EUROGRAPHICS 2012 / P. Cignoni, T. Ertl
(Guest Editors)

Volume 31 (2012), Number 2

Interactive Self-Organizing Windows

Markus Steinberger, Manuela Waldner, and Dieter Schmalstieg

Graz University of Technology, Austria

Abstract

In this paper, we present the design and implementation of a dynamic window management technique that changes the perception of windows as fixed-sized rectangles. The primary goal of self-organizing windows is to automatically display the most relevant information for a user's current activity, which removes the burden of organizing and arranging windows from the user. We analyze the image-based representation of each window and identify coherent pieces of information. The windows are then automatically moved, scaled and composed in a content-aware manner to fit the most relevant information into the limited area of the screen. During the design process, we consider findings from previous experiments and show how users can benefit from our system. We also describe how the immense processing power of current graphics processing units can be exploited to build an interactive system that finds an optimal solution within the complex design space of all possible window transformations in real time.

Categories and Subject Descriptors (according to ACM CCS): H.5.2 [Information Interfaces and Presentation]: User Interfaces—Windowing systems

1. Introduction

During the last years the number of windows present on average computer desktop has dramatically increased. The world wide web offers uncountable pieces of knowledge, people are living their lives connected with others using social networks and instant message services and current computer hardware is able to run concurrent multiple applications, supporting more windows on larger screens. Current computer users keep an average of ten "working spheres" concurrently open [GM04], each of which may require interactions with multiple applications. Unsurprisingly, as users tend to have many windows open, they also invest a significant amount of time in organizing the window layout, resizing the windows, or altering the window stacking order [HSM*04, TCH*09]. The time spent on these operations can be classified as overhead time [HC86], which would be zero in an ideal system.

In this work, we present the design and implementation of an automatic window management technique that uses the available display space to reduce overhead time more efficiently than previous techniques. To achieve this result, we advance from seeing windows as rigid, fixed-sized rectangles with no information about their content to treating them as elastic borders enclosing multiple pieces of information with varied importance. We analyze each window's content, define the coherent pieces of information with their importance, apply non-linear transformations, and cut away unimportant window regions to squeeze more important content into the limited sized display than traditional window managers. To achieve these results, we draw from the fields of

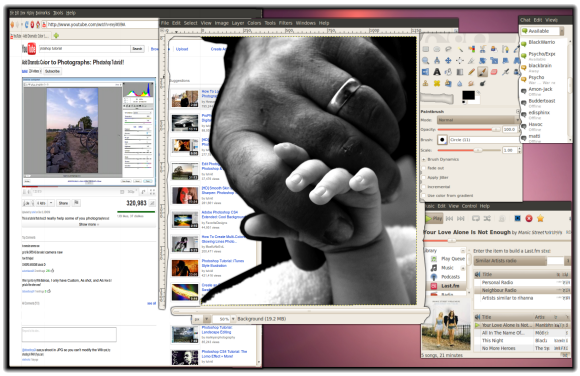

Figure 1: *Our technique relocates windows, non-linearly warps their content, and employs see-through compositing to maximize the amount of important information displayed on the screen. Non-linear transformations are used to show content that would otherwise require twice the display size.*

focus-and-context rendering and content-aware image resizing as well as from previous analyses of window management techniques. We formulate window rendering as an optimization problem, which we then solve with real-time update rates building on the capabilities of the latest Graphics Processing Units (GPUs). In summary, our contributions consist of the following:

- analyzing windows for coherent pieces of information to define content-aware non-linear transformations on them;
- combining non-linear window transformations, available display space and high-level considerations, such as the user's spatial memory in an optimization problem for presenting windows;
- solving this optimization problem with real-time frame rates using the latest GPU technology; and
- integrating the entire technique into an established windowing system.

Finally, we present three application scenarios to demonstrate the utility of the presented technique and show the results of an exploratory user study.

2. Related Work

Researchers have analyzed window management operations for more than a quarter of a century, and these studies should be considered when designing new window management techniques. In summary, the following findings are most relevant for our work.

F1 Window switching occurs often [Gay86, HSM*04].
F2 Users tend to have many windows open, and their number increases with the display size [HSM*04].
F3 Users rely heavily on mouse-based window-switching methods [TCH*09] (*e.g.*, users consider alt+tab for explicit window switching to be tedious [Gru01]).
F4 A user's spatial memory is important for retrieving information [RCL*98, RvDR*00].
F5 Under certain conditions, users do not consider automatic moving and resizing of windows to be distracting [MA99].

We address these findings in our technique's design process.

2.1. Window Management Techniques

One way of supporting the user in managing windows is to automatically arrange them spatially, reducing the overlap between windows [KS97, BF00, BNB01]. Instead of altering window locations to show more content, windows can be made (partially) transparent to enable the user to identify the content underneath the foreground windows [IF04]. By analyzing the window content, window relocation and window transparency can be combined [WSGS11]. Whereas [WSGS11] formulate the process of finding optimal window positions as a cost function, we go one step further and identify connected pieces of information to incorporate non-linear window transformations. Furthermore, we solve the problem with real-time frame rates, which is essential for a windowing system.

Another way of showing more windows concurrently on limited display space is shrinking. Shrinking methods include the 'traditional' resizing, cropping, and uniform scaling [MA00]. Another related technique is the extraction of user-defined regions [HS04, TMC04, MCRT06, SCPR06]. These extraction methods require manual activity by the user to define what is important. Using *Metisse* [CR05], these techniques could be combined. If a description of the user interface elements is available, an optimization problem can be constructed to generate a layout well suited for the user [GW04]. However, without the knowledge about what is important, automatic techniques cannot guarantee that relevant information is displayed when windows are scaled or regions are extracted. We automatically identify important window regions and apply non-linear transformations so that important information is shown with minimal distortions and the size of unimportant areas is greatly reduced.

