Joint Virtual Reality Conference of ICAT - EGVE - EuroVR (2012)
R. Boulic, C. Cruz Neira, K. Kiyokawa, and D. Roberts (Editors)

Networked Displays for VR Applications: Display as a Service

Alexander Löffler[1,2], Luciano Pica[1,2], Hilko Hoffmann[2], and Philipp Slusallek[1,2]

[1]Intel Visual Computing Institute, Saarbrücken, Germany
[2]German Research Center for Artificial Intelligence GmbH (DFKI), Saarbrücken, Germany

Abstract

Stereoscopic Liquid Crystal Displays (LCDs) in a tiled setup, so-called display walls, are rising as a replacement for the classic projection-based systems for Virtual Reality (VR) applications. They have numerous benefits over projectors, the only drawback being their maximum size, which is why VR applications usually resort to using tiled display walls. Problems of display walls are the obvious bezels between single displays making up the wall and, most importantly, the complicated pipeline to display synchronized content across all participating screens. This becomes especially crucial when we are dealing with active-stereo content, where precisely timed display of the left and right stereo channels across the entire display area is essential. Usually, these scenarios require a variety of expensive, specialized hardware, which makes it difficult for such wall setups to spread more widely.
In this paper, we present our service-oriented architecture Display as a Service (DaaS), which uses a virtualization approach to shift the problem of pixel distribution from specialized hardware to a generic software. DaaS provides network-transparent virtual framebuffers (VFBs) for pixel-producing applications to write into and virtual displays (VDs), which potentially span multiple physical displays making up a display wall, to present generated pixels on. Our architecture assumes network-enabled displays with integrated processing capabilities, such that all communication for pixel transport and synchronization between VFBs and VDs can happen entirely over IP networking using standard video streaming and Internet protocols. We show the feasibility of our approach in a heterogeneous use case scenario, evaluate latency and synchronization accuracy, and give an outlook for more potential applications in the field of VR.

Categories and Subject Descriptors (according to ACM CCS): I.3.2 [Computer Graphics]: Graphics Systems—Distributed/Network Graphics, I.3.4 [Computer Graphics]: Graphics Utilities—Virtual Device Interfaces

1. Introduction

Wall setups based on multiple tiled LC displays screens are beginning to replace the classic approach of projection-based installations when designing hardware setups for VR applications. The reasons are manyfold: displays are brighter, they have a wider color gamut, higher color stability, pixel density and lifespan, while being even more energy-efficient and cheaper than projectors. Furthermore, LCDs can be calibrated much more easily, and do not require setups to include additional space for the path of the light projection but occupy hardly more than the room actually required for the area to display on. LCDs always have the drawback, though, that they are limited in physical size and that a wall- or room-sized VR installation will require many displays to be tiled in order to form a larger display area. Today, this automatically results in discontinuities on the resulting display walls due to the bezels surrounding each single display. Although bezels are becoming increasingly slimmer, such that distances of 5 mm between pixels of neighboring displays of a tiled wall are on the market, this will remain an issue for the upcoming years until really bezel-free displays might become available.

Besides bezels, the much larger problem of tiled display installations is how to connect pixel sources to them. If we take a four-sided CAVE installation as a simple example, we could run the respective application driving it on a single machine that is powerful enough and that is equipped with two GPUs with two heads each, outputting to four active-stereo projectors (one for each wall). This theoretical case would induce no need for synchronization of pixel generation as everything is happening on the same machine. However, for a CAVE consisting of multiple LCDs per wall expen-

DOI: 10.2312/EGVE/JVRC12/037-044

sive specialized hardware would be required, like workstation GPUs with dedicated sync mechanisms and cable connections or hardware pixel processors replicating and scaling input video signals to multiple output signals over dedicated video cables. Another way to feed pixels into a display wall installation is the so-called daisy chaining, where single displays not only have video inputs, but outputs, too, and each full video frame coming in is only displayed partially and passed through to the successor display, and so on. Here, resolutions are typically limited to 1080p for the entire wall, thus wasting a lot of the potentially available resolution. In summary, all of these solutions are very expensive, complicated to set up, and extremely inflexible. Making the way pixel generators and consumers (displays) connect more flexible is the main goal of the work presented here.

