The Workflow of C++ Game-Development on a Series 60 Platform device

Andreas Jakl

B A K K A L A U R E A T S A R B E I T

Nr. 238-003-045-2

eingereicht am
Fachhochschul-Bakkalaureatsstudiengang

Multimedia Technology and -Design

in Hagenberg

im May 2004

Diese Arbeit entstand im Rahmen des Gegenstands

Informatik

im

Sommersemester 2004

Betreuer:

Dipl.-Ing. Dr. Christoph Schaffer

Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt und die aus anderen Quellen entnommenen Stellen als solche gekennzeichnet habe.

Hagenberg, am 14th July 2004

Andreas Jakl

Contents

Chapter 1

Abstract

Even though the situation is changing, a developer for mobile phones is still some kind of a pioneer. For platforms like *Windows*, vast resources are available, whereas *Symbian OS* is still a rather young addition to the list of platforms. Nevertheless, or maybe even because of that, it is more fascinating than many other areas of software development.

Unfortunately, at the beginning new technologies are always lacking documentation. Several papers do exist that describe how to use individual functions or give a general overview of the architecture. This term paper targets another sector which is helpful especially to new developers and provides tactics for the workflow of programming for the *Series 60 Platform*.

This includes small tips that can be found out over the time. Having them collected and described in one paper can possibly save much time.

To provide a more practical background, an innovative location-based adventure game called *The Journey* has been developed. Based on this, several components of *Symbian OS* and *Series 60* are described, which lack documentation or where a working implementation might be more useful than reading the theoretical basics.

In general, this paper wants to summarize and transport experience that was collected during development for *Symbian OS*, to aid new developers in getting into an intriguing topic.

Chapter 2

Kurzfassung

Selbst wenn sich die Situation ändert, ist ein Entwickler für Mobiltelefone noch immer eine Art Pionier. Für Plattformen wie *Windows* sind enorme Ressourcen verfügbar, wohingegen *Symbian OS* einen relativ jungen Nachwuchs repräsentiert. Nichtsdestotrotz, oder vielleicht gerade deswegen, ist es faszinierender als viele andere Bereiche.

Unglücklicherweise ist meist wenig Dokumentation am Beginn von neuen Technologien verfügbar. Einige Artikel beschreiben, wie einzelne Funktionen eingesetzt werden, andere geben einen Überblick über die Funktionsweise der Architektur. Diese Arbeit visiert einen weiteren Sektor an, der besonders für neue Entwickler hilfreich ist, und stellt Taktiken für den Arbeitsablauf der Programmierung für die *Series 60 Plattform* zur Verfügung.

Dies enthält kleine Tipps, die man sonst mit der Zeit herausfindet. Diese gesammelt und beschrieben in einer Arbeit lesen zu können, kann sehr viel Zeit sparen.

Um einen praktischeren Hintergrund zur Verfügung stellen zu können, wurde ein innovatives ortsabhängiges Abenteuer–Spiel namens *The Journey* entwickelt. Basierend darauf werden einige Komponenten von *Symbian OS* und *Series 60* beschrieben, welche wenig dokumentiert sind, oder wo eine funktionierende Referenz–Implementation sinnvoller sein könnte, als eine Beschreibung des theoretischen Hintergrundes.

Diese Arbeit will deshalb die Erfahrung, die während der Entwicklung für *Symbian OS* gesammelt wurde, zusammenfassen und transportieren. Das Ziel ist es Entwicklern dabei zu helfen, sich in dieses spannende Thema einzuarbeiten.

Chapter 3

Introduction

This chapter will give you an overview on how the idea for this term paper evolved and sum up the contents of the individual chapters.

3.1 Motivation

Symbian OS is still rather new, compared to the other development platforms which have been around for a much longer time. In contrast to those, no vast resources are available to aid developers in doing their job. A lot of time gets can be spent researching routines and functions, which should be easy to implement.

There are some guides around which target issues that might arise. However, most of them are only about how to use individual functions or give a general overview of the architecture, papers helping with the general workflow are scarce.

This paper tries to provide some helpful tips and methods to make development for *Symbian OS* easier. Even we cover only a tiny fraction of what may need to be done, but every small bit of information will be helpful to the developers. Having many tips collected and described in one paper could potentially save you a lot of time.

To provide a more practical background, a game has been developed. A new kind of location based adventure demonstrates the technics with a real–world background.

3.2 Structure of the Paper

At first, chapter 4 is supposed to provide a short overview of what has to be done before development with Symbian can start. This includes an overview of IDEs, an installation guide for the *Series 60 SDK* and a description of the application concept.

Chapter 5 describes the game that was developed for this paper to demonstrate several components of Symbian. It also gives suggestions for the workflow and has instructions on how to get the game to run with the SDK.

In chapter 6, a more detailed description of the most interesting concepts implemented in the game can be found, along with source code examples.

Finally, chapter 7 provides a conclusion together with a short summary of the most important results and facts.

Chapter 4

Preparations

This chapter gives an overview of what has to be done before the development can start. It also presents the technologies used in this paper, gives reasons why they have been chosen and points out possible alternatives.

4.1 Prerequisites

Knowledge of C++ and object-oriented programming is required [2].

4.2 Development Environment

Currently three integrated development environments (IDEs) have support for development of *Series 60* applications. In this paper *Microsoft Visual Studio 6* is being used, as it is widely available, well–known and the main platform for the *Series 60 SDK*. Its big disadvantage is that it has no native support for mobile development; therefore many things have to be done manually. On the positive side, the user interface is established and efficient. *Visual Studio .NET* is not supported by the SDK[1]. While it does work with some tricks, the application wizard does not, and you have to do even more steps manually.

Alternatives are *Metrowerks CodeWarrior*[2], which is more tailored to mobile development and even supports on–device debugging and *Borland CBuilderX Mobile Edition*[3], which provides assistants for many cumbersome tasks like building menus or bitmap files.

It would certainly be worth downloading the free trial versions of the latter two programs if you have no previous experience with *Microsoft Visual C++*.

[1] See *FAQ-0835* at http://www3.symbian.com/faq.nsf
[2] http://www.metrowerks.com/MW/Develop/Wireless/Symbian/Default.htm
[3] http://www.borland.com/mobile/enterprise/

To develop for *Symbian OS*, a distribution of Perl is also required. *ActivePerl* from *ActiveState*[4] is free and works fine. You also have to download a copy of a *Symbian OS SDK*, in our case the *Symbian OS SDK Series 60*. Currently you can get it from Siemens[5], Nokia[6] or Sendo[7]. Choose the 2.0 SDK if you want to develop for a *Symbian OS* v7.0s phone, otherwise pick the 1.2 SDK. Please note that the game presented in this paper has not been tested with the SDK version 2.0.

