# Programmer's Documentation

# MineZweeper

Markéta Sauerová
Univerzita Karlova MFF
zimní semestr 2018/2019
Programování I.

# Contents

# 1. Specification Breakdown

## 1.1. Brief description

MineZweeper is a modified implementation of the classical game MineSweeper, which is a single-player puzzle video game, written in Pascal, using Wingraph, winmouse and wincrt units for better graphical appearance as well as handling user input.

*Original specification:* Specification-MineZweeper.pdf .

## 1.2. Functional requirements

### 1. 2. 1. Menu

1) CloseGraph request
   *At any stage, program must react to close graphical window request and properly end itself. It's meant to be a prime feature for users.*
2) Ban Console
   *Forbid the console window from appearance, so that user can focus only on graphical window.*
3) Process mouse events
   *Accurately process mouse clicking to prevent the menu from being uncontrollable.*
4) Input reaction time
   *Feedback on mouse input from menu buttons should be fast with a small delay, when creating the notion of pressing the button.*
5) CPU usage management
   *Don't use all the CPU time in main loops, checking for the input. Instead use delay function for resting.*
6) Redrawing graphics
   *Minimize graphics bugs, while redrawing or overdrawing window.*
7) Reduce flickering
   *Reduce flickering of the animated title to the minimum possible.*
8) Execute new process
   *Must be able to transfer prime event handler, close graphical window and exit the old process when executing the game, that is written as a separate program.*
9) File handling
   *File managing for passing data between individual programs.*
10) Control memory
    *Freeing allocated stuff in memory for instance animations, images. Preventing memory leaks.*

### 1. 2. 2. Game

1) CloseGraph request
   *At any stage, program must react to close graphical window request and properly end itself. It's meant to be a prime feature for users.*
2) Ban Console
   *Forbid the console window from appearance, so that user can focus only on graphical window.*
3) Loading time
   *Reduce time spends with loading and initializing to shortest possible.*
4) Difficulty settings
   *Every variable element of the game, that depends on difficulty, must be changed due to settings from Menu output.*
5) Mine distribution
   *Distribution of mines must be random and different for majority of newly started games.*
6) Follow game rules
   *Square manipulation as well as losing or winning the game should be implemented according to Minesweeper rules.*
7) Mouse accuracy
   *The accuracy of clicking on the specific square must be high, avoiding any unwanted squares to be marked.*
8) Mouse click time
   *If the mouse button was pressed the delay when processing should be shortest possible.*
9) Timer
   *The counting of the time must start exactly when the user presses any mouse button.*
10) Execute new process
    *Must be able to transfer prime event handler, close graphical window and exit the old process when executing the Menu or NicknameWindow, which are written as separate programs.*
11) File handling
    *File managing for passing data between individual programs.*
12) Control memory
    *Trying to optimize occupied memory space, freeing allocated stuff in memory. Preventing memory leaks.*
13) Redrawing graphics
    *Minimize graphics bugs, while redrawing or overdrawing window.*

# 2. Project Structure

Project is structured in two main separate programs written in Pascal using Lazarus IDE, **Game01** as the MineSweeper implementation and **Intro1** as Menu and formal entry point of the game. Individual programs are connected by executing each other and exiting their own process.

Along these, minor program **NameWindow**, was also created separately after consideration of possibilities throughout designing the project. Shortly, it serves for obtaining keyboard input from user. Detailed description is situated below.

As mentioned, programs use **wingraph**, **wincrt** and **winmouse** units, which source files are compiled together with code and located in the project folder. Used code will be explained concisely but not in detail. For further usage or interest, it can be found in wingraph unit documentation.

**Sysutils** unit (various system utilities) is also attached and widely utilized.

Programs try to comply with every functional requirement listed above. In this documentation procedures or functions, strictly fulfilling the requirements, will be **marked with number** of the specified one form the list.

For instance, both has the directive **{$APPTYPE GUI}**, which marks the application as a graphical application, so no console window will be opened when the application is run.**[2]**

This documentation is organized according to two main programs and minor third is described afterwards. Procedures or functions are divided in **high-level** and **low-level** design producing distinct blocks handling various parts of the project. In these blocks, used data structures, algorithms, input/output or necessary arguments will be represented and explained.

