Audio Engineering Society

Convention Paper

Presented at the 131st Convention
2011 October 20–23 New York, USA

Ray-traced Graphical User Interfaces for Audio Effect Plug-ins

Benjamin Doursout[1], and Jörn Loviscach[2]

[1] *ESIEA – École supérieure d'informatique, électronique, automatique, 53000 Laval, France*

[2] *Fachhochschule Bielefeld (University of Applied Sciences), 33602 Bielefeld, Germany*

Correspondence should be addressed to Jörn Loviscach (joern.loviscach@fh-bielefeld.de)

ABSTRACT

On the computer, effects and software-based music synthesizers are often represented using graphical interfaces that mimic analog equipment almost photorealistically. These representations are, however, limited to a fixed perspective and do not include more advanced visual effects such as polished chrome. Leveraging the flexibility of the audio plug-in programming interface, we have created software that equips a broad class of synthesis and effect plug-ins with interactive, ray-traced 3D replicas of their user interface. These 3D models are built automatically through an automated analysis of each plug-ins' standard 2D interface. Our experiments show that interactive frame rates can be achieved even with low-end graphics cards. The methods presented may also be used for an automatic analysis of settings and for realistic interactive simulations in the design phase of hardware controls.

1. INTRODUCTION

The graphical user interfaces of most current audio software provide graphical representations of hardware controls. This work looks into a method to further improve visual realism: rendering 3D models of control panels through a ray-tracer that operates at interactive speed thanks to the computational power of today's graphics cards. As opposed to standard z-buffer rasterization as used in almost all computer games and virtual reality applications, ray-tracing enables displaying resplendent control surfaces made of shiny materials, which are typical for high-end audio equipment.

The method may be used for actual man-machine interaction in the digital studio. It may also be helpful for the design of interactive virtual prototypes of audio equipment and music synthesizers such as [4].

We apply image processing to build 3D models of the user interfaces of existing plug-ins. This approach may also be applicable to other problems of automated analysis of plug-ins, for instance for testing purposes during software development.

This paper is structured as follows: Section 2 explains the wrapper solution to extend a given standard plug-in. Section 3 introduces the image analysis of the plug-in's standard user interface. The results of this analysis are used to create a 3D model, as described in Section 4, which is rendered through ray-tracing, see Section 5. Section 6 covers the user's interaction with the ray-traced interface; Section 7 presents a number of results; and Section 8 provides ideas for future extensions and applications.

2. PLUG-IN WRAPPER

To support a broad variety of existing plug-ins and of existing digital audio workstation software, we have developed a wrapper, see Fig. 1, that

- masquerades as a host application to a plug-in,
- masquerades as a plug-in to the host, i. e., the digital audio workstation (DAW),
- employs the Internet protocol (TCP/IP) to communicate with the novel user interface and initially also with the software used for image processing.

The application programming interface (API) supported for the plug-on and the host is VST 2.4 [2]. The communication via TCP/IP enables receiving data from and sending data to any other process running on the same computer—or even running on a different computer, so that one may even compute the visuals on a remote computer. The data are packed into a proprietary UDP datagram format.

On startup, the wrapper presents a dialog to select the plug-in that it should be wrapping. This is the only time the user makes contact with the wrapper. When running, the wrapper lets the messages from the host pass through to the plug-in and vice versa. The wrapper recognizes parameter change commands from either side—that is, the user interface of the plug-in as well as automation from the host—and transmits them as UDP datagrams to attached software. In addition, the wrapper listens for

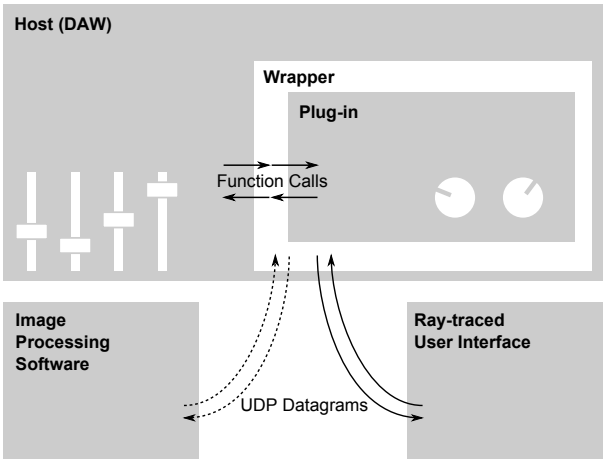

Fig. 1: The wrapper eavesdrops on messages sent as function calls from the host to the plug-in and vice versa. In addition, it uses the Internet protocol to communicate with software external to the process of the host.

incoming UDP datagrams with parameter changes, adjusts the plug-in and informs the host accordingly.