4.3 Installation Guide

Install *Microsoft Visual C++ 6* and the latest Service Pack. Choose to add the paths to the system paths. Then install *ActivePerl*, also allow it to set the system paths. It makes development easier if you map a new drive for *Symbian OS*–related files. This will help you because of short paths to the files. Additionally it allows you to transfer your projects from one PC to another, no matter how and where the *Symbian OS SDK* is installed. To do that, first create a directory like `C:\Symbian\`. Then create a batch file on `C:\` with the content `subst Q: C:\Symbian`. Put this to the Startup group of Windows, so that the drive gets mapped automatically.

Now install the *Symbiasn OS SDK for Series 60* directly to `Q:\`, so that all system paths are set correctly from the beginning. Then create a directory on `Q:\` called `dev`. This is where the source code of your projects will go. As a final step, open `Q:\epoc32\Data\Epoc.ini` and correct the paths (you have to change `C:\Symbian\6.1\Series60\epoc32\wins\d` to `Q:\epoc32\wins\d`). In the folder `Q:\Series60Tools\` you'll find three useful utilities which you should also install, called *mmpclick*, the *Epoc Toolbar* and the *Series 60 AppWizard*. Read the respective text files for installation help.

After everything has been installed, try to create a new test project using the *AppWizard* [7]. Then compile and run the project. If everything worked, a dialogue box will pop up asking you for the executable file, which is the emulator (Fig. 4.1). Specify this file: `Q:\epoc32\release\wins\udeb\epoc.exe`. For the next question, tell *Visual Studio* not to ask you again and click *OK*. The emulator will load and at the end of the menu you should find your application.

[4] http://www.activestate.com/
[5] https://communication-market.siemens.de/portal/main.aspx?pid=1
[6] http://forum.nokia.com/
[7] http://www.sendo.com/dev/index.asp

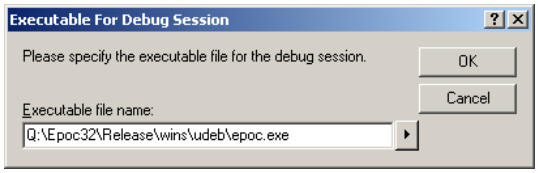

Figure 4.1: Dialog to locate the executable of the Windows emulator.

4.4 Understanding Symbian OS and Series 60

In *Symbian OS* C++ several aspects are handled differently than in standard C++. Classic examples are string handling, memory handling with the unique model of the Cleanup Stack and handling of key press events. To get an overview, the following documents provide a good summary of important aspects: [15] gives you an introduction about the unique aspects of *Symbian OS*. It features a good explanation of how memory handling works and why the concepts introduced are important to follow. Finally, if you have experience with developing in C++ for Windows, [17] explains the differences in *Symbian OS*.

4.4.1 Application Architecture

There are several ways to design an application for the *Series 60 Platform* [8, 16]. For some applications, a dialogue based approach might be useful; for others, the view–based architecture works best. As this paper focuses on a game, this chapter gives a short overview of the basic application structure for a game.

Application This class is the entry point for the application and is responsible for creating instances of the other classes. In *Series 60*, it is derived from `CAknApplication`.

Document Stores the state of the application. If the program does not need to have a persistent file, it is only used to launch the *AppUi*. In *Series 60*, it is derived from `CAknDocument`.

AppUi Handles application–wide events like menus and is responsible for view switching. Normally it does not have screen presence. In *Series 60*, it is derived from from `CAknViewAppUi` or `CAknAppUi`.

View / Controller Handles drawing to the screen and takes care of user input. It is also possible to pass a command to the *AppUi*. In *Series 60*, it is derived from from `CAknView` / `CCoeControl`.

Of course a game needs more classes, this is covered in chapter 5.2.1.

4.4.2 Server–Client Concept

For several tasks, applications need to access servers provided by *Symbian OS*. They are implemented to keep the size of the system small, as common resources can be shared by many clients [12]. The most important example is access to files. All applications define a pointer to an object of the class `RFs` (which accesses the file server), then the framework automatically calls `Connect()` so you can start to use it without creating your own instance of this object. This call is part of the client–side API, which is implemented as a shared library and provides access to the server. Another prominent example is the Telephony server which can be used to do the handling of calls, or, in the case of the game presented in this paper, to get network information.

Chapter 5

The Game

This chapter provides an overview of the game concept, a more in-depth look at the most interesting features and finally how to import the project on your local computer and getting it to run.

5.1 Journey

5.1.1 Idea and Game Concept

This paper describes several concepts and workflow–tactics based on a game that has been developed specifically to demonstrate those. This unique and new game uses several components which are not very well documented, or — as in the case of the Rich Text editor control — not documented at all [17].

The game is called *The Journey* (see Fig. 5.1) and is a location–based adventure game. The user plays the role of a detective that has to solve a case, commissioned by a mysterious man. The player has to move in the real world with the mobile phone to continue the story and to progress in the game.

The story starts in the bureau of the detective. The next part plays out a bar. To continue, the player has to change his location. The phone tracks the movement using cell–ids of the GSM network and continues the story once the player has reached a new location. Those ids are also stored so that the game remembers locations where the player has already been before. That functionality is being used in the game, which requires the user to return to the place where the whole story has started to participate in the showdown.

How far the player has progressed in the story is automatically saved; the player can resume the game if one is active. The story is visualized using text and a picture, which either shows a scene related to the story, or the general location picture.

Figure 5.1: The title screen of the *Journey* game. The screenshot was taken from the Windows emulator with a *Siemens SX1* skin.

5.1.2 Implemented Components

Code samples of several game components are available, and the SDK help file [11] provides descriptions for many of the features. However, many parts of *Symbian OS* are not (yet) documented. The game *Journey* was designed to implement some of those, in an attempt to reduce the time needed for researching how several features work. The implemented components include:

- **RichText editor control:** Only two samples are available which demonstrate the usage of editor controls[1], in the SDK help file [11] no help for these powerful controls is included.
- **Location tracking** — using cell–ids: The header file required to get the network information is not distributed with the *Symbian OS SDK for Series 60*. However, it is part of another *Symbian OS SDK* and can easily be used from within *Series 60* programs. Several commercial products like *MiniGPS* from *Psiloc*[2] use this functionality. Of course, no documentation has been written for it because it is not an officially supported API.

[1] Type Example: http://www.forum.nokia.com/main/1,6566,040,00.html?fsrParam=3-3-/main/0,,1_32_30,00.html&fileID=2809;
Rich-Text Editor Example: http://www.forum.nokia.com/main/1,6566,040,00.html?fsrParam=2-3-/main.html&fileID=3763

[2] http://www.psiloc.com/?id=prod&nrp=44

- **File handling** — saving, reading and deleting a game progress file: While these functions are documented, a fully working example that is also compatible to memory cards is hard to find.
- **Localization:** Supporting several languages is an important aspect of *Symbian OS* applications. More than one approach is possible; this paper provides a complete description of one of them. Most available samples only concentrate on menus and hardly cover in–game text.
- **Bitmap handling:** Some tips can make life with bitmaps easier; also color palettes are an issue which is rarely brought up.

