Detection and Disabling of Explosive Devices

http://people.sc.fsu.edu/∼jburkardt/isc/week12
lecture_23.pdf

....

ISC3313:
Introduction to Scientific Computing with C++
Summer Semester 2011

....

John Burkardt & Detelina Stoyanova
Department of Scientific Computing
Florida State University

Last Modified: 28 July 2011

Detection and Disabling of Explosive Devices

- **Introduction**
- The Minesweeper Game
- Sample the Game
- The Look and Feel
- Game Data
- Game Functions
- Run the Emulation
- Conclusion

INTRO: Schedule

Next Class:

- Final Project Presentations.
  Tuesday, August 2 will be our last class!;

Assignments:

- Homework Program #9 is due today.
  No late work accepted after class on Tuesday, August 2!;

Evaluation:

- Go to http://campus.fsu.edu/esussai and log in.
  The evaluation is anonymous.

INTRO: Almost No Time Left For Projects!

Please remember that your project is due on Tuesday, August 2nd, at class time, 11am! This includes:

- a 5 minute oral presentation;
- a 3-5 page report to be turned in;
- a C++ program, to be turned in.

If you do not present your talk and turn in your report and programs on time, you will not receive a grade for the course.

INTRO: Implementing the Minesweeper Game in C++

Today, I'd like to show you what goes into the creation and design of a C++ program that is a little bigger and more ambitious than the examples we have worked on in class.

You may not realize it, but FireFox, Internet Explorer, iTunes, NetBeans, MATLAB, Angry Birds, Microsoft Word, and even the G++ compiler are all programs, and that means they are written in a programming language, which might be C++.

So it's important to realize that the small programs we have dealt with are stepping stones to the big programs that run the world.

We'll try to get an idea of how a larger program works by looking at the Minesweeper game, and wondering if we could write a version in C++.

- Introduction
- **The Minesweeper Game**
- Sample the Game
- The Look and Feel
- Game Data
- Game Functions
- Run the Emulation
- Conclusion

MINESWEEPER: An Outline of the Game

The game is played on a rectangular grid. A grid of 16 rows and 30 columns is common.

A number of squares in the grid are "mined", as though a land mine had been placed in that location. For the 16 x 30 game board, the number of mines might be 99.

As the game begins, the user sees only a uniform grid of gray squares. Using the mouse, the user selects a square, as though "stepping" on it. If the square is mined, the mine explodes and the game is over.

MINESWEEPER: A Sample Minefield

MINESWEEPER: Stepping on an Unmined Square

If a square is not mined, then stepping on it “clears” it.

If any of the 8 neighbors of the cleared square is a mine, then the cleared square will now display the number of neighbor mines.

But if no immediate neighbors are mines, then the program automatically clears these neighbors, and the neighbors’s neighbors, and so on, continuing until it has identified a continuous border of squares that do not contain mines but which have mine neighbors.

The user tries to use this information to choose the next step wisely.

MINESWEEPER: A Sample Minefield

MINESWEEPER: The Game Ends

At some point, the user is likely to step on a mine.

At that point, the program reveals the location of the remaining mines and stops.

The other possibility is that the user never steps on a mine, and does “clear” every safe region, in which (rare) case, the user wins.

MINESWEEPER: A Sample Minefield

- Introduction
- The Minesweeper Game
- **Sample the Game**
- The Look and Feel
- Game Data
- Game Functions
- Run the Emulation
- Conclusion

SAMPLE: A Demonstration of the Game

Believe it or not, it is now necessary, as part of our class, to take a few minutes to play minesweeper, so we are familiar with it.

Nobody installed the game on our lab machines, I guess because these machines are only for work. Of course, we happen to know that computer games are actually a huge business, so this really counts as technical training!

To begin with, let me start up a copy of the game and take a few steps to demonstrate how it works.

SAMPLE: Now You Try It!

Now please take some time to try out the game yourself, by going to a website like this:

`http://birrell.org/andrew/minesweeper/`

Play a few games, and try to convince yourself you understand what is going on.

