CHALMERS

HMI Toolsuite for Android

Master of Science Thesis in the Programmes Networks and Distributed Systems and Interaction Design

JAKOB STRÖM
JONATAN BROWN

Department of Applied Information Technology
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden, 2012
Report No. 2012:84
ISSN: 1651-4769

Sammanfattning

Allt eftersom antalet datorer och prestandan på dessa ökar i bilar, ökar också kraven på och möjligheterna till att presentera relevant information på ett attraktivt sätt för föraren och passagerare. För att tillmötesgå dessa krav har Mecel tagit fram Populus - en produktsvit för att designa, utveckla och driftsätta användargränssnitt (HMI - Human-Machine Interface) för inbyggda system.

För att öka tillgängligheten på Populus ska denna uppsatsen ta reda på möjligheterna att köra gränssnitt skapade med Populus på en Android-plattform. På så sätt skulle det vara möjligt att använda sig av resurser som både Android samt tredjepartsapplikationer utvecklade för Android erbjuder och därigenom öppna Populus för en större marknad.

I metoden ingår att undersöka möjligheten och svårigheterna med att porta en C/C++-applikation till Android, samt att designa ett interface som på bästa sätt demonstrerar den portade funktionaliteten.

Abstract

As the number of computers and their performance continues to increases in cars, so too do the demands and possibilities to present information in an attractive and relevant way for both drivers and passengers. To face these demands Mecel has developed Populus, a toolsuite to design, develop and run HMIs (Human-Machine Interface) for embedded systems.

To increase the availability of Populus this master thesis will explore the possibility of running user interfaces developed with Populus on an Android platform, using resources published by both Android and third party applications developed for Android.

The method contains an examination of the possibilities and difficulties of porting a C/C++ application to Android, and designing an interface that presents the ported functionality as well as possible.

Acknowledgements

We would like to thank our supervisor Fang Chen for steering us in the right direction as well as helping us with the structure of this master thesis report.

We would also like to thank Mecel for providing us with a development and testing environment and equipment in their office in Göteborg and their employees. Special thanks to Anders Arnholm, who has been our technical supervisor on the company, as well as the rest of the Populus-team.

Contents

List of Figures

1 Background

The Populus[1] suite is a set of tools developed by Mecel to develop user interfaces, or HMIs (Human Machine Interface), for embedded systems, e.g. the dash board in cars. The suite consists of an editor to create and verify the HMIs and an engine to run them on the target environment. Android is a major platform in many areas of technology and is becoming increasingly used in the automotive industry for e.g. infotainment systems.

Since Android as a system is emerging as a popular choice of platform in the automotive industry it would be very rewarding to make the Populus Engine run on Android. The Populus editor will not need porting since all HMI design can still be done on PC and exported to the Android platform.

This would also enable running Populus on tablet devices for demonstration purposes. Android also has a large third party application market[2] where a wide range of informative applications can be downloaded. It is highly desirable to find a way to integrate these applications' way of displaying information with Populus displaying of HMIs resulting in an improved user experience.

2 Introduction

2.1 Objectives

The first objective of this master thesis is to develop a reliable method to port large existing software to Android, running as a native application (an application coded specifically to run on a target platform). The method is intended to aid developers that seek to extend the availability of their software by making it compatible with the Android operating system. This will include adopting the software to work satisfactory, both with respect to usability as well as functionality, in the Android environment. It is preferred to preserve as much of the original code as possible to avoid having to maintain several versions of the software.

The second objective will explore how to integrate Populus with Android. The first goal is to connect Populus touch and window-handling to the corresponding events provided by Android. The second goal is to allow the HMI designers using Populus Editor to be able to design HMIs intended for Android without consideration for platform specifics.

The results from completing the first and second objective will result in an Android app which will be demonstrated in the third objective.The third objective is to demonstrate the results by running a prototype in a target environment simulation consisting of the Populus Engine on an Android tablet device, viewing an infotainment system connected to a car game.

2.2 Research Questions

The objectives, being primarily of practical nature, are the method for attempting to answer the following research questions:

Is it possible to port any large C++ application to Android?
Answering this question will help determine what makes an application eligible for porting.

How is it best done?
The answering of this question will result in a set of guidelines aimed at aiding in both porting and integrating software.

How to best demonstrate the ported functionality?
This question will explore differences in user interface development when primarily intended for demonstrational use, as opposed to user interfaces for everyday use.

2.3 Limitations

The ultimate future goal of the project is to have Android running in a car, with all the necessary software. This means that the requirements on the ported software will also apply to the operating system itself. To be able to meet these requirements, it is very likely that Android will have to be profoundly modified by reducing kernel and package features to speed up the boot time. This is potentially a very complicated task and it will not be addressed by this master thesis.

Furthermore, functionality not strictly necessary for running the final demonstration will be less prioritized.

2.4 Glossary and Abbreviations

- API - Application Programming Interface
- CMake - A cross-platform open-source build system available for Linux and Windows.
- EGL - An embedded systems graphics library, used for connecting APIs such as OpenGL ES and OpenVG to the underlying native platform windowing system.
- Functional Unit (FU) - Small program providing data for the HMI. Can be run in a distributed way or on the same hardware as the Engine. The FU communicates with the Engine using ODI over TCP/IP.
- HMI - Human-Machine Interface
- IDE - Integrated Development Environment
- IL - Interface Layer
- JNI - Java Native Interface. A interface that makes it possible to use Java code together with native code.
- NDK - Native Development Kit
- ODI - Protocol used for communication between the Engine and its Functional Units.
- SDK - Software Development Kit
- VM - Virtual Machine

3 Literature Studies

To create a knowledge foundation to be able to port and integrate Populus, literature studies were conducted on topics related to our task, as well as on documentation on the Populus suit provided by Mecel and Android provided by Google.

The attempts made at finding previous related work from which to gain basic knowledge of the field did not yield satisfying results. As it turned out, the research done on this subject is scarce at best. This resulted in most of the information being gathered from official specifications and various Internet forums dedicated to the subject.

The result of these studies is a base of knowledge allowing to explore and find the best way to port a C/C++ application to Android.

3.1 Populus Editor

The Populus Editor is used to create HMIs - the interfaces that the Populus Engine will display. To create the HMIs no code has to be written, instead the design and logic are defined in an XML database structure which simplifies verification. When the HMI is verified the database is converted into a binary file and downloaded to the target system on which the engine resides.

3.2 Populus Engine

The Populus Engine is the part of the Populus tool suite that handles the displaying of HMIs and their connection to Functional Units (FU). FUs are programs that handle the non interface related program logic as well as user input.

All communication between an FU and the engine is done using a protocol called “Open Display Interface” or simply ODI. There are four types of messages that can be used to transfer data between an FU and an engine.

- **Events** An Event is sent from an FU to the Engine, and contains the ID of a certain event that has occurred, for example a popup that should be shown. The Engine looks up the ID in the database to be able to carry out the appropriate action.
- **Actions** Sent to an FU from the Engine, and contain control input to the FU. For example, user input such as ’Next track’ is sent as an Action-message.
- **Indications** Contain True or False, and are used to notify the Engine of state changes in the FU.
- **Dynamic Data** A richer form of message, able to carry special information such as a certain time, number or text.

