WHITE PAPER

Texas Instruments

Introduction to Graphics Software Development for OMAP™ 2/3

By
Clay D. Montgomery,
Graphics Software Engineer
Media & Applications Processor Platforms
Texas Instruments

Executive Summary

The latest OMAP devices offer breakthrough graphics capabilities for a wide range of applications including personal navigation, scalable user interfaces and gaming. This is made possible by advanced graphics acceleration hardware and support through industry standard APIs like OpenGL® ES, OpenVG™ and Direct3D® Mobile. But which of these is the best solution for your application and how do you get started on software development? This paper attempts to answer these questions without requiring much experience with graphics concepts or terminology.

Introduction

The Texas Instruments OMAP 2 and 3 device families feature powerful graphics acceleration hardware that rival the capabilities of desktop computers of just a few years ago. The field of software tools, industry standards and API libraries for exploiting these new devices is complex and evolving so rapidly that it can be difficult to know where to start. The purpose of this report is to give enough of an introduction to the field of 2D and 3D graphics that informed decisions can be made about which tools and standards best fit the goals of any particular application development project. This is only a starting point and only a minimal understanding of computer graphics concepts is assumed. This report also provides many pointers to more definitive information sources on the various topics that are introduced, for further research. For all topics, throughout this report, be sure to also utilize Wikipedia (www.wikipedia.org).

Computer graphics can be as simple as a library of functions that draw geometric shapes, such as lines, rectangles or polygons on a 2-dimensional plane, or copy pixels from one plane to another. Such planes are called bitmaps, drawing surfaces or canvases. They may represent the pixels on a visible display device (such as LCD), or they may be stored somewhere off screen, in an unseen memory area. Bitmaps have a bit depth that determines how many colors they can represent and their pixels may be defined in either the RGB or YUV color space.

The process of drawing shapes on a bitmap is called rasterization or rendering. This can be done directly by software running on the host processor (ARM®), digital signal processor (DSP) or by graphics acceleration hardware. Generally, it's best to use an accelerator whenever possible because it will require fewer clock cycles and less power to achieve the same rendering.

In 2D graphics, copying pixels from one bitmap to another is a common operation called a BitBlt (Bit Block Transfer), which can be implemented in hardware or software. A typical 2D graphics accelerator is really a BitBlt accelerator, implemented in hardware. It is similar to DMA, but specialized to transferring data on pixel (rather than word) boundaries. BitBlt operations are the foundation of 2D graphics and this is how most graphical user interfaces (GUIs) have traditionally been built.

Simple 2D Graphics

Many applications are built entirely on relatively simple BitBlt operations. Some examples are the graphical user interfaces of Microsoft® Windows®, Linux, MacOS®, set-top boxes and mobile device platforms like Symbian OS™. Each time a user opens or drags a window in a GUI, hundreds of BitBlt operations are instantiated, so the performance of the BitBlts is important to achieve a responsive user interface. The math that is used to calculate BitBlt operations is typically all integer (no floating point), so this helps to achieve good performance.

The main limitation of BitBlt-based graphics is that they do not scale well. User interfaces are typically built for a specific display size (i.e., VGA; 640 × 480 pixels) and it's difficult to make the graphics adaptable to different screen sizes. A good example of this is what happens to the Windows desktop when it's changed from a high-resolution mode to VGA mode. The icons become huge and pixilated and most of them no longer fit on the display.

OMAP 2 devices feature a dedicated BitBlt accelerator which applications can access through the PVR2D API (Application Programming Interface). Under Windows Embedded® CE and Windows Mobile®, their DirectX® drivers will utilize the 2D accelerator directly.

OMAP 3 devices approach 2D acceleration differently. Instead of using BitBlt operations, OMAP 3 devices avoid using BitBlts entirely in favor of scalable 2D graphics programmed through OpenVG™ or a 3D interface, such as OpenGL® ES or DirectX Mobile (for Windows).This approach is described in more detail as "2.5D" in the next section.

2D, 2.5D or 3D?

For those uninitiated to the graphics world, it can be difficult to judge whether a particular application is really implemented with 3D graphics, or if it is just making clever use of 2D, to look like 3D. For example, icons which appear to spin in 3D are often achieved on a 2D GUI by repeatedly copying a sequence of still images (BitBlts). Many games feature 3D-looking characters and objects that really only move in two dimensions (around the screen). This is easy to achieve by applying shading effects to the objects once, then moving them about with BitBlts operations. This typically requires that the "light source" in the scene is at a fixed position; usually somewhere off of the screen. This works well for many applications.

The main criteria that necessitates 3D graphics capabilities is if the user will navigate into the scene at substantial depth, and/or the user's view will rotate into the depth dimension (for example, the user can "turn around" and look at what's behind him). Further, if these depth movements must be performed interactively and in real-time in order to achieve an immersive feel of "being in the scene", this requires the shading of all the objects in the scene be rendered repeatedly to account for each change in the relative position of the user's view and light source(s). These requirements, taken together, demand 3D graphics capabilities.

