

Flat Volume Control: Improving Usability by Hiding the Volume Control Hierarchy in the User Interface

Patrick Baudisch¹, John Pruitt², and Steve Ball³

¹Microsoft Research, ²Microsoft MSX, ³Microsoft eHome
One Microsoft Way, Redmond, WA 98052, USA
{baudisch, jpruitt, steveball}@microsoft.com

ABSTRACT

The hardware-inspired volume user interface model that is in use across all of today's operating systems is the source of several usability issues. One of them is that restoring the volume of a muted application can require an inappropriately long troubleshooting process: in addition to manipulating the application's volume and mute controls, users may also have to visit the system's volume control panel to find and adjust additional controls there. The "flat" volume control model presented in this paper eliminates this and other problems by hiding the hardware-oriented volume model from the user. Using the flat model, users use one slider per application to indicate how loud they want the respective applications to play; the slider then internally adjusts all hardware volume variables necessary to obtain the requested output. By offering a single point of control for each application, the flat model simplifies controlling application volume and restoring muted applications. In our studies, participants completed all four volume control and mixing tasks faster and with less error when using the flat model than when using the existing hardware-oriented volume control model. Participants also indicated a subjective preference for the flat model over the existing model.

Categories & Subject Descriptors: H5.2 [Information interfaces and presentation]: User Interfaces. - Graphical user interfaces.

General Terms: Human Factors, Design.

Keywords: Audio, sound, volume control, user interface.

INTRODUCTION

Imagine the following scenario. In the middle of a presentation, the presenter tries to play a video clip. When hitting 'play' on the software video player, the video starts playing but the audio remains silent. In order to fix the problem, the presenter cranks up the volume slider in the video player, but without success. When realizing that it may not be the player causing the problem, the presenter opens the system's volume control panel and finds the state shown in Figure 1. The presenter notices that the "master volume" slider (labeled "Volume Control", 5) is set to zero, which would explain why the sound did not play. The presenter then cranks the slider all the way up, but still, nothing. After examining the corresponding mute checkbox (6) and the

state of the wave volume slider (3) the presenter notices that the wave channel is muted (4). Unchecking this "Mute" checkbox finally allows the audio to play (although the audio now plays much louder than intended, as the application volume and wave volume sliders were set to their maximum values during the troubleshooting process.)

The problem we are addressing in this paper is that this process takes more time and effort than necessary.

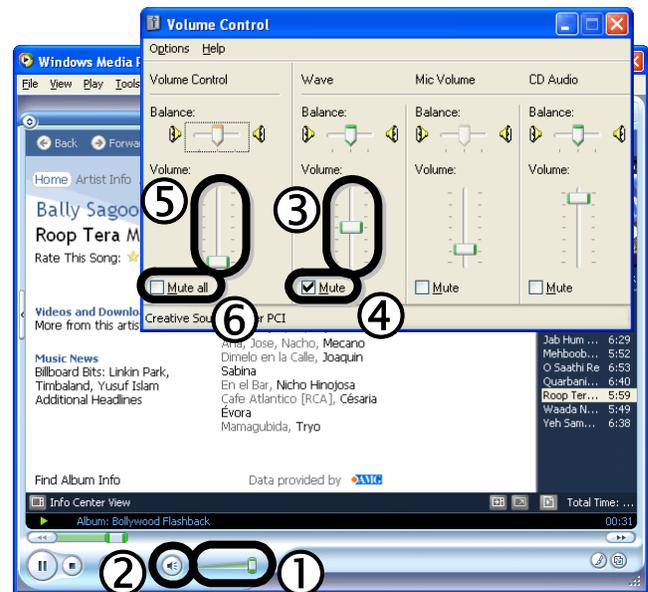


Figure 1: Current volume control model: application audio output is only active when its volume slider (1) and wave and master sliders in the control panel (3, 5) are set to non-zero values and the three mute check boxes (2, 4, 6) are unchecked.

A look under the hood

We claim that the described problem is caused by the fact that existing volume control interfaces expose the volume control structure of the computer's sound card to the user. As shown in Figure 2, the volume variables in today's systems form a hierarchy. Before a sound produced by an application reaches the speakers, it is affected by all sliders and mute widgets in the path between that application and the speakers. The actual *loudness* of an application (we will use the term "loudness" to describe the final audio level that is sent to the speakers and "volume" for internal volume variables) is the product of all volume variables along the path¹. Determining how loud an application actually

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

CHI 2004, April 24–29, 2004, Vienna, Austria.

¹ In addition, there is often an analog knob on the speakers, which is typically not controllable via software on today's average PC.

plays thus requires users to read all volume variables along the path and mentally multiply them.

An application is muted whenever at least one of the multiplied volume variables along its path is zero, i.e., a slider set to zero or a checked mute button, no matter what the state of the other variables in the signal path. Detecting that therefore requires checking all these variables. Restoring a muted application requires restoring all muted volume variables along the path. These cases may also require users to access the system's volume control panel.