3.3 Eclipse IDE

Eclipse is an open source IDE, with support for a wide range of programming languages and frameworks. It is the recommended tool for developing Android applications as it has good integration with the Android SDK. Eclipse can be used for developing everything that comes after building the native code - bundling code and libraries to an installation package (apk) and launching the application in an emulator or physical device for testing.

3.4 Android SDK/NDK

The Android SDK is a framework that allows developing applications for Android using a Java API. The application runs through Android's own Virtual Machine - Dalvik[7]. This is a safe way of developing - memory management is automated, debugging is easy and the productivity of the programmer is high. The SDK allows the use of a wide range of system resources, and this tool is enough for most situations. However, in some cases there are benefits in being able to write unsafe, native code.

Android NDK was developed to allow developers to program their applications against a C/C++ API. It enables the developer to write performance-critical code as C/C++ modules. This thesis was written during revisions 7 through 7c of the NDK.

When coding C and C++ code on Android this is referred to as native code. This can be slightly confusing as Java applications for Android can also be called native applications. But for the remainder of this report, native code means C/C++ code.

The NDK comes with a toolchain (a set of tools used in conjunction to create a final result), which in this thesis will be used together with CMake[10]. There are disadvantages to using the NDK as opposed to the SDK only. Developing and especially debugging the application will always be more complex. Faults in the native code can cause the Virtual Machine(VM) to behave unexpectedly, especially if the bugs are a result of conflicting use of system resources[23].

3.5 Using NDK to Build an Android Application

Android NDK is intended to work in conjunction with the SDK. In fact, the main process of building and deploying the application is almost the same, with the exception of a few steps:

- Creating a 'jni'-folder in the project directory which contains the native source-files that should be included with the application.
- Specifying what libraries should be compiled from what source files in a Makefile-like file called Android.mk.
- Invoking ndk-build, a script that comes with the NDK-package. This will be run on the directory of the Eclipse-project. It determines what version of the Android API is used and selects the appropriate headers and toolchain bundled with the NDK-package to use when compiling the source-files.

After these steps, the resulting libraries end up in a 'lib' folder in the project root, and is automatically included when building the apk-package.

3.6 JNI

The Android NDK makes use of the Java Native Interface (JNI)[8]. Using JNI, the developer can declare native functions in Java, and implement these in native code. The libraries containing these functions can then be loaded in runtime. The JNI provides two means of communication between the VM and the native code. The first one is downcalls, where C/C++ functions can be called from Java code. The second is upcalls, with which the native code can make use of Java. Whenever a native function is called, a Java Native Interface environment pointer (JNIEnv*) is passed along.

Declaring a native method in Java can look like this:

```
public native void nativeString();
```

The implemented function in native code would then look like this:

```
jstring Java_the_package_name_ClassName_nativeString(JNIEnv* env, jobject o)
```

If the method would have an argument on the Java side, that argument would be the third argument in native code. The first argument env is a pointer to the VM environment. Using this, the native code can 'upcall' different methods in the VM to be able to work with Java objects.

By using JNI, the entry point of the application will be a Java Activity. An Activity can be loosely translated into a part of a program that allows user interaction. Whenever a user wants to use a specific part of an application, Android will call the corresponding Activity. This means that it is not uncommon for a simple single purpose application to have only one Activity.

3.7 Native Activity

As of Android NDK r5, it is possible to write an Activity completely in native code. This effectively eliminates the need for Java as an entry point for the application. The application can and should still be developed as an Eclipse project, and the building procedure is done as normal. Changes need to be done in the manifest-file to specify that the entry point activity is native instead of a Java-class.

The application will still be run in Dalvik - the only thing that has changed is that the initialization of the application has been moved across JNI to the native side. JNI can still be used to access Java objects if needed. Creating a native activity can be done in two ways:

Using native_activity.h. This header-file defines an interface that must be implemented to create a native activity. It contains several callback-methods that are called from the VM on events such as input, low memory, and other information from the system. These callback methods must be implemented, and should be non-blocking to prevent the application for being shut down by Android due to being unresponsive.

Using android_native_app_glue.h. This a static helper library that implements native_activity.h. When using this library rather than native_activity.h directly, a thread separate from the main program is created that takes care of all callbacks.

3.8 CMake

CMake is a cross-platform open-source build system available for Linux and Windows. It is used to manage and configure the building process of software using compiler and target environment independent configuration files. Depending on the target platform, CMake can generate e.g. Unix makefiles[9] or Visual Studio-solutions. In this thesis, Populus is built using CMake, and as one of the goals is to make use of the original source code to as great extent as possible, the porting-method will include configuring CMake to build for Android.

CMake makes use of one or more configuration files called CMakeLists.txt, written in a simple language specific to CMake. In this way, it is possible to specify platform-independent configuration options for the building of the application. The execution of CMake begins with reading the top-directory level, and recursively traversing subdirectories.

As an optional input to CMake, it is possible to specify a toolchain file, also written in CMake notation. In this way, it is possible to factor out platform-specific configuration from the CMakeLists.txt files. In this case, it will contain references to the specific tools required to compile and link native libraries from Populus, to make it able to run on Android.

3.9 Shared libraries vs Static libraries

In Linux there are two different types of libraries, shared libraries (.so) and static libraries (.a). Static libraries are statically linked to the application and the library code has to be included into the executable. If several programs are using the same libraries this requires unnecessary hard drive space. It also means that whenever a static library is updated all programs relying on it need to be recompile. The linker is however smart enough to only include the parts of the static library that are actually used into the executable, this way avoiding including a lot of unnecessary code.

When a program uses a shared library, all the executable contains is a reference to the shared library. Then whenever the program is run, the library is loaded into memory along with the program. The library is then referenced directly at run time. This allows several programs to use the same library file, but also allows libraries to be updated without recompiling the programs using them. Shared libraries do however have some overhead during runtime due to the referencing that static libraries avoid.

As Android only accepts shared libraries, any static libraries need to be linked together to a shared before being used in the application.

4 Method

This section will cover which methods will be used during the thesis project.

4.1 Development for Android

The Populus Engine is ported to Android using the Native Development Kit (NDK)[3] instead of the easier-to-use Android SDK[4]. There are two reasons for this. The primary reason is that using the NDK allows writing the applications in C++[5] and C instead of solely Java[6]. Since Populus is originally written in C++, this minimizes the amount of code that needs to be rewritten. The secondary purpose is that the NDK makes increased functionality available to the developer. Since the goal was to make Populus as integrated as possible with Android, the interface made available by the SDK is not enough. The application is supported by Android API 14, for Android 4.0.

The first step is the actual cross-compiling where the source code of Populus is compiled using the stand-alone toolchain provided with Android NDK. This is done using CMake.

The second step is making the compiled code actually run on Android. This involves implementing the required callback methods provided by Android and connecting these to Populus.