On startup, the wrapper calls the VST API to determine the total number, the names and the initial values of the plug-in's parameters. These are transmitted as UDP datagrams. In addition, the wrapper retrieves and transmits the location and size of the plug-in's editor window. These data about the window are used to direct the image processing.

3. IMAGE PROCESSING

Before building an interactive 3D model, the system has to learn which type of control (knob, horizontal fader, etc.) is to be used for each parameter, where the control is located on the screen, what its size is, if the parameter is continuous (such as a level setting) or if it is discrete (such as a waveform selector), and what the number of possible values is for a discrete parameter. Some of these data, such as the number of steps, could in principle be obtained by calls to the VST API. We found, however, that many existing plug-ins do not provide correct answers to such requests. Hence, we decided to collect all of these data from the actual graphics of the user interface.

In the prototype, the analysis phase is currently implemented as a MATLAB® script that communi-

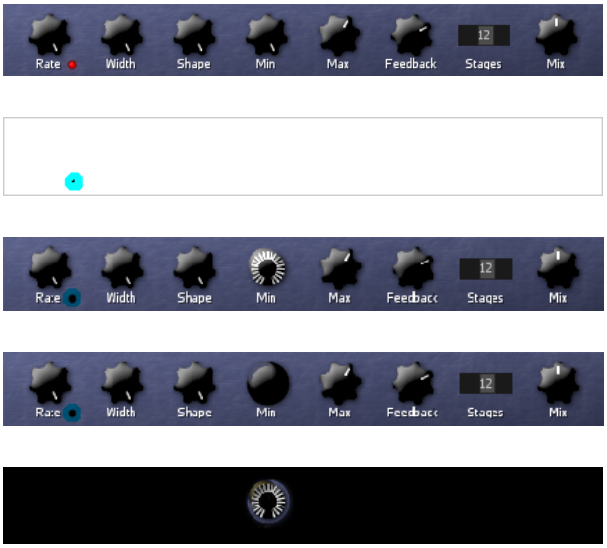

Fig. 2: The plug-in's standard graphical user interface (top; phaser effect from [12]) is checked for autonomous changes to create a mask (second row) of pixels to be ignored during the further analysis. As each parameter is changed, the maximum (third row) and the minimum (fourth row) RGB value per pixel are obtained. The difference image of these (last row) indicates the size and the shape of the control.

cates with the wrapper holding the plug-in. The script uses UDP datagrams to fetch the number of parameters and the size as well as the location of the 2D user interface from the wrapper. It then applies image processing to this area of the screen.

Initially, the script takes a series of screenshots to learn if some parts of the standard interface—such as a blinking indicator light—change autonomously, with no parameter changes being applied. The pixels identified here are masked out in the further process, see Fig. 2.

Then, one parameter after another is adjusted from 0.0 to 1.0 in 21 steps. The script counts how many of these steps lead to a different image on the screen and thus can determine whether the parameter at hand should be treated as continuous or as discrete, and, if the latter is the case, how many steps it has—unless that number is of the order of 20 or more.

In addition, the script determines the per-pixel-maximum and the per-pixel-minimum of the screen shots taken for a single parameter. The difference of the per-pixel-maximum and the per-pixel-maximum per parameter shows which screen area is affected, see the bottom row of Fig. 2. The centroid of this difference image is taken as the geometric center of the control; the standard deviations in the horizontal and the vertical directions are taken as dimensions of the control.

If the standard deviation in the horizontal direction is larger than 1.5 times the standard deviation in the vertical direction, the control is identified as—depending on the number of value steps—a horizontal fader or a horizontal slide switch; similarly for vertical faders and vertical slide switches. In all other cases, the standard deviation in horizontal direction does not differ much from the standard deviation in vertical direction so that the control is classified as a rotary knob or a pushbutton.

The script saves the data on the controls in a structured text file named similar to the plug-in's executable file. This text file is placed in a central directory so that the analysis needs only be executed once; its results can be retrieved easily by other software.

4. BUILDING THE 3D MODEL

The ray-tracer requires a three-dimensional model of the interface. Each time the 3D user interface is started, this model is built from the data stored in the file generated for the specific plug-in during the analysis phase. Knobs, sliders, etc. are placed at the locations and with the dimensions determined in the analysis phase.