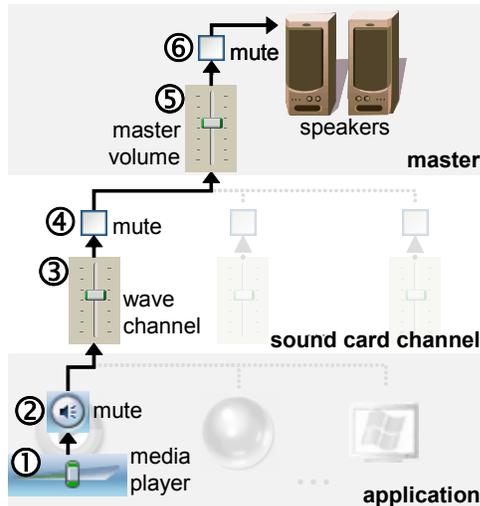


Figure 2: In the existing, hierarchical volume control model (here MS Windows XP™), application sound volume is reduced at up to six points along the signal path (plus the volume knob at the speakers).

While we will use muting as our main scenario throughout the paper, there are other volume control tasks that are complicated by the hierarchical model. The hierarchical model makes it difficult to set an application to a known loudness, as this can require changing multiple variables along the path. Also the task of making an application louder can become complicated; in cases where the application volume maxes out, users need to increase the master volume instead, which in turn has a side effect on the loudness of other applications. We will return to these scenarios later in this paper and also in the user study section.

Even though some of today's operating systems use a simplified interface, e.g., one that hides the sound card channel layer (e.g. Apple Macintosh), they all have the notion of a master volume. Thus, the described problem exists across platforms.

What to do?

This hierarchical model and the resulting multiplicative volume model have desirable properties from an engineering point of view. For example, they offer a wide dynamic range and, properly setup, can maximize the signal-to-noise ratio. However, exposing this architecture to computer users results in complexity that, over time, can lead to additional user effort and error conditions. The professional sound mixing equipment that first used this type of hierarchical volume control model was designed by and for audio

experts, but today's typical computer users do not typically fall into this category.

We therefore propose a new volume control interface model—one that hides the internal hierarchical structure of the sound card from the user. As we show in this paper, this allows users to monitor and control the loudness of applications more efficiently and especially solves the muting problem. At the same time, the proposed model matches and sometimes outperforms the sound quality the traditional model offers. We begin by presenting a walk-through of the flat model and its user interface. Then we briefly look at the related work, followed by details about design and implementation of the flat volume control model and methods for handling legacy issues. Finally, we present the results of the studies we conducted and conclude with a discussion of our findings.

FLAT VOLUME CONTROL

The main benefit of the flat model is that it manages the volume hierarchy for the user. This is realized by changing the semantics of all volume sliders in the system to solely represent *loudness*. Under the traditional model the volume slider in Windows Media Player defined a *single link* in Media Player's volume path; in our redesign as a loudness slider it now defines *how loud* Media Player plays, i.e. the value of Media Player's volume path *as a whole*. By manipulating the loudness slider, users indicate how loud they want Media Player to play, but without defining how this is supposed to be accomplished. It is the loudness slider itself that then determines the best way of realizing the requested loudness in terms of hardware volume variables and that makes the necessary changes. This delegation reduces the user's load and obtains equal or better audio quality, as the slider automatically optimizes the system's signal-to-noise ratio. We will describe the algorithms that accomplish this in detail in the implementation section.

If we redrew the diagram from Figure 2 for the flat model, we would see that the sound card channel and master layers are gone; application loudness widgets are now directly connected to the speakers. The hierarchy has been replaced with a flat structure—thus the name of our approach.

The flat volume control panel

When switching to the flat model, the semantics of all volume control widgets across the system change, including those located in the volume control panel. This requires some changes in the control panel's user interface. Figure 3 shows a screenshot of our volume control panel prototype. This control panel allows users to perform three types of interactions. First, sliders, one per application, allow users to adjust the *loudness* of the respective application. Second, the "all applications" thumbwheel allows users to adjust the loudness of all applications at once. "Spinning" the wheel makes all application sliders move up and down, as illustrated by Figure 4. Third, clicking the "mute all" pushbutton visibly brings all sliders to zero and makes the mute button change its label to "restore all". Clicking the mute button again restores the values of all sliders to their previous states.



Figure 3: The flat volume control panel. Application sliders represent the loudness of that application. Thumbwheel and mute affect all application sliders.

