Elec4622 Lab 1, 2007 S2

David Taubman

July 30, 2007

1 Introduction

The laboratory exercises described here are designed to get you up and running with C programming for multimedia signal processing, as quickly as possible. The focus in this lab is on images, since that is the simplest step up from one dimensional signal processing. However the last exercise looks briefly at a framework which you can use for video processing tasks. In addition to programming in C/C++, the laboratory also introduces you to a software toolkit known as the "Media Interface" which provides a lot of useful functionality for manipulating images and video.

Each of the following sections corresponds to a separate laboratory exercise. There are six of these in all, which is probably more than you will have time to complete in the initial 3 hour laboratory session in Week 2. However, the lab will also be open in Week 3 for those who wish to avail themselves of the opportunity to complete all of the exercises under supervision. You may also continue your work on these laboratory exercises at home, assuming that you have installed Microsoft's Visual C++ development environment (the free "express" version is sufficient). Details are available on the course outline.

2 Creating a Project Workspace

In this first exercise, you learn how to create your own project workspace using the Visual C++ development environment.

1. To begin, you might like to unzip and open an initial sample workspace that I created for you. You will find this on the course web-site[1] under the name "lab1_sample_workspace.zip". You should put the sample workspace under the "Elec4622" directory on the lab computers, so as to avoid creating a mess for other lab users.

   (a) Use the menu items to compile (shortcut F7) and run (shortcut ctrl-F5) the program.

   (b) Use the "Project → Properties" menu item to find out where the executable file gets written to. You can find this listed in the "General" section under the "Linker" properties.

   (c) Open a DOS command prompt[2] and use the "cd" and "dir" commands to change directory to the same location in which Visual Studio is writing the executable. From their you can execute it directly.

[1] http://subjects.ee.unsw.edu.au/~elec4622

[2] Many Windows systems list this program under the "Accessories" list. You might like to create a shortcut to it on the desktop. You can also use "Start → Run → cmd.exe".

2. Create your own project workspace using "File → New → Project". Select the "Win32 Console Application" template and choose a location which is within the "Elec4622" directory.

   (a) Before clicking "Finish", modify the "Application Settings" by **checking** the "Empty Project" box and **unchecking** the "Precompiled Header" box. The newly created project and workspace will have no source files. The development environment creates several files which it uses to keep track of project settings, including compilation options, debug settings and so forth.

   (b) Before doing anything with the workspace, use "Project → Properties" to change the default settings to use ASCII character strings in place of Unicode. You will find this under the "General → Character Set" item, which you should set to "Use Multi-Byte Character Set". Otherwise, you will find a lot of Unicode related warnings and errors give you grief later on.

   (c) Note that you set properties separately for the Debug and Release versions of a program. Use the "Configuration" box at the top of the "Properties" window to select the configuration for which you are setting properties – you can arrange to have the changes made in all configurations. If you do not pay attention to this, you may find later that everything falls apart when you build a release version of your program, since you have not reflected the settings from the Debug version to the Release version.

3. Create your own "hello world" program, using the same source code that you found in the initial sample workspace above.

   (a) To do this, right click on the project name in the solution navigator panel (usually located on the left) and use "Add → New Item" to add a C++ source file (under "Code"). Give it whatever name you like. Don't worry if you only know C programming; C is a subset of C++.

   (b) After entering the simple program, compile and run it as before.

   (c) Use the "Project → Properties" menu to modify the location to which the executable file is written and check that you can run your program from the command prompt as before.

   (d) Modify the program to print its name using something like:

   ```
   printf("Hello, my name is %s\n",argv[1]);
   ```

   (e) Modify the program to conditionally print the first command line argument, using something like:

   ```
   if (argc > 1)
     {
       int val;
       sscanf(argv[1],"%d",&val);
       printf("Hello, my first argument is %d\n",val);
     }
   ```

   Hopefully, this sort of thing will remind you of a bit of C programming, in case you have forgotten.

   (f) Run the program from the command line, as before.

   (g) Now use "Project → Properties" to set up the command line argument you would like passed to the program during debugging. Run the program using "ctrl-F5" to check that all works according to plan.

(h) Finally, set a breakpoint and run the debugger using "F5". To set a breakpoint, all you have to do is to click in the margin on the left of the line in which you want the debugger to break – a red dot should appear. Once in the debugger, the following shortcuts should be useful:

- F5 continues running to the next breakpoint or until the program finishes.
- F10 steps one line in the program.
- F11 steps one line, but goes into a function if that line is a function call.