After playing a few games, ask yourself:

"What would I need to do to make a game like this?"

Now turn off the game, please, so we can have a quiz!!!

SAMPLE: Quiz

- How many rows and columns of squares were there?
- Can you change the number of rows or columns?
- How many mines were there?
- Can you change the number of mines?
- Can you mark a square as a suspected mine?
- Can you unmark it?
- Do you know how many mines are still unmarked?
- Can you lose on the first step?

SAMPLE: More Quiz!

- What information must the program set up, update and remember?
- On each step, what must the program do?
- When is your step (choice of row and column) "legal"?
- What happens when you click in a square already cleared?
- If your step clears a square, how does the program decide what other squares to clear automatically?
- Can you undo a move?
- Can you quit early?
- Can you get help or hints or information?
- Is the puzzle always solvable by logic?
- When have you won the game?
- What is the shortest possible winning game?

- Introduction
- The Minesweeper Game
- Sample the Game
- **The Look and Feel**
- Game Data
- Game Functions
- Run the Emulation
- Conclusion

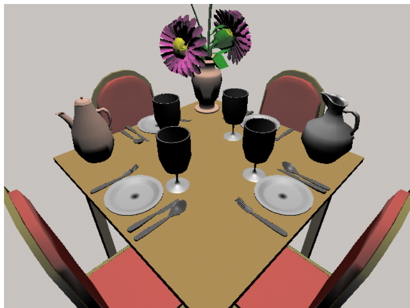
LOOK: Emulating a Program

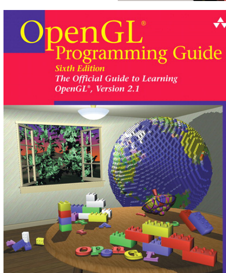
When a mathematician sees water move, and thinks up an equation of motion, the equations *simulate* the water. But when a computer person sees a program, and tries to write another program that does about the same things, we say we are trying to **emulate** the other program.

So our goal is to try to "emulate" the minesweeper game using C++, that is, to write a program which works in a similar fashion.

The first thing we think about when emulating a game is the look and feel, that is, what do we see and how do we interact?

LOOK: OpenGL





LOOK: Interactive Graphics

To make a good emulation of the game, we need interactive graphics.

An example of an interactive graphics program is the “Lights Out” game which was demonstrated in week 3. That program used the OpenGL graphics library so that a C++ program could control the game data, but also receive input from the user’s mouse, and create and update color graphic images on the screen as the game progressed.

LOOK: OpenGL Graphics for Lights Out

LOOK: Interactive Graphics? No Can Do!

Since interactive graphics were not part of this course, we will consider using a simple text graphics system instead.

We will keep the screen size small, say 8 by 8 squares.

The grid will be an 8 by 8 array of characters. A square that has not been examined will have the "-" character. A cleared square will be blank. A mine can be represented by the asterisk "*".

When the user chooses a square, we will have to "type out" the new status of the board once again.

Since we can't interact with a mouse, the user will have to type in the row and column of the square to be examined.

Our graphics are limited, but the game logic is the same!

LOOK: Text-Based Graphics

```
  1 2 3 4 5 6 7 8
1 - - - - - - - -
2 - - - - - - - -
3 - - - - - - - -
4 - - - - - - - -
5 - - - - - - - -
6 - - - - - - - -
7 - - - - - - - -
8 - - - - - - - -
Enter row and column: 3, 2
```

```
  1 2 3 4 5 6 7 8
1       1 - - - -
2       1 - - - -
3   0   1 2 - - -
4         1 - - -
5   1 2 2 2 - - -
6 1 2 - - - - - -
7 - - - - - - - -
8 - - - - - - - -
```

LOOK: Compromise on Graphics, Try Hard on Game Play

If we knew how to work with OpenGL or other interactive graphics libraries, we could develop an emulation of Minesweeper that looked as good as the real thing.

This is a first class in C++; there's even a whole semester course on computer game design that you could take to learn more about how to get graphics in your games. But we'll assume that we can still learn a lot by compromising on the graphics, and concentrating on the game play.