## 2.1. Intro1

Overall it is formal project entry point. One of two main programs. Alone, provides Menu support for users with specified functionalities. Start Game executes the Game01 process and exits current one, Difficulty handles files for passing required settings, Instructions overdraw graphical window and loads specified instructions and game rules, Highscore displays highest score reached in individual difficulties along with the gamer nickname.

### 2.1.1. Global variables

- anim: AnimatType
  *The returned handle of an animation from GetAnim procedure with a bitmap image taken from screen in the defined rectangle. Later serves for animating the title (rotation). Freeing the memory happens at the end by FreeAnim procedure.*
- i: integer

*Helps with preserving animation of the title, which is splitted in two procedures maintained in prime loop (it has to be changeable and accessible from more places). Also used as indicator when should unpressing menu button be performed.*

- colors: array [0..4] of ^longword = ( @red,@orange,@green,@Blue,@Purple)
  *Main menu colors, used for simplifying possible changes in different procedures (LoadFromFile, MenuButtons, PressButton, UnpressButton).*
- word: array [0..3] of string = ( 'Start Game','Difficulty','Instructions','Highscore')
  and word1: array [0..2] of string = ( 'Beginner','Intermediate','Expert')
  *Continuing usage of these strings (MenuButtons, PressButton, UnpressButton, Difficulty).*
- buttonPressed: Boolean
  *For handling the notion of pressing the button at first menu stage. Used in prime loop and changed by procedure ProcessMouseEvents.*
- bitmap: pointer
  *Points to allocated memory block with loaded static image (back.bmp). Used throughout the program, freed at the end or when executing Game01 process.*

### 2.1.2. Procedure Main() [1, 4, 5, 10]

The highest-level procedure, that is the only one performed in the main begin…end block, consists of high-level procedures for **initializing graphics** (*Initialise(), LoadFromFile(), MenuButtons(), AnimateTitle()*), **maintaining graphics** (*PressButton(), AnimateTitle2(), UnpressButton()*) and contains **prime loop** for checking if any mouse input has occurred.

For such action, it uses function *PollMouseEvent (mouseEvent: MouseEventType),* returning boolean. When true, the mouse event is returned in the MouseEvent argument. The *MouseEventType* is a record structure, defined in winmouse unit.

When *PollMouseEvent* returns true, MouseEvent argument is processed in function *ProcessMouseEvents().*

In addition, prime loop contains also procedures for menu functionality, StartGame(), Difficulty(), Instructions(), HighScore(), which call depends on position of pressed menu button returned from the function *ProcessMouseEvents()*.

The loop ends if *CloseGraphRequest* is set to true, which is also wingraph feature returning boolean whether user clicked the close button or not.

Afterwards *Finalise()* procedure is used for **finalizing graphics** and *FreeAnim()* is utilized for freeing the allocated memory by AnimatType, when animating the title. *AnimatType* is a record structure defined in wingraph unit.

#### 2.1.2.1. Initializing graphics

**a. procedure Initialise()**

Uses the wingraph routines for initialization of the graphical window. Also reserves previously obtained *Size* bytes memory on the heap, and returns a pointer to this memory,

then loads an image of back icon from back.bmp file there. Back.bmp file is necessary for good functionality of this procedure.


`b. procedure LoadFromFile (fileName: string; title: boolean)`

If title is set to false, it loads text from a given file until the end of it occurs and displays it on graphical window with stated parameters. Else randomizes given colors from global colors array and use different one of them for every displayed line of the title MineZweeper.

