Using the PIT `NTadt` Image datatypes under Microsoft Developer Studio*

William Rucklidge Matt Welsh

January 21, 1998

1 Introduction

This document describes the use of the `NTadt` library at the Cornell Robotics and Vision laboratory (the unix version [named `libadt`] of this library is no longer maintained.) This library contains routines to load, store, create, destroy, and manipulate multiple kinds of images. The routines are aimed primarily for use in machine vision research.

2 Basic Use

To use the image datatypes:

- Put, somewhere at the beginning of your C or C++ program,
  `#include "NTadt.h"`
  Your source files must have the extension `cpp` for compilation and linking to work properly. (You can however use C or C++ code, but some C++ types such as bool are used as parameters to functions)
- In your Developer Studio Workspace, goto `Project`→`Settings`→`Link`→`Input`, and under the `Object/library modules` setting add `NTadt.lib` to your Release Configurations and `NTadtg.lib` to your Debug Configurations.
- Make sure all the library files are in your build directory (or that you enter the files full pathnames when you #include or link to them.

*The `NTadt` library and documentation is currently being revised and maintained by Walter Bell (`wbell@cs.cornell.edu`)

Type Name	Type Tag	Pixel Type
BinaryImage	IMAGE_BINARY	char
GrayImage	IMAGE_GRAY	unsigned char
ShortImage	IMAGE_SHORT	short
LongImage	IMAGE_LONG	long
FloatImage	IMAGE_FLOAT	float
DoubleImage	IMAGE_DOUBLE	double
RGBImage	IMAGE_RGB	RGB
PtrImage	IMAGE_PTR	void *

Table 1: Types defined by the image module

2.1 Image Types Defined

The image module defines several new datatypes, representing images with different types of pixels. These are all treated identically: any operation works on any type of image (with some minor exceptions). The types defined are shown in Table 1. Thus, for example, a `GrayImage` represents a rectangular area of pixels, each of which is an `unsigned char`. The type tags are constants that are used to tell the image module what sort of image you want to create. They are all constants of type `ImageType`.

The only type here which is not a basic C/C++ type is `RGB`. This is defined by:

```
typedef struct {
    unsigned char r, g, b;
    } RGB;
```

2.2 Accessing Pixels

To access a pixel in an image, you use the `imRef` macro. The syntax is `imRef(im, x, y)` For example:

```
    GrayImage myim;
    int x, y;
    unsigned char pixval;
    ...
    imRef(myim, x - 1, y) = pixval;
```

```
...
pixval = imRef(myim, x + 2, y - 1) - 3;
```

The `x` and `y` coordinates determine which pixel is accessed. Note that you assign to pixels using `imRef`. `imRef` references a value of the type appropriate for the image given as its first argument (here, an `unsigned char` from a `GrayImage`). The coordinates given must be in the valid range of coordinates for that image.

2.3 Getting Information About Images

An image is basically a rectangular area of memory. You can get the height and width of this area using `imGetWidth` and `imGetHeight`:

```
RGBImage im;
unsigned width, height;

width = imGetWidth(im);
height = imGetHeight(im);
```

Note that the values returned by this macro are *unsigned integers*. C is a little odd where unsigned integers are involved; thus, in this piece of code:

```
int y;
for (y = -1; y <= imGetHeight(im); y++) {
    ...
    }
```

the loop will never get executed at all, no matter what the height of the image is. You should use `y <= (int)imGetHeight(im)` to avoid this problem. This is only an issue when you compare a negative (signed) integer with an unsigned integer.

2.4 Creating and Destroying Images

It's hard to work on images if you don't have any images to work on, so I have provided a routine to create one. You tell it what sort of image you want to create, and how big you want it to be. The formal definition is

```
extern void * imNew(ImageType tag, unsigned width, unsigned height);
```

and you might use it thus:

```
DoubleImage im;

im = (DoubleImage) imNew(IMAGE_DOUBLE, 256, 512);
```

This would create a 256 pixel wide, 512 pixel high array of doubles.

