KHRONOS GROUP

OpenWF™ COMPOSITION

OpenWF™ DISPLAY

Updated 28 November 2011

Table of Contents

1. Overview 2
2. APIs for the Windowing System 3
   2.1. OpenWF Composition 4
   2.2. OpenWF Display 5
3. Features and Benefits 6
4. Memory Bandwidth Savings 7
   4.1. GPU-based Composition (without YUV support) 7
   4.2. GPU-based Composition (with YUV support) 8
   4.3. Overlay Composition 9
   4.4. Conclusions on Memory Bandwidth 10
5. Optimized Control Paths 11
6. Deployment Possibilities 12
7. Compliance and Conformance Testing 13
8. Technical Support 13
9. Conclusion 14
10. Contacts & Resources 14
11. Glossary 15
12. Acknowledgements 15

OpenWF COMPOSITION OpenWF DISPLAY

Introducing two new APIs for building composited windowing systems

Audience

The intended audience for this paper includes creators of embedded devices with displays, windowing system developers, graphics/display driver writers, graphics/display silicon vendors, mobile platform vendors and OS vendors. It is of particular relevance for OEMs sourcing graphics and display hardware from multiple vendors as well as hardware vendors supporting multiple operating systems.

1. Overview

Embedded devices are increasingly expected to offer sophisticated user interfaces that combine rich graphics with multimedia content. Graphics and display hardware technologies have evolved to achieve these visuals with significantly higher efficiency than traditional CPUs, delivering greater performance, decreasing memory bandwidth usage and increasing battery life. Making use of this variety of hardware introduces fragmentation as software needs to be adapted to each hardware configuration.

A platform's Hardware Abstraction Layer (HAL) for display and graphics technology allows the applications and middleware layers above to be deployed across a range of hardware without costly porting activities. OpenGL is an example of a graphics HAL that allows portable software to take advantage of a wide range of 3D hardware accelerators.

Windowing systems allow screens to be shared by multiple applications, ensuring that the graphics provided for each application's window is sensibly merged onto the screen. This requires the graphics and display drivers to respect the intentions of the windowing system, which commonly means considerable OS-specific porting work on the part of the device manufacturer when moving to new hardware.

The OpenWF APIs provide an OS-independent and hardware-neutral foundation for building compositing systems, particularly suited to implementing windowing systems. OpenWF acts as a HAL to achieve composition of content and configuration of display devices. The interfaces are designed for use by a single user which could be a central windowing system or, in an application-specific system, may be the application itself.

2. APIs for the Windowing System

OpenWF defines separate interfaces for composition and display control, reflecting the traditional model of independent hardware for each function. Figure 1 depicts OpenWF Composition and OpenWF Display being used by the central windowing system as part of a full graphics and multimedia system.