Another criterion is the need for a transformation matrix to convert the coordinate system in which the scene is built into the coordinate system of the user's graphics display. For example, a VGA display is 640 × 480 pixels, but immersive 3D environments require a much larger range of coordinates than VGA allows. Typically, this requires coordinates with such a large dynamic range that they are best specified with either single- or double-precision floating-point. However, there are a few notable examples of this being done entirely with fixed-point precision (such as the first Sony PlayStation®).

Finally, 3D graphics accelerators are designed to copy bitmaps onto surfaces with any arbitrary orientation (in three dimensions) with respect to the camera (the display) and to transform the pixels so that their perspective is correct on the new shape. This is called texture mapping and it requires interpolation and filtering that is well beyond the BitBlt capabilities of 2D. The example below contrasts a typical BitBlt operation, which simply copies some portion of the source bitmap, with a texture mapping operation, which maps the bitmap to a different plane in 3D space which requires the pixels to be interpolated to achieve the correct perspective in the scene.

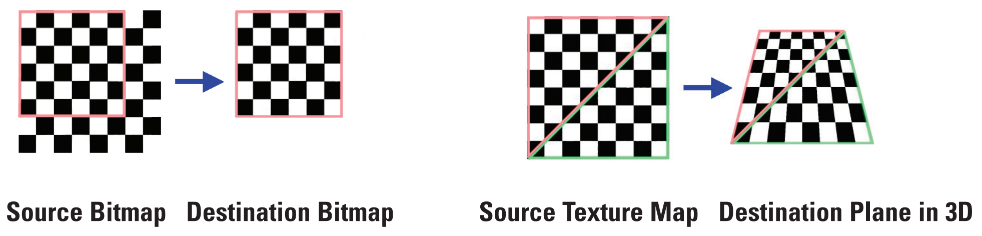


Figure 1. Comparison of a typical 2D BitBlt operation verses a 3D texture mapping operation.

Of course, 3D accelerators add many more capabilities as well, like the ability to handle changes in light sources and removing surfaces which are obscured by other surfaces in the scene (culling).

Another option, referred to as "2.5D" combines the coordinate transformation features from 3D with rendering in only two dimensions. This gives the ability to scale and rotate

2D graphics easily, but it changes without the ability to change the viewer's perspective. See the example below.

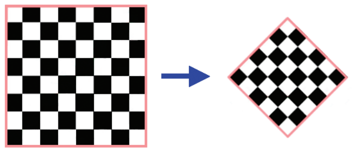


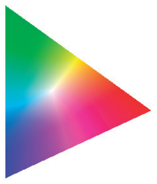
Figure 2. An example of a scalable 2.5D operation.

Adobe® Flash®, Scalable Vector Graphics (SVG) and OpenVG™ are examples of graphics interfaces which are designed specifically for scalable 2D. OpenGL can also be used (and often is) as a 2.5D graphics interface. This is simple to achieve by rendering all of the objects in a scene at a constant depth and with the user's point of view, and light sources, fixed. This approach works well on the OMAP™ 2 and 3 platforms since the rendering and coordinate transformations can be handled by the 3D accelerator.

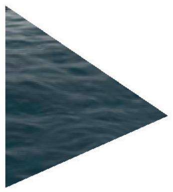
Vertex Shading Verses Pixel Shading

Dynamically shading objects based on light sources and camera position (which may be moving) is a key feature of any 3D graphics interface. There are two approaches to this which are dramatically different; vertex shading and pixel shading. It's important to understand this distinction because it affects the choice of a graphics interface.

Vertex shading is the older approach – supported since the original version of OpenGL and DirectX. However, don't assume that vertex shading will soon be obsolete; it is still widely used and supported because it's simpler to use and still well suited for many applications. In vertex shading, the shading calculations are only performed at the vertices of each triangle in the scene. For example, a triangle has three vertices, so only the colors of the pixels at these three points in space are calculated. In the example below, the colors at the vertices are red, green and blue. The remaining pixels that are visible on the triangle are then determined by smoothly interpolating between the three vertices.



Vertex Shaded with Interpolated Colors



Pixel Shaded with Waves on Water Effect

Figure 3. Examples of vertex-shaded and pixel-shaded triangles.

The main limitation of vertex shading is that objects which are rendered with too few vertices tend to have a blocky and unnatural appearance (see example below). Increasing the quality of rendering typically requires increasing the number of vertices (tessellation), which increases the memory required to store that data and will also reduce the rendering performance.

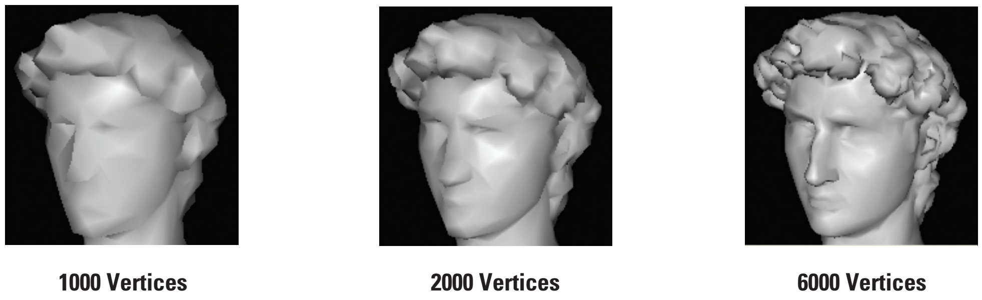


Figure 4. Example of a vertex-shaded 3D model rendered with increasing tessellation.

