Technical Disclosure Commons

Defensive Publications Series

August 21, 2019

Shell Multitasking

Anonymous Anonymous

Shell Multitasking

Abstract

A system for switching among 2D application views or showing multiple windows for different applications at a same time on android in a virtual reality environment is disclosed. The applications render into one or more surfaces using layers, wherein the surfaces are managed by an out-of-process compositor. The layers are means of defining rendering for each of the surfaces during a time warp, which can produce a higher quality and performance of a rendered output. The time warp is a technique that warps a rendered image before sending it to a display. The rendered image is warped in order to make corrections for a head movement that occurs after it is rendered to the surfaces. While rendering to the surfaces, the applications configure the surfaces (as it may require) using various commands. While rendering from each of the applications, a “Surfaces” class and a “SurfaceTextures” class in android are utilized for a cross-process composition. The cross-process composition is a process in which a production of graphics occurs on one thread/process, and a consumption occurs from another thread/process. Further, a cross process layer (called as a VR system service or a VR runtime service) facilitates smooth switching of both rendering of an information and a systemUX (for example, a keyboard, a taskbar, dialogs, notifications, *etc.*) among the applications. Finally, the out-of-processor utilizes an internal API such as a SubmitFrame API to access all the layers, thus enabling the out-of-process compositor to perform a multi-surface and multi-application rendering. This way, by means of the out-of-process compositor, the multi-surface and multi-application rendering allows the multiple windows for different applications to be shown at a same time.

Problem statement

Figure 1 describes a system architecture of a VR (virtual reality) shell in an existing art. The VR Shell offers a shared rendering environment and the systemUX to panel applications (for example, a browser, a gallery, a video, *etc.*). The system accomplishes this through systemUX panel services (as shown in the Figure 1), while hosting external APK content services (for example, a Home/Explore or a Browser).

The VR shell, as shown in the Figure 1, comprises of:

i. VRshell activity
ii. Bait activity

The VRshell activity includes a VRdriver. The VRdriver includes a low-level VR compositor and a time warp component. In the bait activity, VR applications include a loader, which loads and sends the layers to be rendered to overlay surfaces of the VRdriver. The layers are useful for rendering the information such as text, video or textures, *etc*. A path is provided through the overlay surfaces to render some limited UX notifs (notifications) to other foreground activities; however, it does not include a backing application logic, input handling, or a more complicated UI (the taskbar, the keyboard).