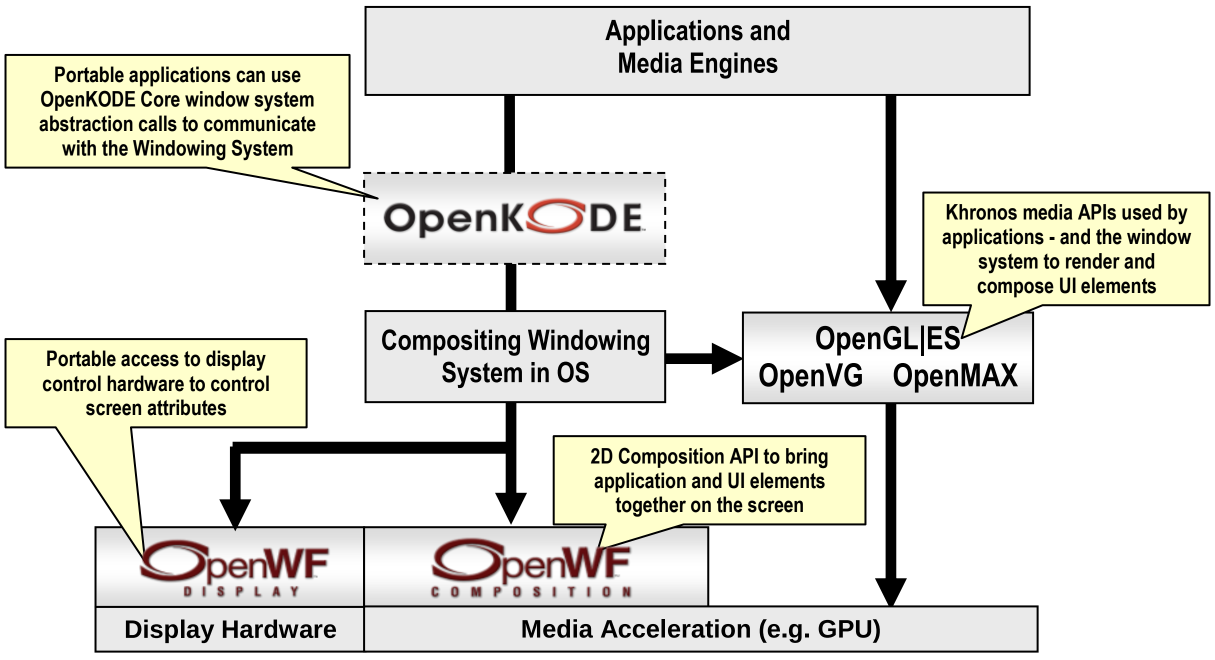
Applications and Media Engines
Portable applications can use OpenKODE Core window system abstraction calls to communicate with the Windowing System
OpenKODE
Khronos media APIs used by applications - and the window system to render and compose UI elements
Compositing Windowing System in OS
OpenGL|ES OpenVG OpenMAX
Portable access to display control hardware to control screen attributes
2D Composition API to bring application and UI elements together on the screen
OpenWF DISPLAY
OpenWF COMPOSITION
Display Hardware
Media Acceleration (e.g. GPU)

Figure 1 - System overview

The OpenWF technologies complement the existing Khronos APIs, defining the underlying route to the display for advanced graphics and multimedia content generated using APIs such as OpenGL|ES, OpenVG and OpenMAX.

In a typical system such as that shown above, the application would access the features of OpenWF via either the native window system or through OpenKODE windowing. For instance, this could take the form of the window system offering the ability to give each window a percentage opacity value to make the window translucent.

2.1. OpenWF Composition

A wide range of visual scenarios can be implemented in terms of 2D layering. OpenWF Composition offers a clean API for achieving system-wide composition of layered content, such as 3D content rendered through OpenGLES hardware and video content decoded using OpenMAX-IL hardware.

Key rendering operations are:

- Scaling (with control over filtering)
- Rotations (90-degree increments)
- Mirroring
- Alpha-blending (per-pixel and global)
- Alpha masking
- Solid background color

The scalable nature of the Composition API allows system adaptors to use acceleration hardware from low-end DSP-based implementations all the way through to high-end Graphics Processing Units. Both render-to-memory and render-to-screen (a.k.a. overlay) hardware can be used to process the content. The technology can be used on systems with unified and non-unified graphics memory.

This implementation flexibility enables the driver writer to dynamically optimize the way content gets merged onto the display according to the specific hardware platform being used. Along with performance/throughput gains, memory bandwidth usage and power consumption can be reduced. This is particularly useful for maximizing battery life during long-running use-cases such as viewing HD video with subtitles.

The Composition API supports both user-driven and autonomous rendering. User-driven rendering means that the windowing system decides when to recompose the scene. Autonomous rendering means that content can be rendered, composed and displayed without the windowing system being involved. The Composition API retains the scene structure information provided by the windowing system, meaning the composition driver has all the information it needs to recompose the scene when new content arrives. On platforms with the appropriate hardware, this can be used to allow hardware video playback directly to the display without per-frame CPU intervention. See section 5 for more details on autonomous composition.

2.2. OpenWF Display

Embedded devices with multiple displays and connections for external displays are becoming increasingly common. These displays offer various degrees of configurability and commonly require custom routines for set up and mode selection. OpenWF Display provides a consistent way to query and control the state of these displays.