As a general approach, the concept of *virtualization* has gained huge popularity in all areas of computing. Virtualization removes the dependence on specific hardware by adding an intermediate software layer emulating the hardware. The software layer offers improved flexibility and new features that would be hard or even impossible to offer on real hardware. For example, this includes running multiple virtual machines on the same hardware for improved efficiency or moving virtual machines between hardware instances for maintenance reasons with minimal or no downtime. The virtualization of computing hardware is also known as *Infrastructure as a Service' (IaaS)* but can be extended also to other hardware, e.g. *Storage as a Service*, as well as to the virtualization of software layers in *Software as a Service (SaaS)*.

1.1. Overview of the Approach

In this paper, we present *Display as a Service (DaaS)* targeting the virtualization of *framebuffers* (the hardware resource applications write pixels to) and *displays* (the resource those pixels are presented on). We expose both hardware resources plus a Web Service control interface on the Internet allowing for flexible remote management. We allow for combining individual virtual frame buffers and displays into new compound devices and finally offer fully synchronized M:N connections between them, such that frame buffer content can be scaled and placed at arbitrary locations on any display.

Content in DaaS is transported over the network using low-delay standard-compliant video encoding and streaming. We even support content that needs strict synchronization, such as active stereo content displayed across multiple independent screens of a compound display. Thus, DaaS goes well beyond just replacing the traditional 1:1 hardware cable connections (e.g., DVI, or HDMI) as done by previous work via Remote Framebuffer Protocol [RL11] or Wireless Display (WiDi) [Int] technology.

Virtual displays in DaaS are registered with their absolute location in 3D space. This opens up new application scenarios for VR and beyond. We propose to integrate DaaS into future display drivers as well as display hardware – which increasingly has the necessary Internet connectivity and computing capability already on board. As a result, DaaS allows to easily connect computers to virtualized displays over standard networks offering the full flexibility of an intermediate software layer.

The remainder of this paper is structured as follows: In the upcoming Section, we discuss related work in the field of display virtualization for VR. Section 3 and Section 4 then describe the concepts and implementation of the DaaS system, respectively. In Section 5, we show how we use DaaS in multiple use cases discussing the achieved results. Section 6 concludes the paper and shows directions opening up for future work.

2. Related Work

When realizing a distributed VR application, there are several stages of processing involved on the way from application to display: controlling input and rendering at the application (which is potentially distributed as well), distributing and transporting rendered pixels across physical hosts, until finally displaying them on a screen. In this process, the responsibility of DaaS begins *after* the generation of pixels and is agnostic to the way those pixels are initially created. As such, DaaS differs from frameworks for distributed rendering like Chromium [HHN*02] or Equalizer [EMP08], which distribute the rendering process itself by, for example, sending OpenGL commands and corresponding geometry to several render servers. Also, all those frameworks include capabilities for receiving and processing user input, for example to move the virtual camera of a rendered scene. As DaaS is renderer-agnostic, it does not assume certain standard interactions for applications built on top of it, but leaves delivering and processing user input to the pixel-generating application, working only on the resulting pixels. So DaaS can not and does not want to fully replace those distributed rendering systems, but could be combined with them and extend their functionality.

The SAGE framework [RRS*04] employs several concepts similar to DaaS, especially assuming raw pixel input into the system, high-resolution streaming of image frames via Internet protocols, which enables connecting even remote locations, and a content management philosophy resembling the window manager of a desktop computer. The main difference of DaaS in contrast to SAGE is the strong focus on synchronization, which allows the former to stream and display even active-stereo content across a display wall. Also, the concept of using the display refresh rate as a clock source for pixel producers (reverse genlock, cf. Section 3.1), allowing more precise motion reproduction, is not used in SAGE. Finally, the usage of standardized video encoding and protocols in DaaS, instead of proprietary ones in SAGE, enables receiving DaaS streams out-of-the-box on a variety of hardware-accelerated platforms. From an architectural standpoint, SAGE operates on a centralized management entity both pixel generators and consumers connect to, whereas DaaS follows a service-oriented approach and makes both resources available in the network allowing them to be interconnected by external service consumers without any global knowledge of the application layout.

To perform output synchronization on composite display walls, all mentioned frameworks typically use a centralized approach with one master that all slave displays report to upon receiving a frame. The master blocks until all slaves have reported and sends a buffer swap signal back. This approach is rather time-consuming and requires extensive back-and-forth network communication that does not scale well [NDV*10]. In addition, to present active-stereo content, synchronizing OpenGL buffer swaps at the display side is not sufficient, because the visual refresh of neighboring displays in a wall setup might still be off up to a full display refresh interval. Thus, to achieve both synchronization of buffer swaps (swaplock) and display refresh (framelock), DaaS adopts and extends clock-based genlock techniques as used in television broadcasting.