5.2 Importing the Project

The following chapters describe how you can import the game project into your own installation of the *Symbian OS SDK for Series 60* and presents several workflow tactics.

5.2.1 Structure of the Game

Journey has been designed with an object oriented approach. Building on the framework provided by the *AppWizard*, several classes have been added to take care of different tasks. The basic concept is the same as described in chapter 4.4.1.

The game uses two different views, one for the menu and another one for the game itself. While this separation is a bit more work to implement, it has many advantages. The better organization of data and variables is an especially important benefit for applications with a more complex graphical menu.

The following chapters describe the classes specific to this application and provide a fundamental understanding of how the game works.

Common Routines

Some functionality needs to be available in multiple areas of the application, including methods to load bitmaps and to get the drive letter. This is implemented by using distinct namespaces. The file `Bitmapmethods.cpp`, provided by *Nokia*, encapsulates many functions which help with the handling of bitmaps. `JourneyFunctions.cpp` contains the function to add the drive and path of the application to a filename so that functions work no matter where the application has been installed.

The Menu View and Control

The classes `CJourneyVMenu` and `CJourneyCMenu` make up the routines that provide the intro picture and partly handle the menu required to start the game.

When the menu is being displayed, the game checks if a saved game file exists. If one does not, the *Continue Game* option is dimmed and not available.

The Game View and Control

While the view class is responsible for handling key presses, the control owns the text box and handles the drawing of the user interface.

Game Engine

This class is responsible for the game logic. It handles the text and pictures that are displayed. It also contains the timer functionality which is activated when the game waits for the user to move to another cell–id. Furthermore, it is responsible for loading and saving the game progress file, details can be read in chapter 6.3.

Location Servicing Class

The `CJourneyLocation` class takes care of getting and managing cell–ids. The game uses a concept of defined areas, which are cell–ids that have a special meaning to the game, such as the *Detective bureau* or the *Bar*. Building on that, the class provides functionality to wait for and check if the user has returned to an area, and to wait until he gets to a new cell–id, which can be defined as the next area or as no special area (*Streets*) if the locations should be farther apart.

Story

The data structure containing game information and the story is prepared by this class and is accessed by the game engine. It mainly associates the text resource ids from the localized language files and the bitmap ids with the `iTalkInfos` array.

5.2.2 Additional Dependencies

As has been outlined previously, getting the network information that contains the cell–id works fine on *Series 60* phones. However, the header file with the definitions of the required functions is only part of the public SDK for the *Nokia Communicator 9200*[3]. The file `etelbgsm.h` has to be ex-

[3] http://www.forum.nokia.com/main/0,,034-68,00.html

tracted from it and put into the `Epoc32\Include\` folder of your *Symbian OS SDK for Series 60* installation[4]. The *Nokia 9200* runs *Symbian OS v6.0* and *Series 80*, but the file is compatible with *Series 60*. While this header enables many other functions, only the code to get the current network information is needed here.

5.2.3 Simplifying Common Tasks

Unfortunately *Microsoft Visual Studio 6* does not have native support for *Symbian OS* development. While the SDK teaches the IDE how to do many of the tasks in the right way, not everything can be done from within the IDE, but rather from a command window. Examples are building the project for the device or building and distributing bitmap compilation files (see chapter 6.5). As having to type in the respective calls every time is a tedious task, it saves a lot of time to create batch files and integrate them into the VC6 IDE.

DoEverything.bat

During development with *Symbian OS*, you might experience problems that can be solved by cleaning and rebuilding the whole project. This procedure is also beneficial when you develop on more than one PC and want to transfer the latest version, to make sure that everything is rebuilt using your new sources.

To automate this task, create a file called `DoEverything.bat` in the folder `Q:\dev\Journey\group`. It should have the following contents:

```
call abld.bat reallyclean all
cd ..\data
call dobitmaps.bat
cd ..\group
call bldmake bldfiles
call abld.bat makefile vc6
call abld.bat build wins udeb
```

The first line removes nearly all files that have been built. The next three lines distribute the bitmap compilation files, as explained later in chapter 6.5. The following line creates a file called `abld.bat`, which serves as an entry point for the creation of workspaces and the compilation. The following call to the just created `abld.bat` creates the workspace for *Visual C++ 6*. Finally, the project is built for the emulator, including debug information.

[4] A copy of the file is included on the CD–Rom.

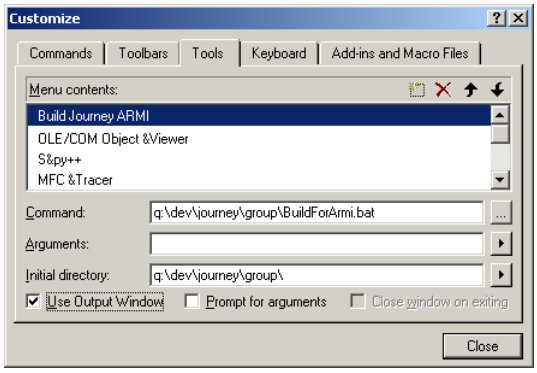

Figure 5.2: Defining a custom tool in the *Microsoft Visual Studio IDE*.

BuildForArmi.bat

Building for the device and creating the installation package has to be done manually. The batch file below, which should be created in the same directory as `DoEverything.bat`, handles the task:

```
call bldmake bldfiles
call abld.bat build armi urel
cd..\install
makesis journey.pkg
```

First the batch file builds the project for the mobile phone as a release version (without debug information), then it changes to the install folder and calls the tool to create the `.sis` file, which can be transferred to your device to be installed.

IDE Integration

As both tasks described in the previous two chapters are quite common, it is convenient to be able to call them directly from the *Microsoft Visual Studio IDE*. Go to *Tools→Customize...→Tools* to get to the dialog shown in Fig. 5.2. There you can define a shortcut to the `BuildForArmi.bat` file. As command, enter `Q:\dev\Journey\group\BuildForArmi.bat` and as the working directory `Q:\dev\Journey\group\`. Activate the option *Use Output Window* to have the text output of any possible error directly in the IDE.

Next, go to the *Keyboard* tab in the same window. For the category, choose *Tools*. Select the command *UserTool1* and assign a shortcut like *Ctrl+Alt+1*.

Once this is done, to compile for the device, you just have to press the shortcut, wait until the `.sis` file is created and then copy it to the device.

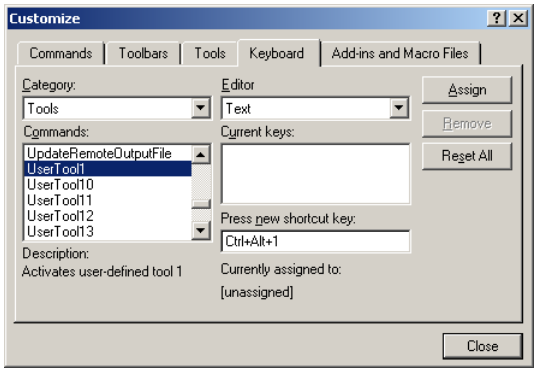

Figure 5.3: Adding a keyboard shortcut for the new custom tool in the *Microsoft Visual Studio IDE*.