Pixel shading (also known as fragment shading in OpenGL®) is relatively new and is used primarily in applications where photorealism is the driving goal. Pixel shading helps achieve this, without the need to increase the tessellation (quantity of vertices). In pixel shading, the color of every pixel in the object, which is visible to the viewer, is calculated individually. The pixel colors are calculated by a shader program, which can be customized and downloaded into the SGX hardware graphics accelerator on OMAP™ 3 to achieve unique special effects. These effects can be regular or random patterns of pixels used to simulate the appearance of rust on steel, ripples on water, or pebbles in concrete, for example. More advanced examples of custom shader programs include bump mapping, normal mapping and shadow and reflection mapping.

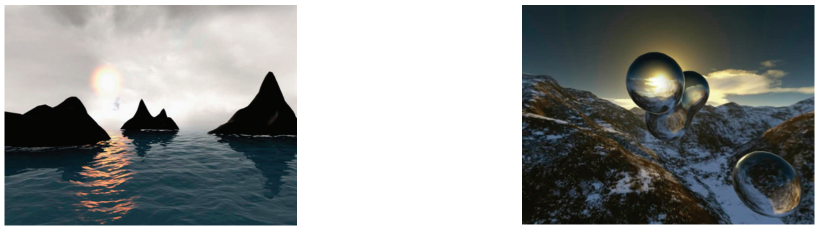


Figure 5. Examples of pixel shading on OMAP 3 (SGX).

Downloadable videos of several vertex- and pixel-shading demo programs for OMAP 2 and 3 are available at:

www.ti.com/omapgaming

Similar effects can be achieved with vertex shading by applying texture maps, but texture maps are usually static images and storing a large number of them will increase storage requirements. Therefore, pixel shading offers tremendous new flexibility to customize photorealistic 3D scenes, without dramatically increasing storage requirements.

OpenGL ES 1.1 only supports a fixed-function 3D pipeline for doing vertex shading, while OpenGL ES 2.0 supports a programmable 3D pipeline where pixel-shading programs can be used. It is still possible to do vertex shading with OpenGL ES 2.0, but it must be emulated by a shading program. This is different than the desktop version of OpenGL 2.0, which supports both vertex and pixel shading, directly. Pixel shading is not available in Direct3D® Mobile for Windows® Embedded® CE or Windows Mobile®. So, programmable pixel shading is only available on the OMAP™ 3 platform with OpenGL® ES 2.0 support.

Pixel shaders are typically implemented with a dedicated pixel shading engine which may have their own instruction set, programming language and compiler tools. Programming the SGX shader requires the OpenGL ES 2.0 Shading Language (GLSL ES). Several tools are available for GLSL development for either the Windows Embedded CE or Linux platforms.

- PowerVR Shaman Shader
- ATI/AMD RenderMonkey
- SourceForge Lumina
- Blender

Note that there are some compatibility issues between GLSL and the embedded version, GLSL ES. The PVRShaman Shader and RenderMonkey provide the best support for GLSL ES development for the SGX.

Figure 6. GLSL and GLSL ES development tools

Fixed Point Verses Floating Point

Using floating-point precision for the calculation and storage of coordinates greatly increases the numerical precision and dynamic range of this data as compared to using fixed-point. All of the OMAP™ 2 and 3 devices feature either the ARM® VFP (Vector Floating Point) or VFP Lite units. Both of these support IEEE 754 compliant single- and double-precision floating-point acceleration. VFP Lite is fully functional and compatible with VFP, but requires more clock cycles to perform floating-point operations. The NEON™ feature of OMAP 3 can be used to improve the speed of single-precision floating-point (but not double-precision) calculations. The chart below summarizes these important architectural features for OMAP 2 and 3 graphics.

Summary of OMAP 2 and OMAP 3 architectural features for graphics

Part Numbers	CPU	Float Support	3D Engine
OMAP 2530	ARM1176 @ 330 MHz	VFP	MBX Lite @ 83 MHz
OMAP 3503/3525	ARM Cortex-A8 @ 660 MHz	VFP Lite with NEON	None
OMAP 3515/3530	ARM Cortex-A8 @ 660 MHz	VFP Lite with NEON	SGX @ 110 MHz

Note that not all OMAP 2 or OMAP 3 parts contain all features. This is just a representation of the superset of available features.

Graphics Interface Standards

OpenGL®

This is the original version of the OpenGL standard interface for 3D graphics on desktop systems. Microsoft also offers a similar standard, Direct3D® (D3D) interface. Both are commonly used on Windows® platforms, but Direct3D® is available only on Windows, where it's dominate. OpenGL is the dominant standard on all other operating system platforms.