The last step is creating an FU which is the interface for communication between Android and the Engine. This is done by using a pre-existing FU SDK provided by Mecel.

4.2 Designing Demonstration

The final part of the thesis is to demonstrate the implemented functionality. An interface is developed in two stages with a user test in between. The interface needs to be designed to in an as good way as possible show all the implemented functionality. Special considerations need to be taken due to the fact that the interface is purely for demonstration and will only be used for a short period of time.

5 Porting Process

The porting process means making the application run on Android. It can be further divided into two steps. The first is compiling all the Populus code into libraries for the Android platform, the second is to make this code actually run in Android.

5.1 Compiling Populus for Android

The end result from the compilation step is a shared library (.so file) to be loaded onto the device. It cannot be a static library since it needs to be loaded at runtime. Since Populus consists of a number of internal libraries the easiest way to get this working in this project was to build these libraries as static libraries. Then, these were built into one single shared library which was finally used as the one and only native library in the application.

Using this method there would only be one shared library or executable referencing the internal libraries, this means the most efficient way to build them is as static libraries.

Normally when building libraries using the Android NDK, the easiest way to go is to use the ndk-build script included with the NDK. It allows specifying a number of options and then passes these to the regular GNU Make that builds the program using the NDK build scripts. However, Populus is already being built for various systems and architectures. In these cases CMake is used to be able to use one platform-independent language for specifying all build options. To be able to build Populus for Android, there are three steps (See figure 1):

Step one is to use the existing CMake-system with a specified toolchain. When running CMake on a project, it is possible to specify a toolchain-file that contains all platform dependent information needed to cross-compile. The toolchain file selects compilers and linkers among the pre-built executables that comes with the NDK-package. The toolchain file can also contain paths to needed header files and libraries, defined pre-processing variables, and other configurations. This will build the Populus source into different static libraries which can be used by ndk-build in the next step.

A toolchain file for CMake is small in relation to the entire CMake-system of Populus, and it will not have to be modified unless there is a new version of the NDK (and maybe not even then).

Step two is to use ndk-build, a program provided with the NDK that uses the Android Makefile language. Using a specific make file called Android.mk, the libraries and source files that should be included in the build are specified. This ensures that the final shared library will be compiled in a correct manner.

Step three aims at using the Android Manifest to package libraries and Java-source code into an installer that is used to install the app on the device.
When compiling code to different platforms a common problem is the fact that some things are implemented differently on different platforms, even though both follow the same basic standard. An example of this is fpos_t, an object used to uniquely specify a position within a file. It is implementation dependent and so needs to be handled differently on different

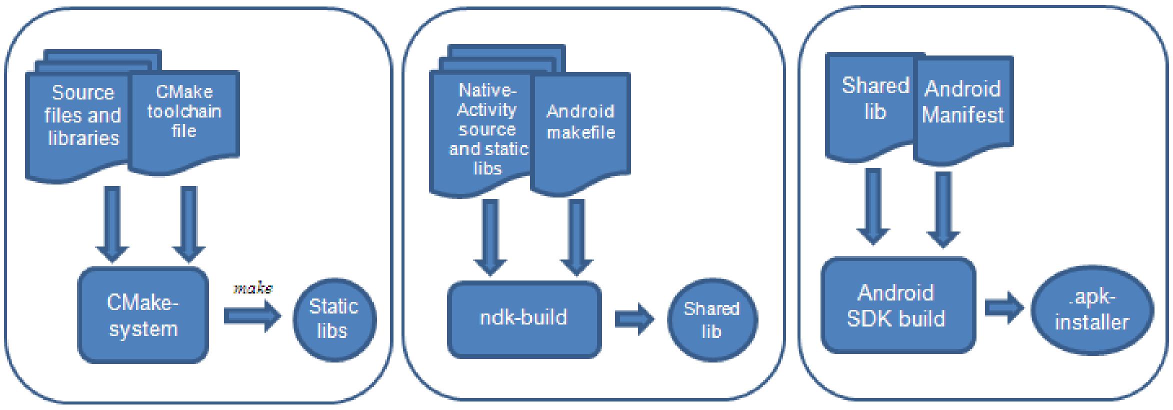

Figure 1: *CMake is used to create static libraries. The toolchain-file is the only input that needs to be specific for Android. The static libraries are then input to the next step, where a single shared library is created, which in turn is packaged into an Android installer (.apk).*

platforms. In Android it is implemented as an int that simply holds the position value, in other cases it is implemented as a struct with a member variable __pos containing the position. Since the code in some places uses a function that accesses the __pos variable of fpos_t that function was redefined to access the value of fpos_t directly instead. This was done in the preprocessing stage and only when compiling for Android.

Another example of this is the way shared memory is handled. Shared memory is used in Populus but only when external programs want to draw things into Populus. Due to this feature not being necessary for the basic functionality of Populus it was disabled entirely when building for Android.

5.1.1 Standard C++ Libraries

The standard C++ library provided with the Android platform is is a minimal library with limited capabilities. It does not provide exceptions, Run Time Type Information (RTTI) or standard C++ library support. Luckily a number or other libraries are included in the NDK and can be used instead. These are Gabi++, STLport and gnustl.

Gabi++ is a recent addition to the NDK containing the same headers as the standard system library with the addition of RTTI support.

STLport is an Android version of the C++ library with the same name. It contains all standard C++ library headers and RTTI but lacks exceptions support.

GNU STL is the standard GNU C++ library. It has RTTI, exceptions and also complete C++ standard library support.

Even though the NDK toolchain does support exceptions they are turned off by default. This due to backwards compatibility issues. If exceptions are needed they can be turned on with the ”-fexceptions” flag to the compiler. This however will only work when using the GNU STL library since it is the only one that supports exceptions.

Populus for Android will not work using the standard library since it uses both exceptions and wide characters which is something the standard library does not support. The GNU STL library therefore seemed like the correct library to use and the first attempts at compiling Populus used this library.

It turned out this did not work. Even though GNU STL supposedly has full standard library support it could not compile any code depending on wide characters, due to wide characters not being supported by Android. However, using STLport the code is able to compile despite the presence of wide character types. This is because it contains stubs which make it possible to create wchar types and store them, but not use them in any way. This means that the application will fail at runtime and workarounds have to be made whenever wide characters are used.

Since exceptions in Populus are only used for testing, no unit tests were compiled from this point and STLport was used instead. This allowed all of the libraries to compile correctly and then be combined into the wanted result, a shared library.

5.2 Making Populus Run on Android

There are two ways of turning Populus into an Android application, using JNI with Java as an entry point - or using NativeActivity to avoid Java all together.

5.2.1 Using JNI

One of the largest disadvantages with using native code in Android is the increased difficulty in debugging. The layer that JNI is between the Java code and the native code also serves as a rather effective barrier for any debugging. For example, when debugging through Eclipse, if there is a problem in the native code, the closest point in the code that will be shown is the declaration of the native method - the debugger does not cross over to step through the native code itself. For this reason, especially if the application to be ported is very large, it is important to have means of debugging that does not go through the JNI. This can be done in two ways:

- Having tests written in native code, that can run without the help of the VM.
- Having executables in native code for the program, so that it is possible to run the application completely in C/C++. This will not be possible on Android as it requires JNI to function (and can therefore not be used to get rid of bugs related to JNI).

