WHITEPAPER

SEPTEMBER 2019

Abstract

The Graphics Sharing and Distributed HMI project is setup to explore the technologies for complex graphics interaction in distributed and consolidated automotive systems. In this context, graphics sharing" refers to "Graphics in the form of bitmap, scene graph or drawing commands generated on one ECU, transferred for display from another ECU (or between virtual machine instances)." Distributed HMI compositing technologies refer to "Methods to turn a multi-ECU system into what appears and acts as a single user-experience."

The underlying challenge is producing a consistent distributed HMI experience across distributed and diverse multi-ECU systems (while fulfilling other system design constraints).

This includes the following aspects:

- Top-level compositing across domains for the same physical display, with or without mixed safety levels
- Diverse operating systems and HMI technologies
- Graphics rendering on virtualized / consolidated systems
- Distributed HMIs that either are, or appear to be, controlled by one single system
- Graphics transfer encoding/technology and composition

Different approaches to address the following graphical use cases are discussed:

1. Multiple ECUs or operating systems sharing a single display
2. Multiple operating systems sharing a single GPU
3. Multiple ECUs or operating systems working together to create a UI in unison
4. One ECU or operating system providing graphical content to multiple ECUs or operating systems

Five primary solution categories for distributing graphics and HMI between cooperating systems have been identified in the project and each is exemplified and studied from its relative strengths and weaknesses.

This paper provides an overview of the technology categories, specific implementations of them, and use cases where these technologies can be applied. Each of these technologies are compared and guidelines are provided that assist in choosing the right technology for any use case. The white paper clarifies these alternative approaches and gives guidance for which method to choose in a given situation. The project ultimately aims to produce guidelines and shared standards described in openly licensed specification and code. We hope to confirm implementation of open specifications among multiple platform and graphics technology vendors. The methods discussed here are practically used and case studies are highlighted to prove the concepts. They include real-world demos of open

source technologies and a few proprietary implementations for illustration purposes. By using these technologies, it is possible to create a complex automotive multi-ECU HMI that appears and acts as a single ECU/system HMI.

Graphics Sharing Categories

Development of a complex automotive HMI has a long history and during this time, a lot of different technologies and systems were created and integrated. New driver assistance systems, new features of infotainment systems and handheld devices like mobile phones continue to appear in the automotive environment and need to be integrated into one system to provide a harmonic experience to a driver and other passengers. Along with other requirements, this integration needs technologies that will allow different devices and systems to communicate, share state and exchange graphical content. This is a challenge especially because some of the devices are originally not developed for in-car use.

To overcome these challenges, several solutions, communication protocols and frameworks have been created. Some of them are proprietary and others are open source and used in non-IVI environments. To handle this challenge, a standardization and robust implementation of sharing technologies is required, and categorization of different approaches is a good step forward in this direction.

Sometimes it might be difficult to assign a concrete sharing technology to a single category. Because of some implementation details, it might fit to several categories at the same time, but this is not a concern of this white paper. Categories should provide an abstraction of complex technologies to design the system without a detailed understanding of concrete implementation.

The following five categories are suggested:

- **Surface Sharing**
  Sub-category: **Virtual Display –** Full display transfer by encoding as a video stream.
- **API Remoting**
- **Shared State, Independent Rendering**
- **Display Sharing**
- **GPU Sharing**

Each category is described and compared in the following chapters.

Surface Sharing

Surface sharing distributes already rendered graphical content representing the intended graphics from the application. A surface is represented as a two-dimensional image (bitmap) in memory, which can be described with width, height, pixel format and some additional meta-data. Along with the image data, other information (e.g., touch events) can be shared but in terms of size, image data would have by far the biggest share. Therefore, sharing of image data should be the driving point for optimization during the definition and implementation of the sharing mechanisms.

When possible, shared memory between systems is desirable. Distributed systems without access to common memory must share all data over a network. To reduce the bandwidth usage, video encoding and decoding hardware that efficiently describe only the differences between subsequent frames can be used with reasonable performance.

Use Cases:

1. Navigation surface rendered by infotainment unit needs to be shown on Instrument cluster.
2. In a virtualization environment, the operating system controlling display wants to show a surface from another operating system.

Surface sharing requires a communication protocol to request or notify about new available graphical content in the system, to forward touch events, and to control the sharing in general. This results in some modification of standard graphical applications. To avoid modification of standard graphical application, virtual display implemented with Waltham allows standard applications designed for a Wayland-based environment to use remote displays.

Wayland is a set of libraries designed to cover messaging between an application and a graphical compositor. Wayland protocols are the high-level definitions of the messages exchanged between client and server. The server using Wayland to listen to clients is a Wayland compositor and the clients are graphical applications. Clients send their surfaces to be shown on display to the compositor. The compositor controls the display and shows client surfaces on display.

