Minesweeper : Redeux

Ryan Pohlman[1] and Jon Doty[2]

University Of Notre Dame, Notre Dame IN 46556, USA

Abstract. The goal of our project was not so much to solve a problem, but to meet a challenge. The challenge was to recreate a popular application that is already present on millions of computers worldwide. Any respectable Windows user is no doubt familiar with the game of Minesweeper. We set out with the intentions of recreating this game and adding new features, as well as throwing in our own personal touch.

1 Introduction

Fig. 1. Original Microsoft version of Minesweeper

Minesweeper is a timed game based on a matrix of squares. The object of the game is to reveal all of the squares on the board without tripping any of the mines. At the start of the game, a certain number of squares are randomly chosen to be mines, and you start revealing the squares one at a time by left-clicking on them. A square will either be a mine, be empty (meaning that there are no neighboring mines), or have a number in it representing how many mines are neighboring it. If you left click on a mine during the game, you "trip" it and lose. However, if you successfully reveal all of the squares around the mines, your time is displayed and you have a chance at making the high scores list.

There are also some special features in Minesweeper that assist you in winning the game. The first of these features is the flag. By right-clicking on an unrevealed square you suspect to be a mine, a flag is drawn over that square to help remind you. The flag can also be withdrawn by right-clicking on it again. Another rule is that the computer will not let you trip a mine on your first guess. This can be implemented in one of two ways. If you click on a mine on the first turn, it can be moved randomly to another square, or the mines can be generated only after your first guess has been made. Our program uses the second option.

Another special case in the game is if you click on an empty square with no neighboring mines. This is called an open field. Once a square on an open field has been clicked, a recursive function begins uncovering each adjacent square until the entire open field has been revealed, including the numbered squares on the boundary of the field. This is to save the player from frantically clicking around until they have uncovered all of the empty squares nearby.

This game makes use of several things we have learned in class including recursive algorithms, matrix and vector data structures, loading/writing information to files, and measuring elapsed time. We also went beyond what we have learned in class by implementing a graphical interface, recognizing mouse events, and general Windows programming.

2 The MineMatrix

Because Minesweeper is a game based almost entirely on matrices, we created our own MineMatrix class to handle all of the data storage and manipulation necessary in the game. The basic structure of this class was derived from CSE331 homework #2's generic two-dimensional data structure, but to make it work as a Minesweeper class many changes were necessary. For one, the number of matrix data structures within the class was changed from one to two. Also, the mathematical matrix operations from homework #2 were not needed for Minesweeper, so they were replaced with important Minesweeper functions instead. These changes are illustrated below.

2.1 Data Structures

Inside of our class are two matrices that hold the necessary data for gameplay. When the Minesweeper application is launched, an instance of the MineMatrix class is instantiated with two parameters: int r and int c. These integers represent the desired number of rows and columns of the play area, and are used to size the two matrices.

MineGrid The two-dimensional matrices used in our program are constructed using a vector of vectors. The first of these matrices is the mineGrid. The mineGrid is an rxc matrix of integers responsible for holding information regarding the location of mines in the play area. The integers in the matrix have possible values ranging from 0 to 9. Every location in the matrix holding a mine is given a value of 9. All other positions are given a value between 0 and 8, representing the number of neighboring mines in the matrix.

VisibilityGrid The second matrix stored in the MineMatrix class is the visibilityGrid, or visGrid. The visGrid is also an rxc matrix, but instead of holding information about the location of mines, it records actions of the user in the play area. In our version of Minesweeper, each square can be in one of three states that are changed by the user. All squares are initially hidden and unflagged. These matrix locations are given a value of 0. Flagged squares have a visGrid value of 1. Finally, a square that has been revealed gets a 2.

Storing data in two separate matrices is convenient for two reasons. First, it helps when programming because the data is conveniently sorted into two categories: location of the mines and visibility of the squares. Second, it separates frequently modified data from less frequently modified data. The mineGrid is only modified once in each game: when the mines are generated. However, the visGrid is modified with every user click in the play area, so it is convenient to edit a separate matrix for these actions.

2.2 Member Functions

The member functions within our MineMatrix class handle every operation needed for typical gameplay. The most important of these are listed below.

GenerateMines GenerateMines is called once during each game. It's purpose is to place mines in random locations on the blank playing field and reflect those changes in the mineGrid. To do this it first determines the number of mines to create. This is based on a function of the dimensions of the matrix (rows x columns / 6.4) rounded to the nearest whole number. Next the function randomly generates coordinates for the mines, making sure not to place two mines over each other. Lastly the function updates the mineGrid to accurately reflect the number of neighboring mines for each square and put those values in the mineGrid.

