Targeting 3D Applications for Mobile Devices Powered by OpenGL* and OpenGL ES*

By Dan Ginsburg

The OpenGL* graphics engine has for years been a sound choice for many game developers, but the proliferation of alternatives such as Linux*, Apple Mac OS*, and Google Android* 3D-enabled devices means that OpenGL and OpenGL ES* (the embedded version of OpenGL) power many current and future games on portable devices. In this article, uncover the history of the migration of OpenGL to mobile platforms, discover the current state of mobile 3D graphics on existing devices, and learn how you can target your 3D application to the wide range of upcoming devices powered by the Intel® Atom™ processor Z6xx series (formerly codenamed “Mooresstown”)—a system-on-chip (SoC) with programmable 3D graphics acceleration that will span a wide range of devices, from handheld phones to netbooks.

OpenGL* on Handheld Devices

OpenGL began its life nearly two decades ago and has become the de facto standard for 3D graphics applications on all flavors of UNIX*, Linux*, and Mac OS* across a wide range of hardware platforms. As handheld and embedded devices began to emerge with 3D graphics capabilities, many industry players identified the need for a standard mobile 3D graphics application programming interface (API). As OpenGL had proven itself as a successful open standard for 3D graphics on various platforms, it was chosen as the basis for creating a new standard called OpenGL for Embedded Systems (OpenGL ES). OpenGL ES was designed to address the specific needs of handheld devices—namely, limited memory, power constraints, a wider heterogeneity of platforms, and a need for simplified drivers.

The first version of OpenGL ES was based on OpenGL version 1.3 and contained a fairly basic set of 3D graphics functionality. OpenGL ES 1.0 proved to be a successful standard and enabled the first generation of handheld 3D applications. Recognition of the constraints of OpenGL ES 1.0, such as no required support for multi-texturing or vertex skinning, provided the impetus for developing OpenGL ES 1.1. This version enabled many more fixed-function features of OpenGL 1.5, including vertex skinning, dot3 bump mapping, and vertex buffer objects. OpenGL ES 1.1 has to date been the most ubiquitous version and was supported in Apple iPhone* (and, in fact, all devices powered by the Imagination Technologies POWERVR* MBX graphics processing unit [GPU]). Figure 1 shows the progression of OpenGL ES development.
Targeting 3D Application for Mobile Devices

Figure 1. The OpenGL ES* development timeline

The major inflection point for OpenGL ES was the release of version 2.0. This version was based on OpenGL 2.0 and introduced the world of programmable vertex and fragment shaders into the embedded space. The amount of functionality now enabled on handheld devices is remarkable. Within a few short years, graphics on handheld devices evolved to Microsoft DirectX* 9–level functionality. Now, OpenGL ES 2.0 is making inroads in many devices and is poised to be the future of graphics on embedded devices, including those powered by the Intel Atom processor Z6xx Series Family.

Intel® Atom™ Processor Z6xx Series Graphics

The introduction of the third generation Intel® Atom™ processor (the Intel® Atom™ processor Z6xx series) is a significant event for the handheld market. Until now, the majority of handheld devices have been powered by ARM*-based CPUs. However, with the Intel Atom processor Z6xx series, Intel is bringing already hugely successful Intel Atom microarchitecture x86 technology from the netbook market and expanding it into handheld devices (see Figure 2). There are variants of the individual Intel Atom processors Z6xx series in terms of clock frequency and power consumption, but all will share a common graphics feature set. The processor comes with the Intel® GMA 600 GPU, which contains a licensed Imagination Technologies POWERVR* SGX 535 GPU. The Intel GMA 600 GPU supports OpenGL ES 2.0, OpenGL 2.1, and DirectX 10.1. The GPU can be clocked up to 400 MHz (whereas the Intel® GMA 500 GPU topped out at 200 MHz). It supports HIS Technology Shader Model* 4.1, which exceeds the requirements of OpenGL ES 2.0.

There are several options for programming to the graphics capabilities of the Intel Atom processors Z6xx series. One option will be on devices that run MeeGo*—a Linux-based operating system that is targeted for a variety of devices, including handsets, netbooks, and tablets. MeeGo is
a joint effort between Intel and Nokia* that will power cell phones and netbooks of the future, and the operating system includes native support for OpenGL ES 2.0. An interesting library for MeeGo named Clutter, on which the MeeGo user experience (UX) is based, abstracts the details of whether the underlying graphics implementation is OpenGL or OpenGL ES, providing an easy way to write portable 3D application to the Intel Atom processor Z6xx series architecture. Table 1 shows the SDKs and APIs available for MeeGo, Microsoft Windows*, and Android.