(i) Use "Debug → Windows → Watch" to open a watch window; you can enter the name of any variable whose value you want to watch during debugging.

shift + F5 in Visual Studio for debugging mode

3 Reading and Writing a BMP Image

In this exercise you will learn how to read image samples into memory from a file and write them back out again to a new image file. To simplify things, I have selected a very simple image file format for you to use, which is also readily recognized by standard image preview tools on most computers.

BMP (Bit MaP) files were created by Microsoft for Windows. The original BMP format was extremely simple, with a small header, followed by raw binary sample values. However, many extension codes have been defined to suit the needs of proprietary applications. Unfortunately, BMP files have two idiosynchracies which separate them from most other image file formats. Firstly, the sample values in BMP files appear in bottom-up rather than top-down fashion. The bottom row of the image appears first in the file, followed by the second row, and so forth. Secondly, full colour image pixels are stored in BGR (blue, then green, then red) order, rather than the RGB order used by virtually all other formats. You should also be aware of the fact that each image row is padded to a whole multiple of 4 bytes.

1. To make things simpler, I have provided some structures and functions which can be used together in a coordinated fashion to read and write BMP images. The source and header files are provided for you on the class web-site, under the name "bmp_io.zip". Unzip these source files and add them to your project workspace – the one you created in the previous exercise, or a new one if you like. You can use "Add → Existing" to add the files after right clicking on the project in the solution navigator panel. Make sure you can compile the program with the new files added.

2. The "bmp_io.zip" file also contains an "example_main.cpp" file which provides a basic outline of the few steps required to open, read/write and close BMP files using the supplied functions. Make sure you understand how it works.

3. I have provided a couple of BMP files for you to play with, again via the class web-site. Download these into the "Elec4622" directory and check that you can read and write these files using the code so far.

4. Make a few changes to the supplied "example_main.cpp" code so that image sample values are changed in a way that you can expect to visualize. For example, you might double all the sample values, or just the green component's values.

4 Installing and Playing with the Media Interface Tools

In this exercise, you will install and practice using the "Media Interface" tools on the lab machines. Everything you need to get going is found in the "mi.zip" file, located on the class web-site. Download

this file, unzip and install it. The "Media Interface" tools contain a great deal more than you will need. In the very first instance, you should try using the "mi_viewer" utility to view images. You can just execute "mi_viewer" from the command-line, as in

```
mi_viewer pens_rgb.bmp
```

You will find that mi_viewer provides you some additional features, such as:

- zoom in/out of the image (use shortcuts "z" and "ctrl-z");
- inspect pixel values (click the mouse of the location of interest – best if zoomed well in); and
- selectively disable colour planes (use chortcuts "ctrl-r", "ctrl-g" and "ctrl-b").

In fact, "mi_viewer" is just a convenient wrapper around an underlying MI module named "view". The Media Interface is actually a framework for building media processing systems by connecting any number of modules in a dynamic network. You are not supplied with the tools for creating your own MI modules in source code here, since the purpose of this course is certainly not to learn how to use the Media Interface. However, the existing modules are sufficient to build simple pipelines to streamline some of the tasks we will need to work efficiently in this course. You can obtain a full list of the installed modules by running "mi_find *" from the command-line. Here, however, is a list of the modules that you are most likely to use in this course:

view Accepts one input stream and produces no output streams. This module displays an incoming image or video stream, as required. The view module is automatically appended to any pipeline constructed on the command-line using "mi_pipe2" if there are output streams. You can, however, explicitly include the "view" module in a pipeline and this allows you to add extra arguments (e.g., play-back rate for video).

read_file Reads one or more image files from its argument list and delivers them as an output stream to downstream modules. If there are multiple files, a video stream is effectively created. Individual file formats are handled by automatically invoking lower level modules, such as "read_file_bmp". The "read_file" module is invoked automatically to process pipline inputs which appear in pipelines constructed on the command-line using "mi_pipe2".

read_file_raw Provides you with the ability to directly ingest raw data (a file of numbers) and give it structure (dimensional information that would normally be provided by a file header).

read_file_jpeg Although this is automatically invoked by "read_file" to read JPEG files, the advantage of including this module explicitly is that it allows you to control the behaviour explicitly. For example, you can instruct it to pull apart the JPEG file without actually decompressing anything. Decompression can always be done later with the module "jpeg_decompress."

frame_repeat Allows you to construct repeating video loops, for playback purposes.

crop_n_shuffle Provides simple cropping and rotation services.

image_arith Allows you form a linear combination of multiple images.

inspect Passes an input stream through to its output while displaying whatever descriptive information is available.

null Discards its input stream – useful if you want to avoid laughing the viewer to handle unconnected module outputs.