Leave Scan

It is convention in *Symbian OS* that all functions that might encounter a problem should have an L at the end of their name to indicate the possibility that they might leave. *Symbian* has released a little–known utility, called *LeaveScan*, that can check if your code follows this convention. It can automatically check if all function names in a source file are correct. The tool as well as installation and usage instructions can be found in the *Symbian OS FAQ database* [14]. It can also be integrated into the IDE.

5.2.4 Testing the Project

If you have a working installation of the *Symbian OS SDK for Series 60* (see chapter 4.3) and have copied the header file for getting the network information (chapter 5.2.2), copy the Journey project folder to your `Q:\dev` and call its `DoEverything.bat`. If you have installed *mmpClick*, you just have to right–click on the `Journey.mmp` file and choose *Open VC Workspace*. Otherwise, the workspace which has been created can be found here:
`Q:\epoc32\Build\Dev\Journey\Group\Journey\Wins`
Press *F5* to run the project and follow the instructions of chapter 4.3.

5.2.5 Useful Tips

Menu Positioning

Applications you develop will always be added at the bottom right position of the menu in the emulator, so that you have to navigate down every time to run it. You can save much time by moving the icon to the upper left corner — the setting is saved and the icon will always stay in this position.

To do this, go to the icon, click the left softkey, choose *Move* and place it as the first icon in the top left corner.

Memory Leaks

If you have a memory leak in your application, it is often difficult to find. The emulator has an important feature that displays an error when you exit your application in the emulator and not all memory has been freed. You can only see this warning when you quit your application and go back to the menu instead of just stopping the process through *Visual Studio* or closing the emulator window. As it will be easier finding the error right after you made it, it is always better to close the application in the emulator. Otherwise you might see the warning several hours later and have no idea where the memory leak might come from.

Extended Error Information

When your application crashes, you only get the message "Program Closed". However, both the device and the emulator are capable of displaying an extended error information, which might be useful in some cases. To activate this, you have to create an empty file called `ErrRd` in the folder `epoc32\Wins\c\system\Bootdata`. On the device you can make the same empty file in `C:\system\Bootdata`. If you do not know how to create a file on the device, you can also install `ExtendedErr.sis`, which can be found at [1].

Enabling Syntax Highlighting for Symbian OS Keywords

The *Symbian OS SDK* comes with many new keywords and common classes that are not known to the *Visual Studio IDE* by default. However, by supplying it with a list of user defined keywords, it is possible to have those commands displayed in another color. Not only does this make the code easier to read, it also helps to prevent typing errors.

EMCC Software has provided a file which contains all keywords. Put the file `usertype.dat` (available on the CD–Rom) into the same directory as `msdev.exe` (usually:
`C:\Program Files\Microsoft Visual Studio\Common\MSDev98\Bin`)
and restart the IDE. By default, those keywords will be displayed in blue, to choose a more discrete color, go to *Tools→Options→Format→Colors→User Defined Keywords* and choose a color like dark purple [3].

Displaying Strings in the Visual C++ Debugger

Due to memory limitations, strings are handled differently in *Symbian OS*. Therefore, the debugger that is integrated into the *Visual Studio IDE* does

not know the internal structure of *Symbian OS* descriptors. It does display some information, but the most important thing is missing — the text itself.

Luckily, the IDE can be taught what to display. Open the file called `AutoExp.dat` in the main directory of your Visual Studio installation (default:
`C:\Program Files\Microsoft Visual Studio\Common\MSDev98\Bin`)
and copy/paste the text from the `SymbianAutoExp.dat` file (available on the CD–Rom) to the end of your `AutoExp.dat`. After restarting *Visual Studio*, you will be able to see the contents of descriptors as normal text in the debugging windows [5].

Chapter 6

Components

The game called *The Journey* demonstrates the use of several components which are not well documented. This chapter provides a description of the key factors that are necessary to implement them. The complete source code can be found on the CD–Rom.

6.1 The RichText Editor Control

The RichText component is well suited for displaying formatted text as it is powerful and handles many aspects automatically. It is also capable of displaying the scrollbar indicators in the softkey bar at the bottom of the screen. Using the interface it provides, it would not be very difficult to implement an editor where you can choose your own fonts, styles and alignment.

In many situations, only static text is needed, for example in an about box, or, as in the case of *The Journey*, to display the game text. The RichText editor has the advantage that the text is automatically wrapped and you do not have to take care of that yourself. The class `CJourneyCGame` includes this control.

Many border styles are available. To use them you have to include an extra library called `egul.lib` in your `.mmp` file. When creating it, the border has to be specified. In this example, no border is being used (the default would be 1 pixel):

```
iTextBox = new (ELeave)
    CEikRichTextEditor(TGulBorder::ENone);
```

Then it is constructed and its flags are set using the command below — in this case they are defined in a way to make the box read only and to hide the cursor.

```
TInt edwinFlags = EEikEdwinInclusiveSizeFixed|
```

```
                    EEikEdwinNoAutoSelection|
                    EEikEdwinDisplayOnly|
                    EEikEdwinReadOnly|
                    EEikEdwinLineCursor|
                    EEikEdwinNoHorizScrolling|
                    EEikEdwinAvkonDisableCursor;
iTextBox->ConstructL(this, 4, 0, edwinFlags,
    EGulFontControlAll, EGulAllFonts);
iTextBox->SetAknEditorFlags(edwinFlags);
```

The scrollbar frame in the softkey bar can now be activated using these calls:

```
iTextBox->CreatePreAllocatedScrollBarFrameL();
iTextBox->ScrollBarFrame()->SetScrollBarVisibilityL
    (CEikScrollBarFrame::EOff, CEikScrollBarFrame::EAuto);
```

Next, the hierarchy is defined — the container window and the observer are set to the owning class, in our case `CJourneyCGame`. After that, the position, size and background color are defined. Finally the control gets the focus.

```
// Sets the containing window for the control
iTextBox->SetContainerWindowL(*this);
iTextBox->SetObserver(this);
// Set the size of the textbox
iTextBox->SetExtent(TPoint(10, 106), TSize(156, 80));
// The background color can be specified quite easily
iTextBox->SetBackgroundColorL(TRgb(74, 56, 28));
iTextBox->SetFocus(ETrue);
```

The last step to getting a good looking text box is to set the font and text color. If they are set at the beginning, the text we will put into the control later will automatically be displayed with the style defined now. Of course it would be possible to use more than one style in a text block using this control. For example in the game this could be useful to format text in direct speech in italics.

```
// Set font type
TFontSpec fontspec = LatinPlain12()->FontSpecInTwips();
TCharFormat charFormat( fontspec.iTypeface.iName,
    fontspec.iHeight );
TCharFormatMask charFormatMask;

// Set text color
charFormat.iFontPresentation.iTextColor = KTextColor;
```

```
// Activate the attributes
charFormatMask.SetAttrib(EAttColor);
charFormatMask.SetAttrib(EAttFontTypeface);
charFormatMask.SetAttrib(EAttFontHeight);

// Apply font to the whole text
iTextBox->SelectAllL();
iTextBox->ApplyCharFormatL(charFormat, charFormatMask);
iTextBox->ClearSelectionL();
```

Defining font parameters works using the `TCharFormat` class. Several individual attributes of it can be set, then a mask has to be defined where the attributes you have just set are activated. Finally you can apply the character format together with the mask to a selected text.