Table 1. SDKs and APIs available for graphics creation

<table>
<thead>
<tr>
<th>Operating system</th>
<th>SDK/API</th>
</tr>
</thead>
</table>
| MeeGo*           | • Intel AppUp℠ Developer Program SDK  
|                  | • Clutter (abstracts OpenGL 2.0 vs. OpenGL ES 2.0 differences) |
| Windows*         | • Intel AppUp Developer Program SDK |
| Android*         | • Android* Native Development Kit (provides OpenGL ES 2.0 support) |

In addition to MeeGo, the Intel Atom processor Z6xx series family will be present in netbooks that run the Windows operating system. For both MeeGo and Windows, Intel has created a developer SDK and app store specifically for Intel Atom processor-based devices called Intel AppUp℠. The Intel AppUp℠ Developer Program SDK and Intel AppUp℠ Center provide an SDK and distribution mechanism to sell your application to the large installation base of Intel Atom processor-based devices. As the processor family moves into other platforms, such as smart phones, Intel will migrate the Intel AppUp Center there, too. For 3D games, you will want to target OpenGL/OpenGL ES to be portable across both MeeGo-based and Windows-based netbooks.

Figure 2. The Intel® AppUp℠ Developer Program Software Development Kit (SDK)
Targeting OpenGL* 2.0/OpenGL ES* 2.0 for 3D Graphics

The majority of the OpenGL ES 2.0 operating system is a subset of OpenGL 2.0, but some features were specifically added for embedded devices, such as offline shader compilation and reduced precision shader variables. However, as OpenGL ES 2.0 has grown in popularity, the OpenGL Working Group has recognized the importance of providing easy portability between OpenGL ES 2.0 and OpenGL. To that end, the OpenGL 4.1 specification incorporates a new extension (GL_ARB_ES2_compatibility) as of July 25, 2010, that makes OpenGL 4.1 fully capable of directly running OpenGL ES 2.0 applications.

As a developer—especially one coming with some background in OpenGL—you will want to be aware of which features you need to target to make your 3D engine easily portable across OpenGL ES 2.0 and OpenGL 2.0. One easy way to get started on a desktop PC is to download the Kronos OpenGL ES 2.0 SDK for POWERVR* SGX. The POWERVR SDK provides an OpenGL ES 2.0 wrapper that runs on top of desktop OpenGL on any PC with an OpenGL 2.0–capable GPU. The SDK also includes documentation, with a set of performance recommendations that will be highly relevant to writing high-performance 3D applications for the Intel GMA 600 GPU.

Be aware that some features available in desktop OpenGL 2.0 do not exist in OpenGL ES 2.0. You’ll need to avoid these features, for example, to attain portability:

- Immediate mode and display list geometry specification
- All fixed-function processing:
  - Fixed transformation and lighting
  - Matrix stack and matrix transformation operations
  - Texture-coordinate generation
  - Fog
- User clip planes
- Point rendering (only point sprites are supported)
- Line/polygon smoothing and stipple
- Pixel rectangles and bitmaps
- 1D textures and some texture parameter/wrap modes
- Occlusion queries

The following sections provide guidance on how to write portable OpenGL code that handles the differences in the geometry specification, geometry transformation, and shaders.

**Geometry Specification**

In OpenGL 2.0, there are four ways to specify geometry: immediate mode, display lists, vertex arrays, and vertex buffer objects. Immediate mode refers to the specification of geometry between glBegin()/glEnd(), where each vertex attribute is specified by a function call. Display lists are used to compile sets of immediate mode calls into later execution. Neither of these features exists
in OpenGL ES 2.0 for performance and driver size reasons. Instead, you must specify all geometry using vertex arrays and vertex buffer objects. In general, use vertex buffer objects, as they reduce the amount of bandwidth necessary for geometry transfer between the CPU and GPU. However, regular vertex arrays can be appropriate in cases where the geometry is dynamic. The easy rule here is that you should specify all your geometry with vertex buffer objects where possible and vertex arrays as a fallback.