In this thesis, both of these cases are present. Since Populus was originally a standalone application written in C++, it can run by itself without the use of a VM. Because of this, it is desired to keep the original code as intact as possible, to maintain the ability to debug and test the program this way.

Creating the Initialization Layer

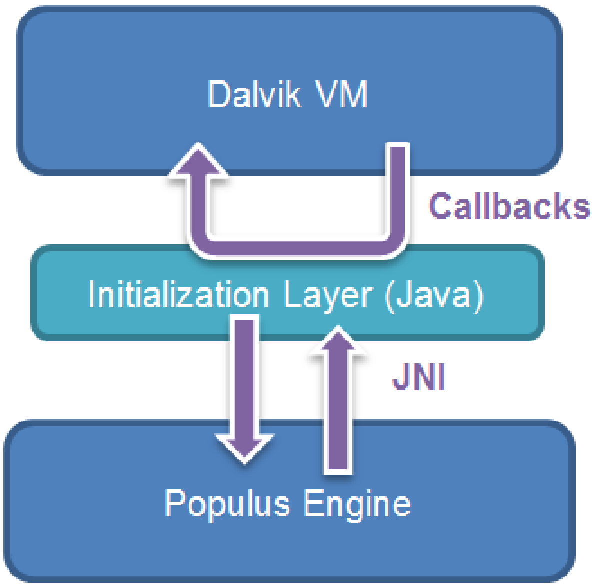

Figure 2: *Visual discription of how JNI would be used for the IL.*

The Interface Layer (IL) is an addition to the ported application that serves as an entry point for Android. When using the JNI approach (see figure 2), the IL is a thin start-up class written in Java. Its purpose is to set up necessary resources needed to start Populus as an application in Android. The class will be a subclass of Activity, just like any normal Android application, and the goal is to make it as transparent as possible. The best case would be to simply load the re-compiled library, declare a native main method, and call this method when the Activity loads. After this point, the application will load and run completely using its original native code, except for when handling events from Android, like touch input etc.

5.2.2 Using NativeActivity

When using NativeActivity[11] the largest difference will be in the IL - which will now be fully in native code (see figure 3). There are several advantages with this approach:

- No Java code is required, which simplifies maintenance.
- Every object that is initialized with an application (listeners, windows etc) will be available natively, and no explicit conversions of objects have to be done between C/C++ and Java.
- Without limitations from JNI, the developer has increased ability to integrate Android-specific functionality in the native code.

The major disadvantage is that the code becomes more complex. The developer will have to take care of many things that are implicit or very simple in Java. An example of this are the callbacks from Android. When developing in Java, the developer rarely has to take threading into account as this is managed automatically, and only has to actually implement the callback methods. When developing in C/C++ however, one must make sure not to block the callback thread or the application will be forced to exit by Android.

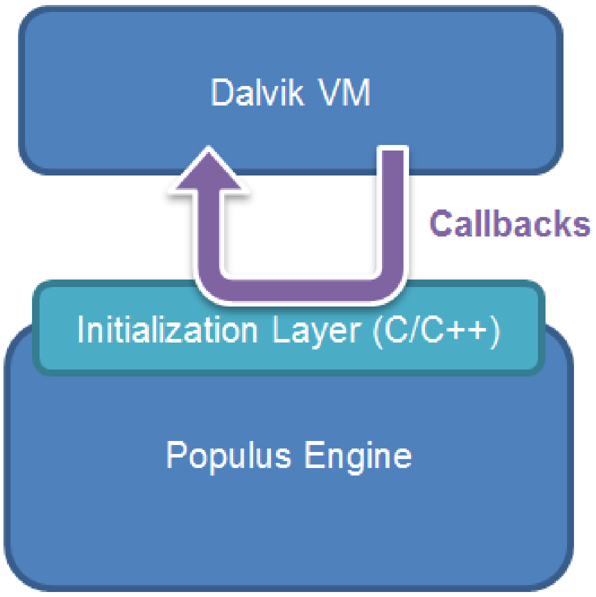

Figure 3: *Visual discription of how NativeActivity would be used for the IL.*

5.2.3 Conclusion

In this thesis, both approaches were tried and NativeActivity was found to be the better choice for one primary reason.

When using NativeActivity, all functionality for allocating Android resources to the application is made available natively. This includes touch input, display changes and many more callbacks from Android. By having access to these objects and signals natively, integrating them with the application was tremendously simplified. For example, when the application is started or resumed, a new Window-object is handed to the Activity. The native application uses this window and EGL to create an OpenGL-context and drawing surface to enable the rest of the Engine to draw on it. Substituting this initialization for Java to create such a context introduced many architectural problems that seemed impossible to solve without rewriting a large portion of the Engine.

In addition, JNI is more suited for applications written majorly in Java with a few methods written natively to enhance performance. [22][14] Porting existing applications in C/C++ using JNI is easiest if the application was written knowing it would be ported to Android, thus being able to avoid the architectural problems mentioned earlier.

Although there is some overhead involved when making JNI calls, this overhead is so small that it had no practical effect on the decision as there would only be very few calls back and forth between C/C++ and Java during initialization. [14]

5.2.4 Porting an OpenGL application

When developing an OpenGL[12] application for Android there are two APIs to choose from. OpenGL ES 1.1 or OpenGL ES 2[13]. Populus has support for drawing both via OpenGL

ES 1.1 and 2, however the support for 1.1 is not as up to date as the support for the newer OpenGL ES 2. OpenGL ES 2 was therefore the choice for graphics API going forward in the porting process. OpenGL ES 2 does however have the limitation that it does not work on the Android Emulator and must therefore be tested on a physical device. This is however a good idea anyway since even when using OpenGL ES 1.1 the emulator support of OpenGL is not optimal.

When using the JNI approach with a Java layer above the C/C++ there are two classes in the Android framework that are used, GLSurfaceView and GLSurfaceView.Renderer. The renderer has a method onDrawFrame() that is repeatedly called when the screen needs re-drawing. This is done automatically by the system and this is where the actual drawing code should go. Either by doing it directly in Java or via a call to a native method and do the drawing in C/C++ as would be the case when porting an existing C/C++ application.

It is however quite possible that the application being ported already has its own update and draw loop, which would create conflicts and some code restructuring would be needed. This is where the benefits of an entirely native solution becomes the most apparent. Doing it this way allows processing input and draw commands on each iteration of the existing render-loop, avoiding any large structural changes in the code.

5.3 Making Populus talk to Android

Android uses Intents as its way for different applications to communicate with each other and the operating system. For example to open the contacts of the phone an Intent is brodcasted that tells the operating system that it should run the application currently bound to that intent. Thus, to be able to further integrate Populus with Android, a specialized FU for Android has to be developed.

