) can be included with

```
#include <tina/sys/sys.h>
#include <tina/image/imageDef.h>
```

function declarations with

```
#include <tina/image/imagePro.h>
```

programs should be compiled with the libraries

```
-ltinaImage -ltinaMath -ltinaSys
```

7.2 Imrects

The basic image structure is the **Imrect** (Image Rectangle) which specifies the type and size of the image, the region of interest it covers, provides an index into the image data itself and has a property list for the flexible storage of extra more specific information about particular images.

Note that numerous imrects may exist for the same image; each with a different, possibly overlapping, region of interest. And that the whole of the image may not be available.

Also note that while the elements that form each row of the Imrect are guarantied to occupy contiguous memory locations adjacent rows will not necessarily do so.

```
typedef struct imrect
{
    Ts_id ts_id;                /* Tina structure identifier */
    Vartype vtype;
    int     width, height;      /* of image */
```

file id	Vartype	Meaning
0	char_v	8 bit character image
1	uchar_v	8 bit unsigned 8 bit characters
2	short_v	16 bit integer
3	ushort_v	16 unsigned integer
4	int_v	32 bit integer
5	uint_v	32 bit unsigned integer
6	float_v	32 bit floating point
7	double_v	64 bit floating point
8	complex_v	struct complex
9	ptr_v	void *

Table 7.1: AIFF image types

```
    Imregion *region;           /* region covered */
    void   *data;               /* array of pointers to rows */
    struct list *props;         /* covers all the extras for an edge
                                 * rect */
} Imrect;
```

7.2.1 Image Regions

The image **width** and **height** represent the physical extent of the image. Hence it is assumed that the image begins at location **(0, 0)** and ends at **(height, width).** Images are indexed in the order row first and column second. The physical image usually begins at the top left hand corner and ends at the bottom right. It is this arrangement for which Tina will generate 3D geometrical primitives measured with respect to a right handed coordinate system.

Within the image the **region** field specifies the region of interest covered by the Imrect. The **region** is a pointer to the structure of type imregion

```
typedef struct imregion
{
    Ts_id ts_id;                /* Tina structure identifier */
    int     lx, ly;             /* top left */
    int     ux, uy;             /* bottom right */
} Imregion;
```

The region specified by the window structure covers the area **(ly, lx)** to **(uy, ux)** in image coordinates. Note that the **region** is allowed to lie outside (either wholly or partially) the underlying image area as defined by the **width** and **height** fields of the imrect data structure. It is useful in some circumstances to allow images to be transformed (eg. to allow for aspect ratio of the cameras or image warping) outside their original confines. In such cases **height** and **width** are used to indicate the size of the original image from which the deformed version was recovered.

7.2.2 Image Types

The image variable type, specified by the **vtype** field of the imrect structure, can be any (if supported on the machine) one of the following (also indicated is the AIFF [Aivru Image File Format] id used used for file storage):

More specific information relating to the type of data that an image represents can be carried in the **imtype** field.

The union **varptrptr** has the following form

```
typedef union varptrptr
{
    char **char_v;
    unsigned char **uchar_v;
    short **short_v;
    unsigned **ushort_v;
    int **int_v;
    unsigned int **uint_v;
    float **float_v;
    double **double_v;
    struct complex **complex_v;
    void ***ptr_v;
    struct vram **vram_v;
} Varptrptr;
```

So that if an **image,** called "im" say, is of **vtype ushort_v** then its image data must be indexed

```
    im->data
```

which is itself a double indirection to an **unsigned short integer** (ie. **unsigned short ****).

Note that the structure complex is defined as follows

```
typedef struct tcomplex
  {
      Ts_id   ts_id;            /* Tina structure identifier */
      double   x;               /* Don't introduce numerical */
      double   y;               /* instability NAT 27/4/95   */
  }
Complex;
```

7.2.3 Image Indexing

In order to index element **j** on row **i** we must write

```
    im->array.ushort_v[i][j]
```

Provided that the image location is within the region of interest covered by the imrect.

An existing contiguous block of data may be wrapped within an Imrect data structure via a call to

```
Imrect *im_wrap_contig(void *mem_ptr, int height, int width,
                       Imregion * region, Vartype vtype)
```

The following shorthand macros provide a more simple syntax for image indexing

```
IM_CHAR(im, i, j)
IM_ UCHAR(im, i, j)
IM_SHORT(im, i, j)
IM_USHORT(im, i, j)
IM_INT(im, i, j)
IM_ UINT(im, i, j)
IM_FLOAT(im, i, j)
IM_DOUBLE(im, i, j)
IM_COMPLEX(im, i, j)
IM_PTR(im, i, j)
IM_VRAM0(_im,_i,_j)
IM_VRAM1(_im,_i,_j)
IM_VRAM2(_im,_i,_j)
IM_VRAM3(_im,_i,_j)
```

Even so it is still necessary to check the type and region covered by the imrect prior to application of the macro. Generic functions for accessing and manipulating imrects of all kinds that explicitly check the region can be found in the list of functions at the end of this chapter.

Note that indexing the scheme is with respect to the coordinates of the underlying image (as defined by the **height** and **width** fields of the imrect) rather than relative to the lower corner (numerically rather than physically) of the region of interest of the current imrect. Furthermore, it is possible that image indices can have negative values.

7.2.4 Pointer Images

Pointer images are used as the primary representation for various types of feature recovered directly from the image (such as edges, regions and corner points). For generic manipulation (functions designed to work on various data types, eg. display functions) the type of feature can be indicated using the **imtype** field. The absence of a features at a particular image location is indicated by a nil pointer.

Iconic representations are often not the most convenient or efficient form of image access, hence it is often useful to use the property list of the imrect to store one or more alternative indexing schemes more suited to the job in hand.

7.3 Image Functions

```
Imrect *im_alloc(int height, int width, Imregion * region, Vartype vtype)
```

Allocate imrect of **vtype** from image of size **height** times **width** to cover **region.** If region is NULL then whole image is created.

```
void    im_free(Imrect * image)
```

Frees the **image** and its associated property list.

```
Imrect *im_copy(Imrect * image)

Imrect *im_cast(Imrect * image, Vartype vtype)

Imrect *im_subim(Imrect * image, Imregion * region)
```

```
void    im_copy_inplace(Imrect * image2, Imrect * image1)
```

Functions to make new copies of images. These use the **im_write** functions above. In **im_copy** the argument **vtype** specifies the type of the new image (subject to the restrictions of

im_write_general_type). In **im_subim** the region covered by the returned imrect is given by the intersection of the region of **image** with that in the argument **region.** If **region** or its intersection with **image->region** is NULL, then the function returns NULL.

```
void    im_put_pix(int pixval, Imrect * image, int i, int j)

void    im_put_pixf(double pixval, Imrect * image, int i, int j)

void    im_put_pixz(Complex pixval, Imrect * image, int i, int j)

void    im_put_ptr(void *ptr, Imrect * image, int i, int j)

void    im_pixf_inc(Imrect * image, int i, int j)
```

General functions for setting imrect pixel values; they deal with all images except those of type **ptr_v.** The image region is examined to ensure that the image pixel at **(i, j)** lies within it. Functions **im_put_pix** and **im_put_pixf** effect only the real part of pixels of type **complex_v.** Similarly, in **im_put_pixz** only the real part of **pixval** is used to update (suitably cast) image pixels of types other than **complex_v.**

```
int     im_get_pix(Imrect * image, int i, int j)

void   *im_get_ptr(Imrect * image, int i, int j)

double  im_get_pixf(Imrect * image, int i, int j)

Complex im_get_pixz(Imrect * image, int i, int j)
```

General functions for retrieving imrect pixel values; they deal with all images except those of type **ptr_v.** The **image** region is examined to ensure that the image pixel at **(i, j)** lies within it. Functions **im_get_pix** and **im_get_pixf** obtain only the real part of pixels of type **complex_v.** Similarly only the real part of the Complex variable returned by **im_get_pixz** is set (the imaginary part is 0) for pixels of types other than **complex_v.**

```
int     im_sub_pix(Imrect * image, double r, double c)

double  im_sub_pixf(Imrect * image, double r, double c)

double  im_sub_pixqf(Imrect *image, double y, double x)

double  im_get_quadinterpf(Imrect * image, float x, float y,
                           float *pdx, float *pdy)

double  im_get_quadmaxf(Imrect * image, float x, float y, float *px, float *py)
```

General functions to obtain approximate image values to at sub pixel image locations (not defined for **ptr_v** or **complex_v** image types). Sub pixel values are obtained by bilinear interpolation for **im_sub_pixf** and quadratic interpolation for **im_sub_pixqf im_get_quadinterpf im_get_quadmaxf** .

```
void    im_put_row(int *line, Imrect * image, int i, int from, int to)

void    im_put_rowf(float *line, Imrect * image, int i, int from, int to)

void    im_put_rowz(Complex * line, Imrect * image, int i, int from, int to)
```

Funtions to write from **line** into part of row **r** of **image.** Arguments **from** and **to** specify the section of row **r** to be read from **line** (ie. **line[c] : from >= c < to**). Relevant sections of **line** that lie outside the region of **image** are not written. Functions are undefined for images of type **ptr_v.**

```
void    im_put_col(int *line, Imrect * image, int i, int from, int to)

void    im_put_colf(float *line, Imrect * image, int i, int from, int to)

void    im_put_colz(Complex * line, Imrect * image, int i, int from, int to)

Vector *im_col_vector(Imrect * im, int x, int ly, int uy, Vartype vtype)
```