Waltham builds on the Wayland protocol but implements a capability to communicate over TCP/IP i.e., it is designed to work with distributed systems. Standard single-system Wayland communication is not appropriate for network traffic because it uses file descriptors to share client surfaces with the server, which suggests this communication is running on a single operating system kernel.

Design Considerations:

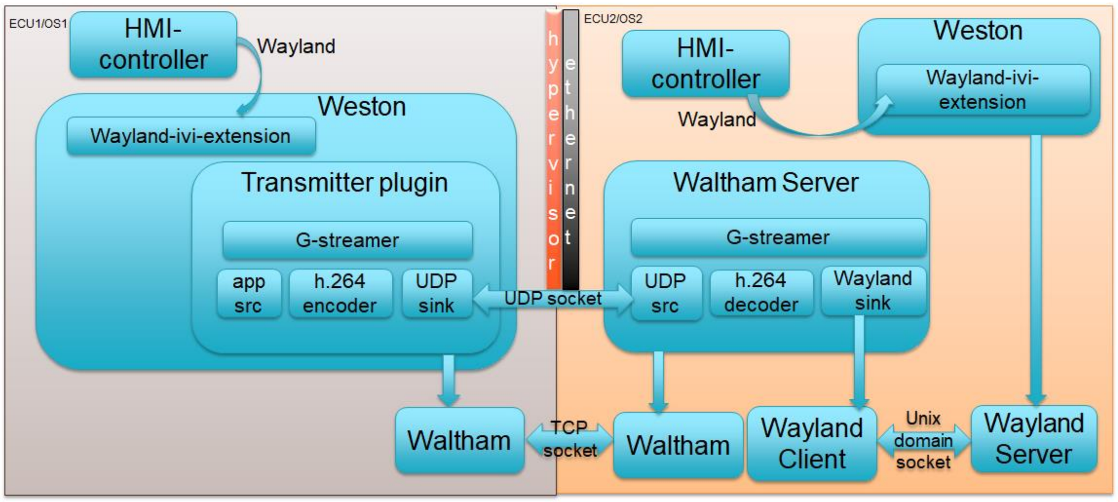
1. In case of distributed systems, high memory bandwidth is consumed for transport of pixel data. Pixel data can be encoded to save bandwidth but there will be loss of quality.
2. Depending on the available bandwidth, HMI/animation performance can vary.
3. Smoothness of animation and UI response additionally depends on network latency.

Virtual Display

Virtual Display is a special sub-category of Surface Sharing.   An important characteristic of the Virtual Display concept is that the entire display content is transferred instead of transferring content from single applications. The applications that provide the content are typically presented with a display which act likes a physical display, but is not necessarily linked to a physical display, so the middleware and applications can use it as usual. It is thus a “Virtual” Display. The implementation of Virtual Display on the producer system should be generic enough to look like the normal display and should take care of the transferring the display content to another HMI unit or another system.

The following diagram illustrates a full solution using the Virtual Display concept and Surface Sharing with communication protocols defined using Waltham libraries.

Example: Waltham concept implemented in Weston



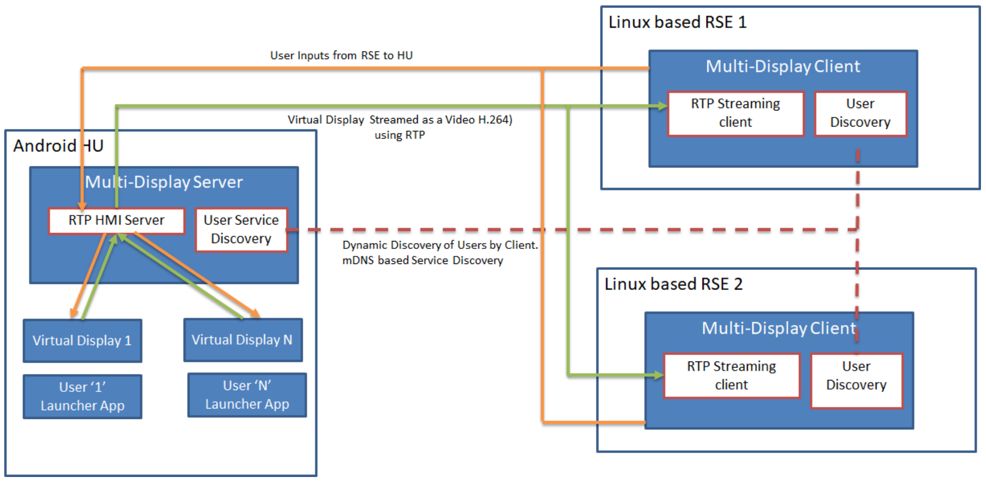
Whereas Wayland is a specification of a protocol and thus allows for any number of implementations, Weston is the widely accepted prototypical reference implementation of a Wayland compositor. Weston provides an example of Virtual Display implementation [1] and it can be configured to create a Virtual Display that is not linked to any physical display. Properties like resolution, timing, display name, IP-address and port can be defined in Weston.ini file. From this configuration, Weston will create a display area and all content which is placed in this area will be encoded to M-JPEG video compression format and transmitted to the configured IP address. The Transmitter plugin mentioned in the above diagram creates the Virtual Display and handles input events arriving over Waltham.

