2016 5th IIAI International Congress on Advanced Applied Informatics

FoXpace: Manipulating Windows Based on the User's Work History

Keisuke Yoshida, Tadachika Ozono, and Toramatsu Shintani
Department of Computer Science,
Graduate School of Engineering, Nagoya Institute of Technology
Gokiso-cho, Showa-ku, Nagoya, Aichi, 466-8555, Japan
Email: {kyoshida, ozono, tora}@toralab.org

***Abstract*—Manipulating windows accordingly is to perform tasks when working. However, a manual window management become difficult with PCs gaining larger display areas, greater processing power, and more applications. In this study, we explored a prototype window manipulating system called FoXpace. FoXpace captures user's work history including window moved, mouse clicks, keyboard inputs, and application-switch. We developed an algorithm to determine the optimal position of each windows based on the user's work history. This paper indicate possibility of an automatical window management for PC working with the user's work history.**

I. Introduction

In this study, we explored a prototype window manipulating system called the Flexible Optimizing Extraction space (FoXpace). FoXpace selects an empty space on the worker's display by estimating the importance of the window regions based on the user's work history. By allocating the application window to the space, FoXpace reduces the costs of manipulating windows in workspace design. The workspace is changed to reflect the number and type of application windows. When using a personal computer (PC) to perform tasks, workers typically have many application windows open. To perform their tasks effectively, a worker has to design a workspace by manually managing the windows. Window management has become more complex with PCs gaining larger display areas and greater processing power. As the worker performs different tasks, the number and type of applications being used change and the design of the workspace must be managed accordingly. To improve work efficiency, a more flexible mechanism of workspace design is required.

One approach to achieving window opacity or event-transparency is to reflect the available information[1][2]. This approach increases the visibility of the window contents. The cost of manual window management increases if recognition is difficult. Therefore, automated window management is desirable. This paper presents a definition of an effective workspace, and investigates the use of the user's work history in the implementation of automated workspace design.

The remainder of this paper is organized as follows. In Section II, we give a brief definition of a satisfactory workspace. We discuss the use of information on the user's work history in Section III. Section IV addresses the implementation of a workspace design mechanism and identifies future research priorities. Finally, Section V presents our conclusions.

II. Definition of Workspace

Workers often preform multiple tasks in parallel on a PC. Two factors dominate workspace design: the combination of applications being used for different tasks and whether window management can be automated. In this study, we did not address the combination of applications, but focused on window management.

Recent operating systems (OSs) such as OS X and Windows 10 include a virtual desktop, designed to represent multiple workspaces on the PC. Workers must allocate windows on the virtual desktop manually, requiring extrapolation of the combination of applications needed for the tasks in hand.

A. Automated Window Management

Automated window management has two aspects: 1) analyzing the information contained in the application window has. 2) Allowing the user to switch applications by directly accessing the window. To address the first point, The Window Distinction, M_{wd}, shows the information being displayed, and the visibility of the window is a measure of the display space management activity[3]. Therefore, M_{wd} can be used as a measure of workspace design. Note that M_{wd} does not depend on the language or execution environment being used on the PC. 2) A user can switch application by direct window access if such a switch is possible using a minimal mouse operation. Switching application methods by mouse operation can be achieved by clicking a window directly or by clicking the application icon in the taskbar or in Docs[4].

There are two main types of window layouts. One is a tiled window layout, and the other is a overlapping window layout. The tiled window layout is any open window is fully visible; windows are not allowed overlap. The overlapping window layout is any open window is allowed overlap and manipulated position in user's operations. A tiled window layout satisfies both requirements 1) and 2). However, this layout is inflexible with regard to the number of windows. Automated window management should therefore take into account the interest of the user in each window. DFW[5] is one way of addressing this. We focus on an overlapping window layout because this layout has more potential of maximizing visibility than a tiled window layout[6]. A user should learn techniques to satisfy both requirements 1) and 2) on an overlapping window layout. These techniques are difficult for beginners of this layout. Therefore, we discuss an automated window management on an overlapping window layout.