3. Concepts

Within DaaS there are two main types of entities involved, representing the opposing views on a DaaS session: *Virtual Framebuffers (VFBs)* and *Virtual Displays (VDs)*. An arbitrary number of VFBs and VDs may be involved in a running session (*m* and *n*, respectively), only limited by available hardware resources. As autonomous service entities of a Service-Oriented Architecture (SOA) [Erl05], VFBs and VDs make themselves available on the network, allowing independent service consumers to either use or interconnect them freely, the latter being the most usual case. All control and pixel information is transmitted between entities using only standard IP networks and protocols, without the need for any dedicated sync or pixel processing hardware. In the following, the two types of entities and their role in the system will be explained in more detail.

3.1. Virtual Display (VDs)

What we call a Virtual Display (VD) in DaaS is the software entity assigned to a single, consecutive or non-consecutive area of screen real estate that is supposed to be used as a whole to present pixels on. Thus, a VD can consist of either a single physical display (either stationary or mobile) or a whole array of those displays (i.e., a display wall). Conceptually, this VD consists of a screen, a network connection for receiving video streams, and a processor able to decode those video streams and present them on the screen. This combination of features is usually present in current-generation mobile devices and “smart” LCD televisions. Nonetheless, the VD concept can be abstracted on a standard computer connected to at least one “non-smart” display. VD services provide detailed information about their capabilities to the network, which outside entities can work with to configure and route the flow of video data. Examples for these capabilities are display size in pixels and millimeters, color depth, 3D stereo features, and exact 3D location in the coordinate system of the session. VDs are addressed by their URL.

VDs that span multiple physical hosts in the network are represented by a single VD entity as well. On a service level, this one VD service represents the entire compound of child displays towards the network. VD services in DaaS follow the *Composite pattern* [GHJV94], in which each VD may be composed of child VDs representing the combined properties of all its children towards the outside. While the service communication with the outside happens only at the root of the VD hierarchy (which represents the entirety of VDs below), video streaming is done in a direct peer-to-peer fashion between pixel generators and the VDs located at the leaves of this hierarchy. In summary, VDs provide the interface to communicate with one physical display in the very same way as with multiple displays forming a consecutive display wall

As an important aspect during display configuration, each participating VD makes its physical size in pixels and millimeters as well as its absolute 3D transformation available to the outside within its service description. Any pixel rectangles that are mapped onto a VD are specified in the coordinate system of the root VD, and may have an additional z-coordinate in order to prioritize the stacking order of overlapping pixels required on the display. Within a composite VD, pixel operations like bezel compensation cutting away pixels "covered" by the framing of single displays can be done automatically and completely transparent for a service consumer. Figure 3 shows an example VD displaying one full-screen video stream of an interactive scientific visualization and a second overlay stream of a running video. As we will see in Section 4.3, even heterogeneous walls with different pixel densities on each participating screen can be set up like this.

The available transformational data of the physical displays can also be used by VR applications to calculate location-dependent data like view frustums on-the fly, and adapt their pixel generation to it. Even scenarios with tracked mobile displays like tablets that update their transformation in real-time are feasible and could be used, for example, to generate magnifying glass or X-ray effects for the areas of a display wall the mobile device currently covers.

To display streaming content in perfect spatial and temporal alignment across a composite VD, we employ multiple layers of synchronization at the VD end. As an important maxim for DaaS, all necessary inter-host communication uses only the IP network as a medium, not any specialized hardware sync cables. Taking a bottom-up approach to synchronization, the first thing to be synchronized to enable displaying active-stereo content as intended is the refresh frequency and phase of the physical displays. This ensures that all displays of a multi-display wall switch in sync with the active-stereo glasses of a spectator, without which stereo display would not be possible. We use the internal clock of the display master and propagate it across the network to all other participating displays of the display wall; in accordance with hardware-based locking mechanisms for synchronization, we call this technique software *framelock*.

Assuming perfectly synchronized display refreshs and employing vertical synchronization (vsync) in the graphics driver within each display we would achieve what is commonly called *swaplock*, that is synchronized framebuffer swaps across multiple hosts. In general, we can not assume the framebuffer swaps to reach the display refresh frequency due to longer application processing times, so an independent swaplock is used to ensure all VDs still swap simultaneously.