Key features are:

- Dynamic discovery of external displays, e.g. cable attach detection
- Power control
- Mode-setting (resolution, refresh rate)
- Rotation and flipping control
- Pipeline management (scaling, rotation, mirroring, alpha masking, alpha blending)
- Retrieval of standardized display information (EDIDv1, EDIDv2, DisplayID)
- Content protection control, e.g. HDCP

The Display API is designed to allow the windowing system to be the focal point for display management. Support is provided for built-in displays, such as embedded LCD panels, and external displays, such as those connected by HDMI/DVI/S-Video. The cable attach sequence allows the system to tailor the display mode based on information dynamically reported from the display device that was connected. The wide range of display types is abstracted so that the windowing system can easily offer higher-level services, such as spanning of windows across displays.

OpenWF Display is a low-level abstraction allowing it to be compatible with a broad range of display control hardware without the need for costly software fallbacks. The Display API reflects common hardware constraints and behaviors such as limitations on the number of display pipelines a device can offer.

3. Features and Benefits

Feature	Benefit
Standardized Interface A well-defined standard abstraction for accessing composition and display control functionality.	*For implementers:* • Reduced hardware and software design costs. • Increased marketability of features. • Maximized opportunity of hardware functionality being utilized. *For users:* • Decreased time to integrate new hardware. • Decreased hardware switching costs. • Portability: Minimized rework of higher level software through consistent driver interfaces.
Extensive Conformance Tests Khronos provides adopters with a rich set of tests that verify compliance with the specification and provide an assessment of visual quality.	*For implementers:* • Reduced costs associated with developing in-house testing infrastructure. *For users:* • Provides a guarantee of quality. • Tests can be reused as a form of acceptance criteria when sourcing implementations.
Conformant Sample Implementations Khronos publishes an open source Sample Implementation of both OpenWF Composition and OpenWF Display that passes the Conformance Tests and can be used to generate reference images for comparison during testing.	*For implementers:* • Provides a concrete example to guide new implementations. • Acts as a focal point for resolving questions over intended behavior. *For users:* • Provides a quick way to try out and learn about the technology. • Offers a foundation to build a Windowing System in preparation for switching to a hardware-based implementation.
Optimal processing & data paths OpenWF describes the intended visual result rather than making assumptions about how the content gets onto the screen. This enables the driver writer to choose best data path and best processing hardware for the job.	• Increased battery life via the ability to use dedicated hardware with fewer gates than full GPUs. • Decreased memory bandwidth usage via the use of overlay composition. • Better performance via reduced processing overheads and ability to run composition in parallel with rendering.
Optimal control paths OpenWF supports streaming content directly to the display. For example, video data can be generated, composed and displayed without the need for per-frame intervention by the windowing system.	• Increased battery life via increased sleep time for the CPU that runs the windowing system; especially during long-running use-cases such as video playback. • Decreased latency between content rendering and display.

4. Memory Bandwidth Savings

Beyond pixel processing power, another important system characteristic of embedded devices is memory bandwidth utilization. The choice of composition architecture plays a significant role in determining how much bandwidth a given use-case will require. To illustrate this we look at an example case study involving a simple composition scene.

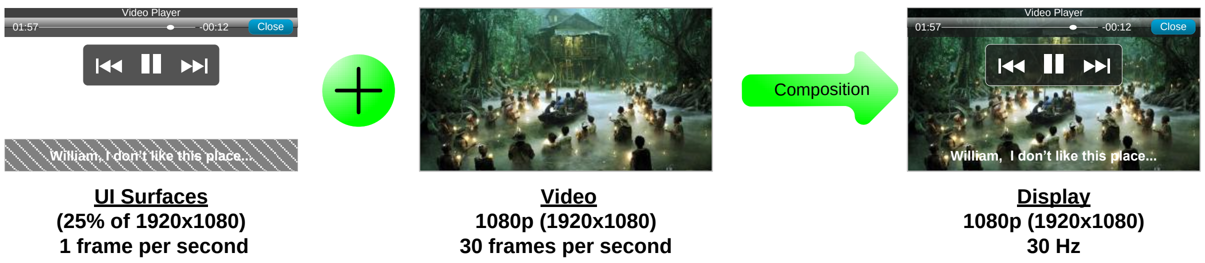

Figure 2 - Use-case overview

In the above scene we show a typical translucent User Interface (UI) being composited on top of full-screen video. The video is playing at 30 frames per second (fps) whilst the UI updates at a slower rate (1 fps). The external screen has a resolution of 1920x1080 and 25% of this is covered by UI rendered at this resolution to multiple 32-bit RGBA surfaces. The video is full HD resolution (1920x1080).

