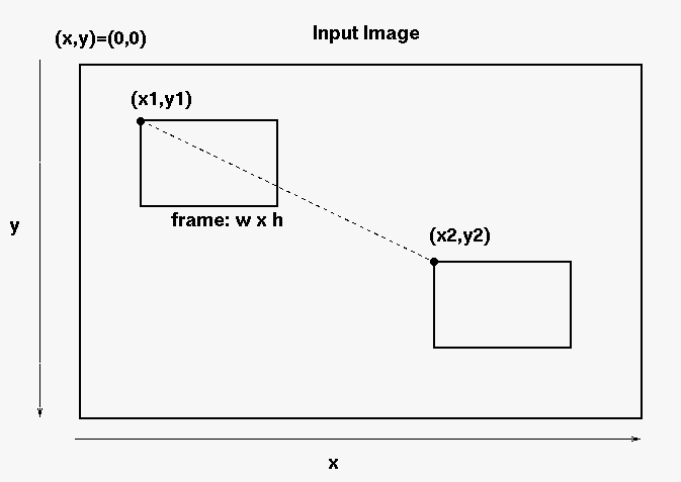
University of Cape Town
Department of Computer Science

CSC3022F
Assignment 2 - Building a 'video' from a large image

March 25, 2021

Provided with a very large input image (for example, 16K x 16K "image moscaic" from an astronomy sky survey) we want to produce a video that captures the movement of a much smaller window across this large image. You can think of this as lens that allows us to focus on a trajectory of interest. To do this, we need to position a rectangular window within this large image and extract all the pixels that overlap it to produce one frame. As we shift the window to a new offset position within the large input image, we will produce a sequence of image frames that can be written out and passed into a video creation tool such as FFMPEG to create a video. See the image below for a representation of this process.



This could be done in one step, extracting and writing directly to disk, but here we assume that we cannot access the large input image again and we want to apply a number of different image operations to the extracted frame sequence, writing out different 'video sequences' in the process. Thus, we wish to extract the frame set once and then transform them.

You will need to create a `FrameSequence` class which stores each extracted frame image as a dynamically allocated `unsigned char` array,

```
std::vector<unsigned char **> imageSequence;
```

where each element of the vector stores a pointer to a dynamically allocated 2D image frame. You must be able to access an element (pixel) in the i'th frame using notation such as `imageSequence[i][row][col]`. Your class destructor must ensure this space is cleaned up correctly.

The input image and output frames should be read/written as PGM (portable gray map) files. This is a simple format which is easy to parse and requires no libraries — See Notes.

Once frame extraction is complete, the memory for the input PGM image must be deleted — you will not refer to it again. All you need is the vector of extracted frames.

Furthermore, you must not change the extracted frame pixel data, which must be stored as is. If you need to modify pixel values (see below) then you must either be able to do this as you write them out, or you need to create a temporary image buffer to work on.

1 Core requirements

The requirements for this assignment are as follows:

1. Create a command line program, called `extractor`, to read in an image and specify the parameters for video sequence extraction/processing (command line parameters below);
2. Set up a `FrameSequence` class with a single default (non parameterized) class constructor (`FrameSequence::FrameSequence(void);`), which sets up the class variables and any necessary state, and class destructor (`FrameSequence::~FrameSequence();` — NOTE the destructor has an empty argument list, not even void!), which frees up resources when the object instance goes out of scope. In this assignment, we are not using smart pointers or RAII, so we will use 'raw' memory allocation (for each frame image) and thus require a destructor to iterate through the vector and free up all the allocated memory (with appropriate calls to `delete`). *The destructor is called automatically when the object goes out of scope.*
3. Write out a sequence of images with appropriate file numbering so they can be passed into FFMPEG.

The command line options for your program is `extractor <inputPGMfile> [options]`, where `<inputPGMfile>` is the large image you read in and `[options]` is a list of the following parameters:

```
-t <int> <int> <int> <int>  # x1 y1 x2 y2  (begin & end origin pixel coords,
                                          for frame trajectory)
-s <int> <int>              # <width>  <height>  (size of small frame in
                                                  pixels)
-w  <string> <string>       # write frames with <operation>  <name>
```

For the `-w flag`, the parameter `<operation>` = `none | invert | reverse | revinvert`, where **none**: no modification of data, **invert**: each pixel value v become 255-v, **reverse**:

reverse output (write frames out from last to first), **revinvert**: reverse and invert output. The parameter `<name>` = name of sequence base e.g. sequence-0000.pgm, sequence-0001.pgm etc will be generated with `<name>` set to 'sequence'.