Funtions to write from **line** into part of column **c** of **image.** Arguments **from** and **to** specify the section of column **c** to be read from **line** (ie. **line[c] : from >= c < to**). Relevant sections of **line** that lie outside the region of **image** are not written. Functions are undefined for images of type **ptr_v.**

```
void    im_get_row(int *line, Imrect * image, int i, int from, int to)

void    im_get_rowf(float *line, Imrect * image, int i, int from, int to)

void    im_get_rowz(Complex * line, Imrect * image, int i, int from, int to)

Vector *im_row_vector(Imrect * im, int y, int lx, int ux, Vartype vtype)
```

Funtions to read part of row **r** of **image** into data array **line.** Arguments **from** and **to** specify the section of row **r** to be copied into **line** (ie. **line[c] : from >= c < to**). Relevant sections of **line** that lie outside the region of **image** are set to zero. Functions are undefined for images of type **ptr_v.**

```
void    im_get_col(int *line, Imrect * image, int i, int from, int to)

void    im_get_colf(float *line, Imrect * image, int i, int from, int to)

void    im_get_colz(Complex * line, Imrect * image, int i, int from, int to)
```

Funtions to read part of column **c** of **image** into data array **line.** Arguments **from** and **to** specify the section of column **c** to be copied into **line** (ie. **line[r] : from >= r < to**). Relevant sections of **line** that lie outside the region of **image** are set to zero. Functions are undefined for images of type **ptr_v.**

7.3.1 Region of Interest Functions

```
Imregion *roi_alloc(int lx, int ly, int ux, int uy)

Imregion *roi_copy(Imregion * roi)

void      roi_update(Imregion * roi, Imregion * copy)

int       roi_inregion(Imregion * region, int x, int y)

Imregion *roi_inter(Imregion * r1, Imregion * r2)

void      roi_fill(Imregion * roi, int lx, int ly, int ux, int uy)
```

With obvious meanings apart from **roi_inter** which performs region intersection.

Chapter 8

Image Processing

8.1 Introduction

The Tina system supports a wide variety of image data types which are processed in an object oriented fashion. The software also supports images of base representation structures such as matrices and vectors. The definition of an image is thus extended considerably in the Tina system so that it can include any spatially organised grouping of data structures in a manner entirely consistent with view based computer vision paradigms. The data representation is sufficiently flexible for the user programmer to develop new algorithms to process novel data structures. This is supported by generic image data manipulation routies.

8.2 Convolution and Filtering

The following section describes specific and general image convolution software.

```
Imrect *im_conv_h(Imrect * im1, Prof1 * prof)
```

Convolves the image im1 with a horizontal (x based) convolution profile prof. Contributions to the convolution external to the original image are set to zero.

```
Imrect *im_conv_v(Imrect * im1, Prof1 * prof)
```

Convolves the image im1 with a vertical (y based) convolution profile prof. Contributions to the convolution external to the original image are set to zero.

```
Imrect *im_conv_separable(Imrect * im1, Prof1 * prof_h, Prof1 * prof_v)
```

Concolves the image with the specified horizontal and vertical decomposition of the required kernel using the routines described above.

```
Imrect *im_filter_cols(Imrect * image, void (*func) ( /* ??? */ ), void *data)
```

Applies the function func(float *line, ly, uy, data) to the specified image. The data field allows extra parameters to be passed to this generic image processing function.

```
Imrect *im_filter_rows(Imrect * image, void (*func) ( /* ??? */ ), void *data)
```

Applies the function func(float *line, lx, ux, data) to the specified image.

8.3 Pixel Processing

General pixel based manipulation including transendental functions is carried out by the following routines. For multiple image manipulation the image type returned is determined by the function im_sup_vtype

```
Vartype im_sup_vtype(Vartype vtype1, Vartype vtype2)
```

which ensures no truncation of the result due to casting (truncation may still occur for numbers beyond the normal range of number representation).

```
Imrect *im_add(double k,Imrect *im)
```

Add the constant k to all pixels in the specified image. For the case of complex images only the real component is modified.

```
Imrect *im_diff(Imrect * im1, Imrect * im2)
```

Subtract the pixels contained in two images and return the difference image.

```
Imrect *im_minus(Imrect * im)
```

Inverts the sign of all pixel values returning a new image.

```
void    im_pixf_dec(Imrect * image, int i, int j)
```

Decrement pixel (i, j) in image. Pixels outside image are ignored.

```
Imrect *im_sum(Imrect * im1, Imrect * im2)
```

Returns an image whose pixels are the sum of those in the input images within the common region of interest.

```
Imrect *im_times(double k, Imrect * im)
```

Returns an image with pixels scaled by k of their input values.

```
Imrect *im_prod(Imrect * im1, Imrect * im2)
```

Returns an image with values given by the product of the pixels in the images im1 and im2.

```
Imrect *im_div(Imrect * im1, Imrect * im2, double thresh, double val)
```

Divide the pixels contained in image im1 by those contained in image im2. Numerical stability is maintained by preventing the denominator reducing below a value of *val* when it's absolute value is less than *thresh*.

```
Imrect *im_sqr(Imrect * im)
```

Compute the sqaure of each image pixel value, all values returned as float_v except complex_v, which is returned as complex_v.

```
void    im_grad(Imrect * im, Imrect ** imx, Imrect ** imy)
```

Compute derivative images for both the x and y components using the functions imf_diffx(im) and imf_diffx(im).

```
void    im_hessian(Imrect * im, Imrect ** imx, Imrect ** imy, Imrect ** imxx, Im
rect ** imxy, Imrect ** imyy)
```

Computes both x and y derivatives, and second derivatives. If the first derivatives are not required then pass null pointers for imx and imy.

```
Imrect *im_log(Imrect * im)
```

Returns an image of natural logariths for all pixels while maintaining sign. The function is a direct inverse of im_exp(). All images are returned as type float_v except for type complex_v which is returned as complex_v.

```
Imrect *im_quad(im)
```

Returns an image which has been vertically and horizontally doubled to produce boudary continuity. Intended for use before Fourier transform operations to remove Gibbs oscillations after deconvolution.

```
Imrect *im_sin(Imrect * im)
```

Returns an image with each pixel given by the trigonometric sine of the initial grey level data.

```
Imrect *im_sqrt(Imrect * im)
```

Returns an image with each pixel the square-root of the initial grey level value. The sign of the original pixel is preserved, for true complex treatment of negative values use imz_sqrt().

```
Imrect *im_asin(Imrect * im)
```

Computes the trigonometric sine of all pixels in the image. All returned images are of type float_v except complex images which are returned as type complex_v.

```
Imrect *im_exp(Imrect * im)
```

Compute the exponentiated value of each pixel with maintained sign. Returns images of type float_v for all cases except complex_v which are returned as complex_v. The function is the direct inverse of im_log().

```
void im_scale_range_inplace(Imrect * im,
                            double oldlow, double oldhigh,
                            double newlow, double newhigh,
                            double threslow, double threshigh)
```

This routine scales the range oldlow, oldhigh in image im newlow, newhigh, thresholding at the upper and lower values given by threslow, threshigh.

```
void im_gamma_scale_range_inplace(Imrect * im,
                                  double gamma,
                                  double oldlow, double oldhigh,
                                  double newlow, double newhigh,
                                  double threslow, double threshigh)
```

Gamma scale the range oldlow, oldhigh in image im using the specified parameters.

```
Imrect *im_shading(Imrect *im,double slant,double tilt,double scale)
```

Produces a shaded image from a 3D image surface using a point light source and lambertian reflectance model at a distant location specified by *slant* and *tilt* .

```
Imrect *im_combine(Imrect * im1, Imrect * im2, void *(*func) (), void *data)
```

Returns an image with pixels computed from the specified input image pixels operated upon by the specified function.

```
Imrect *im_fpp_combine(Imrect * im1, Imrect * im2, void *(*func) (), void *data)
```

General purpose routine for operating on two images of types float_v and ptr_v with the specified function.

8.4 Complex Images

The following routines are specifically for the generation and maipulation of complex images.

```
Imrect *im_arg(Imrect * im)
Imrect *im_phase(Imrect * im)
```

Duplicates functions which generate an output image of the phase angle of the input complex image.

```
Imrect *im_cis(Imrect * im1)
```

Generates a unit complex output image from an image of phase angles.

```
Imrect *im_conj(Imrect * im1)
```

Generates the image of complex conjugates from the input complex pixels.

```
Imrect *im_mod(Imrect * im)
```

Generates an image of unsigned modulus from the input image pixels.

```
Imrect *im_re(Imrect * im)
```

Returns an image of real components of the input complex image.

```
Imrect *im_im(Imrect * im)
```

Returns an image of complex components of the input complex image.

```
Imrect *im_fft(Imrect *im1,Imregion *region)
```

Returns an image of 2D Fourier components of the input image.

```
Imrect *im_fft_inverse(Imrect *im1,Imregion *region)
```

Returns an image of inverse 2D Forier transform of the input complex image.

```
Imrect *im_power_spectrum(Imrect *im1)
```

Returns an image containing the squared modulus of the complex input image pixels.

```
Imrect *im_gabor(Imrect * im1, double phasex, double sigmax, double omegax,
                 double nsigmax, double phasey, double sigmay, double omegay,
                 double nsigmay)
```

Returns an image generated by applying a 2D gabor filter as defined by the specified variables to the input image.

```
Imrect *im_gabor_fft(Imrect * im, double k, double b, double theta)
```

Fft version of 2D gabor filter: k = centre frequency b = 1-sigma bandwidth in octaves theta = orientation.

```
Imrect *im_fgabor(Imregion *roi, double k, double b, double theta)
```

Returns an image generated by applying gabor filtering defined by the specified variables to an image represented in the Fourier domain.

