Politechnika Śląska
Wydział Informatyki, Elektroniki i Informatyki

Computer Programming

WinAuto

topic	Mouse and keyboard recording and playback software
author	Szymon Pluta
instructor	mgr inż. Krzysztof Pasterak
major	Informatics
year	2018/2019
date	29-06-2019
github	**(WinAuto on GitHub)**

1 Project's topic

The goal was to design a software that would allow the user to register his mouse and keyboard actions, that is: cursor movements and mouse/keyboard keypresses. Consequently, the user should be able to retrieve what has been just recorded and replay the recording by simulating it. **WinAuto** provides a solution to this problem, and has some extra features.

2 Presentation, Analysis and Development

I have uploaded sample two minutes video to the streamable website to illustrate how the project works. **(Video URL)**.

Deep throughout API analysis is contained within Technical Documentation pdf file.

The full process is divided into two parts. The first part is the recording process, where keyboard, mouse events and the time duration between subsequent events are all registered. After the first part is done, the second part can be executed, that is: replaying what has been just recorded.

I have chosen to use some of WinAPI library functions, such as:

`‘GetAsyncKeyState‘, ‘SetCursorPos‘, ‘SendInput‘`

hence the software is Windows platform only.

Main issues:

- How to retrieve keyboard events in real time?
- How to synchronize it with retrieving mouse events?
- How to do both of the above while making things simple and optimized?
- Even if I know how to retieve both in synchronized fashion, how do I replay it?
- How to replay it without major glitches? If recording is exactly 60 seconds, how do I make replaying exactly 60 seconds as well?
- How do I make things work well not in just one window, but in any window, swapping between windows, and such?
- Which data structure to choose?

- How do I allow user to break out of the playback at any instant of time?
- How can I accurately simulate smooth cursor movement?
- Am I sure I cannot get better efficiency, time-complexity or space-complexity wise?
- Error handling?
- ...

3 Data structures

There are two major data structures.

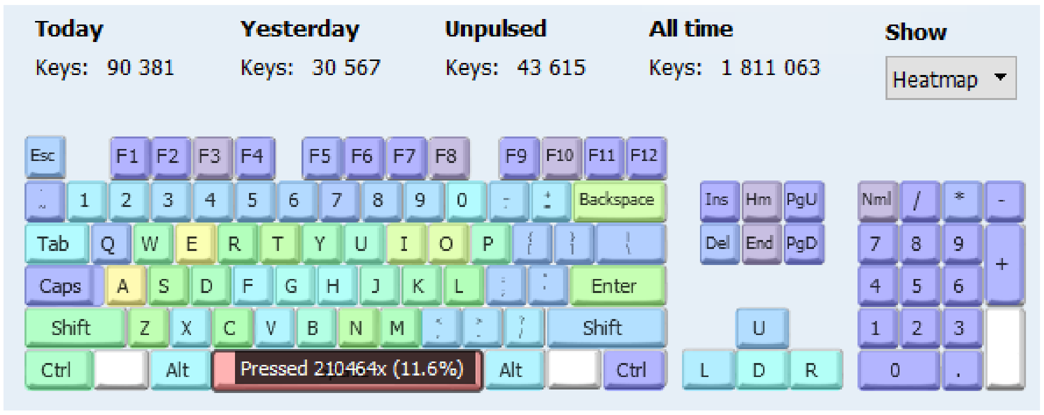
3.1 Array

The *keys-pqueue* which is a simple array that works, in a sense, like priority queue. Each element of the array is a given key's key code (scan code).

When checking if there is currently a keystroke, we first check the first element of the array, then the second, ..., then the last. Therefore, the first element has the highest priority to be checked and the last element has the lowest priority.

The first element has the highest statistical chance (based on my data) to be pressed; the function returns in such a case, if a keystroke is found, and the rest of the array is not iterated over (checked). Performance is gained. The array should be filled in proper order for maximum efficiency, then.

I have filled the array according to the data I pulled off of WhatPulse software based on nearly 2 000 000 keystrokes, making the priority sorting extremely reliable for me.
The array contains 104 elements, thus 104 different keys are supported. Full list of supported keys is available in technical documentation.



Today
Keys: 90 381

Yesterday
Keys: 30 567

Unpulsed
Keys: 43 615

All time
Keys: 1 811 063

Show
Heatmap

Pressed 210464x (11.6%)

Figure 1: Heatmap based on nearly 2 000 000 keystrokes

3.2 Doubly Linked List with conditional cyclic property

The second, and **main** structure is **Doubly Linked List with a property that it conditionally becomes Cyclic Doubly Linked List**, working as a queue.

This is the main *engine* of the recording and playback engine.