978-1-4673-8985-3/16 $31.00 © 2016 IEEE
DOI 10.1109/IIAI-AAI.2016.154

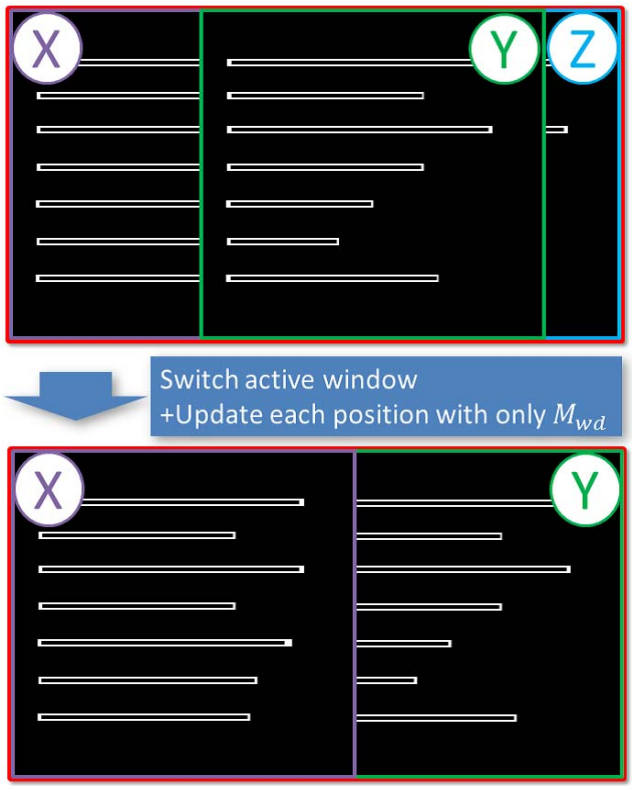

Fig. 1. A method of in which only M_{wd} is used for caliculating a value of information.

B. Quantity of Information

Fig. 1 shows an example of a method in which only M_{wd} is used for automating window management. In Fig. 1, X, Y, and Z denote the application windows. In the top part of Fig. 1, Y is the active window before the user switches active applications. All three windows are visible. A common technique is to make a portion of Z visible, to allow direct access by mouse click[4]. In the bottom part of Fig. 1, X is the active window after the user has switched active applications and updated the position of each window. X and Y are now visible. If the purpose of window management is to increase only M_{wd} as the evaluation value, Z is hidden behind Y. A technique in which that portion of the window is displayed in automatic window management is not available using only M_{wd}. For better automated window management a method of detecting the importance of the window to the user is required.

The information that the user is interested in is defined in the Window Interest Map (denoted as M_{wi}). This can identify the important region within a window, and the user can make just a small part of the window visible to allow application switching. The user can therefore minimize the visible window and optimize the contents displayed in the window.

C. Related Work

Many previous studies have investigated ways of increasing the amount of simultaneously visible information and its accessibility on the screen.

WinCuts allows arbitrary regions of existing of windows to be replicated in separate windows[7]. WinCuts improves the efficiency of the workspace by manipulating the window size. This technique provides a user workspace in which only meaningful windows are displayed. FST and IC are techniques for increasing the visible region by manipulating the opacity of windows[1][2], allowing users to see the window that is placed behind other windows. Switchback Cursor is a technique in which the cursor can be moved behind a window[8], providing the user with a 3D mouse operation. While the active window is normally in the foreground, Switchback Cursor can operate the window in the background. Fold-and-drop is a technique based on paper sheet metaphor, in which users can make the window appear to be folded back[9]. All these approaches attempt to make the workspace visible and accessible. These studies indicate candidates of parameters manipulated automaticaly: position, size, opacity, and event-transparency. We discuss how to set these parameters automatically to increase the window visibility and accessibility.