8.5 Noise Filtering

The following routines can be used for the removal of various forms of noise and unwanted data variation from images.

```
Imrect *im_bthresh(double k,Imrect *im)
```

Returns an image which is a binary thresholded version of the input image around the value defined by k.

```
Imrect *im_thresh(double k,Imrect *im)
```

Returns an image which has been thresholded to set all pixel values below k to zero. This is useful, for example, in the elimination of unwanted terms in the 2D Fourier domain of an image.

```
Imrect *im_corrupt(Imrect * im, int dx, int dy, double a, double b)
```

Generate an image by adding uniform random noise (generated by imf_unif_noise(im-¿width, im-¿height, dx, dy, a, b)) to the input image. Useful in the generation of simulated test data for the evaluation of algorithm stability.

```
Imrect *im_rank(Imrect *im, int range, double noise)
```

Returns an image with each pixel (of measurement accuracy *noise*) given by the rank of its value in the surrounding $2 * range + 1$ x $2 * range + 1$ patch. Useful for enhancing the spatial information content of an image before the process of cross correlation or template matching.

```
Imrect *im_median(Imrect *im)
```

Returns a median filtered (discontinuity preserving) version of the input image (each pixel replaced by the median value of the 3x3 neighbourhood). Useful for the removal of sensor or aliasing derived pixel dropout. This technique should only be used when there is no alternative, generally it is better to eliminate the problem at source).

```
Imrect *im_tsmooth(Imrect * im1)
```

Returns an image which has been smoothed by averaging in a direction tangential to the image gradient at each point (discontinuity preserving).

8.6 Image Warping

The following are routines for modifying the coordinate system and generally warping an image.

```
Imrect *im_subim(Imrect * image, Imregion * region)
```

Returns an image which is a new copy of specified roi of image.

```
void    im_shift(Imrect * im, int dy, int dx)
```

Returns an image which is a shifted copy the specified image shifted by the amounts specified in the given parameters.

```
Imrect *im_bshift(Imrect *im, int y , int x)
```

Returns an image which is a barrel shifted copy the specified image shifted by the amounts specified in the given parameters. The returned image maintains the original co-ordinates of the input image.

```
Imrect *im_rectify(Imrect * im1, Mat3 rect)
```

Returns a new image created by the specified perspective transform of the input image.

```
Imrect *im_warp(Imrect * im1, Vec2(*f) ( /* ??? */ ), void *data)
```

This is the generic image warping routine which applies the inverse warp function specified in order to create the warped output image.

8.7 Feature Location

General purpose feature location and interest operators.

```
Imrect *canny(Imrect * im, double sigma, double precision, double lowthres,
              double upthres, int lengththres)
```

Returns an image of edgel data structures calculated by the canny edge detection algorithm operatin at a scale determined by sigma and a sensitivity determined by lowthres which hysteresis thresholding set by upthres. The resulting edges are segmented into strings with a minimum lenght set by lengththres.

```
Imrect *corner(Imrect *im,double sigma,double precision,double lowthres)
```

Our own implementation of the Harris and Stephens corner detection algorithm, a rationalisation of Moravec's interest operator. It returns an image of subpixel located corners stored as edgel structures.

```
Imrect *imf_nmax(Imrect *im,float thres,int num)
```

General purpose maxima location algorithm for extracting features from feature enhanced images. Features are located wherever a pixel is greater than thres and less than no more than num of it's nearest 9 neighbours. Set num to 2 for ridge detection and 0 for peak location.

```
void im_locate_max(Imrect * im, int *y, int *x)
```

Locate the point of maximum value in the given image.

8.8 Generic Manipulation of mixed image types

This routine is used to apply a function to images of type ptr_v and float_v.

```
Imrect *im_pf_apply(Imrect * im1, double (*func) (), void *data)

Imrect *im_pp_apply(Imrect * im1, void *(*func) (), void *data)

Imrect *im_ppf_combine(Imrect * im1, Imrect * im2, double (*func) (),
                       void *data)

Imrect *im_ppp_combine(Imrect * im1, Imrect * im2, void *(*func) (), void *data)

void    im_ptr_apply(Imrect * im, void (*func) (), void *data)
```

8.9 Manipulating Images of Tina Data Structures

Functions for generating and manipulating images of Vec2 data structures.

```
Imrect *im_vec2(Imrect * im1, Imrect * im2)

void    im_vec2_free(Imrect * im)

Imrect *im_vec2_dot(Imrect * u, Imrect * v)

Imrect *im_vec2_cross(Imrect * u, Imrect * v)

Imrect *im_vec2_diff(Imrect * u, Imrect * v)

Imrect *im_vec2_grad(Imrect * im)

Imrect *im_vec2_sum(Imrect * u, Imrect * v)

Imrect *im_vec2_x(Imrect * v)

Imrect *im_vec2_y(Imrect * v)
```

Functions for generating and maipulating images of pointers to Mat2 data structures.

```
Imrect *im_mat2_det(Imrect * m)

void    im_mat2_free(Imrect * im)

void    im_mat2_grad_hessian(Imrect * im, Imrect ** g, Imrect ** h)

Imrect *im_mat2_hessian(Imrect * im)

Imrect *im_mat2_inverse(Imrect * m)

Imrect *im_mat2_of_cols(Imrect * cx, Imrect * cy)
/* make image of mat2's from images of column vectors */

Imrect *im_mat2_of_rows(Imrect * rx, Imrect * ry)
/* make image of mat2's from images of row vectors */

Imrect *im_mat2_sprod(Imrect * u, Imrect * m, Imrect * v)
/* scalar product of vec2 image, mat2 image and vec2 image */

Imrect *im_mat2_vprod(Imrect * m, Imrect * v)

Imrect *im_mat2_xx(Imrect * m)

Imrect *im_mat2_xy(Imrect * m)

Imrect *im_mat2_yx(Imrect * m)

Imrect *im_mat2_yy(Imrect * m)
```

Chapter 9

Feature Representation

9.1 Introduction

This section describes how the 2D and 3D features are represented in the Tina system. It includes both image features, such as edges and corners, as well as edge strings, and geometrical primitives (both 2 and 3 dimensions) recovered from them. It is possible (highly likely) that further primitives will be added in future releases of the system. Such primitives will be treated in a broadly similar way to those described here.

In the light of experience and to allow the system to be flexible to the requirements of the user, the basic representation of Tina primitives has been kept as simple and uniform as possible. This system has now been used succesfully in a continuing program of computer vision research for several years. Extendibility and generality is achieved by using a property list for the storage of additional information specific to each different application. This property lists field is always defined in other Tina structures as

```
struct list *props;
```

Property lists are used throughout the Tina system and are described in the chapter on list processing.

In addition each data structure contains a structure identifier

```
Ts_id ts_id;
```

which is used to allow data transfer between machines. This is described further in the chapter dealing with the "serialisation process".

Extra information that describes the relationships between primitives (such as edge strings or order along the image rasters) is kept outside the primitives themselves. This approach further enhances the modularity of the representations.

The most important forms of generic data structure used throughout the Tina sytem are lists, doubly directed lists, and strings (these are described in a separate chapter of this programming guide). Additional data structures used to represent more specific data abstractions will be introduced in this chapter.

9.2 Image Features

A number of purely 2D and localised image features are available. These include edges, corners, region boundaries. Latter, a more symbolic 2D primitive, called a Point2, is introduced. This is used for the representation of, amongst other things, vertex locations. All 2D feature location primitives and the

field	mask	value
1	0x0000000F	type
2	0x000000F0	conn
3	0x00000F00	rect
4	0x0000F000	loop
5	0x000F0000	match
6	0x00F00000	corr

Vec2 data type can be treated in a uniform manor using POS2 (Position 2D) macros (see below). This facilitates the generation of primitive independent code (eg. polygonal approximation).

Edges are the most important image feature in the Tina system. Each individual edge location (one per pixel through which the edge passes) is represented the edgel data structure

```
typedef struct edgel
{
    Ts_id ts_id;                /* Tina structure identifier */
    unsigned int type; /* includes information as to rectification status */
    unsigned int label;
    struct vec2 pos;
    float orient; /* these could go to props list perhaps NAT */
    float contrast;
    struct list *props;
} Edgel;
```

Although strictly defined for use to describe edge elements this data structure is sufficiently flexible to allow its use for any well located image feature. Currently the edgel is also used to represent corner points (although this may change in a future release of the system).

The Vec2 **pos** is the sub pixel location of the edge with respect to the underlying image (or a transformed version of it). Where the center of the pixel indexed **[i, j]** is at sub pixel image location **[i+0.5, j+0.5].** The fields **orient** and **con** represent the orientation of the edge and its absolute image contrast. The contrast of an edge is proportional to the absolute gradient across it. The orientation of an edge, given in radians is given by the local direction along the edge (- to degrees with + clockwise). Orientations 0, and - are all horizontal, with 0 being light above dark and and - both being dark above light.

The **type** of an edge is more complex than that of other Tina primitives as it contains several pieces of information. The type is split into 8 4-bit fields the lower 5 of which have already been defined.

type specifies the process that generated the edge, eg.