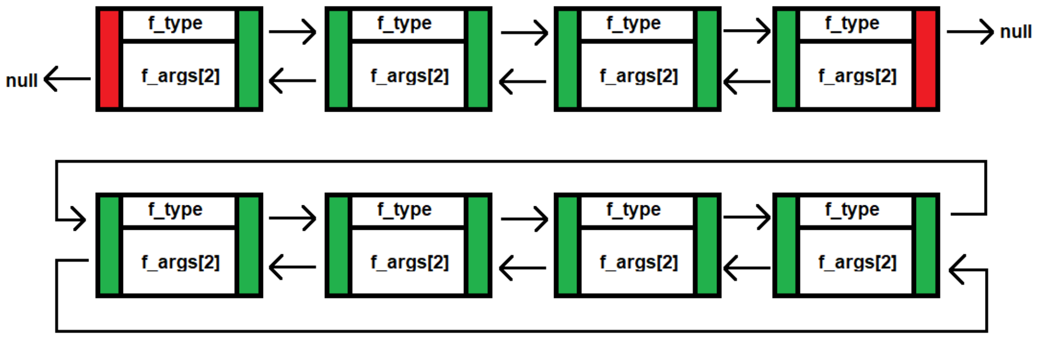


Figure 2: Doubly Linked List with Cyclic Property

This is a queue that keeps the *descriptions* of function calls to be executed during playback phase process. This is a queue that gets filled and gets iterated over during both recording and playback processes. It's a functions queue, or shortly: **f-queue**.

Putting pointers aside, each node consists of two members: *f-type* and *f-args[2]*.

The *f-type* describes function type. It attains only three values: $1, 2$ or 3. If node is of type 1, then it means this node is a function-call description corresponding to Cursor's Position.

If above mentioned Node has *f-args* equal respectively to:

$$100, 950$$

then it means the cursor's position at the time of this Node insertion was at that position.

Also, it means that during the **playback phase** the cursor will be set to that position.

If the *f-type* is of value 2, it is a Node describing keystroke event. In such case only one argument from *f-args* array is needed, and it contains the *KEY-CODE* of the given key.

When recording and the key A is pressed, the Node of *f-type* 2 is inserted and it attains *f-args*: $65, -1$ respectively. 65 is the key-code of letter A and -1 is neglected.

Furthermore, during *playback* phase (not the *recording* (filling queue, saving into file) phase) when iterating over the *f-queue* starting from the tail, and meeting the node of type 2, the program knows it has to be a keystroke and it checks for *f-args* value to know which keystroke should it be. Then, the keystroke is sent and it proceeds iterating over to the next Node.

The Node of type 3 corresponds to the delay in between keystrokes or mouse movements. The delay needs to be captured as well, otherwise during the playback everything would be executed instantly.

4 Algorithms

4.1 Brief Explanation

The program does **not** use any sorting algorithm like Merge Sort, which seeing the pseudocode is not the most difficult to implement. There simply were no usage at all for any sorting algorithm. However, I have introduced some advanced algorithms. All of the code *(except for simple function ends-with-suffix, which I took from stackoverflow and slightly modified)* is written by me.

4.2 Key press

There are few algorithms. Mention above getting the key that is currently being pressed with the highest priority in optimized fashion based on array priority queue is one of them.

4.3 F-queue insertion and memory freeing

Filling the Nodes in main struct **f-queue** is another. The Nodes are filled in such a fashion, that he list's space complexity is as small as possible.

When user does the recording, it might not seem like that, but most of the time he is idle and not taking any action. User does not always move a mouse, often he lets off the cursor stay in one place for even one second. And keystrokes are perhaps few times per second when typing.

That being said, every 10 miliseconds (for high precision) program scans for new keystrokes or mouse movements, and most likely during 10 miliseconds there were no new movements. Therefore most of the nodes would be of type 3 (sleep). That would quickly lead to very large number of nodes that would take large amount of memory.

In order to prevent that two actions are taken:

- if two consecutive nodes are of 3 sleep type, they are summed up together rather than adding brand new unnecessary node

- if there were no cursor movement during the 10 miliseconds period, then the cursor did not move, so there is no need to insert new cursor's position

The memory is **freed** with a function that never uses any excessive if statement. Surely freeing the list of size n still takes $O(n)$, however the real time complexity is slightly lower.

4.4 Extra Feature - ScreenSaver - Theory

The ScreenSaver is an extra feature I have added. All of its code is located in $smooth_cursor.c$. It uses propably the most complex algorithms.

The ScreenSaver (option 4 in program's menu) simply moves the cursor in random directions, in a smooth fashion. It has the ability to move the cursor with 1 frame per second or 150. It can move the cursor from point $P_1\left(x_1, y_1\right)$ to point $P_2\left(x_2, y_2\right)$ within different Δt, relatively short or relatively large.

Cursor moving from P_1 to P_2 in time Δt with precision FPS performs a jump (one cursor move) every:

$$t_{jump} = \frac{1000}{\text{FPS}}\left[\text{ms}\right]$$

and the total number of jumps n is defined as:

$$n = \frac{\Delta t}{t_{jump}} = \frac{\Delta t\text{FPS}}{1000}$$

.

The total distances cursor travels along x, y axises are $\Delta x, \Delta y$ respectively, where:

$$\Delta x = |x_2 - x_1| \quad \wedge \quad \Delta y = |y_2 - y_1|$$

, because cursor's displacement is a non-negative value.