OpenGL is known for its architectural stability. There have been several minor revisions, as the standard has evolved, but only one major revision, 2.0. Prior to the 2.0 version, OpenGL supported only vertex shading. Version 2.0 added pixel-shading capabilities. OpenGL has always required floating-point support, which is usually available on desktop systems.

OpenGL is essentially a library of functions which are designed so that they can be ported to any graphics-acceleration hardware without changing the application code that uses these functions. This library of functions is contained within a device driver known as the Installable Client Driver. Hence, OpenGL implementations are often referred to as "drivers" or "libraries".

One of the great strengths of OpenGL is the abundance of quality resources for learning how to use it, including books, tutorials, online coding examples and classes. In particular, there are two essential books that every developer should have as a starting point – often referred to as the Red and Blue books. These are the official OpenGL Reference Manual and Programming Guide:

- The OpenGL Reference Manual
- The OpenGL Programming Guide

The complete specifications for OpenGL are also available publicly at:
www.opengl.org/

These are great resources for learning OpenGL and for reference during development, but keep in mind that they cover the desktop version which is quite large compared to the embedded OpenGL ES versions supported on the OMAP 2 and 3 platforms.

OpenGL® ES 1.1

This is the fixed-function version of the OpenGL ES standard for doing vertex shading on embedded devices. This is the most widely used 3D graphics interface for embedded and mobile devices today. It is a subset of the OpenGL standard used widely on desktop systems. OpenGL ES also supports 2D graphics features for antialiased text, geometry and raster image processing. This embedded version is royalty-free and the complete specifications are publicly available at:

www.khronos.org/opengles/

There are two profiles of OpenGL® ES 1.1, Common and Common Lite. The Common Lite profile is for devices with no floating-point capability. Since all OMAP 2 and 3 devices have floating-point capabilities in hardware, only the Common profile is supported. Supported OSes include embedded Linux, Windows® Embedded® CE and Symbian OS™. OpenGL ES is the most widely used 3D graphics interface on these platforms. Porting to other operating system platforms is available through TI approved graphics partners.

Although this version of OpenGL ES is designed primarily for vertex shading, there is also support for a type of DOT3 per-pixel lighting known as bump mapping. Refer to the ChameleonMan example program in the software development kit (SDK) for an example of this.

OpenGL ES 1.1 was preceded by version 1.0. The new functionality includes better support for multitexturing, automatic mipmap generation, vertex buffer objects, state queries, user clip planes and greater control over point rendering.

OpenGL® ES 2.0

This is the latest version of the OpenGL ES standard. It is designed specifically for 3D accelerators with a programmable pixel-shading engine, such as the OMAP 3 SGX. It does not support vertex shading directly, but that can be emulated with a supplied shading program. Otherwise, it shares most of the features of the 1.1 version, including being open and royalty-free. The common profile of OpenGL ES 2.0 is fully supported on the OMAP™ 3 platform under embedded Linux, Windows Embedded CE and Symbian OS. Porting to other operating system platforms is available through TI-approved graphics partners. This is the most capable 3D graphics interface on this platform. This version is not available on the OMAP 2 platform because it requires pixel-shading capabilities.

The OpenGL ES standards were created by the Khronos Group, a member-funded industry consortium formed to create and promote open standards for media acceleration and authoring for embedded devices. Since 2006, Khronos has also been home to the committee which controls OpenGL, the leading open standard for 3D on the desktop. The list of current Khronos member companies is available at:

www.khronos.org/members/contributors

Texas Instruments has been an active participant in the Khronos OpenGL ES working group since its first official meeting in 2002, and a TI employee currently serves as the chairman of this working group. TI plays an active role in several other Khronos working groups as well. We believe that this is the best way to create a healthy, dynamic content market and to assure alignment of OMAP graphics implementations and these standards.

GLU and GLUT

These are popular utility libraries for desktop versions of OpenGL. They are not part of the official OpenGL standard, but they are often used in applications on desktop environments. GLU and GLUT are not directly supported in the embedded versions of OpenGL ES, therefore, this is an important issue when an OpenGL application is ported from the desktop to OpenGL ES. Any calls to GLU and GLUT functions must be converted to use new interfaces.

GLU consists mainly of convenience functions, which may require the allocation of large working buffers, but always work through OpenGL, rather than accessing hardware directly. Some of these capabilities are already integrated into OpenGL ES (such as the generation of texture mipmaps) and others are not (such as the tessellation of complex polygons).

GLUT is an abstraction layer for the host windowing system. It standardizes and simplifies the management of windows and events from input devices like the keyboard and mouse. This allows OpenGL applications to be easily ported between Windows® and Linux platforms. For embedded applications based on OpenGL ES, the window management services of GLUT are replaced by a new interface named EGL. But, EGL does not support input devices, so these services must be provided through another interface.