In our analysis we assume that the system has a Unified Memory Architecture (UMA) meaning that all subsystems share the same memory and access it via the same bus, except for the display controller that has a dedicated bus for sending output to the display. This means that the output of the video decoder is directly usable as input to the composition hardware, subject to data format compatibility.

4.1. GPU-based Composition (without YUV support)

In this example, a GPU is used to compose the two input surfaces into a single output surface for use by the display controller. We assume that the video decoder is performing the YUV->RGB conversion as part of the decoding process rather than the part of the composition stage. This reflects the fact that some GPUs do not have the ability to process YUV data directly.

To ensure the color fidelity of the video is preserved, it is output using a 24-bit RGB format. Some GPUs can not process such data directly as they require specific data alignments and layouts. In the following calculations we assume the GPU can process this directly, and so the figure represents a lower-bound on the bandwidth.

The GPU is required to compose full-screen frames at least as often as the rate of the fastest input, which is the video running at 30 fps. We assume updates to the UI surfaces don't cause additional compositions as they are processed together with the video surface updates.

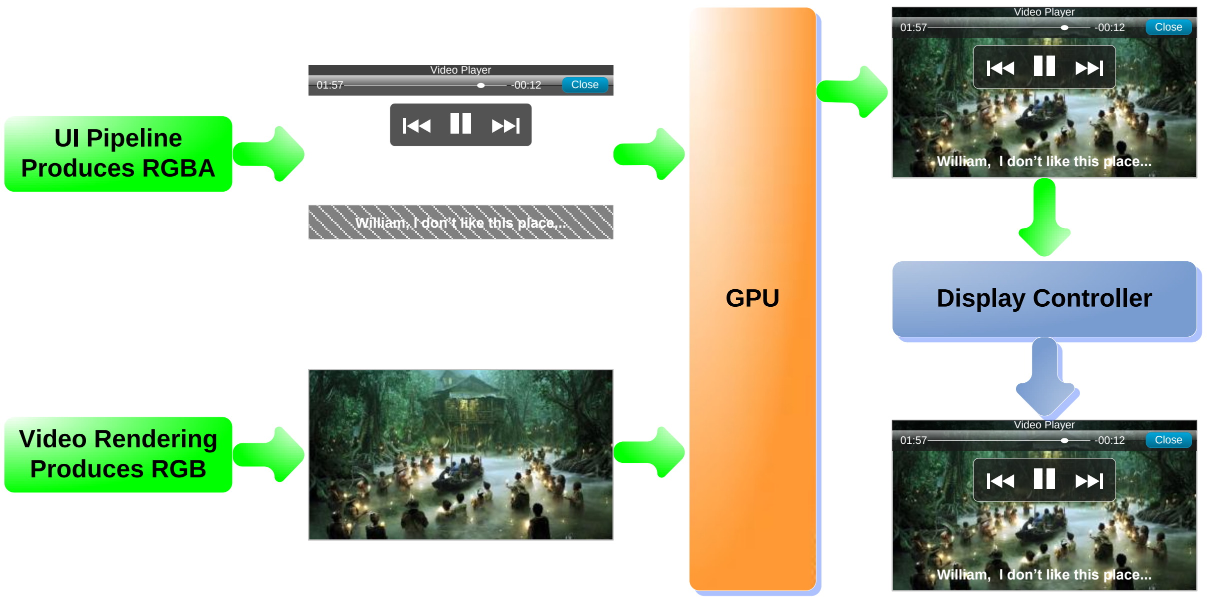
UI Pipeline Produces RGBA
Video Rendering Produces RGB
Video Player
01:57
-00:12
Close
William, I don't like this place...
GPU
Display Controller

Figure 3 - GPU Composition data-path

The total bandwidth required for composition is 810 MB/s.