The source code that is required to realize a virtual display on Weston using Waltham can be found at Waltham [1], Wayland [2], Wayland-ivi-extension [3], and in separate repositories for the transmitter plugin and Waltham server.

[1] https://github.com/waltham/waltham
[2] https://gitlab.freedesktop.org/wayland
[3] https://github.com/GENIVI/wayland-ivi-extension

Example: Android Virtual Display

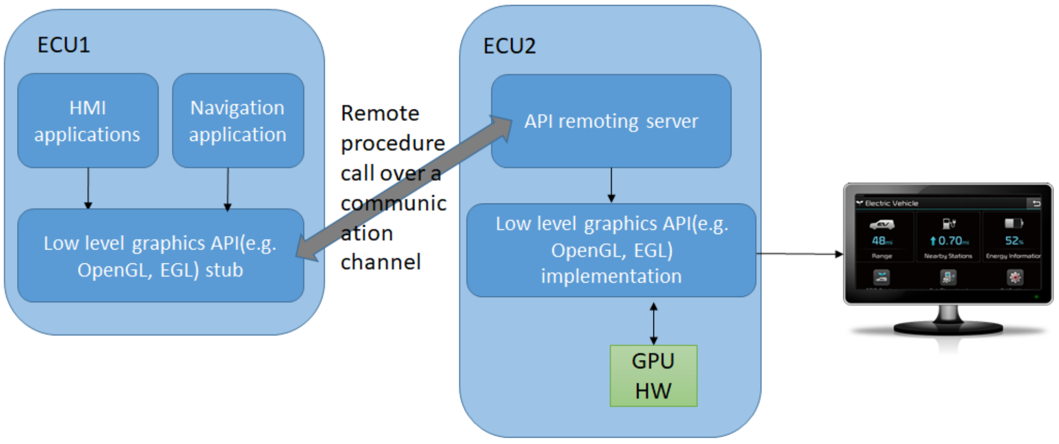
Android® has support for creation of multiple virtual displays and applications can render to these instead of the physical display. Android provides capabilities to access the framebuffer of a virtual display. Sharing of this framebuffer over network or by means of memory sharing is left to individual implementations. One case study in the project is described below:



The above solution from AllGo is explained in this video [4] and in the published Tech Brief on Virtual Display.

API Remoting

API Remoting involves taking an existing graphics API and making it accessible through a network protocol, thus enabling one system to effectively call the graphics/drawing API of another system. In some cases, an existing programming API such as that of a local graphics library (e.g., OpenGL or higher level drawing primitives typically used for a 2D interface), is simply extended to provide network-capability turning it into an RPC (Remote Procedure Call) interface. In other cases, new custom programming interfaces might be created that are specially tailored for distributed graphics.


ECU1
HMI applications
Navigation application
Low level graphics API(e.g. OpenGL, EGL) stub
Remote procedure call over a communication channel
ECU2
API remoting server
Low level graphics API(e.g. OpenGL, EGL) implementation
GPU HW

[4] https://www.youtube.com/watch?v=Joj1GS2qRws

Use Cases:

1. A device with no graphical capability wants to show graphical content on a display connected to a remote server.
2. In a virtualized system, an operating system with no access to GPU wants to render graphical content.

Design Considerations:

1. Depending on the available bandwidth HMI/animation performance can vary.
2. Smoothness of animation and UI response additionally depends on network latency.
3. In the case of API remoting, libraries need updating, then this update needs to be carried on all relevant nodes in network.
4. If standard API (e.g., OpenGL) is converted to RPC, then compatibility of applications is maintained; otherwise, changes are needed for applications to use the custom APIs.

Example: RAMSES

RAMSES is a framework for defining, storing and distributing 3D scenes for HMIs. From a user's perspective, RAMSES looks like a thin C++ scene abstraction on top of OpenGL. RAMSES supports higher-level relationships between objects such as a hierarchical scene graph, content grouping, off-screen rendering, animations, text, and compositing. All those features follow the principle of API remoting, i.e., the commands (that define the state in the RAMSES case) can be sent to another system and the actual rendering is executed at the receiving side.

RAMSES distinguishes between content creation and content distribution; the same content (created with RAMSES) can be rendered locally, on one or more network nodes, or everywhere. The framework handles the distribution of the scenes and offers an interface to control the final composition - which scene to be sent where, shown at which time, and how multiple scenes and/or videos will interact together. The control itself is subject to application logic (HMI framework, compositor, smartphone app, etc.).