For each label, single letters from a map are assembled into a bitmap texture that contains the name of the parameter, see Fig. 3. (The characters are blurred to serve as a bump map, see Section 5.) The varying width of the letters is obtained from the operating system. The labels are generated at a fixed height and hence at a fixed font size unless they collide with neighboring elements, in which case their size is reduced.

The placement of the labels is not known from the analysis phase. Hence, we have to resort to automatic layout. In principle, each control's label could

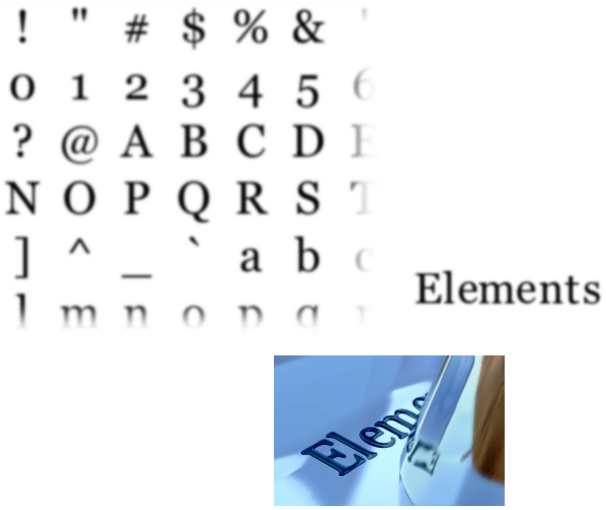

Fig. 3: An image file that contains slightly blurred characters of a given font is prepared upfront (top left). During the startup of the 3D user interface, a texture is built for each parameter's name (top right). During rendering, this texture is used both for coloring and bump-mapping (bottom).

be placed below the control at a fixed multiple of the control's height. However, this solution may interfere with other controls that are located below the current control. Hence, the automatic layout first looks for controls that are located below the current control, for which it takes the widths of the controls into account. If there are such controls, the label is placed between the bottom of the current control and the top of the closest control below.

5. RAY-TRACING

The interactive ray tracer is a C++ program built on Nvidia OptiX 2.1.0 [5, 6]. Like a standard ray-tracer [9], the OptiX framework fires rays through each pixel in the image to be created; the intersections of these rays with objects of the scene and the generation of secondary rays for reflections and shadows are handled through scene-specific programs provided by the application programmer. The exclusive feature of OptiX is that these programs are provided as CUDA code [7]. This code is executed with an extreme degree of parallelism on the graphics processor. Note that graphics processors regularly use a different approach for image synthesis:

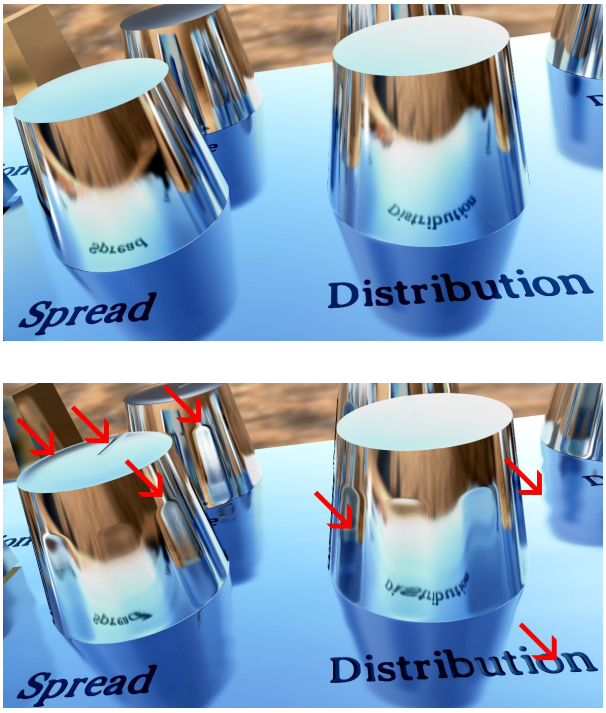

Fig. 4: Bump mapping (top: off, bottom: on) is used to inexpensively create plausible dents and hollows (see arrows).

rasterization of triangles into a color-buffer and a z-buffer [8].