D. Workspace Design

In this study, we develop a workspace design to meet the two key requirements set out above. We introduce $M_{dev}(x, y, t_c)$ as a measure of the value of information at position (x, y) and the time t_c. M_{def} is the sum of M_{wd} and M_{wi}. Thus, $M_{dev}(x, y, t_c)$ balances the amount of information displayed and the user interest in the information. To improve the workspace design, M_{dev} should be maximized. The effectiveness of the workspace is given by

$$Evaluation(t_c) = \sum_{x}^{DW} \sum_{y}^{DH} M_{dev}(x, y, t_c) \quad (1)$$

The larger the value of $Evaluation(t_c)$, the more effective the workspace at time t_c. $M_{dev}(x, y, t)$ represents the value of information at position (x, y) and time t, and DW and DH are the display width and height, respectively. Windows are manipulated to increase the sum of M_{wd} and M_{wi}, so that $Evaluation$ increases. M_{wd} and M_{wi} are measures of the distinctiveness of the information and its interest to the user.

We focus on position that is one of the candidate of four parameters: position, size, opacity, and event-transparency. The positioning of the windows takes into account the user's work history. Our system manipulate all window of the active widow and inactive windows. The system operates flexibly, dynamically changing the number and type of application windows displayed.

III. VALUING INFORMATION BASED ON USER'S INTERACTION

A. Defining a Successful Workspace

As noted above, $M_{dev}(x, y, z, t)$ is a measure of the value of information to the user and the distinctiveness of that information. $M_{dev}(x, y, z, t)$ in Equation 2 presents the value of the information displayed at (x, y)-in the foreground window, and is derived as follows:

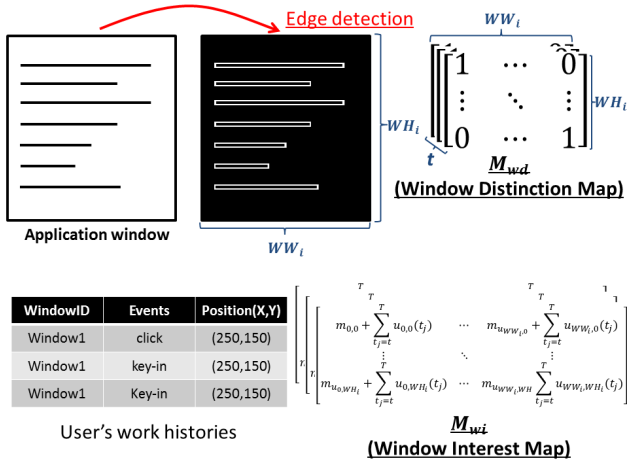

Fig. 2. A summary of M_{wi} and M_{wd} for calculating the value of information.

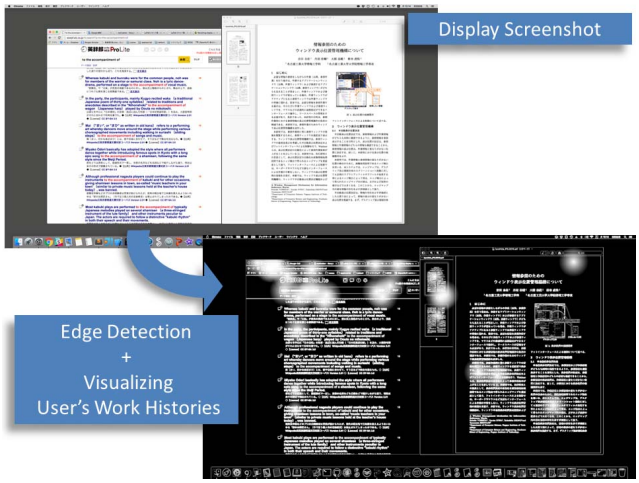

Fig. 3. Visualization of M_{wi} and M_{wd} when evaluating position (x, y).

$$M_{dev}(x, y, z, t) = \delta(x, y)\Big\{\alpha M_{wi}(x, y, z, t) + (1 - \alpha)M_{wf}(x, y, z, t)\Big\} \quad (2)$$