Virtual Desktops or rooms [HC86] and task management systems [RvDR*00, RHC*04] reduce the amount of information displayed simultaneously by showing only those windows needed for a single task. Content analysis allows automatic task groupings [OSTS06, RC07], which increases the power of task management systems. However, single tasks often involve multiple application windows. Additionally, a binary assignment of a single task is often impossible [BSW08], which further increases the number of open windows. Thus, we see our technique not as an alternative to these approaches but rather as a complement.

2.2. Focus-and-Context Techniques

Non-linear transformations are often applied by distortion-oriented focus-and-context techniques to resolve the problem of too much information in too little space. These techniques include the *perspective wall* [MRC91], the *document lens* [RM93], *graphical fisheye views* [SB94], and *melange* [EHRF08]. Another simple but powerful distortion technique can be described by the metaphor of 'stretching the rubber sheet' [SSTR93]. Using this technique, large 2-D layouts can be distorted by adjusting the location of a few horizontal and vertical handles. We employ a similar technique to distort windows. [LA94] and [CKB09] review different kinds of focus-and-context techniques.

Some window management approaches also fall into the category of focus-and-context techniques. For example, the transformation used by display bubbles [CG06] could be seen as a fisheye transformation that distorts the outer regions of the desktop to fit into a constrained space. Content aware layout [IF07] non-linearly distorts windows exceeding the display area along the x-axis to fit on the screen. However, both techniques distort windows without considering their content.

2.3. Content-aware image resizing

If images or videos are presented in sizes or aspect ratios that differ from their original values, a content analysis can help preserve the most relevant information in original size. This analysis permits to either remove or add single seams [AS07], define a constraint transformation for each pixel [WGCO07], combine scaling and stretching [WTSL08], and even combine multiple operators [RSA09]. Even though we aim to fit windows into a constraint space, we cannot apply these approaches to window management directly, because we do not have a fixed frame in which to fit in a window. Instead, we must consider the position and scaling of all windows while we at the same time determine which content of each window to display.

3. Self-Organizing Windows

On limited sized displays, users spend a significant amount of time managing (*i.e.*, moving, switching and resizing) windows to view or access important pieces of information (**F1**). When increasing the display space, users keep more windows open (**F2**). Thus, the overhead of window management cannot be resolved with increasing display space. We therefore need to carefully decide how much display space to assign to each piece of information.

This is only one of the goals of self-organizing windows. There are other constraints that must be considered, such as tearing windows apart wildly, transformed text becoming unreadable, and windows jumping on the screen. Self-organizing windows transform windows so the most important window content for the user's current activity is displayed at its full size. Related pieces of information in other windows are visible and accessible. Secondary windows (*e.g.* instant messengers or music players) are significantly shrunken in size but still identifiably rendered on-screen, such that the user can efficiently switch to them (**F3**). In addition, it should be clear which content belongs to which window, the user should be able to intuit window location, and the overall layout should quickly stabilize.

3.1. Coherent Pieces of Information

Before we can define a method of arranging and transforming windows to present the user with the most important information, we must identify important pieces of information within windows and define each piece's importance for the user's current activity. Identifying important content is a common task in content aware image resizing and media retargeting [AS07, WGCO07, WTSL08]. Often, a model of saliency-based visual attention [IKN98] is used to find regions which are important. Building on saliency-based visual attention, importance maps have recently been introduced as low-level description of importance in windowing systems [WSGS11]. We compute this importance map for each window and threshold it at a low importance value. We assume that connected regions with importance above this threshold are coherent pieces of information. We can then assign an importance value to these pieces of information by computing the average importance of each piece, as shown in Figure 2.

It is difficult to define the importance of each piece of information for the user's current activity because the system cannot know what the user is exactly doing at any given point in time (*e.g.*, which paragraph of text the user is reading, at which image the user is currently looking, or whom the user wants to contact for a coffee date). However, we can infer the importance of information pieces within windows based on the low-level importance and deduce a window ordering based on usage, while allowing the user to easily influence this ordering. In our current system, we use the window stacking order. Thus, all relevant pieces of information from the active window are presented to the user, recently used windows get a high priority, and the system is intuitive. To allow more control, the user can exclude single windows from the self-organizing process.

Figure 2: *We segment windows into coherent pieces of information (the areas surrounded by a color) by thresholding the window's importance map [WSGS11]. Calculating the marginalized window importance (red and blue graphs) and thresholding it defines a rectangular grid on top of the window (short line segments with arrows) with which we distort the window. The grid lines normally coincide with region outlines.*

3.2. Window Parameterization

Multiple transformations, such as distortion-oriented, focus-and-context techniques [LA94, CKB09] or an image resizing operator [RSA09] can be used as a basis for mapping a single window onto a display satisfying the previously mentioned requirements. After testing multiple options, we decided to apply a form of orthogonal stretching [SSTR93], because it preserves the overall shape of the windows, symmetries, and the readability of text, *cf.* Figure 4. The transformation can be described by a few parameters (handles),

which allow an efficient and parallel computable solution to the optimization problem. The individual handle locations h along X and Y are given by $\mathbf{H_x} = [h_{x,0}, h_{x,1}, \cdots h_{x,n}]$ and $\mathbf{H_y} = [h_{y,0}, h_{y,1}, \cdots h_{y,n}]$ respectively. We choose the handle locations according to the previously mentioned definition of coherent pieces of information. Instead of segmenting these pieces in 2-D, however, we compute and threshold the marginalized window importance in the X and Y dimensions separately, a process that assigns the handle locations close to the boundaries of coherent pieces of information (see Figure 2). The transformation for displaying a window can be defined by mapping the handles to new locations $\widetilde{\mathbf{H}}_\mathbf{x}$ and $\widetilde{\mathbf{H}}_\mathbf{y}$ on the display. Given this setup, the areas between two adjacent handles (covering single coherent pieces of information) scale uniformly along each dimension.