NOTE: `-t` and `-s` are specified once, but you can have one or more `-w` operations specifed, each of which outputs a *different* frame sequence.

So, assuming your program was called `extractor`, an example invocation might look like this:

```
./extractor myLargeImage.pgm -t 0 10 5000 5000 -s 640 480
                              -w invert invseq -w none sequence2
```

This would read in a PGM image (myLargeImage.pgm) and define a 640×480 window, with starting origin (0,10) and final origin (5000,5000) and then extract two sequences from this. The first sequence would apply the invert operator and write the output frames to invseq-NNNN.pgm and the second would simply do the frame extraction with no image operation, and write the frames to sequence2-NNNN.pgm

Refer to the image above for the a graphical representation of the `-t` and `-s` parameters required to specify the window dimensions and start and end points (which define the path through the large image)

You must format the output string for the frame file name to have enough leading 0's to ensure you can hold all your frames (up to 4 digits should be fine e.g. someting like "%04d"). There are string format commands to manage this with C++ streams and stringstreams.

Completing the above core work will enable you to score up to 85%. To achieve a higher mark, you should tackle the mastery work outlined below.

2 Mastery work

The remaining 15% will require extending the program as follows:

`10%` — add a frame path operation [this overrides the `-t` option values]

```
-p n x1 y1 x2 y2 ... xn yn
```

that will allow you to specify coordinates that the frame top left corner will move to as it proceeds from the start to end frame coordinates. By default it just goes from start to end, in a straight line — corresponding to

```
-p 2 x1 y1 x2 y2
```

This way you can specify a *poly-line* trajectory.

5% — implement a 'deceleration/acceleration' function when you approach or leave the frame origin coordinates in your list. In other words a function that grows from 1 (pixel) to some maximum, M, when you leave a frame origin, then decreases to 1 pixel again as you arrive at a new frame origin, then starts growing back to some M as you leave again. This is a bit like the way a TV camera pans quickly across a scene and slows down as it approaches the next point of interest. Full (maximum) speed would be some M pixel step — as you slow down you would take steps which are closer and closer to 1 pixel, which means you will end jumping over some frames in each path segment. There are many functions that map a value $t \in [0, 1]$, so you'd probably need to scale your argument t by the length of the line segment, to get an input in $[0, 1]$. An example of a quadratic acceleration function is $f(t) = t(1 - t)$ (which ignores the scaling to M and can be 0, so you'd have to tweak it). You can also experiment with functions that grow from 1 to M, remain there for a while, then start smoothly decreasing to 1 again (so quadratic start to max, then constant, then quadratic decrease back to 1).

3 PGM Images

PGM images are greyscale — meaning they have no colour, and use only one value to encode an intensity that ranges from (black = 0) through to (white=255). Each value is thus stored as an 'unsigned char'. The images we will provide are 'raw' PGM i.e. binary files. However, they have a text header, so you usually open the file as binary (ios::binary), but use the `>>` and `<<` operators to read/write the header information. Getline() is very useful for reading and discarding comment lines. The header is followed by a block of bytes, which represent the image intensities. You can use the read() and write() methods to manipulate this data. Look at the function prototypes to see what arguments they expect (www.cplusplus.com can help here if you have no C++ reference). The PGM images you will receive will have the format:

```
P5
# comment line (there can be more than 1 of these comment lines)
# the string P5 will always be the first item in the file.
 Ncols  Nrows
255
binary_data_block
```

where `Nrows` and `Ncols` are integers representing the rows and columns making up the image matrix. There is a newline after each line — use "ws" to process this correctly. After reading 255 (and using the ws manipulator) you will be at the start of the binary data (Nrows*Ncols unsigned chars) which you can read in using read(). If you write out a PGM image you needn't include any comments, although it is good practice to indicate what application generated the image. You can open the output image as a binary file, write out the text header using the usual text operator `<<` and then use `write()` to output the binary data (in one statment!) before closing the file.

