MineDetector

Written by Matthew Hannon 2217853

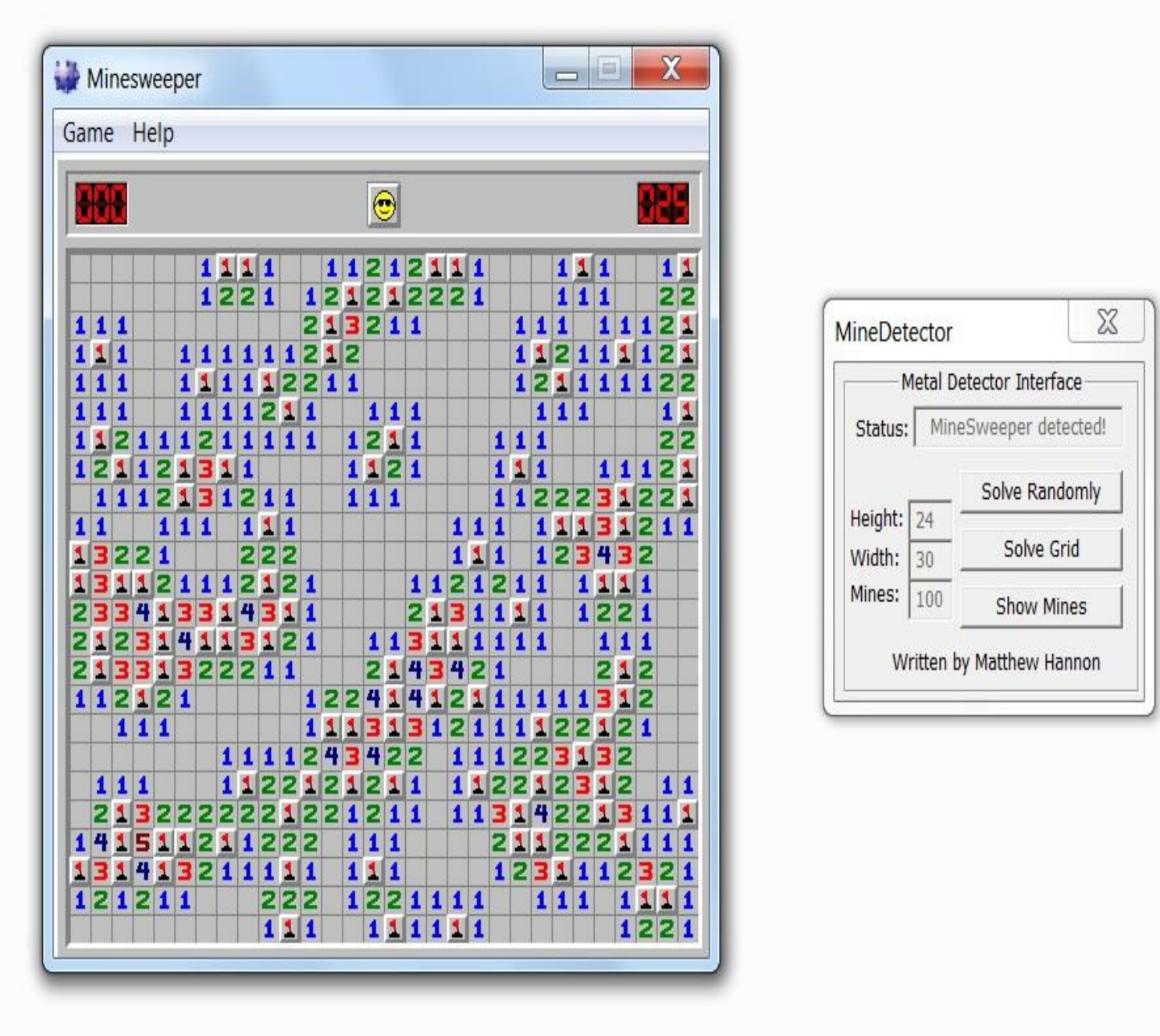

Figure 1: MineDectector beating Minesweeper

Goals

The goal of this proposal is to design software that will win at the game Minesweeper every time. Minesweeper is a popular computer game created by Microsoft. Minesweeper consists of a maximum of 24 by 24 grid of cells where each cell's contents is unknown to the player until they click it. The underlying content of the cell could be a mine which ends the game, or a number which indicates the number of mines around that particular square. The cell can also contain nothing at all. The software created that functions as a Minesweeper autosolver was later given the name MineDetector. The software generally functions by the mouse cursor being controlled by MineDetector which is able to read specific memory blocks inside of the active minesweeper process, and determines the current state of the game as well as decode one of the memory regions into a structural equivalent of the Minesweepr board layout. Another piece of information that must be found is the location of the Minesweeper window, and where inside this window does the mouse need to go to map to a specific region in memory that represents that particular square on the board. Utilizing all this information, a Minesweeper auto-solver can be built.