- EDGE_UNKNOWN
- EDGE_CANNY & produced by the canny edge detector)
- EDGE_SOBEL & produced by the sobel edge detector)

conn describes the connectivity of the edge, eg.

- EDGE_NOLINK & not linked)
- EDGE_TERMIN & link termination point)
- EDGE_CONN & connected to exactly 2 other edges)
- EDGE_JUNC & connected to more than 2 edges)

- EDGE_ISOLATED & an isolated edge feature or corner)

An isolated edge is more important than an edge that is not linked. The edge linker leaves isolated edges unlinked where as the corner matcher flags its edge features as isolated.

rect and *corr* specify the rectification and distortion correction, or other transformation, state of an edge, eg:

- EDGE_NOT_RECTIFIED
- EDGE_RECTIFIED
- EDGE_NOT_CORRECTED
- EDGE_CORRECTED

loop identifies those edges that are part of a looped edge string, eg

- EDGE_NOT_LOOP
- EDGE_LOOP

Raw edges that have none of the above fields set to meaningful values have **type** EDGE_RAW which includes:

EDGE_UNKNOWN, EDGE_NOLINK and EDGE_NOT_RECTIFIED).

The following masks have been defined for setting and getting fields from the edge **type**, eg:

- EDGERAW
- EDGE_NOT_LOOP
- EDGE_LOOP

Raw edges that have non of the above field set to meaningful values have **type** EDGE_RAW

name	mask
EDGE_SET_CORR_MASK	0xFF0FFFFF
EDGE_SET_MATCH_MASK	0xFFF0FFFF
EDGE_SET_LOOP_MASK	0xFFFF0FFF
EDGE_SET_RECT_MASK	0xFFFFF0FF
EDGE_SET_CONN_MASK	0xFFFFFF0F
EDGE_SET_TYPE_MASK	0xFFFFFFF0
EDGE_GET_CORR_MASK	0x00F00000
EDGE_GET_MATCH_MASK	0x000F0000
EDGE_GET_LOOP_MASK	0x0000F000
EDGE_GET_RCT_MASK	0x00000F00
EDGE_GET_CONN_MASK	0x000000F0
EDGE_GET_TYPE_MASK	0x0000000F

For example to set the rectification field to EDGE_RECTIFIED

```
edge->type &= EDGE_SET_RECT_MASK;
edge->type |= EDGE_RECTIFIED;
```

And to get the state of the rectification field

```
edge->type & EDGE_GET_RECT_MASK
```

After linking edges can be accessed through a list of Tina Strings & see
Generic List section of the manual). This is stored on the property list of the edgerect using the defined type STRING. Elements of tina strings or other topological data structures that refer to edge elements use the defined type EDGE.

9.2.1 Image Feature Functions

The following is a list of frequently used image feature functions.

```
Edgel  *edge_alloc(int type)

Edgel  *edge_copy(Edgel * edge)

void    edge_free(Edgel * edge)
```

For allocating copying and freeing individual edges respectively. Freeing an edge will also free its property list. Copying an edge makes a copy of its property list see **proplist_copy).**

```
void    edge_add_prop(Edgel * e, void *ptr, int type,
                      void (*free_func) ( /* ??? */ ),
                      Bool dofree)

void   *edge_prop_get(Edgel * e, int ptype)

void    edge_save_pos_prop(Edgel * edge, int type, int prop_type)

void    edge_get_pos_prop(Edgel * edge, int type, int prop_type)

Vec2    edge_image_pos(Edgel * edge)
```

Edge property lists see list section of the programmers guide) can be manipulated explicitly using the functions **edge_add_prop** and **edge_prop_get** by calling
proplist_addifnp and **prop_get** respectively).

In particular the property list is used to store the position of the edge with respect to different coordinate frames. This is useful when an edge is rectified for example. The function **edge**save_pos_prop saves the current edge location on the property list with the type **prop_type.** The argument **type** is required to allow standard Tina indirect function call application. It also allows the function to be applied indirectly to nonedge structures.
The defined types IMPOS and RECTPOS are used to identify image positions and rectified positions respectively.

The function **edge_image_pos** examines *rect* field the edge **type** to check if the edge location has been altered. If so it checks the property list for IMPOS. If successful that image location is returned. All other situations return the **pos** field of the edge structure.

```
int     er_add_edge(Imrect * edgerect, Edgel * edge)

void    er_put_edge(Edgel * edge, Imrect * edgerect, int i, int j)

Edgel  *er_get_edge(Imrect * edgerect, int i, int j)
```

```
void    er_rm_edges(Imrect * edgerect, unsigned int mask, unsigned int type)

void    er_free(Imrect * edgerect)
```

Edge/Edgerect manipulation functions. **er_add_edge** adds the **edge** according to its internal position truncated) provided that **edgerect** location exists and is still empty & nil). Unsuccessful calls return *false.*

er_put_edge forces the **edge** into the location **i, j** provided it lies within the region covered by the **edgerect.** And **er_get_edge** recovers it.

er_rm_edges removes from the **edgerect** those edges for which

```
type == edge->type(mask
```

is true. While **er_free** removes all edges from the edgerect and frees it up.

```
void    er_set_row_index(Imrect * er)

void    er_find_edge_strings(Imrect * er)

void    er_find_simple_edge_strings(Imrect * er)

void    er_edge_strings_thres(Imrect * er, int length_thres,
                              double contrast_thres)

List   *es_list_thres(List * strings, int length_thres, double contrast_thres)
```

The edgerect supports alternative indices into the underlying sparse edge data. These are accessed through the property list of the edgerect.

The first is a row index. It is stored using the property type ROWS (or ER_ROWS in old fashioned implementations). The row index is defined

```
List **index;
```

and is defined over the vertical extent of the region of interest of the edgerect. The ptr **index**[**r**] is to an ordered list of all the edges on row **r** ofthe edgerect. To get hold of the row index it is necessary to call

```
index = prop_get(er->props, ROWS);
```

Connected edge strings are identified and stored on the property list, using the property type STRING, by calling **er_find_edge_strings.**

Edge strings can be thresholded by both length and max contrast) by calls to either;
er_edge_strings_thres
or:
es_list_thres
which work on the edgerect through its property list) or directly on a list of edge strings, respectively.

```
void    er_apply_to_all_edges(Imrect * edgerect, void (*edgefunc) ( /* ??? */ ),
                              void *data)

void    er_apply_to_all_strings(Imrect * er, void (*func) ( /* ??? */ ),
                                void *data)

void    es_apply_to_all_edges(Imrect *er, void (*func) ( /* ??? */ ), void *data)
```

Functions to apply the **func** and **data** combination to all edges in an edgerect, all strings referenced by an edgerect and all the edges on the strings referenced by an edgerect respectively. They have standard indirect functioncall format except that **er_apply_to_all_edges** has additional arguments **i, j** to indicate the image location concerned.

9.3 Geometrical Primitives

The following sub-sections describe the principal geometrical primitives used n Tina. It is this set of primitives that are managed by the Geomstat
package. The final section describes generic geometry functions that use a **type** field to determine the which of the geometrical data structures has been passed as data. These are very useful where a number if different primitive types are groupd within a more complex data structure, most typically a recursive list.

9.3.1 Scalar

For completeness the scalar geometric type is included. It is defined by the following data structure.

```
typedef struct scalar
{
    Ts_id ts_id;                /* Tina structure identifier */
     unsigned int type;
     unsigned int label;
     float val;
     struct list *props;
} Scalar;
```

While this primitive does not represent physical geometry it can be used for scalar properties of geometry Eg length, radius, etc). It is used in the Geomstat package for this purpose.

```
Scalar *scalar_alloc(unsigned int type)

Scalar *scalar_make(double val, int type)

void    scalar_free(Scalar * scalar)

Scalar *scalar_copy(Scalar * scalar)

int     scalar_number(List * scalars)

void    scalar_format(Scalar * scalar)
```

Functions for manipulating scalars. The functions **scalar_copy** and **scalar_free** copy and free the property list see proplist description for details). **scalar_format** produces a standard formatted output of **scalar** using standard Tina *format* facility. Usually directed to the top level text display window.

9.3.2 Point Geometry

Point geometry in 2D and 3D is represented by the
point2 and **point3** data structures respectively. These are of the form

```
typedef struct point2
{
    Ts_id ts_id;                /* Tina structure identifier */
    unsigned int type;
    unsigned int label;
    struct vec2 p;
    struct list *props;
} Point2;

typedef struct point3
{
    Ts_id ts_id;                /* Tina structure identifier */
    unsigned int type;
    unsigned int label;
    struct vec3 p;
    struct list *props;
} Point3;
```

They are designed to represent physical entities in two and three dimensions. For example in corner stereo processing pairs of matched corners give rise to a **Point3** when projected into world coordinates.

```
Point2 *point2_alloc(unsigned int type)

Point3 *point3_alloc(unsigned int type)

Point2 *point2_make(Vec2 p, unsigned int type)

Point3 *point3_make(Vec3 p, int type)

Point2 *point2_copy(Point2 * point)

Point3 *point3_copy(Point3 * point)

int     point2_number(List * points)

int     point3_number(List * points)

void    point2_free(Point2 * point)

void    point3_free(Point3 * point)

void    point2_format(Point2 * point)
```

```
void    point3_format(Point3 * point)
```

Obvious functions for manipulating point geometry.

The functions **point2_copy, point3_copy, point2_free** & **point3_free** copy and free the property list see proplist description for details).