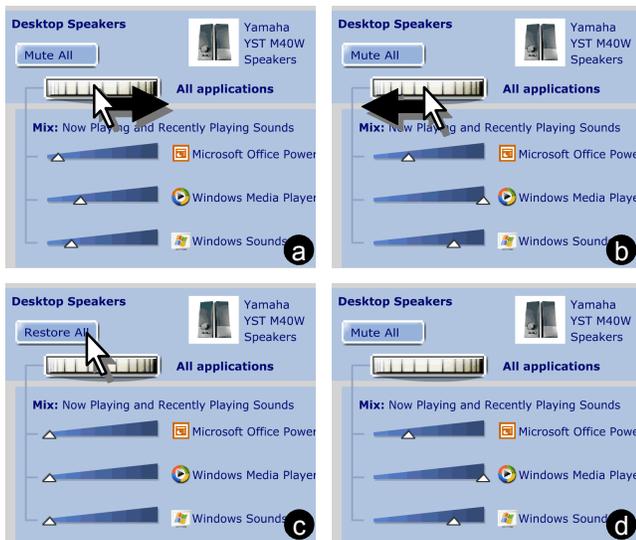


Figure 4: Adjusting the volume of all applications at once using the thumbwheel. (a→b) Dragging the wheel to the right makes all sliders go up *proportionally*. When the first slider maxes out all sliders stop; this preserves the volume mix. (b→c) dragging it to the left makes all sliders go down proportionally until they all hit zero. (c→d) Clicking the “restore all” push button restores to the last non-zero setting.

It is essential to the flat user interface, that widgets cannot only be manipulated directly, but that they also track the value of the variable they represent and update themselves if that value changes. This is necessary because the loudness represented by one widget can be affected by another widget. Clicking “mute all”, for example, can change the value of all other loudness variables in the system. Using

widgets that continuously check the variable they represent assures that the state of the interface remains consistent. As an example, when the user operates a loudness slider in an application, the corresponding slider in the control panel moves in sync and vice versa. Moving the thumbwheel has an impact on the loudness of many applications and consequently, all sliders representing application volume move—in the control panel (Figure 4a→b), as well as in the applications. When an application’s mute button is clicked, the loudness slider of that application jumps to zero and when mute is clicked again the slider restores itself. Whenever an application ends up having zero loudness, its mute button reflects that by changing its state accordingly. If all applications have zero loudness, e.g., because the thumbwheel was spun all the way down (Figure 4b→c), the main mute changes its state to reflect that. Any way of unmuting the system, whether it is hitting “restore all” (Figure 4c→d) or dragging an application loudness slider up, restores the mute button’s “mute all” face.

The flat model solves the muting problem

In the traditional volume control model, the loudness of any application may be reduced by other volume variables, such as the master volume. Sliders therefore only mean “the loudness of this application is at most x ”. In the flat model, however, sliders mean “the loudness of this application *is* x ”. In the state shown in Figure 3, for example, *3D Pinball* is playing at about 80% loudness, *Windows Media Player* at ~40%. One of the main benefits of this paradigm switch is that it solves the muting problem. Detecting that an application is muted becomes straightforward; an application is muted if and only if its loudness slider points to zero. Restoring a muted application is equally straightforward. Any muted application can be restored by dragging the application slider up—the slider will adjust all volume variables necessary. As a result, the need to access the control panel and to check multiple widgets is eliminated.

The flat model also addresses the two other scenarios mentioned earlier. First, since loudness sliders can increase channel and master volume variables when necessary, users can now always access the full possible range of output loudness from inside their applications. Second, the flat model establishes a fixed mapping between slider state and loudness, which allows users to set an application to any known loudness, such as “the loudness for giving slide presentations in this conference room”, by setting its slider to a remembered position.

RELATED WORK

Sound is in wide use in human-computer interaction. Sound allows for eyes free interactions. Since users can detect sounds rapidly, sound was found highly effective for monitoring applications [11] as well as various types of notification [20]. In other areas, sound has been used to make VR environments [14] and reading more immersive [2], and to help improve the usability of devices, e.g., by helping users navigate hierarchical structures [5], acquire buttons on small screen devices [4], or recognize the functions of products [12].

Along with pitch, location, and semantic context, sound *volume* is one of the main cues that help users distinguish sound sources [19]. Controlling volume therefore plays a major role in audio-only media spaces [18], where volume has been used to denote proximity between participants in conversations [1]. Various interface strategies have been suggested for controlling volume, such as hand gestures [7, 10], bar code readers [13], or physical widgets connected to a computer [8]. A broad interest in controlling volume in a convenient way has created a market for such products (e.g. [9]).

The widgets deployed in the volume control interfaces presented in this paper have been studied in various contexts. Interaction techniques inspired by a paint metaphor have been proposed as means for efficiently manipulating larger numbers of sliders [3]. The design of mute buttons is subject to the discussion of how to visualize the state of a button [15, 6]. Thumbwheel widgets have been used to enter variables on an infinite range, e.g. in flight simulation [16] and 3D viewers (e.g., examinerViewer, www.sgi.com).