The handles also span a rectangular grid over each window. The grid-points define a set of vertices, $\mathbf{v_{i,j}} \in \mathbf{V}$ with $\mathbf{v_{i,j}} = (h_{x,i}, h_{y,j})$, with which we define different penalties in the optimization problem. In this setup, orthogonal stretching is similar to feature aware texturing [GSCO06] and optimized scale-and-stretch [WTSL08], with the restriction that the mesh remains rectangular.

3.3. Optimization Problem

Based on the window ordering, we can setup a greedy optimization problem. According to the window stacking order (starting with the top level window), we attempt to find the optimal handle locations, $\widetilde{\mathbf{H}}_\mathbf{x}$ and $\widetilde{\mathbf{H}}_\mathbf{y}$, for one window at a time. The handle locations are optimal if they show the most important window content and neither harm the user's spatial memory (**F4**) nor introduce too much on-screen motion (**F5**). To weigh all these consideration, we formulate a cost function in which each consideration is represented by a single term.

3.3.1. Visibility

As we consider windows consecutively, we track the unused screen pixels in a display-sized availability map, A_d. We initially set all the pixels to free, indicated by a value of 0, while we define used pixel by 1. In case of a multi monitor setup, placing important window content at the boundaries between monitors should be avoided [Gru01]. To include this finding in the optimization problem, we set a one pixel line at these boundaries to unusable. If higher prioritized windows use pixels, the following windows will avoid these pixels and squeeze their most relevant content into the free areas. After a single window mapping has been determined, the pixels covered by this window are marked as used. Because not all pixels contained in windows hold important information (some represent background areas), we do not mark pixels with importance below the threshold. If a following window places content in a background area, the background is cut away from the higher priority window, which enables the user to see through the window.

To formalize the demand for showing each window's most important content, we penalize placing important content on unusable pixels:

$$P_o = \sum_{\mathbf{x} \in display} A_d(\mathbf{x}) \cdot I_{w,\widetilde{\mathbf{H}}_\mathbf{x},\widetilde{\mathbf{H}}_\mathbf{y}}(\mathbf{x}), \tag{1}$$

where $I_{w,\widetilde{\mathbf{H}}_\mathbf{x},\widetilde{\mathbf{H}}_\mathbf{y}}$ represents the window importance mapped to the display according to the transformed handles $\widetilde{\mathbf{H}}_\mathbf{x}$ and $\widetilde{\mathbf{H}}_\mathbf{y}$. Essentially P_o is the sum of the importance that is not shown due to the unavailability of display pixels. To prevent placing window content off-screen, we penalize handle positions outside the display boundaries by multiplying their distance to the boundary by a high value and adding this value to P_o.

3.3.2. Content-aware Scaling

Changing the distance between handles creates non-linear window transformations that shrink or enlarge certain areas. While transformations to unimportant areas are generally acceptable, distorting important regions should be avoided. To formalize this requirement, we penalize scaling along each dimension according to the importance of each area. The scaling penalty along one dimension, *e.g.*, X, is given by

$$P_{s,x} = \sum_{i<|\mathbf{H_x}|} Scl(\widetilde{h}_{x,i}, \widetilde{h}_{x,i+1}) \cdot \sum_{\substack{h_{x,i}<x<h_{x,i+1} \\ h_{y,0}<y<h_{y,n}}} I_w(x,y), \tag{2}$$

where the sum over $I_w(x,y)$ yields the importance of the area limited by the handle locations $h_{x,i}$ and $h_{x,(i+1)}$ and Scl describes the scaling applied to this area:

$$Scl(\widetilde{h}_i, \widetilde{h}_j) = \max\left(\frac{\widetilde{h}_j - \widetilde{h}_i}{h_j - h_i}, \frac{h_j - h_i}{\widetilde{h}_j - \widetilde{h}_i}\right) - 1.$$

$Scl(\widetilde{h}_i, \widetilde{h}_j)$ treats shrinking and enlarging symmetrically.

3.3.3. Distance and Motion

If the user interacts with windows (*e.g.*, alters the content of windows, or moves windows) the optimal window layout may severely and abruptly change. To reduce the influence on the user's spatial memory (**F4**) and keep on-screen motion low (**F5**), we penalize the deviation of every vertex $\widetilde{\mathbf{v}}$ from its original location $\mathbf{v}$ and its previous location $\widetilde{\mathbf{v}}_\mathbf{p}$:

$$P_p = \frac{1}{|\mathbf{V}|} \sum_{\mathbf{v} \in \mathbf{V}} |\widetilde{\mathbf{v}} - \mathbf{v}|_2 \qquad P_m = \frac{1}{|\mathbf{V}|} \sum_{\mathbf{v} \in \mathbf{V}} |\widetilde{\mathbf{v}} - \widetilde{\mathbf{v}}_\mathbf{p}|_2 \tag{3}$$

3.3.4. Combined Cost Function

Combining all the previously mentioned penalties, Equation (1 - 3), gives a common cost function:

$$C(\widetilde{\mathbf{H}}_\mathbf{x}, \widetilde{\mathbf{H}}_\mathbf{y}) = \alpha_o P_o + \alpha_s (P_{s,x} + P_{s,y}) + \alpha_p P_p + \alpha_m P_m \tag{4}$$

The set of handles $\widetilde{\mathbf{H}}_\mathbf{x}$ and $\widetilde{\mathbf{H}}_\mathbf{y}$ minimizing the cost function form the optimal mapping function for the current window.

The α values control the trade-off between the different constraints. For example, increasing α_o allows less overlap between windows by accepting more distortions and more deviation from the original window location.

4. Implementation

Even though we employ a greedy strategy, finding an optimal mapping for a window according to Equation (4) is a complex task. Working with standard desktop applications, the immense processing power of GPUs remains mostly unused. To harness this power, we employ a combination of GLSL shaders for graphics related tasks and use the Open Computing Language (OpenCL) for solving the optimization problem. Furthermore, using a compositing window manager, every window's content is ready for use on the GPU. In the following section, we describe the stages of our technique as shown in Figure 3. We first describe the computation of every window's importance and then discuss our gradient descent approach to finding the minimum of the cost function and why the intermediate results are well suited to be shown to the user. Finally, we give some implementation details and measurements describing the system performance.

