

ECE 1305, Fall 2015

Project 2: add illumination to Digital Elevation Map

For the second part of the project for this semester, extend your project 1 program to add illumination to a digital elevation map (DEM), creating a virtual image as viewed from a satellite looking directly down on the DEM. Write out the image as either a grayscale PGM or, for extra credit, as a PPM color image. (Alternately, add an option to your project 1 to create a virtual image from a DEM.)

Math for Project 2

The math for project 2 is detailed [in this powerpoint presentation](#), (or [PDF](#)) which is a more detailed version of what was covered in class. The slides include example calculations for one pixel in the Mt. Dartmouth elevation map. Here is [a zip file with the DEMs](#), along with sample output images (updated from project 1!).

In a nutshell: we can illuminate a surface by calculating the brightness at each point in a DEM using the dot-product of 2 vectors – the direction to the sun and the surface normal. The surface normal is simply a vector perpendicular to the surface at a point, calculated using the cross-product of two vectors created from the elevation at the point and 2 adjacent pixels. We take the dot-product of that vector with a vector pointing towards the virtual sun, and that gives us the brightness of the pixel.

Note that to get the slopes (and hence the image brightnesses) correct, you need to scale the pixels appropriately. Pixel scaling (distance from one pixel to the next) is given in the header comments in the DEM files (*.egm's). As an example, the Mt. Washington and Mt. Dartmouth images have elevations in feet, but the pixel scaling is 30 feet per pixel; so if you forget the scaling, and assume 1 foot per pixel, your mountains will be 30 times higher than they should be, and the slopes will be much steeper, making for some odd looking virtual images.

Your program does not need to extract the scaling; you can hard-code it into your program based on the filename:

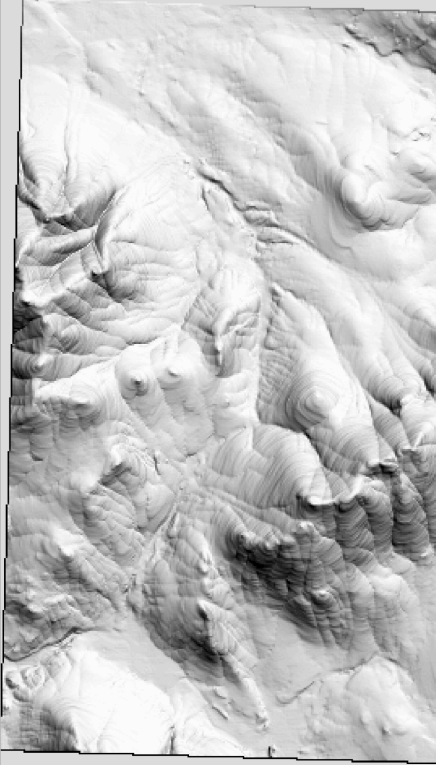
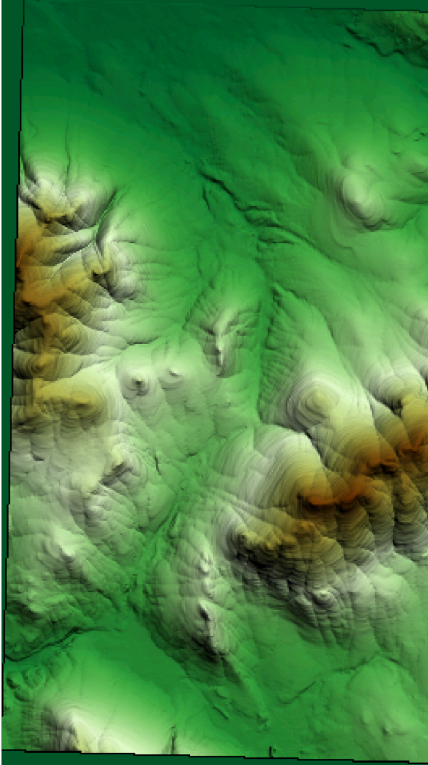
```
if (filename == "mt_dartmouth.egm")
{ ...
```

```

}
else if (filename == ...