$\delta(x, y)$ outputs 1 or 0, showing whether an application is displayed in the foreground at (x, y), and α is a weight based on the importance of the information displayed. M_{wi} calculates the user's level of interest in the information, based on the work history $u(t_c)$ representing the user's work history at the time t_c. M_{dev} is the sum of M_{wd} and M_{wi}. A summary of the operation of M_{wd} and M_{wi} is given in Figs. 2 and 3. M_{wi} is a matrix of the sum of the user's work history. We call M_{wi} Window Interest Map. $M_{wi}(x, y, z, u(t)) \geq 0$ shows their level of interest inthe information presented in a window positioned in the z-th foreground at position (x, y) on the display. $M_{wd}(x, y, z, t) \geq 0$ shows the distinctiveness of

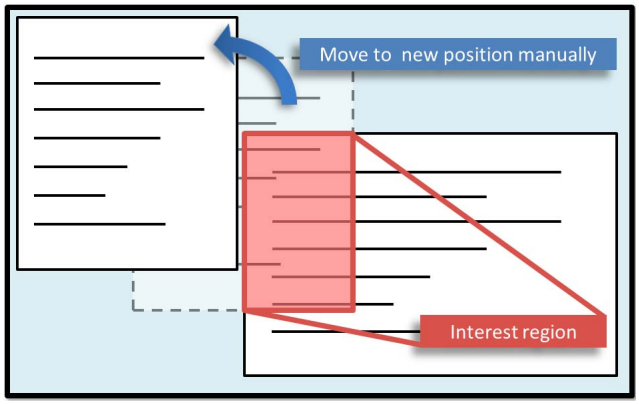

Fig. 4. A summery of M_{wm} for calculation value of interest.

the information in the same window, and represents a matrix of the information windows displayed. We call M_{wd} Window Distinction Map.

B. User's Work History

M_{di} derives the level of user interest in the information at (x, y). When a user actively interacts with a window, this suggests an increase in interest. We use M_{wi} to reflect a change in user interest and to restrict the significance of the work history. M_{wi} is the sum of M_{wm} and M_{wa}, as follows:

$$M_{wi}(x, y, z, u(t)) = M_{wm}(x, y, z) + \sum_{t_j=t}^{T} M_{wa}(x, y, z, u(t_j)) \quad (3)$$

Equation 3 derives a value for the user's interest based on the work history. $M_{wm}(x, y, z)$ shows the level of interest in one region of a window. M_{wm} derives the value of a region when a window is made visible with a mouse operation, and $M_{wa}(x, y, z, u(t))$ shows the level of interest based on the user's work history. M_{wa} derives the value of a position from the user's switching applications, from a mouse click, or from a keyboard input. $M_{wa}(x, y, z, u(t))$ is the sum of $u(t)$ for time T, where $u(t)$ is the user's work history at time t from the list of events, Fig. 2.

M_{wm} shows the level of interest in a region based on user's manual manipulation of a window position. The usual reason for moving a window is to see behind it, which suggests interest in the region made visible. M_{wm} is therefore increased when a user manipulates a window. M_{wm} is not temporary value, because GUI based applications have a compornent place is similar to its window. Fig. 4 shows M_{wm} calculating a value for the level of user interest.

In this study, we use mouse and keyboard operation events to represent a user's work history M_{wa}. There are three types of events: switching applications, a mouse click, and a keyboard input. Application switching events comprise clicking the window directory, clicking an application in the taskbar or

Algorithm 1 setting new window position algorithm

Input: w_{new}, t_c
$evaluation \leftarrow Evaluation(t_c)$
$i \leftarrow 0$
while $isCoverd(w_{new}, w \in W)$ **do**
  $setPosition(evaluation, w_{new}, i)$
  $i \leftarrow i + 1$
end while
for $w' \in \{w | isOverlap(w_{new}, w), w \in W\}$ **do**
  $updatePosition(evaluation, w)$
end for

Docs, and using a keyboard shortcut (Windows: Alt+Tab, OS X: Cmd+Tab).

