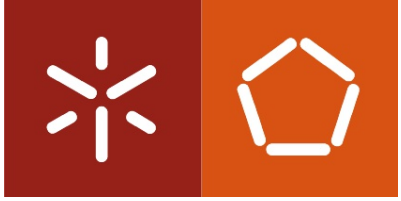
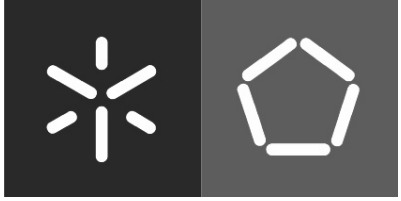


Universidade do Minho
Escola de Engenharia

Leander Edward Bessa Beernaert

Cross Platform 3D Rendering Engines and Mobile Devices/Smartphones

Outubro de 2011



Universidade do Minho
Escola de Engenharia

Leander Edward Bessa Beernaert

Cross Platform 3D Rendering Engines and Mobile Devices/Smartphones

Dissertação de Mestrado
Mestrado em Engenharia Informática

Trabalho efectuado sob a orientação de
Doutor António Ramires Fernandes

Outubro de 2011

DECLARAÇÃO

Nome

_____

Endereço electrónico: _____ Telefone: _____ / _____

Número do Bilhete de Identidade: _____

Título dissertação □ / tese □

_____

_____

_____

Orientador(es):

_____

_____ Ano de conclusão: _____

Designação do Mestrado ou do Ramo de Conhecimento do Doutoramento:

_____

Universidade do Minho, ____ / ____ / _____

Assinatura: _____

Acknowledgements

I would like to express my gratitude to my supervisor, António Ramires Fernandes, whose encouragement, supervision and support throughout my dissertation enabled me to develop a greater understanding of the subject.

To my mother, brother and family, for their continuos encouragement, patience and support.

To David James Peace, for the last-minute proof-reading.

To my friends and all of those who supported me in any respect during the completion of my dissertation.

Abstract

Now more than ever, we live in a cross platform technological world. We are surrounded by various platforms, each with their own set of advantages and drawbacks. We've come to a point where we cannot delay the transition of software from one platform to another. This has become increasingly more visible with the "rise of the smartphones". Their evolution has sparked quite an interest and due to their ubiquitous nature and, CPU and GPU performance, they prove to be very interesting and useful computing devices.

The aim of this dissertation is to port the 3D rendering engine, Curitiba, developed at the Department of Informatics of the University of Minho, currently being developed on Windows, to the second and third most popular platforms: Mac OS X and iOS (iPhone and iPad)[1], respectively, and create one unified project. Due to incompatibilities presented by the wxWidgets [52] toolkit (2.8.x) on Mac OS X (10.6 and greater), we ported Curitiba to the GNU/Linux platform first since it's also POSIX compliant. Sadly, the Android platform had to be left out because, at the time, it lacked the support for C++'s STL and Exceptions.

Throughout this dissertation we shall cover all the challenges faced to transform Curitiba into a cross platform software and the development of the resulting unified project. Our secondary objective is to replace the traditional keyboard and mouse interactions in a 3D rendering engine by implementing new interaction models which make use of the touch screen and/or the sensors available on the iOS platform.

Keywords: Cross platform software, cross platform methodologies, OpenGL, OpenGL ES, Sensors in Mobile Devices, C/C++, iOS, Mac OS X, Windows, GNU/Linux

[1] See this link for more information.

Resumo

Agora mais que nunca, vivemos num mundo tecnológico multi-plataforma. Estamos rodeados de várias plataformas, cada uma com as suas vantagens e desvantagens. Chegamos a um ponto em que não pudemos adiar mais a transição do software de uma plataforma para outra. Isto tornou-se gradualmente mais visível com a ”ascensão dos smartphones”. A sua evolução tem despertado bastante interesse e graças à sua natureza ubíqua e, desempenho ao nível do CPU e GPU. Estes demonstram ser sistemas computacionais bastante interessantes e úteis.

O objectivo desta dissertação é portar o motor de renderização 3D, Curitiba, desenvolvido no Departamento de Informática da Universidade do Minho, actualmente desenvolvido em Windows, para a segunda e terceira plataformas mais populares: Mac OS X e iOS (iPhone e iPad)[2], respectivamente e criar um único projecto. Devido a uma incompatibilidade com a ferramenta wxWidgets [52] (2.8.x) em Mac OS X (10.6 e maior), portamos o Curitiba para GNU/Linux primeiro visto que também implementa as normas POSIX. Infelizmente, tivemos que abandonar a plataforma Android devido a este, na altura, não possuir suporte para o STL e Excepções do C++.

Ao longo desta dissertação vamos abordar as dificuldades encontradas ao transformar o Curitiba num software cross plataforma e o desenvolvimento do projecto unificado. O nosso objectivo secundário consiste em substituir as interacções tradicionais com teclado e rato num motor de renderização 3D com novos modelos de interacção que tiram proveito do ecrã táctil e/ou sensores disponíveis na plataforma iOS.

Palavras-chave: Software multi-plataforma, metodologias multi-plataforma , OpenGL, OpenGL ES, Sensores em Dispositivos Móveis, C/C++, iOS, Mac OS X, Windows, GNU/Linux

[2] Veja este link para mais informações.

Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

Long gone are the days in which developers only targeted one platform to deploy their software on. Nowadays, developers strive to support the maximum number of platforms possible, as ignoring one for whatever reason will exclude a considerable audience from the final product. It should be noted that it is not always possible to develop cross platform software; nonetheless, we will focus our attention towards 3D rendering engines, most of which are cross platform.

Up until 2007, the most popular operating systems (OS) (those which occupied the highest market shares) were made up by our well know trio of GNU/Linux, Mac OS X and Windows. Since the introduction of the iPhone we have witnessed the ”rise of the smartphone”, devices which combine the features of both a mobile phone and a personal digital assistant (PDA) [16]. Due to their ever increasing popularity, the most popular operating systems can now be considered Windows, Mac OS X and iOS, as shown in Figure 1.1.

Most smart phones in 2011 have by far exceeded the simple merging of features between a mobile phone and a PDA, going so far as to create their own ecosystem [30]. Since they adopted an application model similar to the one present on the PC, the user installs the software he/she requires, they have become capable of performing a wide variety of tasks and, in some cases, going as far as replacing laptops and desktops altogether. Due to its ubiquitous nature and high adoption rate, this platform has sparked quite an attention from the industry. Many existing PC-only applications have been adapted to these devices and new applications have arisen up to take advantage of the platform’s unique characteristics.

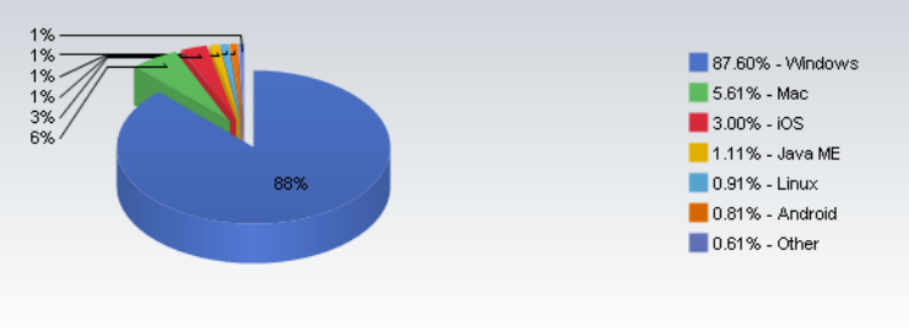


Figure 1.1: Pie chart illustrating the market share each operating system holds in June of 2011, courtesy of NetMarketShare.

Due to the hardware's evolution in these devices, more and more 3D rendering engines are being ported[1] as well (e.g.: Ogre 3D [38], Unity [46] and Unreal [47]).

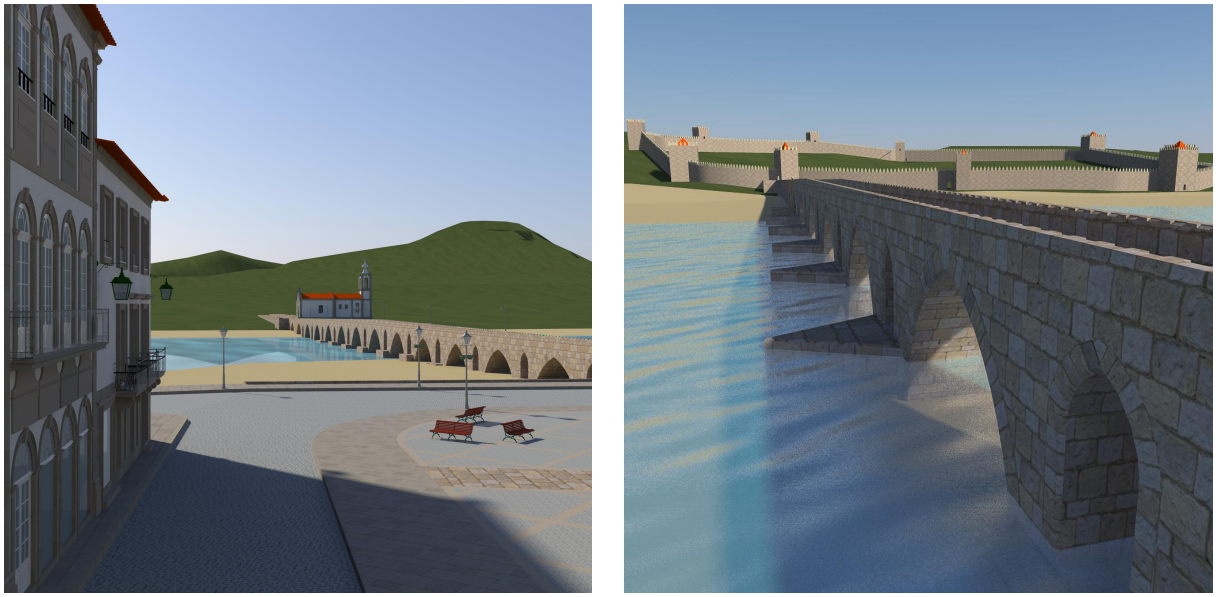
Developing cross platform software is not an easy task, specially from a stand point of compiled languages such as C/C++. When a large piece of software is ported to another platform we are bound to run into various issues, problems and incompatibilities. This further aggravates when developing for mobile platforms, due to the platform's hardware limitations [42]. Mobile devices usually come with limited computational power, CPU cache, memory, disk storage and smaller screen size and may not provide virtual memory.

1.1 Objectives

One of the goals of this project is to port the 3D rendering engine, Curitiba, developed at the Department of Informatics of the University of Minho, currently being developed exclusively on Windows, to the Mac OS X and iOS operating systems. Despite being solely developed on windows, Curitiba's architecture was designed to factor in the possibility of being ported to other platforms in the future. Therefore, it has refrained from using Windows only Application Programming Interfaces (API) and made sure all its auxiliary libraries are cross platform compatible. The porting process may provide valuable insights for other C/C++ based applications which need to be ported between these platforms in the future. In the end, we wish the obtain a single code base and project file capable of targeting all our desired platforms, thus facilitating future developments.

[1] Porting is the process of adapting software so that an executable can be created for a different computing environment than originally intended, such as operating system, CPU, compiler, among others.

The second aim of this project is to study possible interactions between 3D applications and the available sensors present on the iOS platform, such as the multi-touch screen, accelerometer, gyroscope and digital compass. This study is intended to provide an interaction model in which the device acts as a window into a virtual world and displays content based on the user's location. For instance, the Department of Informatics developed a virtual real scale representation of Ponte de Lima [37], Portugal, portraying the 21st and 14th century (Figure 2.2). As a user would move through the physical Ponte de Lima, the device would display, according to its position and orientation, the virtual equivalent of what the user would be seeing if he had happened to be standing at that exact location in the XIV century. The end result would be an augmented reality application which would act as a "window into the past".



(a) 21st century representation

(b) 14th century representation

Figure 1.2: Ponte de Lima 3D Project, a section of the bridge viewed from different locations and time periods.

1.2 Outline

The remainder of the document is organized as follows: In Chapter 2 we cover cross platform development techniques, sensor modules in handheld devices, key concepts regarding 3D

rendering and particular features of each of the targeted platforms. In Chapter 3 we detail the approach taken to port Curitiba as well as all the issues encountered along the way. Throughout Chapter 4 we describe how we developed the iOS application and Chapter 5 presents our resulting unified project. Finally, Chapter 6 provides closing thoughts, summarizes the work outlaid in this document and proposes paths for future development.

Chapter 2

Contextual Overview

2.1 Cross Platform Software

Cross platform software is defined as a software which applies computing concepts and methods in order to be implemented and inter-operate on multiple computer platforms [14]. A computer platform can be regarded as an operating system, a computer architecture or a combination of both. Computer software can be divided into the following categories: compiled, interpreted and pre-compiled bytecode. Compiled software, as its name suggests, requires the software to be compiled on each of the target platforms and is therefore the most prone to cross platform issues. While the last two do not require the software to be compiled on each platform and may be inherently cross platform by nature (e.g.: Java, Python), the underlying mechanism (interpreters or virtual machines) may have to deal with the same cross platform issues faced by compiled software on the targeted platform.

2.1.1 Areas Which Affect Software Portability

Here on out we'll take a view on cross platform software from a C/C++ standpoint, since Curitiba is being written in that language. According to [14, 18, 31] the following areas can affect the portability of any given piece of software.

Language The programming language used to develop the software is one of the key factors which influence software portability. Using an inherently cross platform language usually removes these concerns. Nevertheless, thanks to standardization efforts such as the ANSI [1] and C99[1] standards [31], the C/C++ language has become increasingly more portable. Enforcing these standards in compilers, which support them, and avoiding compiler specific extensions greatly increases the software's portability.

Compilers All C/C++ features covered beyond the language specification are left in the hands of each compiler vendor. A common caveat is the assumption that all the built-in types (short, int and long) occupy the same size in memory and the default char type is signed [31] on all compilers. While this does affect the software's portability, it can easily be overcome by verifying each type's size.

Binary Data One of the most common problems of binary data can be attributed to the platform's endianess, the representation of binary data stored in memory[2]. Directly outputting C structs and raw floating point values into binary form may lead to incompatibilities between compilers on the same or other platforms, since each one may have different forms of representing the same data in memory.

Standard Libraries The issues discussed here are only applicable to the C++ language. C++'s STL (Standard Template Library) extends the core features with additional components, such as containers, algorithms, among others. Although STL is part of the C++ standard, and using it will increase the software's portability, it's not supported on some platforms, mostly mobile or embedded systems where resources are scarce. The latter also applies to C++ Exceptions and Run Type Type Information) RTTI. For instance, some platforms provide support for C/C++ applications, although they do not provide any of the previously mentioned components. Whether to use these components or not should be decided as soon as possible, since they have a major impact on the software's architecture.

[1] Draft of the C99 standard.

[2] For more information please refer to this page.

Operating System Interface To access low-level functions such as process creation, threading, interprocess communication, and so forth, the software resorts to the underlying operating system's API. Since each operating system has its own system architecture, developers are often faced with a different set of API's.

Standards such as the Portable Operating System Interface (POSIX) [12,50] aim to solve this portability issue by providing a common interface. Although POSIX was originally intended to bridge the gap between the various flavors of UNIX at the time of its creation, it can also be applied to other operating systems as well. For instance, GNU/Linux and Mac OS X are POSIX compliant and Windows isn't. However, there are ways to create a POSIX compatible layer on Windows, such as Cygwin [45].

User Interface The User Interface (UI) can be regarded as the least portable feature on each platform, since these usually have their own proprietary toolkit. These toolkits are neither compatible amongst each other nor do they offer the same look and feel. One way around this is a cross platform Graphical User Interface (GUI) library, such as Qt [36] or wxWidgets [52]. Although these libraries ease the porting processes, most of the features they offer are made up of the lowest-common-denominator present on each supported platform and may therefore exclude platform specific components.

Build Tools Each platform provides its own set of build tools, which can vary from anything between build-scripts and integrated development environments (IDE). Any platform specific build tool tends to hinder the cross platform aspect of the application, as it may be limited to one platform only. Several build tools were developed to overcome this issue, such as CMake [27,51], Autoconf [22], QMake [35], among others..., as well as cross platform IDE's. Both allow the same software to be compiled among various platforms.

2.1.2 Design Techniques

Cross platform software can be developed by writing a different version for each of the targeted platforms, which requires everything to be written from scratch each time and is therefor more cost and time consuming, or writing the program in a generic cross platform code, thus using little or no code tailored to the different incompatible platforms. However, in the

long run, a hybrid approach may prove to be more appropriate. Another point to take into consideration, [18] recommends not writing cross platform components which can not easily be abstracted by developers or might adversely affect the users. Keeping the feature set common across all platforms and ensuring a common design will help reduce the creation of different code bases and tailoring work.

To avoid dealing with platform specific code, we could create or use a set of APIs representing an acceptable lowest-common-denominator interface across different platforms, in other words, an abstraction layer to the platform's low level interfaces. For instance, Boost [15] is a particularly well known and excellent cross platform library, which integrates nicely with C/C++. On the other hand, we could create layers of conditional statements which will activate when the respective platform is used. This approach will, however, clutter the code and render it more difficult to read. When writing cross platform code it's crucial to always compile and test the code on all the target platforms as soon as possible to ensure the code behaves as expected and no previous functionalities have been broken in the process.

Most user interfaces are developed using a cross platform GUI toolkit. These are usually available in two flavors: a set of programming interfaces (e.g.: Qt [36], wxWidgets [52]) or as a XML User Interface Language (XUL). The latter employs a XML-like language to describe the layout of the application. Styling is performed through Cascading Style Sheets (CSS), Document Object Model (DOM) and all events are handled through Javascript [14, 31].

In the end, resorting to languages which are inherently cross platform such as HTML, JavaScript, Java, among others, removes most or all cross platform issues from the equation. Despite these advantages, performance critical applications tend to avoid said languages because most of these languages are either interpreted or compiled to byte-code, therefore incurring performance penalties.

2.2 OpenGL for Embedded Systems

OpenGL ES [25] is a 3D graphics API for embedded devices based on a subset of OpenGL [24]. The main goal of this API is to bring high performance 3D graphics to embedded devices while at the same time reducing memory bandwidth to save battery power. Currently there are three specifications available: ES 1.0, ES 1.1 (both ES1.0 and ES1.1 are hereon

out referred to as ES1.x) and ES 2.0. They are based on the OpenGL 1.3, 1.5 and 2.0 specifications, respectively.