The CUDA intersection programs we use for the 3D interface do not employ any triangle meshes, in stark contrast to typical 3D rendering as done in video games. Rather, the surfaces of all objects are defined through analytic equations that determine where viewing rays intersect with objects and what the slope (technically: the normal vector) of the surface is at such a point. This solution speeds up the computation, particularly because it requires far fewer data to be fetched from memory; in addition, it guarantees perfectly smooth rounding.

The base shapes are frusta cut from cones or from pyramids. As these shapes look artificially clean, we decided to create bump maps [1] to simulate dents and hollows with little impact to the performance. This concerns for instance the caps and the sides of the knobs as well as the panel on which they seem

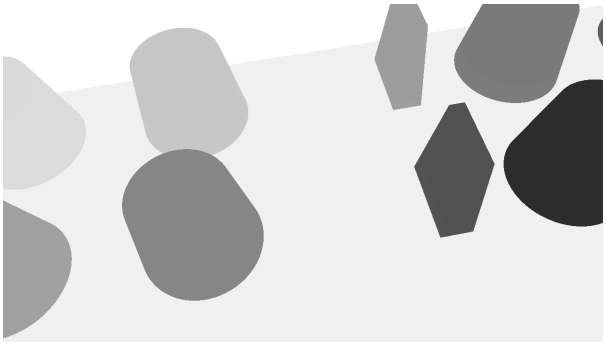

Fig. 5: The identification map quickly answers the question which control is visible at a given pixel of the screen.

to be mounted, see Fig. 4. The labels, too, employ heavy bump-mapping in addition to their dark color so as to appear engraved.

The look of polished chrome requires a colorful environment that is mirrored by the shiny surfaces. To this end, we apply a standard trick from computer graphics: The 3D scene is placed at the center of an infinitely large sphere which is covered from the inside by an 360° panoramic photo, which hence becomes an "environment map".

6. INTERACTION

The ray-traced 3D interface requires a navigation that is different from a regular 2D window: Whereas a 2D window's content can only be scrolled horizontally or vertically, the 3D scene allows placing the camera freely, which can be described by the position of the camera (three coordinates) and its orientation (three angles). We employ a standard game-like interaction with the camera with pan/tilt, strafe (move sideways), and dolly (move closer or away).

To operate one of the controls, the user moves the mouse over the control on the rendering and turns the scroll wheel of the mouse. The corresponding changes are transmitted to the VST wrapper. The other way around, parameter changes sent from the VST wrapper update the 3D scene.

An identification map is employed to determine above which control the mouse pointer is currently hovering. This map is updated every time the user

Fig. 6: Plug-ins by Xhip [12] and Nubi3 [10] served as test cases for the following screenshots.

releases the mouse button after adjusting the camera. Then, the ray-tracer computes an image of the scene in which each object is colored with an individual color, see Fig. 5. This image is stored in memory but never displayed. Rather, the single pixel of this image that corresponds to the mouse cursor's position is retrieved to check whether or not there is a control at this point and, if so, what its number is. As a visual feedback, a small sphere is displayed above the control.

7. RESULTS

For typical VST plug-ins such as the two plug-ins shown in Fig. 6, the system achieves interactive frame rates of five to ten frames per second at 1280×720 pixels already with a low-end graphics card priced at US$ 80 and based on the Nvidia GeForce GT430 chip, see Fig. 7. The frame rate is approximately inversely proportional to the number of pixels on the screen. Through ray tracing, the quality of the reflections is clearly superior to rasterization rendering with environment maps: Regular environment maps as used in rasterization do not show local objects correctly and do not support interreflections.

Fig. 7: Interactive frame rates can be achieved even on a low-end graphics card. The actual frame rate depends on the amount of reflection visible in the image.

For beauty shots, the ray tracer employs antialiasing and depth-of-field blur with continuous accumulation as already implemented in the OptiX SDK's examples, see Figs. 8 and 9. The point above with the mouse is currently hovering is taken as the focal point for depth-of-field blur. Such high-quality images require accumulation over a few seconds; they may be helpful in the design and marketing of hardware at a point of time at which it only exists as a software simulation.

8. OUTLOOK

The presented solution can be extended in a straightforward manner to stereoscopic viewing. However, most current stereoscopic displays (colloquially termed "3D" displays) require the user to wear special glasses. This may not be acceptable in the audio studio. Likewise, head-mounted displays as used for immersion into virtual reality can be ruled out for applications in the audio studio. Such displays may, however, prove beneficial for applications in the design of audio equipment.