Finally, the *redraw* of the video stream needs to be synchronized, such that the same frame of the video is shown on all of the participating displays. This is achieved in DaaS by taking into account presentation timestamps inserted into all video streams and sorting frames into the correct time slot between framebuffer swaps at the VD.

As a final, but optional kind of synchronization we make the swaplock signal of a VD available to pixel-producing applications in order to optionally synchronize their pixel generation to the framebuffer refresh rate of the receiving displays. In the optimum case of fast enough display hardware, this signal corresponds to the framelock signal as well. As this inverts the concept of generator locking (*genlock*), where a sink locks to the clock of its source, we call this technique *reverse genlock*.

3.2. Virtual Framebuffer (VFB)

A Virtual Framebuffer (VFB) is requested by a pixel-generating application as a resource to write its pixels into. As such, they are the points where new pixel data enters a DaaS session to be displayed anywhere. For any application, this is done by providing a global identifier and format information (e.g., resolution or color depth) and the DaaS runtime assigns the application a memory region to write its pixels to. If a VFB should be shared among several processes or even network locations, applications can attach to a previously created VFB, and just specify the region within the whole buffer they want to be responsible for. Afterwards, an application simply writes pixels into the VFB and signal it whenever it has finished writing a frame (end-of-frame signal). Behind the scenes, if at least one service consumer is connected, DaaS performs pixel operations on those frames (e.g., scaling or color conversion) as well as – most importantly – real-time video encoding and streaming (cf. Section 4.1.2).

If a VD is connected to a VFB its internal hierarchy of potential sub-VDs becomes available and is used to stream the video directly to each of the leaves. In doing so, the VFB performs the necessary splitting and scaling of input pixel frames, performing bezel compensation along the way according to the configuration of the connected VD. Saving network bandwidth is a top priority, which is why DaaS performs only downscaling of the input pixels at the VFB end, but scales up at the VD location.

Regarding the resolution to write into a VFB, applications are not limited. They may request and fill as many pixels as they like, being limited only by the performance and memory properties of the host both the application and the (partial) VFB are running on are the limiting factors. Applications are encouraged to provide custom timestamps with their frames, which then can be used directly to insert them in all video streams originating from the same frame of pixels, and to synchronize their playback at the VD end. In doing so, any given timestamps are transposed to the system time on the fly. As mentioned before, as an advanced functionality, VFBs can adapt their pixel generation to the clock available at the display end: Typical usage scenarios are, for example, limiting the frame rate of produced frames to the one the display operates in, or locking the frame generation to a 120 Hz active stereo frame rate (cf. Section 4.2.1).

VFBs are not limited to just one array of pixels to receive from the application and send out to VDs per frame, but can work on many of them simultaneously in what we call separate *channels* of the VFB. Each channel represents one view of a multi-view setup, for example one eye of a two-channel stereo VFB, or one view of a multi-channel VFB to display for example on an autostereoscopic VD later. In a VFB with more than one channel, the end-of-frame signal from the application (and the potentially given timestamp for this frame) indicates the finalization and initiates the internal processing for all channels of that VFB. So frames for the left and right eye of a stereo VFB, for example, all end up with the exact same timestamp at each simulation instant.

The separate channels, although matching in size and position, are provided as separate streams at the VFB, such that VDs can operate in whichever way they like on the available channels and, for example, show only the rendering for the left eye of a stereo VFB on a control monitor but use both channels on a stereo-capable VD. A VD implementation can then fuse the two separate streams again into a format that is suitable for stereo output on the respective device hosting it. It might, for instance, show the unmodified streams on two differently polarized projectors, perform spatial interleaving for line-by-line passive stereo or compose packed frames for outputting to a TV capable of HDMI 1.4(a) side-by-side or top-bottom formats. All this can be done using the timestamps within the video streams uniquely identifying the correlating frames within the streams of each channel.

Figure 1 summarizes the concepts of VFBs and VDs within our DaaS framework again in an example scenario showing a scientific visualization rendered in two distributed VFBs for different output VDs.