`imNew` guarantees that every pixel in the new image will be cleared to zero (of the appropriate type; for example, every pixel in a `PtrImage` will be initially `(void *)NULL`).

If `imNew` cannot create the image, it returns `(void *)NULL`. You should check for this:

```
DoubleImage im;

im = (DoubleImage) imNew(IMAGE_DOUBLE, 256, 512);
if (im == (DoubleImage)NULL) {
    printf("unable to allocate 256 by 512 image of doubles\n");
    }
```

An image can be duplicated by `imDup`. This creates an exact duplicate of the original image: it has the same size, type, contents, offsets and comment as the original. It returns `(void *)NULL` if it is unable to create the duplicate. For example:

```
FloatImage im;
FloatImage imcopy;

imcopy = (DoubleImage) imDup(im);
if (imcopy == (DoubleImage)NULL) {
    printf("unable to duplicate im\n");
    }
```

You also need to get rid of images you no longer need or your program will leak memory. You use `imFree` to do this:

```
imFree(im);
```

This works no matter what type of image you pass it. You should make sure that you free all the images you create. This is not just the images created with `imNew` but also images loaded from files, or created by `imDup` or one of the conversion routines.

Note that if the image is a `PtrImage` and some of the pixels contain pointers to allocated storage that should be freed along with the image, you will have to do this yourself; `imFree` will not do it for you.

2.5 Loading and Saving Images

You can load and save these images using the routines `imLoad`, `imLoadF`, `imLoadH`, `imSave`, `imSaveF` and `imSaveH`. Their definitions are:

```
extern void * imLoad(ImageType tag, char * filename);
extern void * imLoadF(ImageType tag, FILE * inFile);
extern void * imLoadH(ImageType tag, int fileHandle);
extern int imSave(Image im, char *filename);
extern int imSaveF(Image im, FILE *outFile);
extern int imSaveF(Image im, int fileHandle);
```

`imLoad` opens the given file and reads an image from it. The image contained in the file should match the tag (or be promotable to that type; see below). It returns `(void *)NULL` if it cannot open the file, or if the file contains the wrong sort of image (or something that isn't an image), or if the load fails in some other way. If it succeeds, it returns the appropriate sort of Image (i.e. one that matches the tag it was passed).

`imLoadF` and `imLoadH` do the same job as `imLoad`, except that they read from the given stdio stream (i.e. `FILE *`) or file handle respectively. The stream or handle must be initially positioned to just before the image. If the load succeeds, then the stream will end up positioned just after the image. If the load fails, the position of the stream is undefined.

`imSave` attempts to save the given Image (of whatever type) out to the given filename. It returns 0 if it succeeds, -1 if it fails. If it fails, the contents of the file are undefined.

`imSaveF` and `imSaveH` attempts to save the given Image out to the given stream. It returns 0 if it succeeds, -1 if it fails. If it succeeds, the stream ends up positioned just after the newly written image. If it fails, it is undefined what was written to the stream.

It is illegal to attempt to load or save a `PtrImage`.

2.6 What Values Mean

Since images are often used to represent actual scenes, as acquired from a camera or other source, you need to know what values represent what intensity values.

- A pixel in a `BinaryImage` can have the values 0 or 1. By convention, 0 represents white and 1 represents black.

- A pixel in a `GrayImage` can have any value in the range 0 to `COLRNG - 1` inclusive (`COLRNG` is a constant defined in `image.h`). 0 represents black, and `COLRNG - 1` represents white (i.e. higher values represent greater intensities).
- A pixel in an `RGBImage` has three components (red, green and blue). Each component can have a value in the range 0 to `COLRNG - 1`. Higher values represent higher intensities.
- For `FloatImage`, `DoubleImage`, `ShortImage`, `LongImage` and `PtrImage`, the meaning of a pixel's value are left up to you.