Setting the text is straightforward, a `&TDesC` has to be passed to the class, which takes care of text formatting and displaying automatically. It is very important to set the cursor position to the beginning of the line before replacing the text. Otherwise, if the new text is shorter than the previous one and the cursor position is be outside of the new text, the function will leave.

```
iTextBox->SetCursorPosL(0, EFalse);
iTextBox->SetTextL(&textResource);
```

Finally, scrolling should be done when the according key press is registered [6]. There is a function called `MoveDisplayL()`, however, there doesn't seem to be a method to easily check when the bottom of the text has been reached and no scrolling down would be possible anymore, because at that point any further calls to the function will leave. Also the scrollbar would not be updated. Because of that, scrolling should to be done using the (invisible) cursor.

```
iTextBox->MoveCursorL(TCursorPosition::EFPageDown, EFalse);
iTextBox->MoveCursorL(TCursorPosition::EFLineBeg, EFalse);
iTextBox->UpdateScrollBarsL();
```

In conclusion, using the RichText editor control is not very difficult when information about how to use it is available. However, finding instructions is still difficult. Hopefully, these instructions might help and explain some of the foundations.

6.2 Location Service: Getting Cell–IDs

Retrieving the network information and, as part of it, the cell–id only works when the `etelbgsm.h` file is available (see chapter 5.2.2). You need to link your application to `gsmbas.lib` and `etel.lib` (defined in the `.mmp` file).

6.2.1 Telephony Server

For preparation, several steps are required. First, a connection to the telephony server has to be opened (a short overview about the server–client model can be found in chapter 4.4.2). The code should also include proper error handling as opening a connection may fail. These parts have been omitted in this paper, they can be found in the full source code of *The Journey*, which is available of the CD–Rom.

```
_LIT(KTsyName, "phonetsy.tsy");

iTelServer.Connect();

// Load the profile of the phone
iTelServer.LoadPhoneModule( KTsyName );

// Get the phone name
RTelServer::TPhoneInfo phoneInfo;
iTelServer.GetPhoneInfo( 0, phoneInfo );

// Open the phone by name
iPhone.Open( iTelServer, phoneInfo.iName );
```

After the connection has been established, the `LoadPhoneModule()` function has to be executed. The *TSY* file that is loaded is an extension module to the telephony server that handles the interaction between this server and a particular telephony device or family of devices [11].

Next, we fetch phone information, which also contains the name. On the emulator it is called *Calypso*, the codename of the *Nokia 7650*.

Finally the additional header file is needed for the first time. `iPhone` of class type `RBasicGsmPhone` is being opened, with the telephony server and the name of the phone as arguments.

6.2.2 Getting the Cell–ID

If all calls succeed without returning any error code, the network info can be retrieved using one simple function:

```
MBasicGsmPhoneNetwork::TCurrentNetworkInfo ni;
iPhone.GetCurrentNetworkInfo( ni );
```

If the call succeeded without returning an error code (again, error handling has been omitted in the printed source code), `ni` will be filled with some interesting information:

Cell–id Every GSM mast sends out its own cell–id. The number stored in this unsigned integer variable is therefore the code of the nearest mast.

Figure 6.1: A screenshot of the game waiting for the user to move to another location.

MCC Mobile Country Code. The code defining the country in which a mobile subscriber resides [13].

MNC Mobile Network Code. The MCC and MNC together are a unique identification number of the network the phone is logged into.

Location area A location area normally consists of several base stations. It defines an area where the mobile can move without notifying the network about its exact position [13].

Network name Available in a short and a long version, this is name of the network the phone is currently logged into. It is the same network name that is displayed on the idle screen of most mobile phones.

The emulator will always return the cell–id `0`. To test the application in the emulator, you can set a breakpoint after you fetch the ID and then change the content of the variable where you store it through the IDE.

For the *Journey* game, only movement is important (Fig. 6.1), not full location tracking. To achieve this, the area id of the network should maybe be taken into account as well.

6.2.3 Closing Connections

When you do not need to gather information anymore, or when your application is closed, you have to close the connection to the servers. This is done by the following piece of code:

```
iPhone.Close();
iTelServer.UnloadPhoneModule( KTsyName );
iTelServer.Close();
```

6.3 File Access

On mobile devices, applications can not be sure they will be exited through the menu. For example it might be that the device switches off because of an empty battery. Therefore, frequently saving the game progress is very important, especially in an adventure game. Part 2 of the *high score tutorial* has been very helpful for the design of a game progress file [4].

In *Journey*, the game is saved automatically every time the player progresses in the game. This behavior has to have four key functions:

Drive path Users should be able to install the applications on the memory card, as space on the device itself is limited. You have to make sure that the game is compatible with that, and that the drive letter is retrieved during runtime. Giving users a choice about the installation location is enabled by using a `!` instead of the drive letter in the `.pkg` file.

File exists To be able to decide whether to offer the additional option to continue the game or only to start a new game, a function has to check whether a progress file already exists.

Saving In the case of *The Journey*, all game data has to be saved that is needed to put the game into exactly the same state as it had been when the user left the game.

Loading When loading a previously saved game, everything must be back to the correct state again. For example, the timer might have to be restarted.

For the following examples the filename has been set globally using: `_LIT(KFileStore, "Progress.dat");`

6.3.1 Drive Path

It is very convenient to have a function that adds the full system path of the application to a filename. To provide this functionality outside of the `AppUi` class, several header files have to be included.

```
#include <eikenv.h>
#include <bautils.h>
#include <eikappui.h>
#include <eikapp.h>
```

The following function requires the filename as the first parameter and puts the resulting string consisting of the filename and its full path into the second parameter (*pass by reference*).