Since ES1.x was defined relative to the fixed function pipeline of OpenGL, the API is by nature fixed function as well. Meanwhile, ES 2 emphasized OpenGL's 2.0 programmable graphics pipeline, adopting a fully programmable pipeline. Contrary to what we witness with OpenGL, OpenGL ES 2 API is not backwards compatible, since it has no support for the fixed function pipeline.

Despite OpenGL 2.0 having a programmable pipeline, it only allowed us to modify the fixed function's pipeline's behavior. All the data is still supplied through the fixed pipeline variables. ES2's pipeline, Figure 2.1, closely resembles that of the OpenGL 3 Core Profile[3], where we must define what data is passed on to the pipeline and implement its key stages: vertex and fragment shader [33,41]. This allows us to save memory, by reducing the number of variables present in the pipeline, and increase its performance by avoiding unnecessary calculations.

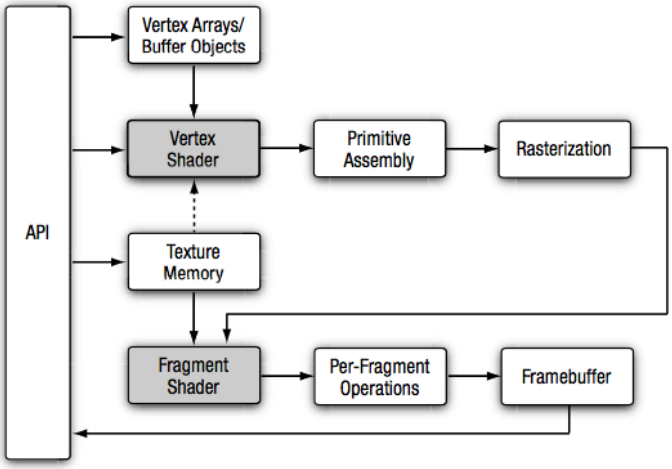


Figure 2.1: Diagram of the OpenGL ES 2 pipeline, courtesy of [33].

2.3 Sensor Modules

In this section we wish to present a brief overview of each of the sensors we might use to replace traditional input methods from the keyboard and mouse. These sensors can either

[3]OpenGL 3 Core Profile

be used independently or in combination with others [26, 40]. Our target devices, iPhone 4 and the iPad 2 3G, have the following sensors:

Accelerometer Measures the linear acceleration of an object around each of its three spatial axes [3, 40]. For instance, it could be used to detect the absolute orientation of the device.

Gyroscope Measures the rate at which a device rotates around each of its three spatial axes [3, 40].

Magnetometer Also known as magnetic compass, measures the orientation of an object relative to the magnetic north of the earth [40].

Assisted GPS Calculates the latitudinal and longitudinal coordinates of the device's location. Contrary to normal GPS's, which use radio signals from satellites alone, A-GPS uses additional network resources to locate and utilize the satellites faster and performs better in poor signal conditions [49].

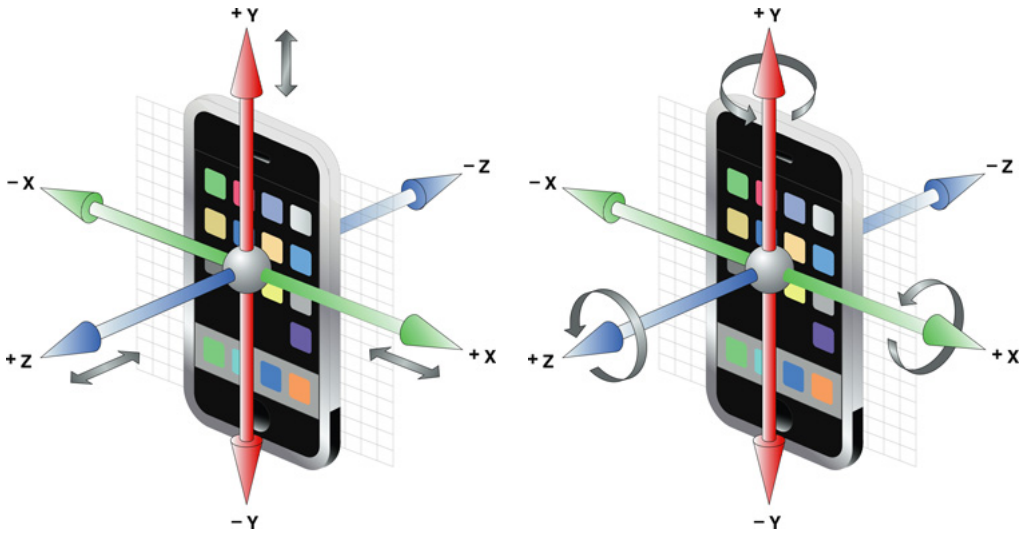


Figure 2.2: Illustration of the spatial axes present on an iOS device for the Accelerometer (left) and Gyroscope (right). Courtesy of [3].

For instance, for the PL3D application we could employ an approach similar to [39, 43]. Our application would use the GPS to determine the position, and the magnetic compass to determine the user's orientation. The accelerometer and the gyroscope could be used as backup systems if one of the previous components ceases to work.

2.4 Apple's Bundles

GNU/Linux and Windows store their application files and resources, header files and libraries in separate pre-determined locations on the filesystem. Mac OS X and iOS on the other hand, group these files into bundles [7]. Application files and resources are grouped into application bundles, and headers and libraries into framework bundles [8]. All the resource files present in these bundles should be accessed with the Core Foundation Bundle Services or the Cocoa NSBundle class to ensure the path to these files is always correct. We'll briefly overview each bundle's anatomy, since later on (Chapter 5) we'll need to reconstruct them manually.

2.4.1 Application Bundles

As mentioned previously, the application bundle is a hierarchical directory that encapsulates everything the application requires to run. The minimal requirements to create an application bundle are a program executable and the Information Property List file (`Info.plist`) [4], which contains configuration information about the application. Apart from these two mandatory files, the application can bundle any other files deemed necessary for its execution.

```
MyApp.app/
   Contents/
      Info.plist
      MacOS/
        MyApp
      Resources/
        MyAppIcon.png
        MainWindow.nib
      Frameworks/
        privateFramework.framework
```

Listing 2.1: Mac OS X Application Bundle directory hierarchy

```
MyApp.app/
   MyApp
   MyAppIcon.png
   MySearchIcon.png
   Info.plist
   Default.png
   MainWindow.nib
   Settings.bundle
   MySettingsIcon.png
   MainWindow.nib
```

Listing 2.2: iOS Application Bundle directory hierarchy

Listing 2.1 and Listing 2.2 illustrate the directory hierarchy in each of the application bundles. On Mac OS X, the application's executable is stored in the `MacOS` directory and all resources are stored in the `Resources` directory. On iOS everything is stored at the top-level bundle directory, except the files which need to be localized. There are more folders and conventions available to each application bundle. Please refer to [7] for more information on the subject.

In both Listing 2.1 and Listing 2.2 we have files ending with the `.nib` extension and in Listing 2.2 there's one file entitled `Settings.bundle`. The former represent binary objects which hold data pertaining to the user interface[4], generated with the interface builder tool, and the latter hold structured data which represents the application's setting in iOS's Settings application (Figure 4.5). According to [5] the application's preferences should either be either implemented inside the application itself or through a Settings bundle.

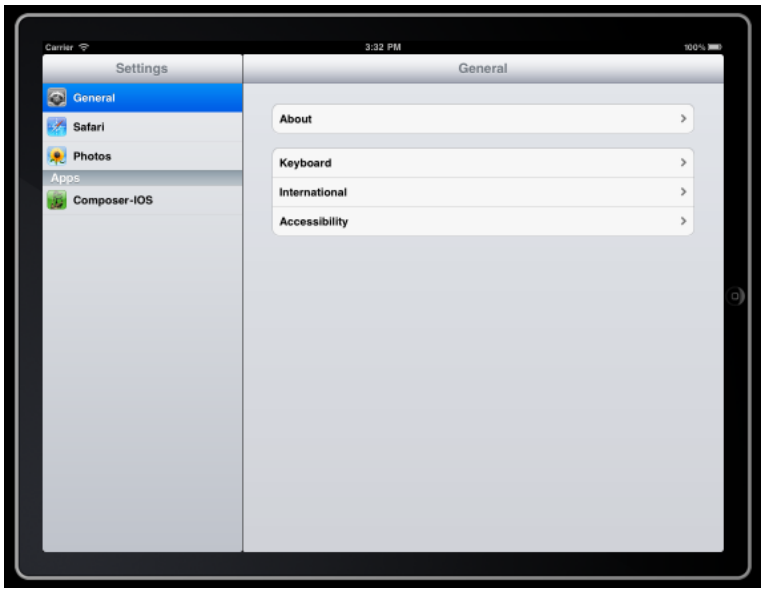


Figure 2.3: Screenshot of the Settings application running in the iPad Simulator in landscape mode.

2.4.2 Framework Bundles

A framework is a hierarchical directory that encapsulates shared resources, such as shared libraries, nib files, image files, localized strings, header files, and reference documentation in a single package. Multiple applications can use all of these resources simultaneously. The system loads them into memory as needed and shares the one copy of the resource among all applications whenever possible [8].

```
MyFramework.framework/
    Headers -> Versions/Current/Headers //Symbolic link
    MyFramework  -> Versions/Current/MyFramework //Symbolic link
```

[4]More information available through Apple Developer website.

```
Resources     -> Versions/Current/Resources //Symbolic link
Versions/
    A/
        Headers/
        MyFramework
        Resources/
            Info.plist
    Current   -> A //Symbolic link
```

Listing 2.3: Mac OS X Framework Bundle directory hierarchy.

The Framework bundle is based on an older bundle format and thus allows for multiple versions of the library to exist inside it. To maintain various versions of the same library, each of the versions is stored in a separate folder under `Versions`. All folders and files present at the root bundle directory are symbolic links which point to the latest version. As seen in Listing 2.3, we also have a resource folder in which a **mandatory** Information Property List file (`Info.plist`) is stored.

Chapter 3

Porting Curitiba

Our main goal is to get Curitiba up and running on both Mac OS X and iOS. The programming interfaces between them are very alike if not identical, except for the user interface. Since Mac OS X is more accessible than iOS[1] it was decided to port the engine to Mac OS X first and then to iOS.

In cross platform development, sooner or later we will encounter some unforeseen issues. Curitiba's front end application, Composer, relies on the wxWidgets [52] toolkit version 2.8.x as the UI component. The Mac OS X version of toolkit depends on the deprecated Carbon API(C based framework, 32bit), which has been discontinued since Mac OS X 10.5 in favor of Cocoa (Objective-C based framework, 64bit) [2]. Even though there is some limited support for 64bit based Carbon, many features have been removed or marked as deprecated.

The machine used for development ran Mac OS X 10.6 at the beginning of the development and, at the time of writing, is now running Mac OS X 10.7. Despite the previous incompatibilities, we've managed to get wxWidgets [52] to compile and run on Mac OS X 10.6. However, the windows were unresponsive to any events. The official wxWidgets [52] page claims the toolkit does work on 10.6, but, sadly, we did not manage to do so, and on Mac OS X 10.7 the toolkit cannot be compiled. wxWidgets [52] version 2.9 is being developed with Cocoa, however it still hasn't reached a stable release and Composer needs to be updated to support the newer version before this option can be considered.

[1] At the begging of the project, there was no iOS device available for testing purposes and developing on the device requires a license from Apple.

Since wxWidgets [52] is cross platform and we needed to ensure the front end application would be cross platform as well, we decided to port Curitiba and Composer to GNU/Linux before moving on to Mac OS X and iOS for to the following reasons:

1. Both GNU/Linux and Mac OS X are POSIX compliant, hence porting Curitiba and Composer to GNU/Linux will ensure a greater level of compatibility with Mac OS X. The time expended on this process won't go to waste and we'll have the added benefit of supporting another platform.

2. GNU/Linux is the most accessible system to us which supports the wxWidgets [52] toolkit and also the only remaining option available to test the front end application.

As mentioned previously, Curitiba is being developed on Windows and has never discarded the possibility of being ported to other platform later on. At first glance, Curitiba's cross platform dependencies seem to be mainly related with file system operations, user interface and the 3D rendering API, but then again these things are never that simple.

3.1 Cross Platform Issues

While reading through this section, we have to take into account that the engine was conceived as a product of a master thesis [19] and further work on the engine has been done by members of the Informatics Department and supervised first year MsC students. The latter aren't always comfortable or sensibilized towards cross platform development with the C/C++ language.

3.1.1 Platform Detection

The first step towards building cross platform software is to become aware of the platform and compiler through which the application is being built. Curitiba's awareness was limited to a statically defined macro which identified the Windows platform. The three most common practices regarding these aspects are: built-in flags, compiler arguments or configurations files.

The first approach attempts to determine the platform and compiler by detecting the presence of specific macros. For instance the Ogre 3D [38], cross platform 3D rendering engine, uses an approach similar to Listing 3.1. The second approach, compiler arguments, requires the platform macro to be supplied at compile time by the build tool. Although it's the simplest form, it's very error prone. The last option takes advantage of the build system's configuration file feature [22, 27], which holds all the configurations necessary for the build process. Since it's generated on a per target basis, it can easily accommodate platform and compiler specific information.

```
#define PLATFORM_MAC 1
#define PLATFORM_WIN32 2
#define PLATFORM_LINUX 3

#if defined( __WIN32__ ) || defined( _Win32 )
    #define PLATFORM PLATFORM_WIN32
#elif defined( __APPLE_CC__) || defined ( __APPLE__ )
    #define PLATFORM PLATFORM_MAC
#else
    #define PLATFORM PLATFORM_LINUX
#endif
```

Listing 3.1: Example of platform determination through the detection of certain macros.

Before our unified project, an approach similar to Listing 3.1 was adopted. However, due to some specific options that required handling by our build tool, we ended up with mixing it with a configuration file. See Chapter 5 for more details.

3.1.2 File System

Curitiba has a fairly basic support for handling file system operations and provides some abstraction classes and utilities. Most code in these classes handled the UNIX filesystem conventions fairly well, except for some small cases which were absent or behaved incorrectly.

As mentioned in Section 2.4, Mac OS X and iOS resort to bundles to store applications and as such require the use of the CFBundle API to access files contained inside these bundles. Needless to say, Curitiba was not sensibilized towards this methodology and Composer always assumed the resources were at the same level as the executable. This was easily

fixed by adding some platform specific code when needed, although this is not a solution but rather a remedy.

There's a certain lack of standards regarding file operations and resource locations. For instance, throughout the engine most file access calls are made through C's `FILE*` and C++'s `std::fstream`. Two cases were found which used Windows specific file API's. These were later corrected to use C's `FILE*` instead. While this does not hinder the engine's portability, it does not contribute towards it's manageability. For instance, if there were certain standards in place, the integrating of the CFBundle API could have been performed in a more graceful manner.

The next point does not affect Curitiba *per se*, it does however, impact its project files and assets. Windows and Mac OS X come installed with a case-insensitive file system, e.g.: filenames `file.txt`, `File.txt` and `FILE.txt` are considered to be the same. GNU/Linux, on the other hand, is case-sensitive, so all the file names in the previous example are considered different. Most project files and assets in use by Curitiba made no assumptions about the file system's case-sensitivity, thus leading to various situations where resources could not be loaded. Since iOS **is also case-sensitive** some care should be taken in the future when writing project files or preparing assets.

3.1.3 Compiler Standards

Prior to this project, Curitiba had only been compiled with Microsoft Visual Studio Compiler (MSVC). Both GNU/Linux and Mac OS X use GNU C Compiler (GCC) as the default compiler. Note that Mac OS X has steadily been transitioning to the LLVM compiler. It's still possible to use GCC compiler as the default compiler though. Various parts of Curitiba were plagued by semantics errors. This can be attributed to GCC being more standard compliant than the MSVC, due to its frequent updates.

Since there are two supported compilers at this point, compiler specific macros need to be encased as seen in Listing 3.2 to ensure they don't interfere with one another.

```
#if defined(CURITIBA_COMPILER_MSVC)
#pragma warning( disable: 4290)
#endif
```

Listing 3.2: Example of compiler specific code encasement.

3.1.4 Encoding in the GUI

Even though the GUI is cross platform, some serious problems arose due to the text encoding used by the toolkit. wxWidgets [52] can be configured to use ASCII encoding or Unicode [17] encoding. Prior to this dissertation, the wxWidgets [52] used to develop Composer was configured with ASCII encoding. Since Curitiba handles text with C++ standard string class (`std::string`), which is ASCII encoded, everything fitted into place.

The issue arose when wxWidgets [52] was configured to use the Unicode [17] encoding, default encoding on GNU/Linux and Mac OS X. wxWidgets [52] advises users to encase implicit string declarations (e.g.: `"Hello world"`) in the macro `wxT()` (Listing 3.3) to ensure the string is handled correctly with both encodings. This practice was practically non-existent throughout the code in Composer.

wxWidgets [52] also provides a class for textual representation, `wxString`, which should be used to supply text to the toolkit's API's instead of `std::string`. In ASCII encoding, the compiler could perform automatic conversion between `std::string` and `wxString`, as they shared the same underlying base type, `char_t`. With Unicode [17] encoding this is no longer true, because `wxString` now uses `wchar_t`. To correctly convert between these two, we must use the methods listed in Listing 3.3.

```
std::string stdstr;
//Declarations such as this need to be enclosed in wxT()
wxString wxstr2 = wxT("A␣String")
//conversion from std::string to wxString
wxString wxstr = wxString::FromAscii(stdstr.c_str());
//conversion from wxString to std::string
std::string stdstr2 = wxstr.mb_str();
```

Listing 3.3: Conversion between string encodings with wxWidgets [52] 2.8.x.

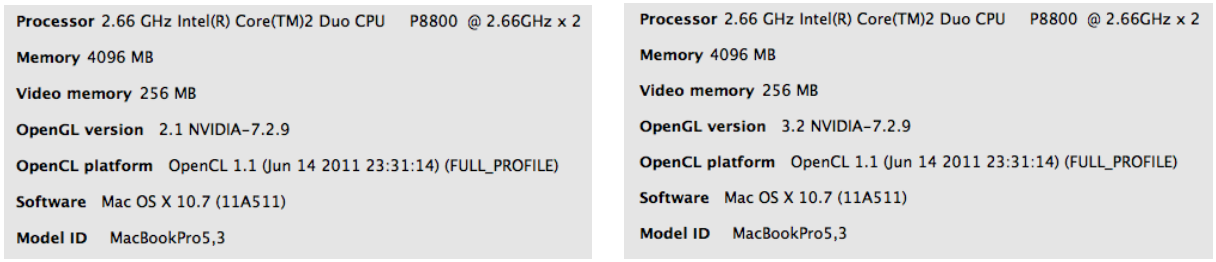
3.2 OpenGL

Curitiba has been making a steady transition from the OpenGL [24] 2.1 fixed function pipeline + shaders to the 3.x Core profile. Since the transition is not yet complete, the engine still mixes 3.x Core features with Compatibility mode features (fixed function). On any system

with a full implementation (Core and Compatibility) of OpenGL [24] 3.x, everything works as expected, but run the same code on a 2.1, 3.x Core or OpenGL ES [25] system and you'll be faced with certain incompatibilities.