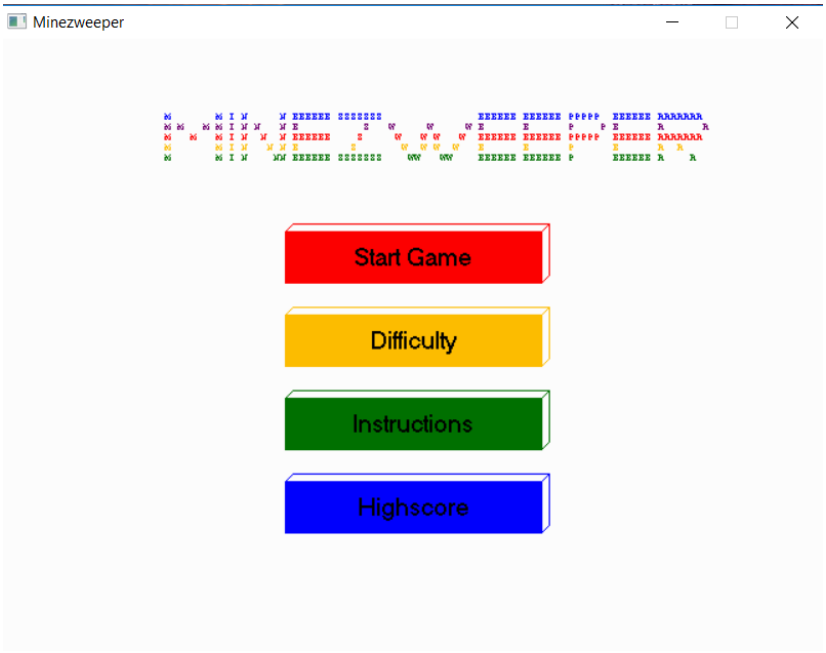
`c. procedure MenuButtons()`

Draws menu buttons with appropriate text and color from global arrays. For 3D effect is used *Bar3D* shape from wingraph unit.

`d. procedure AnimateTitle (var anim: AnimatType)[6,7]`

Gets required number of pixels (loaded title) from screen with the function *GetAnim()* and saves them as *anim*, one of the global variables for further manipulation, when creating a rotation animation of title. Also puts *anim* bitmap firstly on the screen for overdrawing the title using *PutAnim()* and parameter CopyPut, defined by wingraph.

Sets the graphics updating to UpdateGraph(UpdateOff) for reducing flickering of the animation.



### 2.1.2.2. Maintaining graphics

**a. procedure PressButton(j: integer; menu:boolean)[6]**

Clears the area of pressed button, defined by *j*, and draws new rectangle over it with appropriate color and text from global arrays. Using wingraph's functions and *ClearButton()* procedure for clearing the rectangle.

It is used to press menu buttons in *Main()* procedure but also difficulty buttons in *Difficulty()* procedure, it depends on the value of the *menu* boolean.

**b. procedure AnimateTitle2(anim: AnimatType; var i: integer)[7]**

Puts *anim* bitmap on the screen with rotation motion, created by increasing variable i and constant value Pi18 = Pi/18. Firstly by *PutAnim()* and parameter BkrPut and secondly with TransPut, compeling graphics to update now with UpdateGraph(UpdateNow) routine.

**c. procedure UnpressButton(j: integer)[6]**

Clears the rectangle of pressed menu button, defined by *j*, and draws new menu button with appropriate text and color from global arrays over the area. Using wingraph's functions and *ClearButton()* procedure for clearing the rectangle.

At last forces graphics to update with *UpdateGraph(UpdateOn)* routine.

I. **procedure ClearButton(j: integer)[6]**

Clears the viewport of the fixed sized and positioned button, defined by *j*, with white color, manipulating with wingraph's features (*SetViewPort(), ClearViewPort(), SetBkColor()*).

### 2.1.2.3. Prime loop

**a. function ProcessMouseEvents (var buttonPressed: boolean): integer [3]**

For usage and removal of the first mouse event in the queue it uses *GetMouseEvent(mouseEvent)*. Checks if the event is left mouse button click.

If yes, then it gets mouse cursor's x,y coordinates with *GetMouseX()* and *GetMouseY()* and examines if the cursor's location is in the area of any menu button.

Returns the position of clicked menu button in integer and constantly works with global variable *buttonPressed*, so the graphics and individual procedures for menu buttons that are invoked in Main procedure.

**b. procedure StartGame()[8]**

Frees the allocated memory of back icon, closes graphical window and executes process Game01, without inheriting the event handle and exits the Intro1 process.

**c. procedure Difficulty()[1,3,4,9]**

Redraws the graphical window and creates black colored buttons with difficulty options from global array. After these wingraph functions, *UpdateGraph(UpdateOn)* routine is set for updating the graphics.

Afterward loop for checking the mouse input is executed until one of the options was clicked by left mouse button or the close graphical window request was registered and it automatically returns back to menu (redraws graphics) using *BackToMainMenuGraphics()* procedure or ends.