The functions **point2_format** and **point3_format** produces a standard formatted output of point geometry using the standard Tina *format* facility. Usually directed to the top level text display window.

```
Point2 *point2_proj(Point2 * point, Mat3 proj)

Point2 *point2_rectify(Point2 * point, Mat3 rect)

void    point3_transform(Point3 * point, Transform3 trans)

Bool    point3_coincident(Point3 * p1, Point3 * p2, double poserror)

Bool    point3_within_error(Point3 * p1, Point3 * p2)
```

Transformation functions for point geometry.

point2_proj performs a projective transformation of the geometry in

point according to the 3X3 projection matrix **proj_mat** and return a pointer to the transformed version.

point3_transform applies the rotation and translation **trans** directly to the **point** geometry structure.

point3_coincident checks if **p1** and **p2** are spatially coincident within t e allowed position error **poserror. point3_within_error** performs the same test but with an error deined by the **Iso_error** structure.

9.3.3 Line Geometry

Line geometry in 2D and 3D is represented by the **line2** and **line3** data structures respectively. These are of the form

```
typedef struct line2
{
    Ts_id ts_id;                /* Tina structure identifier */
    unsigned int type;
    unsigned int label;
    struct vec2 p1,p2;
    struct vec2 p, v;
    float length;
    struct list *props;
} Line2;

typedef struct line3
{
    Ts_id ts_id;                /* Tina structure identifier */
    unsigned int type;
    unsigned int label;
    struct vec3 p1,p2;   /* end points */
    struct vec3 p;
```

```
    struct vec3 v;
    float length;
    struct list *props;
} Line3;
```

Where **p1** and **p2** are the endpoints of the line, and **v** is the unit vector in the direction of the vector from **p1** to **p2.** The **length** field is the magnitude of this vecto
p specifies an additional point on the line eg the mid point, or its point of closest approach to the origin, etc) which may be of use in a number of applications.

```
Line2  *line2_alloc(unsigned int type)

Line3  *line3_alloc(unsigned int type)

Line2  *line2_make(Vec2 p1, Vec2 p2, unsigned int type)

Line3  *line3_make(Vec3 p1, Vec3 p2, int type)

void    line3_remake(Line3 * line, int type)

Line2  *line2_copy(Line2 * line)

Line3  *line3_copy(Line3 * line)  /* identical copy that shares proplist */

Line3  *line3_clone(Line3 * line) /* identical copy with null proplist */

void    line2_free(Line2 * line)

void    line3_free(Line3 * line)

void   line2_format(Line2 *line)

void   line3_format(Line3 *line)
```

Functions for manipulating line geometry. The functions **line2_copy, line3_copy, line2_free** and **line3_free** copy and free the property list see proplist description for details).
The functions **line2_format** and **line3_format** produces a standard formatted output of line geometry using the standard Tina *format* facility. Usually directed to the top level text display window.

```
Line2  *line2_negative(Line2 * line)

Line3  *line3_negative(Line3 * line)

void    line2_negate(Line2 * line)

void    line3_negate(Line3 * line)

Line2  *line2_proj(Line2 * line, Mat3 proj)

Line2  *line2_rectify(Line2 * line, Mat3 rect)
```

```
void    line3_transform(Line3 * line, Transform3 trans)

int     line3_coincident(Line3 * l1, Line3 * l2, double doterror,
                         double poserror)

Bool    line2_point_on_line(Line2 * line, Vec2 p, double thres)
```

Transformation functions for line geometry. The negate functions change the labeling of the end points of the line and the direction of the vector that connects them.

line2_proj performs a projective transformation of the geometry in **line** according to the 3X3 projection matrix **proj_mat** and return a pointer to the transformed version.

line2_transform and **line3_transform** apply the rotation and translation **trans** directly to the **line** geometry structure.

line2_coincident and **line3_coincident** check if **l1** and **l2** are spatially coincident within the allowed position and rotation errors **poserror** and **roterror** respectively.

9.3.4 Curve Geometry

Conic sections are represented by the following generic structure

```
typedef struct conic
{
    Ts_id ts_id;                /* Tina structure identifier */
    unsigned int type;
    unsigned int label;
    int filler1;
    double a, b, c, d, e, f;   /* algebraic formula */
    double theta, alpha, beta;
    struct vec2 center;
    int filler2;
    double t1, t2;       /* conic params of p1 and p2 */
    int branch;          /* for hyperbola only */
    List *props;         /* property list */
} Conic;
```

In **conic** the arc covered by the curve is between angles **t1** and **t2** in radians). All angles are positive with 0 along the
x axis and *P/2 along the **y** axis. The arc goes from **t1** to **t2** in strictly ascending order, **t1** will be less than 2 *P but
t2 may be larger than 2 *P if the arc passes back through the **x** axis.

In **conic** the implicit algebraic equation that represents its form is given by $ax^2 + 2bxy + cy^2 + 2dx + 2ey + f = 0$ **theta** is the angle of the major axis, **alpha** and **beta** are the lengths of the semi-major and semi-minor axis respectively, so the rotated, translated standard form of the conic equation is $\frac{x^2}{\alpha^2} \pm \frac{y^2}{ta^2} = 1$ The conic **type** can b any one of

```
ELLIPSE
HYPERBOLA
DEGENERATE
```

with obvious meanings. The field **branch** is used to differentiate which part of the hyperbola the data lies on.

The covariance matrix for conic data is defined by

```
typedef struct conic_stat   /* covariance matrix etc for conic */
{
    Ts_id ts_id;                /* Tina structure identifier */
    int filler;
    double x[5];
    double d[5];
    double u[5][5];
} Conic_stat;
```

The location of a conic section in 3D is made explicit using the following structure:

```
typedef struct conic3
{
    Ts_id ts_id;                /* Tina structure identifier */
    int type;
    struct conic *conic;     /* describes the conic in the plane */
    struct vec3 origin;      /* origin point on plane */
    struct vec3 ex, ey, ez;  /* define x, y and z axes: ez = normal */
} Conic3;
```

The coordinate frame in which **conic** lies is at **origin** and has axis **ex, ey,** and **ez.** Wh re the 2D conic lies in the **xy** plane.

```
Conic  *conic_alloc(unsigned int type)

Conic3 *conic3_alloc(unsigned int type)

Conic  *conic_make(int type, Vec2 center, double theta, double alpha,
                   double beta, double t1, double t2, int branch)

Conic3 *conic3_make(Conic * con2, Vec3 o, Vec3 ex, Vec3 ey, Vec3 ez)

Conic  *conic_copy(Conic * conic)

Conic3 *conic3_copy(Conic3 * con3)

void    conic_free(Conic * conic)

void    conic3_free(Conic3 *con3)

void    conic_format(Conic *conic)

void    conic3_format(Conic3 *conic)

void   *conic_prop_get(Conic * conic, int type, int prop)
```

Obvious functions for manipulating conic geometry. Note how the naming function differs from that of other geometry to reflect the fact that as well as representing 2D geometry the Conic data structure is

also part of the r presentation of the
Conic3.

The functions **conic_copy, conic3_copy, conic_free** and **conic3_free** copy and free the property list see proplist description for details). The functions **conic_format** and **conic3_format** produces a standard formatted output of conic geometry using the standard Tina *format* facility. Usually directed to the top level text display window.

```
Conic  *conic_5pt(Vec2 p1, Vec2 p2, Vec2 p3, Vec2 p4, Vec2 p5)

Conic  *conic_circ_3pt(Vec2 p1, Vec2 p2, Vec2 p3)

Conic  *conic_ellipse_5pt(Vec2 p1, Vec2 p2, Vec2 p3, Vec2 p4, Vec2 p5,
                          double min_aratio)

Conic  *conic_ellipse_3pt(Vec2 p1, Vec2 v1, Vec2 p2, Vec2 v2, Vec2 p3,
                          double min_aratio)

Conic  *conic_circ_3pt(Vec2 p1, Vec2 p2, Vec2 p3)
```

Functions for generating conics. Prefixes **conic_, conic_ellipse_** and **conic_circ_** fit totally general conics, ellipses only and circles only respectively. The **min_aratio** argument of **conic_ellipse_** fun tions determines the ellipse aspect ratio below which an ellipse fit
is replaced with a circle fit as is the case if the fit results in a hyperbola).

conic_ and **conic_ellipse_** functions can take either 5 points as arguments or 2 points with tangent unit direction vectors and an additional point.

```
void    conic_set_ends(Conic * conic, Vec2 p1, Vec2 p2, Vec2 pmid)

Vec2    conic_point(Conic * conic, double t)

Vec3    conic3_point(Conic3 * con3, double t)

double  conic_param(Conic * conic, Vec2 p)

double  conic3_param(Conic3 * con3, Vec3 p3)
```

Functions for relating position vectors and conic parameter values.

conic_set_ends sets conic end point parameter values **t1** and **t2.** respectively. The mid point **pmid** is used to identify the order of & p1 and **p2** along the conic.