3.2.1 OpenGL in Mac OS X

Mac OS X 10.6 has support for OpenGL [24] 2.1 and some 3.x extensions, on supported graphics cards [10]. Version 10.7 introduces a full Core implementation of OpenGL [24] 3.2 (Figure 3.1), which means that 3.2 can't be used in Compatibility mode. This led to the geometry not being displayed when the engine first ran on Mac OS X 10.6 with OpenGL [24] 2.1.



(a) Compatibility profile

(b) Core profile

Figure 3.1: Screenshot of the OpenGL capabilities on a Macbook Pro running Mac OS X 10.7 with an Nvidia 9400 M GPU. Information obtained through the OpenGL Extension Viewer Application.

The solution to this particular issues is quite simple actually: the renderer only needs to be aware of which version of OpenGL [24] is available on the system and select the appropriate rendering mode. Despite its simplicity, it took longer than expected since each renderer specific class imported the OpenGL [24] headers over and over. Instead of repeating the same code in each one of the classes, which is a very error prone method, we created a configuration header file which holds all the definitions for the renderer.

There were in fact more issues regarding the use of the Core profile with Curitiba, although these were all discovered while porting the engine to iOS. At the time Mac OS X 10.7 was not publicly available yet and as such they will be addressed in Section 3.2.2.

3.2.2 From OpenGL to OpenGL ES2

As mentioned previously (Section 2.2), OpenGL ES [25] is a subset of OpenGL [24] and version 2 has a fully programmable pipeline just like OpenGL [24] 3.x. Due to the engine's transitory nature, most problems faced in the porting process can be attributed to the reduced feature set and the lack of fixed function pipeline. Most of the issues described here can also be applied to the OpenGL [24] 3.x Core environment.

Text Display To display text on screen, Curitiba employs the FTGL [21] library. Sadly, the library requires the fixed function API to work properly, hence it isn't compatible. Due to the nature of the library, most textual display was made on demand wherever and whenever required. On a fully programmable pipeline, this approach is no longer feasible. Currently, in the main development branch, a new approach to displaying text will be introduced in the near future. This approach takes into account the programmable pipeline and will therefore work with both OpenGL [24] 3.x Core and OpenGL ES [25].

Since, at the moment, the only platform with support for ES2 is iOS, we devised a temporary solution which uses the iOS API to render the text into a texture with the `UIFont`[2] and `CGContextRef`[3] classes. Since the text can not be displayed using the main rendering queue, due to the required perspective transformations, it's drawn at the end of Curitiba's pipeline in a separate pass.

Materials A fully programmable pipeline can be considered shader-centric, since everything is done through shaders. A common approach is to supply a shader set for each type of material. Most materials used by Curitiba's assets do not come with shaders and the engine's default material configurations lack shader as well. This can easily be overcome by adding shaders which support basic rendering capabilities, so that we, at least, can be certain the assets are being rendered on screen.

Bounding Volumes Another issue plagued by the absence of the fixed function API. Bounding volumes are currently drawn on demand, which is not feasible on the ES 2 pipeline. A

[2] UIFont reference.
[3] CGContextRef reference.

solution to this approach is to convert them into a separate mesh which is only updated when required, Ogre 3D [38] employs a similar practice.

Index Type Size OpenGL ES [25] only supports 16bit and 8bit integer as index types. At first glance this only seems to affect geometry with an index count higher than 65536 (2^{16}). On closer inspection we discovered that Curitiba's physic library, Bullet [44], uses 32bit indexes in its calculations and the default index type in Curitiba is a 16bit integer, when using ES2. Before supplying the indexes to Bullet, we convert the data to 32bit integers. Since Bullet's integration is far from complete and there aren't that many projects which extensively use the library, we are unsure of the impact this might have.

3.3 Composer Cocoa Application

Since wxWidgets [52] did not work on Mac OS X, we wrote a small GUI in Cocoa. Contrary to the wxWidgets [52] front end, this application has only a basic set of options which allow it to load projects and models, and hide or show the profiler and bounding volumes (Figure 3.2).

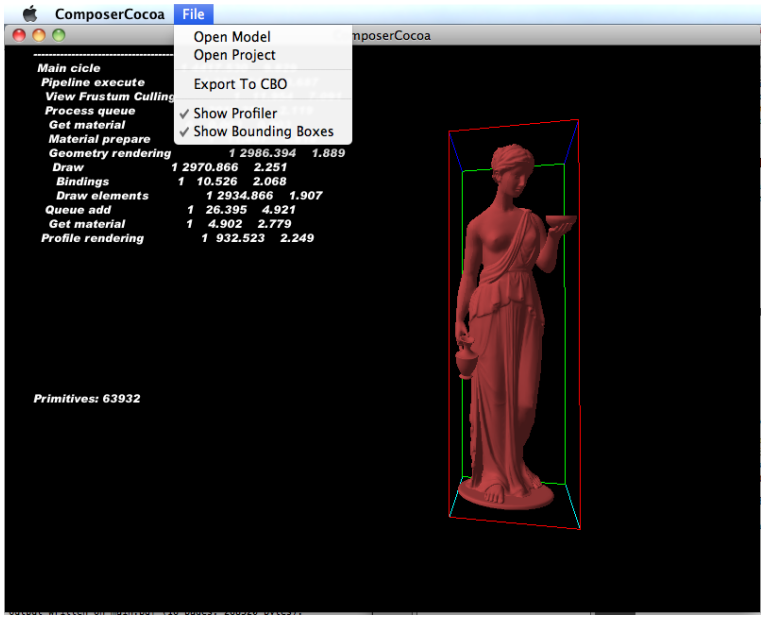


Figure 3.2: Screenshot of the Composer application in Cocoa.

The application was developed with the `NSOpenGLView`[4] class, which encapsulates the required steps to present an OpenGL context. However, Curitiba uses GLEW [48] to ensure all extensions are correctly loaded and this library requires that it imports the OpenGL [24] header files before anything else does. Since the `NSOpenGLView` includes these files before GLEW [48] has a chance to, resulting in errors, we've currently decided to remove the support for GLEW [48] on Mac OS X, for the moment.

[4] NSOpenGLView Class reference.

Chapter 4

iOS Port

With the desktop version of Composer we can easily open a project file from the filesystem, wherever it and its resources may be located. Since there is no description associated with each project, the project is loaded and visualized afterwards. If we then note that this isn't the project we intended to open, we simply repeat the previous process and open a new one. On mobile platforms this approach is no longer feasible since some platforms might restrict access to the filesystem (e.g.: iOS) and constantly visualizing projects will quickly drain battery life as well as waste precious computational resources.

On iOS, each application is sandboxed and only has access to a limited set of pre-defined directories [5], which are usually used for the application's resources and settings. Both features remove any possibility of transferring the files directly into the system's file system. However, since the introduction of document sharing, users can download from and upload documents to their devices from iTunes to a special folder named `Documents`. These documents can later be accessed by the application through the `NSFileSystem` API.

Finally, there's no physical keyboard or mouse available on these platforms. For this reason, we need to develop alternative ways for the user to interact and navigate through the virtual worlds which rely on the touch screen or the available sensors or a combination of both.

4.1 Input Alternatives

Beside the multi-touch screen, our current testing machine, iPad 3G, comes equipped with a GPS, Digital compass and an accelerometer which can be exploited as alternative means of input. Sadly only the iPad 2 and the iPhone 4 come equipped with a gyroscope. The latter is a requirement for using the CoreMotion Framework, which provides us with preprocessed data and can more accurately determine the device's rotary movements [3].

Taking into account that it may be possible to mix both touch and sensory inputs to achieve a certain level of interaction, the application supports one type of touch and sensor input at the same time. Different interaction can be implemented by implementing the `ITouchController` and `ISensorController` protocols, respectively.

4.1.1 Touch Based Input

To take full advantage of the multi-touch screen present on these devices, we divided the screen area into two equally sized sections. The right part of the screen controls the camera's orientation in a similar fashion to what can be expected from moving a mouse on the desktop, while the left side controls the camera's movement through a virtual joystick.

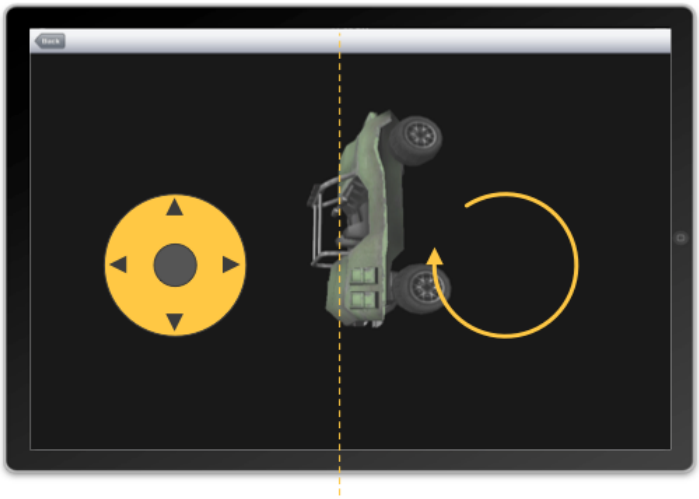


Figure 4.1: Graphical illustration of controls present in the touchscreen input controller. The joystick can be centered anywhere on the left side of the screen.

Contrary to standard joystick controllers, the center of the joystick is always marked by the position where the touch was first initiated. According to direction of the subsequent movement, up, down, left or right, the camera will move forward, backward, strafe left and strafe right, respectively .

Taking into account the possibility of mixing input methods from both the touch screen and sensors, two additional input controllers were developed. These controllers only implement one of the previous features. Thus we have a controller for the camera's movement and another for the camera's orientation. Note that both of them now operate on the entire screen area instead of only a portion.

4.1.2 Accelerometer

As mentioned previously (Section 2.3), the accelerometer indicates the direction gravity is pointing to according to the device's inclination. The direction can be determined based on the values of each of its spacial axes, which vary in the interval $[-1.0, 1.0]$. For instance, positioning the device on a table in landscape with the screen upwards will read $(0.0, 0.0, -1.0)$, with the screen downwards $(0.0, 0.0, 1.0)$ and with screen perpendicular to the table $(-1.0, 0.0, 0.0)$. Performing the same measurements with the device in portrait will yield the same results for the first two measurements and the last will read $(0.0, -1.0, 0.0)$ (if the home button is at the top of the device in this position the value of the y axis will be 1.0). We can make use of this information in two distinct ways: to control the camera's movement or up/down orientation. While both approaches are based on changes in the inclination of the device, they perform slightly different calculations.

	X	Y	Z
No Movement	-	$[-\sigma, \sigma]$	$[-\sigma, \sigma]$
Forward	-	-	$< -\sigma$
Backward	-	-	$> \sigma$
Left	-	$< -\sigma$	-
Right	-	$> \sigma$	-

Table 4.1: Axis values of the accelerometer organized by their resulting camera movement in the application. Note that a value of - indicates that the reading is not used.

Table 4.1 presents the generated movement based on the readings of the accelerometer's axes, mainly Y and Z. The constant σ represents a value in the interval $]0, 0.4]$. It was intro-

duced to delay the detection of the movement, thereby ensuring a more stable navigation. For instance, if σ were 0, the slightest change in inclination would cause the camera to move, making it extremely difficult to keep the camera in one place. The distance covered by the movement is determined through: $\beta \times \alpha$, where β is a distance constant and α the value of the desired accelerometer axis. To bring the camera's movements to a halt, simply put the device in the "No Movement" position (landscape mode with the screen perpendicular to the ground).

To control the camera's up and down orientation, we examine the value of the accelerometer's Z axis only. Since the camera's orientation is implemented in spherical coordinates, we can easily set its orientation through the expression: $\frac{\pi}{2} \times \alpha$, where α is the value of the Z axis. Unlike the readings from Table 4.1, there's no need for a constant such as σ since we wish to establish a direct relation between the axis's value and the camera's orientation.

4.1.3 Digital Compass

The digital compass measures the current heading of the device according to either the magnetic north or the geographic North Pole. The value returned is the angle relative to one of the previous points where 0 means it's pointing north, 90 east, 180 south and so on. After converting these values to radians, we can use them to rotate the camera left or right.

Alone this sensor isn't a great alternative to traditional input methods, since it only allows us to rotate according to one of the reference points. It's preferably used in combinations with other input methods, as we shall see later on. It's possible to perform relative rotations by defining our own reference point, based on the initial reading, and then performing calculations relative to that value. Although this is possible, it's a task better suited to the gyroscope.

4.1.4 Location

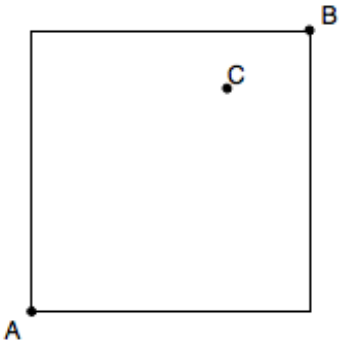
Any iOS device can provide location based services, although only those which have a GPS will provide the most accurate readings. The location service on iOS provides two different updates, a standard update and an update when only a significant change in location has occurred (e.g.: cell towers switch) [6].

Inside buildings the readings do not update very often, this is to be expected since the GPS does not function well inside buildings due to difficulties in communicating with the satellites. Currently, this approach is only intended to be used with the PL3D project [37], where a change in location would cause the camera to move in the resulting direction.

4.2 PL3D Model

Due to delays in the reception of the machine and other difficulties encountered in porting Curitiba, we'll only present a brief theoretical method on how to integrate the previous input methods to provide a virtual "window into the past". First, from the previously covered topics, we'll combine the digital compass and the accelerometer into one controller, thereby ensuring that the camera's orientation is controlled entirely by the device's heading and inclination.

According to the PL3D's developers, the virtual world has been scaled to resemble the actual dimension of the real world. This ensures that a certain distance traveled in the city will cause a movement of equal distance in the virtual world. To correctly determine the user's location in the world we need to construct a two dimensional grid around the virtual world. We require the bottom left and upper right corners of the grid along with their respective latitude and longitude coordinates, altitude and their virtual world's coordinates (x,y,z).



A:	
Latitude:	6.0
Longitude:	-9.0
Altitude:	1.0
x:	-10.0
y:	1.0
z:	-10.0

B:	
Latitude:	7.0
Longitude:	-8.0
Altitude:	3.0
x:	10.0
y:	3.0
z:	10.0

C:	
Latitude:	6.6
Longitude:	-8.3
Altitude:	0.8

Figure 4.2: Example illustration of the points required to determine the user's location inside the virtual world.

To determine the virtual coordinates (x,y,z) of any point in the grid, we need to solve Equation 4.1, Equation 4.2 and Equation 4.3 for x, y and z respectively. For instance, based on

the values present in Figure 4.2, the coordinates of point C would be (8.857,0.8,8.444).

$$x_C = x_A + (|x_B - x_A| \times \frac{longitude_C}{longitude_\sigma})$$
$$longitude_\sigma = \begin{cases} longitude_A & \text{if } |longitude_A| > |longitude_B| \\ longitude_B & \text{if } |longitude_A| <= |longitude_B| \end{cases} \quad (4.1)$$

$$y_C = \frac{altitude_C \times y_A}{altitude_A} \quad (4.2)$$

$$z_C = z_A + (|z_B - z_A| \times \frac{latitude_C}{latitude_\sigma})$$
$$latitude_\sigma = \begin{cases} latitude_A & \text{if } |latitude_A| > |latitude_B| \\ latitude_B & \text{if } |latitude_A| <= |latitude_B| \end{cases} \quad (4.3)$$

Note that these equations require a fairly accurate GPS reading to work, otherwise the user will not get positioned correctly. We suspect these equations will most likely require future adjustments to factor in the GPS's accuracy. If the altitude conversion renders unacceptable results, we could, as an alternative, place the camera at the given location and let the physics engine calculate the correct altitude at which the camera should stand from the ground.

4.3 Project Files

In order to save processing time and battery, it was decided it would be better to present the user with additional information regarding the project before it's loaded. This will prevent the user from loading and opening all available projects until the desired project has been located. Thus each of the projects should contain a description file similar to the one present in Listing 4.1.

```
<project>
  <name> First Project</name>
  <runfile>runtime/project.xml</runfile>
  <image>image.jpg</image>
  <description> First project on iOS. The project loads a model and displays it.</
      description>
  <ios>
    <touchcontroller>multitouch</touchcontroller>
```

```
    <sensorcontroller>accelerometer</sensorcontroller>
  </ios>
</project>
```

Listing 4.1: Example of and iOS project description file.

Since we can only copy one file at a time from within iTunes, each project should be organized in a separate folder. Afterwards, ".bundle" should be appended to the folder name. This will cause iTunes to treat the folder as one file instead of a directory and maintain the documents directory organized and uncluttered. The project file should be named `project.xml` and should be located at the root of the project folder. The remainder of the resources can be organized as the user sees fit.

The project file contains the following fields:

name A string representing the project's name that will be displayed to the user. It should be noted that the project's name serves as an identifier, which means that if there exist two projects with the same name, only one of them will be loaded.

runfile String with the path to the location of the project file readable by Curitiba. The location should always be relative to the project's root folder.

image A string with the path to an image file associated with the project. It's intended to offer the user a preview of the project.

description A string with the description of the project. The text area where it will be displayed is scrollable so we can provide a more detailed description.

ios This section is intended to hold options specific to the iOS platform only. Currently, we only have the following options available:

touchcontroller Contains the name of one of the touch controllers defined in the `ControllerFactory`. Optional, can be omitted.

sensorcontroller Contains the name of one of the sensor controllers defined in the `ControllerFactory`. Optional, can be omitted.

4.4 Project Downloads

Due to the ubiquitous nature of these devices, we thought it to be in our best interests to enable the application to obtain project files through a network connection. Thus, we've developed a simple mechanism to download project files from the internet. First, the application downloads a project list from the url defined by the variable `kDownloadURL` in the file `DownloadList.mm`. The project list must be an XML file similar to the one in Listing 4.2 and, as it name indicates, contain a list of projects with their respective names and download urls.