There are two ways to combine MI modules into media processing systems.

1. The more elaborate method is to use the graphical editor "misedit". This tool allows you to create arbitrary networks of modules, connecting their streams graphically and then using the "GO" button to run the system. Actually, "misedit" creates new (scripted) modules which you can use just like any other module. If the module happens to have no unconnected inputs and no unconnected outputs, it acts like a self-contained program.

2. The simpler method is to dynamically build simple media processing pipelines using the command-line tool "mi_pipe2". This is the method you are most likely to prefer in this course. Here are some examples to get you going.

   (a) Type "mi_pipe2" by itself at the command prompt to get a usage statement.

   (b) Try reading an image using each of the following constructions:

   - `mi_pipe2 -i pens_rgb.bmp ::  view`
   - `mi_pipe2 ::  read_file -f pens_rgb.bmp ::  view`
   - `mi_pipe2 ::  read_file_bmp -f pens_rgb.bmp`
     In this case, the "view" module is automatically added to handle the unconnected output stream.
   - `mi_pipe2 -i pens_rgb.bmp :: inspect :: null`
     In this case, the "null" module prevents the "view" module from being automatically added to the pipeline, since it is a module with no outputs.

   (c) Try getting help information on individual modules by adding "-help" to the module's argument list. You will notice that the actual argument names accepted by modules tend to be longer than the ones we typically use – any unambiguous prefix is acceptable.

   (d) Try creating an animation by cropping regions from a larger image, as in:

   - `mi_pipe2 -i pens_rgb.bmp -o crop1.bmp -form bmp :: crop_n_shuffle -crop 0.2W 0.2H 0.25W 0.25H`
   - `mi_pipe2 -i pens_rgb.bmp -o crop2.bmp -form bmp :: crop_n_shuffle -crop 0.25W 0.25H 0.25W 0.25H`
   - `mi_pipe2 -i pens_rgb.bmp -o crop3.bmp -form bmp :: crop_n_shuffle -crop 0.3W 0.3H 0.25W 0.25H`
   - `mi_pipe2 -i pens_rgb.bmp -o crop4.bmp -form bmp :: crop_n_shuffle -crop 0.35W 0.35H 0.25W 0.25H`
   - `mi_pipe2 -i pens_rgb.bmp -o crop5.bmp -form bmp :: crop_n_shuffle -crop 0.4W 0.4H 0.25W 0.25H`

   and then viewing all the files as a repeating video with

- `mi_pipe2 :: read_file -f crop1.bmp crop2.bmp crop3.bmp crop4.bmp crop5.bmp :: frame_repeat :: view -play -rate 2`

-repetitions 20 write_file -file_name vid.mflex

(e) If you like, you can save any stream at all in an MFLEX file. This is the default output format selected by "mi_pipe2" if you omit a "-format" argument. Amongst other things, this allows you to put video sequences into a single file for viewing later on. Try saving the animation above into an MFLEX file "vid.mflex" and then playing it with "mi_viewer" or an "mi_pipe2" pipeline.

play vid.mflex: mi_pipe2 :: read_file -f vid.mflex :: view -play -rate 2

5 Memory Organization

In this exercise, we return to the BMP file reading and writing code of Section 3 to consider memory organization. In the "example_main.cpp" file, the image is read into a single block of memory (an array), dynamically allocated using "new". In media processing, you pretty much always allocate arrays on the heap using "new" and free them using "delete".[3] However, the way in which media data is organized in memory is up to you. Rather than keeping the colour samples in an interleaved memory buffer, in this example we are going to separate them out into separate blocks of memory for each colour plane.

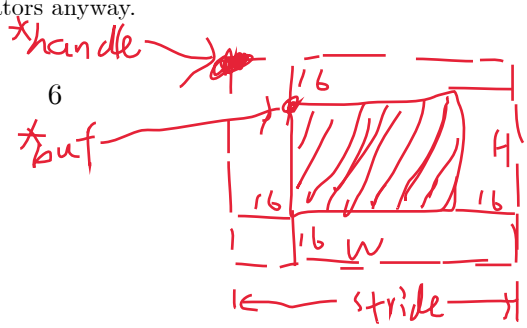
1. Start by defining a structure to handle a single image component:

```
struct io_comp {
    int width, height, stride;
    int *buf, *handle;
  };
```

