24-783 Problem Set 3

Due: Monday 2/20 noon. (Two-week assignment)

(*) In the instruction (and in all of the course materials), substitute your Andrew ID for when you see a keyword *yourAndrewId*.

In this problem set, you will practice:

- Bitmap handling
- Calculating a hash code from a bitmap image.
- Hash table

Before starting, update the public repository and course_files by typing:

```
svn update ~/24783/src/course_files
svn update ~/24783/src/public
```

I am assuming that you have the same directory structure as we used in the first assignment.

START EARLY!

It really takes two weeks!

Preparation: Set up CMake projects for bitmap and utility libraries and ps3 executable

You first create projects for the two projects for the event-driven style version of Cannon-ball game and Bouncing Ball.

1. In the command line window change directory to:

   ~/24783/src/*yourAndrewId*

1. Use "svn copy" to copy the bitmap-class and hash-table class source files explained in class to this directory. The files are in course_files. The command you type is:

   ```
   svn copy ~/24783/src/course_files/simplebitmap .
   svn copy ~/24783/src/course_files/utility .
   ```

   "simplebitmap" and "utility" are the bitmap class and the hash-table class respectively we did in class. The command "svn copy" is to copy a file or a directory within the same repository. This command automatically adds copied files/directories to the SVN's control.
2. Create a sub-directory called:

   ps3

   and then, inside ps3 create sub-directories:

   ps3_1
   ps3_2

   File/Directory names are case sensitive. Use underscore. NOT hypen.
   Also the directory structure under your SVN directory is important. The grading script expects that the directory structure under your SVN directory is:

   ps3
   - ps3_1
   - ps3_2

   utility
   simplebitmap
3. Create an empty main.cpp as:

   ```
   int main(int argc,char *argv[])
   {
         return 0;
   }
   ```

   in ps3_1 sub-directory.
4. Copy the application template file:

   ~/24783/src/public/src/fslazywindow/template/main.cpp

   to ps3_2 directory. You can also copy the bitmap-viewer class instead if it is easier.
5. Write CMakeLists.txt for ps3_1 sub-directory. The project is a console application. Therefore DO NOT use MACOSX_BUNDLE keyword. The project name must be "ps3_1". Case sensitive and use underscore. Do not use hyphen. It must link "simplebitmap" and "utility" libraries.
6. Write CMakeLists.txt for ps3_2 sub-directory. The project is a graphical application. Therefore use MACOSX_BUNDLE keyword. The project name must be "ps3_2". Case

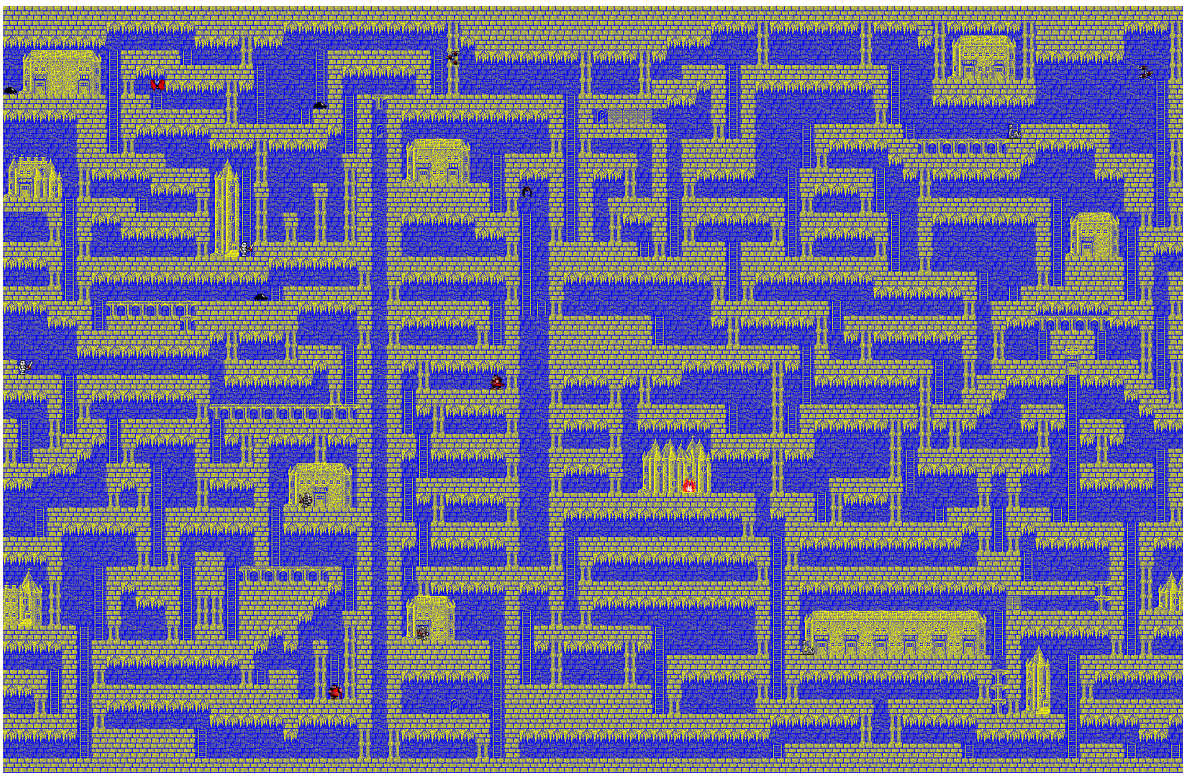
sensitive and use underscore. Do not use hyphen. It must link "fslazywindow", "simplebitmap", and "utility" libraries.