When an application switching is recorded, the M_{wa} of all new active windows is increased. Mouse clicks are recorded unless they involve application switching, and the M_{wa} near the position at which the event happened is increased. Keyboard inputs are also disregarded if they trigger an application switching. Otherwise, a keyboard input increases M_{wa} near the position of the most recent mouse click.

Fig. 3 shows a before-after visualization of the interest level of the information. Interest in information in indistinct positions is indicated in white. Interest in information resembles a tab or search form in this example.

C. Edge Ditection

M_{wd} shows the distinctiveness of the information at (x, y). Our system uses an edge detection technique to extract M_{wd}. We take a gray-scale transformation of each window and apply edge detection with spatial filtering using Laplacian filter. Fig. 3 shows a before-after view of edge detection. Areas in white show the position of the edges. The system recognizes both characters and graphics as information. As these have a different color from the background, differences in color are used to detect borders. Edge detection shows areas where information is present.

IV. Implementation

A. Algorithms

In designing the workspace, candidate parameters for automated window management are position, size, opacity, and event-transparency[1][10][7][2]. Our design takes account of the legibility of the window, and the position and size are manipulated manually to ensure that the windows do not overlay each other. Integrated Development Environment (IDE) requires the area of the desktop to provide a comfortable work environment. Automatic manipulation of the size parameter disturbs the user's work pattern. Opacity and event-transparency are challenging as the user may become confused. In this study, we focus on automated manipulation of the window position to optimize its visibility and accessibility.

Our system updates the window position if a new application window is created or an active application is switched. When a user creates a new application window, the workspace and M_{wd} change significantly, reflecting a turning point in the task. Thus, the user needs to manipulate the window significantly. If a user switches an active application, M_{wi} again changes significantly. Concretely, the user is now less interested in the formerly active windows and more interested in the currently active window. If two windows are activated in alternate shifts, M_{wi} treats these as representing a group for the performance of a task. When applications are used in parallel, each application window should be maximally visible. M_{wd} and M_{wi} can use the progress of work on a task to reflect the information needed.

Algorithm 2 $setPosition(evaluation, w_{new}, i)$

Input: $evaluation, w_{new}, i$
$checkRow \leftarrow screenWidth/w_{new}.size.width$
$checkCol \leftarrow screenHeight/w_{new}.size.height$
$CA \leftarrow \langle\rangle$
for $j = 0$ to $checkRow$ **do**
  for $k = 0$ to $checkCol$ **do**
    for $dir \in \{TL, TR, UL, UR\}$ **do**
      $w_{candidate} \leftarrow candidate(w_{new}, dir, j, k)$
      $w_{candidate}.score \leftarrow evalPortion(w_{candidate})$
      $CA \leftarrow append(CA, w_{candidate})$
    end for
  end for
end for
$CA \leftarrow sort(CA)$
$w_{new}.pos \leftarrow CA[i].pos$

TL:Top Left	TR:Top Right	BL:Bottom Left	BR:Bottom Right

Algorithm 1 gives the setting of a new window position. It ensures that one application window is not masked by another, and derives the position that maximizes M_{dev}. The input values are the creation of a new window W_{new} and the current time t_c. The algorithm sets W_{new} in an empty space and operates in two steps.

Step 1 sets up a new window in an empty space unless this would mask existing windows. The function $isCoverd(w_{new}, w \in W)$ outputs 1 if this is the case and, 0 otherwise. The function $setPosition(evaluation, w_{new}, i)$ places the new window in the i-th emptiest space in the separated displays on the grid. Step 2 detect existing windows that overlap with the new window. The function $isOverlap(w_{new}, w)$ outputs 1 if an existing window overlaps the new window, and 0 otherwise. Algorithm 2 has the function $updatePosition(evaluation, w)$. This sets the existing window at a position that maximizes M_{dev}. These steps take into account that when switching windows with a mouse operation, the user tends to click on the window directly[4].