The fully automated generation of the 3D model does not lead to satisfactory results with all plug-ins. Uncommon controls are hard to map automatically. For instance, the graphical user interfaces of some plug-ins contain dots inside rectangular areas to control two parameters with one handle. In addition, many software synthesizer plug-ins provide on-screen keyboards, which—not corresponding to VST parameters—lead to corresponding empty areas on the panel of the 3D model. An editor could help reorganizing the controls found in the automated analysis. For use in product design, the editor may even provide a set of building blocks to create 3D models from scratch.

A deep and general question to ask is whether photo-realistic user interfaces are beneficial, and—if so—why and how. The interference of visual information and other knowledge with auditory perception is well known to sound engineers. James Johnston reported [11] about an informal experiment in which he pretended to switch between a shiny high-end tube amp and a cheap transistor amplifier; test persons claimed to hear a difference—even though the switch was not connected to anything.

Aesthetics have a known halo effect onto how customers perceive other attributes of a product. The interplay between aesthetic qualities and traditional concepts of usability is still under debate. Graphical metaphors may be preferred even though their usability is poorer both objectively and in the users' perception [3]. This phenomenon seems to be prevalent in today's audio software; much research may still be required to find the optimal balance between aesthetics and purely operational aspects. The method presented can offer an inexpensive way to test aspects of this halo effect.

9. REFERENCES

[1] J. F. Blinn. Simulation of wrinkled surfaces. *SIGGRAPH Comput. Graph.*, 12:286–292, August 1978.

[2] R. Boulanger and V. Lazzarini. *The Audio Programming Book*. The MIT Press, 2010.

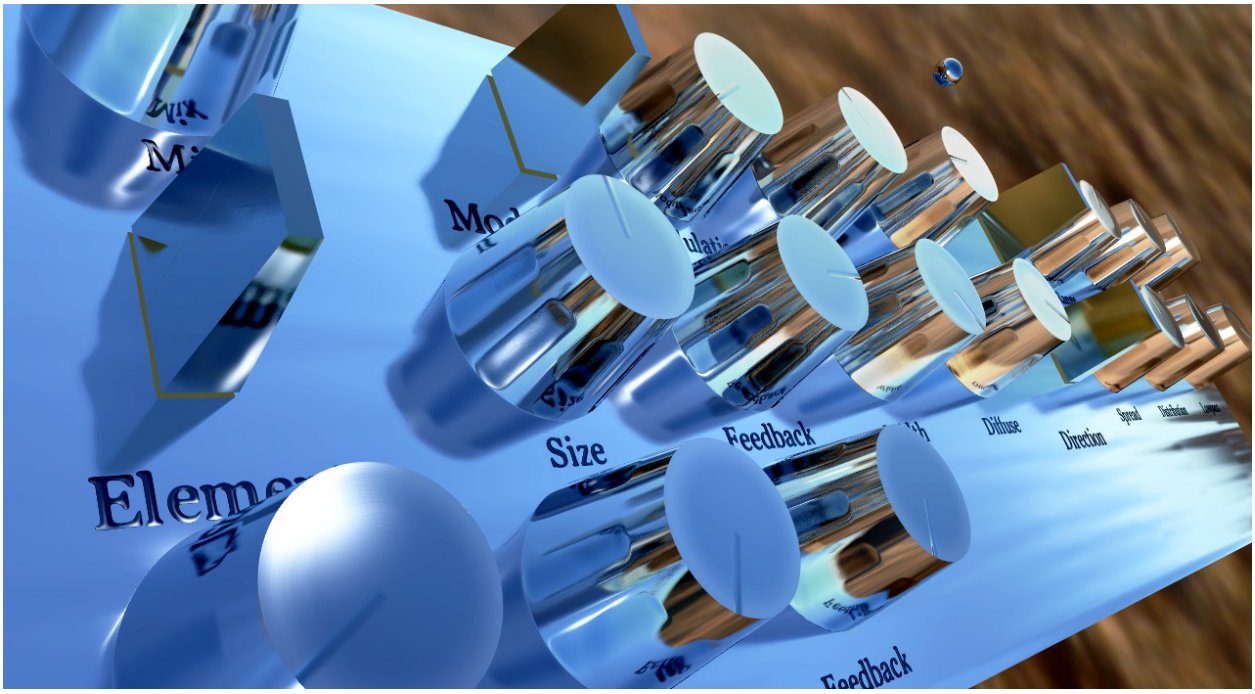

Fig. 8: By accumulating frames over several seconds, the system can produce beauty shots with high-quality antialiasing sufficient for product brochures.