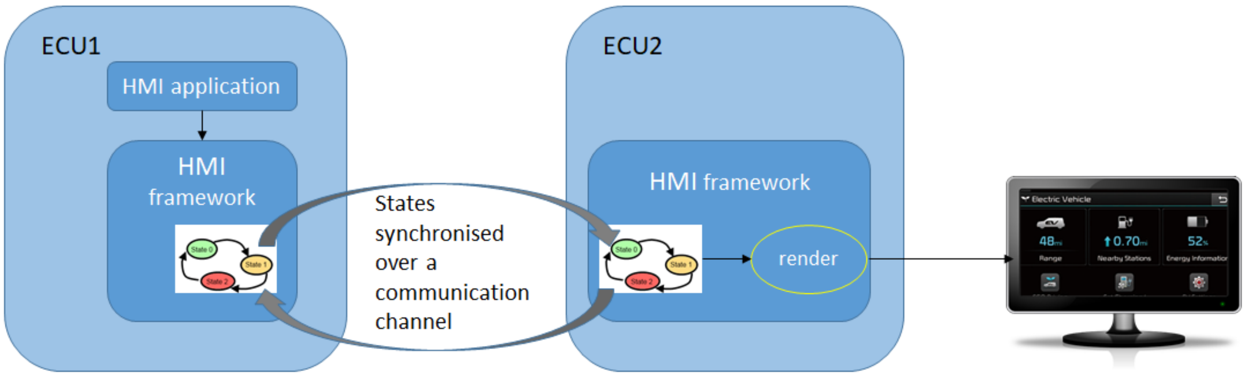
RAMSES is open-source licensed and available on GENIVI's GitHub account [5]. An overview and deployment examples as well as links to further documentation are available at the GitHub wiki pages [6].

RAMSES is a low-level framework, closely aligned to OpenGL. Migrating from an existing OpenGL application to RAMSES usually involves providing a software wrapper that creates RAMSES objects (shaders, geometry, rendering passes, transformation nodes, etc.) instead of sending OpenGL commands directly to the GPU. The main difference (and thus cause of migration effort) is that OpenGL sends a command stream per rendered frame, whereas RAMSES requires a scene definition which is updated only on change. For example, a static image would have to be re-rendered in every frame, starting with glClear(), setting all the necessary states and commands, and finishing with eglSwapBuffers(). With RAMSES, the scene creation code would look similar to an OpenGL command stream, but once the scene has been defined, its states and properties don't have to be re-applied every frame, only changes to it. Migrating from OpenGL to RAMSES is comparable to migrating from an OpenGL-centric rendering engine to a scene-graph-centric rendering engine. Depending on application, this effort may vary.

[5] https://github.com/GENIVI/ramses/
[6] https://github.com/GENIVI/ramses/wiki

Shared State, Independent Rendering

The Shared State category refers to rendering independent but coordinated UIs by Multiple ECUs or operating systems, by sharing only the HMI defining state and data rather than direct graphical elements. UI state such as the position of window, other data defining the displayed content and input events are distributed over a communication channel with constant notification when values change. By both sides implementing the same graphical elements and look-and-feel, this can still provide a synchronized user experience where displays connected to different ECUs or operating systems appear as if they belong to a single system. At the same time, it decouples the systems and often minimizes the required data transfer.



In the above diagram HMI application on ECU1 was able to render to a display on ECU2 by means of sharing the state in HMI framework .The HMI framework automatically delivers updates of changed values and thereby synchronizes the states on both ECUs. This way the HMI application can indirectly affect what is drawn. Also, configuration can be done in a way that the HMI application does not even know where it is rendered and displayed. Events from input devices result in change of state in HMI framework on ECU2. States are synchronized between ECU1 and ECU2 HMI frameworks. Thus, a completely interactive UI can be realized.

Comparison:

The boundary between Shared State method and API Remoting method can be somewhat blurred in some hybrid solutions. The key differentiator in terms of categorization is that API Remoting is slanted towards one system directly controlling *what* and *how* things should be drawn on the receiving system, whereas the Shared State method primarily exchanges some useful underlying data. In Shared State, the decision for how and what to draw based on that data is done by the receiver. This distinction may seem less relevant when designing the combined system and graphics for a seamless HMI in a setup of multiple ECUs, because of the intent to implement an HMI with a seamless interface. Therefore, the graphical behavior of the receiving system is often well known or even specified by the system designer. However, there is still a distinction between exchanging just the data model, compared to explicit “drawing instructions”.

