Challenges in multi-platform 3D-games development

Maciej Kamiński

February 8, 2013

1 Reasons for multi-platform development

Multi-platform application development is crucial both for business reasons and for research reasons.

Business advantages of multi-platform development include better investment protection and broader customer base. While the second advantage is pretty obvious (supporting more platforms gives access to more customers) the first point needs to be elaborated further. The less application is platform dependent, the less likely its market share is likely to suffer when one of supported OSes becomes obsolete. Moreover: the less application is platform-specific, the easier it is to adapt application to new popular platforms (and therefore to take advantage of market changes).

Research advantages include ability to study differences between computers (hardwares and operating systems), as well as ability to study capabilities and limitations of various programming tools/techniques in addressing these systems (and other normal programming challenges).

2 Challenges imposed by multi-platform applications with focus on interactive 3D games

Each class of applications is bothered with different set of technical challenges. For games one of crucial concerns are interactivity and graphics, but also sound. Actually graphics API, OpenGL, is one of most widely respected standards in game industry. While alternatives do exist (both as high level libraries and as close-to-the-metal APIs preferred by video game console manufacturers), OpenGL in one form or another is available on almost all computing platforms in existence.

Additionally OpenGL is a vendor-neutral standard governed by community of all interested parties (software and hardware companies participating in Khronos Group, earlier OpenGL Architecture Reviev Board), which makes it a de facto standard for all graphics applications designed with portability in mind.

Thanks to quality and and availability of OpenGL implementations, in typical OpenGL based game rendering pipeline is actually the most portable part, and one that requires smallest amount of platform-specific adjustments. On the other hand OpenGL standard covers only rendering-related functions, which is insufficient for game development.

To display textured model, files must be somehow accessed, textures have to be decoded, et caetera, et caetera. Each mobile platform has its own specific APIs for these things, and some mobile platforms lack even most common libraries like JPEG (which are available on almost all PC based operating systems). Also setting up rendering context, interactivity and many other things are handled differently, however all this things can be handled for all platforms well without significant penalty in performance or code quality.

Regarding 3D sound, which is another integral component of 3D game, there is OpenAL standard which, while not as common as OpenGL, is quite easy to use and mature.

3 Labyrinth as multi-platform 3D game

(Labyrinth is working title of game, I am writing as part of this research project)

To verify ideas of portability, I have developed a (simple) multi-platform 3D game of RPG genre[1]. Currently single code-base builds for 4 different configurations:

- iOS - with XCode and iOS SDK - Can be tested on iPhone Simulator and on any appropriately licensed iPhone with current Firmware.
- Android - with Android NDK, using NativeActivity (no Java code[2]). Can be tested on any Android-based phone with 2.3.3 or higher version of Operating System. Can't be tested on Android Emulator from Google Android SDK, because it doesn't support OpenGL|ES 2.0.
- Linux with native EGL/OpenGL|ES 2.0 - useful only with Intel[3] drivers.
- UNIX port with glX/OpenGL 2.1. (works on Linux, Mac OS X and possibly other Unix implementations)

3.1 Why two Linux ports

Actually only one of ports is Linux-only, ES 2.0 one. OpenGL 2.1 port is UNIX (which is superset of Linux).

Linux with native OpenGL|ES 2.0 is actually the first version of game that worked, and OpenGL|ES on Linux is very useful for testing all gameplay and OpenGL|ES 2.0 related features. While game is targetted for Android/iOS, it is useful to have it working on development platform. Moreover, working Linux port gives ability to run Linux-specific

[1]As of writing this, February 7th, game isn't yet complete
[2]NativeActivity actually interfaces with Android using Java-Nati
[3]possibly AMD too, not tested. Certainly not NVidia.

memory-debugging tools such as Valgrind (utility that detects sources of memory leaks and causes of memory corruption).