Have a look at `https://en.wikipedia.org/wiki/Netpbm` for more information on the image format (and it's history).

Remember that image data (being a 2D array of unsigned char) is indexed from [0][0] and that this is the top left hand corner of the image. Generally we process image data line by line, starting from the top. This is in fact how the binary image data is stored in the PGM format — so you can read/write the entire image with one read()/write() statement! This is very convenient.

We will provide some PGM images. You can view the effects of your image operations using an image viewer like Gimp on Ubuntu.

Additional Notes

1. When parsing command line parameters, remember to include

   ```
   int main(int argc, char *argv[])
   ```

   when declaring `main()`. You will then have access to argc (argument count) and argv (an array of argument char* strings) which you can then process with string processing and conversion operations. Remember to do error checking when parsing your command line (there is a BOOST library that can help, but it is not part of the C++ core and your program may not use non-standard libraries. This example is also simple enough to avoid that, and it's good practice). Note that `argc` also counts the program name itself, which is accessible as `argv[0]`.

2. You can use `pnmtopng` (Ubuntu) to convert PGM to PNG files and then use FFMPEG to generate a small video clip.

3. You always move in 1 pixels steps through the input image (unless your tackle the mastery work section and use an acceleration function). The frame origin is the top left corner of the frame. Remember that images are indexed with (0,0) being the top left screen corner, x increasing left to right and y increasing down the screen — exactly like a matrix/array (which is why we use them for images of course).

4. Since we have not covered classes in much detail, please ignore the usual requirements for move/copy constructors, alternate non-default constructor and so on. Simply ensure code like `FrameSequence myVideoFrames;` is valid and will compile. That is all you need to do here — create an instance of the class on the stack. To set class variables, use setter methods e.g. `myVideoFrames.setFrameSize(width, height);` and so on. We will deal with classes in detail later.

Please Note:

1. A working Makefile must be submitted. If the tutor cannot compile your program on nightmare.cs by typing make, you will only receive **50%** of your final mark.

2. You must use version control from the get-go. This means that there must be a .git folder alongside the code in your project folder. A **10%** penalty will apply should you fail to include a local repository in your submission.

   With regards to git usage, please note the following:

-10% - usage of git is absent. This refers to both the absence of a git repo and undeniable evidence that the student used git as a last minute attempt to avoid being penalized.

-5% - Commit messages are meaningless or lack descriptive clarity. eg: "First", "Second", "Histogram" and "fixed bug" are examples of bad commit messages. A student who is found to have violated this requirement for numerous commits will receive this penalty.

-5% - frequency of commits. Git practices advocate for frequent commits that are small in scope. Students should ideally be committing their work after a single feature has been added, removed or modified. Tutors will look at the contents of each commit to determine whether this penalty is applicable. A student who commits seemingly unrelated work in large batches on two or more occasions will receive this penalty.

Please note that all of the git related penalties are cumulative and are capped at -10% (ie: You may not receive more than -10% for git related penalties). The assignment brief has been updated to reflect this new information.

We cannot provide a definitive number of commits that determine whether or not your git usage is appropriate. It is entirely solution dependent and needs to be accessed on an individual level. All we are looking for is that a student has actually taken the time to think about what actually constitutes a feature in the context of their solution and applied git best practices accordingly.

3. You must provide a README file explaining what each file submitted does and how it fits into the program as a whole. The README file should **not** explain any theory that you have used. The README be used by the tutors if they encounter any problems.

4. Do **not** hand in any binary files. Do **not** add binaries (.o files and your executable) to your local repository.

5. Please ensure that your tarball works and is not corrupt (you can check this by trying to downloading yoru submission and extracting the contents of your tarball - make this a habit!). Corrupt or non-working tarballs will not be marked - **no exceptions.**

6. A 10% penalty per day will be incurred for all late submissions. No hand-ins will be accepted if later than 3 days.

7. **DO NOT COPY. All code submitted must be your own.** ***Copying is punishable by 0 and can cause a blotch on your academic record.*** **Scripts will be used to check that code submitted is unique.**