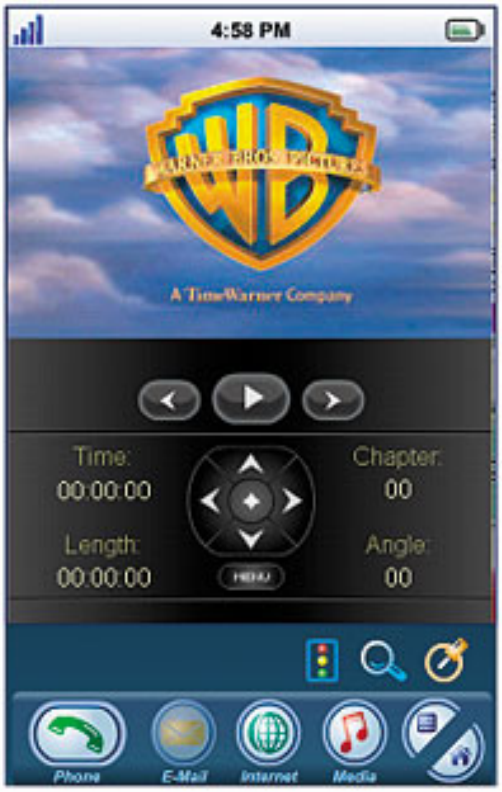
Building Automotive GUIs “In a Flash”



According to Adobe Systems, over 300 million mobile devices have graphical user interfaces (GUIs) based on Adobe Flash technology – a number that may exceed a billion by 2010. Developers of in-car navigation and infotainment systems are also beginning to embrace Flash, for a simple reason: it can reduce the time to build a GUI by up to 50%. In the past, software teams had to translate their GUI prototypes into C, C++, or Java code, a labor-intensive process that can take many months. Now, teams can prototype their GUIs with high-level Flash tools and run those GUIs directly on embedded Flash players, without having to write graphics code.

Flash is gaining momentum in the automotive embedded market for several reasons:

- Over a million graphics designers worldwide use Flash authoring tools, providing automotive teams with an immense pool of expertise to draw upon.
- Compared to desktop Flash players, new embedded Flash players (for example, Flash Lite 3) use less memory and provide faster rendering with less CPU overhead.
- Automotive-grade CPUs and graphics chips now support the frame rates needed for a pleasing Flash experience on VGA and larger displays.

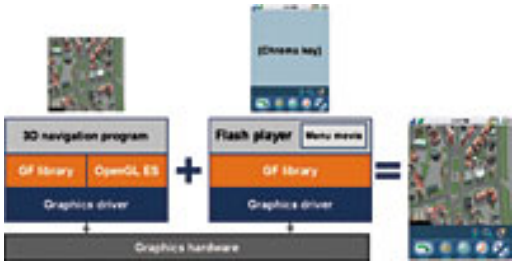
Flash provides a domain-specific environment for graphics and multimedia that offers almost endless possibilities for building user experiences. As a result, automotive developers can quickly implement animations and special effects. Also, Adobe certification of Flash players ensures that Flash-based applications look and

work the same across hardware platforms, allowing developers to create GUI components once and then deploy them across a family of products.

Nonetheless, to satisfy the requirements of automotive developers, an implementation of Flash must address several issues. Among them:
• How do you combine Flash content with other graphics programs, such as web browsers and 3D navigation applications?
• How do you make the Flash-based user interface perform consistently under all CPU load conditions?
• How can you keep the Flash-based user interface reliable? Can you monitor for failures and gracefully recover from them?
• How do you control how the Flash content interacts with operating system services, such as multimedia and video playback?

Let's explore how some of these issues can be addressed.

Integrating Flash with non-Flash graphical applications



[1]

Traditionally, a Flash player runs within a web browser or is launched from a windowing system. However, GUI development can be greatly simplified by turning this model on its head and making Flash the main environment that launches all other graphical applications, including web browsers, 3D navigation programs, and even other Flash applications. Flash excels as a screen manager, allowing the graphics designer to intimately control menu transitions and audio effects; it also simplifies customization by allowing designers and developers to freely position, resize, and configure graphical components.