Operation	Resolution (pixels)	Pixel size (B)	Frame rate (fps)	Bandwidth (B/s)
Rendering of UI surfaces	1920x1080x0.25	4	1	2,073,600
GPU read of UI surfaces	1920x1080x0.25	4	30	62,208,000
Rendering of Video surface	1920x1080	3	30	186,624,000
GPU read of Video surface	1920x1080	3	30	186,624,000
GPU write of display surface	1920x1080	3	30	186,624,000
Display read of display surface	1920x1080	3	30	186,624,000
				810,777,600

Table 1 - Bandwidth requirements for GPU composition without YUV support

4.2. GPU-based Composition (with YUV support)

Next we consider a similar system in which the GPU can read YUV data directly. YUV formats can provide a more compact representation of video data than RGB formats. In this example we pick the YUV420 format as it uses on average 12 bits per pixel (bpp) whilst achieving almost the same visual appearance as 24-bit RGB, halving the memory required to store a Video frame.

Note that Khronos rendering APIs such as OpenGLES and OpenVG do not currently expose YUV input capabilities. Future extensions for video processing are likely to allow this.

With the support for direct YUV processing, required bandwidth is reduced from 810 MB/s to 624 MB/s – which is about 23% less.

Operation	Resolution (pixels)	Pixel size (B)	Frame rate (fps)	Bandwidth (B/s)
Rendering of UI surfaces	1920x1080x0.25	4	1	2,073,600
GPU read of UI surfaces	1920x1080x0.25	4	30	62,208,000
Rendering of Video surface	1920x1080	1.5	30	93.312,000
GPU read of Video surface	1920x1080	1.5	30	93,312,000
GPU write of display surface	1920x1080	3	30	186,624,000
Display read of display surface	1920x1080	3	30	186,624,000
				624,153,600

Table 2 - Bandwidth requirements for GPU composition with YUV support

4.3. Overlay Composition

Finally we show a system that does not use render-to-memory hardware but instead utilizes the display controller's ability to composite multiple layers on-the-fly, a.k.a "overlays". In this case the display controller supports both translucent overlay planes and YUV video planes allowing it to handle all the composition operations.

The key difference to note between this and the previous examples is the absence of a display surface in system memory. This is because the output of display controller is sent directly over the display bus. We assume that the display controller has the ability to adapt its refresh rate so that it can be used at 30Hz to match the video rate, which might or might not require a remote framebuffer depending on the display type.

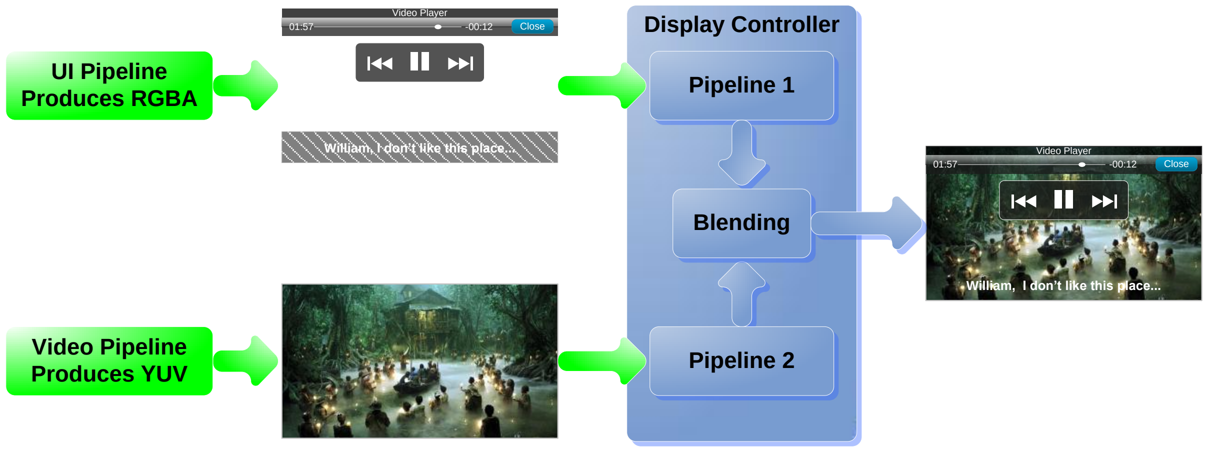
UI Pipeline Produces RGBA
Video Player
01:57
-00:12
Close
William, I don't like this place...
Video Pipeline Produces YUV
Display Controller
Pipeline 1
Blending
Pipeline 2
Video Player
01:57
-00:12
Close
William, I don't like this place...