THE DESIGN OF THE FLAT VOLUME CONTROL PANEL

In this section, we take a closer look at the design shown in Figure 3 and point out design alternatives. Before we focus on the widgets that form the interface of the flat architecture, we give a quick overview of other aspects of the flat control panel (Figure 3), i.e., the changes that make it different from the Windows XP control panel shown in Figure 2. These changes are independent of the flat concept, so they may also be applied to a non-flat control panel or removed from the flat control panel altogether.

General design changes

First, the control panel shown in Figure 3 does not expose sound card channels, such as “wave”. The primary reason for that is that today virtually all PC sounds go through the wave channel, so that all other channels have become obsolete. Hiding channels reduces clutter and brings this dialog up to par with the Apple Macintosh, the designers of which chose not to expose sound card channels in the first place. Note that the flat volume control concept works with volume hierarchies of any depth, so it remains applicable even if sound card channel volume was exposed.

As an alternative to sound card channels, and unlike the Windows XP control panel, the flat panel lists “Now Playing and Recently Playing Sounds”, i.e., applications. This provides users with direct access to application loudness even including applications that do not offer volume control in the application itself, such as the *3D Pinball* game in Figure 3. The implementation section of this paper explains how this is implemented using so-called ‘shims’.

Finally, the control panel was uncluttered by moving application-specific mute buttons and all balance sliders into an “advanced mode” panel. Layout and graphical design were changed as well.

The thumbwheel and the mute button visuals

Figure 5a shows a design alternative we explored, called *flood mark* design. The input capabilities of this widget are

equivalent to the thumbwheel shown earlier. However, this design offers additional functionality by providing a *flood mark*, a vertical line that always remains in touch with the knob of the loudest application. The flood mark gives users a visual indication for the current overall loudness of their system. The attempt to drag an application slider beyond the flood mark makes the flood mark slider go up in parallel (Figure 5a→b, the topmost of the three sliders was dragged up); lowering the loudest application makes the flood mark slider follow until it hits the knob of another application slider. Adjusting the flood mark slider itself scales all application sliders proportionally (Figure 5b→c).

Despite the flood mark design’s potential for contributing to a more powerful interface, we chose to pursue the thumbwheel design when early usability testing indicated that the additional information provided by the flood mark made this design slower to learn and read than the thumbwheel design. Also, some users who had extensive experience with the traditional volume control panel falsely identified the slider as a traditional master volume slider, which caused them to read application loudness incorrectly.

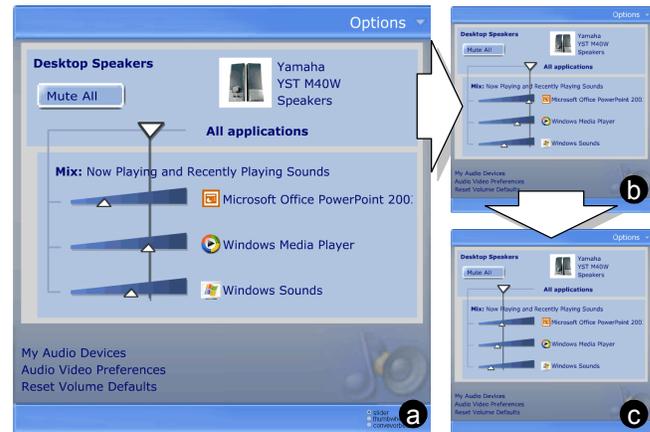


Figure 5: The ‘flood-mark’ on the control panel design allows users to read how loud their system is.

The point behind using a thumbwheel is that thumbwheels allow users to manipulate a variable without the state of that variable being exposed; for our purposes, the thumbwheel is basically a slider with the knob being deliberately hidden. In our design, this reduces the cognitive load for the user and avoids the risk of misinterpretation. The flat model provides visual feedback by moving application sliders instead.

Our initial concerns that Windows users would be unfamiliar with thumbwheels went away during our usability study (see the respective section). Our participants seemed to be fairly familiar with thumbwheels, which might be explained by the fact that many consumer devices, such as cell phones, Sony PDAs, and Microsoft mice utilize them.

The visual design motivation of the “Mute all” button is similar to the thumbwheel. Since all relevant loudness information is contained in the application sliders, there is no need for users to read the state of the mute button. In order to discourage users from reading the button, we gave it the

visual appeal of a push button—a widget that, unlike checkboxes, is generally used as a pure input widget [15].

IMPLEMENTATION AND ALGORITHMS

Adjusting the loudness of a single application can require the flat system to apply changes to a larger number of volume variables in the underlying volume variable hierarchy. Imagine, for example, that, while the overall system is muted, a user unmutes an application by dragging its application slider up. This requires the flat system to deactivate the main mute, but to prevent other applications from starting to play as well the flat system also needs to mute all other applications. In this section, we describe the system design and algorithms that accomplish this.

Components: In order to be able to make global adjustments, all loudness computation is done from within a central program that maintains back-and-forth communication with all loudness widgets, whether in the volume control panel or an application. This communication is built into a customized loudness slider; applications that use that slider are automatically loudness-enabled.