Figure 1 shows an example of using Flash as a screen manager. The program on the left draws 3D maps in OpenGL ES, a standards-based 3D API for embedded systems. The program has loaded three components directly into its application space: 1) a 2D graphics library, 2) the OpenGL ES 3D library, and 3) a graphics driver that controls the graphics hardware. Loading the driver in this way allows the program to control the graphics chip directly and thereby increase performance. The program on the right is a Flash player. Like the OpenGL ES program, it also directly controls the graphics hardware, ensuring high performance.

Many graphics chips for automotive systems now support multi-layering. Developers can take advantage of this feature to seamlessly blend Flash and non-Flash programs on the same display. In Figure 1, the Flash player draws on a foreground layer and controls the drawing of the 3D map on a background layer. To make the 3D map visible, the developer used a chroma key technique on the

foreground layer. Since the 3D rendering and the Flash rendering take place on independent layers, the graphics controller can refresh the moving 3D map without having to redraw the Flash content – this eliminates flicker and reduces the load on the CPU.

Ensuring predictable response times

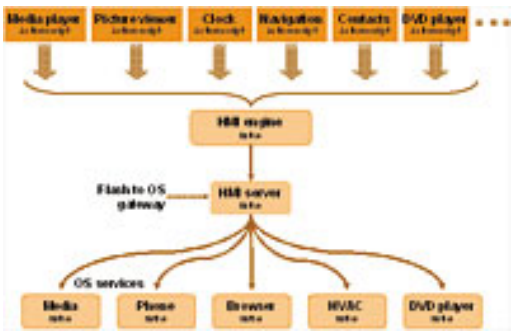
An automotive GUI should always respond promptly to user commands, even when the system is running a variety of CPU-intensive tasks, including multiple graphics programs. One solution is to create a central display manager that uses thread priorities to determine when each graphics program gets control of the CPU.

In this approach, a program (for instance, a DVD player) that wants to join the graphics environment sends a request to the display manager. The manager responds with a yes or a no, depending on whether the program has sufficient permissions to join. Upon joining, the program gains access to a mutual exclusion lock (mutex). When the program wants to draw something to the screen, it will wait on the mutex, acquire it, draw directly to the graphics chip, and then release the mutex. Every graphical program competes for this mutex based on its individual priority. Because the highest-priority graphics program will always acquire the mutex first, this approach ensures a level of realtime performance and a consistently fast user experience.

Managing failure conditions

To prevent system downtime, many embedded systems require some level of dynamic fault recovery. Using fault-notification mechanisms provided by the underlying operating system, the display manager described above can learn about graphical applications that fail. If a Flash-based or other graphical program fails while holding the mutex, the display manager can release the mutex and give it to the next program in the priority queue. The manager can also recover any resources that the failed program was using and restart the program.

Interacting with Flash



[2]

To integrate Flash successfully, embedded developers need to manage two types of interactions:

- Using Flash content to launch and control other Flash content
- Enabling communication between Flash content and OS services

Figure 2 provides an example of how to implement these interactions. An HMI

engine allows master Flash applications to manage secondary Flash applications (media player, picture viewer, etc.), while an HMI server provides a gateway to native OS services.

Consider how this design can work in a DVD player. If you look at Figure 3, you'll see a user interface that consists of:

- a viewing area that displays the movie (top half of screen)
- a DVD player Flash application that provides controls for play, pause, next, previous, etc. (middle of screen)
- an HMI engine, written in Flash, that controls the main menu buttons (bottom of screen)

In this example, the HMI engine launches and controls the DVD Flash player (Flash controlling Flash). The DVD Flash player then communicates with the DVD playback OS service through the HMI server (Flash content communicating asynchronously with native OS services).

The above examples show how real time and reliability can be preserved in an embedded system while allowing graphic designers to leverage the power of Adobe Flash. Modular HMI frameworks then become possible that provide a blending of 2D/3D applications, Flash applications, and multimedia.

Links:
[1] http://www.ecnmag.com/uploadedImages/Ecn/Articles/ec82if101a_large.jpg
[2] http://www.ecnmag.com/uploadedImages/Ecn/Articles/ec82if101b_large.jpg