5.3.1 Android FU

The Android FU functions as a kind of middleware between Populus and Android allowing Populus to access Android resources by translating information between Populus and Android (see figure 4). An example of this is allowing an Android application to be opened by sending a Populus action to the FU, which in turn translates it into an Android intent. In addition to allowing Android applications to be launched from Populus, the FU can make available all information that Android publishes (such as time, location or accelerometer data), and translate it into a “Dynamic Data” message that Populus can make use of. This requires the FU to have permissions to access those resources.

The FU is run as an Android service, meaning it is basically an activity but without user interaction or a window (or in more general terms, simply a background process).

As previously mentioned the communication between the Populus Engine and FUs are done with ODI, and there is no difference here. When the FU starts, it waits until the engine starts up and then establishes a TCP/IP connection with it and then communicates through ODI.

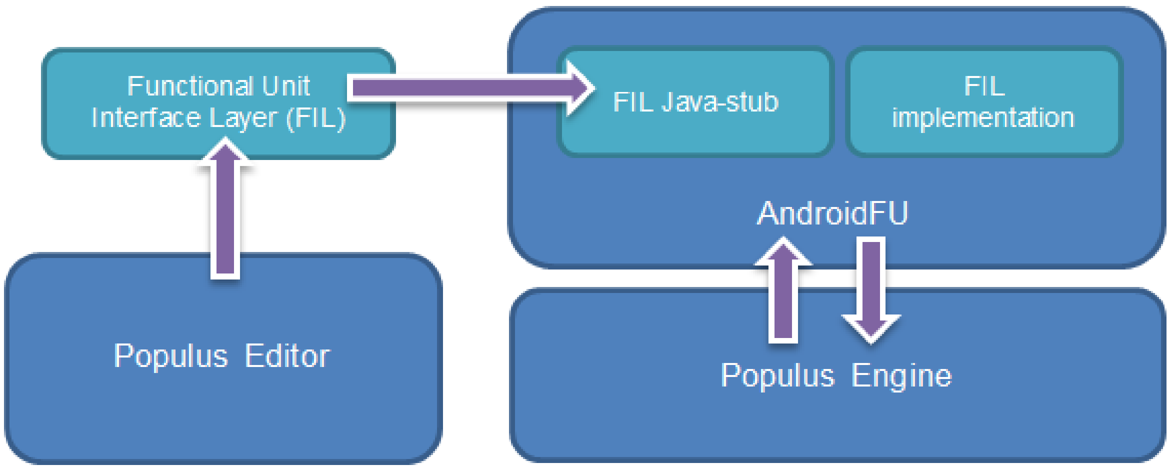

Figure 4: *When designing an HMI in the Editor, one works against a Functional Unit Interface Layer (FIL) that specifies the functions and data that will be available through the FU. These can be used to generate Java-stubs that when implemented and wrapped in an Android Service makes up for a FU able to run on Android, and communicate with the Engine.*

The FU was built using a a Java FU SDK developed by Mecel. This generates a number of Java files, amongst them an interface with methods corresponding to the different functions the FU should provide. These methods were then implemented to be able to provide the link between Android and Populus.

5.4 Results of the Porting

The porting process was successful and a resulted in Populus being runnable as an Android application. The following is a summary of the changes and additions that were made:

Wide Characters (wchar)

Since the wchar functionality in the standard libraries for Android are simply stubs and not actually functionally supported, the way Populus converted from and to wchars needed to be redefined in every place this was done. This was accomplished using the preprocessor to redefine a few lines of code to do the conversion using a different function.

Input and Callback Handling

Taking care of callbacks from Android required an addition to the main loop, that linked the signals from Android to appropriate actions within Populus.

Entry Point

To make Populus become a NativeActivity, some additions need to be made to turn the main()-function into the Activity's entry point.

Native Window

The OpenGL-initialization (and re-initialization) had to be adjusted so that it would use the window object provided by Android for the Activity to draw on.

Summary

Table 1: Code Adjustments and Additions

	Lines of Code
wchar	200
Input and Callback	55
Entry Point	15
Native Window	40
Miscellaneous	50
Total	360

There are additional files for building the shared library (Android make-files and manifest) that are separate from Populus and should need little maintenance. The total number of lines of code in Populus is 186000 (excluding header-files) of which 0.19% was changed to make Populus run on Android (see table 1). Note: The largest of these changes involved wchar workarounds, which is naturally only a problem if the application to be ported uses wchar.

6 Demonstration

The final part of this thesis concerns designing the best way to display the possibilities of running Populus on an Android platform.

6.1 Requirements

Together with Mecel a number of features were decided upon as being the most interesting to show during a demonstration. These were: Launching third party applications, time, location, compass information, information about next calendar event and some car related information provided by a separate FU. Other than that these features needed to be present, there were no restrictions on how the interface needed to look or behave. (Other than the restriction that it needed to be possible to create in the Populus Editor).

In addition to this, the demo should run well on the tablet provided (An ASUS Transformer TF101).

6.2 Design Guidelines

Since the demonstration application was to be run on a mobile device, following design guidelines for creating mobile interfaces seemed like a good starting point. However, since most design guidelines are pertaining only to applications meant for actual practical usage, some might not be entirely applicable to the specific situation at hand. This section will outline a number of common design guidelines (both general and mobile specific) and finally discuss which of these might be more or less applicable for designing a purely demonstrational application.

Avoid excessive scrolling

One of the largest hurdles when designing for a mobile device is the lack of screen real estate. This leads to a lot of scrolling when presenting a large amount of content on the same screen. There is a high likelihood this will cause the user to “get lost” on the page and not remember where that interesting piece of information was located after looking at the entire page. [15]

The importance of Colors

Colors are a great tool to convey the intention or meaning of an interface item. It is useful for establishing relations between different parts of the interface so the user immediately can connect a certain action to its result. Colors should however only be used as a secondary cue, the interpretation of its meaning can differ between context and people. [16] It is also very important to follow color coding conventions, although these may vary culturally, there are a few colors that have the same meaning globally. These are primarily red, yellow and green. [17] To avoid putting too much emphasis on colors to convey information, the best way to design an interface is to do it monochromatically, and then later add colors to emphasis certain parts of the UI. [16][17]

Design Consistency

It is extremely important to maintain consistency when designing interfaces. This allows users to create a mental model and predict what will happen when taking certain actions. A good example of this is colors. Colors need to be used consistently throughout the entire application or the user might misinterpret the meaning of a message or UI element. [18] Care also needs to be taken to make sure that similar buttons are placed in similar locations throughout the application.

Reduce Latency

This guideline concerns avoiding unnecessarily long responses to user input. If the response to an action can possibly be instant, prolonged visual acknowledgement should be avoided. [16] An example of doing this wrong is drop down menus or popups that take too long to display.

Match with the real world

Most interfaces have some connection with real world information, this connection should be used since it means the user can have some idea what effect different actions can have without having used the interface previously. This is probably most relevant when designing icons that should immediately let the users know what it is for. In for example Microsoft Word the save icon is a floppy disk to represent the fact that documents were often saved on floppy disks. (It is however a bit interesting why they still have a floppy disk as their save icon when most younger users probably haven't ever seen one in reality, but that is a discussion for a different paper)

Order visibility by importance

One of Constantine and Lockwoods [19] UI design principles is the visibility principle, it states that

"Your design should keep all needed options and materials for a given task visible without distracting the user with extraneous or redundant information. Good designs don't overwhelm users with too many alternatives or confuse them with unneeded information."