Data structures: All computation is based on two main data structures, i.e., hardware-oriented volume and flat loudness. Hardware-oriented volume is a tree structure that contains the variables shown in Figure 2; mute is represented as variables that take on the values 0% and 100%. Flat loudness uses the same hierarchical structure, but different semantics. Leaves in the tree represent application loudness; nodes are defined as the maximum of their children. Leaves and nodes determine the value of the interface widgets each of them is associated with. By definition, a mute widget is muted if the associated variable is zero.

When the flat system is launched, it reads the hardware volume state and converts it to flat format. From then on, volume is managed in flat format. Whenever loudness is adjusted, the flat format is translated to hardware-oriented format and sent to the system's audio API in order to make the changes audible.

Algorithms: Hardware-oriented volume is converted to flat loudness by multiplying node values by their parent's value in a top-down traversal and then setting all nodes to the maximum of the values of their child nodes in a bottom-up traversal. The back conversion is done by dividing all node values by their parent's value in a top-down traversal. When the user adjusts the loudness of some node, the flat structure preserves its consistency by repairing the path from the node to the root, as well as the node's sub tree.

Note that this algorithm explains all the behavior described in the interface walkthrough, such as the tight coupling between mute and sliders or the behavior of the flood mark, which is simply coupled to master volume. The hardware-oriented volume states created by this algorithm have the following properties. The master volume is always minimized, the wave channel is set to a constant 100%, all other channels are muted, and application volume variables are always maximized. This optimizes the system's signal-to-noise ratio—better than a typical user might configure.

The current status of our implementation is that the volume control panel shown in Figure 7 is implemented in Windows native code, while the add-ons required for the flat volume control model are implemented as prototype code (Macromedia Flash) to allow for more efficient experimentation with different algorithms and interfaces.

DEALING WITH LEGACY ISSUES

Implementing the full extent of the flat volume control model requires the participation of applications and volume control panel. This would suggest that introducing the flat model would face a huge hurdle, as it is unlikely that a user would upgrade the operating system and all applications at the same time. Fortunately, this hurdle can be overcome by the use of application 'shims' and 'flood mark sliders'.

Legacy applications in a flat system

The handling of legacy applications depends on their volume control capabilities. Applications that produce sound, but offer no volume control interface are particularly easy, as the flat system can simply manage loudness for them. Legacy applications *with* internal volume control interface need to be kept in sync with the flat system. While the flat system cannot control user interface elements inside the legacy application (volume control widget inside the application may therefore at times reflect an incorrect loudness value), the flat system can still control the application's volume and thus apply the flat model. For that purpose, the flat system uses so-called *application compatibility shims*. Shims are callback functions inserted into the get and set volume functions that applications call. While shims are active, legacy applications effectively communicate with the flat system rather than the sound hardware, which allows the flat system to manage loudness for the application. This also allows the user to adjust the application's loudness through the control panel.

Flat applications in legacy operating system

The opposite case, a flat application running in a legacy system, is of particular interest, as it allows deploying the presented concepts on a per-application basis—an easier step than adoption on an operating system-wide scale.

The slider inside a flat-enabled application always represents loudness, also when running in a legacy operating system. If necessary, the slider itself now increases the system's master volume and mute to achieve the requested loudness. The only difference compared to a full flat implementation is that the flat application cannot establish the shim mechanism and thus cannot prevent other applications from getting louder in that case. In order to warn users of this side effect, sliders may optionally display a little horizontal line across the slider (Figure 6a) to indicate: "Dragging the knob beyond this line will increase the loudness of all other applications." This line is called flood mark—vaguely related to the flood mark control panel design presented earlier (but not subject to its usability issues). Dragging the slider knob beyond the flood mark drags the flood mark with it (Figure 6b→c). Dragging the knob back down leaves the master volume unchanged (Figure 6c→d). This

prevents reducing the loudness of one application from reducing the loudness of or muting others. Flood marks can be complemented with a handle, shown as a small rectangle attached to the flood mark. By dragging the flood mark users adjust the master volume directly (Figure 6d→e).

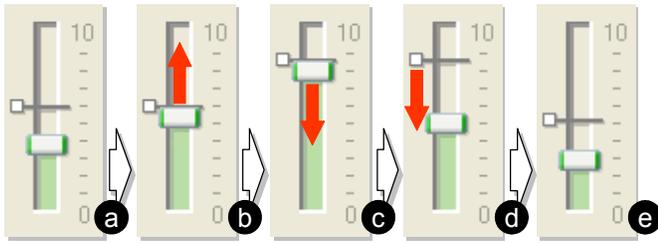


Figure 6: (a) A flood mark slider consists of a slider and a flood mark that warns users “exceeding this loudness will increase the loudness of other apps”, (b-e) walkthrough.