[3] J. Hartmann, A. Sutcliffe, and A. D. Angeli. Towards a theory of user judgment of aesthetics and user interface quality. *ACM Trans. Comput.-Hum. Interact.*, 15:15:1–15:30, December 2008.

[4] M. Irmer. Tyrell: a synth designed by readers. `http://www.amazona.de/index.php?page=26&file=2&article_id=3191`, last visited 2011-07-24.

[5] Nvidia. Developer zone: Optix. `http://developer.nvidia.com/optix`, last visited 2011-07-24.

[6] S. G. Parker, J. Bigler, A. Dietrich, H. Friedrich, J. Hoberock, D. Luebke, D. McAllister, M. McGuire, K. Morley, A. Robison, and M. Stich. Optix: a general purpose ray tracing engine. *ACM Trans. Graph.*, 29:66:1–66:13, July 2010.

[7] J. Sanders and E. Kandrot. *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Addison-Wesley Professional, Boston, MA, USA, 1st edition, 2010.

[8] P. Shirley and S. Marschner. *Fundamentals of Computer Graphics*. A. K. Peters, Ltd., Natick, MA, USA, 2009.

[9] K. Suffern. *Ray Tracing from the Ground Up*. A. K. Peters, Ltd., Natick, MA, USA, 2007.

[10] VST Planet. Spinner LE. `http://www.vstplanet.com/VST_effects/Modulation/SpinnerLE.dll`, last visited 2011-07-24.

[11] E. Winer. Audio myths—defining what affects audio representation. Workshop at the 127th Convention of the AES, 2009.

[12] Xhip. Xhip Effects release 2. `http://xhip.presetexchange.com/effects/`, last visited 2011-05-28.

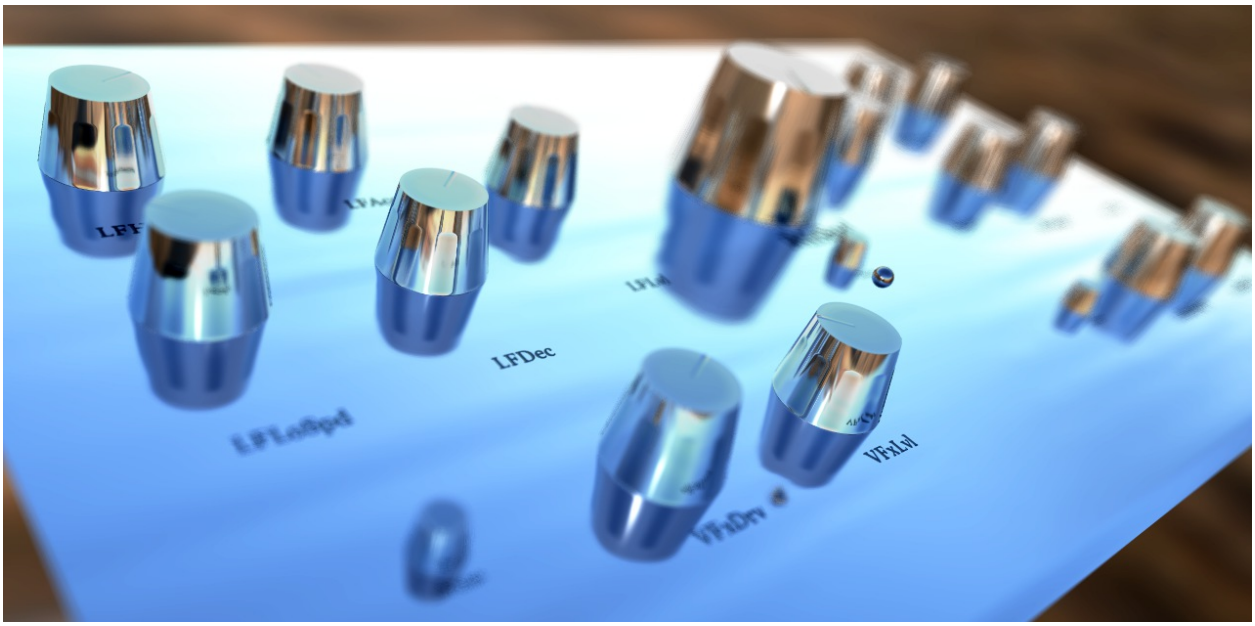

Fig. 9: Another use of the accumulation buffer is to create a depth-of-field blur effect that guides the view to the control that is currently active.