4.1. Importance Analysis

To segment every window into coherent pieces of information and define the handles, $\mathbf{X_{H,O}}$ and $\mathbf{Y_{H,O}}$, we compute and store one importance map per window. We also keep the marginalized importance and the sum of the importance between two handles in GPU memory, reducing the number of operations required during the gradient descent stage.

4.1.1. Incremental Importance

We compute the importance map, I_w, from each window texture using a model of saliency-based visual attention [IKN98]. In standard desktop applications, such as word processors, web-clients or image processors, only parts of these windows are usually subject to changes. To benefit from this observation, we perform importance updates only for the areas that changed compared to the last frame. To calculate the importance, we compute an image pyramid using GLSL shaders, according to the saliency model given in [IKN98]. We deduce the updated region by circumscribing a rectangle around all changed areas and increasing its size while taking the pyramid depth and filter extent into account. According to our experiments, for an average desktop setup with ten windows or more, the time spent performing these saliency computations is reduced by approximately 95%, if saliency updates are computed for changed areas only.

4.1.2. Transformation Handles

To determine the handle locations, $\mathbf{H_x}$ and $\mathbf{H_y}$, we use the marginal importance of every window as previously described. To compute the marginal importance for each dimension, we use an OpenCL kernel to compute the sum of the importance values along the codimension. We again update these values only for the areas that have been altered. To define the homogeneous regions of information along every dimension, we normalize the marginal importance with the window importance average and threshold it with the empirically determined value 0.2, *cf.* Figure 2. According to our experiments, a small variation of the threshold value does not have any influence on the end result.

If the displayed content changes, the importance is altered, and the handle locations $\mathbf{H_x}$ and $\mathbf{H_y}$ also change. Because the window may already be distorted, we transform the changed handle locations according to the old transformed handle locations $\widetilde{\mathbf{H}}_\mathbf{x}$ and $\widetilde{\mathbf{H}}_\mathbf{y}$, and smoothly interpolate from the old handle locations to the new ones.

4.2. Optimization

The cost function given by Equation (4) must be minimized, to determine the transformation applied to each window. Because this cost function is based on the irregular availability map, A_d, and the window's importance, I_w, its shape can be complex with a large number of local minima. To build an interactive system, we intend not to find the global optimum of the cost function in every frame, but rather to locally improve the mapping of the last frame. This also reduces the influence on the user's spatial memory and guarantees smooth movements during system interaction.

4.2.1. Gradient Descent

We use a gradient descent-based optimizer to implement the search for the optimal window transformation. We work on all handles in parallel to evaluate the gradient direction. According to the OpenCL definition, we use one block of 256 threads to compute the influence of one grid cell on its surrounding handles. This work distribution results in $(|\mathbf{H}_x|-1)\cdot(|\mathbf{H}_y|-1)$ blocks of 256 threads being processed in parallel, which is normally sufficient to fully utilize standard consumer graphics cards. Each thread draws samples from A_d and I_w to approximate the gradient of Equation (1) using central differences. A parallel reduction in local shared memory makes this part of the gradient available for all threads. The first thread of every block analytically computes the other parts of the gradient and stores the influence of its block on the entire gradient computation using atomic operations.

As a starting point for the gradient descent, we use a location close to the transformation of the previous frame. We therefore evaluate the cost function for 20 randomly offset transformations in every frame, use a parallel reduction to find the one with the minimal cost, and use this transformation as a starting point for the search. The deviation from the

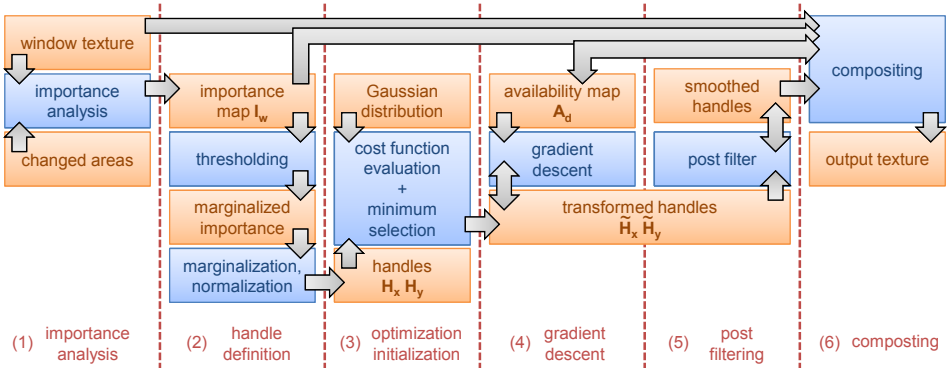

Figure 3: *An overview of the pipeline run through by every window (blue boxes correspond to OpenCL kernel launches, and orange boxes illustrate memory objects). (1) The importance map is calculated from the window texture using a filter pyramid. (2) The marginalized importance defines the handle locations. (3) The randomly offset handle locations are compared, and the best is used as the starting point for a fixed number of gradient descent steps (4), which advance the current window transformation closer to the optimum. (5) Post filtering keeps the transition toward the optimum smooth. (6) The output texture and the availability map A_d are updated according to the current handles.*

previous transformation is selected using a Gaussian distribution with a standard deviation corresponding to approximately 100 pixels. Although this strategy seems rather limited, the random initialization in every frame helps overcome the local minima.