Other functions convert between parameter and point values.

```
void    conic3_negate(Conic3 * con3)

Conic3 *conic3_negative(Conic3 * con3)

Conic  *conic_proj(Conic * conic, Mat3 proj)

void    conic3_transform(Conic3 * conic, Transform3 trans)

Bool    conic3_coincident(Conic3 * c1, Conic3 * c2,
```

```
                        double doterror, double poserror)

Bool    conic3_overlap(Conic3 * c1, Conic3 * c2, float *t1, float *t2)
```

Transformation functions for conic geometry. The negate functions change direction of the normal of a 3D conic the 3D locations of the end points are unchanged.

conic_proj performs a projecti e transformation of the geometry in **conic** according to the 3X3 projection matrix **proj_mat** and return a pointer to the transformed version.

conic3_transform applies the rotation and translation **trans** directly to the **conic** geomet y structure.

conic3_coincident checks if **c1** and **c2** are spatially coincident within the allowed position and rotation errors **poserror** and **roterror** respectively.

9.3.5 Plane Geometry

```
typedef struct plane
{
     Ts_id ts_id;               /* Tina structure identifier */
     unsigned int type;
     unsigned int label;
     struct vec3 p;
     struct vec3 n;
     struct list *props;

} Plane
```

data structure is used to represent planes in 3D.

```
Plane  *plane_alloc(unsigned int type)

Plane  *plane_make(Vec3 p, Vec3 n, int type)

Plane  *plane_copy(Plane * plane)

void    plane_free(Plane * plane)

void    plane_format(Plane * plane)


void    plane_transform(Plane * plane, Transform3 trans)

Bool    plane_coincident(Plane * p1, Plane * p2,
                         double dotthres, double poserror)
```

9.3.6 Transformations

The explicit definition of 3 vector transformations or the implicit coordinate frames they represent is done with the Transf3 data structure.

```
typedef struct transf3
{
     Ts_id ts_id;               /* Tina structure identifier */
```

```
     unsigned int type;
     unsigned int label;
     Transform3 T;
     struct list *props;
} Transf3;
```

Where Transform3

```
typedef struct transform3
{
    Ts_id ts_id;               /* Tina structure identifier */
    int type;
    struct mat3 R;
    struct vec3 t;
}   Transform3;
```

and also

```
typedef struct transform2
{
    Ts_id ts_id;               /* Tina structure identifier */
    int type;
    struct mat2 R;
    struct vec2 t;
} Transform2;
```

represent a rotation and translation matrix. Transformations are achieved
according to $x' = Rx + t$ The physical meaning of the transformation in terms of coordinate frames is that the application of the transformation takes geometry from the current frame to the new frame see the Math Library). Hence $\mathbf{t}$ is the transformation of frame 1 with respect to frame 2 and the column vectors of $\mathbf{R}$ are the unit x, y and z vectors of frame 1 with respect to the coordinates of frame2.

Again Tranf3 is principally used by Geomstat.

```
Transf3 *transf3_alloc(unsigned int type)

Transf3 *transf3_make(Transform3 T, int type)

Transf3 *transf3_copy(Transf3 * transf)

void     transf3_free(Transf3 * transf)

void     transf3_format(Transf3 * transf)

int      transf3_number(List * transfs)
```

9.3.7 Generic Geometry Functions

The geometrical primitives described above are grouped together in a body of Tina code that is able to treat them generically. These functions use pointers to geometrical rimitives and the following predefined type definitions:

```
SCALAR
POINT2
POINT3
LINE2
LINE3
CONIC2
CONIC3
PLANE
TRANSF3
```

Tina functions that apply to generic geometry are prefixed *geom_*. Some functions may not be defined for all possible types in which case they will have no effect and/or return ULL.

Geometry is often grouped and manipulated using grouping data structures described in other sections of the programmers guide. Most common amongst these is *reclist* data structure. This allows underlying image structure to be preserved here
it is thought necessary. Some functions, however, only allow simple lists of geometric primitives. Hence conversion is sometimes necessary.

```
void   *geom_alloc(unsigned int type, unsigned int internal_type)

void   *geom_copy(void *geom, int type)

void   *geom_update_copy(void *geom, int *type)

void    geom_format(void *geom, unsigned int type)

int     geom_label_get(void *p, int type)

void   *geom_getbylabel(List * geom, int label, int *type)
```

Generic functions for primitive structure manipulation. The function

geom_update_copy is the same as **geom_copy** in except that it uses a pointer to the type rather than the type itself this allows it to be used in conju ction with *reclist* and other functions that allow for changes in the form of a structure or the elements it access Eg:

reclist_update and **reclist_flat)**

geom_label_get recovers the label field of the geometrical primitives they all have one). As they are all unique over all structures) they can be used as a universal
identifier.

```
void   *geom_prop_get(void *p, int type, int prop)

void    geom_prop_add(void *geom, int type, void *prop, int prop_type,
                      void (*freefunc) ( /* ??? */ ))

void    geom_prop_addifnp(void *geom, int type, void *prop, int prop_type,
                          void (*freefunc) ( /* ??? */ ), Bool dofree)

void   *geom_prop_update_get(void *p, int *type, int prop)
```

Generic property list manipulation functions.

```
void    geom_negate(void *geom, int type)

void   *geom_negative(void *geom, int type)

void    geom_transform(void *geom, int type, Transform3 * trans)
```

Generic transformation functions.

```
List   *geom_list_make_flat(List * list)

List   *geom_string_make_flat(Tstring * string)
```

Functions to make recursive lists into flat lists. The old recursive list data structure is destroyed but not the elements it points to. They are now pointed to by the new flat list data structure.

Chapter 10

Camera Geometry

10.1 Introduction

The camera models introduced here are intended to be as general as possible. We adopt the conventional rotation matrix and translation approach rather than the more popular homogenous notation and "essential matrices" commonly found in the literature. There are two reasons for this, the first is that all cameras (that we are aware of) have orthogonal measurement axes and the real world is cartesian. Therefore the ability to represent arbitrary skews is unnecessary and deliberately mixing this with spatial warping is unhelpful [1] when attempting to parameterise a system. In addition, though homogenous notation may appear superficially simple, software implementations of conventioanl co-ordinate transformations are more easily interpretted by PhD level researchers than the equivalent homogenous form (which requires significant additional documentation to explain).

Single, binocular, trinocular, temporal and other more esoteric combinations of camera geometries can be representated. Camera structures represent both the physical properties of cameras (intrinsic parameters) and information required to compute the mapping between standard iconic image coordinates and camera based image coordinates (extrinsic parameters) from which actual measurements in the world can be derived (including the removal of any distortion that is known to be present as a result of the imaging process).

Enough information is maintained in the camera for its parameters to be rescaled to any size of image. Hence, it is possible to derive from the camera model associated with an image of one size the camera models of all scaled variants of it. These may result from either hardware of software quantization or image expansion (with interpolated pixel values).

Tina camera structure definitions and typedefs (amongst other things) can be included with

```
#include <tina/vision/visDef.h>
```

function declarations with

```
#include <tina/vision/visPro.h>
```

programs should be compiled with the libraries

```
-ltinavision -ltinamath -ltinasys
```

[1] Though it appears to result in many publication possibilities

10.2 Describing Cameras

In Tina cameras are represented by the following structure

```
typedef struct camera
  {
      Ts_id   ts_id;            /* Tina structure identifier */
      /** camera info **/
      unsigned int type;
      unsigned int label;

      /** physical parameters **/
      float   f;                /* focal length */
      float   pixel;            /* notional pixel size */
      float   ax, ay;           /* x and y expansion factors (aspect ratio ) */
      float   cx, cy;           /* x and y image centre coordinates */
      int     width, height;    /* image height and width for which relevant */

      /** transformation from world to camera frame **/
      Transform3 *transf;

      /** optical distortion **/
      void   *distort_params;
      void   *(*copy_dist_func) ();
      Vec2 (*distort_func) ();
      Vec2 (*correct_func) ();

      /** projection from unit camera to image coordinates **/
      Mat3    cam_to_im;
      /** projection from image to unit camera coordinates **/
      Mat3    im_to_cam;
  }
Camera;
```

The **type** and **label** fields allow the system/user to keep a track on arbitary numbers of cameras.

A number of physical parameters are given. These describe a simple imaging model based on a pinhole camera. Some are image size dependent, and hence **width** and **height** fields are included to keep track on the size of image for which the camera definition is valid. For generality two expansion factors (with respect to the given **pixel** size) **ax** and **ay** are used to define the aspect ratio of the camera. For CCD arrays it is possible to fix at least one of these scale factors, to unity (as it is the same in all cameras of that type). Usually **ay** will be 1.0 as the spacing between scan lines is fixed.

The focal length **f** (if known or recoverable by calibration) is in the standard units of the Tina system (usually millimeters). Image centers **(cx, cy)** are in pixels and scale with the **width** and **height** of the image.

If available the **transf** field is used to store a pointer to a **Transform3** which represents the transformation that takes points in a world coordinate frame into the coordinate frame of the physical camera. That is, at the optic center, with **z** along the principle axis and **x** and **y** parallel to the imaging plane; **x** along the image horizontal (+ve right) and **y** vertical (+ve down).

The two **Mat3** fields **cam_to_im** and **im_to_cam** describe projective mappings between iconic image coordinates (measurements in pixels, origin at the top left of the image, etc) and camera image coordinates (metric units, origin at the center, **x** axis horizontal +ve to the right, **y** axis vertical +ve down, etc). That is

```
q' = Pq
```

where **q** and **q'** are 3 vectors representing points in each image coordinate frame and **P** is the projective mapping. Note that the 3 vector **q = (x, y, w)** represents the 2 vector image coordinate **(x/w, y/w).** The projection matrix **cam_to_im** takes camera image coordinates to their standard counterparts and **im_to_cam** the reverse. Each is derived, to some extent, from the physical parameters described above, and are dependent upon the **height** and **with** of the image (hence must be recomputed if the size of the image is changed).