PILOT STUDY

Before conducting our actual user study, we carried out a pilot study to get a first impression of the learnability and usability of the flat volume control concept and our control panel prototype and to identify potential usability bottlenecks. Moreover, we wanted to get a general sense of whether the flat volume concept combined with the other design changes described earlier were perceived as improvements over the volume control panel of the currently most widely used operating system. We therefore compared our flat volume prototype with the volume control interface of Microsoft Windows XP.

The *XP interface* was implemented as a Windows XP system running three sound sources—Windows media player, 3D Pinball, and Windows new message notifications. The *flat interface* offered identical functionality, but its sliders represented loudness and it used the control panel shown in Figure 3 in a desktop with matching visuals (Figure 7 shows a screenshot of that desktop, but with a different control panel). Desktop, control panel, and all relevant application functionality were simulated using Macromedia Flash. Both interfaces were run on a PC running Windows XP on a 20” LCD screen at 1024x768 pixel resolution.

Seven participants were recruited from the larger Puget Sound area. All were fairly experienced Windows users and had used their PCs for sound and audio purposes, e.g., playing music via ripped format, listening to internet radio, etc. All participants were familiar with the standard Windows XP volume control panel.

During the study, participants received verbal instructions via a voice connection and we observed the participant’s actions from behind a two-way mirror. Participants received no training. We started by asking participants to make the PC, which was currently muted, play sounds. Then we asked them to increase the volume of the CD. Next, we called them on the phone, on which we hinted that we couldn’t hear them very well and if necessary prompted them to turn down the volume. After the phone call ended we asked them to restore the volume. We then

asked them to turn all volume up or down a little. After participants had completed this walkthrough, we encouraged them to explore the volume control panel and application volume controls at their own pace. Then we repeated the same sequence with the other interface. At the end of the study, we interviewed participants and assessed their comprehension of the involved interface elements. The study lasted about 30 minutes.

Results

All seven participants were able to complete all tasks involved in the walkthrough with either interface. It was observed that participants had no trouble operating the flat interface, despite their lack of familiarity with this interface style. The flat interface received a number of positive comments. Most relevant for this study, all participants recognized and liked the functionality of the thumbwheel and the fact that it raised or lowered the volume sliders of all applications in unison. When explicitly asked whether the fact that sliders moved at different speeds (this was caused by the *proportional* slider motion, see Figure 4a→b) would be strange, participants disagreed and stated that this behavior was logical and intuitive. The additional changes were well received as well. All seven participants liked the fact that the flat control panel listed applications instead of the sound card channels. The application volume sliders were considered useful for adjusting the volume of the Pinball game, which by itself offered no volume control.

The main shortcoming of the flat interface mentioned by the participants was the discoverability of the thumbwheel. All participants discovered the functionality of the wheel, but 6 out of 7 did not touch it during their initial exploration of the control panel; in most cases, not until the part of the experiment where we asked them to “turn everything up a little”. In subsequent prototypes, we have addressed this issue by adding a brief 10-pixel back and forth rotation animation to the wheel when the control panel opens.

All seven participants expressed a strong preference for the flat interface. While this did not clearly tell us which aspect of the new control panel was responsible for the preference, it indicated that the design of our control panel as a whole was on the right track.

USER STUDY

With the improved thumbwheel design, we conducted a formal user study to isolate and evaluate only the flat volume control aspects. We did this by comparing the flat interface with a system that was visually and functionally identical, but used the traditional hierarchical volume control system instead (Figure 7).

Our hypothesis was that participants would troubleshoot volume control, restore volume, and increase volume faster when using the flat interface. This should result in higher subjective satisfaction for that interface. However, we expected the flat interface to score lower in terms of learnability, as some participants would be already familiar with the concept behind the control interface while the flat concept would be new to them.

Interfaces: The *flat interface* used in this study was identical to the flat interface in the pilot study reported in the previous section, but it used the improved thumbwheel. The *control interface* (Figure 7) was identical to the flat interface except that it implemented the traditional hierarchical volume control model. Its sliders represented volume not loudness and its control panel offered a master volume slider and a main mute checkbox instead of the thumbwheel and mute push button offered by the flat interface. The control interface’s control panel also exposed per-application mute checkboxes. This was necessary to allow users to notice and unmute applications that had been muted from within the application. The flat interface did not need to expose these, as the flat model manages mute indirectly through the loudness widgets. Besides this, both interfaces used the same visuals (Figure 7 vs. Figure 3).