4.2.2. Post Filtering

To achieve faster convergence speeds, we employ a normalized gradient descent with a momentum term, a large step length, and exponential decay on the step length. Although gradient descent is computationally inexpensive, finding the exact location of a minimum can take many steps (too many for an interactive system, eventually). Thus, we have chosen a different approach to this problem. We do not try to find the minimum in every frame and present intermediate results to the user instead. As the gradient moves from the vicinity of the previous transformation to a better one, this process morphs the window from one transformation to the optimum. Because the trail of the gradient is generally to rough for the user, *i.e.*, the windows would jitter too much, we introduce a post filter. This filter simply mixes the previous transformation with the transformation given by the current gradient descent location:

$$\mathbf{H_{t+1}} = \beta \cdot \mathbf{H_t} + (1-\beta) \cdot \mathbf{H_{gradient}},$$

where $\mathbf{H_t}$ and $\mathbf{H_{t+1}}$ describe the handle locations of the previous and current frame and $\mathbf{H_{gradient}}$ corresponds to the current location of the gradient search. We choose a β value around 0.9, yielding an infinite impulse response filter that generates smooth trajectories for the transformation handles. This filter helps to hide the jittery movement of the gradient and removes jumps due to the random initializations. Using this approach, we present the user with an image every 20 gradient descent steps with real-time update rates (see Section 4.5).

4.3. Compositing

After finding the window transformation, we use an OpenCL kernel to render the window into a common desktop-sized texture and update the availability map A_d. We do not use OpenGL for this task, because we must simultaneously read from and write to A_d, to evaluate if a pixel is still free and mark it as used. To visually aid the separation between different windows, we add boundary shadows at cut-away areas, similar to [WSGS11] (see Figures 4 and 6). After rendering the last window to the common texture, we render this texture with a screen sized quad using OpenGL.

4.4. Input Redirection

Because current windowing systems do not support distorted windows or cut-away areas within windows, we must modify the mouse input so it reaches the right window and the correct location within the window. We accomplish this modification using an input redirection map built during the compositing stage that holds information about the window displayed at each pixel. Every entry in this map stores three values: an identifier of the window and the two coordinates describing the pixel's location in the undistorted window. Whenever the mouse is moved, we query this map to determine the active window and the target position for the mouse pointer redirection.

4.5. Implementation Details and Performance

Our system is implemented as a plugin for the OpenGL-based compositing window manager Compiz that builds on the X Window System. The CPU portion of the implementation is written in C/C++ and primarily issues OpenCL kernel calls. OpenGL is only used for computing the importance map and displaying the final composited texture to the user.

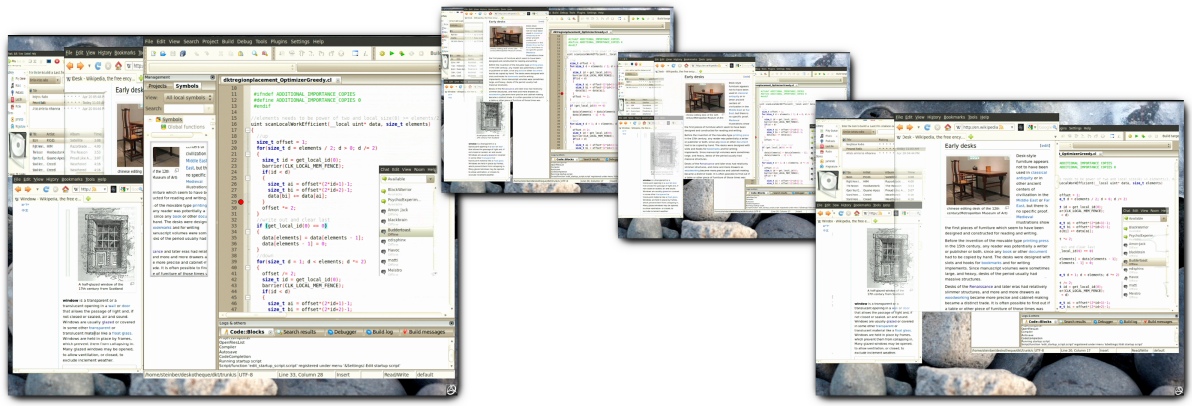

Figure 4: *A user changes the focus from a development environment to a web browser. The web browser is thus assigned the highest priority and its size is restored. The development environment's size is adjusted to better fit the available space. See-through compositing using cut-aways allows the lower-prioritized applications to shine through the top-level windows, making the instant messenger visible and accessible here.*

To assess the system's performance, we measured the time spent on every stage of the system for windows of different sizes, as shown in Table 1. On an Intel Quad-Core 2.80 Ghz CPU with NVIDIA GeForce GTX 480, the system handles 15 windows of variable size with an average frame rate of 50 fps at a display resolution of 1280×1024,

	Imp.	Seg.	GD	Comp.	Sum
256x256	2216	897	741	803	6250
512x512	5080	1013	1512	1250	11235
1024x768	13700	1035	3079	1540	23512

Table 1: *The performance measure (in µs) include the generation of a full window importance map (Imp.), window segmentation (Seg.), 20 gradient descent steps (GD), window compositing (Comp.), and all other overhead. Note that the generation of the importance map is rarely run fully, but is partially updated for changed areas.*

5. Usage Scenarios

Every task involving more than a single window requires the user to switch from one window to another. Our system can thus benefit users in many scenarios. We begin with a general case showing how self-organizing windows are transformed if the active window is altered. We then show how the user can benefit from the distance and motion penalties to bring related information spatially together on the screen and how window switching operations can be performed more efficiently with our technique.

5.1. Focus-and-Context

On a small screen self-organizing windows are the most beneficial when they reveal otherwise occluded window content. In the example in Figure 4, a user works with a development environment while the system scales and relocates peripheral windows, such as the two web browser applications and the music player to better fit into the available space. The otherwise hidden instant messenger application is presented to the user in an empty area of the development application. Due to self-organizing windows, at least small portions of the most important windows are presented to the user, which enable the user to switch to other application windows efficiently. In this example the user starts interacting with the web-browser, which triggers a dynamic adjustment of the presented content, greatly reducing the development application in size.

5.2. Combining Related Information

Because our system attempts to show as many important regions as possible and also considers the location of the information, the user can change the layout to bring related pieces of information together. This ability helps the user compare similar pieces of information, find matches in the data, and generate more pleasant layouts. In Figure 5, a user has placed images of replicas of Statue of Liberty replicas next to an image of the original. The system automatically stretches the unimportant areas of the non-focus window to show the text for the image of the original, which would otherwise remain hidden.