There is no direct function to retrieve the application path, so first it is necessary to get the full name and path of the application DLL. Based on this string, a useful utility function is able to extract only the drive letter and the path, removing the application name from the string. Finally, the filename can be appended to the path.

```
const void AppendFullPath(const TDesC& aFilename,
                          TFileName& aFullName)   {
    TFileName appNamePath;

    // Get the full name and path of the application
    appNamePath = CEikonEnv::Static()->EikAppUi()->
                  Application()->AppFullName();
    // Gets the drive letter and path from a file name.
    aFullName   = BaflUtils::DriveAndPathFromFullName
                  (appNamePath);

    // Always use c drive letter for the emulator.
    #if defined(__WINS__)
    aFullName.Replace(0,1,_L("c"));
    #endif

    // Combine the path and the filename
    aFullName.Append(aFilename);
}
```

6.3.2 File Exists

One method to check if a file exists is to try to open it in read only mode and check for a *file not found* error code (`KErrNotFound`). Because of the client–server model of file access in *Symbian OS*, the file server session (`RFs iFileServerSession;`) does not have to be created like an object, it is just necessary to connect to it (`iFileServerSession.Connect();`) and close the connection (`iFileServerSession.Close();`) when it is no longer needed. This is done outside of the loading function.

```
TBool GameProgressFileExists() {
    TBool returnVar = ETrue;
    TFileName fullName;
    AppendFullPath(KFileStore, fullName);

    // Try to open the file
    RFile checkFile;
    TInt errorCode = checkFile.Open(iFileServerSession,
                     fullName, EFileRead);
```

```
        // If the file is not found, return false.
        if (errorCode == KErrNotFound)
            returnVar = EFalse;

        // Close the handle to the file
        checkFile.Close();

        return returnVar;
    }
```

6.3.3 Saving Files

The current state of the game will, most of the time, be managed in RAM memory. For this case, using streams and stores is a good approach for easily writing it to a file. This method does not require taking care of file management directly. First, a *file store* is created, which is equivalent to the file itself. A *stream* can be put into it; it would also be possible to have more than one of them in one store. This example uses a *direct file store*, which does not allow modifying the data once it has been written — it can only be replaced by a new file. This behavior is fine for the needs of saving the current state of the game.

Once the store and the stream are ready, data can be written into the stream. Several functions are available to write variables of different data types. It is also possible to save the data of objects to a file store, the process is called *externalizing*. You have to implement two functions called `ExternalizeL(RWriteStream& aStream)` for writing to the stream and `InternalizeL(RReadStream& aStream)` to read the saved data from the stream again and then save it in the member variables of the class. In those, you have to take care of writing all necessary data to the stream. Externalizing the object works by `stream << myObj;`, Internalizing is equally simple by using `stream >> myObj;`.

When everything has been written to the stream, the changes have to be committed first to the stream and then to the store, where the stream is set as the root stream.

```
    void SaveGameProgressL() {
        TFileName fullName;
        AppendFullPath(KFileStore, fullName);

        // Create a direct file store that
        // will contain the game progress
        CFileStore* store = CDirectFileStore::ReplaceLC
            (iFileServerSession, fullName, EFileWrite);
        store->SetTypeL(KDirectFileStoreLayoutUid);
```

```
    // Create the stream
    RStoreWriteStream stream;
    TStreamId id = stream.CreateLC(*store);

    // Write game progress
    stream.WriteInt16L(iVar);
    // ...

    // Commit the changes to the stream
    stream.CommitL();
    CleanupStack::PopAndDestroy();  // stream

    // Set the stream in the store and commit it
    store->SetRootL(id);
    store->CommitL();
    CleanupStack::PopAndDestroy();  // store
}
```

6.3.4 Loading Files

Loading files works like saving files. The store has to be opened and then the root stream has to be found. From it, the variables can be read in exactly the same order as they were saved.

```
void LoadGameProgressL() {
    TFileName fullName;
    AppendFullPath(KFileStore, fullName);

    CFileStore* store = CDirectFileStore::OpenLC
        (iFileServerSession, fullName, EFileRead);

    // Open the data stream inside the store
    RStoreReadStream stream;
    stream.OpenLC(*store, store->Root());

    // Read all the data
    iVar = stream.ReadInt16L();
    // ...

    // Remove from cleanup stack (store, stream)
    CleanupStack::PopAndDestroy(2);
}
```

6.4 Localization

As the name indicates, *mobile* phones are intended to be used in multiple locations. Therefore it is important to prepare applications for the global market and provide support for different languages. This should be considered right from the beginning, and no text should be hard coded in source files. Care should also be taken about varying text lengths in different languages [8].

6.4.1 Menus

Two areas of the application have to be customized — the menus (if the game uses the standard system menu and does not have its own implementation) and the text in the game itself. Menus are straightforward to do, however this is a lot of work if everything has to be done manually. IDEs such as *Borland C++BuilderX* support language localization and automate a large part of the work. The following steps are necessary to set it up manually.

1. In the `mmp` file of the project, the languages have to be defined. Example for English and German: `LANG 01 03` — a list of the languages *Symbian OS* supports and their respective code numbers can be found in the definition of `TLanguage` in the file `e32std.h`.

2. The resource file (`.rss`) of your application has to include a `.loc` file. For the game this would be: `#include "journey.loc"`

3. The `.loc` file can either directly define the strings of the individual languages, or include an external file depending on the current language. For bigger applications that use more text, this method is recommended as it makes it easier to give the individual language files to translators. The content should look like this:

```
#if defined(LANGUAGE_01) /* English */
#include "Journey.l01"
#elif defined(LANGUAGE_03) /* German */
#include "Journey.l03"
#else /* Default to UK English */
#include "Journey.l01"
#endif
```

4. The individual `.l`*xx* files finally define the strings for the individual languages. Example:

```
#define qtn_menu_continue "Continue Game"
#define qtn_menu_startgame "Start New Game"
```

5. These strings are directly used in the menu definitions of the `.rss` file.

```
RESOURCE MENU_PANE r_journey_view1_menu
{
items=
    {
    MENU_ITEM { command=EJourneyCmdContinueGame;
                txt = qtn_menu_continue;  },
    MENU_ITEM { command=EJourneyCmdStartGame;
                txt = qtn_menu_startgame; }
    };
}
```

   The compiler will automatically create multiple resource files with the endings `.r01`, `.r03`, etc.

6. Finally, the application should only be installed the phone with the correct language. If available, the installation will automatically use the language which corresponds to the current phone language. The `.pkg` file based on which the `.sis` file is created has to define which files should be copied for the individual languages. An example:

```
{
    "\Epoc32\release\armi\urel\Journey.r01"
    "\Epoc32\release\armi\urel\Journey.r03"
}  -"!:\system\apps\Journey\Journey.rsc"
```

   The package file also has to specify the title of the application for all languages. If you use the additional line specifying that the installation should only be compatible to a specific version of *Series 60* (for example v0.9 and higher, meaning that it is compatible with all *Series 60* phones), this string also has to be provided multiple times — one for each language:

```
#{"Journey", "Journey"},(0x06EE1210),1,0,0

(0x101F6F88), 0, 0, 0,{"Series60ProductID",
    "Series60ProductID"}
```

6.4.2 Application Text

The localized strings defined in the resource files can also be accessed from within the application, using *resource readers*.