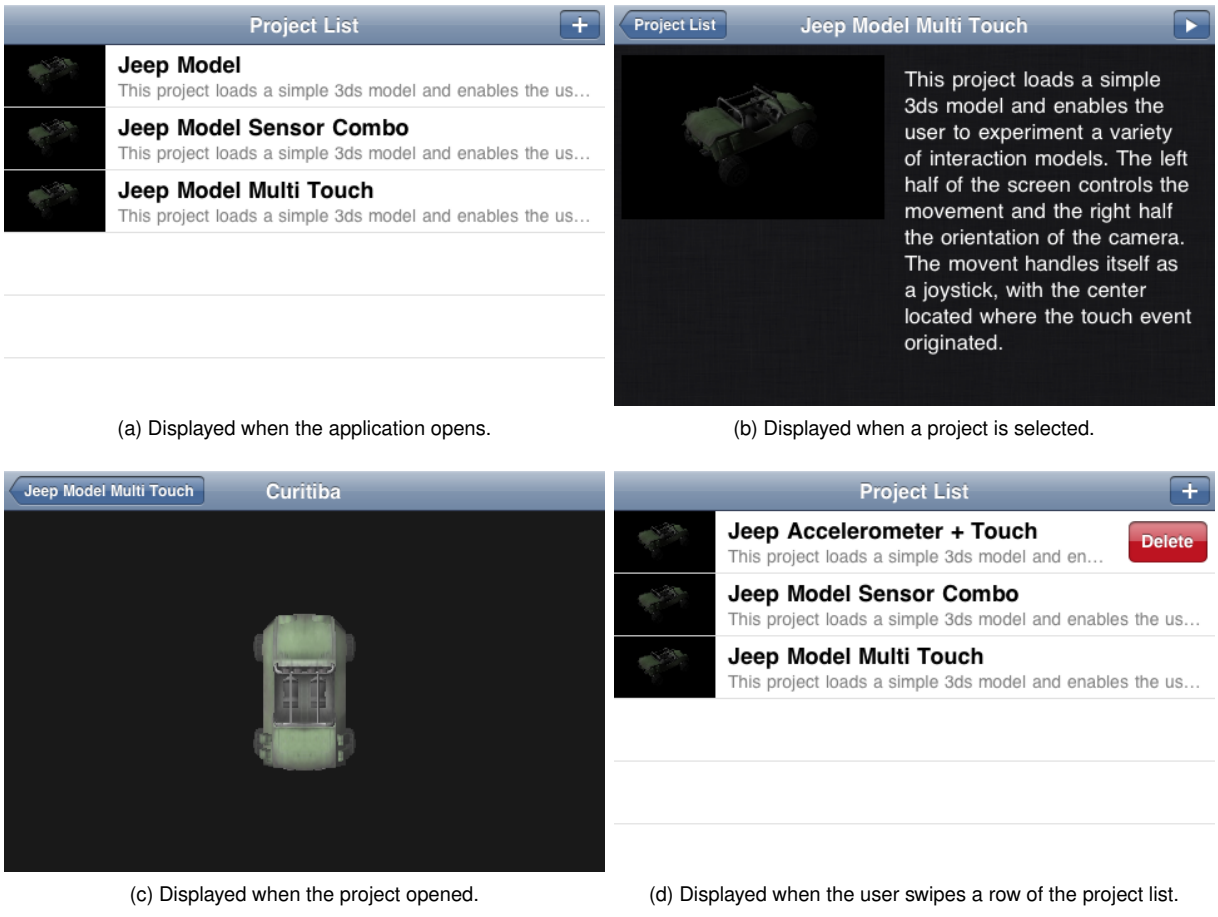
```
<downloads>
    <download>
        <name> Porject Name </name>
        <url>http://...</url>
    </download>
    ...
</downloads>
```

Listing 4.2: Example of the download list file.

Each of the projects should be defined by a `download` tag and the url should point to the location of the project's zip file. We have opted to distribute the files as a compressed archive since it allows us to retrieve the file with one request and save bandwidth (if we distributed the folder instead, we would have to download all of the files individually). It should be noted that once the zip file is downloaded, the unzipping process will overwrite all files in the documents folder which are identical to ones contained within the zip archive.

4.5 Application

Taking the previously detailed conventions into account, we've developed a Universal Application (a single application which will run on both the iPhone and the iPad). The application scans the Documents directory when opened and displays all the valid project files it could locate (Figure 4.3a). When we select a project from the list we are presented with the project's full description and a larger preview (Figure 4.3b). Finally, when we press the triangle shaped button in the top right corner, the project is loaded and displayed to the user (Figure 4.3c). To navigate backwards, simply press the navigation buttons in the top left corner.



(a) Displayed when the application opens.

(b) Displayed when a project is selected.

(c) Displayed when the project opened.

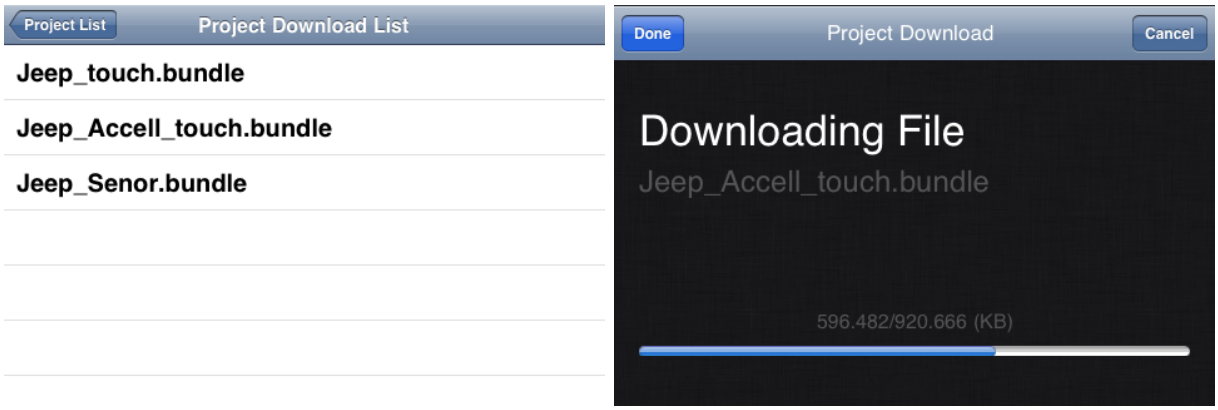
(d) Displayed when the user swipes a row of the project list.

Figure 4.3: Screenshots of the iOS Application running in the iOS Simulator.

Due to the device's mobility an option has been added to remove projects. This will allow the user to free up space without being connected to iTunes. To do so, simply swipe a finger over the row of the desired project to remove from the project list and a red delete button will appear in the same row (Figure 4.3d). Press it and the project will be erased.

To download new project files, the user must press the "+" shaped button present in Figure 4.3a. This will bring up a new view which attempts to obtain the project list and then displays the available projects for download (Figure 4.5a). When one of the projects is clicked on, a new view will appear where the project will be downloaded (Figure 4.4b). To cancel the download, press the "Cancel" in the top right corner. When the download has finished, press "Done" to return.

The iPad interface has the same behavior as the iPhone interfaces depicted in Figure 4.3 and Figure 4.4. The only differences come down to a bigger screen size, and so more details



(a) Displayed when the add project button is clicked.

(b) Displayed when a project is being downloaded.

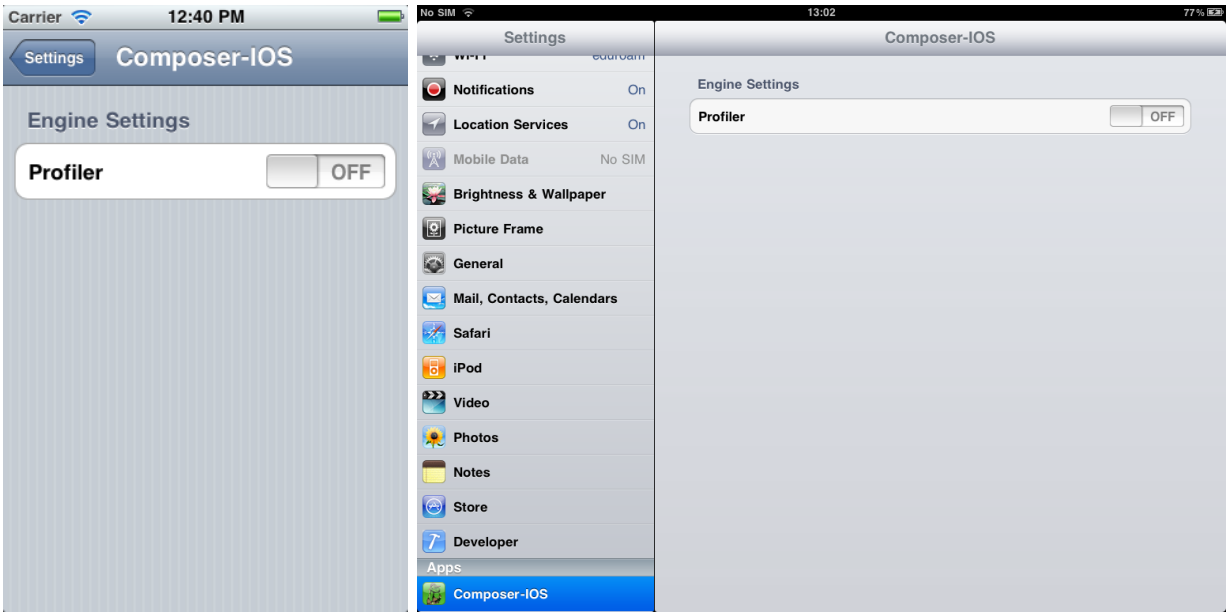
Figure 4.4: Screenshots of the iOS Application running in the iOS Simulator.

can be displayed. Due to the size of the screenshots, the iPad's interfaces have been placed in the Appendix section under Section B.1.

Finally to access Curitiba's settings we decided it would be best to separate the settings from the application and define them in a Settings bundle. These settings can be accessed in the application through the `NSUserDefaults` API. To ensure the application detects changes while it's maintained in the background, we need to subscribe to the `NSNotificationCenter` with the name `NSUserDefaultsDidChangeNotification`. Currently, the available settings only enable or disable the Profiler inside Curitiba (Figure 4.5). For more information on the settings bundle and how to extend it, see the "Implementing Application Preferences" section from [5].

4.6 Notes on Submission to the App Store

The built application tries to conform as much as possible to the application guidelines defined by Apple. Apple requires that the application lists all the required senosors in its Information Property List file so that it can determine which device is able to run the application. Currently, since our current testing machine (iPad 3G) lacks the gyroscope sensor, it has not been listed, otherwise our device wouldn't be able to run the application. If in the near future the when application can be tested on an iOS device with a gyroscope, the key should be listed. This will, however, restrict the application to devices which incorporate all of the



(a) (b)

Figure 4.5: Screen shot of the settings available to Curitiba through the settings bundle on the iPhone (left) and the iPad (right).

required sensors (currently only the iPhone 4, iPhone 4S and the iPad 2 have all the required sensors).

Chapter 5

Project Management

After getting Curitiba up and running on our targeted platforms, we ended up with four distinct project files: two XCode [11] projects for the Mac OS X and iOS platforms, one QtCreator [34] project for GNU/Linux (although there are many more build tools and IDEs available for the GNU/Linux platform, this was the one we were most comfortable with at the time) and a Visual Studio project for Windows. At this point, the source code is shared among all of the projects. Unfortunately, each time we need to add new source files or change the build settings, among other tasks, these need to be manually applied to each of our projects. This does not contribute to the project's management and is a very error prone approach.

A far better approach to this problem is to employ a cross platform build tool. There are a variety of tools available which are cross platform. We, however, require one that will generate native IDE project files for Xcode (since iOS development depends on a great number of features present only in this IDE) and Visual Studio on Windows. In the end, all projects were ported to the CMake [27] build tool. Its core philosophy revolves around managing projects in an compiler independent manner and using the operating system's native build tools [51] (e.g.: Makefiles on GNU/Linux and Visual Studio on Windows).

Why CMake [27]? QtCreator [34] uses its own build tool called qmake [35], which has been designed to be a cross platform replacement for makefiles. Although QtCreator [34] has the ability to generate native Visual Studio projects, it can't generate XCode [11] projects. Autoconf [22] was our next candidate, since it has the ability to generate configuration files through a set of macros which are later converted to shell scripts and could therefore easily

be adapted to each platform. Sadly, it isn't supported on Windows (natively, it does work under Cygwin [45]) and it lacks the ability to generate project files. CMake [27] on the other hand has the ability to generate project files for each platform's most popular IDE and also incorporates several aspects of the Autoconf [22] tool.

CMake [27] commands and terminology are well documented in its wiki [28], nonetheless there's a lack of practical examples available which illustrate how exactly we should use most of CMake [27]'s features. Thus, in the following sections of this chapter we shall thoroughly overview and discuss our CMake [27] project files and by doing so we provide an in-depth tutorial and practical example with the tool.

5.1 CMake Overview

5.1.1 Projects

Creating CMake [27] projects is a fairly easy task, Listing 5.1 depicts how simple the process is. At the root of our source directory we create a file named "CMakeListst.txt". Inside the file we start with defining the minimum version of CMake [27] required to interpret and configure the project through `cmake_minimum_required`, followed by the project's name, `project`. Next, we make sure the compiler treats the source directory as a header include directory through the `include_directories` command. Afterwards, we declare the library and executable files through `add_library` and `add_executable`, respectively. Finally, we link the library with an executable through `target_link_libraries`.

```
cmake_minimum_required(VERSION 2.8)
project(TinyXML)

include_directories("${PROJECT_SOURCE_DIR}")

set(TINYXML_HDR
    tinystr.h
    tinyxml.h
    )
set(TINYXML_SRC
    tinystr.cpp
    tinyxml.cpp
    tinyxmlerror.cpp
```

```
    tinyxmlparser.cpp
    )

add_library(tinyxml ${TINYXML_SRC})

add_executable(tinyxml_test tinyxml_test.cpp)
target_link_libraries(tinyxml_test tinyxml)
```

Listing 5.1: CMake project for the TinyXML library.

The previous examples (Listing 5.1) is but the simplest of projects and barely grasps all of which CMake [27] has to offer. For more information and detailed description of each of the CMake [27] commands please refer to the CMake Wiki and the CMake Documentation.

5.1.2 Options

Sooner or later we might end up with a situation in which the user is required to select an option which will enable or disable certain features in the project. Through the `option` command we can present the user with a binary choice. The result of that choice can then be used throughout the rest of the project to, for instance, adjust the source files of a library and/or executable.

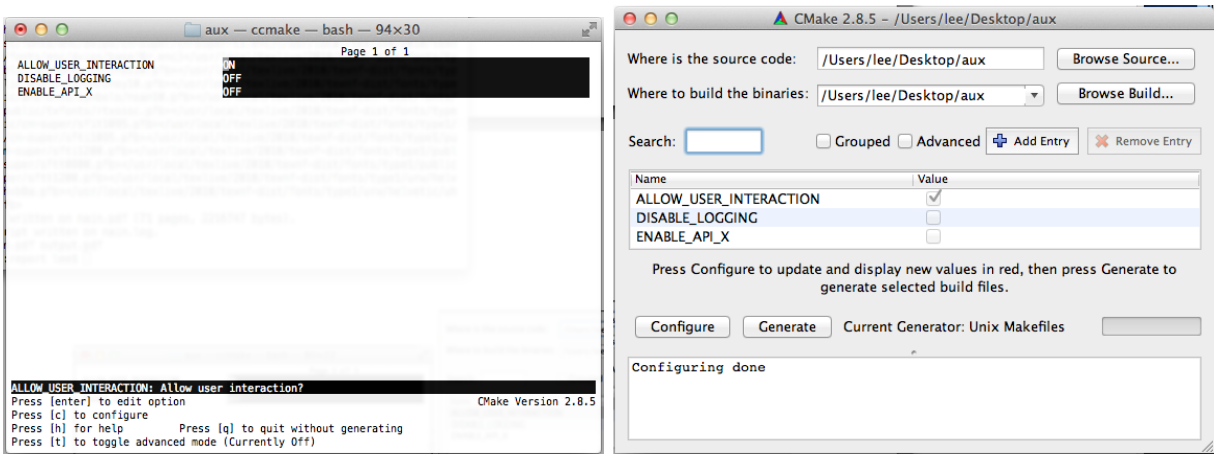
```
option(ENABLE_API_X"Enable API x?")
option(ALLOW_USER_INTERACTION "Allow user interaction?" ON)
option(DISABLE_LOGGING "Disable logging?" OFF)
```

Listing 5.2: Examples of the `option` command in CMake.

As seen in Listing 5.2, the first argument is the variable in which the resulting value will be stored, followed by a string which describes the option and ending with an optional `ON` or `OFF` value to indicate if the option is enabled or disabled, respectively. These options will then be presented to the user as depicted in Figure 5.1. In Figure 5.1a the description is displayed below when the option is selected and in Figure 5.1b it'll be displayed if the user hovers the cursor on top of the name of the variable used by the option.

5.1.3 Finding Dependencies

Curitiba and Composer use a fair amount of external libraries, each one with its own set of headers and library files. Since the location of these files differs across platform and even



(a) CMake Terminal Dialog.

(b) CMake GUI Dialog.

Figure 5.1: Screenshots of CMake project configuration dialogs.

on the same platform, managing these include and library directories manually can be a daunting task. Luckily, CMake [27] provides a mechanism to easily overcome this nuisance: the `find_package` command.

For instance, we require the OpenGL [24] library and header files to compile CURITIBA, so we proceed according to Listing 5.3. The `find_package` command will attempt to locate the CMake module `Find<LIBNAME>.cmake`, in this case, it'll attempt to locate the module `FindOpenGL.cmake`. This module comes bundled with the CMake installation and custom modules can be added by updating the `CMAKE_MODULE_PATH`.

```
FIND_PACKAGE(OpenGL REQUIRED)
#Add include directories
INCLUDE_DIRECTORIES(${OPENGL_INCLUDE_DIRS})
#Append library to library list
SET(LIBS ${LIBS} ${OPENGL_LIBRARIES})
```

Listing 5.3: Example of the `find_package` command to locate the OpenGL headers and libraries.