5.3. Pull-Through Window Switching

Empty space in the focus window can be used to display downsized versions of other full-screen windows, as shown in Figure 6. Due to content-aware scaling, areas with high importance are distorted less than other regions allowing users to identify the window. Clicking on a context window brings it to front, pulling it through the holes in the focus window.

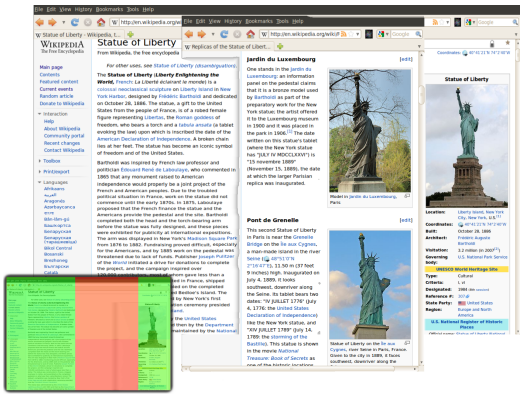

Figure 5: *The system automatically stretches the whitespace in the context window to display the window's text. The small color-coded insert shows the context window and its scaling.*

6. Evaluation

We performed an exploratory user study to evaluate whether our implementations met the self-organizing windows design goals, to understand how users interact with the technique, and to investigate the scenarios in which users can imagine using this technique.

6.1. Method

We recruited a total of 20 participants (aged 18 to 30, 14 males and 6 females) from a local university. During the study, they were asked to create a desktop application setup resembling an everyday computer-use scenario. After an initial demonstration by the experimenter, the participants set up multiple windows and tried to replay some common workflows using self-organizing windows, *e.g.*, planning a holiday trip with a map application and multiple browser windows, writing an article with the help of external sources, or, comparing tabular data with different visualizations.

We informally observed the participants while using the system and established a 'thinking-aloud' protocol to assess how they interacted with the self-organizing windows. After the initial experimentation phase, which lasted approximately 30 minutes, the participants were asked to complete a questionnaire comparing self-organizing windows to traditional window management. We then reconfigured the system to resemble several previously proposed window management techniques by varying the cost function. After another experimentation phase, the participants were asked to answer a second questionnaire, which allowed them to rate the techniques they had just seen and indicate if they could imagine using self-organizing windows in their everyday computer work. After the experiment, we asked the participants to take part in a follow-up experiment evaluating self-organizing windows over a longer period of time. Four of the 20 participants agreed to use our technique for at least one day of their typical computer work.

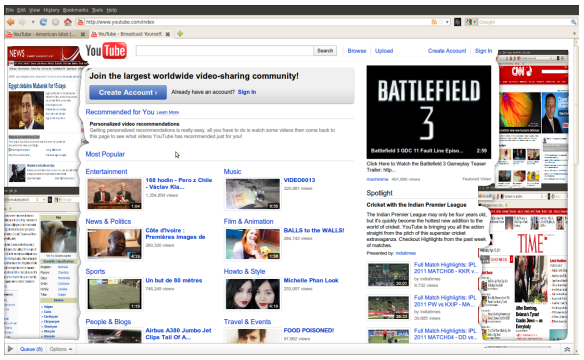

Figure 6: *Pull-Through Window Switching. The user can pull a downsized version of a context window through the empty space in the full-screen focus window to make the context windows the new focus.*

6.2. Results

The first questionnaire contained seven questions. The answers used a 7-point bipolar Likert scale in which -3 meant that self-organizing windows were greatly inferior to traditional window management, 0 meant that they were equal and $+3$ indicated that self-organizing windows were greatly superior. On average, the participants considered self-organizing windows superior in *usefulness*, *amount of visible useful information*, *time to access information*, *visual appeal*, and *window management overhead*. The questionnaire item accessing *the clarity of the association between windows and information* was considered equal to traditional window management. On average, the participants found the *on-screen motion* to be slightly disturbing. See Figure 7(a) for the detailed results of the questionnaire.

The second questionnaire contained six questions. The answers used a 5-point unipolar Likert scale in which 0 rated the technique as 'not useful' and 4 as 'very useful'. The evaluated techniques were *automatic window relocation* (as used in our system and similar to [BF00]), *see-through compositing using cut-aways* (as implemented in our system and in [WSGS11]), *see-through compositing using ghosting* ([WSGS11]), *opaque windows*, *uniform window scaling* (as proposed in [MA00]), and *content-ware scaling* (as proposed by us). Figure 7(b) contains an overview of the mean and standard error of the answers. Using non-parametric Friedman's tests, we compared the see-through compositing variants *cut-aways*, *ghosting* and *opaque windows* and found no significant differences among them ($\chi^2(2) = 3.361$, $p = .186$). A Wilcoxon signed-rank test revealed a significant difference between *content-aware scaling* and *uniform window scaling* ($Z = -3.695$, $p < .001$).

During the final experimental phase, all twenty participants stated that they would use self-organizing windows for single windows in their everyday computer work. Fourteen participants could imagine using self-organizing windows for all windows on their desktop.