EGL

This is the native platform interface for OpenGL ES and OpenVG – both are supported by this single unified interface. The native windowing system is usually either X11 (for Linux) or Windows GDI. Whichever is used, EGL provides the mechanism for creating drawing surfaces (bitmaps) onto which OpenGL ES and/or OpenVG can render. EGL also handles graphics context management, surface/buffer binding and rendering synchronization. This unified interface provides the ability to mix 2D and 3D rendering as well as independence from the Linux and Windows platforms. EGL drivers are provided in the SDKs for the OMAP 2 and 3 platforms. The complete specifications for EGL are publicly available at:

www.khronos.org/egl/

OpenVG™

This is a relatively new graphics interface standard created specifically to support 2.5D graphics applications on mobile and embedded device platforms. 2.5D combines the coordinate transformation features from 3D with rendering in just two dimensions. Coordinate transformations are limited to affine transforms, which are smaller matrices that support scaling, translation and rotation, but only in two dimensions. This is well suited for applications that need to animate, scale and rotate 2D graphics, such as advertising messages in web browsers or maps for navigation.

OpenVG™ is not a subset of OpenGL. It takes a very different approach. OpenVG does not have features like triangle strips, texture mapping, lighting, pixel or vertex shading, which are the foundations of OpenGL. Instead, OpenVG has a more extensive set of features specifically for drawing 2D shapes. For example, OpenVG allows polygons of arbitrary complexity, line widths, line join styles, and polygon fill styles (see examples of linear and radial gradient fill styles below). OpenVG also makes it easier to animate, rotate and scale 2D graphics without concern about pixel aliasing effects. This is because output images are filtered to prevent aliasing.

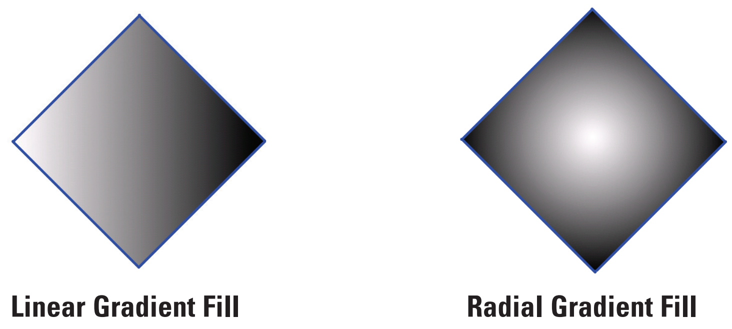


Figure 7. Examples of some OpenVG polygon fill styles.

The capabilities of OpenVG closely follow those of SVG (Scalable Vector Graphics), which is a slightly older standard created to be an open alternative to Adobe® Flash®. On the OMAP 3 platform, the same OpenVG driver can be used to support the acceleration of Adobe Flash.

OpenGL ES can also be used (and often is) as a 2.5D graphics interface. This is simple to achieve by rendering all of the objects in a scene at a constant depth and with the user's point of view, and light sources, fixed. So, which is better for 2.5D applications, OpenVG or OpenGL ES? The short answer is that OpenVG makes it easier to develop scalable 2D graphics content, but the design of OpenGL ES is better matched to the capabilities of the graphics hardware and therefore offers better performance.

The more extensive 2D drawing capabilities in OpenVG make it easier for software developers (and artists) to create 2D graphics content. However, many of these capabilities (complex polygons and gradient fills) are not well matched to the design of 3D accelerators, so they must be emulated by software which can lead to performance problems. Such performance problems can largely be avoided by limiting the use of OpenVG features which are known to be problematic, when the graphics content is created.

Of course, this assumes that the source of the content is controlled. If the application is a browser that can pull any content from the web, it may be impossible to limit the OpenVG feature set that is required to display the content.

OpenVG does produce higher quality images than OpenGL ES. The higher quality output is due to OpenVG's powerful antialiasing filters which are designed to handle a very wide range of scaling and rotation. Images can be scaled from the size of less than one pixel to full screen (or more) without aliasing or tessellation artifacts that can occur in OpenGL ES.

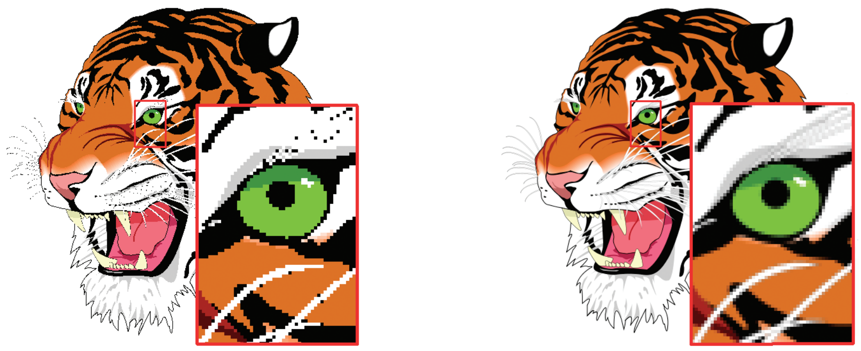


Figure 8. Sample OpenVG images without and with antialiasing filter.

The PowerVR SDK provides a set of documentation, source code and utilities that help developers get started creating applications using the OpenVG graphics library on OMAP 2/3 platforms.

Adobe® Flash®, Flash Lite and SVG

Adobe Flash is a popular proprietary standard for delivering animated 2D graphics, video, audio and interactivity for web pages. All of the major desktop browsers support Adobe Flash and it has become popular on browsers for mobile devices as well. Virtually any website with animated 2D advertisements or video is implemented with Flash today. Adobe develops and licenses both the tools for authoring this content and the web browser plug-ins to play the content.