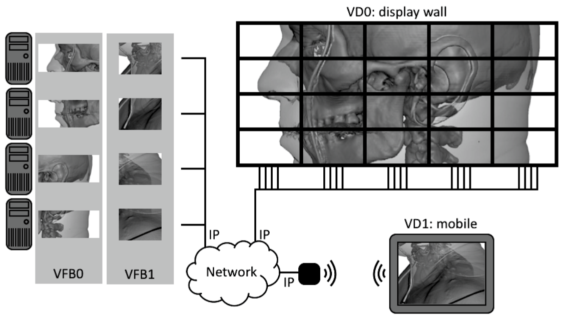

Figure 1: *Four render servers writing into two VFBs. Two VDs connecting over (wired or wireless) IP networking using one of these VFBs each in their visualization.*

4. Implementation

DaaS is implemented in C++ on Linux and Windows, using the *Internet Communications Engine (Ice)* [Hen04] as messaging framework between its entities, but also in its service API towards the outside. VFBs and VDs define their interfaces in the specification language for Ice (Slice), which is precompiled to C++ and used at compile time in the service implementation. Ice is available for several more platforms and languages, such that we can assume good connectivity for any future developments or ports of DaaS.

The DaaS implementation covers the two views on the overall system as introduced in the previous Section: the VFB view, where pixels are generated and fed into the system for streaming them to the network; and the VD view, where video streams are received and displayed on output devices. Furthermore, an implementation of a web-based management interface acts as an example service consumer, which provides simple means to map available VFBs to available VDs.

4.1. VFB Service

An application that wants to push pixels into the DaaS system requests a VFB entity through the DaaS API by specifying format parameters (e.g., pixel ordering, number of channels) and may start pushing arrays of pixels to it, which are immediately available in the network. Overall, the tasks covered by the application are signaling the VFB to start streaming, to send the next full frame after it finished writing, and, ultimately, to stop streaming again. This happens in three simple method calls, `signalStart()`, `signalEOF()`, and `signalEnd()`, respectively. If the application wants to operate on its own timeline, it may provide a custom timestamp with each end-of-frame call, otherwise the internal timer is used to generate a live timestamp.

Furthermore, the application may register for callbacks from the VFB to receive function calls in case certain external events occur (e.g., resizing or moving of the pixel rectangle at the display end). The application then may react on those events; for instance, it might produce less pixels if the current VFB size is not needed for any connected VD anymore, and thus decrease its workload.

For each channel, the VFB provides a memory area to write into. The VFB method `getBuffer(`*i*`)` returns a pointer to the memory location corresponding to the *i*-th channel, where the application then writes pixels to as raw bytes. Whenever an end-of-frame signal is received from the application, the current content of all buffers in all channels is passed to the *buffer distributor*. All of the following entities are hidden from the application, which just communicates with the VFB entity.

4.1.1. Buffer Distributor

The buffer distributor exists once per channel of a VFB and is the first component that receives a buffer from the outside (i.e., the pixel-generating application). It furthermore has the knowledge of all connected VDs, their physical properties, spatial layout and channel processing capacities. Using that knowledge, the buffer distributor takes the incoming pixel frames, splits them up, and pushes the resulting sub-buffers into *transport pipelines*, which are generated according to the layout of a VD whenever it connects to the VFB.

4.1.2. Transport Pipeline

Core components of any channel entity are the transport pipelines. Whenever a VD is connected to a VFB, at least one transport pipeline is created (when streaming the entire VFB frame to a single-host VD). There can be many more pipelines when dealing with a multi-display VD, a multi-channel VFB, or a combination of both, all of which resulting in streaming only parts of the VFB pixels to distinct leaf VD entities.

Each transport pipeline consists of single processing elements called pipeline stages. Stages are daisy-chained inside the pipeline, such that each stage receives the result of its predecessor stage, with the first stage directly receiving pixels from the buffer distributor and the last stage releasing its output data to the network. Stages implemented so far target the basic application scenario of streaming the pixel content of an application to the network: a scaling stage, a color conversion stage to perform RGB-YUV conversion, an encoding stage performing video encoding according to the H.264 standard [ITU12], and a streaming stage providing the encoded video as a stream to the network. The implementation allows for freely replacing stages as long as input and output data formats remain identical, such that we can exchange parts of the processing pipeline with different implementations later.

