Evaluating commercial GUIs for Ericsson Mobile Platforms

Wilhelm Lundgren, di99wln@cs.umu.se
Robert Märklund, di99rmd@cs.umu.se
Abstract

In September 2001 Ericsson Mobile Platforms (EMP) was established to help drive the development of the mobile industry. One of their platforms, the U100, consists of reference hardware and software to facilitate creation or porting of mobile phone applications. The platform offers a set of low level drawing primitives to construct the graphical components that will build the graphical user interface (GUI). EMP customers are expected to port their existing GUI towards these drawing primitives, demanding a certain level of application and GUI competence within the company. For those customers that do not have the capability or resources to develop their own GUI and applications, EMP is developing a generic user interface (UI) toolkit combined with a set of standard mobile applications. But developing a stable UI-toolkit often takes more time than expected, that is why EMP is interested in evaluating and benchmarking commercial products that often have several years lead in development. This master thesis focuses on examining a set of commercial UI-toolkits, from which one later is chosen for evaluation. To achieve this evaluation, the chosen UI-toolkit is ported to the U100 platform and benchmarked against the UI-toolkit being developed by EMP. A major part of this report consist of information on the platform and the UI-toolkit, as well as how the actual porting and benchmarking procedure was done. The conclusion of this project is that the chosen UI-toolkit can be offered to customers as a graphically light, low footprint alternative.
Contents

1 Background
2 Project Description  3
   2.1 Thesis overview  .................  3
3 Developing GUIs for handheld devices  4
   3.1 GUI objects  .....................  4
   3.2 Increasing the virtual screen  ........  5
      3.2.1 Zooming interfaces  ............  5
      3.2.2 Large virtual displays  ..........  7
      3.2.3 Focus+context displays  .........  9
      3.2.4 Event Horizon  ..................  11
   3.3 Projecting the screen  .............  12
   3.4 Summary  ..........................  14

4 Platform and UI-toolkit introduction  16
   4.1 Getting to know the Ericsson U100 Platform  ....  16
      4.1.1 OPA  .............................  17
      4.1.2 ECM Technology  ..................  19
      4.1.3 Platform communication mechanisms  ......  19
      4.1.4 Platform text handling  .............  21
   4.2 The UI-toolkits  ....................  21
      4.2.1 Trigenix  .......................  22
      4.2.2 QT/Embedded  ....................  23
      4.2.3 Peg  .............................  24
      4.2.4 Summary  .......................  25

5 Realization  26
   5.1 Choosing a toolkit  ..................  26
   5.2 Porting PEG  ........................  26
      5.2.1 Analyzing the systems  ..........  27
      5.2.2 Compile and link  ...............  29
      5.2.3 Porting  ........................  30
      5.2.4 Test applications  ...............  33
      5.2.5 Benchmarking  ....................  33
   5.3 Remaining integration  ...............  37