```

By calculating the brightness for each pixel of the DEM for a chosen sun angle, we can create a grayscale PGM of the scene. By assigning colors based on elevation, we can colorize the virtual image, as shown below (but this is not a required part of the assignment). The steps to illuminate each pixel and to colorize it are given in the slides.

	
grayscale of Mt. Dartmouth	Mt. Dartmouth, colored with colormap_mountains

Your program should write the image out as either a PGM (grayscale) or PPM (color) file. The output file can be either text-based (type P2 or P3) or, for extra credit, can be binary (P5 or P6). If you wish to write binary images, I recommend you get it all working with the basic text files, then work on writing a binary version.

Examples of binary images are available in the sample_images.zip, available on the course materials page.

Output file names:

You can prompt the user for both the input and output image file names, or generate the output file automatically from the input file name. Here's how I do it:

```

        // strip extension from filename, append "illum"
and      // write out image (either grayscale or color)

string basename = strip_ext(fname);
if (!use_color)
{
    outname = basename+"_illum." + "pgm";
    writePGM( outname, nc, nr, 255, pix, cmap);
}
else
{
    outname = basename+"_illum." + "ppm";
    writePPM( outname, nc, nr, 255, redpix, grnpix,
blupix);
}
    cout << "illuminated surface saved as " << outname <<
endl;
    ...

// function to strip extension from a filename
// start at end of string fname, work backwards till you
// find the first period, and return everything up to the
// period.
//
// Inputs:
//     fname      string, containing filename
// Outputs:
//     returns a string containing everything up to the
//     last period in the filename.
//     if no periods present, returns entire original
//     filename.

string strip_ext(string fname)
{
    int i=static_cast<int>(fname.length())-1;
    while (fname[i] != '.' && i>=0)
        i--;
    return(fname.substr(0,i));
}
```

Suggestions

See the sample code above as an example of how I break the tasks down into smaller subtasks handled by separate functions, how I document my functions, and the level of comments I write in my code to properly document my code. (Note the explanation of the inputs and outputs of the `strip_ext(...)` function.)

As you can see, the code uses a function to strip the extension from a filename, and has separate functions for writing a P2 PGM or a P3 PPM image. Here are some other routines I have in my code. (These are taken from the top of my main program, where I put all my prototypes):

```
// Low-level routines
// math routines for dot & cross products,
normalizing a vector
float dot (float x1, float y1, float z1, float x2,
float y2, float z2);
void cross(float ax, float ay, float az, float bx, float
by, float bz, float &cx, float &cy, float &cz);
void normalize (float &x, float &y, float &z);
// routine to calculate the surface normal
at a point.
// h0, hx & hy are the elevations at points
(x,y), (x+1,y), and (x,y+1)
void surf_norm(float h0, float hx, float hy, float xscale,
float yscale, float &dsx, float &dsy, float &dsz);

// Medium-level routines to do individual steps needed
void segment (int nc, int nr, const float pix[], float
outpix[], int minlvl, int maxlvl, int nlvls);
void threshold(int nc, int nr, const float pix[], float
outpix[], float thresh, float lowval=0, float hival=255);
void hderiv (int nc, int nr, const float pix[], float
outpix[]);
void vderiv (int nc, int nr, const float pix[], float
outpix[]);
void add (int nc, int nr, const float pix[], float
outpix[], const float addpix[]);

// high-level routines to do contour or illuminate a
DEM
// contour uses segment, hderiv,
vderiv & edges
void contour (int nc, int nr, float pix[], float
contour[], float stepsize);
// Illuminate use surf_norm, which
uses dot, cross & normalize
```

```
void illuminate(int nc, int nr, float surf[], float img[],
float sun_az, float sun_el, float skyfraction, float
xscale, float yscale);
```

With the above, my contour(...) routine is quite simple. It is passed in a digital elevation map array and returns a PGM array with contours drawn. It calls segment(...), edges(...) and then threshold(...) and it's done.

Edges(...) merely calls hderiv(...) and vderiv(...) to find the edges.