When a user switches an active window, the sum of the previously active windows, M_{wi}, is increased and that of the newly active windows, M_{wi}, is reduced. The algorithm that updates the existing window position when the active window is switched calls $updatePosition(evaluation, w)$ in each window that overlaps with the newly active window. This approach supports the performance of tasks requiring multiple windows. The algorithm allows each window position to reflect the changed status of the task.

Algorithm 2 sets the window position at the i-th emptiest

Algorithm 3 $updatePosition(evaluation, w_{target})$

```
1: Input: evaluation, w_target
2: stopCount ← 0
3: current ← evaluation
4: while stopCount < 5 do
5:    distance ← random()
6:    CA ← ⟨⟩
7:    for dir ∈ {T, B, L, R, TL, TR, BL, BR} do
8:       w_candidate.pos ← move(w_target, dir, distance)
9:       w_candidate.score ← evalAll(w_candidate)
10:      CA ← append(CA, w_candidate)
11:   end for
12:   w_candidate ← MAX(CA)
13:   if current > w_candidate.score then
14:      stopCount ← stopCount + 1
15:   else
16:      stopCount ← 0
17:      w_target.pos ← w_candidate.pos
18:   end if
19: end while
```

T:Top	B:Bottom	L:Left	R:Right
TL:Top Left	TR:Top Right	BL:Bottom Left	BR:Bottom Right

space in the separated displays in the grid. The inputs are the sum of M_{dev} at the current time $evaluation$, a new window w_{new}, and a number of order i. In lines 2 and 3, $checkRow$ and $checkCol$ are calculated to separate the displays in the grid. In lines 5 to 13, candidates are selected for the new window position and given scores. This algorithm separates the display in grids using basing points dir, which are elements of the top left, top right, bottom left, and bottom right. $candidate(w_{new}, dir, j, k)$ outputs a candidate for the separated display in the j-th row and k-th column from dir as $w_{candidate}$. $evalPortion(w_{candidate})$ outputs the sum of a portion of M_{dev} giving a position for $w_{candidate}$ as a score. $append(CA, w_{candidate})$ outputs an array that is CA appended to $w_{candidate}$. Lines 14 and 15, sort CA with these scores and set w_{new} to the i-th position that has the lowest score in CA.

Algorithm 3 updates the window position. The purpose of Algorithm 3 is to set each window as part of a group to perform a common task. The inputs to this algorithm are the sum of M_{def} at the current time $evaluation$, and a window w_{target}. This algorithm searches for a local miximize M_{div} using hill climbing. It sets the window at a position that is local miximize of M_{dir}. In line 5, distance is assigned a random number by $random()$ as the moving distance. Lines 7 to 11, line up candidates for new window positions and score them. $move(w_{target}, dir, distance)$ outputs a candidate to be moved from dir and $distance$, and $evalAll(w_{candidate})$ outputs the sum of M_{dev} as a score. The candidate with the maximum score in CA is selected as $w_{candidate}$. If $w_{candidate}$ has a larger score than the current score $current$, w_{target} is updated with $w_{candidate}$. If $w_{candidate}$ has a lower score than the current score, $stopCount$ is incremented. The termination condition is that $stopCount$ exceeds a value of five, indicating that w_{target} has not been recently updated.

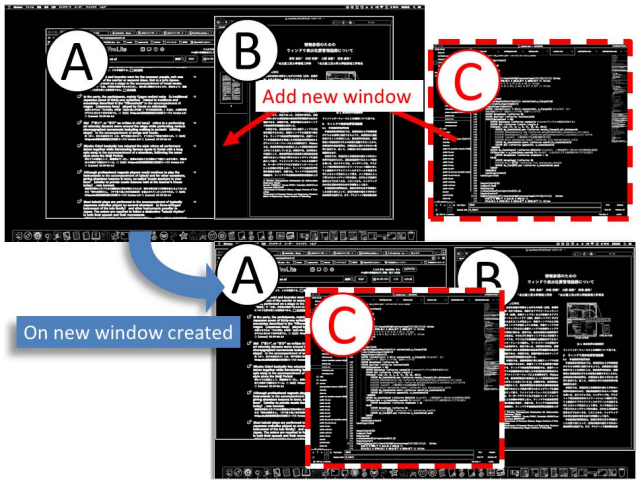

Fig. 5. An example of manipulating windows on a new window created.