7. Modify top-level CMakeLists.txt so that your build tree includes bitmap, utility, and ps3/ps3_1 and ps3/ps3_2 sub-directories.
8. Run CMake, compile, and run ps3_1 and ps3_2 executables.
9. Add all the files you created to the control of svn. Svn-copied files are already under SVN's control, and you don't have to add them.
10. Commit to the SVN server.
11. Optional: Check out your directory in a different location and see if all the files are in the server.

PS3-1 Finish a missing function of SimpleBitmapTemplate classes

1. Write CutOut function in the SimpleBitmapTemplate class. (See comment lines in simplebitmap.h for more detailed specification.) If you write correctly, you can use CutOut function of SimpleBitmap class.
2. Write a program in main.cpp in ps3_1 sub directory.
    a. Read a .PNG bitmap specified by the first argument to the command.
    b. Cut out 40x40-pixel blocks of the input PNG and save to individual .PNG files in the current working directory. The name of the PNG files must be 0.png, 1.png, 2.png, …. Output PNGs must be 40x40 regardless of the dimension of the input PNG. When the resolution of the input bitmap is not of 40*Nx40*M (N,M are integers), the last block in each row and column will have some transparent pixels. Stop writing after writing 200 PNGs or entire input PNG is covered, whichever happens first.
    c. If the user does not provide the first argument, print a usage information as:
    `Usage: ps3_1 <pngFileName.png>`
    d. If the program cannot read the .PNG image, print an error message as:
    `Error: Failed to read a .PNG file.`
    e. It is ok to print some additional information if you need for debugging.
3. Compile and run the program. You can use PNG files in course_files/ps3/png for testing. Since it takes a .PNG file as a command-line argument, it should be easier to run it from the command line.

PS3-2 How many kinds of blocks in this map?



This map is from a retro PC game called Xanadu by Nihon Falcom Inc., which made a huge hit in 1986 in Japan. It sold in total 400K copies. Consider the number of PCs available in that day. You can imagine the magnitude of the popularity of this game. You can find images of the maps in course_files/ps3/png.

In actual game, only 9x9 blocks are visible, but this map was generated by stitching screenshots together. I ended up wasting time during this winter break playing this game and writing a program for stitching the screenshot simultaneously.

The question here is how many kinds of blocks this map is made of, and that is the goal of your program.

Your program in ps3-2 directory must take a PNG file name as input, and read it into SimpleBitmap data structure in Initialize event-handler function.

Then, your program must assign an ID number (an integer value) for each type of a 40x40 block. Cut out 40x40 block, if an ID number is not given to the block pattern, assign a number, and then add the block-number pair to the hash table. The ID number is what you find in the hash table. The 40x40 bitmap is a hash key.

The first block type should be given 0, and whenever your program finds a new type of a block increment the ID number.

You need to make a hash table that finds an integer value from a bitmap. You need to specialize the HashCode function

You need to calculate a hash-code from a 40x40 bitmap. Essentially you need to calculate a hash-code from an array of unsigned chars. There is no unique way of calculating a hash-code from a sequence of unsigned integers. You can come up with your own, or you may try the method described in:

http://stackoverflow.com/questions/11128078/android-compute-hash-of-a-bitmap

One thing I don't like about this approach is it will for sure let the hash code overflow during the calculation. What you get from overflow is undefined in C/C++ specification.