Resize This function is called whenever a new game is started. It resizes the matrix to the appropriate dimensions. In our game, the play area can be initialized to a beginner, intermediate, expert, or custom size. Custom allows the user to enter their own dimensions, from a 9x9 to a 32x24 playing area. Custom also allows you to enter a specific number of mines for your new game, between 10 and 99 mines. There is a maximum cap of the number of mines however to prevent outrageous filling of a small space with too many mines.

Flag A simple function that toggles the visGrid value of a square between a one and a zero. One represents a flag, zero means the square has not yet been revealed. This function is called whenever the user right clicks on a square.

Reveal Reveal is called every time the user left clicks on a square. It changes that location's visGrid value to a two (revealed), and also checks if the current reveal caused you to either win or lose the game. Winning or losing is flagged by changing the value of the bool variable winGame or loseGame to true.

RevealField RevealField is one of the most complicated functions in this program. It's job is to reveal an entire group of squares that have no neighbors. To do this, it receives the coordinates of the first square as two integers: row and col. It then sets visGrid[row][col] to a two (revealed). If the current square has no neighboring mines, the funtion continues to call itself recursively with the coordinates of all neighboring squares one at a time. When this has all been completed, the screen refreshes, revealing a field of blank squares surrounded by squares with numbers reflecting the number of neighboring mines.

3 Windows Programming and the Microsoft Foundation Classes

3.1 Message Handling

Programming for Windows is significantly different than creating a program to run in a UNIX console, which is what most of our previous class work had been based on. Windows programs are run and developed according to an event-driven format that differs from the usual sequential based programs. To interact with the OS and run properly, the Windows program must know how to deal with window messages that are constantly being sent between the program and the operating system itself. The programmer's job is to create functions and assign them to handle these window messages. A few examples of window messages are as follows:

WM_PAINT - a message sent when the screen is to be redrawn

WM_SIZE - message sent when the program window is resized
WM_LBUTTONDOWN - message sent to indicate that the left mouse button has been depressed
WM_LBUTTONUP - message sent to indicate that the left mouse button has been let go
WM_DESTROY - indicates the closing and destruction of the program window

There are many more, as these are just examples. The programmer attaches functions to these messages that are called whenever the specific message is received by the program. In addition, the program itself can create custom messages that it handles internally, perhaps relating to buttons or text boxes that are within the program. This system of messages and message handlers is the basis for the functionality of any Windows program.

3.2 Microsoft Foundation Classes

To ease the pain of this organization, Microsoft has created a set of code libraries called the Microsoft Foundation Classes (hereto referred to as the MFC). The MFC contains a number of pre-made classes that can be used as a base to create a window within the program. Each class has its own member variables and functions. These classes are not used directly, rather a new class is defined by the programmer that inherits from these base classes. The programmer can then add his own variables and functions that are specific to the program while using the MFC supplied functionality in the class it is derived from. The MFC classes take care of much of the overhead of the physical creation of a window and its relation to other windows within the application and with the OS. The MFC Wizard, a feature built in to Visual Studio, generates a good deal of the code needed to establish the framework of the application. The ClassWizard (shown in Fig 2), also included, is another invaluable tool that makes it easy to create objects, attach message handlers and associate variables to windows. This is a huge load off the programmer's back and frees him up to concentrate on the more creative elements of the application.

The specific class we chose for Minesweeper was the CDialog window class. It is the simplest of the classes provided in the MFC and contains only one document window and one class. It is only designed for simple I/O operations, but for the purpose of Minesweeper it does just fine. All the windows used within Minesweeper are inherited from the CDialog class. A flow chart of our class hierarchy is shown below in Fig 3.

Windows programming is from top to bottom an object-oriented experience, and this theme of data abstraction extends into the data types within the program. Char* and string data types will simply not work within a windows program, as we need data formats compatible with the MFC classes. Luckily these are provided by the MFC as well. All strings that are displayed