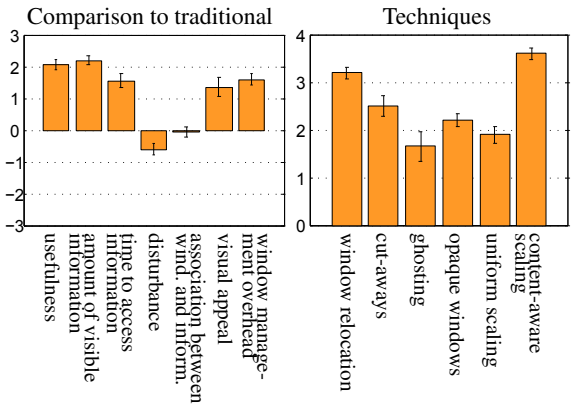

Figure 7: *Questionnaire results obtained from 20 participants. (a) compares self-organizing windows to traditional window management (positive values favor our technique). (b) rates the different techniques for window management employed by our system. In the proposed configuration, our system combines the newly introduced content-aware scaling with window relocation and cut-aways.*

6.3. Discussion

Our exploratory user study indicates that we met the design goals of self-organizing windows, because the users thought that self-organizing windows were useful, presented more important information, allowed accessing window content more quickly, were visually appealing, and reduced the overhead time for window management when compared to traditional systems. The participants indicated that the system was *'intuitive'*, produced compositions which *'could not be generate manually'*, and enabled them to access information *'more efficiently'*. On the other hand, participants found the overall on-screen motion slightly disturbing. Especially during the follow-up experiment participants demanded that the 'windows reached a stable state more quickly, to more efficiently interact with the windows'. We think this goal can be met by tuning the optimization parameters and enforcing movement constraints on the windows. See-through compositing received the most diverse feedback. For some participants see-through compositing was *'very fancy'* and *'appealing'*, while others found it *'irritating'*. One participant of the follow-up experiment noted that the 'cut-away regions were irritating if they were small'. We thus think that the see-through compositing requires improvement and should be offered as an optional feature.

In terms of application scenarios, the majority of the participants indicated that they would use the system for all of their desktop windows. The remaining participants thought that our technique was best applicable for small-to-medium-sized context windows that stay *'accessible and visible'* when they are made self-organizing. Three participants also wanted to define the area in which the self-organizing windows should reside, *e.g.*, a pool of context windows limited to a second monitor.

During the follow-up experiments one participant said that he would like the system to *'determine which windows were logically related and arrange these windows in a coherent fashion'*. This indicates that a combination with task management systems or rooms [HC86] seems to be a promising direction. Two of the four follow-up experiment participants noted, that they especially liked the way context windows, like search dialogs or floating toolsbars, were transformed to fit into the content of the main application. They also mentioned that it took them some time to get used to the fact that windows seemed to be *'floating'* on the desktop. At first, they were not able to predict how the final layout will look like when they moved windows, but they were *'most often in a good way surprised by the composition'*.

7. Conclusion and Future Work

We have demonstrated that treating windows as a collection of coherent pieces of information allows the use of advanced strategies for displaying important window content. We combined content-aware scaling with window relocation and see-through compositing to formulate an optimization problem for displaying the most important pieces of information for a user's current activity. Using our windowing manager, the available display space is used more efficiently because important window content is squeezed into information-free spaces. This technique allows the users to work with a main application while important related items from other applications are still available. An exploratory user study indicated that self-organizing windows can be more useful than traditional window management, while they still need some improvement in terms of on-screen motion and see-through compositing. Self-organizing windows greatly benefit from using current GPU technology to provide real-time frame rates while leaving the CPU's processing power available for desktop applications. Because the system works directly on the window textures, no access to the applications themselves is required and we were able to integrate it as a plugin for a widely used window manager.

With small adjustments to the algorithm, our system can emulate different techniques. Any combination of window relocation, uniform scaling, content-aware scaling, see-through compositing and opaque windows can be set up in our system and applied for real world tasks. We intend to compare these individual factors in a quantitative user study of information discovery, window switching, and visual search tasks. We will make self-organizing windows publicly available to obtain feedback from a wider user base.

8. Acknowledgments

The authors would like to thank E. Groeller and his team for the original suggestion of the 'Broken Windows' design concept. This work was funded in part by the Austrian Research Promotion Agency (FFG) BRIDGE 822716, as well as by the Austrian Science Fund (FWF): P23329.

References

[AS07] AVIDAN S., SHAMIR A.: Seam carving for content-aware image resizing. *ACM Trans. Graph.* (2007). 3

[BF00] BELL B. A., FEINER S. K.: Dynamic space management for user interfaces. In *Proc. UIST 2000* (2000), ACM, p. 239–248. 2, 8

[BNB01] BADROS G. J., NICHOLS J., BORNING A.: Scwm: An extensible Constraint-Enabled window manager. In *Proc. USENIX 2001* (2001), USENIX, p. 225–234. 2

[BSW08] BERNSTEIN M. S., SHRAGER J., WINOGRAD T.: Taskposé: exploring fluid boundaries in an associative window visualization. In *Proc. UIST 2009* (2008), ACM, p. 231–234. 2

[CG06] COTTING D., GROSS M.: Interactive environment-aware display bubbles. In *Proc. UIST 2006* (2006), ACM, p. 245–254. 2

[CKB09] COCKBURN A., KARLSON A., BEDERSON B. B.: A review of overview+detail, zooming, and focus+context interfaces. *CSUR 41* (2009), 2:1–2:31. 2, 3

[CR05] CHAPUIS O., ROUSSEL N.: Metisse is not a 3d desktop! In *Proc. UIST 2005* (2005), ACM, p. 13–22. 2

[EHRF08] ELMQVIST N., HENRY N., RICHE Y., FEKETE J.: Melange: space folding for multi-focus interaction. In *Proc. CHI 2008* (2008), ACM, p. 1333–1342. 2

[Gay86] GAYLIN K. B.: How are windows used? some notes on creating an empirically-based windowing benchmark task. In *Proc. CHI 1986* (1986), ACM, p. 96–100. 2

[GM04] GONZÁLEZ V. M., MARK G.: "Constant, constant, multi-tasking craziness": managing multiple working spheres. In *Proc. CHI 2004* (2004), ACM, p. 113–120. 1

[Gru01] GRUDIN J.: Partitioning digital worlds: focal and peripheral awareness in multiple monitor use. In *Proc. CHI 2001* (2001), ACM, p. 458–465. 2, 4

[GSCO06] GAL R., SORKINE O., COHEN-OR D.: Feature-aware texturing. In *Proceedings of Eurographics Symposium on Rendering* (2006), p. 297–303. 4

