CSci 1113, Fall 2016  
Lab Exercise 11 (Week 13): Discrete Event Simulation

It's time to put all of your C++ knowledge to use to implement a substantial program. In this lab exercise you will construct a complete simulation program that employs many of the object-oriented techniques presented in class to model and visualize a forest fire.

Forest fires that threaten property and lives have been prominent in recent news items. Computer simulations of forest fires provide information that fire fighters can use to stop the fire. In this project you will simulate the spread of a fire using a Tree class and a Forest class with the forest represented by a two-dimensional array of Tree objects. Simple rules, together with the rand() pseudo-random number generator, will be used in the simulation.

-------------

1. Adapted from Introduction to Computational Science: Modeling and Simulation for the Sciences by Angela B. Shiflet and George W. Shiflet, Princeton University Press, 2006

**Warm-up**

1) Introduction to GraphX
If you want a more detailed description of the GraphX class, see the Appendix. Download the sample.cpp from the website and compile it by running (in a terminal with all the files in the current directory):

```bash
 g++ GraphX.o -lX11 sample.cpp
```

Note: the third word is “minus ell (capitol) ex eleven”. Everything is also case sensitive, so ensure the line is exactly as above. Then run the program by typing:

**NOTE2:** GraphX.o will only work with the Unix X-window library. You might not be able to use this on your own computer.

**NOTE3:** You might need to run the following command to compile (you will have to type this once every time you start a terminal):

```bash
 module add soft/gcc/4.9.2
```

```bash
 ./a.out
```

Once you grow bored of the simple program, look at sample.cpp and see how it works. The majority of work is done by the GraphX class, which you should take a look at in GraphX.hpp. Once you look over what you have to work with, show the TAs the sample.cpp program running to be checked off.

**Stretch**

1) Orbit
Using GraphX, write a simple program that will display an "orbiting" circle animation. The computational approach is straightforward: Given an angular measurement theta (in degrees) and a fixed radius value r, construct a simple loop that will continuously increment the angular value from 0 through 360 degrees and plot a filled-circle at the x,y coordinates given by the polar coordinates (theta, r). This should continue until some mouse event takes place to terminate the loop.

Your program should perform the following GraphX operations:
• Instantiate a GraphX object
• Change the units scale of the GraphX window to 400x400 by invoking the following GraphX method:

    .scale(400,400)

• Set the plotting symbol to "solidcircle" using the .symbol method
• Set the symbolsize to 30 pixels
• Set the pencolor to some color other than black or white
• Implement a while loop that will continue running until the GraphX input is something other than "none"
• The loop should compute the x,y coordinates of the orbiting circle and plot it relative to the center of the window (e.g., x + 200, y + 200), then increase the value of the angle theta by 1 degree (mod 360) and repeat the process.
• In order to animate the orbiting circle, immediately after plotting the filled circle, set the drawing delay to .02 seconds and then .erase() the circle you just plotted. (The delay will cause the circle to show for a brief time before plotting the next circle). The effect should be the filled circle orbiting the window center.

[ Hint: the polar coordinate conversion must use the cmath functions cos and sin which require the angle in radians ]

**Workout**

Now that you are able to plot and draw using GraphX, you are ready to tackle the Forest Fire simulation. This is a discrete simulation that will use basic object-oriented techniques to model trees and forests and visualize the progression of the fire using graphical output.

Complete each of the following in order.

1) **Tree Class**

The Tree class can represent various kinds of trees, including a null tree, i.e., a location in the forest that does not have a tree. The Tree class will have the following:

Private data members:
• type (string) - the species of the tree (e.g., "Oak", "not_a_tree")
• probCatch (double) - the probability (0.0 to 1.0) of the tree catching on fire
• status (int) - the tree's status as follows:
  0 - not a tree (for a forest location without a tree)
  1 - a live tree (i.e., not burning)
  2 - a burning tree
• wetness (double) - a number between 1.0 and 100.0 to indicate wetness (default 1.0)
• burnTime (int) - the number of time steps for a tree to burn (default 1)
• xpos (int) - the x position of the tree within a GraphX window (default 0)
• ypos (int) - the y position of the tree within a GraphX window (default 0)
• symbol (string) - the symbol used to draw the tree within a GraphX window (e.g. “solidtriangle”)

Public member functions:
• A default constructor to set the default values ("not_a_tree", 1.0, 0, 1.0, 1, 0, 0, “solidtriangle”)
• A constructor with parameters to let the user initialize data members
• getProbCatch() - an accessor function for the probCatch data member
• setProbCatch() - a mutator function to set probCatch
• getStatus() - an accessor function for the status data member
• setStatus() - a mutator function to set status
• getWetness() - an accessor function for the wetness data member
• setWetness() - a mutator function to set wetness
• getBurnTime() - an accessor function for the burnTime data member
• setBurnTime() - a mutator function to set burnTime

Put the declaration part of the Tree class in file Tree.hpp and the implementation part in file Tree.cpp. Write a driver program to test the Tree class.