Use Cases:

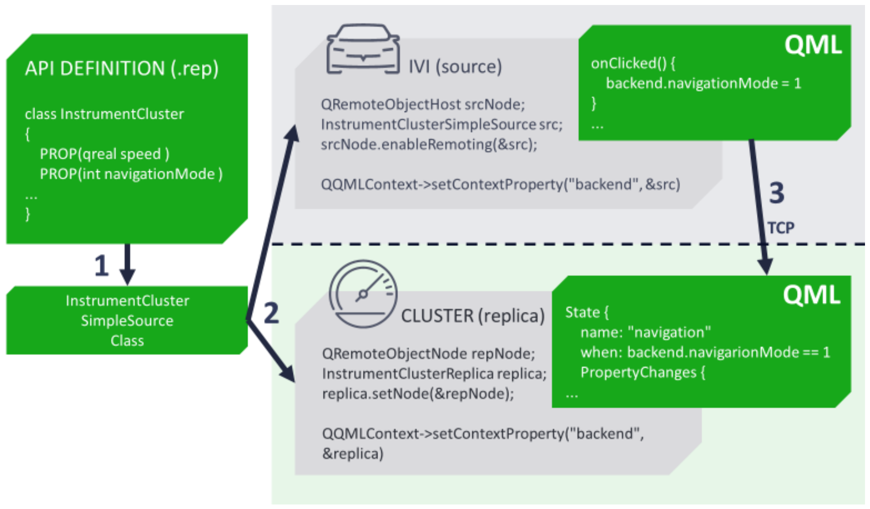
1. A user wants to drag and drop a phone contact list from multi-media system display to Instrument cluster display.
2. A user wants to drag and drop the navigation application from multi-media system display to Instrument cluster display.

Design Considerations:

1. Seamless user experience depends on network bandwidth and latency.
2. Using only shared state results in duplication of UI resources (images, scene graph, navigation data). On the other hand, sharing these resources over network impacts performance.
3. Synchronized software update is required between ECUs using shared state concept.
4. Although this is true for most of the categories involving network traffic, synchronizing state between the collaborating parts could come in conflict with any real-time requirements and the designer must be well aware to avoid any race-conditions.
5. Shared state is often a feature of a UI toolkit. If the same UI toolkit is available on different operating systems, then this approach is rewarding in terms of portability.

Example: Using Qt remote objects for shared graphics state

Qt Remote Objects [7] is an inter-process and domain communication module developed for Qt. It is designed as a generic framework for easy proxies to remote data objects and not designed specifically for graphics data. Nonetheless, it can be utilized to implement shared state between two or more objects and play a part in such a graphics setup. Below is an example diagram showing Qt Remote Objects to share navigation state between IVI and cluster.



```
class InstrumentCluster
{
   PROP(qreal speed )
   PROP(int navigationMode )
...
}
```

```
QRemoteObjectHost srcNode;
InstrumentClusterSimpleSource src;
srcNode.enableRemoting(&src);

QQMLContext->setContextProperty("backend", &src)
```

```
onClicked() {
   backend.navigationMode = 1
}
...
```

```
QRemoteObjectNode repNode;
InstrumentClusterReplica replica;
replica.setNode(&repNode);

QQMLContext->setContextProperty("backend",
&replica)
```

```
State {
   name: "navigation"
   when: backend.navigarionMode == 1
   PropertyChanges {
...
```

1) A QtRO compiler .rep file is shared between the entities and used to define an API to be shared (properties, signals and slots)
2) The QtRO compiler generates SimpleSource class that can be inherited or used as is by creating source and replica objects and exposing those as QML context properties.
3) Exposed properties can be used in QML application code [8], and changes made to properties are propagated to all replica objects. The cluster goes to navigation state when IVI changes navigationMode property in onClicked() handler.

[7] https://doc.qt.io/qt-5.12/qtremoteobjects-index.html
[8] https://doc.qt.io/qt-5/qmlapplications.html

Example: Harman shared state concept

One case study that was demonstrated in a previous GENIVI Tech Summit was the system using shared-state to implement innovative and interactive moving of an interactive navigation map and some other features by dragging from an IVI system to a cluster system display.

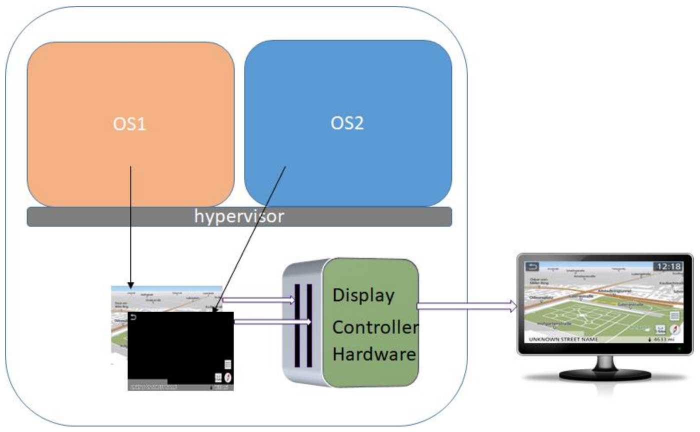
For more information, please view the associated Tech Brief [9] and presentations from the 2018 GENIVI Tech Summit [10] in Bangalore, India.

Display Sharing

In display sharing, a display is shared between multiple operating systems, with each of the operating systems thinking that it has complete access to the display. The operating systems are co-operative so that content on the screen is meaningful. Also, it could be that specific areas of the display are dedicated to each of the operating systems.