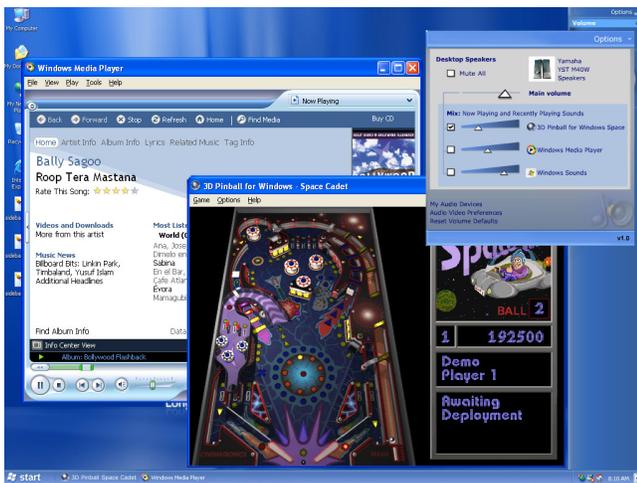


Figure 7: The experimental environment used in the pilot study and the user study, here with the volume control panel of the control interface.

Participants: Seven participants were recruited from the larger Puget Sound area. Each participant had access to a Windows PC in their place of residence and used it at least once a month. Actual computer tenure and level of expertise varied from “casual user” to “skilled daily PC user”. All participants indicated that they had some experience listening to sounds or music on the PC and all had adjusted the sound level of their PC multiple times in the past either through software or hardware controls.

Procedure: Each interface was evaluated in two steps. To assess the learnability of the interfaces, participants first completed the same walkthrough procedure that we had used in the pilot study. At the end of this part, participants were quizzed about the interface’s functionality. Then participants completed four qualitative tasks as described below and filled out a questionnaire. Then they repeated the same procedure with the other interface. Interface and task order were counterbalanced. The entire study lasted about 1 hour.

Tasks: In all tasks, trials started by participants clicking a button labeled “start”, adjusting volume either in media

player or the control panel, and completing the trial by hitting the space key. Each task consisted of four training and eight timed trials.

1. *Unmute task.* The participant’s task was to make Media Player’s audio play from a muted state. The Media Player was muted by its application volume being zero or muted, the master volume being zero or muted, or any combination thereof.

2. *Restore task.* Participants found the volume/loudness settings initialized to a randomly chosen reference setting making Media Player play at reference loudness. The participants task was to hit a button labeled “randomize” that changed the volume settings and then to bring back the loudness of Media Player to this trial’s reference loudness.

3. *Maximize-one task.* The participant’s task was to maximize the loudness of Media Player, while affecting the volume of all other applications as little as possible.

4. *Maximize-all task.* The participant’s task was to make the PC play as loud as possible, while affecting the relative mix between applications as little as possible.

Results

Task completion time: Confirming our first hypothesis, subjects achieved significantly better task completion times in all four tasks when using the flat interface (Figure 8). All differences between interfaces were significant at $p < 0.01$.

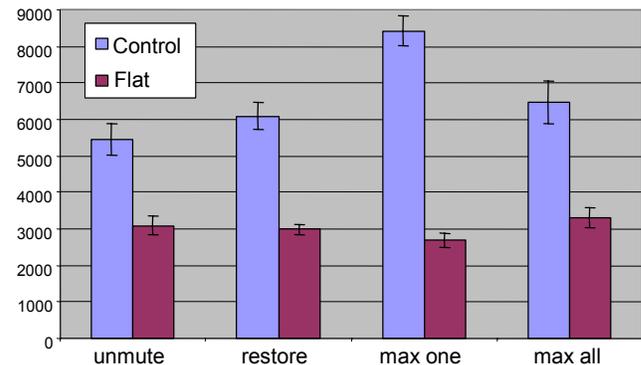


Figure 8: Average task completion times in milliseconds (and Std. Error of Mean)

A 2 (flat v. control) x 4 (4 tasks) repeated measures Analysis of Variance (RM ANOVA) was carried out on the task time data. We observed a significant main effect for Interface, $F(1,18)=23.8$, $p < 0.01$. However, there was no significant effect for Task or the interaction. The control interface was consistently slower than the flat interface for input across all tasks.

Error rate: For each of the four tasks, participants produced lower error rates when using the flat interface than when using the control interface (Table 1). Due to the different nature of the four tasks, error was computed differently for each task. *Unmute task:* number of cases where Media Player was not successfully unmuted. *Restore task:* percentage the entered loudness was off with respect to reference loudness. *Max one:* percentage the loudness of the other two applications was off from their starting loud-

ness. *Max all*: percentage the loudness of the other two applications was off from the loudness that would have preserved the mix. Paired t-tests for interface showed significant effects for restore, $t(6)=3.70$, $p<0.01$ and max one $t(6)=4.39$, $p<0.01$ and borderline significance for max all, $t(6)=2.37$, $p<0.06$.

task	flat interface	control interface
unmute	2 of 64	5 of 64
restore	4.4% (1.1%)	11.8% (2.6%)
max one	0.4% (0.4%)	27.7% (6.1%)
max all	1.0% (0.7%)	8.4% (2.7%)

Table 1: Error rate (and Std. Error of Mean)

Subjective satisfaction: On a five-point Likert scale (1 = strongly disagree, 5 = strongly agree), average ratings on learnability (“self explanatory”, “simple”, and “clear”) ranged between 3.9 and 4.3 out of five for both interfaces. “Making changes was simple” was rated 4.6 out of 5 for the flat interface vs. 3.6 for control and “Making changes was efficient” was rated 4.3 for flat vs. 3.4 for control, but none of the differences were statistically significant. When asked, 5 of the 7 participants indicated specific trouble on certain tasks when using the control interface. The restore task seemed particularly hard with this interface. Participants reported no problems for the flat interface.