6 Results  39

7 Conclusion  41

8 Acknowledgments  42

A Appendix 1  46

B Appendix 2  47
## Terminology

<table>
<thead>
<tr>
<th><strong>Definition/Abbreviation</strong></th>
<th><strong>Explanation</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>API</td>
<td>Application Programming Interface.</td>
</tr>
<tr>
<td>BEA</td>
<td>Application infrastructure software company.</td>
</tr>
<tr>
<td>Canvas</td>
<td>Area on which to draw text and graphics.</td>
</tr>
<tr>
<td>CID</td>
<td>A unique 16 byte component identification number.</td>
</tr>
<tr>
<td>ECM</td>
<td>Ericsson Component Model: standard for how the application software entities are implemented.</td>
</tr>
<tr>
<td>EDGE</td>
<td>Enhanced Data GSM Environment, a faster version of GSM. EDGE enables data to be delivered at rates up to 384 Kbps on a broadband.</td>
</tr>
<tr>
<td>EIDL</td>
<td>Ericsson Interface Definition Language.</td>
</tr>
<tr>
<td>EMP</td>
<td>Ericsson Mobile Platforms.</td>
</tr>
<tr>
<td>FIFO</td>
<td>First In, First Out.</td>
</tr>
<tr>
<td>GPRS</td>
<td>General Packet Radio Service, a standard for wireless communications which runs at speeds up to 115 Kbps, compared with current GSM systems' 9.6 Kbps</td>
</tr>
<tr>
<td>GSM</td>
<td>Global System for Mobile Communications, one of the leading digital cellular systems.</td>
</tr>
<tr>
<td>GUI</td>
<td>Graphical User Interface.</td>
</tr>
<tr>
<td>HAL</td>
<td>Hardware Abstraction Layer.</td>
</tr>
<tr>
<td>IID</td>
<td>A unique 16 byte interface identification number.</td>
</tr>
<tr>
<td>JDBC</td>
<td>Java Database Connectivity. API for database access.</td>
</tr>
<tr>
<td>MIDP</td>
<td>Mobile Information Device Profile. Defines a set of API's appropriate for small mobile devices.</td>
</tr>
<tr>
<td>MMI</td>
<td>Man Machine Interface: different mechanisms of interaction between user and the platform.</td>
</tr>
<tr>
<td>ODM</td>
<td>Original Design Manufacturer.</td>
</tr>
<tr>
<td>OEM</td>
<td>Original Equipment Manufacturer.</td>
</tr>
<tr>
<td>OPA</td>
<td>Open Platform API.</td>
</tr>
<tr>
<td>PDA</td>
<td>Personal Digital Assistant.</td>
</tr>
<tr>
<td>PEG</td>
<td>Portable Embedded GUI: library and development tool designed for embedded systems.</td>
</tr>
<tr>
<td>PNG</td>
<td>Portable Network Graphics: an extensible file format for the lossless, portable, well-compressed storage of raster images.</td>
</tr>
<tr>
<td>RTOS</td>
<td>Real Time Operating System.</td>
</tr>
<tr>
<td>UI</td>
<td>User Interface.</td>
</tr>
<tr>
<td>UMTS</td>
<td>Universal Mobile Telecommunications System, a 3G mobile technology that will deliver broadband information at speeds up to 2 Mbps.</td>
</tr>
<tr>
<td>VM</td>
<td>Virtual Machine.</td>
</tr>
<tr>
<td>Z-order</td>
<td>Z-axis position of a window, where z corresponds to the depth of the screen. A window with a higher z-order is positioned in front of other windows.</td>
</tr>
<tr>
<td>ZUI</td>
<td>Zoomable User Interface.</td>
</tr>
</tbody>
</table>
1 Background

The mobile handset business is undergoing significant transformation - in much the same way as the personal computer industry did a few years ago - from being a 'vertical' industry with a few companies that supply complete products to a 'horizontal' industry with many specialized companies, each supplying their part of the products. The market for mobile phones and other portable devices is also becoming more complex, moving from the supply of basic GSM mobile phones and PDAs to the development of more advanced, and more specialized, 2.5G and 3G devices.

In September 2001 Ericsson Mobile Communications was divided into Ericsson Mobile Platforms (EMP) and Sony Ericsson. EMP was established to help drive the development of the mobile industry. It was one of the first companies in the world to license open-standard 2.5G and 3G technology platforms to manufacturers of mobile phones and other wireless information devices.

EMP is developing a base platform for mobile phones, internally called A1. The platform facilitates a high degree of customizibility and can be configured to be a low- (GSM/GPRS), middle- (EDGE) or high-end (UMTS) phone depending on the customer needs. External (market) designation of the A1 UMTS platform is U100.

Generally speaking, the platform consists of reference hardware and software to facilitate creation or porting of mobile phone applications. Application porting consists of exposing platform functionality to the end-user (e.g. audio codecs, video telephony, etc), as well as providing a link between application GUI and the platform GUI functionality. The graphical user interface of the platform is limited to a rich set of graphical drawing primitives (drawing operations), i.e. basic tools for creation widgets (graphical components). Platform customers are expected to port the GUI in their existing applications and application frameworks - or develop new applications - towards this low level functionality. This approach assumes rather high level of GUI and application competence within the customer organization. This category comprises most of the phone producers such as Sony Ericsson, LG Electronics, Sharp etc.