Realizing Display sharing requires the "Hardware Layer" feature in Display controller or linked image processing hardware. Each Hardware Layer holds a framebuffer and supports operations (e.g. scaling, color key and so on) on the pixels. Frame buffers of each of the Hardware layers are combined using a fast (hardware assisted) copy operation to form the final image which goes to the display.

Each operating system renders to one or more Hardware layers. Each Hardware layer is associated with one operating system.



In the above diagram, the black slit in the display controller hardware can be imagined as hardware layer, with each layer accepting a frame buffer. The display controller has composition capability* and combines the hardware layers and sends final image to display.

**Unlike a software compositor such as a Wayland compositor, content is not expected to be animated, scaled or repositioned here. Layer composition usually combines the graphical layers as they are, applying only a layer ordering (graphics from one layer hides the layer behind it but where there is no graphics, the lower layer can show through), with possible settings for transparency and masking.*

[9] https://at.projects.genivi.org/wiki/download/attachments/16026116/GENIVI_HARMAN%20Shared%20State%20Rendering_TechBrief_20180414.pdf
[10] https://at.projects.genivi.org/wiki/display/WIK4/GENIVI+Technical+Summit+Session+Content+2018

Use Cases:

1. A display shared by instrument cluster and multi-media system. If each of these are controlled by different operating systems in a virtualized environment, display sharing can be used.

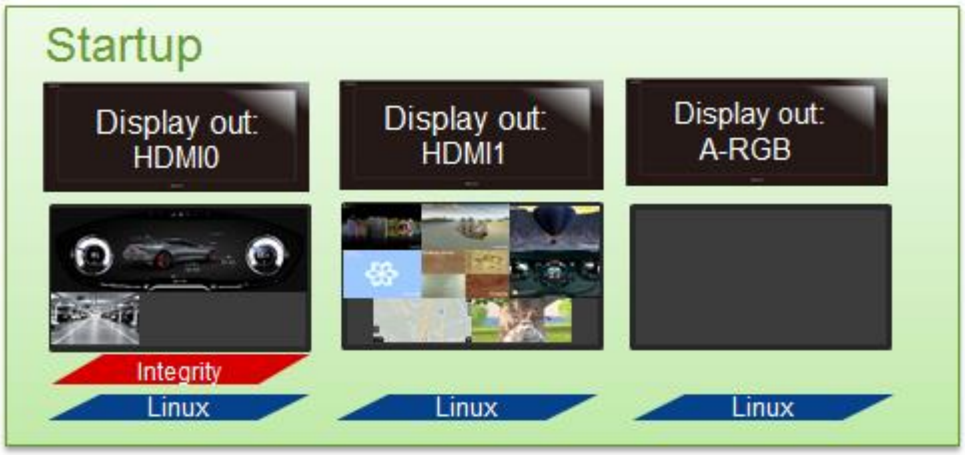
If each of the operating systems renders to specific area of the display, then using display sharing technology operating systems can work without knowing about each other (i.e. there is no need of communication between operating systems to share the display). However, if the operating systems present content on a display which is dependent on other operating system then communication is required.

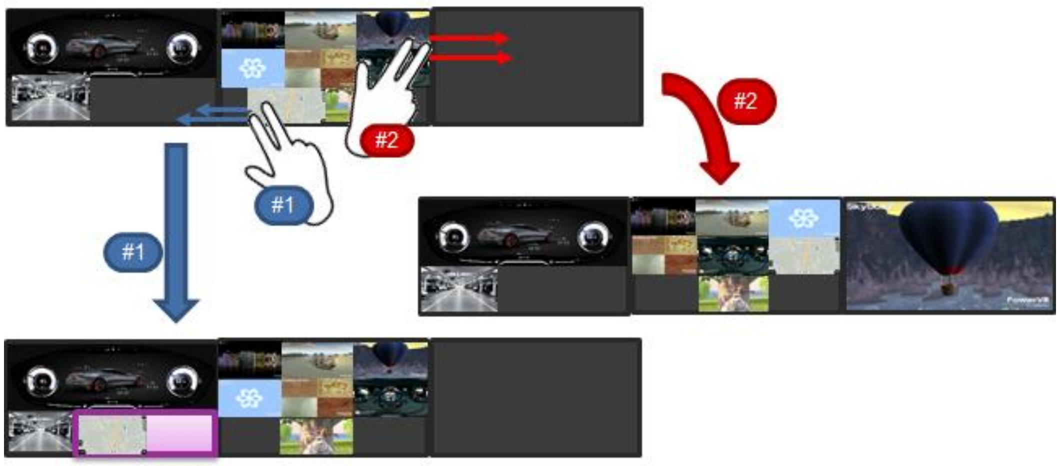
Where hardware display layers are not available, a form of Display Sharing may be provided in software by the Display Manager of certain Hypervisors. The Display Manager provides a mechanism for guest VMs to pass display buffers, which it then combines.