1. Again, the text has to be specified as above in the `.lxx` files. Those defines have to be referenced in the `.rss` file and are put into `TBuf`'s:

```
RESOURCE TBUF r_areaname_00 {buf = qtn_areaname_00; }
RESOURCE TBUF r_areaname_01 {buf = qtn_areaname_01; }
```

2. Those resources can be read and be put into `TBuf`'s from within the application. To make this call available in your own class, it has to be derived from `CCoeControl` (requires the header file: `#include <coecntrl.h>`).

```
TBuf<30> textResource;
CEikonEnv::Static()->ReadResource(textResource,
                                  R_AREANAME_00);
```

6.5 Bitmap Handling

Series 60 provides tools to easily put multiple bitmap files into `.mbm` files and to access those from within the application. While being quite straightforward to use, they have several disadvantages:

- The pictures are stored as RLE compressed bitmaps, a compression algorithm which does not create the smallest possible files.
- Bitmaps using 256 colors automatically get a standard palette assigned and do not keep their own. It is possible to specify an extra palette file with a list of the colors, which is then global to the whole `.mbm` file. However, standard graphical applications can not export the color palette of an image into the format required by the conversion utility from *Symbian*.

6.5.1 Creating the .mbm file

Still, if only a few bitmaps are required, this method is simple to use and getting the bitmaps from within the application does not take much time. It is possible to specify the parameters for the creation of the `.mbm` file in the `.mmp` file. However, you have more control over the process and its recreation, if you write your own batch file. Call this file `dobitmaps.bat` and place it together with the bitmaps in the `\data\` directory or your project.

Initially, the *bmconv* utility is called. The first parameter specifies that a header file called `journey.mbg` should be generated, containing an enumeration of the bitmaps stored in the `.mbm` file. The second parameter defines the name of the `.mbm` file to create. After that, add the individual bitmaps that should be part of the file. `/c12bmpFileName.bmp` specifies a 12–bit color bitmap. `/c8bmpFileName.bmp` an 8–bit file with the standard palette, unless an extra palette file is specified with `/pPaletteFile`. Masks should be 2–bit files (`/2bmpFileName.bmp`), using only black and white.

```
bmconv /hjourney.mbg journey.mbm /c12InterfaceTop.bmp ->
    /c12InterfaceBottom.bmp /c8areaStreets.bmp
```

The two files are generated in the current directory and have to be copied to the right paths so that they can be found from within the application.

```
copy journey.mbm Q:\epoc32\Release\armi\urel\journey.mbm
copy journey.mbm Q:\epoc32\Wins\c\System\Apps\Journey\ ->
    journey.mbm

copy journey.mbg Q:\epoc32\include\journey.mbg
```

6.5.2 Loading and Displaying Bitmaps

Getting the bitmaps out of `.mbm` files is easy and fast. The header file created by the conversion utility has to be included; it contains an enumeration of the bitmaps that are inside the `.mbm` file. Open it to see how your bitmaps are called. Add the following two lines to the class where you want to load and display bitmaps:

```
#include <journey.mbg>      // Bitmap enumeration
_LIT(KJourneyMbm, "Journey.mbm");
```

To be able to display them on the screen, you have to create a bitmap object out of them. The *Nokia* graphics examples that come with the *Symbian OS SDK for Series 60* use a collection of functions in the file `bitmapmethods.cpp`. The function `CreateBitmapL` creates an instance of the `CFbsBitmap*` class that we need. It then also adapts the color–depth to the display–depth of the device. Doing this only once when loading the image is useful because otherwise it would have to be converted every time it is displayed. This file is part of the game and can be found on the CD–Rom.

```
TFileName fullName;
AppendFullPath(KJourneyMbm, fullName);
// Load the bitmap
CFbsBmp* iBmp = NBitmapMethods::CreateBitmapL(fullName,
                                EMbmJourneyJourneytitle);
```

Once the file is loaded, displaying it is easy. To use it in the `Draw()` method of a control, you just have to copy the bitmap to the screen using `BitBlt()`. If you want to update the display with the new graphics in another place of your application, you have to force a redraw. The following code defines the redraw rectangle with the size of the picture and a fixed position. Next, this rectangle is invalidated, telling the *Symbian OS* window server that its content is outdated. Right after this, the program starts a redraw of the area and then copies the bitmap to the screen. Finally, it tells the window server that redrawing is finished.

```
TPoint bmpPos(10, 29);
TRect redrawRect(bmpPos, iCurLocationBmp->SizeInPixels());

CWindowGc& gc = SystemGc();
gc.Activate(Window());
Window().Invalidate(redrawRect);
Window().BeginRedraw(redrawRect);

// Draw it onto the screen
gc.BitBlt(bmpPos, iBmpIntroScreen);

Window().EndRedraw();
gc.Deactivate();
```

For games which require a frequently updated graphical view, it is recommended to use a background buffer bitmap of the size of the screen where all the drawing goes to. Only once a loop in the game is finished, this bitmap is copied to the screen. To get the best performance, it is possible to bypass the *Symbian OS* window server by using *Direct Screen Access*. An example demonstrating this technique is available for download at *Forum Nokia*[1].

[1] http://www.forum.nokia.com/

Chapter 7

Conclusion and Future Works

7.1 Next Steps in Development

7.1.1 Game Play Improvements

The main purpose of the game presented in this paper is to demonstrate some aspects of *Symbian OS* and *Series 60* coding. To publish the game, more interactivity would have to be added. The player should not be forced to go to the locations one after another but be able to move freely around, have an inventory and generally more freedom in what to do.

In games, it is generally good to have your own graphical menu instead of using the default menu provided by *Symbian OS* [10]. This would also be one of the next steps in the development of *The Journey*. Of course the general appearance of the game could be enhanced by using sound effects or a custom font [9].

7.1.2 Location–Based Extensions

In the current version of the game, only the movements of the person playing are tracked. Even more interesting would be to connect it with real locations. However, this would require finding and writing down cell–ids for every city and network provider. Additionally, cells are rather large. Network operators are able to detect a more accurate position by not only analyzing the id of the strongest signal, but also of the other base stations near the mobile phone. This, connected with their internal database of base stations, enables real position detection. Unfortunately, through *Series 60* it is only possible to get the id of the nearest mast. Also, the databases with the location of the cell–ids are not public. A game like that would therefore require the cooperation of network operators.

In the future, a game with this concept will be interesting when the assisted global positioning system (*A-GPS*) is more widely available in mobile phones. This would allow an accurate detection of the position of the phone. Coupled with location of companies this would also be an interesting alternative for marketing activities.

7.2 Conclusion

The mobile telecommunications world is progressing fast. There has not been much time between the introduction of color screens and the first 3D games for mobile phones. Comparing that to the world of the traditional PC, which also had a quick development, this is even more fascinating. It requires daily work to be able to keep up with the latest developments in this sector.

This does not leave much time to produce a lot of documentation; it took quite a while until the first *Symbian OS* book was released. When developing, it is not uncommon to spend a lot of time researching on the Internet how to use certain functions.