Flash supports two major file types, .swf and .flv. The first is for 2D graphics and the later is for compressed video. The graphics format is another example of a 2.5D interface which combines the coordinate transformation features from 3D with rendering in just two dimensions. Coordinate transformations are limited to affine transforms, which are smaller matrices that support scaling, translation and rotation, but only in two dimensions. This is well suited for applications that need to animate, scale and rotate 2D graphics, such as advertising messages in web browsers or maps for navigation.

To implement support for Flash® on a new device requires licensing the Flash player software from Adobe®.

www.adobe.com/licensing/

On the OMAP™ 3 platform, Adobe Flash is supported under Linux by the accelerated OpenVG™ driver. Adobe offers an implementation of their Flash player specifically for the ARM® Cortex™-A8 (OMAP 3) platform which is optimized to utilize the Cortex and NEON™ accelerator. Please contact Adobe® for more information.

The Lite version of Flash is targeted specifically at embedded devices which do not have hardware floating-point capability, but this is not an issue on the OMAP 2 or 3 platforms.

Flash is typically a hard requirement for devices that allow users to surf the public Internet because so many websites use Flash today. YouTube is a good example. However, there is a non-proprietary, open alternative to Flash which is gaining traction called SVG (Scalable Vector Graphics). SVG offers the scalable 2D functionality of Flash, but SVG is an open standard.

www.w3.org/Graphics/SVG/

Graphics Performance Benchmarks

Care must be taken when comparing graphics performance numbers. So many variables affect the measurement of graphics speed that it's virtually impossible for two different organizations to construct comparable platforms for making such measurements. Graphics performance is really the end result of overall system performance, including CPU speed, cache sizes, memory bandwidth, floating-point performance, operating system and graphics interface overhead, and finally, the configuration of scene lighting, texturing, display size, etc.

It's common for data sheets to state performance in terms of raw triangles drawn per second, but these numbers are often just estimates based on available memory bandwidth and maximum clock rates. Therefore, raw triangles per second are really theoretical maximums and useless for comparison purposes. The performance in real applications will usually be lower.

FutureMark is a company that attempts to benchmark the actual, measured performance of major 3D graphics solutions with their 3DMarkMobile06 tool. Their results are well documented, publicly accessible and realistic. Their certified results are available at:

futuremark.com/bdp/certified/imagination/

FutureMark has not yet released certified benchmark results for OMAP 3.

PowerVR Insider Developer Forum

This is another important resource for developers creating graphics applications on the OMAP™ 2 and 3 platforms. Developers can find news, additional SDKs and content authoring tools compatible with the OMAP 2 and 3 platforms here:

www.imgtec.com/PowerVR/insider/

The PowerVR SDK

The first step to get started developing software applications for the OMAP 2 or 3 platform is to install the SDK (Software Development Kit). Distinctive SDKs are provided for each OMAP platform and embedded operating system combination. The contents of the SDKs are similar except that each contains device drivers which are specific to the target platform. Here is a summary of the typical contents:

Contents of the PowerVR Graphics SDK

OpenGL ES Driver	Compiled OpenGL ES driver for OMAP 2 (1.1) or OMAP 3 (1.1 and 2.0) for Linux or Windows®.
OpenVG Driver	Compiled OpenVG driver for OMAP 2 or OMAP 3 for Linux or Windows.
Builds	Compiled libraries and header files for linking into applications.
PVR Shell	A C++ class used to make programming for PowerVR platforms easier and more portable.
Documentation	Essential documentation including SDK Users Guide, Recommendations and Release Notes.
Demos	Source code of several complete programs that demonstrate how to program OpenGL ES and OpenVG.
Training Course	A tutorial for learning how to use OpenGL ES and OpenVG.
Tools	A library of some useful functions in source code form.
Utilities	A collection of essential tools to aid the development of graphics content.

The PowerVR Utilities

Several powerful utilities are provided in the PowerVR SDK to help developers create exciting 3D content. These are summarized in the following pages.

PVRTexTool and PVRTC Library

Applications which utilize texture mapping techniques often require a large number of images to use as texture maps, so typically they are compressed to conserve memory. Compressing the images is a computationally intensive process that is usually performed in advance, on a PC, when the graphics content is authored. The PVRTexTool is a utility that is provided to do this image compression. It will convert many popular image file formats (e.g., BMP, JPG, PNG, TGA, etc.) to any of the texture map formats supported by the MBX or SGX hardware, including PVRTC2, PVRTC4, DXT and ETC compressed formats. It can also perform other common image conversion tasks, such as adding borders, generating MIP maps, etc. Several versions of PVRTexTool are supplied, including GUI-based versions and command line versions for both Windows® and Linux.

The functionality of PVRTexTool is also available from within several popular game content authoring tools, such as 3D Studio Max® (versions 6 through 8), Maya® and Adobe® Photoshop®, once the supplied plug-ins are installed. A plug-in is a software module that adds new functionality to these software applications. This tighter integration of PVRTexTool and the authoring applications can help increase content developer productivity.