OpenGL 2.1 port on the other hand is recent. It has two basic advantages: first of all it works on many UNIX/X11 platforms, and secondly it verifies source-level comatibility (sort of, will be elaborated in further section) between desktop OpenGL and OpenGL|ES, and therefore proves that it is possible (if needed) to do Windows or Native MacOS port if needed (none of these platforms contains native OpenGL|ES).

This OpenGL 2.1 UNIX port needs very little alterations (conditional compilation) to make it useful on both, Linux and Mac OS X. Build process is of course different (Mac OS version uses XQuartz, while Linux uses distribution-provided X.Org)

3.2 Ports comparison

Table 1: Comparison of platform specific features

Platform	iOS	Android	PC Linux (Native ES)	UNIX (glX)
Language	Objective C/C++	C/C++	C/C++	C/C++
Compilers	Apple LLVM 4.2	GCC 4.6.3	GCC 4.7.1	GCC 4.7.1 LLVM 4.2
Testing platforms	iOS 6.1	Android 2.3.3 Android 4.0	Debian	Debian OS X Lion
OpenGL	ES 2.0	ES 2.0	ES 2.0	2.1
Context management	EAGL	EGL	EGL	glX
	Non-graphics-specific			
Asset access	limited filesystem	proprietary	filesystem	filesystem
OpenAL	system	self-ported	system	system
input handling	proprietary	proprietary	Xlib	Xlib

Despite these vast differences between platforms, out of 10328 lines of game code 8451[4] isn't platform specific at all. Out of lines of platform-specific code, there is:

- 558 lines of iOS specific code (objective C and C++)
- 511 lines of Android specific code (C and C++)
- 808 lines of UNIX specific code (C++) which includes two different OpenGL drivers. (Native Linux OpenGL|ES version and Unix OpenGL2.0 version have common subset)

[4]As of writing this, nothing is complete, but proportion is likely to remain constant. When chapter will be included in final thesis, figures will be updated

4 iOS

4.1 Common problems

iOS port of an application requires that code interfacing with Operating System (especially touchscreen input processing, GL context management and application state management) is done using Objective C - language favored by Apple. However this does not affect rest of the application (OpenGL rendering, OpenAL playback etc.) which can be written in more common languages of C/C++.

iOS application can access its assets (textures, config files, audio files) via filesystem, but when it is installed, all files are included in single directory. therefore it is impossible to have two assets with the same names[5]. Normal POSIX API (fopen, fclose) works.

iOS has its own OpenAL implementation, but doesn't have OGG/Vorbis codecs, so when one wants to include audio tracks in this format, one has to build OGG/Vorbis libraries for iOS. Similarly libJPEG is missing in Operating System and must be compiled for application and included with it.

iOS SDK is available only for MacOS X, therefore Apple computer has to be used during iOS specific development.

Several standard headers on iOS are located differently than on most systems (this can be fixed with `#ifdef __APPLE__`).

Table 2: Header location

iOS	Most Systems
`#include <OpenAL/al.h>`	`#include <AL/al.h>`
`#include <OpenAL/alc.h>`	`#include <AL/alc.h>`
`#include <OpenGLES/ES2/gl.h>`	`#include <GLES2/gl2.h>`

4.2 Good things

iOS SDK includes iPhone Simulator, which is good. iOS application is (for purpose of being executed on said simulator) compiled to native x86 code, and therefore yields reasonable performance. Simulator can represent virtually all devices ever manufactured by Apple with iOS operating system, and simulates many events that happen on such device, therefore it is an useful development tool.

iOS market is tightly controlled by Apple, and it's pretty uniform with strict quality standards.

[5]It's not a problem in Linux or Android, where assets are in hierarchical directory structure. One must however be careful when naming files: adding two files with the same name to iOS application will result in adding only one of these files, at random. No warning from build tool.

5 Android

5.1 Common problems

Android (since API level 10) allows applications to be written in native code (earlier versions of Android required applications to be partially or fully written in Java). This is not a new feature, but still native development framework for Android is less than mature. Many Android APIs are available only via JNI[6].

Android SDK includes an emulator, but it is terribly slow and does not support OpenGL|ES 2.0 at all, therefore it is essentially useless for game development.