Major Tasks

The summarizing task that was involved was information gathering. This information was found from the examination of the Minesweeper's dependencies, active memory, and binary assembly analysis. The initial thought was to simply use memory pattern searching to identify the memory addresses for important Minesweeper variables such as the square height and width of the board, and the number of mines currently active on the board. It was determined that in order to fulfill the goal of having an equivalent board model inside of MineDetector, further analysis of the assembly code comprising Minesweepers board layout functionality had to be

investigated and semi-reverse engineered. The functionality that controlled how Minesweeper renders its board also had to be determined because information about the size and local window coordinate of a particular square is needed to fulfill the external mouse movement and clicking. Further information required was how Minesweeper referenced individual grid cells and what kind of structural information would it store to describe it. The final major task was to develop the Windows based software that would incorporate all of the information gathered, and functionally accomplish the goal of interfacing with the Windows API such that the mouse would be controlled and placed in the right position and clicked, the software would be 'smart' enough to realize if Minesweeper is even running; if it is then to open and read Minesweeper's memory as to be ready when the GUI button is clicked by the user. The software should be robust enough to respond to any changes in Minesweeper's process such as if it is suddenly closed or if the number of mines is altered by the player.

Elaboration and Implementation

The information gathering and software implementation of MineDetector will be described in the order as shown in Figure 2.

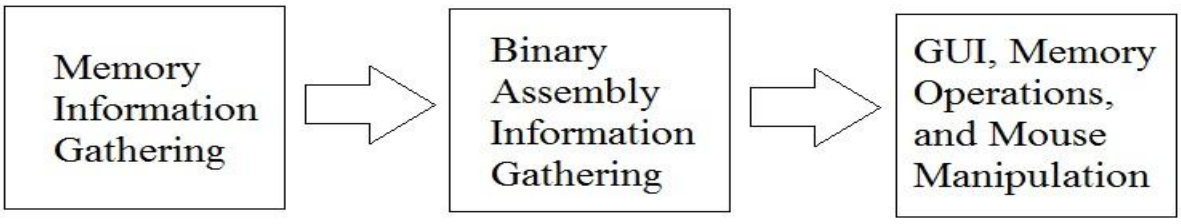

Figure 2: Flow diagram of MineDetector construction

Memory information gathering was accomplished by the use of the software called Reclass[1] which is able to display any portion of a particular processes memory in hexadecimal

notation. The feature that makes Reclass useful for this project is that it is able to detect changes in the portion of memory currently being viewed, and display them in red so they are noticed. This capability combined with Reclass's additional feature of selecting the type of variable interpretation such as a QWORD, DWORD, WORD, BYTE, as well as all the signed (negative) variations. This makes Reclass a necessity when one is attempting to decode or construct some kind of pattern to an otherwise meaningless chunk of arbitrary memory. The memory gathering phase was able to yield the memory addresses for the current height and width of the grid of cells, the total and active number of mines, and the desktop X and Y coordinate of the upper left window position. These values were all found by searching Minesweeper's memory for a specific value, then change that value, and search the previously found search results for the newly changed value. This process repeats until you isolate the search results down to a low number. The final step is to edit each one of the remaining low number of search results individually, and correlate that with an active change in the Minesweeper process. For example if I change the number of mines inside of Minesweeper's GUI, and then search for that number. I then change the number of mines again, and search the previous results for that new number of mines. This continues until one gets less than approximately five addresses. The final step is to use Reclass to manually change the value of each memory address and if it matches up with a live change in Minesweeper, you know you have found the right memory address. Note that the memory addresses are static, and not dynamically allocated with each running instance of Minesweeper. Therefore as long as the same build of Minesweeper is used with MineDetector, the memory address for say the number of mines will be valid.

The next phase is the binary assembly information gathering. The software used was a disassembler called OllyDbg[2] This stage was the toughest because it required the analysis of

large listings of x86 assembly code and the ability to decipher where in this listing I can potentially find any linkage to the assembly code that I am searching for. Many times once you think you found the right section of assembly code, one still has to rigorously examine the logic in the assembly and convert it to higher level language and interpretation to get a verification of whether or not it is the assembly listing you wanted to find. This paper will only look at a small portion of the reverse engineering of the board layout memory structure assembly. However included with the code and binaries is a document called Functions.txt which shows the functions that were at a minimum given label and maximally partially reversed.

