ECE2036 **Fall Semester, 2015**

Wireless Path Loss

Assigned: October 16, 2015 *Due: October 27, 2015, 11:59pm*

Given a wireless base station with a given transmitter power P_t, we want to compute the received power at a wireless mobile device at every point in a building. The formula we will use to compute this is:

$$P_r = P_t + G_t + G_r - 10\log_{10}(\frac{4\pi f}{C}) - 20\log_{10}R - \sum_{n=0}^{n=k-1} X_n$$

Where:

P_r is the received power at the wireless mobile device (dBm)

P_t is the transmit power at the wireless base station (dBm)

G_t is the antenna gain factor for the transmitter (dBm)

G_r is the antenna gain factor for the receiver (dBm)

f is the carrier frequency for the signal (hz)

C is the speed of light (meters/sec)

R is the distance from the transmitter to the receiver (meters)

k is the number of walls in the building

X_n is the path loss through wall n (dBm).

For this assignment, use the following values:

P_t = 20dBm

G_t = 0

G_r = 0

f = 2.4GHz

$C = 3 \times 10^8$ meters/sec

X_n = 5 dBm if the line segment from the transmitter to the receiver intersects wall n, 0 otherwise.

Skeleton code. The skeleton code and other needed files are on deepthought in `nethome/ECE2-36/PathLoss`. Use `rsync` as before to copy the entire PathLoss directory to your home directory. In addition to the pathloss-skeleton.cc there is a testqt.cc that simply verifies that the QDisplay library can open a window and manipulate some pixels. You can use the testqt binaries provided, or recompile the testqt.cc program as follows:

```
make -f Makefile.testqt testqt
```

This will build an up-to-date version of testqt.

After copying the `pathloss-skeleton.cc` to `pathloss.cc` and editing `pathloss.cc` to enter your code, you can build `pathloss` by using `make`

```
make pathloss
```

Problem Description. Assume a square, single story building of 256 meters by 256 meters in size. The input file walls.txt describes the walls found in the building. Each line in the file describes one wall, specified as the starting and ending point of the wall in x and y coordinates, in units of meters. For example:

10 50 10 100

describes a vertical wall that is 50 meters long from (x=10,y=50) to (x=10,y=100). For simplicity, in this assignment all walls will be either horizontal or vertical. Note that the "final point" (10/100 in the above example) is *one*

beyond the end, meaning that you don't put a pixel at the last point. There will likely be a corner there, and the same pixel will be the starting point of another line.

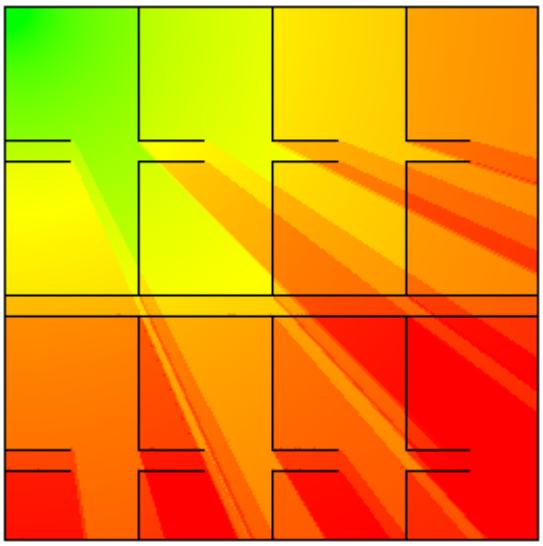
Your program is to compute the P_r at every point in the building (on a 1 meter granularity) that is NOT inside a wall, using the above formula, and display a graphical representation of the calculated strength. A green pixel would mean "good" signal, yellow means "marginal", and red means "poor".

Point and Line Classes To help with the management of the various points, create a `Point` and `Line` class. The description of the requirements for these is given in the `Point.h`, `Point.cc`, `Line.h` and `Line.cc` files. You should maintain your walls in an array of `Line` objects. There are 56 walls in the image, and you can hard-code the array size as 56 Lines. There are better ways to do this using the Standard Template Library (STL) but we have not covered this in class yet.

Program Details. Your program is to do the following using the QDisplay API that will be discussed in class.

1. Create an blank 276 pixel by 276 pixel window using the supplied `qtwindow` graphics library, and specify a color depth of 32 bits. This represents a geographic region of 256 meters by 256 meters, with a 10 pixel border on all sides that is not part of the region. This means that each pixel is equivalent to 1 meter of distance.
2. The `ImageData` function returns a pointer to an array of values of type `QRgb`, which represents a 32 bit color value with red, green, and blue components. There is an example below that shows how to both define the pixel array and how to create a pixel of the desired RGB color.
3. Read the walls.txt file and draw the walls on the window using *black pixels*. Add each wall to the array of walls for later finding wall intersections. Assume each pixel represents one square meter. Of course this means that we have walls that are 1 meter thick!
4. Assume the transmitter is at location (1,1) (NOT (0,0) which is inside a wall!)..
5. For every square meter (pixel) in the region that is not inside a wall, calculate the received signal power per the above formula.
6. Color the pixel on a scale from completely green (for P_r = -20dBm) down to completely red (for P_r = -80dBm), with solid yellow at the value of -50dBm. Use linear interpolation for values in between these specified ones. For any calculated P_r less than -80dBm, use solid red, and for any calculated P_r more than -20dBm, use solid green. Set the appropriate pixel color using the QRgb class provided.
7. For the grading script, after you have calcuated the signal strength and pixel color for each point, call the `PrintPixel` function provided in the given skeleton. The arguments are:
   (a) Pixel index - the integer index into the one-d pixel array from the image.
   (b) dB - the computed signal strength in dB
   (c) wallCount - the number of wall intersected by the vector from the (1,1) point (the base station) to the current point.
   (d) red - the computed red component of the pixel color
   (e) green - the computed green component of the pixel color
   (f) blue = the computed blue component of the pixel color.

The expected results are shown below.



Program Design Hints. The only tricky part is finding how many walls a given line segment intersects. We will discuss options on this in class.

You will need to create the 32–bit RGB values to store in the pixel buffer. The easiest way to do this is to use the `QColor` helper class, as follows (for example):

```
QRgb* p = (QRgb*)d.ImageData();
p[i] = QColor(255, 0, 0).rgb();
```

The above example uses the QColor constructor that accepts three 8–bit values representing the magnitude of the red, green, and blue components respectively. The QColor object has an `rgb()` method to convert the information to a 32–bit pixel representation. This examples sets the i'th pixel to bright red.

Resources. All skeleton files are in the usual place, on `deepthought` in the `PathLoss` subdirectory.

Turning in your Project. Run the usual `turnin-ece2036b` or `turnin-ece2036a` script as always.