2) Drawing a Tree
Trees will be displayed as graphical objects on a GraphX display. To display a Tree, add a void public member function to Tree named draw. The draw method will take a GraphX object (passed by reference) as its only argument and do the following:
• Move the pen to the x,y coordinates of the tree (make sure the pen is up!)
• Set symbol size to 10 pixels
• Set symbol to the symbol specified by the tree attribute
• Get the status of the tree. If the status is:
  0: set the pen color to white
  1: set the pen color to green
  2: set the pen color to red
• plot the symbol

Test the output of your Tree by instantiating a GraphX object of size 800 by 800 in your main function and drawing default, living, and burning trees on it at different positions.

Note: GraphX will exit immediately unless you devise a way to sleep the program after displaying your trees!

3) Forest Class
A forest will be represented by a two-dimensional array of Tree objects with 33 rows and 33 columns. The Forest class will initially have the following:

Private data member:
• grid – A 33x33 array of Tree objects

Public member functions (see descriptions below):
• Forest() - default constructor
• draw(GraphX&) - calls the draw method for each Tree object in the grid

The default constructor should initialize each boundary cell of the grid array to a Tree object named "not_a_tree" with probCatch = 0, status = 0, and symbol = "solidtriangle".
The interior cells should each be initialized to a Tree object with probCatch = 0.80, status = 1, and symbol = "solidtriangle" except for the single cell in the center of the grid which should be initialized to a burning Tree object, i.e., one with status = 2. Use wetness = 1.0 and burnTime = 1 for all trees.

To initialize the position of the trees, each tree will be placed at 25 pixel intervals. For example:
• grid[0][0] should be at pixel location 0,0
• grid[0][1] should be at pixel location 0,25
• grid[1][0] should be at pixel location 25,0
• grid[32][32] should be at pixel location 800,800

Here is the default grid array:
Put the declaration part of the Forest class in file Forest.hpp and the implementation part in file Forest.cpp. Write a driver program to test the Forest class.

4) Burning Trees
Two member functions need to be added to the Forest class to cause the fire to spread from one tree to another: nextStatus() and applyNextStatus()

Note that each cell of grid is a Tree object. Use the getStatus() function of the Tree class to get the status of each cell and the setStatus() function to change it. Also, each Tree object has a probCatch value.

nextStatus ()
Function nextStatus() has int parameters i and j for the row and column indices of grid and returns the status value of cell grid[i][j] for the next time step as follows:
- If the cell is empty (status = 0) it remains empty
- If the cell has a tree (1) then it will become a burning tree (2) if one or more of its neighbors (top, bottom, left, or right) is burning AND a random number between 0.0 and 1.0 is less than the object's probCatch value; otherwise it stays a tree
- If the cell has a burning tree (2) it becomes empty (0)

applyNextStatus()
Function applyNextStatus() applies nextStatus to each interior cell of the grid to generate a new grid array representing the next time step in the simulation. The function should:
- Copy grid to a new 2D array of Tree objects named next_grid
- Use a doubly nested loop and the nextStatus function to determine the status values for the interior cells of next_grid
- Copy next_grid to grid

Note: when a member function of a class calls another member function, the dot operator is not used. For example, to call nextStatus from applyNextStatus do

```cpp
int stat;
stat = nextStatus(i,j);
```

Test the nextStatus() and applyNextStatus() methods in a calling program.

5) The Forest is Burning! (Fire Simulation)
Write a Forest method named isBurning() that returns the boolean value true if at least one cell in grid has a burning tree (i.e., status = 2) and false otherwise.

Now write a main program that will:
- Instantiate a default Forest object.
- Use a while loop that calls applyNextStatus until isBurning becomes false.
- Draw the Forest at each iteration of the loop.