The above mappings are able to remove known projective distortions of the image. Additionally, optional distortion and correction functions are supplied to deal with non projective distortions, these are **distort_func** and **correct_func** respectively. The field **distort_params** is available to store information required by these distortion functions. The standard form of one of these functions is

```
Vec2    cam_distort(Vec2 w, double *a)
```

Distortions/corrections take place in camera image coordinates and not standard image coordinates. Hence in order to obtain corrected standard image coords points must first be projected using **im_to_cam,** the correction applied using **correct_func,** and reprojected using **cam_to_im.**

This camera arrangement allows considerable leeway as to the actual representation of data. For instance, the projection plane of the camera image is not defined. Two obvious candidate locations exist, that is either at unity or at the focal length of the camera. For reasons of uniformity, we have adopted the former (unity) in the projection functions used in Tina.

By way of an example the sequence to of operations to project a point **v** in word coordinate frames to the point **w** in the standard iconic image frame is as follows

```
Vec3 v;
Vec2 w;
Vec2 rectify_pos();

/** transform into camera frame **/
if (cam->transf != NULL)
    v = trans3_pos(*cam->transf, v);

/** project onto unit focal plane **/
w = proj2_of_vec3(v);

/** allow for optical distortion **/
if (cam->distort_params != NULL &&      cam->distort_func != NULL)
    w = cam->distort_func(w, cam->distort_params);

/** change to image coordinates **/
w = rectify_pos(cam->cam_to_im, w);
```

The macro **proj2_of_vec3** takes the **Vec3 (x, y, z)** to the **Vec2 (x/z, y/z).** The function

```
Vec2    rectify_pos(Mat3 rect, Vec2 p)
```

applies the projective transform represented by **P** to image coords **q** and returns new image coordinates.

10.2.1 Parallel Stereo Camera Geometry

It is helpful to represent arbitrary stereo camera geometries in terms of their relationship to an equivalent parallel camera stereo geometry. In such a geometry the "epipolar" matching constraint becomes a same raster constraint and the projection of disparity coordinates to world coordinates is much simplified. This arrangement is represented in Tina using the **parcam** structure

```
typedef struct parcam
  {
      Ts_id   ts_id;            /* Tina structure identifier */
      /** camera info **/
      unsigned int type;
      unsigned int label;

      float   f;                /* notional focal length */
      float   I;                /* interocular separation */
      float   pixel;            /* notional pixel size */

      /* cameras and rectified counter parts */
      struct camera *cam1;      /* original camera 1 */
      struct camera *rcam1;     /* rectified camera 1 */
      struct camera *cam2;      /* original camera 2 */
      struct camera *rcam2;     /* rectified camera 2 */

      struct mat3 rect1;        /* rectification matrix for camera 1 */
      struct mat3 derect1;      /* derectification matrix for camera 1 */

      struct mat3 rect2;        /* rectification matrix for camera 2 */
      struct mat3 derect2;      /* derectification matrix for camera 2 */

      struct mat3 e;            /* epipolar colineation matrix */
  } Parcam;
```

Here focal length **f** has a notional value which may or may not be derived from that of either physical camera. In the case of edge based stereo processing in Tina, it is convenient for the representation of edge data transformed into the parallel image representation to choose **f** to be the the same as the focal length of the camera **cam1.** Similarly **pixel** is the pixel size used to represent image related data in the parallel camera image frames.

I is the magnitude of the interocular separation between the optical centers of the two cameras (either physical or parallel). It is assumed that the two cameras from which the parallel camera geometry is derived, **cam1** and **cam2,** have non NULL **transf** fields that are defined with respect to the same world coordinate frame.

rcam1 and **rcam2,** if not NULL, are camera structures that represent a parallel camera geometry equivalent to the general camera geometry defined by **cam1** and **cam2.** The **transf** of all cameras are defined with respect to the same world coordinate frame.

Projective transforms to and from each camera and respective parallel cameras are represented by **rect1, derect1, rect2,** and **derect2. rect,** short for the process of rectification, map image coordinates in non parallel cameras to parallel cameras, and **derect** the converse. Again considerable flexibility exist here and the exact definition of which coordinates the rectification is between is left to the requirements of the system using the **parcam** structure.

Finally the **Mat3** field **e** is included to represent the epipolar constraint matrix. This is useful for situations in which the relationship between the cameras are not derived from the physical transformation (known or obtained from prior calibration) but are computed from the image data itself.

10.3 Camera Functions

```
Camera *cam_alloc(int type);

Camera *cam_make(unsigned int type, Transform3 * transf, double f,
                 double pix, double ax, double ay,
                 double cx, double cy, int width, int height);

Camera *cam_copy(Camera * cam);

void *(*distort_param_copy_func)(double *a);

void    cam_free(Camera * cam);
```

Camera allocation, make, copy and free functions.

```
Bool    cam_scale_to_image(Camera * cam, Imrect * im);
```

Scale current camera to match image, and return **true** on success. The image **height** and **width** must be in the same integer ratio with respect to the existing camera **width** and **height.**

```
Parcam *parcam_alloc(unsigned int type);

Parcam *parcam_make(Camera * cam1, Camera * cam2, unsigned int type);

Parcam *parcam_scaled_make(Camera * cam1, Camera * cam2, double scale,
                           unsigned int type);
void    pcam_free(Parcam * pcam);
```

Parallel camera allocation and make functions. **parcam_make** constructs a standard form of parallel camera from the arbitary (but sensible) camera geometries defined by **cam1** and **cam2.** The rectification functions are set to map between standard image coordinates of the non parallel cameras and camera based image coordinates for the parallel cameras. The image plane of the later and the focal length ield of the **parcam** struvture are set to the focal lenght of **cam1** as this simplifies the transformation of edge data prior to stereo matching.

```
Vec2    cam_proj(Camera * cam, Vec3 v);

void    cam_ray(Camera * cam, Vec2 u, Vec3 * p, Vec3 * v);
```

Use **cam** to relate image and world coorinate frames. **cam_proj** projects the point **v** in world coordinates into the standard image coordinates associated with **cam. cam_ray** computes in world coordinates a ray (specified by a point **p** and unit direction vector **v)** that passes through the standard image location **u.** The point **p** is actually the location of the optic centre and hence identical for all image points associated with the same camera.

10.4 Rectification Functions

```
extern Vec2    rectify_pos(Mat3 rect, Vec2 p);
```

```
extern double  rectify_orient(Mat3 rect, Vec2 p, double or);

extern void    rectify_pos_and_dir(Mat3 rect, Vec2 * p, Vec2 * v);
```

Various rectification function. Each computes projective maps for image locations, orientations and direction vectors. Image loaction and direction are represented as a **Vec2** and orientation in radians. The projective transform is given by the **Mat3 rect.**

```
void    edge_apply_rect(Edgel * edge, int type, Mat3 * rect);

void    edge_apply_derect(Edgel * edge, int type, Mat3 * rect);

void    edge_add_rect_prop(Edgel * edge, int type, Mat3 * rect);

void    er_add_rectpos_prop(Imrect * er, Mat3 rect);

void    er_rectify(Imrect * er, Mat3 rect);

void    er_derectify(Imrect * er, Mat3 rect);
```

Functions to rectify an individual **edge** or all edges in the edgerect **er.** The **type** argument in **edge_apply_rect** and **edge_apply_derect** conforms to standard Tina indirect calling functions.

10.5 Parallel Projection

Simple module to convert parallel camera disparity vectors to 3D world vectors and their complements. Disparity is defined as right camera image coordinates minus left camera image coordinates. Points in disparity space are of type **Vec3** with **x** and **y** in the image **z** set to disparity. Values of focal length **f** and interocular separation **I** must be recovered from approriate **parcam** structure. Avoids the necessity of carrying round extraneous camera definitions at the expence of some generality.

```
extern void    par_proj_set(double fnew, double Inew);

extern void    par_proj_get(float *fp, float *Ip);
```

Module initialisation function that sets current **f** and **I** values. Suitable values must be provided prior to projection.

```
extern void    par_proj_ray(Vec2 u, Vec3 * p, Vec3 * v);

extern Vec3    vec3_par_proj_3d(Vec3 p);

extern Vec3    vec3_par_proj_disp(Vec3 p);

extern void    vec3_pp3d_inplace(Vec3 * p);

extern void    vec3_ppdisp_inplace(Vec3 * p);
```

Project a single point **p** between disparity and 3D coordinate frames.
vec3_par_proj_3d takes a point in disparity and returns a 3D point and
vec3_par_proj_disp the converse.

```
extern void    line3_par_proj_3d(Line3 * line);

extern void    line3_par_proj_disp(Line3 * line);

extern void    plane_par_proj_3d(Plane * plane);
```

Projection functions for **Line3** and **Plane**.

Chapter 11

Stereo matching

This section provides an overview of the philosophy used in edge string based stereo matching.

11.1 Stereo Index

Prior to matching primitives are indexed according to epipolar bands. This simplifies (in terms of both code complexity and computational efficiency) the subsequent matching problem. The easiest way to do this is to transform edge (or other feature primitives) to a suitable and equivalent virtual parallel camera set up. It is important here to note that it is only the geometry of the primitive, as opposed to the physical image location through which it is indexed (cf. the Imrect data structure), that requires transformation.