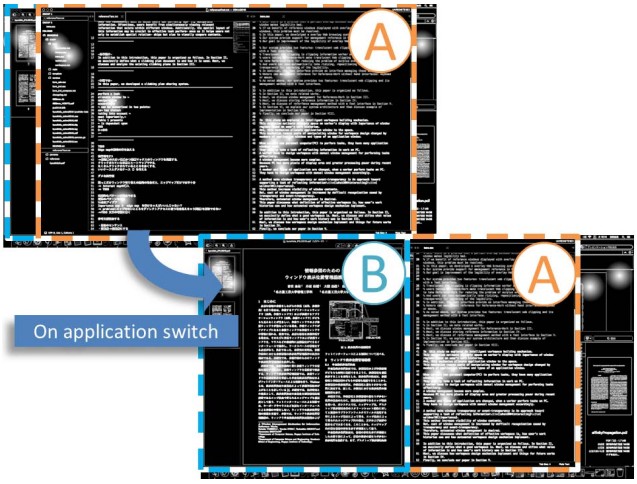

Fig. 6. An example of manipulating windows on an active window switched.

B. Example

A before-after example of the execution is given in Fig. 5 and Fig. 6. In Fig. 5, window A and window B existing windows showing a web browser and a PDF viewer. Window C is a new window showing a text editor. When the new window is created, it has the least information position as its initial position. At this time, the new window is set in a position that overlaps with the existing windows, and the overlaid portion is moved to position M_{dev}, representing the local maximum. In Fig. 5, window A moves to the top left and window B moves to the top right. This example shows our system locally maximizing the amount of information by manipulating a single window position. In Fig. 6, there are four windows on the desktop. Window A and window B are existing windows showing a text editor with two columns and a PDF viewer. When a user switches the active window from window A to window B, window A is overlaid by window B and moves to the left to maximize M_{dev}. The level of interest in the

information in the indistinct position (indicated in white) is six. There is interest in the information to the right of window A, so window A is moved to the left to show more information behind the two windows. This example shows how our system retains the possibility of direct access using a mouse click.

These examples of the execution demonstrate the more effective use of the display area. Our system increases the amount of information shown in the display, and allows each window to be directly accessed.

C. Discussion

In our system, M_{dev} is the sum of M_{wd} and M_{wi}. This allows windows to be accessed directly. However, M_{wd} is equally reflected in meaning less information. For example, in an editor like Microsoft Excel, part of the region may represent empty space to the user, but M_{wd} gives a large value from the edge detection of the ruled region. M_{dev} requires a method of eliminating regular patterns like ruled spaces. We predict that the efficiency of M_{dev} can be improved by introducing the product of M_{wd} and M_{wi}.

Peripheral vision acuity is weaker than central visual acuity. There is a limit to the information that can be seen when windows are displayed simultaneously, even when a large monitor or multiple monitors are used. When windows are widely separated on a large display, switching the gaze between windows alternately is difficult. We closely group related windows may improve the user's work efficiency. M_{wi} measures user interest in information based on the user's work history, and thus, M_{dev} may be used to vary M_{wi} (VAR_m). As VAR_m decreases, each interest of information is closely.

Files, folders, shortcuts, and applications can be placed directly on the desktop (DesktopFolder). DesktopFolder covers almost the complete region of the desktop, and is a special folder that can be easily accessed. Users often place temporary files and folders of projects in progress in DesktopFolder. These are easily accessed by clicking. NMs leave desktop icons uncovered, allowing direct access[11]. We assume that desktop icons are important when performing tasks and M_{dev} should be expanded to include a Desktop Icons Map M_{di}, showing the position of the desktop icons.