Each one of these modules contains instructions on how to locate the header and library files. If, for some reason, the required files cannot be found, an error will be triggered and the configuration process will come to a halt, otherwise a set of variables will be filled out according to the following convention:

`<LIBNAME>_FOUD` Will be set to `TRUE` if all the files were found, `FALSE` otherwise.

<LIBNAME>_HEADER_DIR Contains the paths necessary to supply to the compiler so that the headers required by the library and its dependencies can be found.

<LIBNAME>_LIBRARIES Contains all the library files (static or shared) required by the current library.

Besides these standard variables, a module may provide more variables which may hold information of interest to the user. Therefore it's always recommended that these modules be checked when using them.

Even though CMake [27] has a decent set of find modules, sooner or later we'll likely end up writing our own find module. In our case we needed to supply a find module for the FTGL [21] and GLEW [48] libraries. As an example we shall dissect the "FindGLEW.cmake" module (Listing 5.4).

```
# FindPackge GLEW SCRIPT
# GLEW_FOUND - If found
# GLEW_INCLUDE_DIR - GLEW Include Directory
# GLEW_LIBRARY - GLEW Library

find_path(GLEW_INCLUDE_DIR GL/glew.h glew.h
    PATH_SUFFIXES include
    PATHS
    /sw/include
    /opt/local/include
    )

find_library(GLEW_LIBRARY NAMES GLEW glew
    PATH_SUFFIXES lib
    PATHS
    /opt/local
    /sw
    )

include(FindPackageHandleStandardArgs)
FIND_PACKAGE_HANDLE_STANDARD_ARGS(GLEW  DEFAULT_MSG  GLEW_LIBRARY  GLEW_INCLUDE_DIR)

set(GLEW_LIBRARIES ${GLEW_LIBRARY})
mark_as_advanced(
    GLEW_FOUND
    GLEW_INCLUDE_DIR
    GLEW_LIBRARY
    GLEW_LIBRARIES
    )
```

Listing 5.4: Contents of the FindGLEW.cmake script.

To locate the GLEW [48] header file, `glew.h`, we use the `find_path` command and to locate its library, `libGLEW`, we use the `find_library` command. If these files are found, the variables will hold the directory of the header file and the location of the library file, respectively. Finally we invoke the `FIND_PACKAGE_HANDLE_STANDARD_ARGS` which checks if the variables are valid, otherwise the configuration processes is terminated. The `mark_as_advanced` command will hide the variables contained within from the basic configuration menu. Fear not, they can still be viewed in the advanced view mode of the configuration menu.

5.1.4 Configuration Files

Anyone who has ever come into touch with Autoconf [22] tool will be familiar with configuration phase of the build process. This phase generally checks for the availability of certain libraries, header files, compiler flags, among others. The result of this configuration process is then written to a header file. CMake [27] offers various mechanisms to replicate said behavior. We've already seen how to find libraries in the previous section, now we'll see how to create and fill out a configuration file.

For starters, in the desired project file we simply add the command `configure_file` which receives two arguments, the path to the input file and the path to the output file. The input file should end with a *.cmake* extension and will look similar to the example presented in Listing 5.5. Note that the name of output file can be different than the name of the input file. Another thing one should take into account is placing the output configuration file in the binary directory (directory in which the project is being build, accessible through the `PROJECT_BINARY_DIRECTORY` variable), this will enable the user to have various build configurations at the same time while sharing the same source files.

```
#ifndef CONFIG_H
#define CONFIG_H

#cmakedefine OPTION_1
#cmakedefine OPTION_2

#define VAR_1 @CMAKE_VAR_1@
#define VAR_2 @CMAKE_VAR_2@

#endif
```

Listing 5.5: Example of a configuration file used as input file with CMake.

Now suppose the values of the previously listed variables (Listing 5.5) are defined according to Listing 5.6. When the output configuration file is generated, it will look like Listing 5.7. Through observation we can conclude that a variable defined through `#cmakedefine` will only be defined in the output file if its CMake equivalent has the value `ON`, while variables enclosed in "@" will acquire the value defined for that specific CMake variable.

```
OPTION(OPTION_1 "Option 1 description" ON)
OPTION(OPTION_2 "Option 2 description" OFF)

SET(CMAKE_VAR_1 1)
SET(CMAKE_VAR_2 10)
```

Listing 5.6: Example of CMake variables for the configuration files in Listing Listing 5.5.

```
#ifndef CONFIG_H
#define CONFIG_H

#define OPTION_1
//#undef OPTION_2

#define VAR_1 1
#define VAR_2 10

#endif
```

Listing 5.7: Example of the output configuration file based on the variables defined in Listing 5.6.

5.2 Unified Project

5.2.1 A quick note on inter-project dependencies

Most of Curitiba's dependencies can easily be installed on various platforms through install scripts, pre-built binaries or packages, or by building them. Finding a version of those libraries for iOS can be rather tricky. For starters, most libraries are deployed using the Autoconf [22] build tool or Make files for Unix based systems and Visual Studio projects for Windows. The Autoconf [22] tool can be adapted to use the iOS SDK build tools instead of the Mac OS X SDK. Converting Make files is not impossible, but it is a little more difficult and time consuming. Finally, projects configured with CMake [27] are the easiest to port, we just

replace the CMAKE_OSX_SYSROOT (explained later on) with the respective iOS SDK directory and we're done.

Table 5.1 illustrates the dependencies between Curitiba and each of its dependencies. As we can see, the minimum set of dependencies required to built Curitiba on all of our supported platforms are: Bullet, IL and TinyXML. Thus we have included these libraries in our project, mainly to ease the building process on the iOS platform. Any of the other supported platforms can take advantage of this and built these libraries if they have not yet been installed.

Depends On	jpeg	zlib	png	IL	TinyXml	Bullet	Freetype	FTGL
png		•						
IL	•	•	•					
Curitiba				•	•	•	•	•
Curitiba (iOS)				•	•	•		

Table 5.1: Dependencies between Curitiba and the libraries it depends on.

5.2.2 Main Project File

The main project file sets up the environment variables and organizes our project according to the platform we're building for. First, we start by defining the required CMake [27] version and the name of our project (Listing 5.8). Next we make sure our own CMake [27] modules are available by adding the directory in which they reside to the CMAKE_MODULE_PATH.

```
cmake_minimum_required(VERSION 2.8.4)
project(CURITIBA_CMAKE)

set(CMAKE_MODULE_PATH
    "${CURITIBA_CMAKE_SOURCE_DIR}/CMake"
)
```

Listing 5.8: Excerpt from the main CMake project file.

GNU/Linux

When compiling on the GNU/Linux platform we initialize the variable CURRENT_PLATFORM with a custom name (the variable's usage is explained in Section 5.2.2) and we enable the

variable BUILD_PLATFORM_UNIX, which is used throughout the project to identify our current platform as GNU/Linux. We could as matter of a fact use UNIX to check wether we are developing on GNU/Linux, although this variable is also active on Mac OS X.

```
if(UNIX AND NOT APPLE)
    set(BUILD_PLATFORM_UNIX TRUE CACHE INTERNAL "Build Projects for the GNU/LINUX platform
        ")
    set(CURRENT_PLATFORM "unix")
```

Listing 5.9: Excerpt from the main CMake project file.

Mac OS X

Once again we set the values for CURRENT_PLATFORM and activate BUILD_PLATFORM_MAC to reflect the current built platform. We also enable an option to to build the project for the iOS platform, since this can only be done on Mac OS X and there's no standard mechanism in CMake [27] to perform this operation otherwise. Finally we configure the project to be compiled in 32bits by forcing the value of CMAKE_OSX_ARCHITECTURES to i386, due to the default architecture in Mac OS X 10.6 and later being 64bit.

```
elseif(UNIX AND APPLE AND NOT BUILD_PLATFORM_IOS)
    set(BUILD_PLATFORM_MAC TRUE CACHE INTERNAL "Build Projects for the Mac OS X platform")
    option(BUILD_PLATFORM_IOS "If active will build projects for the iOS platform" OFF)
    set(CURRENT_PLATFORM "osx")
    set(CMAKE_OSX_ARCHITECTURES i386 CACHE STRING "Supported build architecures" FORCE)
```

Listing 5.10: Excerpt from the main CMake project file.

iOS

First, we need to ensure that we are generating XCode [11] projects, since most of the functionalities for the iOS environment are only available through XCode [11]. Next we need to force all libraries to be built as archives, since it's the only method allowed to link libraries on iOS[1].

[1] See Apple's App Submission guidelines for more details.

A standard XCode [11] project has the ability to automatically link the correct version of required frameworks when building for the simulator or a device. Since we need to manually specify the frameworks to link with, we need to create separate configurations for the device and the simulator. Therefore, we introduce the option `BUILD_PLATFORM_IOS_DEVICE`. When enabled, the project will be compiled for the device, otherwise for the simulator. The values of `IOS_DEVROOT` and `IOS_SDKROOT` are updated to reflect the path to the current development root and SDK root, respectively. These are later used to update the `CMAKE_OSX_SYSROOT` with the current SDK (Simulator or Device).

```
elseif(UNIX AND APPLE AND BUILD_PLATFORM_IOS)
    unset(BUILD_PLATFORM_MAC CACHE)
    if(NOT CMAKE_GENERATOR STREQUAL "Xcode")
        message(SEND_ERROR "Composer-iOS can only be built with Xcode")
    endif()
    option(BUILD_PLATFORM_IOS_DEVICE "If True will build the project for the devic
        otherwise the project will be build for the simulator." ON)
    set(BUILD_SHARED_LIBS OFF CACHE STRING "iOS only supports static linking" FORCE)
    set(CURRENT_PLATFORM "ios")
    set (IOS_SDKVER "4.3" CACHE STRING "iOS SDK version")
    #iOS Vars
    set (IOSDEV_DEVROOT "/Developer/Platforms/iPhoneOS.platform/Developer" CACHE INTERNAL
        "iOS Development Root")
    set (IOSDEV_SDKROOT "${IOSDEV_DEVROOT}/SDKs/iPhoneOS${IOS_SDKVER}.sdk" CACHE STRING "
        iOS SDK Root" FORCE)
    #iOS Simulator Vars
    set (IOSSIM_DEVROOT "/Developer/Platforms/iPhoneSimulator.platform/Developer" CACHE
        INTERNAL "iOS Simulator Developer Root")
    set (IOSSIM_SDKROOT "${IOSSIM_DEVROOT}/SDKs/iPhoneSimulator${IOS_SDKVER}.sdk" CACHE
        INTERNAL "iOS Simulator SDK Root")
    #Set current SDK VALUES
    if(BUILD_PLATFORM_IOS_DEVICE)
        set (CMAKE_OSX_ARCHITECTURES "armv7" CACHE STRING "Supported build architecures"
            FORCE )
        set(IOS_DEVROOT ${IOSDEV_DEVROOT})
        set(IOS_SDKROOT ${IOSDEV_SDKROOT})
    else()
        set (CMAKE_OSX_ARCHITECTURES "i386" CACHE STRING "Supported build architecures"
            FORCE )
        set(IOS_DEVROOT ${IOSSIM_DEVROOT})
        set(IOS_SDKROOT ${IOSSIM_SDKROOT})
    endif()
    #Compiler settings
    set (CMAKE_OSX_SYSROOT "${IOS_SDKROOT}" CACHE STRING "The product will be built
        against the headers and libraries located inside the indicated SDK." FORCE)
```

Listing 5.11: Excerpt from the main CMake project file.

Windows

Besides the usual variable definitions, `CURRENT_PLATFORM` and `BUILD_PLATFORM_WIN32`, we need one additional step. On Windows we have a pre-built version of wxWidgets [52] available. Due to its location it cannot be found by the `find_package` command. So we first check if wxWidgets [52] is present on the system. If not, we set an environment variable which will allow the script to locate our pre-built version later on.

```
elseif(WIN32)
    set(CURRENT_PLATFORM "win32")
    set(BUILD_PLATFORM_WIN32 TRUE CACHE INTERNAL "Build Projects for the Windows platform
        ")
    find_package(wxWidgets QUIET)
    if(NOT wxWidgets_FOUND)
        #Set environment variable so that find_package can locate wxWidgets
        #in the dependency directory
        set(ENV{WXWIN} "${PROJECT_SOURCE_DIR}/DependAux/${CURRENT_PLATFORM}/wxWidgets
            -2.8.12")
    endif()
```

Listing 5.12: Excerpt from the main CMake project file.

Auxiliary Libraries

In some cases it might prove more useful to supply pre-built version and header files for a certain library. To include these files into the search paths of the $find\_package$ command, we add the header, library and binary paths to the `CMAKE_INCLUDE_PATH`, `CMAKE_LIBRARY_PATH` and `CMAKE_PROGRAM_PATH`, respectively (Listing 5.13). To further ease the processes, the project automatically set these paths to point to the $DependAux$ folder. This folder abides by the following conventions:

- Each platform is separated according to the name given to `CURRENT_PLATFORM`: $win32$ for Windows, ios for iOS, osx for Mac OS X and $unix$ for GNU/Linux;
- Header files must be placed inside a folder named $include$;
- Library files must be placed inside a folder named lib;
- Binary files must be placed inside a folder named bin.

```
set(CMAKE_INCLUDE_PATH "${PROJECT_SOURCE_DIR}/DependAux/${CURRENT_PLATFORM};${
    PROJECT_SOURCE_DIR}/DependAux/${CURRENT_PLATFORM}/include"
    CACHE STRING "Auxiliary Include path for dependencies")
set(CMAKE_LIBRARY_PATH "${PROJECT_SOURCE_DIR}/DependAux/${CURRENT_PLATFORM}/lib"
    CACHE STRING "Auxiliary Library path for dependencies")
set(CMAKE_PROGRAM_PATH "${PROJECT_SOURCE_DIR}/DependAux/${CURRENT_PLATFORM}/bin"
    CACHE STRING "Auxiliary binary path for dependencies")
```

Listing 5.13: Excerpt from the main CMake project file.

Dependencies & Composer

As we mentioned previously, some dependencies are included with the project and these will be automatically activated if they cannot be found. The user is also presented with an option which will allow him to skip the build of the front end application for Curitiba, Composer. If active, the platform specific project will be added to the build list.

```
#DEPENDENCY PROJECTS
add_subdirectory("${CMAKE_SOURCE_DIR}/Depend")

#CURITIBA LIBRARY
add_subdirectory("${CMAKE_SOURCE_DIR}/Curitiba")

#COMPOSER
option(BUILD_COMPOSER "Build the front end application for Curitiba" ON)
if(BUILD_COMPOSER)
    if(APPLE AND NOT BUILD_PLATFORM_IOS)
        add_subdirectory("${CMAKE_SOURCE_DIR}/Composer-Cocoa")
    elseif(APPLE AND BUILD_PLATFORM_IOS)
        add_subdirectory("${CMAKE_SOURCE_DIR}/Composer-IOS")
    elseif(NOT APPLE)
        add_subdirectory("${CMAKE_SOURCE_DIR}/Composer")
    endif()
endif()
```

Listing 5.14: Excerpt from the main CMake project file.

5.2.3 Curitiba

This project file configures and prepares the Curitiba library. First we start by defining the include directories for the project and where the configuration file should be placed (Listing 5.15). We place the configuration file in the project build directory (defined by the

PROJECT_BINARY_DIR variable) to allow several build configurations at once, provided they are kept in separate directories. We also need to add it to the include directory since it is located outside of the source directory.

```
include_directories("${PROJECT_SOURCE_DIR}/src")
include_directories("${PROJECT_BINARY_DIR}/include")

#configuration file
configure_file(
  ${PROJECT_SOURCE_DIR}/config.h.cmake
  ${PROJECT_BINARY_DIR}/include/curitiba/config.h
)
```

Listing 5.15: Excerpt from the Curitiba CMake project file.

Configuration Options

The following two excerpts(Listing 5.16 and Listing 5.17) represent configuration options for Curitiba, which will then be passed to the *config.h* file after the project is generated by CMake [27] (the CMake input file can be viewed in the Appendix Section A.1). In terms of platform specific configuration we disable the CURITIBA_WX_PRESENT option for Mac OS X and iOS, since wxWidgets [52] is not supported on these platforms.

```
option(CURITIBA_WX_PRESENT "Include support for wxWidgets in Curitiba" ON)

if(BUILD_PLATFORM_MAC)
    set(CURITIBA_WX_PRESENT OFF CACHE STRING "Currently wxWidgets does not work on Mac OS
        X" FORCE)
elseif(BUILD_PLATFORM_IOS)
    set(CURITIBA_WX_PRESENT OFF CACHE STRING "wxWidgets not supported on iOS" FORCE)
endif()
```

Listing 5.16: Excerpt from the Curitiba CMake project file.

Next we take care of the renderer settings. Curitiba supports a wide range of options regarding the OpenGL [24] renderer due to certain platforms limitation. The configuration is also used to enable the OpenGL ES [25] renderer on iOS and to disable it on the remaining platforms (Listing 5.17). Each of the CURITIBA_RENDERER_* option's descriptions are self-explanatory and we've only taken some extra care to ensure that the selected combinations are valid.

```
option(CURITIBA_RENDERER_OGLES2 "Build the OpenGL ES 2 Renderer" OFF)
option(CURITIBA_RENDERER_OGL "Build the OpenGL 2.1 Renderer" ON)
option(CURITIBA_RENDERER_OGL_GLEW "Enable the GLEW library" ON)
option(CURITIBA_RENDERER_OGL_3 "Enable OpenGL 3.x support" OFF)
option(CURITIBA_RENDERER_OGL_3_CORE "Enable OpenGL 3.x core mode" OFF)

if(BUILD_PLATFORM_IOS)
    set(CURITIBA_RENDERER_OGLES2 ON
        CACHE STRING "IOS only supports the OpenGL ES 2 renderer" FORCE)
    set(CURITIBA_RENDERER_OGL OFF
        CACHE STRING "IOS only supports the OpenGL ES 2 renderer" FORCE)
    set(CURITIBA_RENDERER_OGL_GLEW OFF
        CACHE STRING "IOS does not support GLEW" FORCE)
    set(CURITIBA_RENDERER_OGL_3 OFF
            CACHE STRING "IOS only supports the OpenGL ES 2 renderer" FORCE)
    set(CURITIBA_RENDERER_OGL_3_CORE OFF
            CACHE STRING "IOS only supports the OpenGL ES 2 renderer" FORCE)
else()
    set(CURITIBA_RENDERER_OGLES2 OFF
        CACHE STRING "IOS only supports the OpenGL ES 2 renderer" FORCE)
endif()

if(BUILD_PLATFORM_MAC)
    set(CURITIBA_RENDERER_OGL ON
        CACHE STRING "Mac OS X only supports the OpenGL renderer" FORCE)
    set(CURITIBA_RENDERER_OGL_GLEW OFF
        CACHE STRING "At the moment the Composer app does not support GLEW" FORCE)
endif()

#Core profile settings
if(CURITIBA_RENDERER_OGL_3_CORE)
    set(CURITIBA_RENDERER_OGL_GLEW OFF
        CACHE STRING "OpenGL 3 Core profile is not supported by GLEW." FORCE)
    set(CURITIBA_RENDERER_OGL_3 ON
        CACHE STRING "OpenGL 3 Core profile requires OpenGL 3 support." FORCE)
endif()
```

Listing 5.17: Excerpt from the Curitiba CMake project file.