For mouse input, it uses function *PollMouseEvent (mouseEvent: MouseEventType),* returning boolean. When true, MouseEvent argument is processed in function *ProcessMouseEvents().*

Button pressing graphics is maintained with the procedure *PressButton()*, described above and right argument chosen are passed through text file by *RewriteMenuResolution().*

`d. procedure Instructions()[6]`

Clears the graphical window with white color- *ClearDevise(),* loads predefined instructions from file *Instructions.txt – LoadFromFile()*. Instructions text file is not necessary for the game, but contains useful information for user, which could not be displayed without the file.

Furthermore uses *DrawBack()*, which draws back icon, updates graphics and *Back()* procedure with loop, checking mouse input for going back to the menu.

`e. procedure HighScore()[6]`

Clears the graphical window with white color- *ClearDevise(),* loads predefined instructions from file *Highscore.txt – LoadFromFile()*. Highscore text file is essential for correct functionality of the Game01 program.

Furthermore uses *DrawBack()*, which draws back icon, updates graphics and *Back()* procedure with loop, checking mouse input for going back to the menu.

I. `procedure BackToMainMenuGraphics()[6]`

Clears the graphical window with white color- *ClearDevise()* and draws menu buttons – *MenuButtons()*.

II. `procedure RewriteMenuResolution(j : integer)[9]`

Chosen difficulty, defined by *j*, is written in *MenuResolution.txt*, necessary text file for running the program. File is afterwards read in Game01.

III. `procedure DrawBack()`

Draws a loaded image *back.bmp* in left upper corner of the window.

IV. `procedure Back()[1,5]`

Contains loop for checking mouse input with *the function PollMouseEvent (mouseEvent: MouseEventType),* returning boolean. When left mouse button was clicked it gets mouse cursor's x,y coordinates and examine if was clicked upon the area of back icon. If so then it returns to menu by *BackToMainMenuGraphics(). Uses delay function for CPU resting.*

#### 2.1.2.4. Finalizing graphics

a. procedure Finalise()[10]

Used for preservation of the graphical window. After close graphical window request occurs, it releases allocated memory with bitmaps and closes graphical window by *CloseGraph()* procedure.

## 2.2. Game01

Second main program Game01 is the modified MineSweeper implementation. It consists of two highest-level procedures, *Load(),* used for all initialization and loading requisite data to memory, and *Main()*, containing the prime loop, which checks for user input and processes it, starts timer and ends if close graphical request occurs.

### 2.2.1. Global variables

- bitmaps: array [0..11] of pointer
  *Points to allocated memory blocks with loaded static images, used throughout the program, freed at the end or when executing Intro1 process.*
- ended, startTime, first, ex: boolean
  *All of these booleans are helping with determining whether some functions should or shouldn't be executed. Ended controls the prime loop, startTime carries if time should be started, first determines if it is the first use and ex says when to exit, because other process has been executed.*
- rows, cols, count, c: smallint
  *Rows and cols are actual values of rows and cols of the grid in played difficulty. C is a constant calculated from number of cols and rows, used for positioning images and text and count is number of mines appropriate also for the grid size. All of these smallints are widely used throughout the program.*
- timeCount: integer
  *Counter of the time (score), it's used at several parts of the code, so it's convenient and easier to maintain it as global variable.*
- mines, seconds: string
  *Auxiliary strings. When drawing text to graphical window happens, string is required, so count and timeCount are used for calculations, mines and seconds for displaying continuously changing text on screen (number of mines and number of seconds passed).*
- grid: array[0..24, 0..16] of STATE

  *Maintains player's visible grid. STATE is an enumerated type used for easier understanding of current square state. STATE = (opened,closed,flaged)*
- grid2: array [0..24, 0..16] of Boolean
  *Game grid for mines. True if the mine is located on the square.*
- grid3: array [0..24, 0..16] of integer

*Game grid for calculating and reading numbers around mines.*

Grid 2D arrays don't have dynamic length according to different grid sizes for different difficulties. Instead they all have size of the biggest one. It's a more reasonable option.

### 2.2.2. procedure Load() [3, 4, 13]

One of two highest-level functions. Serves as an initialization for every structure or construct in the game. It consists of high-level procedures divided in theoretical blocks: **Initializing graphics** (*Initialise(), GameStatus(), Menu(), MineCounter(), Timer(), CreateGrid()*) and **initial calculations** (*DistributeMines(), NumbersAroundTiles()*).

#### 2.2.2.1. Initializing graphics

a. `procedure Initialise()[4]`

Reads from MenuResolution.txt and sets number of cols and rows. Uses the wingraph routines for initialization of the graphical window with appropriate size for the chosen difficulty.