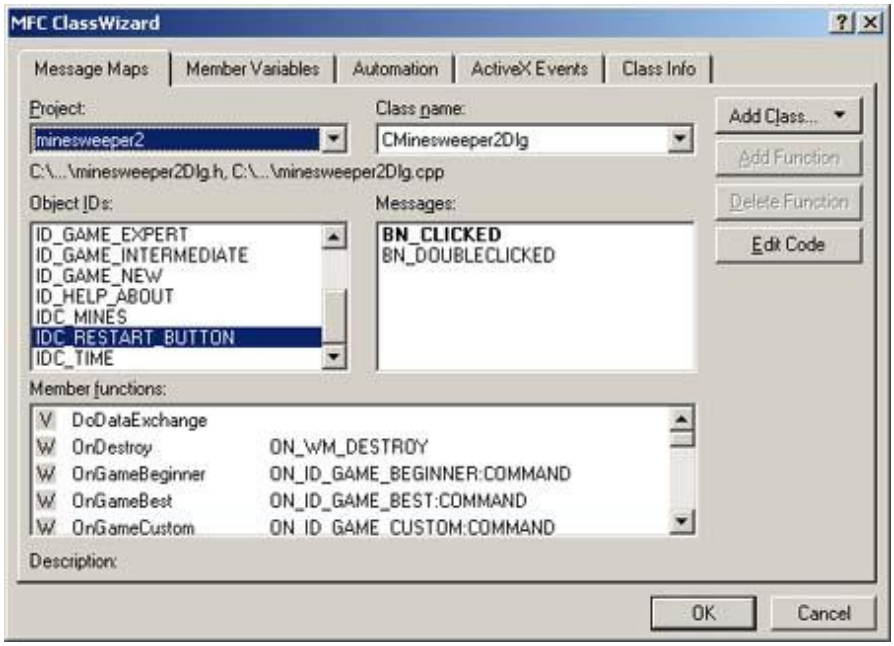

Fig. 2. The ClassWizard in action

within Minesweeper are of the CString type, which has a member function Format() which handles formatting of the string for display. File access is handles by the CFile and CArchive classes which setup file streams similar to the fstream type classes used in our assignments. To display the bitmaps in our application we had to learn a bit about memory contexts and device contexts, which are data types that specify how memory is set up and how it is manipulated. For example, to display a bitmap you must first import the bitmap into a HBITMAP variable which is then translated into an hDC memory context. This memory context is then combined with the device context that represents the display area through a the StretchBlt() function. In essence, we must setup memory to hold our picture and then import this memory into the memory representing the screen in a specific way. How this is done is specified by the arguments passed to these functions. The entire process is fairly complicated, and luckily for us we needed only a small understanding to do what we needed to do. A typical sequence to draw a bitmap would be as follows:

```
//create the paint device context
CPaintDC dc(this);
//load the bitmap into a bitmap object
```

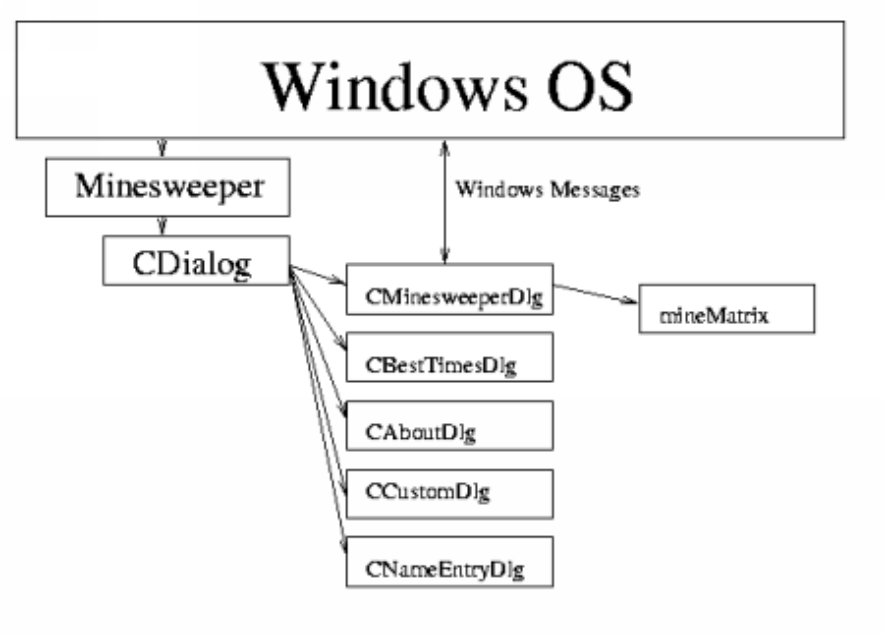

Fig. 3. Class hieracrchy and OS relationship

```
HBITMAP hbitmapMineHit =
::LoadBitmap(m_hInstance, MAKEINTRESOURCE(IDB_MINEHIT) );

//create a memory context to hold the bitmap
HDC hMemDC = ::CreateCompatibleDC(NULL);

//choose our bitmap object
SelectObject(hMemDC, hbitmapMineHit);

//add it to the display memory context
::StretchBlt(dc.m_hDC, 0 + 16*i, 50 + 16*k, 16, 16, hMemDC, 0,
   0, 16, 16, SRCCOPY);

//free up the used memory
::DeleteDC(hMemDC);
::DeleteObject(hbitmapMineHit);
```

Detailed information regarding the use of the Microsoft Foundation Classes is available on the Microsoft Developer's Network, which is included in our references.