Source Files

All source files must be declared in the *CMakeLists.txt* file in the *src* directory of the CURITIBA library folder. The source files are divided into groups to which the `_INC` and `_SRC` suffixes are appended to indicate header our source files, respectively. This separation allows for an easy composition of the necessary source files required by the library and variations due to the available options.

Currently the are five source groups available:

CURITIBA_CORE Source files which represent the core of the Curitiba library without any platform or renderer specific dependencies.

CURITIBA_MAC Source files which should be included when building on the Mac OS X platform.

CURITIBA_IOS Source files which should be included when building on the iOS platform.

CURITIBA_RENDERER_OGL Source files which should be included when the OpenGL [24] renderer is active.

CURITIBA_RENDERER_OGLES2 Source files which should be included when the OpenGL ES [25] renderer is active.

Finally, the resulting header and sources lists are stored in the `CURITIBA_HEADERS` and `CURITIBA_SOURCES` variables, respectively.

Library Dependencies

All dependencies are located with the $find\_package$ command and all the include directories and libraries are added to the project settings. To facilitate the inclusion of Curitiba in other projects (Composer), we export three CMake [27] variables `CURITIBA_INCLUDE_DIRS`, `CURITIBA_LIBRARY` and `CURITIBA_LIBRARIES`, which hold all the required include directories, the Curitiba library and all the libraries necessary to link with it, respectively. This way other projects simply add `CURITIBA_INCLUDE_DIRS` to their include directories and link with the libraries present in `CURITIBA_LIBRARIES` and they are set to go.

There is one particular issue we must pay attention to, the $find\_package$ command will attempt to locate frameworks first on Mac OS X, but it won't be able to locate the frameworks when building for iOS since the iOS SDK is not in the default list of search directories. Therefore, when building for iOS we need to restrain the search directories to the iOS SDK only (Listing 5.18), because some frameworks on iOS are also available for Mac OS X.

```
if(APPLE)
    if(BUILD_PLATFORM_IOS)
        find_library(FOUNDATION_LIB Foundation PATHS
            "${IOS_SDKROOT}/System/Library/Frameworks/" NO_DEFAULT_PATH )
    else()
        find_library(FOUNDATION_LIB Foundation )
    endif()
    if(${FOUNDATION_LIB} STREQUAL "FOUNDATION_LIB-NOTFOUND")
        message(SEND_ERROR "Could not locate Foundation Framework.")
    endif()
    set(CURITIBA_LIBRARIES ${CURITIBA_LIBRARIES} ${FOUNDATION_LIB}  CACHE INTERNAL "
        Curitiba Libraries")
endif()
```

Listing 5.18: Excerpt from the Curitiba CMake project file.

Framework

The most common library distribution method on Mac OS X is through frameworks and the libraries contained in these frameworks are shared libraries. iOS also supports frameworks, although due to the previously discussed limitation (Section 5.2.2) all libraries need to be statically compiled(archives). CMake [27] provides means to automatically create frameworks, but these can only be applied to shared libraries. To overcome this limitation, a small python script was written to create frameworks (Appendix Section A.2). The script is then invoked after the library has been built through a post build command (Listing 5.19).

```
if(BUILD_FRAMEWORKS)
    #Locate the output library and
    get_target_property(LIBNAME curitiba LOCATION)

    #Build framework
    add_custom_command(TARGET curitiba POST_BUILD
    COMMAND python "${CMAKE_SOURCE_DIR}/Scripts/mkframework.py"
        "curitiba" "${CURITIBA_VERSION}" "${LIBNAME}"
        "${PROJECT_SOURCE_DIR}/src" "${CURITIBA_HEADERS}"
    COMMENT "Creating curitiba framework..." VERBATIM)
endif()
```

Listing 5.19: Excerpt from the Curitiba CMake project file.

5.2.4 Composer Cocoa

Besides the usual settings for each project, include directories and libraries, we need to take into account three features which are specific to the Mac OS X platform: Objective-C programming language, application bundles and interface files.

Objective-C

Mac OS X uses Objective-C as its main programming language for the user interface and general programming. Objective-C [9] is a superset built around C and as such allows us to mix legacy C code with Objective-C code (`.m` extension). There is also a variant called Objective-C++, which allows us to mix C++ code and Objective-C code (`.mm` extension). By default, CMake [27] does not recognize Objective-C/C++ files. Since the GCC compiler is used to compile Objective-C/C++ files on Mac OS X with the flag *-x objective-c++*, we simply need to enable this flag on all our source files ending in `.m` or `.mm`. This is achieved through the `set_source_files_properties` command (Listing 5.20).

```
foreach(source ${composer_cocoa_SRC})
    if(${source} MATCHES "mm$|m$")
    set_source_files_properties(${source}
        PROPERTIES COMPILE_FLAGS "-x objective-c++")
    endif()
endforeach()
```

Listing 5.20: Excerpt from the Composer Cocoa CMake project file.

Application Bundles

As mentioned previously, , the applications on Mac OS X are all distributed in self-contained bundles. CMake [27] has built-in features which allow us to create application bundles. When we declare our executable we need to inform CMake [27] that it'll be a bundle with the flag `MACOSX_BUNDLE` (Listing 5.21).

```
add_executable(composer_cocoa
    MACOSX_BUNDLE
    ${composer_cocoa_HDR}
    ${composer_cocoa_SRC}
    ${xib_list}
)
```

Listing 5.21: Excerpt from the Composer Cocoa CMake project file.

Each bundle comes with an *Info.plist* file. To generate this file we fill out the bundle properties as listed in Listing 5.22. Note that the `MACOSX_BUNDLE_NSMAIN_NIB_FILE` and `MACOSX_BUNDLE_NSPRI` need to be set according to the classes present in our application.

```
set(MACOSX_BUNDLE_INFO_STRING "${PROJECT_NAME}")
set(MACOSX_BUNDLE_GUI_IDENTIFIER "pt.uminho.di")
set(MACOSX_BUNDLE_LONG_VERSION_STRING "${PROJECT_NAME} Version 1.0.0")
set(MACOSX_BUNDLE_BUNDLE_NAME ""${PROJECT_NAME}"")
set(MACOSX_BUNDLE_SHORT_VERSION_STRING "1.0.0")
set(MACOSX_BUNDLE_BUNDLE_VERSION "1.0.0")
set(MACOSX_BUNDLE_COPYRIGHT "Copyright 2011. All Rights Reserved.")
set(MACOSX_BUNDLE_EXECUTABLE_NAME "composer_cocoa")

set(MACOSX_BUNDLE_NSMAIN_NIB_FILE "MainMenu")
set(MACOSX_BUNDLE_NSPRINCIPAL_CLASS "NSApplication")

set_target_properties(composer_cocoa PROPERTIES MACOSX_BUNDLE_INFO_PLIST ${
    PROJECT_SOURCE_DIR}/ComposerCocoa-Info.plist)
```

Listing 5.22: Excerpt from the Composer Cocoa CMake project file.

Finally, the resources need to be added manually to the bundle's resource folder. We do this by invoking a copy command for each resource that will be executed once the the application has been built and the bundle created (Listing Listing 5.23). The `\${CONFIGURATION}` part in the destination directory ensures that we copy our resource to the correct build configuration folder when using standard makefiles or XCode [11]. If we are using XCode [11] generated project, `\${CONFIGURATION}` will take the value of either *Release* or *Debug*, otherwise it will be automatically omitted.

```
foreach(res ${composer_cocoa_RES})
    add_custom_command(TARGET T composer_cocoa POST_BUILD
    COMMAND cp -r ${PROJECT_SOURCE_DIR}/${res}
        ${PROJECT_BINARY_DIR}/\${CONFIGURATION}/composer_cocoa.app/Contents/Resources/
    COMMENT "Copying ${PROJECT_SOURCE_DIR}/${res} to Resource folder")
```

```
endforeach()
```

Listing 5.23: Excerpt from the Composer Cocoa CMake project file.

Interface Files

On Mac OS X the user interface can either be written manually or be saved as `.xib` file through the Interface Builder application. CMake [27] does not have any means to deal with these files automatically, thus we need to manually compile the interface files and then add them to the resource folder of our application bundle.

First we need to locate the `ibtool` program, which compiles the `.xib` files into binary `.nib` files. We locate the program with the `find_program` command and then add a rule so that each `.xib` file is compiled and placed in the resource folder after the application is built (Listing 5.24).

```
find_program(IBTOOL ibtool HINTS "/usr/bin" "${OSX_DEVELOPER_ROOT}/usr/bin")
if (${IBTOOL} STREQUAL "IBTOOL-NOTFOUND")
  message(SEND_ERROR "ibtool can not be found and is needed to compile the .xib files. It
      should have been installed with
                        the Apple developer tools. The default system paths were searched
                            in addition to ${OSX_DEVELOPER_ROOT}/usr/bin")
endif()

#compile xib files
foreach( xib ${composer_cocoa_XIB})
    add_custom_command(TARGET composer_cocoa POST_BUILD
    COMMAND ${IBTOOL} --errors --warnings --notices --output-format human-readable-text --
        compile ${PROJECT_BINARY_DIR}/\${CONFIGURATION}/composer_cocoa.app/Contents/
        Resources/${xib}.nib ${PROJECT_SOURCE_DIR}/${xib}.xib COMMENT "Compiling ${
        PROJECT_SOURCE_DIR}/${xib}.xib")
endforeach()
```

Listing 5.24: Excerpt from the Composer Cocoa CMake project file.

5.2.5 Composer iOS

This port of Composer closely resembles the Composer Cocoa project file (Section 5.2.4), they differ only in the Information Property List (*Info.plist*) and framework locations. Despite

the support CMake [27] has for the Mac OS X Information Property List, it lacks the necessary fields required by the iOS operating system. Hence, we need to supply these files for both device and simulator builds and copy them into the application bundle.

Since the iOS SDK is not located in the standard include and library directories, we need to instruct the compiler to search these directories instead, just as in Section 5.2.3. We force `find_package` command to only search the SDK directory for the required libraries (Listing 5.25) by supplying a path after the `PATH` argument and ending with the `NO_DEFAULT_PATH` option.

```
find_library(QUARTZ_LIB QuartzCore PATHS
    "${IOS_SDKROOT}/System/Library/Frameworks/" NO_DEFAULT_PATH )
if(${QUARTZ_LIB} STREQUAL "QUARTZ_LIB-NOTFOUND")
    message(SEND_ERROR "Could not locate QuartzCore Framework.")
endif()
```

Listing 5.25: Set the compiler search paths and force the ***find_package*** *command to only search the SDK directory.*

5.2.6 Composer wxWidgets

This is the standard version of the Composer program. It's developed with the wxWidgets [52] GUI toolkit and currently is only supported on GNU/Linux and Windows. The project follows practically the same guidelines as all the previous ones. However, some special care needs to be taken on the Windows platform.

On Windows we need to explicitly define that our application is a window based application by adding the `WIN32` flag when defining the executable (Listing Listing 5.26). If this flag is not present, the application will be treated as a console application and all the headers and libraries refraining to Windows window system won't be available.

```
if(WIN32)
add_executable(composer WIN32
    ${COMPOSER_HDR}
    ${COMPOSER_SRC}
    )
    link_directories(${wxWidgets_LIB_DIR})
else()
    add_executable(composer
```

```
    ${COMPOSER_HDR}
    ${COMPOSER_SRC}
    )
endif()
```

Listing 5.26: Excerpt from the main Composer CMake project file.

Due to unknown reasons, on Windows we need to include all the wxWidgets [52] modules instead of just the required ones. This is easily solved by scanning the wxWidgets [52] library folder and linking with all library files present in that directory (Listing Listing 5.27).

```
if(WIN32)
    file(GLOB wxLIBS ${wxWidgets_LIB_DIR}/*.lib)
    target_link_libraries(composer ${wxLIBS})
endif()
```

Listing 5.27: Excerpt from the Composer CMake project file.

As you may have noticed in the previous project files, the resources were copied with the `cp` command. Since these project are only intended to run on the Mac OS X platform, they don't pose any problem. This project, however, is meant for both GNU/Linux and Windows, and only the first has the `cp` command.

To perform the copy process in a cross platform manner, we need to invoke the CMake [27] executable with the flag `-E`, followed by the intended operation. This allows common operations such as copying files, creating directories, among others, to be performed without needing to know the underlying operating system's commands. The commands we're interested in are *copy_directory* and *copy*, which copies a directory and a file, respectively (Listing 5.28).

```
foreach(res ${COMPOSER_RES})
    if(IS_DIRECTORY "${Composer_SOURCE_DIR}/${res}")
        add_custom_command(TARGET composer POST_BUILD
        COMMAND ${CMAKE_COMMAND} -E copy_directory "${Composer_SOURCE_DIR}/${res}"
         "${PROJECT_BINARY_DIR}/${res}" COMMENT "Copying resource ${res}...")
    else()
        add_custom_command(TARGET composer POST_BUILD
        COMMAND ${CMAKE_COMMAND} -E copy "${Composer_SOURCE_DIR}/${res}"
         "${PROJECT_BINARY_DIR}/${res}" COMMENT "Copying resource ${res}...")
    endif()
endforeach()
```

Listing 5.28: Excerpt from the main Composer CMake project file.

5.3 Building the Project

There is more than enough information available on the internet on how to build and use the CMake [27] configuration tools. Nonetheless we'll discuss a few minor details regarding the project's configuration process and installation. First of all, CMake [27] encourages out of source builds. This means our build directory should be located outside the source folder.

For instance, suppose our project folder lies in the following directory: `/var/project`. The recommended build directory in this case would be `/var/project_build`. The name of the build directory is irrelevant and it should only be located at the same level as the source directory. This practice will keep our source directory clean (CMake [27] generates a great quantity of auxiliary files) and will allow the source files to be accessed from within the chosen IDE (if any).

Configuring

When generating project files CMake [27] can either specify the build type or allow the generated project to decide. The former applies to the Eclipse CDT and Unix Makefiles projects, while the latter is applied to Visual Studio and Xcode. Despite the standard Eclipse CDT project being able to generate debug and release builds simultaneously, CMake [27] creates an Eclipse CDT project which uses the CMake [27] makefiles to build the project instead.

To configure our project we can use either the command line utility `ccmake` or the GUI application. There are some flags which should be specified before configuring the project, `CMAKE_BUILD_TYPE`, `CMAKE_INSTALL_PREFIX` and the generator. The first indicates which build type to use and should be either Release or Debug. The second defines a directory prefix where the Curitiba library will be installed. The last defines the project generator to use. If we intend to use either Visual Studio or XCode [11] as IDE we need not worry with the first flag and the second flag can be omitted if we do not wish to install the library on our system.

When using the command line utility, the first two flags are enabled with `-D` and the generator is specified with `-G`. For instance the example, Listing 5.29 instructs the configuration program to build the project in release mode, install the resulting files into the `out` folder present in the current directory and to generate an Eclipse CDT project file.

```
ccmake <path to source directory> -DCMAKE_BUILD_TYPE=Release -DCMAKE_INSTALL_PREFIX=‘pwd‘/
    out -G"Eclipse CDT4 - Unix Makefiles"
```

Listing 5.29: Configuration parameters when using the command line utility.

When using the GUI application (Figure 5.2), we start by defining the location of the source and build directory. Next we add the two flags by clicking the ”Add Entry” button. Once the flags have been added, we click ”Configure” and select our desired build tool.

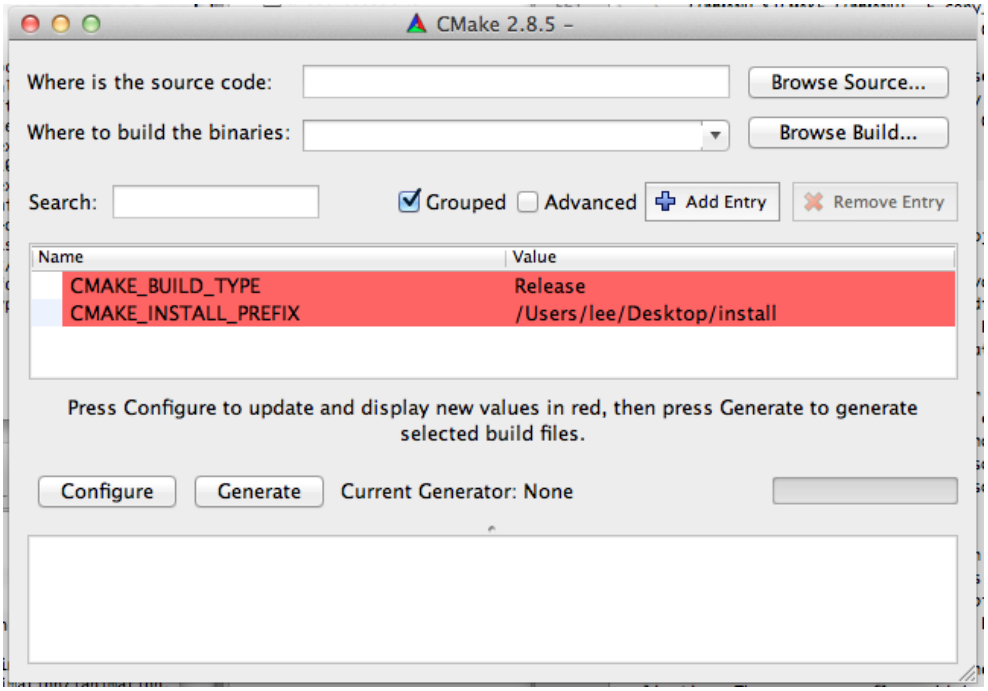


Figure 5.2: Screenshot of the configuration process with the CMake GUI application.

Regardless of the application being used to configure the project, after the first configuration routine has completed, the user should enable or disable all the options deemed necessary and press ”Configure” again. Note that it may be necessary to press ”Configure” for a third or forth time, due to certain options only presenting themselves when others have been enabled or disabled. The configuring process will be completed when there are no more * present in the options list (ccmake) or no more highlighted options in red (CMake GUI application). When done, press the ”Generate” button to generate the project files with the selected generator and build the project.