4.2. VD Service

The implementation of the Virtual Display service roughly inverts the VFB implementation, which is why we can keep its description here rather short. VDs receive video streams iff they do not have further VD children (i.e., they are leaves in their VD hierarchy). To do this, VDs make use of transport pipelines again (cf. Section 4.1.2), within which they use new stages for *receiving* buffers and performing H.264 *decoding*, but are able to reuse the exact scale and color conversion stages from the VFB end. Transport pipelines at the VD end of a DaaS session are created only on a direct trigger from the outside, which initiates the mapping of external video streams to the VD in question. This trigger originates either at an external service consumer directly, or from the direct VD parent in the hierarchy. The final state of buffers passing any pipeline here are raw pixels again. Normally, one pipeline ends up being mapped to one *canvas* to displays those pixels, but for streams coming from a multi-channel VFB, multiple streams may be composited into a single canvas. The exact technique depends on the capabilities of the output device (e.g., line-interleaved for a passive stereo display).

A single VD service may hold one ore more *screens*. Although the usual mapping is just one screen per VD, the additional screen abstraction allows to integrate legacy setups with, for instance, four displays connected to one computer. The screen component does the actual drawing of pixels onto the display. It holds multiple canvases, each of which representing the rectangular portion of one incoming video stream (or several composited ones) displayed on its screen. A display wall VD consisting of four displays and displaying a full-screen stream across the entire screen space, for example, would have overall four screen instances and one canvas per screen, each displaying a quarter of the video stream. Canvases in addition have a z-coordinate, so they can be correctly sorted in overlapping cases like the example in Figure 3 shows. The screen and canvas concepts in DaaS are abstract interfaces and can be implemented in many ways; we did implement them using OpenSceneGraph [BO04]. At this, each screen is realized as a full-screen OpenGL render window with each canvas being a screen-aligned quad showing a video texture of the pixel content.

4.2.1. Synchronization

The synchronization architecture of DaaS includes swaplock and framelock portions (cf. Section 3.1): the OpenGL buffer swap is locked to the swap interval of the display master of the VD, the presentation timestamps (PTS) within incoming video streams are used to either drop the current frame (PTS in the past) or wait until the presentation time is closest to the next buffer swap (PTS in the future). At the source VFB, the PTS is set with a certain offset to the current time, starting at 100 ms and being dynamically adapted during runtime. Here, the VD evaluates the time difference between a buffer being ready for presentation at a canvas and the actual PTS for that buffer. Those values are smoothed across multiple frames and reported back to the VFB as an adjustment hint for its offset calculation. For a standard LAN setup, offset adjustment to balance frame loss and low latency ceases after a few dozen seconds and then remains in a stable range as long as no new workload is added either at the VFB or VD ends.

Framelock is taking place outside the VD implementation as a separate system process. At this, we lock the refresh rate of all physical displays to the one at the display master by using small UDP packets containing a *display clock reference (DCR)* timestamp. All slaves that receive the DCR assume it to be spaced equidistantly on the time axis of the master display and adapt their own frequency and phase to the one of the master. The absolute master clock is deduced at all receivers from the DCR in the packet and a roundtrip time estimate. The local clock is deduced from the timestamps of the local vertical blanking interrupts. The local display clock is actually modified by modulating the *voltage controlled crystal oscillator* (VCXO) of the local system. At the time of writing, the Intel CE4100 digital set-top box is the only hardware where we managed to modulate the VCXO via software, but we are looking for further ways how to modify display refresh rates via software interfaces. Miroll et al. [MLM*12] provide a full treatment of the synchronization architecture used in DaaS.

4.3. Service Consumers

Using the service interfaces provided by VFBs and VDs, service consumers can create mappings from pixels generated somewhere in the network to displays displaying those pixels. We created an example web application, which makes use of those interfaces and visualizes available VFBs and VDs available in the network. Mappings are created in the browser by dragging VFB representations onto VD representations, allowing VFB resizing and repositioning. The result is shown immediately on the respective output devices that are part of the VD. Figure 5 shows a screenshot of a scenario consisting of three VFBs mapped onto a 21-screen VD both in the browser configuration interface and on the resulting real-world display wall.

4.4. Software Architecture

The DaaS software architecture is summarized in Figure 2 and is strongly based on dynamic composition of entities. At the pixel-producing end of the depicted scenario, the VFB service is created by the application. It contains one or more channels, which in turn contain a buffer distributor distributing incoming buffers in transport pipelines. Pipelines are generated for each connected pixel receiver and contains a number of processing stages up to the streaming stage releasing the stream in the network.

At the opposite end, this stream is then received by a VD service, or, to be more precise, the respective stage inside the transport pipeline responsible for this stream. Passing the pipeline, the finally available raw pixel data is mapped to a canvas, which is a defined pixel rectangle on a physical display device. In case multiple screens are part of one VD, a VD may contain multiple Screen entities, but the usual mapping is one screen per (leaf) VD. VDs can again be composited into larger VDs (cf. Section 3.1, not shown in Figure 2).