Detection and Disabling of Explosive Devices

- Introduction
- The Minesweeper Game
- Sample the Game
- The Look and Feel
- **Game Data**
- Game Functions
- Run the Emulation
- Conclusion

DATA: Two Tables, One Complete, One Partial

Our program data will be information that describes the status of the game.

The most obvious thing we need to keep track of is the **map**, that is, the table of mine locations.

Now the whole point of the game is that player starts out not knowing the information on the map, and gradually reveals it. One way we can make this happen is to have two copies of the map:

- **real_table[][]** records all the information;
- **vis_table[][]** records the "visible" information the user sees.

DATA: We Think of a 2D Array, But Use 1D

We will describe the two tables as 2D arrays; inside the program, we will need to squash these 2D arrays into 1D arrays. If we use an 8x8 array, then here is how the two methods of storage compare:

```
        TABLE[I][J]                    VECTOR[I*8+J]

     0    1    2    3    4    5    6    7     0  1  2  3  4  5  6  7
-+------------------------------------+------------------------+
0| 0,0  0,1  0,2  0,3  0,4  0,5  0,6  0,7  |  0  1  2  3  4  5  6  7|
1| 1,0  1,1  1,2  1,3  1,4  1,5  1,6  1,7  |  8  9 10 11 12 13 14 15|
2| 2,0  2,1  2,2  2,3  2,4  2,5  2,6  2,7  | 16 17 18 19 20 21 22 23|
3| 3,0  3,1  3,2  3,3  3,4  3,5  3,6  3,7  | 24 25 26 27 28 29 30 31|
4| 4,0  4,1  4,2  4,3  4,4  4,5  4,6  4,7  | 32 33 34 35 36 37 38 39|
5| 5,0  5,1  5,2  5,3  5,4  5,5  5,6  5,7  | 40 41 42 43 44 45 46 47|
6| 6,0  6,1  6,2  6,3  6,4  6,5  6,6  6,7  | 48 49 50 51 52 53 54 55|
7| 7,0  7,1  7,2  7,3  7,4  7,5  7,6  7,7  | 56 57 58 59 60 61 62 63|
-+------------------------------------+------------------------
```

Thus, TABLE[3][5] corresponds to VECTOR[3*8+5] = VECTOR[29]. We can store a 2D table as a 1D vector, it's ju a little painful.

DATA: The REAL_TABLE Data

The variable **real_table** will store the true map, with all the information. We can put a blank for safe squares, and a '*' for bombs. That means we want the type of **real_table** to be **char** so that it can store characters.

However, when we played the game, when we uncovered some safe squares, they didn't show up as blanks, but rather contained a number indicating the number of neighboring bombs. This is also information we could store.

So an entry of **real_table** will be

- *blank*, if the square is safe, and isolated;
- a *digit*, if the square is safe, but has mine neighbors;
- an *asterisk*, if the square is a mine.

This information can be set once when the game begins.

DATA: The VIS_TABLE Data

The variable **vis_table** stores information the user has uncovered. We initialize this array to dashes, and "uncover" entries gradually.

Moreover, the user may wish to use the array to record guesses as to the location of mines, or questionable squares. That information can be added or deleted from the map based on user input.

So an entry of **vis_table** will be

- a *dash*, if the square has not been touched;
- *blank*, if the square was touched, and is safe, and isolated;
- a *digit*, square was touched, is safe, but has mine neighbors;
- a *question mark*, if the user thinks the square is a mine;
- an *asterisk*, if the square was touched, and is a mine.

This information is updated every step and redisplayed.

DATA: Other Data

The tables are the most important items in the program.

However,some other pieces of data include:

- **n**, the number of rows and columns;
- **row** and **col**, identifying the user's chosen square;
- **pos**, converts **row** and **col** to a 1D array index;
- **mines**, the number of mines;
- **counter**, the number of safe squares uncovered.

DATA: The NEIGHBOR List

There is one more piece of data, an array called **neighbors[]**, which allows us to keep track of the neighbors of any square.