2.7 An Example

You've just been given a lot of information which may not make a lot of sense. With any luck, the example program shown in Figure 1 will make things clearer. It is a program which reads in a binary image and generates an `RGBImage` which is basically a coloured version of the original image, padded out (or truncated) to a certain size (640 by 480). It reads the original image from `stdin` and writes the final image to `stdout`.

There are a few things to note about this program. First, note that it makes sure that every reference will be in range before using `imRef`. Second, note that the inner loops are in the order

```
for (y = 0; y < dispheight; y++) {
    for (x = 0; x < dispwidth; x++) {
```

The program would work if these loops were interchanged. However, due to the way that the virtual memory and cache systems work on most computers these days, it might run considerably slower, so you should use this order for your loops whenever possible.

2.8 Talking To The World

The files produced by `imSave` for `BinaryImage`, `GrayImage` and `RGBImage` images are in the popular PBM format. This format is accepted by a number of utilities. In particular, `xview` will display them and `pnmtops` will convert them to PostScript so that they may be printed. Files produced by saving other types of images are not in any widely accepted format.

```
#include "NTadt.h"
#include <stdio.h>
#include <io.h>

/* Background colour - light blue */
static RGB bgcolour = { COLRNG / 3, COLRNG / 3, COLRNG - 1 };

/* Foreground colour - white */
static RGB fgcolour = { COLRNG - 1, COLRNG - 1, COLRNG - 1 };

int main(int argc, char **argv)
{
    BinaryImage bim;
    RGBImage im;
    unsigned dispwidth = 640, dispheight = 480;
    int x, y, xoff, yoff;

/* Set stdin to take binary input */
int result= _setmode(_fileno(stdin), _O_BINARY);
if(result==-1) exit(1); /* Can't set stdin */

    /* Load the original image from stdin */
    bim = (BinaryImage)imLoadF(IMAGE_BINARY, stdin);
    if (bim == (BinaryImage)NULL) {
        exit(1);        /* Can't load image */
        }

    /* Create the new image */
    im = (RGBImage)imNew(IMAGE_RGB, dispwidth, dispheight);
    if (im == (RGBImage)NULL) {
        exit(1);        /* Can't allocate new image */
        }

    /* Determine where to put the original image, in order to make it centred */
    xoff = ((int)dispwidth - (int)imGetWidth(bim)) / 2;
    yoff = ((int)dispheight - (int)imGetHeight(bim)) / 2;

    /* Fill in the new image */
    for (y = 0; y < dispheight; y++) {
        for (x = 0; x < dispwidth; x++) {
            /*
             * If the pixel falls inside the offset original image, and that original
             * pixel is on, it's foreground; if not, it's background.
             */
            if ((x >= xoff) && (x < xoff + (int)imGetWidth(bim)) &&
                (y >= yoff) && (y < yoff + (int)imGetHeight(bim)) &&
                (imRef(bim, x - xoff, y - yoff))) {
                imRef(im, x, y) = fgcolour;                /* Foreground */
                }
            else {
                imRef(im, x, y) = bgcolour;                /* Background */
                }
            }
        }

/* Set stdout to take binary input */
result= _setmode(_fileno(stdout), _O_BINARY);
if(result==-1) exit(1); /* Can't set stdout */

    /* Write out the new image */
    if (imSaveF(im, stdout) < 0) {
        exit(1);        /* Can't write new image */
        }
    return(0);          /* Success */
    }
```

Figure 1: An Example Program

3 Advanced Use

This section describes some of the advanced features of the image module. You don't need to know any of this to make good use of the module, but there are a number of things which people commonly do which are supported.

3.1 Image Comments

Every image may have a string attached to it, known as its comment. This string can be obtained by `imGetComment(im)`, which returns a `char *` pointer to the (null-terminated) string. This string must not be modified or freed. If the image has no comment, `imGetComment` will return `(char *)NULL`.