In our proposed system, we specifically focused on the position of windows. As noted above, size, opacity, and event-transparency are also important. A window management system should understand the user when manipulating the window size automatically. This should take into account not only resizing of the window but also rescaling. M_{wd} is changed by window rescaling, giving an indication of how a window is rescaled.

M_{dev} is useful for automating the control of window's opacity and event-transparency. However, this may also reduce legibility, for example, when translucent characters overlay other characters. Thus, the automatic opacity and event-transparency need to be controlled in a way that improves legibility[10].

V. Conclusion

In this paper, we explored a prototype window manipulation system. We presented a definition of an effective workspace, and discussed the use of a user's work history to automate workspace design. We defined M_{wi}, based on the user's work history, as a sum of M_{wm} and M_{wa}, taking account of window movement events. We used M_{wa} to set values from the user's work history based on application switching, mouse clicks, keyboard inputs. We set M_{wd} based on edge detection. We defined M_{dev} as the sum of M_{wi} and M_{wd}, and used it to automate window management. We discussed potential future improvements to automated workspace design. Our approach can reduce the cost of window manipulation when performing tasks on a PC.

Acknowledgment

This work was supported in part by JSPS KAKENHI Grant Number 15K00422, 16K00420.

References

[1] E. W. Ishak and S. K. Feiner, "Interacting with hidden content using content-aware free-space transparency," in *Proceedings of the 17th Annual ACM Symposium on User Interface Software and Technology*, ser. UIST '04, 2004, pp. 189–192.

[2] M. Waldner, M. Steinberger, R. Grasset, and D. Schmalstieg, "Importance-driven compositing window management," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ser. CHI '11. ACM, 2011, pp. 959–968.

[3] G. Robertson, M. Czerwinski, P. Baudisch, B. Meyers, D. Robbins, G. Smith, and D. Tan, "The large-display user experience," *Computer Graphics and Applications, IEEE*, vol. 25, no. 4, pp. 44–51, 2005.

[4] D. R. Hutchings, G. Smith, B. Meyers, M. Czerwinski, and G. Robertson, "Display space usage and window management operation comparisons between single monitor and multiple monitor users," in *Proceedings of the Working Conference on Advanced Visual Interfaces*, ser. AVI '04. ACM, 2004, pp. 32–39.

[5] H. Shibata and K. Omura, "Reducing the cost of window operations by docking windows," *International Journal of Innovative Computing Information and Control*, vol. 9, no. 12, pp. 4665–4679, 2013.

[6] S. A. Bly and J. K. Rosenberg, "A comparison of tiled and overlapping windows," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ser. CHI '86. ACM, 1986, pp. 101–106.

[7] D. S. Tan, B. Meyers, and M. Czerwinski, "Wincuts: Manipulating arbitrary window regions for more effective use of screen space," in *Proceedings of the CHI '04 Extended Abstracts on Human Factors in Computing Systems*, ser. CHI EA '04. ACM, 2004, pp. 1525–1528.

[8] S. Yamanaka and H. Miyashita, "Switchback cursor: mouse cursor operation for overlapped windowing," in *Human-Computer Interaction–INTERACT 2013*. Springer, 2013, pp. 746–753.

[9] P. Dragicevic, "Combining crossing-based and paper-based interaction paradigms for dragging and dropping between overlapping windows," in *Proceedings of the 17th Annual ACM Symposium on User Interface Software and Technology*, ser. UIST '04. ACM, 2004, pp. 193–196.

[10] K. Yoshida, , Y. Niwa, T. Ozono, and T. Shintani, "A method for improving the legibility of overlay web browsing," in *The Institute of Electronics, Information and Communication Engineers*, vol. 115, no. 381, 2015, pp. 25–30.

[11] D. R. Hutchings and J. Stasko, "Revisiting display space management: Understanding current practice to inform next-generation design," in *Proceedings of Graphics Interface 2004*, ser. GI '04. Canadian Human-Computer Communications Society, 2004, pp. 127–134.