Figure 4 - Overlay Composition data-path

Direct overlay processing requires 250 MB/s which is a 59% saving over YUV-based GPU processing and a 69% saving over RGB-based GPU processing.

Operation	Resolution (pixels)	Pixel size (B)	Frame rate (fps)	Bandwidth (B/s)
Rendering of UI surfaces	1920x1080x0.25	4	1	2,073,600
Display read of UI surfaces	1920x1080x0.25	4	30	62,208,000
Rendering of Video surface	1920x1080	1.5	30	93,312,000
Display read of Video surface	1920x1080	1.5	30	93,312,000
				250,193,600

Table 3 - Bandwidth requirements for overlay composition

4.4. Conclusions on Memory Bandwidth

OpenWF Composition and OpenWF Display allow driver writers to take advantage of the full range of hardware available for composition. Choosing wisely between the different types of GPU composition and overlay composition enables memory bandwidth saving of over 69% on some use-cases.

5. Optimized Control Paths

Traditional integration of composition systems relies on the graphics/multimedia driver signaling the windowing system every time a new frame of content is ready to be composed onto the screen. The OpenWF APIs do not have this limitation and permit composition to occur without per-frame windowing system interaction whilst still keeping the windowing system in control of where content is displayed.

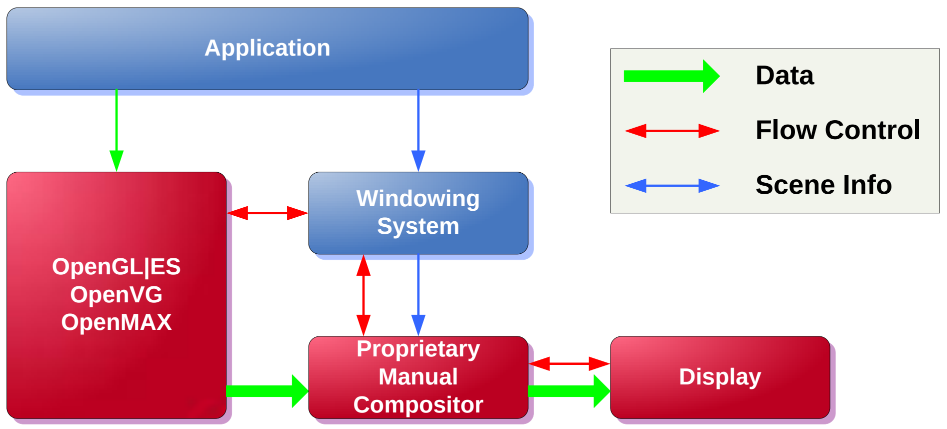

Figure 5 - Non-optimized control-path

Figure 5 depicts the traditional control-path in red. This consists of the signaling that provides notification of new content, transfers ownership of buffers throughout the chain of components and may also communicate audio-visual synchronization information.

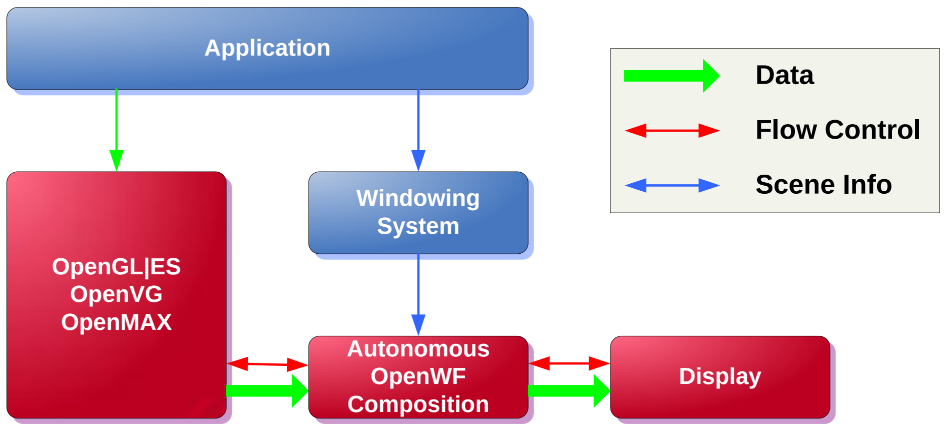

Figure 6 - Optimized control-path

Figure 6 depicts the optimized control-path enabled by OpenWF. Using this model, the windowing system need only be active whilst changing the scene geometry. On systems that contain dedicated hardware for graphics and multimedia, the main benefit is that it offers more time for the main CPU to sleep during long-running use-cases with mostly static scene geometry, e.g. video playback with subtitles. Such systems can also benefit from the reduced latency that comes from new frame notifications bypassing the windowing system.