Assets in Android Application are part of APK file. While they do form normal tree-like directory structure, they can't be accessed using normal POSIX filesystem API - instead Android provides proprietary asset access API defined in `"android/asset_manager.h"`

Android doesn't provide OpenAL API. Instead it provides OpenSL|ES API, which is more complicated and of lower level. My approach was to port OpenAL-Soft library with OpenSLES backend to Android.

Android Application lifecycle dictates, that when application is minimized, OpenGL context is invalidated. Therefore after resuming application from background all graphical assets (textures, shaders, VBOs) must be reinitialized[7].

iOS has its own OpenAL implementation, but doesn't have OGG/Vorbis codecs, so when one wants to include audio tracks in this format, one has to build OGG/Vorbis libraries for iOS. Similarly libJPEG is missing in Operating System and must be compiled for application and included with it.

Android phone market is highly fragmented and it is impossible to test application on all of them (I am testing application on two phones from different manufacturers, with different OS versions - and I have observed, that sometimes app works well on one and doesn't work at all on the other).

5.2 Good things

Android applications can be developed using any PC or Mac and any Android-based telephone, therefore it's cheap.

Android NDK (Native Development Kit) contains ndk-gdb tool, that allows debugging application running on telephone connected over USB cable.

5.3 Regarding implementation of Android lifecycle management

Android application (this is not different on other platforms) receives sequence of commands (messages) from operating system. To process these messages, application has to

[6]Java Native Interface - technology allowing calling Java routines from Native Code and vice versa.

[7]Android 4.0 provides functionality to preserve OpenGL context/state when application is minimized, but I haven't yet tested. And besides that: many good Android-based phones won't get official 4.0 upgrade

define callback function with signature `engine_handle_cmd(struct android_app* app, int32_t cmd)`. Of these messages particularly important are `APP_CMD_INIT_WINDOW` and `APP_CMD_TERM_WINDOW`: these are issued every time, app is hidden/shown. Every time, window is hidden, all objects held by GPU (textures, shaders, vertex buffer objects) become invalid. Therefore initialization routines have to be called every time, window is shown. Only OpenGL assets need to be initialized - all other handles (as file handles or OpenAL handles) remain valid.

As game assets (unlike on other platforms) have to be reinitialized many times, it is important to avoid any form of memory leaks (otherwise, memory footprint of application may grow considerably during application runtime).

6 Regarding desktop OpenGL vs. OpenGL|ES 2.0

OpenGL|ES 2.0 standard was developed from OpenGL 2.1, however it is neither its superset nor its subset. OpenGL|ES 2.0 eliminates fixed pipeline alltogether, but it includes for instance glGenerateMipmap routine which is not present in OpenGL 2.1. Also shader syntax is different for OpenGL|ES 2.0. ES requires precission statement, while standard OpenGL 2.1 does not support them.

First version of desktop OpenGL that is fully superset of OpenGL|ES 2.0 is OpenGL 4.2, bit it is quite rare (only minority of GPUs/drivers support it).

Still it is possible to have common desktop and ES code using OpenGL 2.1 and OpenGL|ES 2.0, as common subset of these two is usable.

Each shader can for example conditionally include precision statement

```
#ifdef GL_ES
     precision mediump float;
#endif
```

Regarding mipmap generation similar approach works:

```
#ifdef DESKTOP_OPENGL
  gluBuild2DMipmaps(GL_TEXTURE_2D,h->tx.color_mode,
                    h->tx.width, h->tx.height,
                    h->tx.color_mode,
                    GL_UNSIGNED_BYTE, h->tx.data);
#else // mobile OpenGL
  glGenerateMipmap(GL_TEXTURE_2D);
#endif
```

7 Summary

Application that has successfully been deployed on two vastly different platforms is much easier to port to any third platform than application that is developed and deployed on one platform only. Labyrinth has been succesfully deployed on four different operating systems (Android, iOS, GNU/Linux and OS X).