To set an image's comment, use `imSetComment(im, char *newstr)`. This makes a copy of the given string and attaches it to the image. It returns non-zero if it successful, and zero if it fails. Note that a copy of the string is made; the original string will no longer be referenced, and may be freed if desired. To make an image have no comment associated with it, use `imSetComment(im, (char *)NULL)`.

An image's comment (if any) is stored along with the image when it is saved using `imSave` or `imSaveF`. Due to a restriction in the file format, any lines in the string which are longer than 69 characters will be broken. If you put newlines (`'\n'`) in the string, they will be obeyed (will cause line breaks).

Any comment saved out with an image will be read in when the image is loaded, and attached to the image. Comments are stored in the file as lines beginning with `#`.

3.2 Offset Images

In some applications, it is useful to have the top left corner of the image have some coordinates other than `(0, 0)`. For example, if we want to put borders around the original contents of an image, but want to maintain the original coordinate scheme (i.e. the pixel which was `(0, 0)` in the original image is still `(0, 0)` in the expanded image, even though there is now data above it and to its left), we will need to have some pixels with negative coordinates.

The routine `imNewOffset` creates an image with an origin which is not at `(0, 0)`. Its definition is

```
extern void * imNewOffset(ImageType tag, unsigned width, unsigned height,
                          int xbase, int ybase);
```

This creates an image of the type associated with `tag`, which is `width` pixels wide by `height` pixels high, and for which the top left pixel has coordinates `(xbase, ybase)` (i.e. `imRef(im, xbase, ybase)` will refer to the top left pixel). In other ways, it acts like `imNew`: it returns `(void *)NULL` on failure, and the newly created image on success. In fact, `imNew(tag, width, height)` is exactly equivalent to `imNewOffset(tag, width, height, 0, 0)`. Note that all images loaded using `imLoad` or `imLoadF` have offsets of `(0, 0)`, regardless of what the offset was when they were saved.

It is possible to change the offset of an image after it has been created:

```
extern void imSetOffset(Image im, int newxbase, int newybase);
```

This alters the image so that its top left pixel has the coordinates `(newxbase, newybase)`. It does not change the image size or data at all.

If you are using an image with an offset origin, you will need to know what the location of the origin is, and what the extent of the image is.

- `imGetXBase(im)` returns the x-coordinate of the origin of the image `im`.
- `imGetYBase(im)` returns the y-coordinate of the origin of the image `im`.
- `imGetXMax(im)` returns the greatest valid coordinate of the image `im`. This is equal to `imGetXBase(im) + (int)imGetWidth(im) - 1`.
- `imGetYMax(im)` returns the greatest valid coordinate of the image `im`. This is equal to `imGetYBase(im) + (int)imGetHeight(im) - 1`.

Any pixel in the range `(imGetXBase(im), imGetYBase(im))` to `(imGetXMax(im), imGetYMax(im))` *inclusive* is valid; pixels whose coordinates lie outside this range are invalid. See Figure 2 for the use of these routines.

3.3 Pixel Pointers

Using `imRef` to perform some operation at every pixel in an image (as in Figure 1) works well, and reasonably efficiently. However, there are still some inefficiencies due to the fact that every `imRef` involves some arithmetic and memory references. If performance is essential, this can be a problem. I

```
GrayImage im;
int x, y;

for (y = imGetYBase(im); y <= imGetYMax(im); y++) {
    for (x = imGetXBase(im); x <= imGetXMax(im); x++) {
        imRef(im, x, y) = imRef(im, x, y) / 2;
        }
    }
```

Figure 2: A Sample Inner Loop

have therefore provided a method which preserves the abstraction barriers of the image module while giving the ability to use efficient methods of access: pixel pointers. Figure 2 shows a sample inner loop using `imRef` and Figure 3 shows it using pixel pointers.