Our end result is not entirely complete, because our time-intensive coding efforts prevented us from creating all the graphics we wanted for the program. However, we were still able to create a fully playable version of Minesweeper

with nearly all the features of the original Microsoft version included. A screen shot of our Minesweeper in action is below in Fig 4.

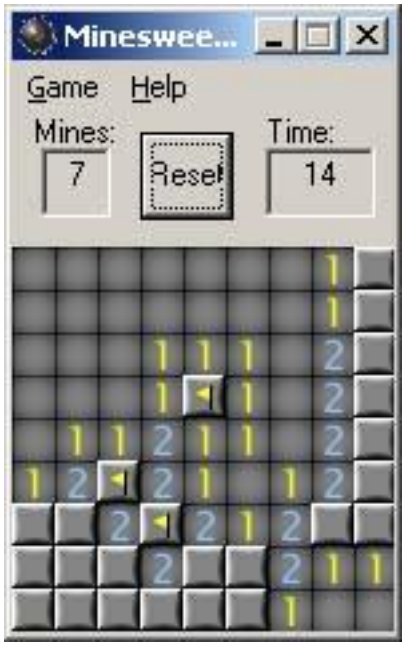

Fig. 4. Our version-to-date of Minesweeper

4 Known Bugs

1. If cancel is pressed while entering data in the Custom Game dialog, the program accepts values of zero for the width, height and mine variables. This causes a resize of the window that makes it almost invisible and forces the user to close the program. A bad-data trap needs to be implemented to avoid this situation.
2. If the game ended while the timer was still a zero, the program would continue to accept input even though the game was over. This was dealt with by forcing the timer to one immediately when the game is started. This avoids the bug but causes the timer to be offset by one second.
3. When running the game for this first time on a computer, a file access error is generated while the config file is being created. This only happens once and the program runs fine afterwards.

5 Future Considerations

in addition to addressing the bugs mentioned above, we hope to expand our program by adding the following features:

1. Expand the best times dialog to include the top 10 times for each difficulty level, rather than only the single best time.
2. Completely redo the graphics for the game, allowing for selectable graphic themes (such as fire or ice). We could also create a standard graphics format which would allow users to create their own "skins".
3. Add sounds to the game.
4. Implement network support to facilitate a multiplayer game. Players could go head-to-head and compete for the fastest time.

6 Conclusions

The end result of our efforts may seem small and simple, but we can assure you we had no shortage of roadblocks and troubles. Implementing the rules of the game was a challenge in itself, and the fact that neither of us had extensive experience programming for Windows only compounded our tribulations. Our first version of the game had to be done as a console application so that we could test the game itself while we figured out how to create the GUI in Windows. Windows programming, while done in C++, is entirely different from the programming style we have been tested in so far and took some time to get the hang of.

While not solving a practical problem like some of the other projects, we feel we learned great deal from our experiences creating Minesweeper. Knowing how to develop applications for Windows is an extremely valuable skill that should serve us nicely in the future. The fact that we successfully recreated a program that is already wide-spread is also extremely satisfying, since it is proof that we have the intellectual capacity to create marketable end-user software.

7 References

Book References

1. Cormen, Thomas H. Introduction to Algorithms. Cambridge: MIT Press, 2001.
2. Gurewich, Ori and Nathan Gurewich. Teach Yourself Visual C++ 4 in 21 Days. New York: Sam's Publishing, 1995.
3. Weiss, Mark Allen. Data Structures and Problem Solving Using C++. Reading: Addison-Wesley Publishing, 1999.

Internet References

1. Microsoft Developer Network. Microsoft. 20 November 2001 <http://msdn.microsoft.com>.

8 About the Authors

8.1 Jonathan Doty - jdoty@nd.edu

Jon is a Computer Science major at Notre Dame, originally from Chanhassen, MN. Besides writing code in the Fitzpatrick lab late at night, Jon also enjoys singing in the Glee Club, participating in the Zahm Hall Mass Choir, playing soccer, rollerblading, and staying up until 5am most nights for no apparent reason. After graduation, he hopes to get a job in the computer game industry, hopefully employing his skills in game design, sounds and graphics. But really any job would be nice.

8.2 Ryan Pohlman - mpohlman@nd.edu

Ryan is a Computer Engineering major from Gibsonia, Pennsylvania, which is a northern suburb of Pittsburgh. A resident of Sorin Hall, he is actively involved in interhall basketball, baseball and football. The rest of his time is spent in a constant effort to keep pace with the alcohol consumption habits of his friends, most of which are Business or Arts & Leisure majors. Assuming

he graduates with his sanity (and his pride), Ryan plans to attend law school and never write a line of code again.