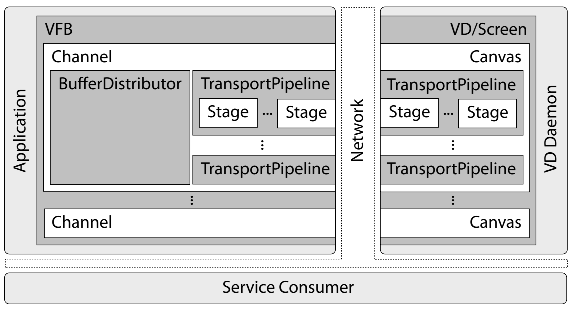

Figure 2: *System architecture of DaaS. Depicted is an pixel-producing application providing a VFB service and a deamon application providing a VD service. They are interconnected by a service consumer through the network. Note that the transport pipelines of one VFB and one VD usually do not directly map, but one VFB streams to many VDs.*

5. Results

We implemented the following two use cases in DaaS for performance evaluation. In terms of user interaction both scenarios assume direct user input at the VFB host, that is, the location where the pixels are generated. We allow this simplification, because a full treatment including distributed input was out of scope for this research.

5.1. Heterogeneous Monoscopic VD

The use case "Heterogeneous Monoscopic VD" is the one shown in Figure 5. All three VFBs available and mapped to the screen originate from an Intel Core i7 quad-core laptop connected to the rest of the system via Wi-Fi: two VFBs (resolutions 1080p and 2130x1130) are created by two instances of an application that reads a video file, decodes it and writes the decoded frames to the VFB, the third VFB (resolution 4x1080p = 4320x1920) is created by a volume visualization application rendering the "Visible Human" dataset into a VFB. The VD used in this setup consists of a 5x4-screen display wall (overall resolution: 12800x6400) and an adjacent WQXGA projection screen. The use case aims at testing VFB placement across bezels and heterogeneous VDs, synchronization of monoscopic content across hosts, the frame drop situation over the Wi-Fi connection, and the overall systematic latency in using the interactive rendering application.

VFB placement works as expected with bezels and gaps around the installations as well as the different pixel densities of display wall and projection being correctly compensated just by evaluating the absolute transformational properties of each participating VD. This setup only synchronizes up to the point of swaplock, because all participating VDs are not running on the set-top box hardware capable of our software framelock (cf. Section 4.2.1). The wireless connection leads to occasional frame drops in the video stream, which are correctly handled by the synchronization (it continues to show the previous frame). We want to further improve performance in the future by employing wireless error correction schemes on top of the UDP transmission protocol in use.

The offset configured in each VFB, that is the overall latency from VFB to screen, influences the ratio of dropped frames due to frame loss on the network or belated arrival at the display. Configuring a lower offset from the timestamp at frame generation until its presentation at the display means a lower latency, but also more frames that cannot reach this deadline and are dropped at the VD. The automatic adjustment of this presentation offset starts at 100 ms and reaches an average of 47 ms transport latency across the described three-VFB setup after stabilization. We plan to use hardware-supported pixel operations (e.g., encoding) in future pipeline stage implementations and expect to lower the time spent in the transport pipeline even more significantly in comparison to this first software implementation.

5.2. Stereoscopic VD

The use case "Stereoscopic VD" aims solely at testing the synchronization in a stereoscopic VFB and VD setting. It consists of a simple, three-screen setup using commercial off-the-shelf stereo TV sets in 1080p resolution each, where we removed the outer rim, resulting in a very thin bezel of only 8 mm between pixels of neighboring screens. Attached to those screens are three instances of the Intel CE set-top box, one for each screen. Stereoscopic channel separation happens using active 120 Hz stereo. The content is written on a separate machine into a two-channel VFB with a 1080p resolution per eye. The stream is then mapped full-screen onto the three-screen VD, doing an upscaling at the VD. Figure 4 shows the bezels and the resulting channel separation of this setup in a closeup picture.