Many other EMP customers do not have above mentioned legacy code and/or capability and resources to develop own GUI and applications. They require a higher level of graphic capability in the platform as an extension (plug-in), i.e. presence of an User Interface Toolkit (UI-toolkit). UI-toolkit facilitates fast development of applications and definition and customization of the visual appearance and navigation paradigms ("look-and-feel") of the GUI. This is very helpful for OEM and ODM houses in branding (creating customer
specific look-and-feel, e.g. Vodafone Life) of the complete product based on EMP platforms.

EMP has initiated and is developing a generic UI-toolkit, which in combination with a collection of mobile applications constitutes as a foundation for creation mobile phone MMI. This product is not considered to be a part of the platform’s core software, it is mainly a platform value enhancer. The customers are free to choose one of the commercially available UI-toolkit solutions as an alternative. But development of an flexible and stable UI-toolkit often requires much more effort than expected and commercial products usually have several years lead in development, stabilization and optimization. It is therefore very interesting for EMP to benchmark EMP UI-toolkit and compare the solution with other products. Porting of an external UI Toolkit towards the platform can also be a proof-of-concept that EMP design of MMI functionality is correct and efficient. Another positive side effect is the fact that it provides alternative solutions if EMP would fail in producing its own competitive UI-toolkit. The ultimate goal for EMP would be to provide porting and adaptation strategies of several commercial UI-toolkits towards the platform. Platform customers would also benefit from larger range of selection - which increases flexibility.
2 Project Description

The main goal of this master thesis is to study and evaluate one or several commercial UI-toolkits. These studies, mainly aimed at technical issues such as memory and time demands, will be used as a benchmark to the EMP UI-toolkit, revealing how well it works compared to corresponding products. The evaluation will also result in strategies and guidelines facilitating future porting of other UI-toolkits as alternatives for EMP customers. To effectively compare EMP UI-toolkit with a commercial one, the latter has to be ported to the platform. If the design or the implementation of the platform proves to be very incompatible with the analyzed UI-toolkit, a proposal on changes in the platform in order to facilitate porting shall be given. If the port is successful, the performance of the ported UI-toolkit is to be compared to the one EMP is developing. This will be achieved via time measurement of specific drawing methods in the different systems.

The platform has no explicit support for touch-screen features, and the effort of adding this feature to the EMP UI-toolkit is very high. It is expected that many high end phones will be equipped with touch-screen, therefore having this feature integrated in the platform would increase the overall value of it. And since most commercial UI-toolkits already have touch-screen or mouse support integrated in their environment, this should be enabled and used. Therefore the extended target for this master thesis project is to enable this functionality by implementing corresponding functionality in the platform core code. This implies implementation of low-level drivers for the new hardware and modification in platform core code as well as additions and changes in OPA. Major effort of the master thesis will be on the ported UI-toolkit size (obtaining a low foot-print by avoiding duplicate functionality) and efficient porting (e.g. by clever solutions and thin porting layers) of one of the analyzed UI Toolkits towards GUI/MMI categories in OPA. Since different customers prefer different look and feel of their GUI this study also includes investigating the possibility to customize the look and feel of the UI-toolkits.

2.1 Thesis overview

This report will first introduce the reader to the area of developing GUI for mobile devices. This includes a presentation of the standard GUI objects used today in most devices, and also how the GUI can be changed to enhance the experience for the user. After that an introduction to the EMP platform and the commercial GUI candidates is given. This is followed by the actual porting procedure, the techniques used and the benchmarking results.
3 Developing GUIs for handheld devices

One of the currently fastest growing areas in computing deals with mobile devices such as mobile phones, personal digital assistants (PDA) and handheld computers. A big problem when interacting with these kinds of devices is the limited screen size. It is not mainly the technology that prevents the screen from becoming bigger, the physical size of the devices is a limit. These devices must, from the user’s point of view, be able to fit in a hand or a pocket very easily. [4] [22] A couple of years ago the low screen resolution and the lack of colors prevented the developers from creating graphically advanced GUIs, but these days a typical screen can have a 16 bit color depth and a high resolution, thus giving room for more advanced GUI approaches.