If other authoring tools are used, the PVRTC Library can be adapted to a custom environment. This is the same texture compression algorithm used by PVRTexTool, but in a form that can be easily integrated with custom authoring tools, under either Windows or Linux.

PVRMAXExport

This is the PowerVR plug-in for 3D Studio Max® (versions 6 through 8) and Autodesk® Maya® (versions 7 and 8) for Windows®. A plug-in is a software module that adds new functionality to an established application for graphics content authoring. This provides tight integration of these tools that can help increase developer productivity. The new functionality includes the texture compression features of the PVRTexTool, exporting of model data in formats optimized for the MBX or SGX plus some special features such as tangent space generation and bone batching.

PVRGeoPOD

This is another plug-in for 3D Studio Max (versions 6 through 9) and Autodesk Maya (versions 7 and 8) that exports geometry and animation models from these applications into the PowerVR POD file format. It also supports some other special features, including tangent space generation and bone batching. The POD file format is designed for optimal performance with the MBX and SGX and example code is provided in the SDK for loading this format into applications.

PVRVecEx

This is a plug-in for Adobe® Illustrator® (CS1 and CS2) which enables vector graphics artwork to be exported to the PowerVR Vector Graphics (PVG) file format. This is a binary format which is optimized for efficient rendering through OpenVG. It is typically half the size of equivalent SVG files (which are XML based) and can be loaded into OpenVG applications easily. Example code is provided in the SDK to load this format into applications.

Collada2POD

This utility converts geometry and animation models from the Collada™ format to the PowerVR POD file format. Collada is a Khronos standard file format for exchanging 3D graphics content between authoring tools of different vendors. It provides comprehensive encoding of visual scenes including geometry, texture maps, GLSL shaders and physics. However, since it is XML-based, it is not particularly compact. Collada2POD allows this standard format to be converted into the more compact POD format.

PVRShaman

This is an integrated development environment designed specifically for creating GLSL ES (OpenGL ES 2.0 Shading Language) vertex and pixel shading programs for the SGX. It features an integrated editor, compiler and emulator which shows the output from the shading program immediately after each compile. This makes it easy to debug and experiment with new shading programs.

PVRShaman accepts geometry from POD files (which are typically exported from applications like 3D Studio Max®) and applies the compiled shader program to that geometry. Collada files are also supported for importing models. Shader programs are stored in PowerVR FX (PFX) files, along with textures, so these files completely define the appearance of a material, or a group of materials. Example code is provided in the SDK to load this format into applications. Both Windows and Linux versions of PVRShaman are provided. This tool is not available for OMAP™ 2.

VFrame

This is also known as the PowerVR PC emulator. It is a set of libraries which emulate OpenGL® ES 1.1, 2.0 or Direct3D® Mobile function calls by mapping them to equivalent OpenGL or Direct3D functions on a desktop PC. The OpenGL ES version is available for both Windows® and Linux, whereas the Direct3D Mobile version requires Windows.

This allows graphics application software to be developed for OMAP 2/3 devices on a desktop PC, even without the actual OMAP 2/3 hardware. Such applications can then be easily ported to the target device, when ready. VFrame emulation requires a PC that is equipped with an OpenGL 2.0 and/or DirectX9 compliant graphics accelerator. Emulating OpenGL ES 2.0 requires a graphics card that supports Microsoft shader model 3.0 or better.

PVRTune

This is a remote performance profiling utility for the SGX accelerator on OMAP 3. It consists of two parts, an instrumented driver named SGXPerfServer, which runs on the target device and collects information on the SGX in real-time. This driver then transmits this information over a network connection to PVRTune, which runs on either a desktop Windows or Linux system.

Various statistics on geometry, texture and shader memory usage are collected and charted which can help identify bottlenecks in the SGX processing pipeline due to how it is being used by the application. This profiling information can be used to improve the application software so that it will utilize the SGX more efficiently by avoiding the identified bottleneck conditions, and thereby improve performance. PVRTune also allows some commands to be sent to the SGX drivers running on the target device. This tool is not available for OMAP™ 2.

The Demonstration Programs

The PowerVR SDK also contains several complete demonstration programs in source code form. These programs demonstrate how to use many important features of OpenGL® ES and how to achieve the best possible graphics performance on the OMAP 2 and 3 platforms. Some of these programs demonstrate advanced techniques which are not necessarily applicable to any given application, but others show basic techniques which are essential for applications developers to understand. This is a great place to start the development of any new graphics application for OMAP 2 or 3. These programs can also be compiled to run on a PC using the VFrame emulator, which requires no target OMAP device.

ChameleonMan

This demo shows a matrix skinned character in combination with DOT3 per-pixel-lighting. This is a specific form of pixel shading (bump mapping) that is implemented using OpenGL ES 1.1, even though it does not support programmable shaders. The user can select between the vertex-shaded mode and the DOT3 pixel-shaded mode to show the stunning visual improvement made possible by DOT3 bump mapping.