6. Deployment Possibilities

OpenWF Composition and OpenWF Display are distinct interfaces reflecting the traditional separation of the hardware. The APIs can be used together in the same system or as standalone APIs. This independence enables system builders to take a phased approach to adopting OpenWF, for example by retaining an existing proprietary API for display control whilst migrating to OpenWF Composition for composition activities, or vice versa.

Figure 7 shows a typical data path in a system in which OpenWF Composition is implemented using a simple hardware blitter whilst OpenWF Display is a direct abstraction of a single display controller. Graphical and multimedia data, commonly stored as pixel buffers in video memory, is routed by the windowing system into the compositor which passes the results to the display controller.

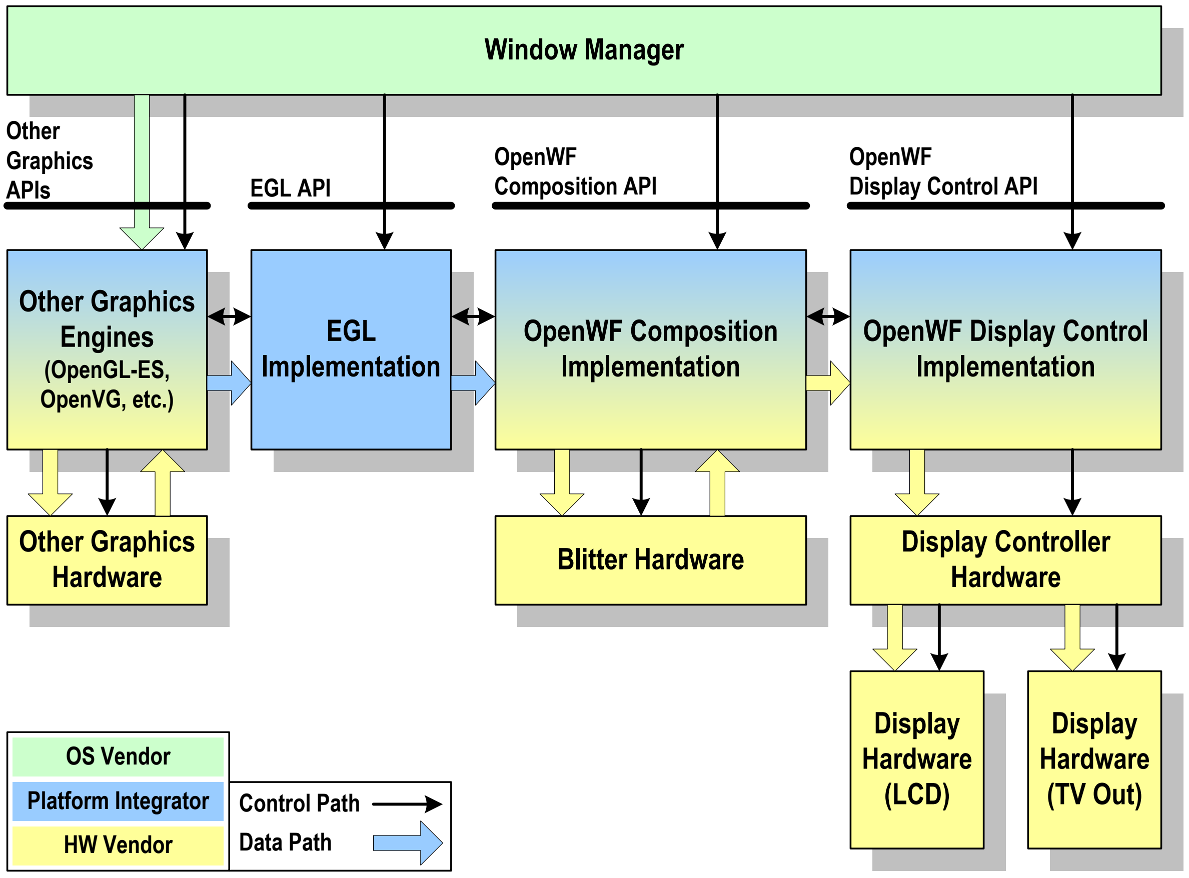

Figure 7 - A typical system based on OpenWF Composition and OpenWF Display Control