Here, the `stride` value remembers the number of samples which separate consecutive image rows within the buffer; `buf` points to the top-left corner in the image component; and `handle` points to the start of the memory block obtained using the "new" operator, so we can properly delete it with "delete" when we are done. In the simplest case, we can always set `stride = height` and `buf = handle`. However, by keeping these quantities separate, we can allow space to store additional samples outside the original image boundaries, with negative indices used to address samples which lie to the left or above the original image region. To see how this is done, consider the following code snippet:

```
io_comp comps[3];
for (int c=0; c < 3; c++)
  {
    comps[c].stride = comps[c].width + 32; // 16 extra cols left & right
    int true_height = comps[c].height + 32; // 16 extra rows top & bottom
    comps[c].handle = new int[comps[c].stride*true_height];
    comps[c].buf = comps[c].handle + comps[c].stride*16 + 16;
  };
```

[3] Actually, these are the C++ heap management operators. If you prefer pure ANSI C, you can use "malloc" and "free", which are invoked by the C++ "new" and "delete" operators anyway.



The resulting `buf` pointer refers to an array which can be addressed as `buf[r*stride + c]`, where the row index `r` can range from `-16` to `height+15` and the column index `c` can range from `-16` to `width+15`. This is very useful in practice, since filtering operations invariably require access to samples beyond the boundaries of the original media source.

2. For increased generality, write a structure to hold any number of image components, as in:

```
struct io_image {
     int num_components;
     io_comp *comps;
  };
```

You can allocate the `comps` array dynamically, using something like:

```
bmp_in in;
bmp_in__open(&in,fname);
io_image image;
image.num_components = in.num_components;
image.comps = new io_comp[in.num_components];
for (c=0; c < in.num_components; c++)
  {
     io_comp *comp = image.comps + c;
     comp->width = in.cols;
     ...
```

3. Put the above structure definitions in a header file of your choosing and then write a program, following the suggestions given above, which transfers a BMP input file into the image component buffers. While you are about it, vertically flip the file so that the first line of the file which you read in is at the bottom and the last line is at the top of each image component buffer. This is the more natural organization. Your program should contain one function to read the image, creating the buffer structure and filling it in. Your program should contain another function which can write a BMP file with the contents of the image component buffers. Write a third function which adds 60 to all samples in the first image component buffer, to be called before the image is written out into a BMP file. What happens? There is plenty of food for thought here[4].

6 Processing a Sequence of Images

It is unlikely that you will make it to this point by the end of the first laboratory session; however, there is an extra session available in Week 3 for you to finish things off. In this exercise, you extend the code developed in Section 5 to handle the reading and writing of multiple images, one after the other. These

[4] At this point, there is something worth noting about our image component buffers. We are storing a full integer (32-bits) at each sample location, whereas the file stores only 8 bits per image sample. The availability of integers allows us to do processing which might cause overflow in an 8-bit representation. Before writing the data back out to a file, however, you need to convert to 8 bits, checking for the overflow condition as you do so.

will serve the role of a video stream. We can always play such image sequences back as video by using Media Interface modules, as in Section 4.

Specifically, in this exercise, you should extend the program to accept an extra command-line argument, specifying the number of image files to read. You should then form the names of the input and output files by simply appending the number to the base of the filename. Here is a code snippet to show you a way forward:

```
int num_frames=1;
    if (sscanf(argv[3],"%d",&num_frames) == 0)
      { fprintf(stderr,"Invalid frame count argument.\n"); exit(-1); }
    for (int f=0; f < num_frames; f++)
      {
        char *input_name = new char[strlen(argv[1])+20];
        char *output_name = new char[strlen(argv[2])+20];
        strcpy(input_name,argv[1]); strcpy(output_name,argv[2]);
        char *in_sep = strrchr(input_name,'.'); // Find separator
        char *in_suffix = argv[1]+(in_sep-input_name);
        char *out_sep = strrchr(output_name,'.'); // Find separator
        char *out_suffix = argv[2]+(out_sep-input_name);
        sprintf(in_sep,"%d%s",f+1,in_suffix);
        sprintf(out_sep,"%d%s",f+1,out_suffix);
        bmp_in in;
        if (bmp_in__open(&in,input_name) != 0)
          { fprintf(stderr,"Cannot open %s\n",input_name); exit(-1); }
        ...
        ...
        bmp_in__close(&in);
      };
```

Don't forget to delete buffers that you allocate within the frame loop, or else the memory demands of your program will continue to grow as you process more and more frames until the system runs out of memory. Similarly, remember to close files when you are done with them, as suggested by the above example.