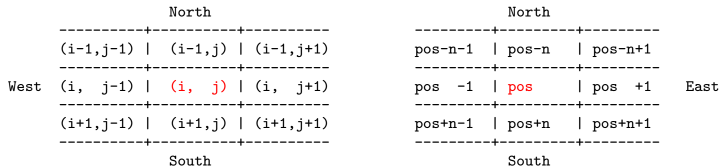
We need to know the neighbors for two reasons:

- When we are setting up the **real_table** array, we need to count the number of mines that are neighbors of each safe square. Safe squares with mine neighbors are then marked with the number of neighboring mines.
- When the user selects a square, if it is safe, we uncover that square. But we also uncover squares which are immediate neighbors and safe.

Given a square **(i,j)**, the **neighbors[]** array is computed to contain the 1D indices that would correspond to the 8 possible neighbor locations. However, we may have to discard some of these locations, because a cell in the last column has no neighbors "to the east", for instance.

DATA: The NEIGHBOR List

The NW, N, NE, W, E, SW, S, SE neighbors of (I,J) or POS:



The NEIGHBORS array tells us where we might find neighbors. Of course, some of these neighbors will not actually exist, if (I,J) or POS is in the first or last row or column.

Detection and Disabling of Explosive Devices

- Introduction
- The Minesweeper Game
- Sample the Game
- The Look and Feel
- Game Data
- **Game Functions**
- Run the Emulation
- Conclusion

FUNCTIONS: The MAIN Program

When you write a large program, it's important to break it down into pieces, that is, C++ functions.

The C++ functions should each do a separate part of the task, and the names of the functions should suggest what is going on.

Since we always start in the **main()** program, it is typical that the main program simply sets aside some memory, and then calls the underlying functions to work on the problem.

For our minesweeper program, the main program follows this design idea:

FUNCTIONS: The MAIN Program

Include statements and function declarations up here....

```
int main ( )
{
  int n = 8;
  int mines = 10;
  char real_table[n*n];
  char vis_table[n*n];
//
//  Initialize the random number generator.
//
  srand ( time ( NULL) );
//
//  Initialize both arrays with '-'.
//
  initialState ( vis_table, real_table, n );
//
//  In the real grid, replace N occurrences of '-' by '*'.
//  This is where the mines are.
//
  placeMines ( real_table, n, mines );
//
//  Count the mines adjacent to each square.
//
  calculateNumbers ( real_table, n );
//
//  Start the game.
//
  play ( real_table, vis_table, n, mines );

  return 0;
}
```

FUNCTIONS: The Initialization Functions

Before allowing the user to play the game, the program must get its data in order. It does this by calling a sequence of initialization functions, most of which do fairly obvious tasks:

1. **initialState()** sets **real_table** and **vis_table** to "-";
2. **placeMines()** sets random entries of **real_table** to "*";
3. **calculateNumbers()** counts the mine neighbors of each safe square, and puts the nonzero mine counts into **real_table**;
4. **countMines()** counts the mine neighbors for one safe square;
5. **getneighbors()** locates the neighbors of a square;
6. **isRealNeighbor()** makes sure each neighbor is inside the grid.

How would you carry out the task of **placeMines()**, which must set exactly 10 random locations to mines?

FUNCTIONS: The Play Functions

To play the game, we only need three more functions:

1. **play()** gets the move from the player and updates **vis_table**;
2. **openSafe()** clears safe squares neighboring a safe square;
3. **printTable()** displays the updated version of **vis_table**;

How would your version of **play()** determine if the player has won the game?

FUNCTIONS: The Play Function

The **play()** function handles each choice the player makes:

1. It prompts for a new row and column;
2. It deals with special user input ("Q", "M", "U");
3. If a bomb was chosen, it ends the game *(player loses)*;
4. If a safe square was chosen, it makes that square visible;
5. If the safe square has no mine neighbors, it opens up safe neighbors;
6. If all safe squares are open, it ends the game *(player wins)*;
7. It displays the updated version of **vis_table**;

FUNCTIONS: The openSafe() Function

The **openSafe()** function is actually tricky to set up.