[GW04] GAJOS K., WELD D. S.: SUPPLE: automatically generating user interfaces. In *Proc. IUI 2004* (2004), IUI '04, ACM, p. 93–100. 2

[HC86] HENDERSON J., CARD S.: Rooms: the use of multiple virtual workspaces to reduce space contention in a window-based graphical user interface. *ACM Trans. Graph.* (1986), 211–243. 1, 2, 9

[HS04] HUTCHINGS D. R., STASKO J.: Shrinking window operations for expanding display space. In *Proc. AVI 2004* (2004), AVI '04, ACM, p. 350–353. 2

[HSM*04] HUTCHINGS D. R., SMITH G., MEYERS B., CZERWINSKI M., ROBERTSON G.: Display space usage and window management operation comparisons between single monitor and multiple monitor users. In *Proc. AVI 2004* (2004), ACM, p. 32–39. 1, 2

[IF04] ISHAK E. W., FEINER S. K.: Interacting with hidden content using content-aware free-space transparency. In *Proc. UIST 2004* (2004), ACM, p. 189–192. 2

[IF07] ISHAK E. W., FEINER S.: Content-aware layout. In *Proc. CHI EA 2007* (2007), ACM, p. 2459–2464. 2

[IKN98] ITTI L., KOCH C., NIEBUR E.: A model of Saliency-Based visual attention for rapid scene analysis. *PAMI 20* (1998), 1254–1259. 3, 5

[KS97] KANDOGAN E., SHNEIDERMAN B.: Elastic windows: evaluation of multi-window operations. In *Proc. CHI 1997* (1997), ACM, p. 250–257. 2

[LA94] LEUNG Y. K., APPERLEY M. D.: A review and taxonomy of distortion-oriented presentation techniques. *ACM Trans. TOCHI 1* (1994), 126–160. 2, 3

[MA99] MIAH T., ALTY J. L.: Vanishing windows: an approach to adaptive window management. *Knowledge-Based Systems 12*, 7 (1999), 381–389. 2

[MA00] MIAH T., ALTY J. L.: Vanishing windows–a technique for adaptive window management. *Interacting with Computers 12*, 4 (2000), 337–355. 2, 8

[MCRT06] MATTHEWS T., CZERWINSKI M., ROBERTSON G., TAN D.: Clipping lists and change borders: improving multitasking efficiency with peripheral information design. In *Proc. CHI 2006* (2006), ACM, p. 989–998. 2

[MRC91] MACKINLAY J. D., ROBERTSON G. G., CARD S. K.: The perspective wall: detail and context smoothly integrated. In *Proc. CHI 1991* (1991), ACM, p. 173–176. 2

[OSTS06] OLIVER N., SMITH G., THAKKAR C., SURENDRAN A. C.: SWISH: semantic analysis of window titles and switching history. In *Proc. IUI 2006* (2006), IUI '06, ACM, p. 194–201. 2

[RC07] RATTENBURY T., CANNY J.: CAAD: an automatic task support system. In *Proc. CHI 2007* (2007), ACM, p. 687–696. 2

[RCL*98] ROBERTSON G., CZERWINSKI M., LARSON K., ROBBINS D. C., THIEL D., VAN DANTZICH M.: Data mountain: using spatial memory for document management. In *Proc. UIST 1998* (1998), ACM, p. 153–162. 2

[RHC*04] ROBERTSON G., HORVITZ E., CZERWINSKI M., BAUDISCH P., HUTCHINGS D. R., MEYERS B., ROBBINS D., SMITH G.: Scalable fabric: flexible task management. In *Proc. AVI 2004* (2004), ACM, p. 85–89. 2

[RM93] ROBERTSON G. G., MACKINLAY J. D.: The document lens. In *Proc. UIST 1993* (1993), ACM, p. 101–108. 2

[RSA09] RUBINSTEIN M., SHAMIR A., AVIDAN S.: Multi-operator media retargeting. *ACM Trans. Graph.* (2009), 23:1–23:11. 3

[RvDR*00] ROBERTSON G., VAN DANTZICH M., ROBBINS D., CZERWINSKI M., HINCKLEY K., RISDEN K., THIEL D., GOROKHOVSKY V.: The task gallery: a 3D window manager. In *Proc. CHI 2000* (2000), ACM, p. 494–501. 2

[SB94] SARKAR M., BROWN M. H.: Graphical fisheye views. *Commun. ACM 37* (1994), 73–83. 2

[SCPR06] STUERZLINGER W., CHAPUIS O., PHILLIPS D., ROUSSEL N.: User interface facades: towards fully adaptable user interfaces. In *Proc. UIST 2006* (2006), ACM, p. 309–318. 2

[SSTR93] SARKAR M., SNIBBE S. S., TVERSKY O. J., REISS S. P.: Stretching the rubber sheet: a metaphor for viewing large layouts on small screens. In *Proc. UIST 1993* (1993), ACM, p. 81–91. 2, 3

[TCH*09] TAK S., COCKBURN A., HUMM K., AHLSTRÖM D., GUTWIN C., SCARR J.: Improving window switching interfaces. In *Proc. INTERACT 2009* (2009), Springer, p. 187–200. 1, 2

[TMC04] TAN D. S., MEYERS B., CZERWINSKI M.: WinCuts: manipulating arbitrary window regions for more effective use of screen space. In *Proc. CHI EA 2004* (2004), ACM. 2

[WGCO07] WOLF L., GUTTMANN M., COHEN-OR D.: Non-homogeneous content-driven video-retargeting. In *In ICCV07* (2007). 3

[WSGS11] WALDNER M., STEINBERGER M., GRASSET R., SCHMALSTIEG D.: Importance-Driven compositing window managment. In *Proc. CHI 2011* (2011), ACM. 2, 3, 6, 8

[WTSL08] WANG Y., TAI C., SORKINE O., LEE T.: Optimized scale-and-stretch for image resizing. *ACM Trans. Graph.* (2008), 118:1–118:8. 3, 4