3D Graphics: Adding a New Dimension to Your Embedded System

Dave Donohoe, Paul Leroux, Randy Martin
QNX Software Systems
randy@qnx.com
An emerging standard

Using 3D technology, embedded developers can create user interfaces that are more engaging, and, in many cases, more useful than conventional 2D or text-based displays. An in-car navigation system, for instance, can use 3D to present an intuitive “bird’s eye” view of the road ahead or to display an accurate rendering of local buildings and landmarks. In fact, several automakers and major auto suppliers have already introduced car infotainment systems based on 3D technology.

The adoption of 3D interfaces among designers of in-car systems and other embedded devices may seem surprising, at first. After all, 3D has traditionally required far too much memory and processing power to run on low-cost computing platforms. The situation is changing, however, thanks to two recent developments: the availability of embedded chip-sets that offer 3D acceleration and the emergence of a lightweight graphics API called OpenGL ES.

Low-risk 3D

First released in 2003, OpenGL ES provides an API for implementing 3D graphics in resource-constrained embedded systems. Despite its small footprint, the API supports advanced features such as alpha blending, Gouraud shading, and texture mapping, along with modeling, transforms, lighting, and numerous other techniques.

Features aside, OpenGL ES offers embedded developers two key benefits: reduced risk and higher return on investment. To begin with, OpenGL ES provides a well-defined subset of OpenGL, the most widely used 3D graphics API in the computer industry. As a result, development teams can tap into a large pool of OpenGL programming expertise and source code, not to mention a wealth of documentation, both online and in print.

As a vendor-neutral, multi-platform API, OpenGL ES also allows developers to reuse 3D code in new projects or across an entire product family. An OpenGL ES application can, without code modifications, run on multiple graphics chips and operating systems; it can also migrate from a low-cost system that uses software rendering to a more expensive system that uses a 3D acceleration chip to improve frame rate or resolution. Better yet, code developed in OpenGL ES can subsequently migrate to a high-end system that uses full OpenGL. Again, no code changes are required.

Last, but not least, OpenGL ES has gathered a phenomenal amount of industry support. Companies that either provide OpenGL ES-compliant products or belong to the Khronos group — the consortium responsible for OpenGL ES — include ARM, ATI, Intel, Motorola, Nokia, Nvidia, QNX Software Systems, PalmSource, Renesas, Sun, and Texas Instruments. (QNX is a contributing member of the Khronos group, with full API working group participation and voting rights.)
High-end graphics, low-end hardware?

Even with the advent of OpenGL ES, embedded engineers who implement 3D face a number of design challenges. To begin with, most embedded systems use an inexpensive, lower-end CPU and must consume as little RAM and flash memory as possible.

OpenGL ES helps to address these challenges in several ways. First, it uses a carefully selected subset of OpenGL, allowing a typical implementation of the library to consume only a few hundred kilobytes of memory. Second, it can work with a variety of low-cost graphics controllers that offer 3D acceleration, thus reducing the need for a high-performance CPU. And third, OpenGL ES supports profiles for both floating-point and fixed-point systems, allowing system designers to choose CPUs that, to reduce system cost, omit a floating point processor. (The fixed-point profile may provide less precision than the floating point version, but the difference is often negligible or unnoticeable on smaller displays.)

![Figure 1 — A navigation system that employs a combination 3D/2D interface. Courtesy of Ssangyong Motors.](image)

Techniques for minimizing memory footprint

Often, 3D techniques that work perfectly well on the desktop are too memory-intensive for an embedded device. Take, for example, depth buffering. Using this technique, you can ensure that objects which appear closer to the user aren’t obscured by more distant objects drawn later. The problem is, the depth buffer can consume as much memory as the display frame buffer. If this becomes an issue, try changing the order in which objects are rendered. By doing so, you may eliminate the need for a depth buffer entirely.
Heavy use of textures can also consume excessive memory, so use them only when necessary. In many cases, you can achieve the desired effect by using advanced lighting and shading techniques instead.

If your system does need textures, consider using the low-footprint texture formats supported by OpenGL ES, particularly when a texture doesn’t have to be highly detailed. These formats include both 4-bit and 8-bit paletted textures, allowing as few as 4 bits of data per texel, with as little as 32 bytes of per-texture overhead for the palette. Before choosing a texture format, however, determine how your system will handle it. For instance, if your hardware doesn’t support the texture directly, your OpenGL implementation will first need to convert the texture data to another supported format. By knowing this beforehand, you can make the best trade-off between memory usage and image quality.

Techniques for maximizing graphics performance

Graphics chips vendors are already developing embedded solutions that will provide acceleration for most or all of the OpenGL ES feature set. Nonetheless, you should take the capabilities of your current hardware into account when designing 3D applications. By doing so, you can determine which ES features to avoid (for now) and ensure that all rendering is performed in your 3D hardware.

Consider, for instance, tri-linear filtering, also known as mip-mapping. Using this technique, you can reduce aliasing artifacts in texture maps and thereby improve the appearance of the final image. However, if your 3D hardware doesn’t support tri-linear filtering, the frame rate could drop by at least an order of magnitude. If that’s the case, you can try using bi-linear filtering instead.

Another example is multi-texturing, which lets you apply multiple texture maps to a rendering operation, all in a single pass. If your hardware doesn’t support multi-texturing, you could avoid software rendering by performing the operation in two passes, using blending to combine the two textured images.

With or without windows?