7. Compliance and Conformance Testing

Only products that have been certified by Khronos as being conformant with the specifications may use the OpenWF logos or trademarks. This means that any product displaying the logo or advertising compliance has successfully passed the extensive Conformance Test Suite (CTS) defined and provided by Khronos. The purpose of the test suite is to promote consistent, high-quality cross-vendor implementations. Test results are peer-reviewed by members of the OpenWF working group to ensure integrity.

The test suite includes comprehensive interface testing, rendering functionality and quality tests, stress tests and animated use-case based testing. Tests can be run in automated and interactive modes, with full recording of visual results. The framework is extensible so that members may contribute further test cases and improvements over time. The test suite is OS-agnostic and is designed to be ported to multiple operating systems. At the time of writing, specific ports are already available for Linux and Symbian OS.

Vendors gain access to the Conformance Test Suite by becoming Khronos Technology Adopters. Please see http://www.khronos.org/adopters/ for full details and benefits of becoming an adopter.

8. Technical Support

Users and implementers of OpenWF benefit from a range of resources.

Public Resources

Khronos offers open-source Sample Implementations of OpenWF Composition and OpenWF Display. These are provided under the permissive Khronos Free Use license (similar to the MIT license) and are run on Linux. The implementations are conformant and can be used as the basis of hardware-specific or optimized implementations.

The Khronos website also hosts a public forum for OpenWF for general discussion as well as soliciting feedback from working group members.

Khronos members-only

Members have access to the internal mailing list and have the opportunity to participate in the ongoing evolution of the specification as well as ability to input and track discussions on bugs, test improvements, limitations.

9. Conclusion

Integrating new graphics and display hardware with an existing software stack, such as an OS windowing system or application-specific engine, can take considerable time and resources. OpenWF Composition and OpenWF Display reduce this integration effort, giving OEMs a wider choice of hardware and a quicker time to market for devices.

OpenWF provides access to high-performance low-cost graphics functionality that can otherwise be left underused. Using the right hardware for the job can have a significant quantitative effect on the graphical and multimedia use-cases a device can support, as well maximizing device battery life.

10. Contacts & Resources

Khronos Group
www.khronos.org

OpenWF
www.khronos.org/openwf

OpenWF FAQ
www.khronos.org/faq/category/C265

OpenWF Working Group Forum
www.khronos.org/message_boards/viewforum.php?f=47

Khronos Group Public Relations
Elizabeth Riegel
elizabeth@goldstandardgroup.com
+1 740 649-7755

About Khronos
The Khronos Group is a member-funded industry consortium focused on the creation of open standards such as OpenKODE™, OpenGL® ES, OpenWF™, OpenMAX™, OpenVG™, OpenSL ES™, OpenML™ and COLLADA™ to enable the authoring and acceleration of dynamic media on a wide variety of platforms and devices. All Khronos members are able to contribute to the development of Khronos specifications, are empowered to vote at various stages before public deployment, and are able to accelerate the delivery of their cutting-edge media platforms and applications through early access to specification drafts and conformance tests. Please go to www.khronos.org for more information.

11. Glossary

HAL	Hardware Abstraction Layer. A general term for a software interface used by hardware independent software to make use hardware. OpenGL is a HAL that abstracts graphics acceleration hardware.
GPU	Graphics Processing Unit. Programmable hardware designed to accelerate graphics.
RGB	Standard colorspace used to encode computer graphics. Red/Green/Blue.
YUV	Standard colorspace frequently used to encode video data. Generally more efficient than RGB encoding when expressing colors for human visual perception.

12. Acknowledgements

Robert Palmer, Nokia (main author)
Pasi Keränen, Nokia
Jarkko Kemppainen, Symbio