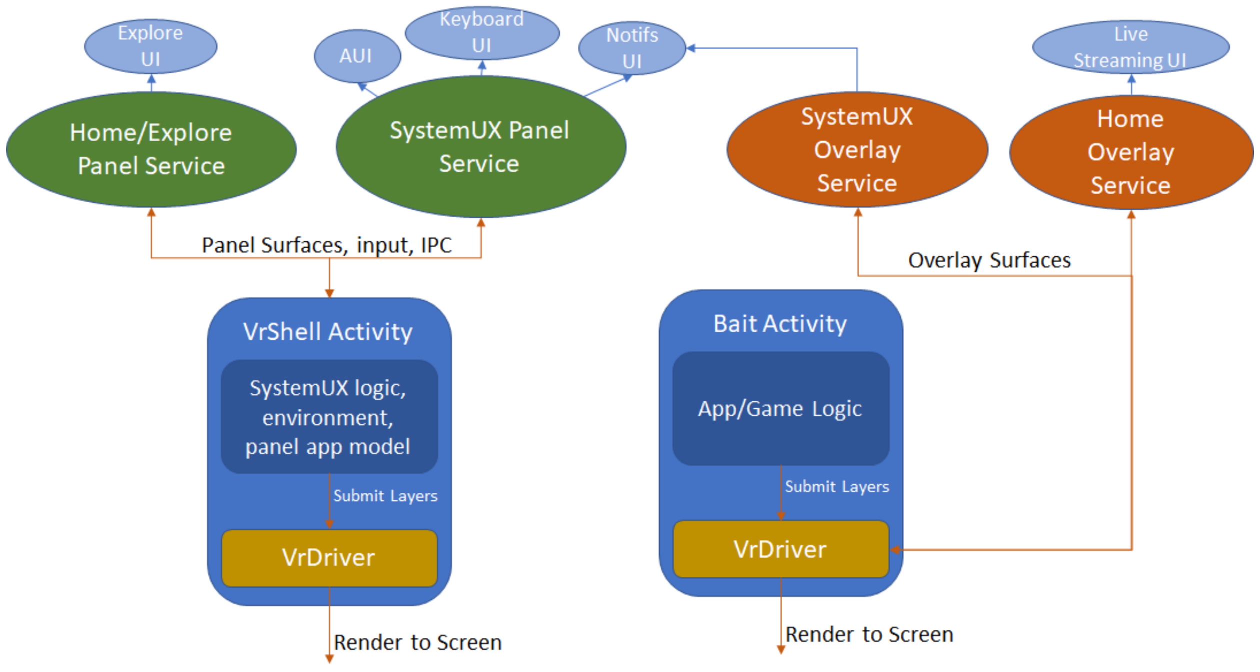

Figure 1: The system architecture of the VR Shell in the existing art

However, there are several aspects of this architecture that have certain limitations. Firstly, a logic and a state for the systemUX are embedded in the VRshell activity that does not run along with other applications (for example, the VR applications in the bait activity). Also, third-party applications render the System UX (via an in-process VRdriver), which cause readback security issues. Further, instances of the VRdriver cannot coordinate or transition properly, since job of the VRdriver is VR composition and it is hosted in-process, which makes it difficult to execute the cross-process composition smoothly.

The present disclosure proposes a novel solution to overcome the above-mentioned limitations.

System and working

The present disclosure describes a system for switching among 2D application views or showing multiple windows for different applications at a same time on android in a virtual reality environment. The applications render into one or more surfaces, which are managed by an out-of-process compositor.

A surface is an object that holds pixels that are being composited to a screen. The surfaces are used to represent a view or a window in an architecture of the system. The view is an interactive UI (user interface) element within the window. The window has a view hierarchy attached to it, which provides a complete behavior of the window.

The out-of-process compositor allows panel applications (for example, a browser, a gallery, a video, *etc.*) to render information to the one or more surfaces. The surfaces provide a platform specific way to share the information, such as graphics output across processes. This is referred to as a cross-process synchronization of the graphics output. Each of the panel applications renders to the surfaces using layers. The layers are useful for rendering the information, such as text, video or textures, *etc.* The layers are a way of defining the rendering for the surface during a time warp, which can produce a higher quality and performance of a rendered output. The time warp is a technique that warps a rendered image before sending it to the screen. The rendered image is warped in order to make corrections for a head movement that occurred after it was rendered to the surfaces. While rendering to the surfaces, the applications configure the surfaces as it may require. The surface can be configured as a layer. During the timewarp, for example, it can be configured to render as a cylindrical, a planar or an equirectangular presentation. This configuration allows an achievement of high-quality rendering through customized timewarp shaders. The high-quality rendering is achieved since the customized timewarp shaders do a texture sampling of high quality.

To configure the surfaces, following commands are used:

1. resize<width> <height> <shape> <stereo> //a resize command sent by any of the applications when to change a layout, shape and stereo of the surface, parameters (in the resize command); in this case, controls aspects of rendering the surface as the layer.
2. layerResize<layer name> <width> <height> <shape> <stereo> //a newer multi-surface variant of the resize command that allows to target changes per layer.

The system, by means of the surfaces, obtains the information from the processes of each of the applications that it wants to display, wherein the out-of-process compositor and the layers define how the information will be rendered. For example, a "Surfaces" class and a "SurfaceTextures" class in android are used for this purpose. The “Surfaces” class and the “SurfaceTextures” class in android are wrappers around memory. The “Surfaces” class and the “SurfaceTextures” class are sent across a process boundary and they get hooked up to specialized queues, which enable a production of graphics on one thread/process, and a consumption from another thread/process. This is referred to as a cross-process composition.