Note on building with XCode 4

There's a problem with the build settings on Xcode 4 with a project generated through CMake [27] (tested with versions 2.8.4 and 2.8.5). It will always build the generated projects in debug mode **even if the configuration is set to build in release mode inside Xcode**. To build the projects in release mode, open the Terminal application and navigate to the project build directory. Then, execute the following command: `xcodebuild -configuration Release -sdk iphonesimulator4.3` to build a release build for the iOS simulator and `xcodebuild -configuration Release -sdk iphoneos4.3` for the iOS device.

Installation

The installation option will install the Composer executable (if any) and the Curitiba header files and library into the folder bin, $include$ and lib, respectively. The path to these folders is defined by the `CMAKE_INSTALL_PREFIX` variable. If not defined, this variable will have a default path which varies according to the platform on which the project is being built.

There's one important issue we need be aware of. At the current state, the resulting Composer application cannot be distributed as a standalone application when Curitiba and other dependencies are compiled as shared libraries (GNU/Linux and when built with Frameworks on Mac OS X). In other words, the application will attempt to locate the libraries on startup and, failing to do so, will crash. The application doesn't crash when run from the build directory, because CMake [27] defined the paths to the previously built libraries.

Therefore, to distribute the Composer application, Curitiba and its dependencies must be present on the system. When linking statically, these concerns need not apply because the libraries will be directly integrated with the executable.

Chapter 6

Conclusion

6.1 Review

Porting Curitiba was far from easy, then again these things are never that simple. Each platform presented new challenges or broke some existing functionality. Nonetheless, Curitiba now runs on four distinct platforms: iOS, GNU/Linux, Mac OS X and Windows. The Android platform was left out since the Android Native Development Kit (NDK) [23] at the time, r4, did not include C++ Exceptions, RTTI and STL support[1]. Curitiba relies heavily on STL and can't be compiled without it. The lack of Exceptions would require Curitiba's error handling mechanism to be redesigned from the ground up and the adoption of an alternative error handling method. See [20] for more information on this topic.

It wasn't possible to achieve an equally portable version of Composer on GNU/Linux, Mac OS X and Windows due to the wxWidgets [52] limitations on Mac OS X (Chapter 3). This will be solved in the near future as soon as version 2.9 of wxWidgets [52] is declared stable and the codebase is upgraded to this newer version. Due to the availability of only the Core OpenGL [24] 3 profile on Mac OS X, GLEW [48] will remain unsupported on this platform, since GLEW [48] is intended to be used in Compatibility mode only.

The iOS port works, although it should not be considered a complete port of the engine. This can all be attributed to the transition from OpenGL [24] to OpenGL ES [25]. Many sections

[1] See this topic and the **CPLUSPLUS-SUPPORT.HTML** documentation file included in the Android NDK for more information.

of the renderer code are still dependent on fixed function functionality. The main branch of Curitiba is currently making its transition from the fixed function pipeline to the OpenGL [24] Core Profile. Many of the issues we encountered here also affect the main branch and will therefore be fixed in the main branch. Once that happens, the code base must be merged and the ES2 renderer updated. Only then shall we have a complete port of the library on iOS. It's important to note that all the issues affecting the ES2 port have been reported to the main branch and have already been resolved or will be in the near future.

Creating the unified project with CMake [27] took longer than expected. Even though the CMake wiki [28] has some examples and detailed explanations of all the commands and syntax, everything is very fragmented. For simple projects the answers are readily available, but for larger projects such as Curitiba, it can take some time before finding the answer we're looking for. The resulting project file has been thoroughly documented so that anyone can quickly get up to speed with the CMake [27] build system, and can also be used as a practical example or tutorial.

Despite delays in the porting process and the arrival of a test device, we've managed to implement a few feasible replacements for the traditional mouse and keyboard model. Since our current testing device lacked the gyroscope sensor, we could not take advantage of the core motion framework. Nevertheless, we've implemented a temporary replacement with the digital compass and presented an initial approach to the virtual guide for PL3D [37].

6.2 Future Work

In terms of future work, Curitiba can now be ported to the Android platform. Starting with the NDK r5, we now have support for Exceptions and STL. The main challenge regarding this platform is the integration of the build system with the CMake [27] build tool. The NDK uses Makefiles and shell scripts, although with some reverse engineering they can be easily integrated through a CMake Toolchain [29].

Now that Curitiba supports the three major platforms (GNU/Linux, Mac OS X and Windows), another worthy expansion would be a web plugin, similar to the one provided by Unity [46]. This will allow Curitiba to be run directly from the browser, provided the plugin is installed. The plugin is developed through the Nestcape Plugin API [32], which allows the browser to

communicate with native libraries through Javascript. This approach is also used by other popular browser plugins, such as Adobe's Flash.

The PL3D model could be further extend and improved if we were to introduce a model based tracking component, which uses information from 3D models to perform visual tracking [13]. For instance, [39] and [43] use model based tracking to enhance the detection accuracy, which is affected by imprecisions in the GPS readings. We could apply a similar approach in PL3D as means to increase the accuracy of our location in the virtual world. Based on our location and orientation we could take advantage of the engine's spacial partitioning system to exclude models which shouldn't be visible. Next we use the information present in those models and try to identify them, with an appropriate metric, on the captured image. If we have a match and the position in the virtual appears to be wrong, we could automatically adjust it (if at all possible).

6.3 Closing Thoughts

All things considered, we've managed to improve Curitiba's cross platform state and despite partial implementations in certain cases (OpenGL ES [25] on iOS), we've laid out the foundations for an easier and unified development across all of our target platforms.

Bibliography

[1] ANSI. American national standard institute (ansi). Accessed on the 10th of August 2011. 6

[2] Apple. 64 bit guide for carbon developers. Accessed on the 4th of April 2011. 15

[3] Apple. Event handling guide for ios. Accessed on the 21st of August 2011. xi, 10, 26

[4] Apple. Information property list key reference. Accessed on the 26th of September 2011. 11

[5] Apple. ios application programming guide. Accessed on the 11th of August 2011. 12, 25, 34

[6] Apple. Location awareness programming guide. Accessed on the 23th of September 2011. 28

[7] Apple. Mac os x bundle programming guide. Accessed on the 11th of August 2011. 11

[8] Apple. Mac os x framework programming guide. Accessed on the 11th of August 2011. 11, 12

[9] Apple. The objective-c programming language. 53

[10] Apple. Opengl legacy and core capabilities table for mac os x. Accessed on the 13th of August 2011. 20

[11] Apple. Xcode ide. Accessed on the 21st of May 2011. 37, 45, 46, 54, 58

[12] IEEE Standards Association. Portable operating system interface (posix). Accessed on the 10th of August 2011. 7

[13] Ronald Azuma. A survey of augmented reality. *Presence: Teleoperators and Virtual Environments*, 6:355–385, 1997. 63

[14] J. Bishop and N. Horspool. Cross-platform development: Software that lasts. *Computer*, 39(10):26 –35, oct. 2006. 5, 8

[15] Boost. Boost c++ libraries. Accessed on the 21 of June 2011. 8

[16] A Charlesworth. The ascent of smartphone. *Engineering Technology*, 4(3):32, 2009. 1

[17] The Unicode Consortium. Unicode text encoding standard. Accessed on the 17th of March 2011. 19

[18] Michael A. Cusumano and David B. Yoffie. What netscape learned from cross-platform software development. *Commun. ACM*, 42:72–78, October 1999. 5, 8

[19] Bruno Miguel Ferreira de Oliveira. The anatomy of a real-time rendering engine. Master's thesis, University of Minho, 2007. 16

[20] Volman Engineering. Exception handling alternatives in c++, 1999. Accessed on the 19th of August 2011. 61

[21] FTGL. Ftgl - cross-platform open source c++ library which uses freetype2 to simplify rendering fonts in opengl applications. Accessed on the 21st of May 2011. 21, 41

[22] GNU. Autoconf - automatic configuration build tool for the unix platforms. Accessed on the 21st of May 2011. 7, 17, 37, 38, 42, 43

[23] Google. Android native development kit. Accessed on the 19th of August 2011. 61

[24] Khronos Group. Opengl - cross platform 3d rendering api. Accessed on the 10th of March 2011. 8, 19, 20, 21, 23, 40, 49, 51, 61, 62

[25] Khronos Group. Opengl es - cross platform 3d rendering api for embedded systems. Accessed on the 10th of March 2011. 8, 20, 21, 22, 49, 51, 61, 63

[26] J.D. Hol, T.B. Schon, F. Gustafsson, and P.J. Slycke. Sensor fusion for augmented reality. In *Information Fusion, 2006 9th International Conference on*, pages 1 –6, July 2006. 10

[27] Kitware. Cmake - cross platform build tool. Accessed on the 21st of May 2011. 7, 17, 37, 38, 39, 40, 41, 42, 43, 44, 45, 49, 51, 52, 53, 55, 56, 57, 58, 60, 62

[28] Kitware. Cmake wiki. Accessed on the 19th of August 2011. 38, 62

[29] Kitware. Cross compiling with cmake. Accessed on the 19th of August 2011. 62

[30] Feida Lin and Weiguo Ye. Operating system battle in the ecosystem of smartphone industry. *Information Engineering and Electronic Commerce, International Symposium on*, 0:617–621, 2009. 1

[31] Syd Logan. *Cross-platform development in c++: building mac os x, linux, and windows applications*. Addison-Wesley Professional, first edition, 2007. 5, 6, 8

[32] Mozilla. Nestcape plugin api on mozilla wiki. Accessed on the 19th of August 2011. 62

[33] Aaftab Munshi, Dan Ginsburg, and Dave Shreiner. *OpenGL(R) ES 2.0 Programming Guide*. Addison-Wesley Professional, 1 edition, 2008. xi, 9

[34] Nokia. Cross platform c/c++ ide. part of the qt sdk. Accessed on the 21st of May 2011. 37

[35] Nokia. Qmake - cross platform build tool developed by qt team. Accessed on the 21st of May 2011. 7, 37

[36] Nokia. Qt gui toolkit. Accessed on the 21st of May 2011. 7, 8

[37] Department of Informatics of University of Minho and County of Ponte de Lima. Ponte de lima 3d. Accessed on the 11th of August 2011. 3, 29, 62

[38] Ogre3d cross platform rendering engine. Accessed on the 21st of May 2011. 2, 17, 22

[39] G. Reitmayr and T.W. Drummond. Going out: robust model-based tracking for outdoor augmented reality. In *Mixed and Augmented Reality, 2006. ISMAR 2006. IEEE/ACM International Symposium on*, pages 109 –118, October 2006. 10, 63

[40] Jannick P. Roll, Yohan Baillot, and Alexei A. Goon. A survey of tracking technology for virtual environments. 2008. 10

[41] Dave Shreiner and The Khronos OpenGL ARB Working Group. *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Versions 3.0 and 3.1*. Addison-Wesley Professional, 7th edition, 2009. 9

[42] S. Srinivasan, Zhen Fang, R. Iyer, S. Zhang, M. Espig, D. Newell, D. Cermak, Yi Wu, I. Kozintsev, and H. Haussecker. Performance characterization and optimization of mobile augmented reality on handheld platforms. In *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, pages 128 –137, October 2009. 2

[43] Gabriel Takacs, Vijay Chandrasekhar, Natasha Gelfand, Yingen Xiong, Wei-Chao Chen, Thanos Bismpigiannis, Radek Grzeszczuk, Kari Pulli, and Bernd Girod. Outdoors augmented reality on mobile phone using loxel-based visual feature organization. In *Proceeding of the 1st ACM international conference on Multimedia information retrieval*, MIR '08, pages 427–434, New York, NY, USA, 2008. ACM. 10, 63

[44] Bullet Team. Bullet physics library. Accessed on the 14th of August 2011. 22

[45] Cygwin Team. Cygwin - linux api layer for windows. Accessed on the 10th of August 2011. 7, 38

[46] Unity Technologies. Unity - cross platform game engine. Accessed on the 10th of August 2011. 2, 62

[47] Unreal Technologies. Unreal - cross platform game engine. Accessed on the 10th of August 2011. 2

[48] Unknown. The opengl extension wrangler library. Accessed on the 21st of May 2011. 23, 41, 42, 61

[49] Wikipedia. Assisted gps on wikipedia. Accessed on the 21st of August 2011. 10

[50] Wikipedia. Portable operating system interface (posix). Accessed on the 10th of August 2011. 7

[51] Martin Wojtczyk and Alois Knoll. A cross platform development workflow for c/c++ applications. *Software Engineering Advances, International Conference on*, 0:224–229, 2008. 7, 37

[52] wxWidgets. wxwidget gui toolkit. Accessed on the 21st of May 2011. v, vii, xv, 7, 8, 15, 16, 19, 22, 47, 49, 56, 57, 61

Appendix A

Source Code

A.1 CMake Configuration Input

```
#ifndef CONFIG_H
#define CONFIG_H

//Renderer configurations
#cmakedefine CURITIBA_RENDERER_OGL
#cmakedefine CURITIBA_RENDERER_OGLES2
#cmakedefine CURITIBA_RENDERER_OGL_3
#cmakedefine CURITIBA_RENDERER_OGL_3_CORE
//whether to use glew or not
#cmakedefine CURITIBA_RENDERER_OGL_GLEW


//Platform definitions
//#cmakedefine CURITIBA_PLATFORM_IOS
//#cmakedefine CURITIBA_PLATFORM_WIN32
//#cmakedefine CURITIBA_PLATFORM_OSX
//#cmakedefine CURITIBA_PLATFORM_LINUX


#if defined(__APPLE__) && __APPLE__
    #include <TargetConditionals.h>
    #define CURITIBA_PLATFORM_APPLE 1
    // iOS Platform
    #if defined(CURITIBA_PLATFORM_IOS) || defined(__IPHONE__) ||  \
    (defined(TARGET_OS_IPHONE) && TARGET_OS_IPHONE) || \
    (defined(TARGET_IPHONE_SIMULATOR) && TARGET_IPHONE_SIMULATOR)
        #undef CURITIBA_PLATFORM_IOS
        #define CURITIBA_PLATFORM_IOS 1
        #if defined(__arm__)
```

```
            #define CURITIBA_PLATFORM_IOS_DEVICE
        #elif defined(__i386__)
            #define CURITIBA_PLATFORM_IOS_SIMULATOR
        #endif
    // OSX platform
    #elif defined(CURITIBA_PLATFORM_OSX) || defined(__MACH__)
        #undef CURITIBA_PLATFORM_OSX
        #define CURITIBA_PLATFORM_OSX 1
    #elif
        #error Unknown Apple Platform
    #endif
// GNU/Linux
#elif defined(CURITIBA_PLATFORM_LINUX) || (defined(__linux) || defined(__linux__))
    #undef CURITIBA_PLATFORM_LINUX
    #define CURITIBA_PLATFORM_LINUX 1
// Windows
#elif defined(CURITIBA_PLATFORM_WINDOWS) || defined(_WIN32) || defined(__WIN32__) \
        || defined(_WIN64)
    #undef CURITIBA_PLATFORM_WIN32
    #define CURITIBA_PLATFORM_WIN32 1
#endif

#if defined(__GNUC__)
    #define CURITIBA_COMPILER_GNU
    #define CURITIBA_COMPILER_NAME "GCC"
    #define CURITIBA_COMPILER_VERSION (__GNUC__ * 1000 + __GNUC_MINOR__)
#elif defined(_MSC_VER)  // MSVC
    #define CURITIBA_COMPILER_MSVC
    #define CURITIBA_COMPILER_NAME "MSVC"
    #define CURITIBA_COMPILER_VERSION _MSC_VER
#endif


//Wether wx is present on the system
#cmakedefine CURITIBA_WX_PRESENT


#if defined(CURITIBA_COMPILER_MSVC) && CURITIBA_COMPILER_VERSION >= 1400
#ifndef _CRT_SECURE_NO_DEPRECATE
    #define _CRT_SECURE_NO_DEPRECATE
    #define _CRT_NONSTDC_NO_DEPRECATE
#endif
#endif
#ifndef _USE_MATH_DEFINES
#define _USE_MATH_DEFINES
#endif

#define CURITIBA_RENDER_FLAGS 1
#define CURITIBA_PROFILE 1
#endif
```