Similarly, illuminate(...) is just a few lines, to loop through all the pixels; it calls surf_norm(...) and dot(...) to illuminate each pixel in the virtual image. (It also does a little extra work to handle the special cases of the last row and last column slightly differently, since those pixels don't have the necessary neighbors to calculate the slope.)

Each of the medium-level routines are only a few lines long, consisting of 2 nested loops to iterate through the pixel array, with a little math in the center.

The low-level routines do the basic vector math and calculate the surface normal at a point.

Excluding comments, printing some diagnostics to help debug, and some initialization and cleanup, none of my functions are more than about 10 lines long, and reasonably easy to follow.

Here are the comments from my illuminate(...) function, which pretty much walk you through the process:

```
        // calculate sun vector (x,y & z) from sun's az & el
        // iterate through all the rows
for (...
{           // and all the columns
    for (...
    {
        // get the indexes for the current pixel and
adjacent 2 pixels

        // then calculate the surface normal (using
elevation changes & pixel scaling)

        // calculate the reflected sunlight and
illumination from the sky
```

```

        // add them together to get the output pixel
    }
}
// copy last row & column from previous
for (...)
// copy last row & column from previous
for (...)

```

Requirements

Your program should be able to work with any of the .egm Digital Elevation Maps, available (along with sample output images) in [this zip file](#). (Note: this is updated from project 1's DEM zip file).

As stated above, you can either prompt the user for the scaling along with the filename, or hard-code it into your programs and select it using the filename.

The basic program should prompt the user for an input filename and whether to output a contour map, a virtual image or both. For virtual images, it should prompt for a sun location, given as azimuth and elevation angles in degrees. For contour maps, it should display the elevation range in the DEM header, then prompt for a contour stepsize.

It should then read the image, do all calculations needed, and write output image(s) with no additional input needed. (You may also prompt for pixel scaling and output file name if needed.) It should NOT require doing all steps individually to create the output, although it can have options to do this (see 2nd paragraph below).

Your program should use dynamic memory allocation to create (and delete) arrays. This wasn't needed for project 1; your project 1 portion can continue to use static arrays if you desire, though it would make sense to use dynamic arrays there as well. It should use functions, and pass parameters by value or by reference as appropriate. It should read the DEMs into float or double arrays, and produce integer PGMs or PPMs as output.

You are welcome to have additional options, such as the ability to execute individual steps to create the outputs 'manually' by doing each step individually, as my demo program shown in class does (such as, for project 1, reading, segmenting, edge detection, writing) .

Include a text file with instructions on how to create the required output. If it's not obvious to the grader, you will likely receive a low grade. Briefly describe any additional options your program can accomplish.

Grading

The goal is to pull together all that we have covered this semester.

A well-written program will break the task into separate functions for each subtask (as I have done in my solution), and be cleanly indented and well commented, and (most importantly) demonstrate to the grader and me that you have mastered the material of the course.

A well written program will make extensive use of functions to make the code readable and easily extended and maintained.

It will use dynamic memory allocation to create the necessary arrays. (It does not need to use binary input and output).

A program that does contours but not virtual images is still a major accomplishment for the semester, and will earn up to a B (if well written). A program that also does virtual images will earn up to an A. A program that can only do a few of the steps for project 1 can still earn a passing grade if it compiles and runs – it demonstrates that you've mastered some of the material for the course!

You should include a text file explaining clearly how to run your program, so the grader can quickly verify if your code produces the correct output. It should clearly state where the input files should be located, and where the output files will be written. Since it will be graded after classes finish, you won't have a chance to go back to the grader for regrading if she can't get it to run properly.

A program that goes beyond the requirements (lets user do individual steps, draws color images, draws combined contour & virtual image) will earn extra credit. I apply the extra-credit after calculating grades and drawing grade cutoffs, and it often raises a person from just below the cutoff to above it. It will be recorded on Blackboard in a separate grade-column, and after deciding on grade cutoffs, so it doesn't 'penalize' other students who just tackled the basic assignment.

Your program **MUST** compile under Visual Studio, as that is the platform the grader will be using to compile and run your program. A program that does not compile