Thereafter, to facilitate switching among multiple applications, the out of process compositor is created, which acts as a foundation for a cross-application functionality and the rendering of the information. Figure 2 and Figure 3 show the architecture of the system, which enables the cross-application functionality. The system introduces a cross-process layer (called a VR system service or a VR runtime service), as shown in the Figure 2 and the Figure 3, which encompasses both rendering and routing commands, for example “launch a system keyboard” to a VRshell service. The applications communicate directly to the VRshell service through the VR system service. Thus, a shell functionality lives out of the process in the VRshell service and there are no duplicate services or mechanisms for sharing the surfaces or input. The surfaces and the input are shared through a common path through the VRshell service as shown in the Figure 2. In the VRshell service, a systemUX controller logic is hosted out of a VRshell activity and into a shared service as well. A systemUX (for example, a keyboard, a taskbar, dialogs, notifications *etc.*) can be hosted by the VR system service on top of any of the applications.

As shown in the Figure 2, there exists a shell rendering engine for each of the applications. The shell rendering engine is a small kernel for hosting a shell-like environment, which can be used as an application shell or by the application shell. The applications submit the layers to the VR system service through the shell rendering engine. The VRshell activity, as shown in the Figure 2, is an extremely thin kernel for communicating to the VRshell service through the VR system service. It is the same small kernel that is used by the application shell and could even be an instance of the application shell customized as a system home application.

This way, by means of the VR system service, the system allows the rendering and the systemUX to switch smoothly among the applications.