What can be taken from that is that the currently most important interface options should be very easily available and less important options should be secondary, or somehow hidden altogether. It also dictates that the most frequently used options should be the most easily available.

Appropriate level of response

All meaningful actions in an interface should have some kind of feedback to let the user know they interacted with the application somehow. This feedback should be on level with the frequency of the action. (20) Keeping frequent feedback to a low level, such as a discrete sound, will prevent the user from getting annoyed by its frequent repetition. A good example of this is the click sound when using the scrollwheel from old iPods.

Visual Appeal

This guideline states that the visual appeal of UI elements or the UI as a whole is important to the apparent usability of a UI. Interface elements should be inviting and make the user feel a desire to use them. A study by Phillips and Chaparro [20] suggests that visual appeal affects the users opinion on an interfaces usability to a large degree. A bland and boring interface will feel less usable than a aesthetically pleasing one.

6.3 Designing a demonstrational interface

The above mentioned design guidelines are guidelines formulated while having the design of "real world interfaces" in mind rather than the kind of demonstrational interface developed during this thesis. This does not mean that these guidelines do not apply, most of the theory should still be sound. There are however a few differences that need to be considered in the design of a demonstrational interface compared to a regular application UI.

The first of these is the fact that users will never use the interface for a prolonged period of time, they will use it for a short duration during a demonstration or at a conference just to quickly get a grasp of what it is about and what possibilities it has. This primarily impacts three of the mentioned design guidelines, Reduce Latency, Visual Appeal and Appropriate Level of Response.

Reducing Latency is not at all as important when designing an application that will never be used during regular work. The users simply won't have time to get tired of slow animations or transitions when they are only using the application for a short time, this is much more of an issue when you have to sit through the same animation time and time again during day to day work.

Visual Appeal becomes even more important when the user only has a very limited time to use the interface. The limited time means that the first impression will be the only thing that ever counts and should therefore be highly prioritized.

Appropriate Level of Response as a guideline makes sure the user does not get annoyed by the application feedback. Much like reducing latency this will mostly be a problem when using an application for a prolonged amount of time. The short time should allow some more leeway when giving feedback to the user, allowing more bombastic feedback for rather common actions.

Secondly, the demonstrational interface exists only to showcase a number of functions and does not necessarily have to be practically usable.

This consideration makes the Match with Reality guideline quite important. The user needs to understand what is being demonstrated since the demonstration can be run in an environment very different from where it would be practically used. The demonstration in this thesis is an application that is meant to be run in the infotainment system in a car, and it's important for the interface to convey this clearly.

The Order Visibility by Importance guideline is also affected by this. It's very likely the case that the most emphasis should not be put on the most commonly used interface elements, but rather on the ones you care the most about showing off.

With all this in mind an initial version of the interface was developed (see figure 5). The goal of the interface was to in an as good way as possible show off the features previously mentioned. A number of sketches were done and presented at a meeting. Where one was decided upon as the one to go forward with.

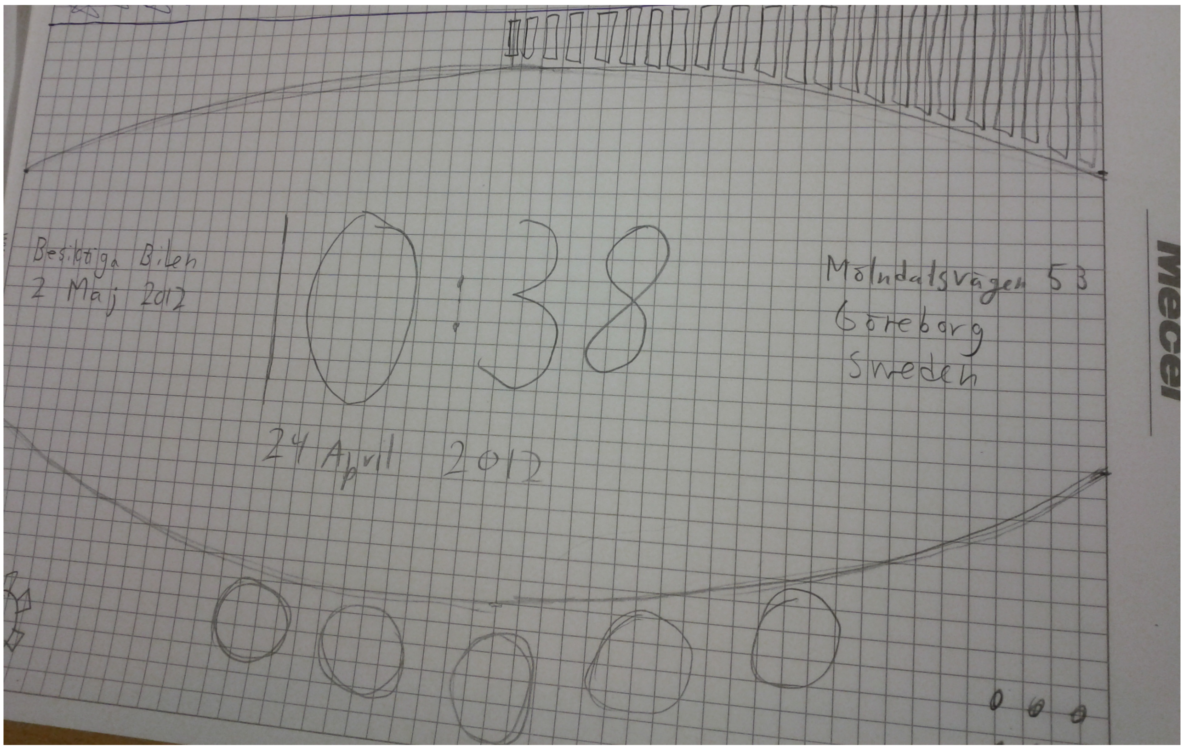

Figure 5: *The first sketch of the interface that was chosen.*

6.4 Testing the interface

To ensure that the first version of the interface (see figure 6) really demonstrated the intended functionality, and to find ways to improve it a test was run. The test was designed to confirm that the interface really presented the functionality, but also that the users understood and reflected on the presented possibilities. And failing this, hopefully some conclusions that could help us improve upon it in a second version.

The intended subjects for the test was someone with at least some knowledge of Populus, how it works and what limitations it has. At least basic knowledge of what Android is was also required. This should reflect the intended audience of the final demonstration in a realistic way.