An explanation of what is going on here:

- There are new variables, of type `GrayPixPtr`. Each one is a pointer to a pixel in some `GrayImage`.
- The variable `x` has been eliminated due to the restructured inner loop.
- The function `imGetPixPtr` is used to get pointers to some pixel in an image. The line

  ```
  pp = imGetPixPtr(im, imGetXBase(im), y);
  ```

  causes `pp` to be set to a pointer to the pixel at `(imGetXBase(im), y)` in the image `im`; this is the leftmost pixel in row `y`.
- `endpp` is set to the rightmost pixel in row `y`.
- The inner loop performs its operation on the current pixel using `imPtrRef(im, pp)`, which dereferences `pp`. `pp` is then "moved right" using `imPtrRight(im, pp)`, unless it is at the end of the row; this is determined by comparing it against `endpp` using `imPtrEq(pp, endpp)`.
- We must check that the image has a non-zero width, since if it had a zero width, the `imGetPixPtr` references would be illegal.

```
GrayImage im;
GrayPixPtr pp;
GrayPixPtr endpp;
int y;

for (y = imGetYBase(im); y <= imGetYMax(im); y++) {
    if (imGetWidth(im) > 0) {
        pp = imGetPixPtr(im, imGetXBase(im), y);
        endpp = imGetPixPtr(im, imGetXMax(im), y);
        while (1) {
            imPtrRef(im, pp) = imPtrRef(im, pp) / 2;
            if (imPtrEq(im, pp, endpp)) {
                break;
                }
            imPtrRight(im, pp);
            }
        }
    }
```

Figure 3: A Sample Inner Loop, Using Pixel Pointers

The code generated by the compiler for this rewritten inner loop is more efficient than the code generated for the original loop. However, the source code is considerably more complex and less readable; this is the usual trade-off. Unless you are using this module in an application where speed is of primary importance, you should not use pixel pointers.

The types provided are `BinaryPixPtr`, `GrayPixPtr`, `ShortPixPtr`, `LongPixPtr`, `FloatPixPtr`, `DoublePixPtr`, `RGBPixPtr` and `PtrPixPtr`; each is a pointer to the associated type of pixel. The operations defined on these are:

`pp = imGetPixPtr(im, x, y)` Set `pp` to be a pointer to pixel `(x, y)` in image `im`. The coordinates must be valid for that image.

`imPtrRef(im, pp)` This refers to the pixel in `im` at which `pp` currently points. This is used just like `imRef`: it can be used as a value, or as the target of an assignment.

`pp2 = imPtrDup(im, pp1)` This sets `pp2` to be a copy of `pp1`: a pointer to the same pixel in `im`.

`imPtrLeft(im, pp)` This changes `pp` so that it points to the pixel immediately to the left of the pixel that it used to point to (i.e. it decrements the x coordinate of the pixel pointed to).

`imPtrRight(im, pp)` This changes `pp` so that it points to the pixel immediately to the right of the pixel that it used to point to (i.e. it increments the x coordinate of the pixel pointed to).

`imPtrUp(im, pp)` This changes `pp` so that it points to the pixel immediately above the pixel in `im` that it used to point to (i.e. it decrements the y coordinate of the pixel pointed to).

`imPtrDown(im, pp)` This changes `pp` so that it points to the pixel immediately below the pixel that it used to point to (i.e. it increments the y coordinate of the pixel pointed to).

`imPtrEq(im, pp1, pp2)` This returns non-zero if `pp1` and `pp2` refer to the same pixel of `im`, zero if they do not.

The pointer-bumping functions (`imPtrLeft`, `imPtrRight`, `imPtrUp` and `imPtrDown`) must be given a pointer that refers to a pixel inside the range of the image. However, it is permitted to call these functions in such a way so as to produce a pointer which lies outside the image (for example, if `pp` referred

to the rightmost pixel in some row, then `imPtrRight(im, pp)` is still a valid operation). However, any pointer produced by such an operation *must not be used*. I only allow it to be created because allowing this simplifies programming some forms of loops.