A partial summary of how Minesweeper handles the logic that controls the grid of cells is that it allocates 0x360 (864 bytes) of static memory. The top 0x20 bytes are not used for anything besides being initialized to a 0x0F. The remaining 0x00 through 0x340 bytes are split up into 26 blocks of 0x20 bytes each. The assembly code that will be described next is the function that gives structure to the newly allocated 0x360 byte chunk of memory. The description of what I think this function does is it first fills the entire chunk of memory with 0x0F byte value. It then adds the 'edges' of the board based on the value of the width and height of the board. This happens by filling the memory range of 0x00 to 0xWidth+0x02 with the byte value 0x10 and pad the remaining memory up to the 0x20 block barrier with 0x0F, although this last step should have already been done by the initial operations of this function. The second stage of this function makes divisions with 0x10 increments as the border for the width cells in-between. Again the block is padded until the next 0x20 increment. This entire process of creating the 'edges' is repeated for the height. The third and final part of the function works backwards from the end and creates an epilogue at steps of width+2 of a value of 0x10. The reason for the plus 2 is so the board divides properly for the 0x20 block memory layout, and is also the reason

that the one can specify any combination of height and width up to the value of 24 by 24. At this value, the height cannot be increased by any amount and also keep the width the same. However the width can be increased by 1 to a value of 25 while keeping the height a value of 24. Adding to the reasoning behind the plus 2 is that this third part is working backwards and had just completed initializing the height 'edges'. Therefore to get to closest width block is 2 blocks decrement, and for this same reason there will always be room in the memory structure for more width than height blocks even though you can size it such that there are visually more height grid cells than width grid cells. The following is an assembly listing of the function that I labeled FixArrayFormat during my binary analysis phase. Please note than I initially made some comments next to some of the lines of assembly to help with figuring out what was happening.

```
FixArrayFormat proc near
        mov     eax, 360h
        loc_1002EDA:            ; tVar--;
        dec     eax
        mov     byte ptr MainBitCodes.anonymous_0[eax], 0Fh
        jnz     short loc_1002EDA ; for(all) array[x] = 0Fh
        mov     ecx, gRegWidth
        mov     edx, gRegHeight
        lea     eax, [ecx+2]    ; tVar = &(width+2)
        test    eax, eax        ; if(0)
        push    esi
        jz      short loc_1002F11
        mov     esi, edx
        shl     esi, 5
        lea     esi, startArray.anonymous_0[esi]
        loc_1002F03:            ; if(width+2)-- == 0
        dec     eax             ; redo
        mov     byte ptr MainBitCodes.anonymous_0[eax], 10h ; SetBegin of array
        mov     byte ptr [esi+eax], 10h ; End
        jnz     short loc_1002F03 ; if(width+2)-- == 0 ; redo
        loc_1002F11:
        lea     esi, [edx+2]
        test    esi, esi
        jz      short loc_1002F39
        mov     eax, esi
        shl     eax, 5
```

```
lea     edx, MainBitCodes.anonymous_0[eax]
lea     eax, [eax+ecx+1005341h] ; -2
loc_1002F2A:
sub     edx, 20h
sub     eax, 20h
dec     esi
mov     byte ptr [edx], 10h
mov     byte ptr [eax], 10h
jnz     short loc_1002F2A
pop     esi
retn
```

This previous function just created the structure for the board memory layout. In the function that called this function I labeled FixArrayFormat is the assembly which goes into randomizing and setting the mines within the layout just initialized full byte values 0x10 and 0x0f. A summary of the known byte values that specify specific actions of the Minesweeper board can be found in MineDetector.h in the code delivery.

The final major task that needs to be described is the GUI, memory operations, and proper mouse manipulation. The first thing to mention is that MineDetector when running is completely external to Minesweeper. All interactions that take place between the two processes must be routed through a third party process which can be called the window manager. This idea is shown in Figure 3.

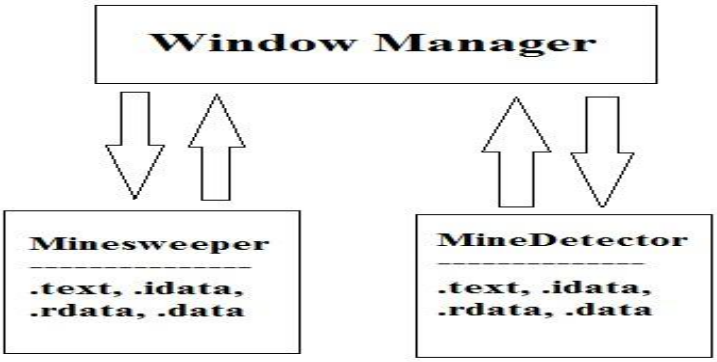

Figure 3: Process Interaction

The general organization of the operational modes MineDectector can be in is shown in Figure 4.