I rather would go with multiplying different prime number for different location in the array. Also, make it a non-4-byte cycle. Since all alphas of the map is zero, making a 4-byte cycle may make a pattern, which is not a good characteristic for a hash-code. For example, you may want to multiply:

2 to the (5*n)th byte,
3 to the (5*n+1)th byte,
5 to the (5*n+2)th byte,
7 to the (5*n+3)th byte,
11 to the (5*n+4)th byte,

and add them up.

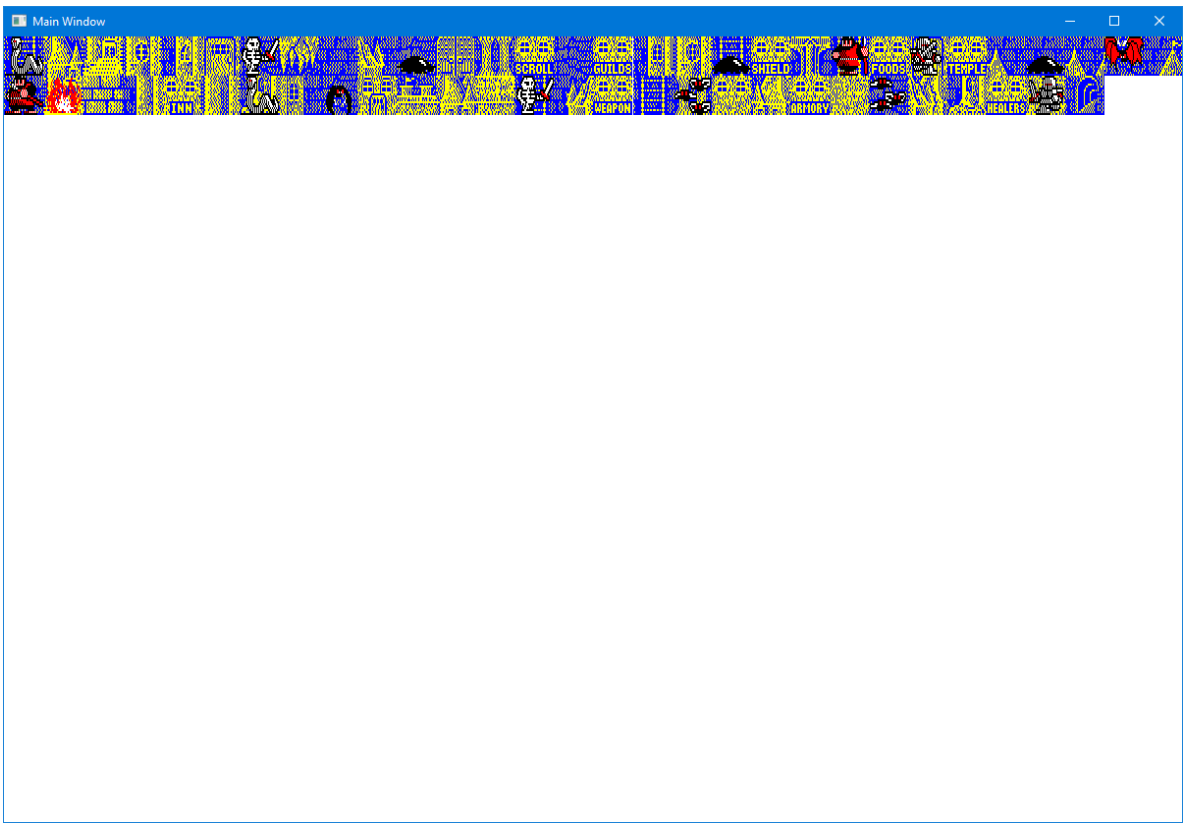
You also need to define operator== in SimpleBitmap class.

Then, in Draw function, draw blocks that your program finds on the window. The framework by default opens 1200x800 window. Draw blocks like tiles from top-left of the window. 30 blocks per row. The order can be arbitrary, but the same block must not be drawn twice.

Commit your files to the server.

Also, check out your submitted files in a different location and make sure all files are submitted, and the file contents are the ones you wanted to be graded.