A 3D application based on OpenGL ES can run without a windowing system, thus reducing memory usage to a bare minimum. On some devices, however, 3D applications must share the display with web browsers, radio controls, multimedia players, HVAC (heating, ventilating, air conditioning) displays, Java applications, and so on. If so, the device will likely need a true windowing system that allows multiple 3D and 2D applications to coexist seamlessly on the same screen.

This requirement raises a couple of issues. First, the device manufacturer may wish to build a product family, with each device offering a different set of features and a different price point. Consequently, the same 3D code that runs in direct mode on one device may need to run in
windowed mode on another, more feature-rich device. It's important, therefore, that system designers choose an implementation of OpenGL ES (or OpenGL) that supports both approaches.

The other problem is memory — or, to be precise, the lack of it. Unfortunately, conventional windowing systems, with their multiple megabytes of code, far exceed the memory budget of most embedded systems.

One solution is to use a windowing system based on a graphical microkernel. Like a microkernel OS, a microkernel windowing system starts with a tiny process (the microkernel) that implements only a few fundamental primitives. All other GUI services — graphics drivers, input drivers, window managers, and so on — are provided by optional, user-space processes that can be dynamically loaded or unloaded as required.

This architecture yields several benefits. First, it is easy to omit any GUI services not required by a memory-constrained device. The windowing system can, in effect, grow or shrink according to system requirements. Second, you can dynamically replace or upgrade almost any component of the windowing system individually, without rebooting the entire system. Third, if a graphics driver or any other component fails, the component can be automatically restarted, without a system reset and without involving the user. The QNX® Photon® microGUI® windowing system offers a good example of this architecture.

![Diagram of a microkernel windowing system]

**Figure 2** — By providing a highly modular architecture, a microkernel windowing system can reduce the memory required to run both 3D and 2D applications.
Techniques for combining 2D and 3D

Even if a modular, small-footprint windowing system is available, developers must still determine how their 3D applications will share the screen with conventional 2D programs and components, such as browsers and radio controls. A couple of options are available. The first is to use the multi-layering features now found in several graphics chips. For instance, you could use a background layer to display a moving 3D map, while using a foreground layer to display 2D touchscreen buttons. Since the 3D rendering and 2D rendering take place on independent layers, the display controller can refresh the 3D map without having to redraw the buttons — and without placing a burden on the host CPU.

You can also use techniques such as alpha blending and chroma-keying to make the buttons semi-transparent. That way, the buttons can appear directly over the map, rather than next to it. Using these same techniques — which allow more information to fit onto a small screen — you can also create semi-transparent menus, dialog windows, and other components.

All too often, however, programmers must implement multi-layer interfaces “by hand,” at a low level and in a hardware-specific way. This defeats a key advantage of OpenGL ES: hardware portability. To address the problem, a few operating systems, such as the QNX Neutrino® RTOS, support a high-level API that lets you create multi-layered displays in a hardware-independent fashion. For instance, applications written in the QNX API can, without recoding, run on graphics chips that support a different number of layers. Such an API should also let you choose whether each application manages its own layer or whether a central program manages how multiple client applications share the layers dynamically.

The second way to mix 3D and 2D applications is by synchronizing your 2D and 3D engines. While OpenGL ES is primarily a 3D API, it provides a mechanism that lets an application
switch between OpenGL rendering and the 2D native rendering provided by the windowing system, such as the QNX Photon microGUI. So, even if layering isn’t available, you can use platform-specific features to perform 2D rendering, as well as to combine 2D and 3D rendered primitives in the same image. In any case, it’s advantageous to use a software platform that employs the graphic card’s 3D engine to provide hardware acceleration for both 3D and 2D applications.

**Managing failure conditions**

Sophisticated systems that involve 3D and 2D graphics can, by the very nature of their complexity, experience occasional error conditions. The ultimate failure of a graphical interface is, of course, a system freeze and hardware-watchdog reset — much to the detriment of the user experience.

Other failure conditions are just as damaging to the user experience, even if they don’t lock up the system. For example, under heavy CPU load conditions, graphics performance could degrade to the point where user frustration results. This effectively constitutes a system failure.

If you’re building a closed system where usage cases can be controlled, you may be able to prevent these performance failures. But if you’re building a connected device that can dynamically support new content or applications, such prevention becomes much more difficult to achieve. It becomes critical, therefore, to partition the graphics environment so that its behavior under heavy CPU load remains consistent. This typically requires an operating system that allows you to place graphics programs and other software processes into secure partitions, where each partition is allocated a guaranteed share of CPU time and other resources. These resource guarantees can ensure responsive graphics performance, even when the system’s CPU reaches 100% utilization.

A graphical program may also terminate unexpectedly due to error. Consequently, system designers need a framework that can catch such failures and recover the graphical environment, without the need for a system reset. Such a framework also becomes important for devices that may download and run software from non-trusted sources.

**Ease of access**

By taking advantage of OpenGL ES, development teams can implement high-performance, memory-efficient 3D interfaces that migrate easily to new hardware. Better yet, OpenGL ES is based on the widely supported OpenGL standard, allowing developers to access a wealth of existing documentation and programming expertise.

For more information on these and other benefits of OpenGL ES, visit the Khronos website at http://www.khronos.org. For more information on QNX advanced graphics, including support for OpenGL ES, visit the QNX web site at http://www.qnx.com.
About QNX Software Systems

QNX Software Systems is the innovation leader in embedded technology with middleware, development tools, operating-system software and services. Global leaders such as Cisco, Daimler, General Electric, Lockheed Martin, and Logitech depend on QNX technology for a wide range of OEM applications. QNX Software Systems is headquartered in Ottawa, Canada, and distributes products in over 100 countries worldwide.

www.qnx.com