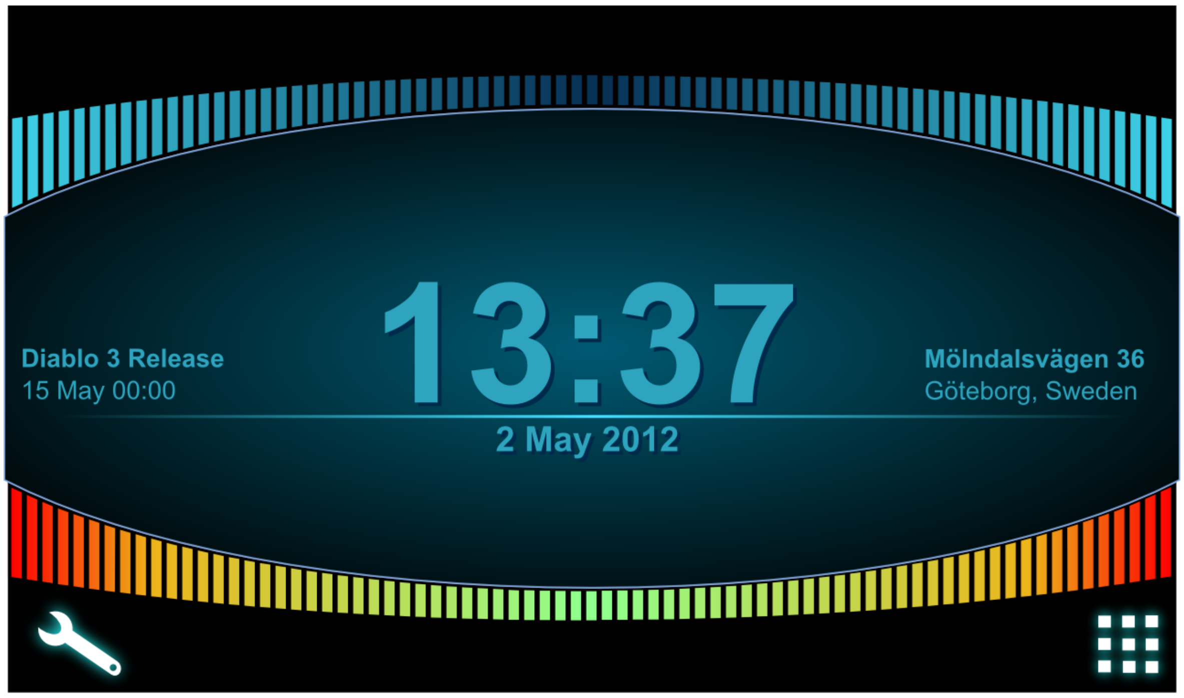

Figure 6: *This image depicts the first version of the interface. There is a large clock in the middle displaying time and date. On the left side of it the next coming calendar event is displayed from the user's calendar and on the left the user's current location is shown. There are also bars supposed to show the speed and RPM if used in an automotive environment. At the very bottom of the interface are two buttons, the left one brings up the Android settings menu and the right one a menu from which to launch other applications.*

6.4.1 Method

The test was carried out by letting the subjects explore the interface for two minutes, knowing they would afterwards answer a number of questions about the available functionality. The test subjects were five people who had some basic knowledge of Populus and Android. The test was composed of the following questions and reasons for their inclusion:

1. *Is it possible to launch a third-party Android application? (If yes, how? If no, why?)* Has the test subject understood that it is possible?
2. *Is it possible to change the brightness of the interface? (If yes, how? If no, why?)* Has the test subject understood that it is the Android settings that are available through the interface, and that changes in the settings may or may not affect the interface?
3. *Where does the location data come from?* Has the test subject understood that information existing in or collected by Android can be made available in Populus?

4. *Is it possible to change the information for the next calendar event? (If yes, how? If no, why?)* Has the test subject understood that by using third party apps, one can affect information shown in the interface?

5. *Is it possible to change the background of the interface? (If yes, how? If no, why?)* Has the test subject understood that it is the Android settings that are available through the interface, and that changes in the settings may or may not affect the interface?

Table 2: Test Results

Q1	Q2	Q3	Q4	Q5
Don't know	No	Android GPS	Yes	Yes
Yes	Yes	Android GPS	No	Yes
Yes	Yes	Android GPS	Yes	Yes
Yes	Yes	Google Maps	Yes	Don't know
Yes	Yes	Android GPS	Yes	No

6.5 Test Results

The test clearly showed that the application launching aspect of the interface was well presented (see table 2). It was also clear that the Android settings were accessible through the interface as all but one subject came to the conclusion that brightness could be changed. There was however some confusion regarding whether it was the settings for Populus or Android since only one subject figured out that the background could not be changed. This is not necessarily a problem as Populus should be tightly knit with Android. The question was included to examine whether the users would make this distinction or not. The conclusion here then is that the settings functionality is integrated in a good enough way.

Another problem was that the location and event data was presented in an unclear way. It was not even noticed by some of the subjects, but rather glanced over as part of the background. Also, the way this data is presented is too subtle for demonstration, as this data has no personal connection. Had the user been the one to input this data it would be obvious what it represented, which is not the case for a quick demonstration.

Some subjects tried to interact with the calendar and location data by tapping on them and expecting a way of editing these - functionality that is implemented but not available in this interface.

Suggestions for improvements

- Make it more obvious what the presented data is by adding graphics that represent the data.
- Make it possible to interact with relevant third party applications directly by tapping on the data.

6.6 Second Version

Based on the information gathered during the test of the first version of the interface a second version was developed. This tried to fix the shortcomings of the first. A recurring point of feedback was that users wanted to be able to access calendar and location applications directly by interacting with the information shown. This functionality was implemented, so when a user taps the calendar information, the current calendar application will be launched from Android, and likewise, tapping the location information will bring up the maps application.

The other point of feedback was that the information was not instantly identified as actual dynamic data. Some users glossed over it since it was not directly apparent what it represented. This would probably in part be remedied by the previously mentioned change, since if the users try to click on the information, they would immediately realize what it represents. As mentioned however, a lot of the users simply glossed over this information. The chosen solution to this problem was to add some kind of graphic to give the user a hint to what the information represents.

7 Discussion

The result of this thesis can be used as a guide when both deciding whether it is feasible to port an existing application to Android, and when actually carrying out the porting, and should therefore be of great use to any developer attempting this in the future.

The method used turned out to to be a good way to gather the needed information to answer the research questions. Using a practical method ensures that any unforseen problems surface and that solutions to these can be included in the thesis.

However, the downside of our method is that only one application is used as foundation. Although any missing parts have been filled by also conducting a literature study, this may cause the result to be incomplete.

7.1 First Objective - Porting

This thesis describes in some detail the tools and techniques used for porting Populus, but also some of the options that were explored but not used. It also describes a lot of useful tools and techniques that are helpful for porting. Even though a lot of issues surfaced during the porting process, the ones focused on in this thesis are more general problems that most applications should experience when being ported to Android. As expected the result was that even though there are a lot of problems with porting that are very application specific, there still are quite a few issues that can be generally described and solved. A number of these have been described and discussed in this thesis and together form a kind of method for porting, even if it is not a completely described process but more of a guide.