Challenge

1) Build a Better Forest
You now have a forest fire simulator, but the forest has only one kind of tree and the fire always starts with one burning tree in the center. So, add a member function to the Forest class named setGrid() that has six parameters:
- i, j - the coordinates of a grid cell
- prob - used to set the value of probCatch for the Tree object in the cell
- stat - used to set the value of status for the Tree object in the cell
- wet - used to set the value of wetness for the Tree object in the cell
- burn - used to set the value of burnTime for the Tree object in the cell

For example, the following will set the Tree in location 3,6 to have probCatch = 0.95, status = 1,
wetness = 1.0, and burnTime = 1:

Forest blue;
blue.setGrid(3, 6, 0.95, 1, 1.0, 1);

After you’ve tested setGrid, write a program to instantiate a default Forest object and then use setGrid to populate the northern half of the forest with trees that burn easily (probCatch = 0.8) and the southern half with trees that do not (probCatch = 0.2).

Then set a fire somewhere and burn the forest to see what happens.

2) A Lightning Storm
A forest fire can be started by lightning. Add a member function to the Forest class called lightning to cause trees in a forest to catch fire if struck by lightning. The lightning function will have a parameter lightProb of type double, where lightProb is the probability of a tree catching on fire if struck by lightning (a value between 0.0 and 1.0).

Use the following rule to create the function: check grid location containing a tree (status = 1); it becomes a burning tree (status = 2) if a random number between 0.0 and 1.0 is less than lightProb. (Generate a new random number for each location containing a tree.)

Write the statements that would appear in a calling function to
• Instantiate a Forest object
• Invoke the lightning method to have lightning strike the forest (try probability = 0.05)

With this initial configuration, run the fire simulation to burn the forest.

Appendix:

Graphical output will be used to display the progress of the simulation as it unfolds in time. We’ve created an object-based software package called GraphX for you to use that simplifies graphical output in C++ programs. Using the GraphX class is simple; you simply instantiate a GraphX object and interact with it using simple object method calls. When you first create a GraphX object, the graphics environment is initialized and a blank window appears on the computer display. You draw lines and shapes on the window using various methods of the GraphX class.

GraphX models drawing on a blank sheet of paper using a simulated pen. The graphics pen can be in one of two states: up or down, and can be moved anywhere within the window by providing new x (horizontal) and y (vertical) pen coordinates. If the pen is up, moving it has no effect other than changing its location. If the pen is down, moving it will cause a line to be drawn from its original location to the new one. Initially, the pen location is at x = 0, y = 0, the coordinates of the lower left corner of the window.

To invoke the graphics display engine and create a visible window on your terminal, you simply declare a xnamed instance of a GraphX object, e.g.,:

GraphX mywin;
Instantiating a GraphX object initializes the graphics functions and displays an 800 x 600 pixel graphics window on the terminal. The window will remain visible and active until the GraphX object goes out of scope and is de-allocated. If the GraphX object is declared locally in your main() function or in the global name space, it will remain visible and active until the program is terminated. You can create multiple graphics windows by declaring additional GraphX objects.

You can also alter the size of the initial window by providing the width and height to the GraphX constructor:

```c
GraphX mywin(800,800);
```

The width and height are specified in pixel units. Each pixel is one "dot" on the video display. Interacting with a GraphX display is straightforward. After the object is declared, you simply invoke member functions as needed to draw lines and plot symbols. Here's an example that draws a square:

```c
mywin.clear(); //
mywin.setpos(50,50); // move the pen to x=50, y=50
mywin.pendown(); // enable drawing by putting pen down
mywin.setpos(50,150); // draw a vertical line upward
mywin.setpos(150,150); // draw a horizontal line to the right
mywin.setpos(150,50); // draw a vertical line downward
mywin.setpos(50,50); // complete the square
```

It's not really any more complicated than that! In fact, it is often easier as there are pre-built functions to draw common shapes (and you can specify the size). Here is a short list of important drawing methods:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>void .clear()</td>
<td>Clear the graphics display</td>
</tr>
<tr>
<td>void .setpos( x_location, y_location )</td>
<td>Move the pen location to the specified coordinates. x_location and y_location are integer distances from the origin at the lower left corner</td>
</tr>
<tr>
<td>void .pendown()</td>
<td>Enable line drawing</td>
</tr>
<tr>
<td>void .penup()</td>
<td>Disable line drawing</td>
</tr>
<tr>
<td>void .delay( delay_time )</td>
<td>Delay the start of the next GraphX operation. delay_time is a floating-point value specifying a fractional number of seconds (e.g., 1.25)</td>
</tr>
<tr>
<td>string .input()</td>
<td>Get mouse/keyboard event status</td>
</tr>
</tbody>
</table>