If you implement correctly, you will see a window like the following. (Input=Level1.png)

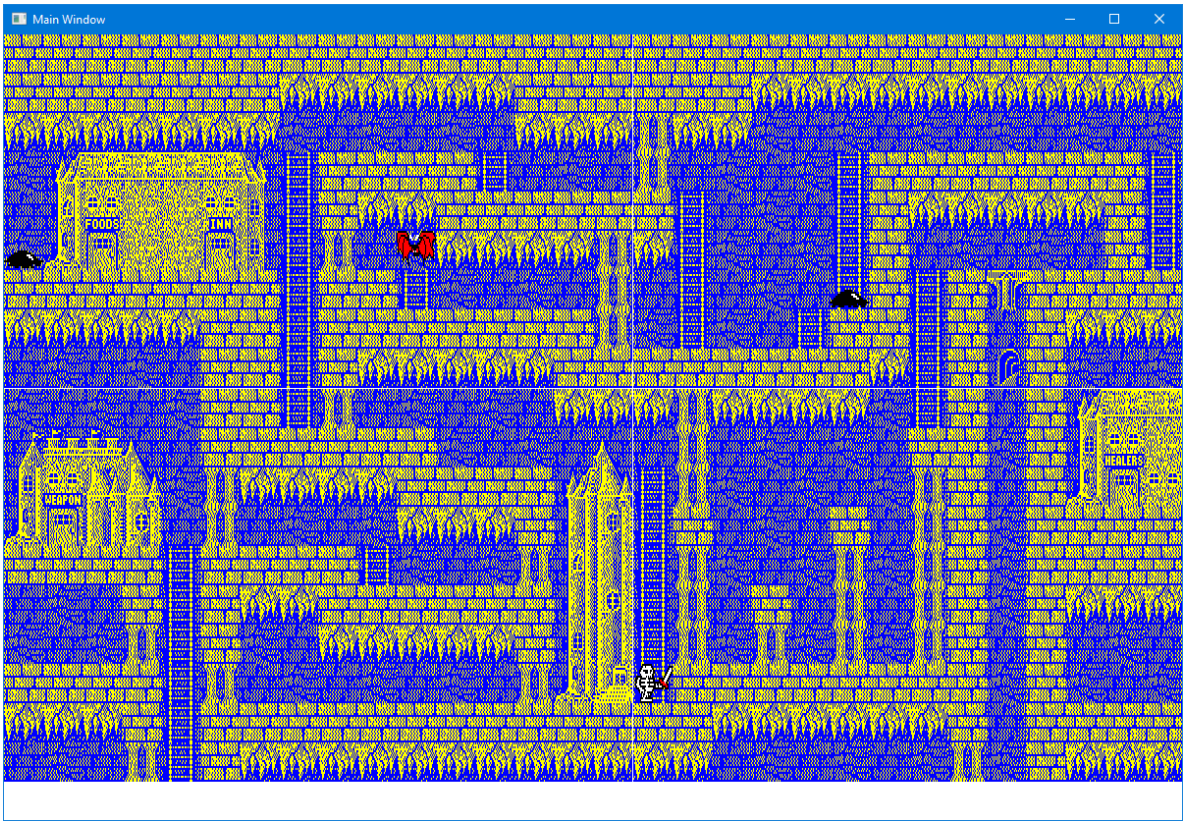


BONUS PROBLEM (+5 extra points):

After showing the tiles of the unique blocks, when the user presses the space key, draw the map on the window. However, since the map is large, it does not fit within the window. Therefore, the user must be able to scroll the map by arrow keys. (40 pixels at a time).

Since this is a bonus problem, TAs will give only limited hints for your questions.

The following is the result from Level1.png.



BONUS PROBLEM (+20 more extra points):

This requires the previous BONUS PROBLEM.

Surprisingly, the background map is using only Blue and Yellow. Other game characters use Red, White, and Black. Therefore, this game only uses five colors in total. It is an art of pixel graphics.

Now, because we know that the map consists only of Blue and Yellow, we can identify background block types where a character is on. (Every character move 40 pixels at a time, by the way).

You can compare a block that includes Red, White, or Black pixel with the matching block with only Blue and Yellow. You can calculate the similarity by comparing how many Blue and Yellow pixels match (ignoring all Red, White, and Black pixels)

After satisfying the condition in PS3-2 and the first BONUS PROBLEM, when the user presses the SPACE key for the second time, draw the map on the window, but all characters erased.

If you successfully implement it, you will get an image like the following from Level1.png.