A problem that comes with the field is that it is constantly and rapidly changing, therefore a method for porting has much to gain from not being too specific but rather trying to describe issues on a higher level. There is very little previously written on the subject of porting C/C++ applications to Android, and very little on developing applications entirely natively in Android at all. This led to it being very hard to find reliable information on the subject. Especially due to the fact that there is a lot of out of date and contradicting information floating around on the Internet, another symptom of the rapidly changing field.

7.2 Second Objective - Integration

The integration was done in two areas. The first one was in the Populus Engine itself, and this is where there can be potential problems depending on the structure of the application to be ported.

Whenever the Engine started or resumed, Android provides a window-object that would be used by the Engine to draw. This object needed to be inserted into the initialization/reinitialization in place of whatever corresponding object is there from the previous architecture, without ruining any existing OpenGL-context. In addition to this, the touch inputs from Android had to be connected to the Engines touch handlers. The touch events are sent to a callback that must be received regularly, preferably in a main loop.

Neither of these parts provided much problem, as the touch and window-handling was separated enough from other code to allow the Android-specific insertions with relative ease. There was also a main loop well suited to receive callbacks. However, in a more complex or less structured application these modifications can be significantly harder, and these three points are something that should be looked at first when assessing the portability of an application.

The second area was to create the Android FU. Running in the background, the FU would provide Android-specific data and functionality to the HMI. By using this FU as an interface, HMI designers can use these resources without any need for Android development competence.

7.3 Third Objective - Demonstration

The demonstration interface was developed in a two-step process with a user test in between. Good feedback was gained from the test and it was very helpful for the design process since it revealed a large issue that we had not foreseen during the initial design. Namely that the location and especially event information was not very clear to users who had not created the event themselves or knew the exact address they are currently at.

7.4 Further Work

During this thesis Populus had been integrated with Android. However, as of now only the most interesting Android functions and information are made available to Populus. This means that the Android FU could be developed to provide further integration.

No effort has been made in the area of optimization. If Populus was to run on Android in an automotive environment, it would have stringent requirements on performance, particularly loading and startup times.

The interface displays the most important parts of integration using basic features of the Populus editor. An improvement to this would be the addition of some more advanced graphical effects.

8 Conclusion

This thesis presents and explores techniques and methods useful when porting existing applications to the Android platform. The experience and knowledge gathered from the process of porting the Populus engine has been used as a base for this thesis.

Populus is an application that has been ported to many different platforms previously, this means that even though the creators of Populus never had Android in mind, the general code structure might have eased the porting process considerably. Very little actual code needed to be written, meaning that, at least for programs with a good structure, porting to Android should not be too large of a project. Also, Populus already had support for rendering with OpenGL ES. Had this not been the case the amount of rewritten code would have increased drastically.

A demonstrational interface was designed to display the Populus/Android integration in a good way. A number of design guidelines were derived by studying previous work and these were then applied to the more specific case of a demonstrational interface. A user test was also designed and ran to ensure that the demonstration had the desired effect. Some understandability issues surfaced during this test and shed some light on the specific difficulties of designing something purely for demonstration.

9 Bibliography

References

[1] q Populus (2012). *Mecel Populus Suite*[Online]. Available from: http://www.mecel.se/products/mecel-populus [Accessed: May 9th 2012]

[2] App Market (2012). *Android Application Market*[Online]. Available from: https://play.google.com/store [Accessed: May 9th 2012]

[3] Android NDK (2012). *Android-NDK Specification*[Online]. Available from: http://developer.android.com/sdk/ndk/index.html [Accessed: May 9th 2012]

[4] Android SDK (2012). *Android-SDK Specification*[Online]. Available from: http://developer.android.com/sdk/index.html [Accessed: May 9th 2012]

[5] C++ (2012). *Information about C++*[Online]. Available from: http://www.cplusplus.com/ [Accessed: May 9th 2012]

[6] Java (2012). *Information about Java*[Online]. Available from: http://www.java.com/ [Accessed: May 9th 2012]

[7] Dalvik (2012). *Dalvik VM Internals*[Online]. Available from: https://sites.google.com/site/io/dalvik-vm-internals/2008-05-29-Presentation-Of-Dalvik-VM-Internals.pdf [Accessed: May 9th 2012]

[8] JNI (2012). *JNI Specification*[Online]. Available from: http://java.sun.com/docs/books/jni/ [Accessed: May 9th 2012]

[9] Unix Makefiles. *Unix make manual*[Online]. Available from: http://www.openbsd.org/cgi-bin/man.cgi?query=make [Accessed May 9th 2012]

[10] CMake. *CMake website*[Online]. Available from: http://cmake.org/ [Accessed May 9th 2012]

[11] NativeActivity. *NativeActivity specification*[Online]. Available from: http://developer.android.com/reference/android/app/NativeActivity.html [Accessed May 9th 2012]

[12] OpenGL. *OpenGL website*[Online]. Available from: http://www.opengl.org/ [Accessed May 9th 2012]

[13] OpenGL ES. *OpenGL ES website*[Online]. Available from: http://www.khronos.org/opengles/ [Accessed May 9th 2012]

[14] Lee, S. & Wook Jeon, J (2010) Evaluating Performance of Android Platform Using Native C for Embedded Systems *International Conference on Control, Automation and Systems*(Gyeonggi-do, Korea, 2010) P. 1160-1163

[15] Wobbrock, J. O., Forlizzi, J., Hudson, S. E. & Myers, B. A., WebThumb: Interaction techniques for small-screen browsers, *Proceedings of the 15th Annual ACM Symposium on User Interface Software and Technology* (Paris 2002) P. 205-208.

[16] Sajedi, A. & Mahdavi, M & Shir Mohammadi, A. & Monajjemi Nejad, M. *Fundamental Usability Guidelines for User Interface Design* Department of Computer Engineering, Azad University of Lahijan, IRAN

[17] Color Guidelines. *Windows User Experience Interaction Guidelines*[Online]. Available from: http://msdn.microsoft.com/en-us/library/windows/desktop/aa511283.aspx [Accessed May 9th 2012]

[18] Q. V. Turnell, M. & Eustáquio R. de Queiroz, J. (1996) *Guidelines - An Approach in the Evaluation of Human-Computer Interfaces* Electrical Engineering Department, Federal university of Paraiba

[19] Constantine, L & Lockwood, L *Software for Use: A practical Guide to the models and Methods of Usage-Centered Design* Reading, MA: Addison-Wesley, 1999

[20] Brian Stone, R (2002) *Designing Screen-Based Interfaces for Advanced Multimedia Functionality* Department of Industrial, Interior, and Visual Communication Design, The Ohio State University

[21] Visual Appeal. *Visual Appeal vs. Usability: Which One Influences User Perceptions of a Website More?*[Online]. Available from: http://www.surl.org/usabilitynews/112/aesthetic.asp [Accessed May 9th 2012]

[22] When to use the NDK. *Android NDK overview*[Online]. Available from: http://developer.android.com/sdk/ndk/overview.html [Accessed May 9th 2012]

[23] Czajkowski, G. & Daynks, L. & Wolczko, M. (2001) *Automated and Portable Native Code Isolation* Sun Microsystems Laboratories