A.2 mkframework Script

```
'''
Created on Jun 20, 2011
Python script to create Mac OS X Frameworks. Developed since there is no way
to create a static framework for the iphone through CMake. This script can
also be used to create frameworks for dyamic libraries (default in CMake).
@author: Leander
'''
import os
import shutil

#import sys

class InfoPlist:
    '''Class which Generates a the framework's info.plist file in the resource
    folder of each version.'''
    info_plst = '''<?xml version="1.0" encoding="UTF-8"?>
    <!DOCTYPE plist PUBLIC "-//Apple Computer//DTD PLIST 1.0//EN" "http://www.apple.com/
        DTDs/PropertyList-1.0.dtd">
    <plist version="1.0">
    <dict>
    \t<key>CFBundleDevelopmentRegion</key>
    \t<string>English</string>
    \t<key>CFBundleExecutable</key>
    \t<string>${NAME}</string>
    \t<key>CFBundleIconFile</key>
    \t<string></string>
    \t<key>CFBundleIdentifier</key>
    \t<string>pt.uminho.di.${NAME}</string>
    \t<key>CFBundleInfoDictionaryVersion</key>
    \t<string>6.0</string>
    \t<key>CFBundlePackageType</key>
    \t<string>FMWK</string>
    \t<key>CFBundleSignature</key>
    \t<string>????</string>
    \t<key>CFBundleVersion</key>
    \t<string>${VERSION}</string>
    \t<key>CFBundleShortVersionString</key>
    \t<string>${VERSION}</string>
    \t<key>CSResourcesFileMapped</key>
    \t<true/>
    </dict>
    </plist>'''

    def __init__(self, name, version):
        """Constructor
        @param name: Framework name
        @param version: Framework version"""
        self.name = name
        self.version = version
```

```
    def generate(self, file):
        """Generate the info.plist file.
        @param file: Fullpath to which the information should be written"""
        outstr = InfoPlist.info_plst.replace("${NAME}", self.name).replace("${VERSION}",
            self.version)
        file = open(file, "w")
        file.write(outstr)
        file.close()



def create_if_not_exists_dir(path):
    """Simple routine to create a directory if does not exist or notify

    the user if the file is not a directory.
    @param path: Path to the directory"""
    if(not os.path.exists(path)):
        os.mkdir(path)
    elif(not os.path.isdir(path)):
        raise Exception('Path␣"' + path + '"␣is␣not␣a␣directory!')

def create_or_update_links(source, lpath, name):
    """Routine to create a symbolic link. We are required to be at the
    directory where the link will be created and the link path must be relative.
    The routine will remove any previous links if they exist

    @param source: Original file
    @param lpath: Link destination folder
    @param name : Link name"""
    curpath = os.getcwd()
    srcpath = os.path.realpath(source)
    if(os.path.exists(source)):
        os.chdir(lpath)#os.path.join(curpath,lpath))
        if(os.path.exists(name) or os.path.islink(name)):
            os.remove(name)

        os.symlink(os.path.relpath(srcpath, os.getcwd()), name)
        os.chdir(curpath)
    else:
        raise Exception('Source␣path␣"' + source + '"␣does␣note␣exist!')




def mkframework(name, version, library_path, header_root="", headers=None, path_to_config=
    None,
split=0):
    """ Create a Mac OS X Style framework and copy the library and
    headers to it, if any.
    @param name: Framework name
    @param version: Framework version
```

```
    @param library_path: Path to the library file
    @param header_root: Root folder where all the header are located
        used to determine relative paths, so that it'll maintain
        the directory structure (Optional)
    @param headers: List with all the header files to copy (Optional).
    @param path_to_config: Location of the configuration header (Optional)
    @param split: Remove the first N chars from each header
    @note: Omit the last two parameters if no headers are to be copied"""
    f_name = name + ".framework"
    f_versions = os.path.join(f_name, "Versions")
    f_cv = os.path.join(f_versions, version)
    f_cv_headers = os.path.join(f_cv, "Headers")
    f_cv_resources = os.path.join(f_cv, "Resources")
    f_cv_lib = os.path.join(f_cv, name)

    #create directories
    create_if_not_exists_dir(f_name)
    create_if_not_exists_dir(f_versions)
    create_if_not_exists_dir(f_cv)
    create_if_not_exists_dir(f_cv_headers)
    create_if_not_exists_dir(f_cv_resources)

    #copy library
    shutil.copyfile(library_path, f_cv_lib)
    shutil.copystat(library_path, f_cv_lib)

    #copy headers (if any)
    if(not headers == None):
        for header in headers.split(';'):
            header = os.path.join(header_root, header[int(split):])
            hdr_dir = os.path.join(f_cv_headers, os.path.dirname(os.path.relpath(header,
                header_root)))
            if(os.path.isdir(os.path.dirname(header)) and not os.path.exists(hdr_dir)):
                os.mkdir(hdr_dir)
            shutil.copy(header, os.path.join(f_cv_headers, os.path.dirname(os.path.relpath
                (header, header_root))))
    #copy header config file (if any)
    if(not path_to_config == None):
        if(os.path.exists(path_to_config)):
             shutil.copy(path_to_config,os.path.join(f_cv_headers,"curitiba/config.h"))

    #create info.plist
    info = InfoPlist(name, version)
    info.generate(os.path.join(f_cv_resources, "info.plist"))

    #create links
    create_or_update_links(f_cv_lib, f_name, name)
    create_or_update_links(f_cv_headers, f_name, "Headers")
    create_or_update_links(f_cv_resources, f_name, "Resources")
    create_or_update_links(f_cv, f_versions, "Current")

import sys
```

```
if (__name__ == "__main__") :
    argc = len(sys.argv)

    if(argc == 4):
        mkframework(sys.argv[1], sys.argv[2], sys.argv[3])
    elif(argc == 7):
        mkframework(sys.argv[1], sys.argv[2], sys.argv[3], sys.argv[4], sys.argv[5],sys.
            argv[6])
    elif(argc == 8):
        mkframework(sys.argv[1], sys.argv[2], sys.argv[3], sys.argv[4], sys.argv[5], sys.
            argv[6],sys.argv[7])
    else:
        sys.exit(1)
    sys.exit(0)
```

Appendix B

Screenshots

B.1 iPad User Interface in the iOS Application

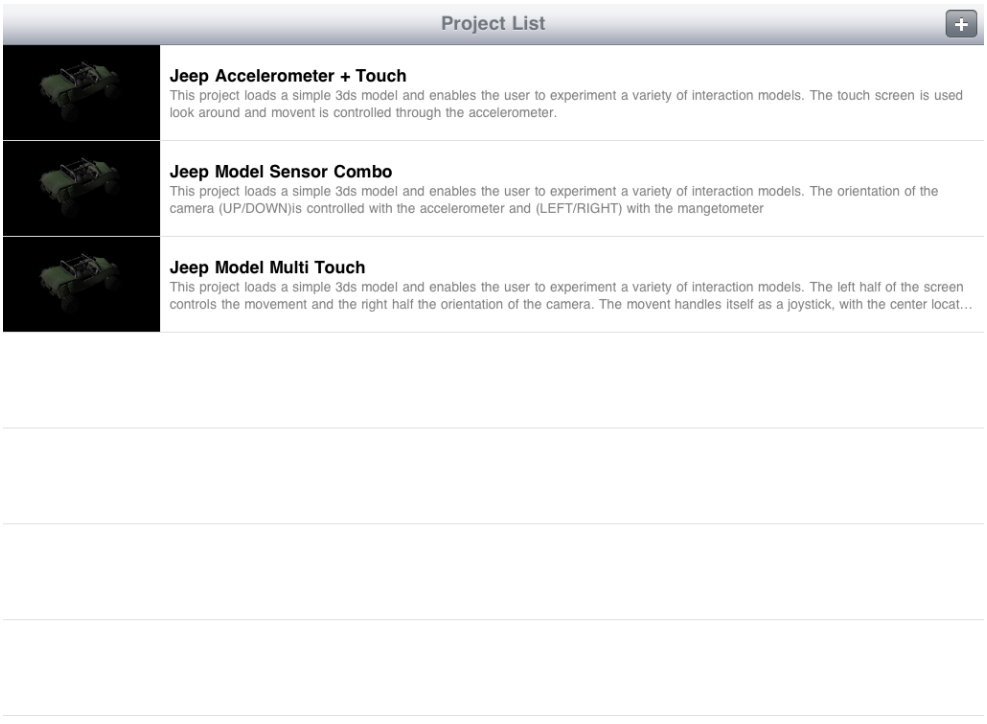


Figure B.1: Screenshot of the iPad Interface in iOS. Displayed when the application opens.

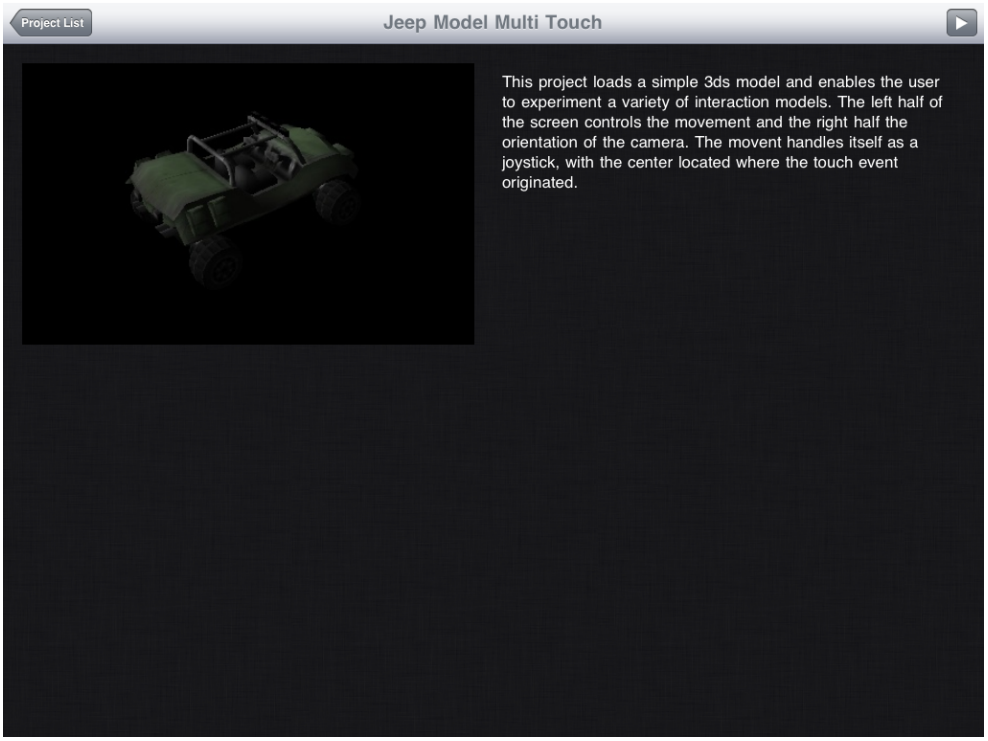


Figure B.2: Screenshot of the iPad Interface in iOS. Displayed when a project is selected.

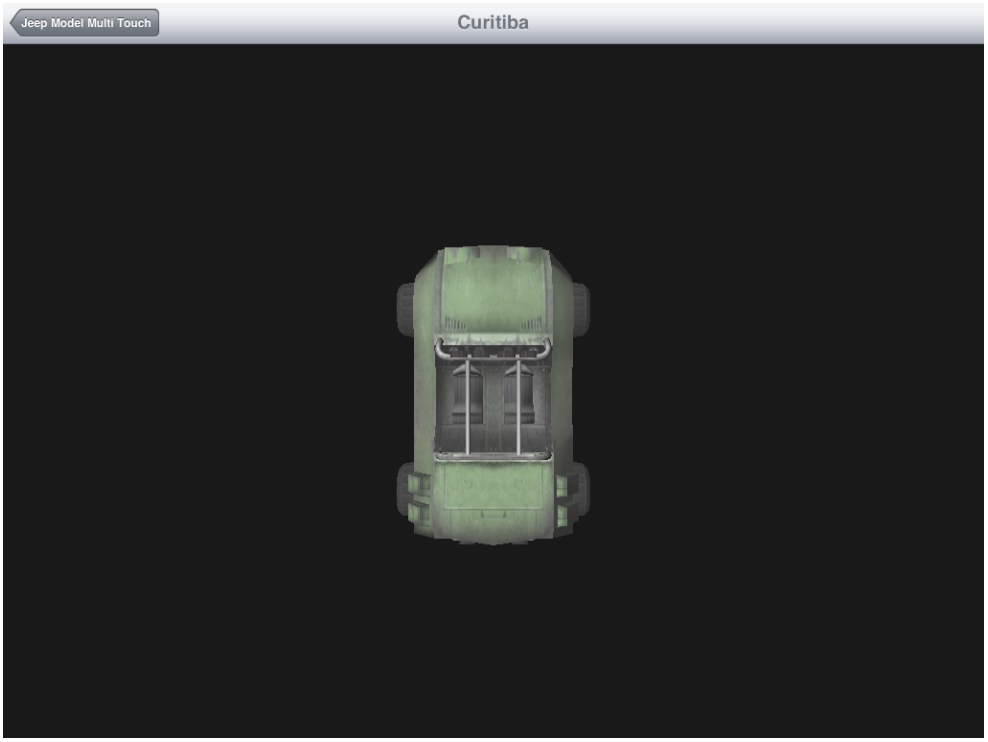


Figure B.3: Screenshot of the iPad Interface in iOS. Displayed when the project is opened.

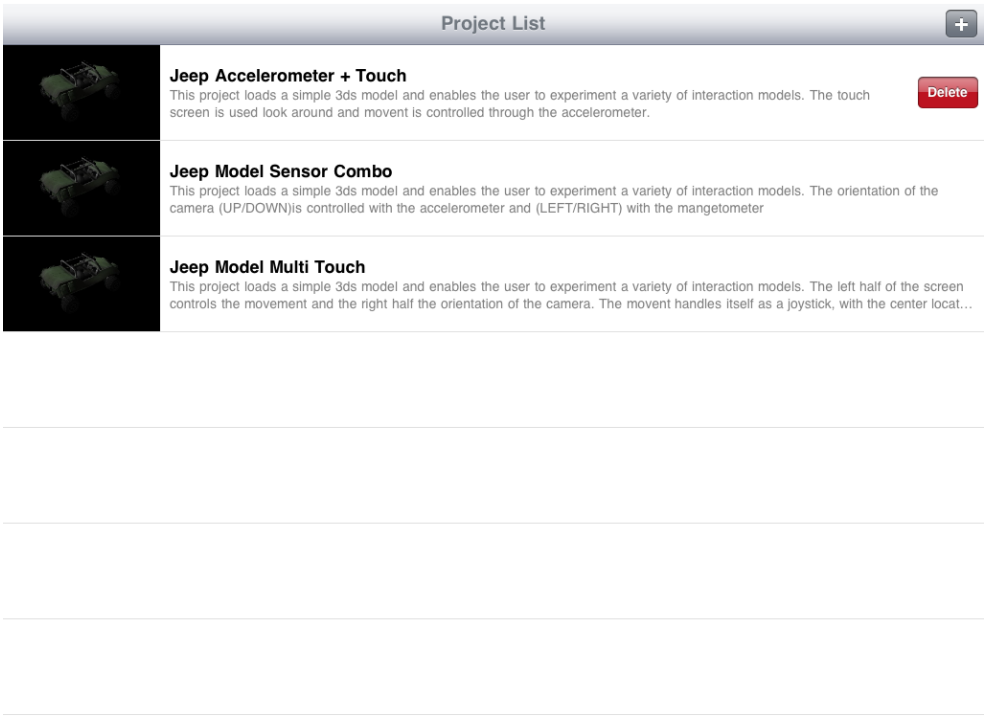


Figure B.4: Screenshot of the iPad Interface in iOS. Displayed when the user swipes a row of the project list.

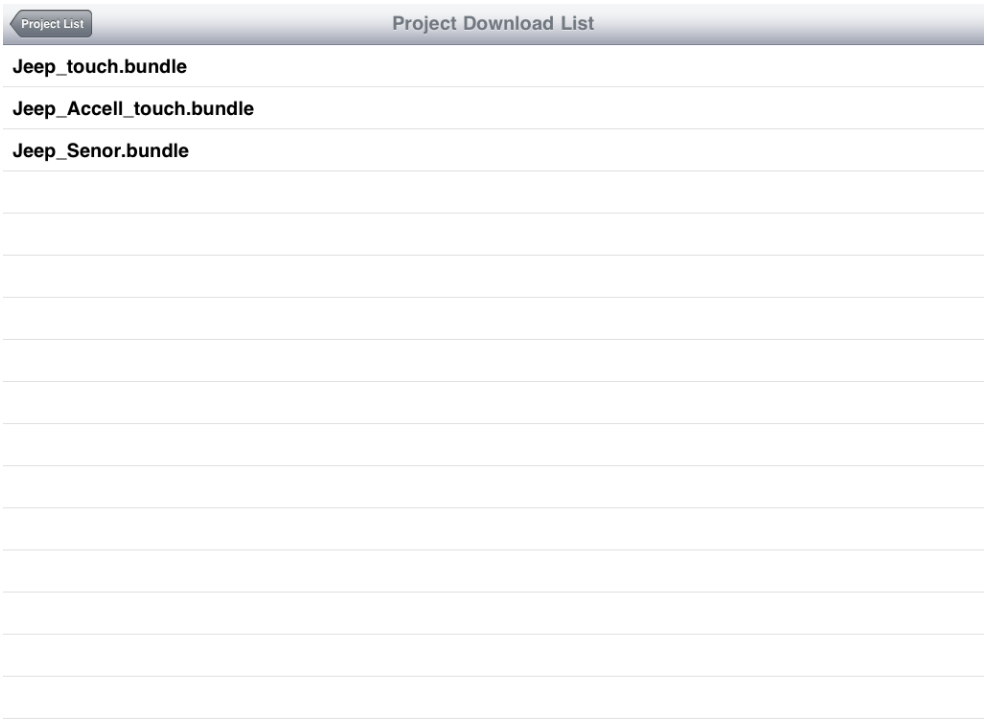


Figure B.5: Screenshot of the iPad Interface in iOS. Displayed when the add project button is clicked.

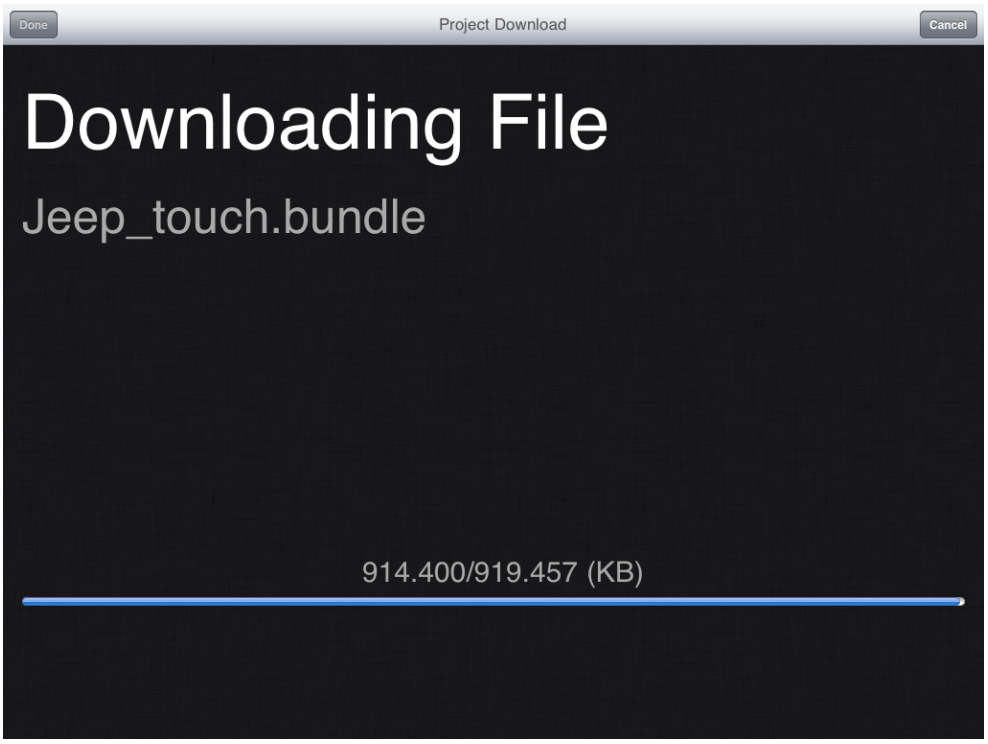


Figure B.6: Screenshot of the iPad Interface in iOS. Displayed when a project is being downloaded.