In this setup, we reach a residual phase error of the display refresh measured in software of around $\pm 100\mu s$ and long term frequency identity. The absolute phase offset due to round-trip time estimation error is assumed negligible in a LAN. Given phase error corresponds to $\pm 1.2\,\%$ of the 120 Hz refresh rate, which we deem sufficient for good stereo separation. Current drawback of the implementation is the comparatively long time needed until initial synchronicity is established, which is about 3 minutes because the VCXO pull range is limited and the resulting adaptation of the clock is slow. We hope to mitigate these issues with different hardware platforms in the future.

6. Conclusion and Future Work

In this work, we have shown DaaS, a software framework to flexibly manage pixel transport using only IP connectivity. DaaS includes the concepts of Virtual Framebuffers (VFBs) and Virtual Displays (VDs) representing resources for pixel generation and presentation, potentially distributed all over the network. VFBs may contain multiple channels for stereo or other multi-view content, which can be presented on output devices using arbitrary channel separation techniques. VDs in the system are registered with their exact physical location, such that for example automatic bezel correction or application-specific view-dependent rendering are easily possible, even for live updates of display transformations.

Besides the issues mentioned in Section 5 we most prominently want to push the DaaS framework further towards immersive VR scenarios. We realize that the current delay introduced by our first software implementation of the transport pipeline might be too high for some use cases involving tracked users, but we know that real-time video encoding and network transport can be sped up tremendously, even over the Internet as cloud gaming platforms like OnLive [OnL] show. We furthermore want to work on the quality of the video encoding, which already is very nice, but there might be applications (e.g., visualizations with very fine wireframe meshes) that want to fine-tune the encoding even more.

Acknowledgements

The work presented in this paper has been co-financed by the Intel Visual Computing Institute and the Software-Cluster Germany. The content is under sole responsibility of the authors.

References

[BO04] Burns D., Osfield R.: OpenSceneGraph: Introduction, examples, and applications. In *VR '04: Proceedings of IEEE Virtual Reality 2004* (Mar 2004). 6

[EMP08] Eilemann S., Makhinya M., Pajarola R.: Equalizer: a scalable parallel rendering framework. In *ACM SIGGRAPH ASIA 2008 courses* (New York, NY, USA, 2008), SIGGRAPH Asia '08, ACM, pp. 44:1–44:14. 2

[Erl05] Erl T.: *Service-Oriented Architecture: Concepts, Technology, and Design*. Prentice-Hall International, 2005. 3

[GHJV94] Gamma E., Helm R., Johnson R., Vlissides J.: *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994. 3

[Hen04] Henning M.: A new approach to object-oriented middleware. *IEEE Internet Computing 8*, 1 (2004). 5

[HHN*02] Humphreys G., Houston M., Ng R., Frank R., Ahern S., Kirchner P. D., Klosowski J. T.: Chromium: a stream-processing framework for interactive rendering on clusters. In *SIGGRAPH '02: Proceedings of the 29th Annual Conference on Computer Graphics and Interactive Techniques* (2002), pp. 693–702. 2

[Int] Intel® Wireless Display (WiDi). http://download.intel.com/network/connectivity/products/prodbrf/323116.pdf (May 18, 2012). 2

[ITU12] ITU: *Advanced Video Coding for Generic Audiovisual Services*. ITU Recommendation H.264 (01/12), International Telecommunications Union, 2012. 5

[MLM*12] Miroll J., Löffler A., Metzger J., Slusallek P., Herfet T.: Reverse genlock for synchronous tiled display walls with smart internet displays. In *Proceedings of the 2nd IEEE International Conference on Consumer Electronics (ICCE-Berlin)* (September 2012). 6

[NDV*10] Nam S., Deshpande S., Vishwanath V., Jeong B., Renambot L., Leigh J.: Multi-application inter-tile synchronization on ultra-high-resolution display walls. In *Proceedings of the first annual ACM SIGMM conference on Multimedia systems* (New York, NY, USA, 2010), MMSys '10, ACM, pp. 145–156. 3

[OnL] Play on-demand video games over the internet – OnLive.com. http://www.onlive.com (May 21, 2012). 8

[RL11] Richardson T., Levine J.: *The Remote Framebuffer Prototcol*. RFC 6143, Internet Engineering Task Force (IETF), 2011. 2

[RRS*04] Renambot L., Rao A., Singh R., Jeong B., Krishnaprasad N., Vishwanath V., Chandrasekhar V., Schwarz N., Spale A., Zhang C., Goldman G., Leigh J., Johnson A.: SAGE: the Scalable Adaptive Graphics Environment. In *Proceedings of WACE 2004* (September 2004). 2