With more and more examples becoming available to the developer community, this process is slowly becoming easier. This paper provides an introduction for several aspects of *Symbian OS* and *Series 60* and also tries to help by presenting several workflow tactics and tools. Instead of finding those by chance over time during the time development, an overview is given in one convenient place: this paper.

Appendix A

Contents of the CD-ROM

File System: Joliet

Mode: Single-Session (CD-ROM)

A.1 Term Paper

Path: /

symbiangames.pdf . . .	Paper (PDF-File)
symbiangames.dvi . . .	Paper (as DVI-File, without graphics)
symbiangames.ps	Paper (PostScript-File)

A.2 Journey

Path: /Journey/

journey.sis	The installation file of the game for *Symbian OS Series 60* devices.

Path: /dev/Journey/

aif/	Application icons and `.aif`–file (*Application Information File*).
data/	Resource files and bitmaps.
doc/	Documentation of the source code.
group/	Contains the project information file and batch files to build the project.
inc/	Header files for the game and localized text files.
src/	Source code files.

install/	The package information file which contains information about creating the `.sis`–file (*Symbian Installation System*).

A.3 Literature

A.3.1 General Symbian OS

Path: /Literatur/General/

CodingIdoms/	A very good explanation of memory management in *Symbian OS* [15]
DisplayStringsInVC/ . .	Describes how to display *Symbian OS* descriptors in the *Visual Studio IDE* [5]
ExtendedErrors/	`.sis` file to display extended error information on the device plus a copy of the page from [1]
HighScoreTutorial/ . . .	Copy of the High Score Tutorial as `.pdf` [4]
LeaveScan/	The *Symbian LeaveScan* utility and usage instructions [14]
SendoGlossary/	Copies of the quoted pages of the *Sendo* Glossary [13]
SymbianWindows/ . . .	An overview of issues a Windows developer might face when working with *Symbian OS* [17]
SyntaxHighlighting/ . .	A guide on how to enable syntax highlighting for *Symbian OS* keywords in the *Visual Studio IDE* [3]

A.3.2 Series 60 Specific

Path: /Literatur/Series60/

DesigningApplications/	Provides an overview of the architecture of *Series 60* C++ applications [8]
EditorExamples/	Two examples from *Nokia*, demonstrating the Editor controls
FrameworkHandbook/ .	Covers several aspects of *Series 60*, including scrollbars, quitting applications or the applications of different architectures [6]
GameProgramming/ . .	Aspects of *Symbian OS* which are useful for developing games are presented in this document [9]

GettingStarted/	A guide that helps with creating the first *Series 60* application [7]
Tools/	An overview of the tools that come with the *Series 60 SDK*
UsabilityGuidelines/ . .	A detailled overview of guidelines that games should follow [10]

Bibliography

[1] BenK: *Display the extended panic code in Emulator or Device*. URL, http://www.newlc.com/article.php3?id_article=150, August 2003. Copy on CD-Rom; ExtendedError.pdf.

[2] Breymann, U.: *C++ Einführung und professionelle Programmierung*. Carl Hanser Verlag, 6th ed., 2001.

[3] EMCC: *Enabling Syntax Highlighting for Symbian OS Keywords*. URL, http://www.emccsoft.com/devzone/cvs.html, 2004. Copy on CD-Rom; EnablingSyntaxHighlighting.pdf, usertype.dat.

[4] NewLC: *Creation of a high score table*. URL, http://www.newlc.com/article.php3?id_article=30, March 2003. Copy on CD-Rom; HighScores_*.pdf.

[5] NewLC: *How to display Symbian strings and descriptors in Visual C++ debugger*. URL, http://www.newlc.com/article.php3?id_article=274, January 2004. Copy on CD-Rom; NewLC_DisplayStringsInVC.pdf, AutoExp.dat.

[6] Nokia: *Series 60 Application Framework Handbook*. URL, http://www.forum.nokia.com/main/1,6566,21,00.html?fsrParam=1-3-/main/0,6566,21,00.html&fileID=2489, August 2002. Copy on CD-Rom; Series_60_App_Framework_Handbook.pdf.

[7] Nokia: *Developer Platform 1.0 for Series 60: Getting Started with C++ Application Development*. URL, http://www.forum.nokia.com/main/1,6566,040,00.html?fsrParam=2-3-/main.html&fileID=3922, November 2003. Copy on CD-Rom; DP_1_0_for_S60_Getting_Started_v1_0_en.pdf.

[8] Nokia: *Developer Platform 2.0 for Series 60: Designing C++ Applications*. URL, http://www.forum.nokia.com/main/0,6566,040,00.html?fsrParam=1-3-&fileID=3773, October 2003. Copy on CD-Rom; Series_60_Designing_CPP_Applications.pdf.

[9] NOKIA: *Series 60 Developer Platform 1.0/2.0: Programming Games in C++*. URL, http://www.forum.nokia.com/main/0,6566,040,00.html?fsrParam=1-3-&fileID=4542, March 2004. Copy on CD-Rom; Series_60_Developer_Platform_1_0_2_0_Programming_Games_v1_0_en.pdf.

[10] NOKIA: *Series 60 Developer Platform 2.0: Usability Guidelines For Symbian C++ Games*. URL, http://www.forum.nokia.com/main/1,,040,00.html?fsrParam=3-3-/main.html&fileID=4624, March 2004. Copy on CD-Rom; Series_60_DP_2_0_Usability_Guidelines_For_Symbian_Games_v1_0_en.pdf.

[11] NOKIA: *Series 60 SDK for Symbian OS*, 2004. Copy on CD-Rom; help.chm.

[12] PYSSYSALO, T.: *Programming for the Series 60 Platform and Symbian OS*. John Wiley and Sons Ltd, February 2003.

[13] SENDO: *Sendo Developers – Glossary*. URL, http://www.sendo.com/kb/glossary.aspx, 2004. Copy on CD-Rom; Sendo_Glossary_Index*.pdf.

[14] SYMBIAN: *LeaveScan Utility*. URL, http://www3.symbian.com/faq.nsf/0/F3765F69E4FB9BAA80256A570051B952?OpenDocument, August 1999. Copy on CD-Rom; LeaveScan.pdf.

[15] SYMBIAN: *Coding idioms for Symbian OS*. URL, http://www.symbian.com/developer/techlib/papers/cpp_gettingstarted.html#two, October 2002. Copy on CD-Rom; 2002_10_09_codingSymbianOS.pdf.

[16] SYMBIAN: *Writing Good Symbian OS Applications*. URL, http://www.symbian.com/developer/techlib/papers/goodapplications/goodapps.html, May 2004. Copy on CD-Rom; WritingGoodSymbianOSApps.pdf.

[17] WEINSTEIN, A.: *Symbian OS C++ for Windows C++ programmers*. URL, http://www.symbian.com/developer/techlib/papers/cpp_migrating.html#three, October 2002. Copy on CD-Rom; Windows_SymbianOS.pdf.