Presented below are some GUI approaches attempting to improve the interaction between the device and the user. These approaches attempt to enhance the user interaction with the mobile device by making it easier to retrieve information and to increase the overview of the device functionality.

3.1 GUI objects

The standard GUI objects used for interaction with a user involves buttons, lists, menus, scrollbars, windows and other similar graphical objects. In most cases these widgets (graphical objects) have been taken straight from the desktop PC GUI (where screen space not are the biggest problem) and applied directly to the mobile devices. This has resulted in widgets that are hard to use, with small text that is difficult to read and little contextual information. One problem when presenting these GUI objects in a limited screen area is that the objects should be of such a large size that the user has no problem identifying the object. Adding design effects such as 3D makes the object more attractive, but increases the size of the object even more. At the same time, the screen must be able to present more information than just one object. For example, making a button too small complicates the user interaction, making the button too big leads to less information on the screen. [4][14]

In the article Ubiquitous Computing and Cellular Handset Interfaces by Marsden and Jones[13], the problems with the standard phone GUI and how to enhance it is described. They mention the problem with displaying one menu option per page, and how this could be enhanced by creating a grid of images that represent all the options on one page. GUI developers understood that this way of representing the phone options is effective and using the right images for the right option leads to a good affordance and less key
presses. Many phones today already use this system. A way to create smaller graphical objects, such as buttons, without reducing the interaction level was found by Stephen Brewster who tried elaborating with sounds connected to each button. When a specific button was pressed, a sound was played to confirm the user of the action. He discovered that by using these sound notifications, the button area on a 3Com Palm could be reduced from 5 square mm to 2.5 without any loss of usability for the user. This will, however, increase the workload for the mobile device. [4][14]

The conclusion is that to enhance this desired usability we have to look at a higher level than the GUI objects them self. There are two main approaches to solve the problem with small screen space. The first approach is to reduce the number of visible objects on the screen. This can be achieved by creating a hierarchy, showing one level of the hierarchy instead of all objects at once. This approach is widely used in traditional GUIs, often with the negative effect of an increase of key presses. The other approach is to increase the screen size. This can be done either physically or virtually. Obviously, increasing the physical size of the screen, leading to an increased size of the mobile device, is not wanted by the end users. Therefore we are left with the option of increasing the virtual screen size.

3.2 Increasing the virtual screen

There are several techniques for increasing the virtual screen size. The section below gives an introduction to the most common ones.

3.2.1 Zooming interfaces

Several of the new user interface concepts use zooming as the main technique. These systems are built on a big virtual display where the user can control the number of visible objects by pushing or pulling objects farther or closer. By zooming the display out, the user can see a much larger number of objects than would be possible in a regular, non-zooming user interface. By zooming the display in, the user can focus on the details of individual objects. Some systems that utilize this technique even use zooming as a way of opening objects. To open an object the user simply zooms in on that object and starts working on it. By zooming out the user stops working on the object and can zoom in on another object to do something else. This technique is ideal for surfing the web, where the user in a zoomed out position can, for example, read the headlines of the news and zoom in to the interesting ones. [15][20]

Zoomable user interface (ZUI) design can utilize the following fundamental ideas, made possible by the availability of a zoomable dis-
play surface.

- **Navigation** Smooth zooming and/or panning is an intuitive method for navigating through a large work surface. The system can offer animated moves, in which the system automatically zooms out to an overview and pan/zooms to a new location.

- **Structured Information Organization** A collection of information objects can be structured for presentation in both scale (zoom) and space.

- **Selective Presentation of Detail** A range of visible detail about one object can be structured within the scale dimension around that object. The presentation of more/less detail is often coupled with spatial zooming in/out on an object, and is called semantic zooming. In semantic zooming, an object’s visual representation and the information it has changes along with the amount of spatial zooming of the object. The more an object is zoomed, the more information is shown.