3.4 Internal image storage

In general, you should use `imRef` or one of the pixel pointer routines described above to access image data. However, for various reasons you may be interested in mucking about with the internals of the image structure.

Images are stored in memory as a 2-D array, each element being of a certain type. This array is declared as

```
<elemtype> **data;
```

within the image structure, where `<elemtype>` is the pixel type based on the type of image you're dealing with. These are listed below.

Image type	Pixel type	PixPtr type
`GrayImage`	`unsigned char`	`GrayPixPtr`
`FloatImage`	`float`	`FloatPixPtr`
`RGBImage`	`RGB` (see below)	`RGBPixPtr`
`DoubleImage`	`double`	`DoublePixPtr`
`BinaryImage`	`char`	`BinaryPixPtr`
`LongImage`	`long`	`LongPixPtr`
`PtrImage`	`void *`	`PtrPixPtr`
`ShortImage`	`short`	`ShortPixPtr`
`HSVImage`	`HSV` (see below)	`HSVPixPtr`
`RGBFloatImage`	`RGBFloat` (see below)	`RGBFloatPixPtr`

`RGB`, `RGBFloat`, and `HSV` are defined as:

```
typedef struct {unsigned char r, g, b;} RGB;
typedef struct {float r, g, b;} RGBFloat;
typedef struct {float h, s, v;} HSV;
```

As described above, the image data is declared as a 2-D array (actually a pointer to a pointer) named `data`. Therefore,

```
imRef(theImage,x,y)
```

is equivalent to

```
(theImage)->data[y][x]
```

The image data itself is actually stored in memory as a 1-D array. That is, the image data is contiguous in memory. A 2-D pointer is used only to simplify and speed up accesses using `imRef`. You can assume that rows (and columns) of the image are stored in sequential memory addresses. Therefore, if you construct a pointer to some pixel of an image (either directly or, preferably, through the PixPtr routines above) you can easily add, subtract, multiply, etc. to this pointer to determine the locations of other pixels in the image.

The macro `imGetStore(i)` returns a pointer to the first pixel of the image `i`, and `imGetStoreLen(i)` returns the size of the data portion of the image in *pixels* (not bytes; multiply by `sizeof(elemtype)` to determine this).

3.5 Additional `adt` primatives

The basic NTadt primatives should be enough for most needs, but in addition to the basics, we've added a few more primatives for commonly done tasks. They are defined as follows:

```
extern AnyImage imBorders(AnyImage wo, int borderSize);
extern AnyImage imCrop(AnyImage im,
  int topX, int topY,
int bottomX, int bottomY);
extern AnyImage imCropInto(AnyImage into, AnyImage from,
    int top, int left,
    int bottom, int right,
    int toleft, int totop);
extern AnyImage imScale(AnyImage input,
 int newWidth, int newHeight);
extern AnyImage imDupInto(AnyImage into, AnyImage in);
```

Since checking for coordinate validity at every pixel reference can be quite costly, imBorders allows you to generate a duplicate image to the input, but with a specified border around it, allowing you to write code assuming pixel references are in bounds and avoid special cases. For more manipulation, imCrop and imCropInto are provided; imCrop generates a new image which is the original image cropped down by the input rectangle. imCropInto allows you to do the cropping of imCrop, but place the cropped bits at a

particular position in another image. imScale provides mathematical scaling of an image, returning a new image that is the original image scaled to the specified pixel size in each direction. imDupInto allows you to copy the bits of an input image into another image (assuming the two images have the same dimensions and types.)

3.6 Bugs / Comments

The NTadt library is constantly under change and alteration; should you think that you've found any bugs, or have any comments, questions, changes or proposed additions to NTadt, feel free to mail Walter Bell (`wbell@cs.cornell.edu`).