Matrix skinning is a technique where a vertex is animated over time given a set of matrices and blend weights assigned to those matrices. The ChameleonMan model has 19 bones and an animation cycle of 16 frames. For each frame, the matrix set is recomputed based on time. For example, to render the model at time point 5.25, the application linearly blends between the matrices stored for frame 5 and 6 using weights of 0.75 and 0.25, respectively. Up to three matrices per vertex from the matrix set, along with three weights are used by the vertex program to update the vertex position to obtain the current animation frame position.

To implement DOT3 per-pixel-lighting, this demo supplies a normal, binormal and tangent per vertex. These tangent space vectors are transformed by the vertex program into world space coordinates. The light vector is then transformed into this local skinned tangent space. The tangent space light vector is then used to do the DOT3 lighting calculation, which is finally combined with the base texture using multitexturing.

EvilSkull

This demo shows a morphing model of a human skull in combination with multitexturing. Morphing creates the final object using weighted blending between a number of different configurations of the same skull model. Each configuration controls a specific aspect of the skull. By setting different weights for the vertices that correspond to the eyebrows and cheekbones, a large variety of facial expressions is generated. The background effect is created by independently animating two multitextured layers.

FiveSpheres

This demo illustrates five different graphics primitive types of OpenGL® ES. These are named GL_POINTS, GL_TRIANGLES, GL_LINE_STRIP, GL_TRIANGLE_FAN and GL_TRIANGLE_STRIP.

Lighting

This demo shows a sphere which is lit by eight differently colored spotlights using the ambient, diffuse and specular lighting components of OpenGL ES.

Mouse

This demo shows a dancing mouse with a hard black outline and banded shading to achieve a cartoon appearance. This is cell shading, which is a rendering technique to make character models look like cartoons. It is also known as toon shading. These effects are created using optimized vertex shading.

OptimizeMesh

This demo shows an important technique for improving the performance of OpenGL ES. A texture-mapped model is rendered in two different modes; one using a non-optimized triangle list and another using an indexed triangle list that has been sorted. The demo switches mode after a few seconds or when the user presses the action key. Both models were generated using the PVRMAXExport plug-in utility in the SDK, but the optimized triangle list was generated with the PVRTTriStrip triangle sorting option enabled. Note that this performance difference may not be visible when run on the VFrame emulator.

Particles

This demo shows a physics-based particle effect displaying a total of 900 particles (600 real plus 300 reflected). Some of the particles also demonstrate alpha blending techniques.

PhantomMask

This demo shows a mask shaded using spherical harmonics lighting and regular diffuse vertex lighting, so that the performance of these techniques can be compared. Spherical harmonics lighting preprocesses the projection of the lights, the model and the transfer function, on a spherical harmonic basis, to render realistic scenes using any type of light source. It is primarily used to reproduce diffuse lighting effects.

The diffuse vertex lighting case requires at least four light sources to implement, which is roughly equivalent to the spherical harmonics case for this scene. But, implementing four diffuse lights requires more vertex program instructions than the spherical harmonics calculations. Realistically, many more vertex lights would have to be added to fully match the result of the spherical harmonics case.

PolyBump

This demo illustrates an implementation of bump-mapping techniques, which enhances rendering quality without increasing the tessellation of the models. A highly detailed human head is shown even though there are only 276 polygons in this model.

A DOT3 normal map is generated in advance using a highly tessellated model. Then, this normal map is applied to a low tessellation version of the same model. There is virtually no visible loss in rendering quality because the normal map preserves the detail from the original version of the model, but the rendering time is reduced due to the reduction in the geometry.

ShadowTechniques

This demo illustrates several different techniques to simulate shadows in a lighted scene. These techniques are called dynamic blob, projected geometry and projected texture.

The dynamic blob method draws a transparent blob under an object and the blob moves based on the center of the object casting the shadow. The projected geometry method draws the object projected onto the plane of the floor. Only black shadows are possible here because multiple polygons are being projected onto the same location. The final method is a projected texture where the projected shadow texture is updated dynamically each frame. This is generated by rendering the scene from the point of view of the light source. Unlike the previous methods, this approach also works correctly for shadows cast on non-planar objects.

Skybox

This demo shows a technique often used to construct backgrounds for 3D scenes in games. A skybox is a set of large texture maps that are positioned to surround the active area of a 3D game scene. These textures are designed to transition smoothly from the active area so that they appear to be a continuation of the active area. Typical skybox images might show a mountain range or cityscape on the horizon, for example.

This demo shows a balloon in the air inside a skybox. The skybox texture image was compressed with PVRTC4 using the skybox compression feature of the PVRTextureTool in the SDK.

Trilinear

This demo illustrates the visual difference between using the different texture filter modes available in OpenGL ES by allowing them to be compared. The supported texture filter modes are GL_LINEAR (Bilinear with no MIP Mapping), GL_LINEAR_MIPMAP_NEAREST (Bilinear with MIP Mapping) and GL_LINEAR_MIPMAP_LINEAR (Trilinear).

UserClipPlanes

This demo illustrates how to use user-defined clip planes. A rotating sphere is shown cut by six user clip planes.

Vase

This demo shows a slowly rotating transparent vase with dynamic reflections on its metallic sections.