After transformation the pseudo rasters of the parallel camera images correspond to the original epipolar indices. When identifying which primitives are located on each raster care must be taken to allow for possible distortion in the image geometry that may have occurred during the transformation process. This can result in both multiple and zero mappings from physical image to pseudo image rasters (see below for the case of edge strings).

The Windex data structure is used

```
typedef struct windex /* generic window index structure */
{
    Ts_id ts_id;                /* Tina structure identifier */
    int type;
    int m, n;  /* m rows n cols */
    struct imregion *region; /* region covered */
    void ***index;
} Windex;
```

This is a general purpose 2D index. In the stereo case the first dimension is epipolar/raster and the second position along the raster quantised by m and n (ie m buckets of width n pixels). The region of interest covered by the index is given by the Imregion structure *region.*

```
Windex *w;
int x,y;

   .
   .
```

```
   List *r_index;
   Imregion *roi;
   int c;

   roi = w->region;
   c = (x-roi->lx)/w->n;  /* column offset within region of interest */

   if (c<w->m) /* covered by region */
  r_index = w->index[y][c]
```

The above code sample illustrates use of the indexing scheme to recover a list (ordered along the raster) of all primitives that occur in and beyond the raster bucket that includes edge location column x on raster y (where x and y are in chosen image coordinates). All edges in the raster can be indexed by the degenerate case

`w->index[y][0]`

Note that all indices refer to the same list, that is

`for all (j>i) w->index[y][j] is a subset of w->index[y][i]`

Stereo indices can be allocated and freed.

```
Windex *sx_alloc(Imregion *region,int n,int type)

void sx_free(Windex *w)
```

11.2 Edges

In the case of previously transformed edge data (using an appropriate rectification function) the stereo index is compiled by traversing edge strings the stereare traveresed

11.3 Matches

Matches are sought at the edge string level. That is string structures of edgels. These are not whole strings but those previously indexed to run along rasters of the parallel camera geometry. Each such string approximates that part of the path of an edge along rasters of the image.

11.4 Edge and Stereo Data Structures

This section presents a series of figures and extended legends that illustrate the principle data structures used represent connected edge strings and stereo matches. These are the data structures that result from canny edge processing and linking and are used by the current stereo algorithm. They could also result from other edge processes and edge based stereo algorithms.

Other data structures may exist transiently in the processing stages that compute linked edges and locate stereo matches. There existence however can not be assumed outside the particular processing stage.

Even these approved data structures may not exist by default. But once in existence they can be utilised by other processing modules than the one used to create them.

In the light of experience and to allow the system to be flexible to the requirements of the user, the basic representation of Tina primitives has been kept as simple and uniform as possible. Extendibility and generality is achieved by using a property list for the storage of additional information specific to each different application. The property lists field is always defined in other Tina structures as

```
struct list *props;
```

Property lists are used throughout the Tina system (they are described in the chapter of the programmers guide on list processing).

Extra information that describes the relationships between primitives (such as edge strings or order along the image rasters) is kept outside the primitives themselves. This approach further enhances the modularity of the representations though at the cost of an increas in complexity. The most important forms of generic data structure used to combine data throughout the Tina system are lists, doubly directed lists, and strings (see the programmers guide).

11.5 Edgerect

See figure 1

The **Imrect** data structure (details in programmers guide) with **vtype** set to **ptr_v** is used as the principal iconic representation for edge data.

```
typedef struct edgel
{
    Ts_id ts_id;                /* Tina structure identifier */
    unsigned int type; /* includes information as to rectification status */
    unsigned int label;
    struct vec2 pos;
    float orient; /* these could go to props list perhaps NAT */
    float contrast;
    struct list *props;
} Edgel;
```

The Vec2 **pos** is the sub pixel location of the edge with respect to the underlying image (or a transformed version of it). Where the center of the pixel indexed **[i, j]** is at sub pixel image location **[i+0.5, j+0.5].** The fields **orient** and **con** represent the orientation of the edge and its absolute image contrast. The contrast of an edge is proportional to the absolute gradient across it. The orientation of an edge, given in radians is given by the local direction along the edge (- to with + clockwise). Orientations 0, and - are all horizontal, with 0 being light above dark and and - both being dark above light.

Individual edgels in the Imrect **er** can be indexed

```
er->array.ptr_v[i][j]
```

or using the shorthand macro

```
IM_PTR(er, i, j)
```

which will be NULL if no edgel is present at image location [i, j] (row i column j). The edge array is only defined for the **region** defined in the Imrect er.

The sparse edgel array can be searched more rapidly using the raster index **rows** that is accessed through the property list of the Imrect **er** by the proplist type ER_ROWS.

```
typedef struct rindex /* raster index to each row of an imrect */
{
    Ts_id ts_id;                /* Tina structure identifier */
```

```
    int type;                /* some form of feature pointers */
    struct imregion *region; /* region covered */
    void **index;            /* array of raster lists */
} Rindex;
```

where

`rows->index[i]`

is the list of edges on row i of the edgel array and is allocated using **rx_alloc()** .

11.6 Edge Strings

See figure 2.

Spatially connected edge strings are represented using the generic tina string structure (see programming guide). The **start** and **end** fields point to a pair of elements in a doubly directed list which in turn point to the individual **edgels** (one for each element in the list). Sub-strings chosen from complete edge strings can, if required, be constructed from the same list elements (they just have different start and end points).

The string to which an individual **edgel** belongs can be placed upon the the property list of the edgel using the key STRING. The list of strings for a particular edgerect (imrect of edges) is stored on its property list using STRINGS (or ER_STRINGS in old versions).

11.7 Stereo Index

See figure 3. Prior to matching edges are indexed according to epipolar bands. This simplifies (in terms of both code complexity and computational efficiency) the subsequent matching problem. This is done by transforming each edge to a suitable and equivalent virtual parallel camera set up. It is important here to note that it is only the geometry of the edge, as opposed to the physical location in the edgerect array, that is transformed. Additional distortion in the image geometry (eg radial distortion) that may have occurred during the transformation process can also be incorporated at this stage.

Note that edge transformation (called rectification) can result in both multiple and zero mappings from physical image to virtual image rasters.

The stereo index, used to index rectified rasters, is stored on the property list of the edgrect using SINDEX, a structure of type Windex (above). This is a general purpose 2D index. In the stereo case the first dimension is epipolar/raster and the second position along the raster quantised by **m** and **n** (ie **m** buckets of width **n** pixels). The region of interest covered by the index is given by the Imregion structure **region.**

```
Windex *w;
int x,y;

   .
   .

    List *r_index;
    Imregion *roi;
    int c;

    roi = w->region;
    c = (x-roi->lx)/w->n;  /* column offset within region of interest */
```

```
    if (c<w->m) /* covered by region */
      r_index = w->index[y][c]
```

The above code sample illustrates use of the indexing scheme to recover a list (ordered along the raster) of all primitives that occur in and beyond the raster bucket that includes edge location column x on raster y (where x and y are in chosen image coordinates). All edges in the raster can be indexed by the degenerate case

`w->index[y][0]`

Note that all indices refer to the same list, that is

`for all (j>i) w->index[y][j] is a subset of w->index[y][i]`

In order to avoid unnecessary excess ambiguity, stereo matches are not sought between individual edgels. Instead connected edges that cross an individual raster are grouped into sub-strings. These have type field set to FORWARD or BACKWARD depending upon the direction they run along the raster. Note that to satisfy the situation where no edge from a particular edge string transforms to lie upon a particular virtual raster that the string crosses, then the same edge sub-string is allowed to be referenced on a number of adjacent virtual rasters (determined from the meta connectivity of the edge string).

The sub-string used in the stereo index (which can of course refer to a single edgel) can be referenced through the SINDEX property of the edgel.

This rather complex data structure has the considerable advantage that stereo matching can be carried out at a different resolution to the original image by simply changing the rectification function. Increasing the coarseness of the epipolar band does not increase the degree of within epipolar ambiguity. It follows also that the left and right hand images do not have to be the same size as long as the rectification functions produce virtual images of a constant size. In Tina the rectification functions are currently standardised about the left image camera model.

11.8 Matching Sub Strings

See figure 4.

Matching is performed left to right. Sub-strings in the left image have an MLIST property which points to a list of matches within which they are included (always on the **to1** field)

```
typedef struct match   /* a genericly defined matching structure */
{
    Ts_id ts_id;               /* Tina structure identifier */
    int type;
    int label;
    float weight;
    void *to1;
    void *to2;
    struct list *props;
} Match;
```

Hence individual edgels are not matches as such.

Following the application of a suitable stereo matching algorithm the match list for each edge string is reduced to correct matches only (possibly NULL).

11.9 Matching Whole Strings

See figure 5. Matches between sub-strings can be grouped at the whole string level. Again matching is based firmly on the left hand image. The representation of matches between whole strings is very different. The edge string property MLIST refers to a list of lists of matches, each sub list is associated with a particular edge string from the right image. The integer value of the pointer to the string in question is used for the type field of the appropriate element in the top level list (this is used to group and discriminate matches to the same edges string).

Appendix A

Type Definitions

The following is a list of enum, union, and integer type definitions currently used in Tina. They are used throught the code to specify the types and meanings of varoius structures and substructures.

Integer type definitions fall into the following categories which are grouped in terms of their numeric values

1.....99 used to define system data structures

100...199 used to define math types

200...299 used to define Tv types

300...999 used to define Tina data structures

1000... available to the user

Note that the NULL type 0 is not used explicitly. This allows the type NULL to be used both as a catch all and in situations where it is thought unnecessary to specify the type of the elements.