Design Considerations:

1. If multiple operating systems render to a fixed area of display then display sharing can be used without any communication between operating systems. The setup requires a hypervisor or similar coordinator and specific implementations of those are often closed-source.

Example: Renesas R-Car Canvas Demo





The Renesas R-Car Canvas Demo demonstrates consolidation of cockpit functionality onto a single H3 SoC running on a Salvator-X board driving three displays and multiple operating systems. In the version discussed here, the Integrity Multivisor hypervisor is used to virtualize an Integrity RTOS instance running a cluster demo and Linux operating system instance running an IVI stack and other functionality. The first display is shared between Integrity and Linux, whilst the second and third display is dedicated to Linux. For display one, Integrity and Linux are assigned different hardware display layers in the VSPD hardware block, with Linux being placed behind the Integrity cluster in the Z-ordering to ensure the Linux applications cannot overwrite the cluster graphics.

More information and video demonstration is in this presentation [11] from the 2018 GENIVI Tech Summit [12] in Bangalore, India.

GPU Sharing

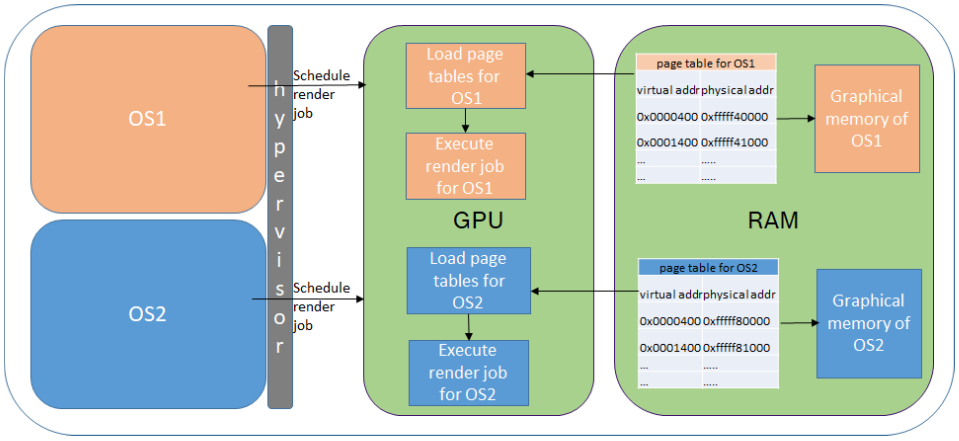
GPU sharing is referred to sharing of GPU hardware between multiple operating systems running on a hypervisor. When GPU is shared, each operating system works as though GPU hardware is assigned to it.

GPU Sharing can be implemented by:

1. Providing virtualization capabilities to the GPU hardware
2. Using a virtual device which schedules a rendering job on GPU by communicating with the operating system holding GPU hardware access.

GPU Sharing Using Hardware Virtualization Capability:

Modern GPUs are equipped with their own MMU. Additionally, GPU hardware can implement functionality to identify the operating system that is scheduling a rendering job. One way of doing this is to use a domain ID of the operating system. When a rendering job is executed, a GPU with the capability to track such IDs can load the page tables corresponding to that domain (operating system). It ensures that a render job from one operating system cannot access the memory dedicated to other operating systems. Additional capabilities can be provided for further isolation. For example, if any of the domains is safety relevant then GPU faults of non-safe domains should not affect the other and it should be possible to prioritize the render jobs according to criticality.



[11] https://at.projects.genivi.org/wiki/download/attachments/28412356/canvas-demo_public_embedded_lowres_video.pdf
[12] https://at.projects.genivi.org/wiki/display/WIK4/GENIVI+Technical+Summit+Session+Content+2018

GPU Sharing Using Virtual Device:

Domains running on the hypervisor can also be categorized as host and guest. This is most typically seen in Desktop virtualization solutions (VirtualBox, VMWare, Hyper-X, QEMU/kvm…) that allow running Windows in a Virtual Machine on a workstation running Linux or MacOS or opposite combinations, but the principle can be applied also in embedded virtualization scenarios.

In these setups, the host domain has full access to the GPU hardware whereas the guests use a different GPU driver that communicates with host. In other words, the guest operating system accesses a *Virtual* device for the graphics hardware. By using slight modifications to the guest operating system (paravirtualization), good performance can be achieved compared to a full virtualization of the GPU hardware. After the specialized driver has communicated with the host over a mechanism provided by the Hypervisor, the rendering job of the guest is then scheduled on the GPU by the Host system.

Use Cases:

1. Instrument cluster and infotainment unit running as different domains on a hypervisor. Both need access to GPU hardware.