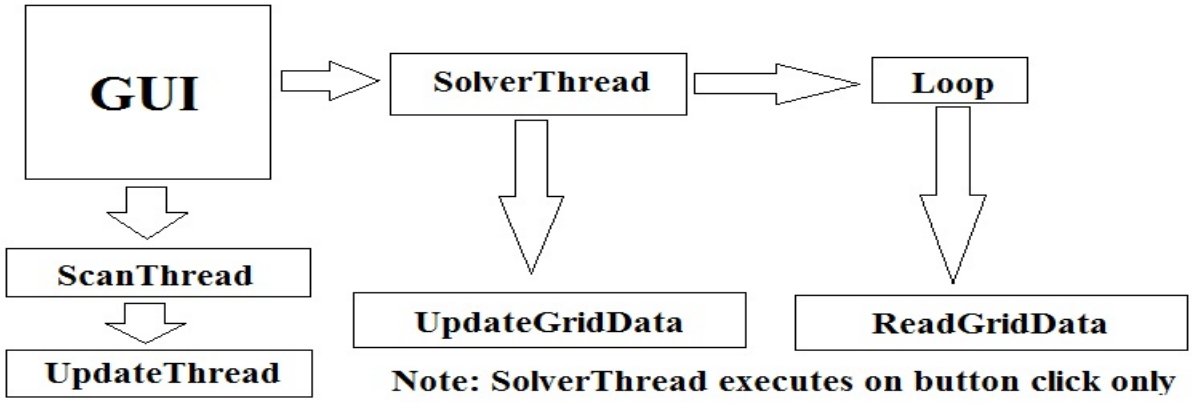

Figure 4: Process flow of thread creation and execution

The GUI handles events like a button being clicked or a thread needing to be executed after a waiting period. The ScanThread is responsible for constantly looking for the process with the name winmine.exe to be executed. When this occurs, ScanThread enables the three dialog buttons so they can be clicked. These buttons are to solve the grid randomly, solve it sequentially, or just to show all the mines on the board. The final act of ScanThread is to create UpdateThread which updates every one second and whose job is to read the height, width, and number of mines from the just created Minesweeper process. This continues until either process is closed, or one of the buttons on the dialog box are clicked.

The program flow once a button is selected is to create the SolverThread which calls UpdateGridData. UpdateGridData is responsible for reading in the entire 0x360 chunk of memory from the Minesweeper process, and a variety of other needed variables. The next thing SolverThread does is to call the locally defined function ReadGridIndex(row, col) which returns what is in the specific cell selected based on the most up to data memory reading which just occurred from calling UpdateGridData. ReadGridIndex has some error detection inside of it which tries to ensure that what it is reading is actually the board layout structure previously

explored. One of these checks is to ensure that there is a 0x10 byte value at offset 0x20. Once what is in the block is known, the program can decide to click it or not. In order to do this, a function called SelectCellviaMouse is called from SolverThread's loop which has the capability to match the row and column value to a pixel area range inside of the Minesweeper window's board row and column.

Concluding Remarks and Future Extensions.

Wrapping all that has been said about MineDetector in this document does not encompass nearly all of the functionality and logic that MineDetector implements. Many more areas could have been talked about such as the reversing of how Minesweeper draws the board and how this particular section of assembly was found through the use of the imported APIs BitBlt and SetDIBitsToDevice which Minesweeper uses to tell the operating system where and what to draw. The overall goal of winning at Minesweeper every time was accomplished with the development of MineDetector.

Future capabilities that could be added are dynamic memory address identification which would be crucial to have MineDetector work with different versions of Minesweeper. This could be accomplished by searching through all assembly instructions for a specific opcode pattern, and if this pattern is found then offset a particular number of bytes to the memory address that you are actually interested in reading. The ultimate future extension for MineDetector is to be redeveloped to work with the new and improved edition found in Windows 7.

A link for the source code and binaries can be found in the reference section.

Refernces

[1] - **ReClass aka Structbuild 2 – Username:** DrUnKeN ChEeTaH
http://www.gamedeception.net/archive/index.php?t-12041.html

[2] - **OllyDbg** -- Oleh Yuschuk http://www.ollydbg.de/