**Geometry Transformation**

Perhaps the change that takes the most getting used to between OpenGL 2.0 and OpenGL ES 2.0 is the lack of transformation functions. In OpenGL 2.0, the API has function calls for loading the model view, projection, and texture matrices. There are also several convenient functions for assembling these matrices, such as `glTranslate`, `glRotate`, and `glScale`. None of these functions exists in OpenGL ES 2.0, nor do any of the lighting-related functions: OpenGL ES 2.0 removed the fixed-function pipeline, and you are responsible for computing your own matrices and performing transformations and lighting in the vertex shader.

A common solution here is to develop your own matrix library in your engine that provides matrix functions similar to those in OpenGL. In both OpenGL and OpenGL ES, you can load your own computed matrices into uniform matrix variables in the vertex shader. If you develop your own library that performs the matrix math, you can use the same code and shaders for both APIs.

**Shaders**

The next big difference between OpenGL 2.0 and OpenGL ES 2.0? OpenGL has vertex shader built-in variables to access fixed-function state (such as `gl_ModelViewMatrix`). OpenGL ES 2.0 does not include the fixed-function pipeline, so these variables are no longer present. So, you must manage this state by loading it into uniforms.

In addition, in OpenGL, the attributes associated with vertices have meaning relative to the fixed-function pipeline, such as `color`, `secondary color`, `texture coordinate`, and `normal`. The fixed-function pipeline would then use these attributes in specific ways. For example, you would use `normal` for lighting and the `texture coordinate` for fetching from textures on the associated texture unit. In OpenGL ES 2.0, vertex attributes are generic—that is, the meaning of the data associated with each vertex attribute is up to the vertex shader. As such, the vertex array entry points for loading fixed-function vertex arrays such as `glColorPointer()` and `glNormalPointer()` do not exist. Instead, you specify all geometry through `glVertexAttribPointer()`.

Fortunately, there is another easy rule here: Use only generic vertex attributes. Further, do not use any of the fixed-function built-in variables in your shaders. If you follow these simple rules, you are most of the way to making your shaders portable across OpenGL and OpenGL ES. There are some other minor differences between the shaders (such as precision qualifiers), but generally speaking, the differences between OpenGL ES 2.0 shaders and OpenGL 2.0 shaders are easy to account for. In fact, if you think about what is going on inside the POWERVR OpenGL ES 2.0...
wrapper, the wrapper is doing programmatic transformation to the OpenGL ES 2.0 shader to make it run and compile on the underlying OpenGL 2.0 implementation.

**OpenGL ES 2.0 Feature Additions**

So far, I've been talking about features that exist in OpenGL 2.0 but not OpenGL ES 2.0. Now, let's look at the features that were added specifically for the needs of embedded devices:

- Shader binaries
- Framebuffer objects (available only as extension in OpenGL 2.0 and added to core OpenGL in version 3.0)
- Precision qualifiers
- Shader invariance

**Shader Binaries**

The shader binaries feature was introduced in OpenGL ES 2.0 and did not make its way back into OpenGL until version 4.1. The idea behind shader binaries is that you can compile your shaders offline (for example, before your application runs), and then at runtime load the compiled shader binaries onto the device. This feature was added to OpenGL ES 2.0, because shader compilation can be costly in terms of memory and time—especially on memory-limited handheld devices (see Figure 3). The catch with shader binaries is that the binary format is opaque. In other words, a shader compiled into a binary for one device is not guaranteed to (and is in fact unlikely to) run on another vendor's device. A strategy that is often used is to generate the binaries at installation time on the device and load them at runtime. Another good practice is to check the GL_VERSION string at runtime and recompile the shaders if a change is noticed. (New driver revisions might have newer formats or a compiler producing better output.)
Figure 3. Games such as "G: Into The Rain" are able to take advantages of netbook strengths to provide a rich graphical interface.

Summary

The Intel Atom processor Z6xx series family brings the x86 architecture to a wider range of devices than ever before and provides a powerful GPU that will enable 3D accelerated applications on netbooks running MeeGo and Windows along with handheld devices. The common API connecting your application to this 3D power is OpenGL and OpenGL ES. By targeting the OpenGL ES 2.0 subset of OpenGL 2.0, your engine can work across this wide range of devices and provide easy portability to the next generation of Intel Atom processor–based devices.

About the Author

Dan Ginsburg currently works at Children’s Hospital Boston as a Principal Software Architect in the Fetal-Neonatal Neuroimaging and Development Science Center where he uses the GPU for accelerating neuroimaging algorithms. Dan has been working with GPUs for more than ten years in medical imaging, graphics, and gaming. He is co-author of the OpenGL ES 2.0 Programming Guide and contributed to the OpenGL Shading Language 3rd Edition.