Design Considerations:

1. GPU sharing requires a lot of attention to security aspects, specifically when one or more of the domains are safety relevant. By mistake or malicious intent it is theoretically possible to execute dangerous code – the most obvious being a *denial-of-service* in which the safety-critical system is starved of resources by the execution of graphical operations initiated by the non-critical system. In some systems, carefully crafted graphical operations could potentially even manipulate memory to achieve other types of attacks. A system that can provide full memory isolation between domains can solve the latter, and a system that is able to separate timing/execution resources might solve the former.

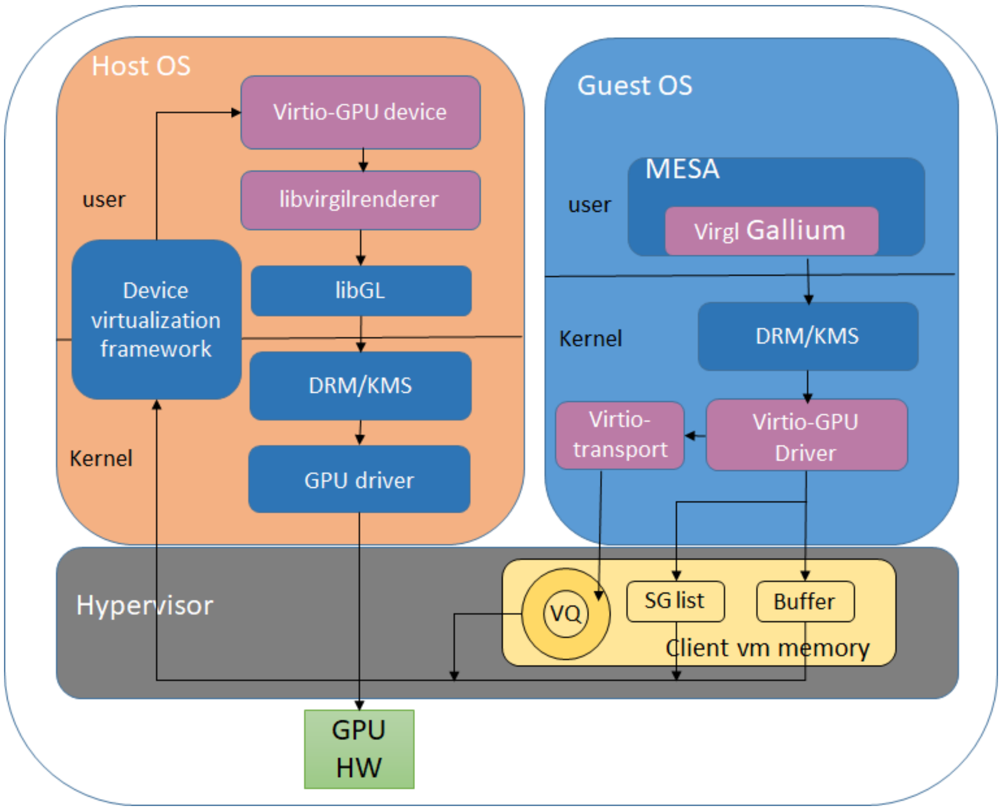
2. GPU sharing without GPU virtualization capabilities introduces communication and memory copy overhead. This affects performance.

Example (Hardware): GPUs with built-in virtualization support

GPUs from Intel and imagination tech are examples of GPUs providing virtualization capability.

Example (Software based solution / Virtual Device): virtio-gpu 3D

virtio-gpu 3D is a virtual device available in Linux systems. This is an open source implementation, where some modifications have been done on guest side to the Mesa library (an open-source implementation of many graphics APIs, including OpenGL, OpenGL ES v1/v2/v3 and OpenCL). Applications on the guest side still speak unmodified OpenGL to the Mesa library. But instead of Mesa handing commands over to the hardware they are channeled through virtio-gpu to the backend on the host. The backend then receives the raw graphics stack state (Gallium state) and interprets it using virglrenderer from the raw state into an OpenGL form, which can be executed as entirely normal OpenGL on the host machine. The host also translates shaders from the TGSI format used by Gallium into the GLSL format used by OpenGL. The OpenGL stack on the host side does not have to be Mesa and could be some proprietary graphics stack.



Architecture using Virtio-GPU 3D

Conclusions

This paper has demonstrated a set of categories and just about any approach to graphics interaction between distributed and integrated systems can be identified as part of one of these categories, or occasionally a combination of both.

By having a common set of general approaches, the design of a particular system can be started on a higher level using this shared vocabulary and understanding of the consequences of approaches, and then be refined with the details, rather than having to consider all at once without structure.

Depending on each system's design constraints, different approaches will fit differently, and this document described design constraints and characteristics to guide the selection process.

Authors

- Harsha Manjula Mallikarjun, Bosch
- Eugen Friedrich, ADIT
- Qt Company
- Stephen Lawrence, Renesas
- Gunnar Andersson, GENIVI

+ some content from previous Tech Briefs with additional authors.