In the final ranking, the majority of subjects (5/7) indicated a preference for the flat interface over the control.

DISCUSSION AND CONCLUSIONS

Overall, our user study confirmed our hypotheses and provided the evidence that the flat volume control model leads to actual time savings when troubleshooting audio, as well as subjective preference over the existing hierarchical volume control model. While these time savings are a clear indication for the usefulness of the flat system, the results of the walkthrough part of the study are at least as important. Unlike the quantitative part, which required participants to solve the same volume control problem repeatedly and thus measured how long it takes users to *execute* a troubleshooting interaction, the walkthrough required participants to *solve* volume control problems. The virtual absence of complications during this part of the study indicates the probably most valuable benefit of the flat model: the flat model prevents users’ volume control needs from becoming problems in the first place.

As future work, we plan to explore in how far the concepts described in this paper transfer to other application areas, such as sound studio equipment or even non-audio applications, such as multi-stage gamma corrections.

Acknowledgements

Thanks to Ed Cutrell and Mary Czerwinski for their comments on this paper. Thanks to Frank Yerrace, Annette Crowley, Larry Osterman, Frank Wong, and Jeremy Knudsen for their contribution to the volume control project.

REFERENCES

1. Aoki, P. et al. The mad hatter's cocktail party: a social mobile audio space supporting multiple simultaneous conversations. In *Proc CHI'03*, pp. 425–432.
2. Back, M., Cohen, J., Gold, R., Harrison, S., and Minneman, S. Listen reader: an electronically augmented paper-based book. In *Proc CHI'01*, pp. 23–29.
3. Baudisch, P. Don't Click, Paint! Using Toggle Maps to Manipulate Sets of Toggle Switches. In *Proc. UIST'98*, pp. 65–66.
4. Brewster, S. Overcoming the Lack of Screen Space on Mobile Computers. *Personal and Ubiquitous Computing* 6(3):188–205, May 2002.
5. Brewster, S. Using non-speech sounds to provide navigation cues. *TOCHI* 5(3):224–259, Sept. 1998.
6. Carr, D.A. Specification of Interface Interaction Objects. In *Proc. CHI'94*, pp. 372–378.
7. Freeman, W.T. and Weissman, C.D. Television Control by Hand Gesture. In *IEEE Intl. Workshop on Automatic Face and Gesture Recognition*, 1995.
8. Greenberg, S. and Boyle, M. Customizable physical interfaces for interacting with conventional applications. In *Proc. UIST'02*, pp. 31–40.
9. <http://www.griffintechology.com/products/powermate>
10. Kohle, M. Special Topics of Gesture Recognition Applied in Intelligent Home Environments. *Lecture Notes in Computer Science* 1371:285–296, 1998.
11. Kramer, G. An Introduction to auditory displays. In *Auditory Display: Sonification, Audification, and Auditory Interfaces*. Addison-Wesley 1994, pp 1–78.
12. Lee, C.H., Kim, S., Chae, C.S., Chung, K.H. Sound: an emotional element of interactions a case study of a microwave oven. In *Proc. DIS'00*, pp. 174–182.
13. Masui, T., and Siio, I. Real-World Graphical User Interfaces. In *Proc. HUC'00*, pp. 72–84.
14. Naef, M., Staadt, O., and Gross, M. Spatialized audio rendering for immersive virtual environments. In *Proc. VR Software and Technology*, pp. 55–72.
15. Plaisant, C., Wallace, D. Touchscreen Toggle Switches: Push or slide? Design issues and usability study. University of Maryland, *CS-Tech Report 2557*, 1990.
16. Rushby, J. Analyzing cockpit interfaces using formal methods. In H. Bowman (editor), *Elsevier Electronic Notes in Theoretical Computer Science* 43, Oct. 2000.
17. Shneiderman, B., *Designing the User Interface: Strategies for effective human-computer interaction*, Third edition, Reading MA: Addison-Wesley, 1998
18. Singer, A., Hindus, D., Stifelman, L. and White, S. Tangible progress: less is more in Somewire audio spaces. In *Proc CHI'99*, pp 104–111.
19. Wickens, C.D. and Hollands, J.G. *Engineering Psychology and Human Performance*. Third Edition, Prentice Hall, NJ, 2000.
20. Jones, D. The cognitive psychology of auditory distraction. The 1997 BPS Broadbent Lecture. *British Journal of Psychology*, Vol 90(2), May 1999, 167–187.