Afterward loads twelve images to memory by *LoadStaticImage()* procedure. Initializes grid and grid2, two 2D arrays of STATE and boolean values to closed and false. Also sets variables timeCount, seconds, first and ex. For determining seconds uses the sysutils unit feature TimeToStr(Time), where Time is a record structure and the conversion gives time as 11:23:56 string.

b. `procedure GameStatus(bitmap: pointer)`

Creates an emoji image for equivalent game progress by wingraph's *PutImage()* procedure. Uses constant c for correct positioning and given pointer decides on what image is drawn.

c. `procedure Menu()`

Creates an image for going back to menu option by wingraph's *PutImage()* procedure. Uses constant c for correct positioning.

d. `procedure MineCounter()`

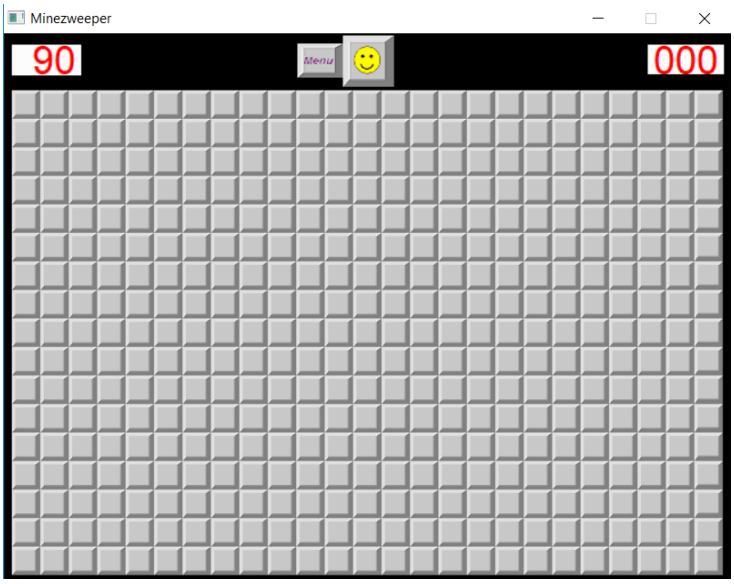
Initializes mine counter on the left upper corner and draws total number of mines.

e. `procedure Timer(seconds: string)`

Initializes and overdraws timer on the right upper corner every second. Gets time in string for easier drawing on screen and puts before it one or two zeros.

f. `procedure CreateGrid()[3,4]`

The last three images stored in memory and accessible through *bitmaps* array pointers are grids with different sizes of cols and rows. This procedure puts equivalent bitmap on screen by wingraph's *PutImage()* and releases all three images from memory. Called only once and grid images are necessary for the right game functionality.




I. `procedure LoadStaticImage(filename:string; n, m:smallint; var bitmap: pointer)`

Loads a 24 bits per pixel image from a given file, *n* stands for image width and *m* for height. Edits the given pointer form global array of pointers.

#### 2.2.2.2. Initial calculations

a. `procedure DistributeMines(count1: smallint)[4,5]`

For random distribution of mines over the specified game grid. Procedure is called with total number of mines (-1) = count1. It uses *Randomize* and *Random()* function, returning random integer between zero and given parameter, for choosing the square for placing the mine. If any collision happens (Random has chosen number where mine was already placed) count2 will increase. Furthermore, the procedure will call itself recursively with count2 parameter until all mines hasn't been distributed.

b. `Procedure NumbersAroundTiles()[5]`

Fills grid3, 2D array of integers, with numbers depending on the location of mines. For every field in grid3, it tries all neighbor fields around and increases the number of mines (max 8) starting at zero. If the examined grid has mine the number sets to -1.

### 2.2.3. procedure Main()[1,6,8,9]

Contains the **prime loops** for checking if any mouse input has occurred. For that it uses function *PollMouseEvent (mouseEvent: MouseEventType),* returning boolean. When true, the mouse event is returned in the MouseEvent argument. The *MouseEventType* is a record structure, defined in winmouse unit.