Remember why it exists. If you step on a square, and it's not a bomb, then it may have a number on it, indicating that this square is safe, but next to at least one bomb. In that case, the program simply has to replace the "-" symbol by the number of mine neighbors.

But if the square is not next to any mines, then not only does that square become "visible", that is, the "-" is replaced by a blank, but all neighboring squares that are not mines will become visible... and their neighbors, and so on.

One way to deal with this seemingly endless problem is to use recursion. You will notice that the **openSafe()** function calls itself if it realizes a new neighbor needs to be investigated.

- Introduction
- The Minesweeper Game
- Sample the Game
- The Look and Feel
- Game Data
- Game Functions
- **Run the Emulation**
- Conclusion

EMULATION: Try the C++ Version

The C++ emulation of the minesweeper game is available on the web page or in Blackboard, as the file **minesweeper.cpp**.

As usual, if you copy the source code, you can make an executable by commands such as the following:

```
g++ minesweeper.cpp
mv a.out minesweeper
./minesweeper
```

Let's try to run the program and compare it with the version we saw earlier.

EMULATION: Normal Input, Row and Column

The program displays the current 8x8 board, and expects you to enter the coordinates of a square to investigate. Although C++ uses indices that start at 0, the program lets you specify rows and columns the "normal way", that is, as numbers between 1 and 8.

Thus, to investigate the square at row 2, column 5, after the program prompt, you type these values, with no commas:

```
Enter row and column:
2 5
```

EMULATION: Special Input, Quit, Mark, Unmark

To quit the game immediately, type a **Q**:

```
Enter row and column:
Q
```

To place a question mark at a square, use the **M** command:

```
Enter row and column:
M
Enter row and column:
2 5             <-- Places a "?" at row 2, column 5.
```

Remove a question mark with the corresponding **U** command:

```
Enter row and column:
U
Enter row and column:
2 5             <-- Restore the "-" at row 2, column 5.
```

EMULATION: We Got Pretty Far without Graphics

Our emulation of the minesweeper game is incomplete;

The most noticeable difference is the lack of a graphical interface. It makes a big difference when we have to type in the row and column of a square, instead of pointing to it with the mouse.

Smaller differences include the fact that the grid size is fixed, and that in our emulation, it's possible to "die" on the very first step.

But overall, the underlying game is really the same. And you should see that if we simply knew how to add the graphics component, we would have seen how C++ can be used to build a small but usable "product".

Detection and Disabling of Explosive Devices

- Introduction
- The Minesweeper Game
- Sample the Game
- The Look and Feel
- Game Data
- Game Functions
- Run the Emulation
- **Conclusion**

CONCLUSION: Conclusion

In this course, I have only been able to show you some of the basic features of C++ programming.

In particular, I have held back from you most of the "object-oriented" parts of C++; these features are much loved by programmers, but I think are too much for beginners to handle.

We were only able to graphics using the **gnuplot** program. I would have preferred to teach you how to use some graphics libraries that you could call directly for scientific graphics, or OpenGL for games and images.

If you go further in programming, these are two areas you shou
try to learn more about.

CONCLUSION: Future Classes?

There are still some openings in our department's class *Introduction to Game and Simulator Design*, which includes an introduction to interaction, graphics, and programming in a team.

If you want to learn more about C++, consider a second course in C++, perhaps from the Computer Science Department.

If you found yourself interested in the methods for computing approximate solutions to nonlinear equations, differential equations, and so on, you may be interested in the course *Algorithms for Science Applications I*, taught by the Department of Scientific Computing.

If you want to study the theory behind the computational algorithms, consider Numerical Analysis, perhaps as taught in the Mathematics Department.

CONCLUSION: Programming Can Take You Places

I hope that by exposing you to many programs, algorithms, and computational problems, you've got a rough road map of the possibilities there are in programming, so you can choose now where you want to go with this knowledge in the future.