Lastly, the x_{jump} (and y_{jump} is symmetrical) are defined respectively:

$$x_{jump} = \frac{\frac{1000}{\text{FPS}}\Delta x}{\Delta t} = \frac{t_{jump}\Delta x}{\Delta t} = \frac{\Delta x}{n}$$

The algorithm gets the screen's resolution using WINApi functions.

The screen saver is move-cursor-in-random-directions *fun* bonus feature. It can when the user gets away from the computer and does not want anyone to interact with his computer. (it is difficult to move cursor during this animation).

4.5 Playback of the Recording

Playback of the recording is simply iterating over the main struct, starting from the tail, and going towards the head direction. When iterating over new node it is parsed and proper function call is executed: *SetCursorPos, SendInput, Sleep* respectively for values $1, 2, 3$ of *f-type*.

4.6 Misc

The doubly linked list becomes cyclic list if the user wishes to infinitely playback the recording. In such case, the only stop condition is pressing the hotkey. After pressing the hotkey, the cyclic list becomes doubly linked list again and memory is freed.

The *init-menu* is a function that controls the entire interface process, workflow of the program. It sends specific (i.e. error) menu flags to the function and works recursively. The flags are based on the previous' program behaviour and print proper messages to the user.

5 External Specification

5.1 GUI

This is a command line program. The project's core is its engine. Surely GUI would increase its attractiveness, however it would not impact the core in any way. Hence it is ommited. Conditional compilation with GUI version has been considered.

5.2 Switches

The program runs perfectly fine without any switches, however $-h$ switch can be invoked to display help messages.

5.3 Input files

The program needs input **.txt** files in order to playback the recording. However, the **.txt** files are generated with the program anyway.

Therefore, if user aims to create his own recordings within the (2) menu option, he does not need any input files.

Further program executions, if provided with .txt files generated earlier, can directly lead to (3) menu option usage.

$WinAuto.exe - h$ invokes short help message.

6 Internal Specification

Detailed information of every function and structure is contained within *Technical Documentation.pdf.*

7 Output, Errors, Warnings

7.1 File input error

Recording function (2) always generates well-working files. However, if a situation that the generated files are manipulated on or modified by the user outside of the application usage, the program should detect that the input file is incorrect. Such file is considered **corrupted**.

During the reading file phase (3) menu option, if a *.txt* file is found to be corrupted, the reading is immedietely stopped and memory is freed.

7.2 Input file names

User can input file names when using (2) or (3) menu options. Files are considered good if and only if they exist, are accessible and not corrupted. File needs to end with **.txt** suffix and it is checked by the program if it does.

7.3 Scanf values

The program checks if scanf values match the expected values, and only such values are accepted.

7.4 Reading hotkey

Default hotkey is **F5**. However user can always set up his own hotkey by using (1) menu option. Later on that hotkey will be used within (2), (3), (4) menu options.

Blacklisted hotkeys are:

- Left mouse button *(keycode 1)*
- Right mouse button *(keycode 2)*
- Return button *(keycode 13)*
- Default hotkey itself *(keycode 116)*

When setting up a hotkey, it is remembered until the program ends or until next hotkey modification.

7.5 Program Errors

The program compiles without errors and warnings. No memory leaks were detected. The program should never crash during its execution.

8 Testing

The program has been tested mostly manually. Several testing functions are still located inside *tests* directory. Most testing functions are integrated within the program by now.

The accuracy of recording vs. playback has been measured, and it most often coresponds to about 1 percent off accuracy. It can be dependant on the user's CPU and how CPU heavy the recording is. (A lot vs. just a few operations)

The program does not take a big load of CPU because of being based on the *Sleep* function, which prevents CPU from going to 100 percent usage during heavy operations

9 Not featured

The program does support only keys located in the technical documentation. *(almost every single key from the standard keyboard)*

The program does not support using extra mouse buttons.

The program does not support holding multiple keyboard's buttons at the same time. It was never intended to. If two or more buttons are pressed simultaneously, the button with higher priority is chosen.

The program does not support prolonged keystrokes nor prolonged mouse clicks. All keystrokes are assumed to be instantaneous in time, brief.

The program should work fine when using several monitors, but has not been tested in that direction.

The program supports only Windows platform.

The program does not support extremely fast keyboard-typing, due to *GetAsyncKeyState*'s nature. It does suppor fast typing, just not extremely fast. Fast mouse movements are supported, though.

User should pay attention while switching windows during recording phase. During transition between windows (i.e: Google Chrome -> Notepad), the user should retain himself from doing heavy recording actions, like typing very fast. Very rarely small glitches happen.

10 Program's Applications

The program can find applications in anything that requires organized automation of the work process.

The (4) bonus feature can find a *fun* application and can be launched when going away from the Computer, if the user wishes not to be afraid that anybody would touch his computer while he is away.

The *Press hotkey to stop* console text can be removed and some combinations of keys can be disabled, thus making it fairly difficult to turn off the program without knowing what the hotkey is.

11 Conclusion

The program meets the requirements mentioned in the first **Project's Topic** section. It allows the user to record and simulate the recording afterwards, in a rather efficient manner. The program is considered finished.