When *PollMouseEvent* returns true while the game has not ended, MouseEvent argument is processed in the procedure *ProcessMouseEvents(). Otherwise* MouseEvent argument is processed in the procedure *ProcessMouseEventsAfterTheGameHasEnded().*

Therefore,there are **individual loops before the game has ended and after**, both are indicated by *ended* boolean, so they also contain condition for closing graphical window request. The first one manages timer (it freezes after ending), it starts when the user first clicks on any square and changes every second. The maximum number of seconds is 999 (16 minutes 39 seconds).

#### 2.2.3.1. Game loop

a. `procedure ChangeTimer(seconds1: string)[9]`

Seconds1 is the current time with precision in seconds from *Initialise()* procedure, given as global variable seconds, which is changed by this procedure when one second has passed. In that case, it also calls *Timer(),* described above, for redrawing timer.

b. `procedure ProcessMouseEvents()[6,7,8,9,10,12]`

Handles user's mouse input. For usage and removal of the first mouse event in the queue it uses *GetMouseEvent(mouseEvent)*. Then it gets coordinates of clicked square by procedure *GetCoordinates()*. Calculates number of cols and rows of the clicked square x1, y1 and examines what mouse button was pressed, left or right.

If left, always creates surprise face on game status icon with *GameStatus()* procedure. Afterward, if mouse cursor's coordinates was in the area of game grid and the clicked square was stated as closed, it reveals the square by procedure *OpenSquare()* for managing graphics and *DisplaySquare()* for displaying the number of mines around the square or an area without any mines. It also states the square in grid global array as opened. But if mine is present on the square, the game loop ends and procedure *MineActivated()* is called.

If mouse cursor was pointing elsewhere when clicked, examines whether it was game status icon or menu icon. In both cases, creates an illusion of pressing icon by calling *PressGameStatus()* or *PressMenu()*, releases allocated memory from remaining images, closes graphical window, executes equivalent process (Intro1 or Game01) without inheriting the event handle and exits by setting ex to true.

Furthermore, the game status icon is changed back to smiley face after a short delay.

If right mouse button was pressed then, if the mouse cursor's coordinates are in the area of game grid, it flags unrevealed (stated as closed) square by *Flag()* procedure or unflags square stated as flaged by *Unflag()* procedure.

In both cases the mine counter changes it's value by decreasing or increasing in procedure *ChangeMineNumber().* According to rules flaging a square also checks if all mines weren't flaged with the *IsGameWon()* function. If so, game ends and *ProcessWonGame()* is called.

I. `procedure GetCoordinates(var x,y: smallint; maxX,maxY: smallint)`

Using *GetMouseX()* and *GetMouseY()* gets the coordinates of the top left corner of the square, upon which has been currently mouse button pressed.

II. `procedure OpenSquare(x,y: smallint)[7]`

Draws a gray rectangle over the square. X, y are coordinates of the left upper corner.

III. `procedure DisplaySquare(x,y,x1,y1: smallint)[7]`

Displays the number of mines around a square or displays an area without any mines. If Displayed square has a mine around, it draws a number from grid3 global array with specified color for every number. Else it continuous displaying recursively surrounding squares until they all has a mine around or the border of the grid appears.

IV. `procedure MineActivated(x,y: smallint)[6]`

Graphics after uncovering a mine. Changes game status icon and puts image of mine upon all squares, where mines was. At the and puts one image of activated mine (red surrounding) at the coordinates of activating (arguments).

V. `procedure Flag(x,y: smallint)`

Draws an image of flag over the square, defined by given coordinates.

VI. `procedure UnFlag(x,y : smallint)`

Draws an image of square over the flaged square, defined by given coordinates.

VII. `procedure ChangeMineNumber(subtract: boolean)`

Subtract or adds one for every unflaged or flaged square, depends on the value of the boolean, and overwrite mine counter.

VIII. `function IsGameWon(): Boolean`

Returns true if every mine was flaged, else returns false.

IX. `procedure ProcessWonGame()[6,13]`

Creates the wining screen by filling a rectangle over the window with Xhatch fill, drawing text “You won”, score and equivalent high score, obtained from HighScore.txt file by *GetHighScore()* function.

Also evaluates if player’s score was lower than high score (the lowest yet) and calls appropriate procedure for updating it *UpdateHighScore(),*which triggeres execution of NameWindow program.

X. `procedure GetHighScore(i : smallint): string`

Returns equivalent high score from HIghScore.txt file, i indicates chosen difficulty and line, where the high score is located.

XI. procedure UpdateHighScore(j : smallint; score: string)[11]

Executes process NameWindow. Then reads gamer's nickname from created name.txt file and erase that file.