You can also display symbols at the current pen location using the .symbol() and .plot() methods. Here's an example that will move the pen and plot a green filled-circle at the pen location:

```c
mywin.penup(); // just move, don't draw!
```
mywin.setpos(300,400); // move to location x=300, y=400
mywin.symbol("solidcircle"); // we'll draw a filled-in circle
mywin.symbolsize(30); // 30 pixels across
mywin.pencolor("green"); // and green
mywin.plot(); // plot it at the pen location

And if you want to erase the symbol that was just plotted, it's easy:

mywin.erase(); // remove the symbol at the pen location

There are many GraphX methods but you will only need a few for this Lab Exercise. Here's a summary of some of the more useful pen and plotting methods:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>void .plot()</td>
<td>Plot a symbol at the pen location</td>
</tr>
<tr>
<td>void .symbol( symbol_name )</td>
<td>Specify the symbol that will be plotted when .plot() is invoked. symbol_name is a string from the following list:</td>
</tr>
<tr>
<td></td>
<td>&quot;circle&quot;</td>
</tr>
<tr>
<td></td>
<td>&quot;square&quot;</td>
</tr>
<tr>
<td></td>
<td>&quot;diamond&quot;</td>
</tr>
<tr>
<td></td>
<td>&quot;triangle&quot;</td>
</tr>
<tr>
<td></td>
<td>&quot;invtriangle&quot;</td>
</tr>
<tr>
<td></td>
<td>&quot;solidcircle&quot;</td>
</tr>
<tr>
<td></td>
<td>&quot;solidsquare&quot;</td>
</tr>
<tr>
<td></td>
<td>&quot;soliddiamond&quot;</td>
</tr>
<tr>
<td></td>
<td>&quot;solidtriangle&quot;</td>
</tr>
<tr>
<td></td>
<td>&quot;solidinvtriangle&quot;</td>
</tr>
<tr>
<td>void .symbolsize(size)</td>
<td>Set the symbol size in integer number of pixels</td>
</tr>
<tr>
<td>void .pencolor(color)</td>
<td>Specify the line and symbol color to use for drawing. color is a string from the following list:</td>
</tr>
<tr>
<td></td>
<td>&quot;black&quot;</td>
</tr>
<tr>
<td></td>
<td>&quot;white&quot;</td>
</tr>
<tr>
<td></td>
<td>&quot;red&quot;</td>
</tr>
<tr>
<td></td>
<td>&quot;yellow&quot;</td>
</tr>
<tr>
<td></td>
<td>&quot;blue&quot;</td>
</tr>
<tr>
<td></td>
<td>&quot;green&quot;</td>
</tr>
<tr>
<td></td>
<td>&quot;orange&quot;</td>
</tr>
<tr>
<td></td>
<td>&quot;purple&quot;</td>
</tr>
</tbody>
</table>

**Obtaining Input**
GraphX input is not like the >> and << operations you've been using. The stream input and output operations are blocking, that is they do not return control to your program until the user actually enters something. The GraphX input() function is non-blocking. It returns status immediately.
regardless if the user has entered anything or not. The input() member function immediately returns
one of the following event strings: "leftbutton", "rightbutton" or "keypress" corresponding to a left
mouse button press, a right mouse button press or a keyboard key press respectively. If no input has
occurred since last checking the input status, the input() function will immediately return the string
"none".

Non-blocking input requires that your program continually check (using a loop) the status of
graphics input events. For example, the following loop checks for either left or right mouse button
press events:

```cpp
bool done = false;
while(!done)
{
    string input = mywin.input();
    if (input == "rightbutton")
        done = true;
    if (input == "leftbutton")
        processButtonPress();
}
```

**Getting started**

Download the files GraphX.o and GraphX.hpp from the "Lab Information" section of the class
website. You do this by simply clicking on them. These files go directly into your downloads
(~/Downloads) folder.

Copy them into your active lab directory using the Unix cp (copy) command. To use GraphX in your
program, you must first include the header file in your source code:

```cpp
#include "GraphX.hpp"
```

The quotation characters are used in place of "< >" to indicate that the header file is located in your
current working directory.

The GraphX class implemention and the Unix graphics library must be included when you compile
your program. To do this, you must use the following compile command:

```bash
g++ GraphX.o -lX11 yourprogram.cpp yourclass.cpp
```

The third word is “minus ell ex eleven”.

NOTE: GraphX.o will only work with the Unix X-window library. You might not be able to use this
on your own computer.