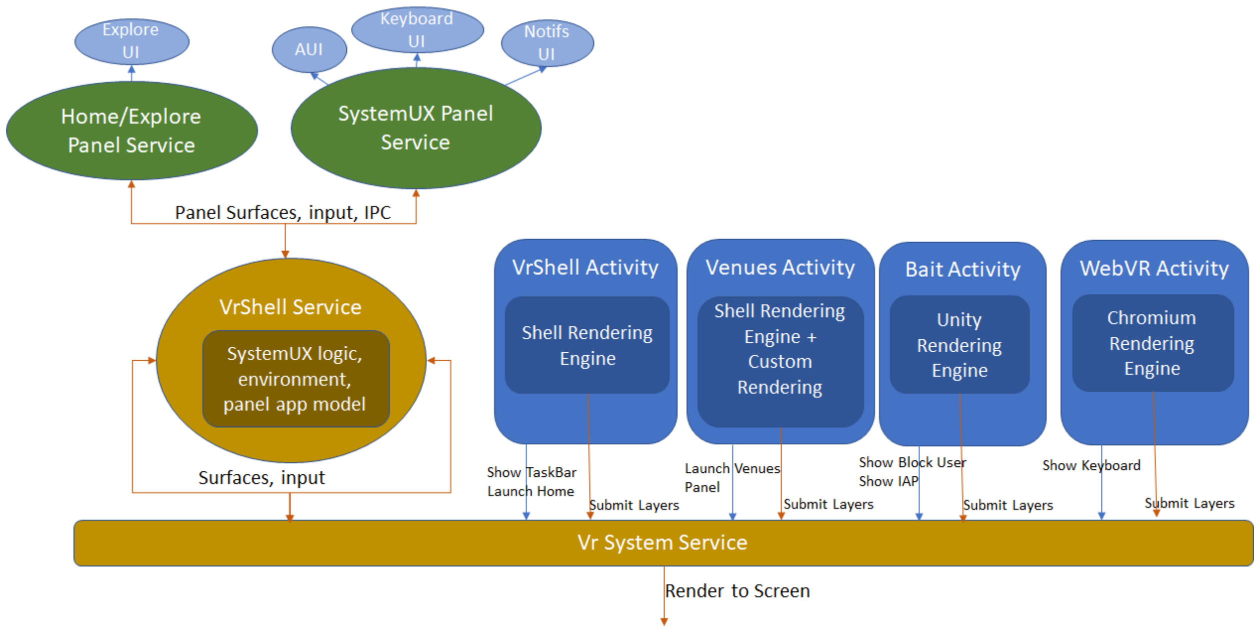

Figure 2: The system architecture for cross-application functionality

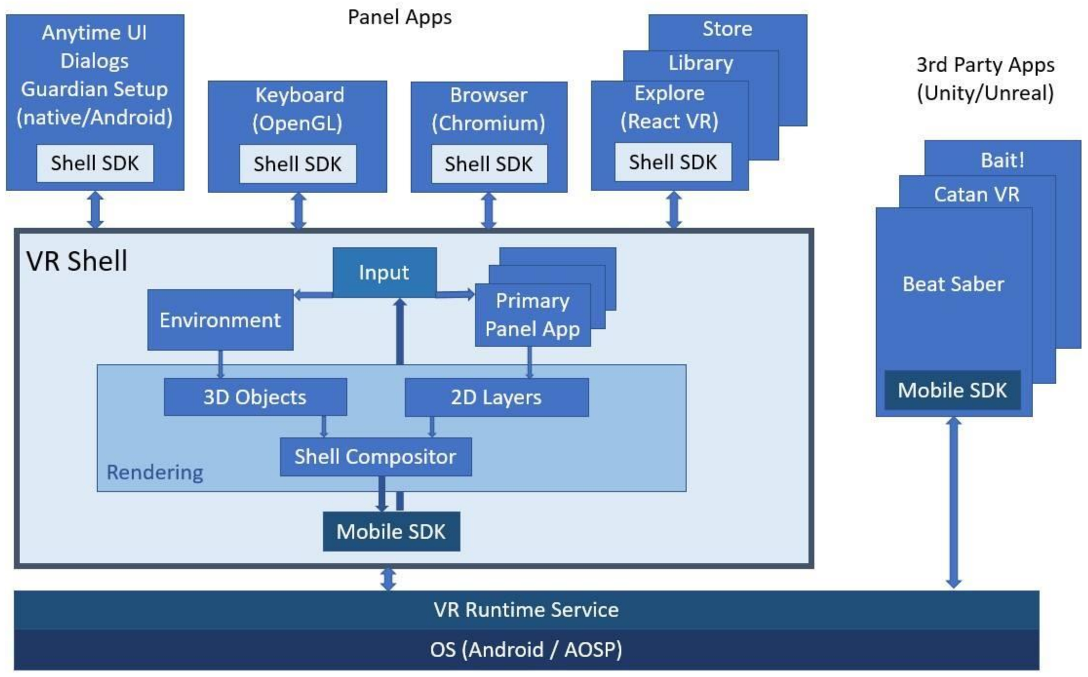

Figure 3: The system architecture for cross-application functionality

Further, the out-of-process compositor (shown as a shell compositor in the Figure 3) has a private access to a mobile SDK compositor through internal APIs. For example, a temporary version of a SubmitFrame API can be used. The SubmitFrame API captures a current status of the layers and displays it on the screen. This allows for a full use of 16 layers in the out-of-process compositor, which is needed to perform a multi-surface and multi-application rendering, while giving an access also to some of layouts of the layers required to render static environments that are appealing to a user. The out-of-process compositor then combines multiple surfaces and sends pixels to the screen, thus showing the multiple windows for different applications at the same time.

Additional Embodiments

In an additional embodiment, virtualized surfaces are enabled for the applications to further extend capabilities of the out-of-process compositor. Any of the applications may render to a region in each of the virtualized surfaces, the region being much larger than the surface itself.

In another embodiment, the applications might be allowed to control the configuration of the surface as the layer. Due to this, new types of layer rendering, functionality and then upgrading of the applications is possible without requiring them to take an update directly.

In a yet another embodiment, merging of the surfaces is done at a system level for the surfaces, which are configured to live within a space of a single layer. The applications that are not often updating their textures may be merged into the single layer to be rendered all at once. This optimization would be completely transparent to the applications.

Conclusion

Virtual reality is going to change the way we use computers in the very near future. Truly disruptive applications for this emerging technology are going to be around productivity and efficient handling of tasks. With the proposed architecture in the present disclosure, a step is put forward in this direction. There are several advantages of the proposed architecture over the architectures in the existing arts. For instance, introduction of a cross process layer called as the VR system service facilitates the rendering and the systemUX to transition or switch smoothly among the applications. Also, the proposed out-of-process compositor allows multitasking by enabling the multi-surface and multi-application rendering.