Afterward it copies HighScore.txt file to HighScore(1).txt with the exception of the line which should be updated, writes nickname and score there. Then erases the HighScore.txt file and renames HighScore(1).txt to HighScore.txt.


### 2.2.3.2. Ended game loop

a. procedure ProcessMouseEventsAfterTheGameHasEnded()[6,10,11,12]

Enables to start a new game after lost or won one by clicking on the gameStatus icon.

For usage and removal of the first mouse event in the queue it uses *GetMouseEvent(mouseEvent)*. Checks if the event is left mouse button click. If yes, then it gets mouse cursor's x,y coordinates with *GetMouseX()* and *GetMouseY()* and examines if the cursor's location is in the area of game status icon or Menu button located beside.

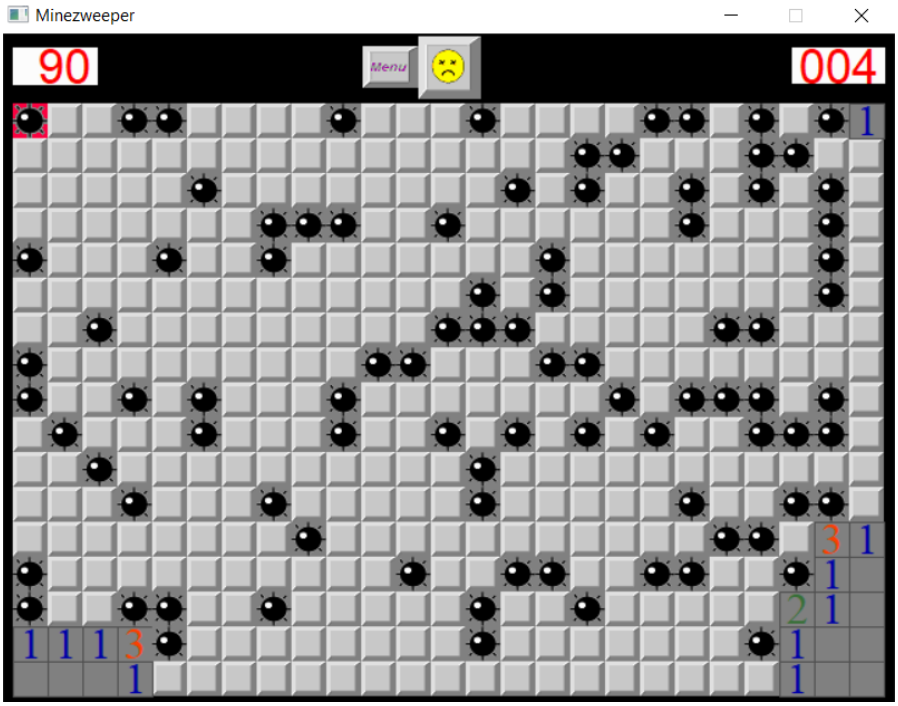
If so, then in both cases, procedure creates an illusion of pressing icon by calling *PressGameStatus()* or *PressMenu()*, releases allocated memory from remaining images, closes graphical window, executes equivalent process (Intro1 or Game01) without inheriting the event handle and exits by setting ex to true.

I. procedure PressGameStatus()

Draws gray rectangle over the game status icon. Using c constant for positioning and delay function.

## II. procedure PressMenu()

Draws gray rectangle over the game menu icon. Using c constant for positioning and delay function.




#### 2.2.3.3. Finalizing graphics

a. `procedure finalise()[12]`

Used for preservation of the graphical window. After close graphical window request occurs, it releases allocated memory with bitmaps and closes graphical window by *CloseGraph()* procedure.

## 2.3. NameWindow[11]

Third minor program, used for initialization of small graphical window, when inserting the gamer's nickname at the end of the won game while beating the highest score.

Takes in only letters and numbers as keyboard input and enter for confirmation. Then creates file name.txt for passing the nickname argument. It exits itself by closing the graphical window and user is back in the Game01 program, where the file name.txt is immediately erased.

One solution to this situation, because wingraph unit can't open another graphical window in or over the active, could be getting the nickname at Game01 by redrawing graphics, but it would be harder for implementation and not so good looking. Other, use parallel threads for executing the graphical window or visualize console window, but it ends with the same bad results as the firs solution.