Humans possess highly developed spatial abilities, which can be used in the design of a zoomable user interface. These spatial abilities include navigating to a specific point/area of a populated surface, recalling the position or context of an object on the surface, reasoning, and direct manipulation of objects. The zoomable interfaces have proved to be most effective when the two-level zoom technique is used. Two-level zoom only has two states, a zoomed out state on the entire working area, and a zoomed in state that works similar to the virtual large displays. [2]

Jazz is a new Java toolkit that supports the development of extensible 2D object-oriented graphics with zooming and multiple representations. While Jazz does not introduce many substantially new individual ideas, it is novel in bringing together a variety of techniques from different domains. Jazz takes scene graphs from 3D graphics, screen and interaction techniques from 2D widgets, functionality from previous ZUI systems, and puts these elements together with clean decoupled object oriented design. This enables developers to write zoomable applications and advanced visualizations with a clarity and efficiency that has not been possible before. [3]

In addition to the zoom, one can add the feature of lenses to the ZUI. A lens shows a visual representation of a particular set of objects that overlap the lens. The user interacts with these objects through the lens, or interacts with the lens through different kinds of controls embedded in the lens itself. What a lens displays is called
a view. Lenses provide the following benefits in the design of user interfaces [20]:

- Lenses reduce graphical display clutter and informational display clutter. Instead of using global overlays, a lens can be scaled to cover a portion of the display or a smaller lens can be used over an area of interest. Multiple kinds of data can be viewed simultaneously, without clutter problems, by using multiple lenses.

- Lenses help the user maintain spatial and information context. A lens and its view are always visible within the context of whatever is displayed in the global screen display. Selective use, placement and scanning of lenses allows a user to view exactly the necessary data throughout the process of doing an analysis task.

- New lenses can be constructed by stacking and optionally gluing multiple lenses together. The user can (interactively) create a complex lens from primitive lens elements to display any combination of information.

- Lenses provide real-time response to the user, even for operations that involve intense computation. For example, a lens can perform image enhancement in real time. This is possible because a lens typically shows data for a small area, which can be processed in real time. Large areas can be quickly scanned by a small lens.

- Lenses are a natural fit for spatial database queries and visualization, and the iterative construction of complex database queries.

One potential problem with these techniques is that they demand a quite high resolution screen. If the user frequently needs to move objects back and forth to improve legibility or to see more objects, the value of the user interface will decrease substantially. Further, the technique of zooming smoothly and realistically demands quite a lot computing power. [2][20][15]

### 3.2.2 Large virtual displays

This kind of technique, using a lens to complement the zooming, is closely related to the area of creating infinitely large virtual displays. This technique presents the user with a 2 dimensional screen area where the user has the physical screen as a peephole on the virtual display. There is no hierarchical structuring of user interface
space: all objects are assumed to reside in the same world. To facilitate navigation and understanding of global context, additional tools such as the radar view are provided. When reading a text document the user can move to the left upper corner of the screen and then move along as he reads the text. This technique can also be seen in existing systems: for example, many laptops allow the display to be set to a resolution higher than what the screen can show; the user can then make use of the larger space by panning the screen with the mouse. [22][21] Navigating maps is a great challenge for applications running on handheld devices. Because of the limited screen size only a small fraction of a map can be seen in full detail at once. Thus usually map navigation is often associated with intense panning and zooming, making many users frustrated. One way to make map navigation easier is to show arrows in the border region of the screen that point in the direction of the off screen locations and add numbers to the arrows to indicate the distance. The problem with this approach is that locations sharing the same bearing have intersecting arrows and can lead to confusion. HALO was introduced to overcome this issue by using circles around the off screen locations with a radius large enough to range into the border region of the screen. The human perception is very efficient.

![Figure 1: The HALO system.](image)

to reconstruct a circle by even small fractions, this is why arcs in the border region are optimal for indicating direction and distance of the center of the circle. As intersection also can happen with HALO when two off screen locations are very close to each other,
HALO provides a technique to merge the intersecting arcs to double arcs. By using thick arcs instead of thin ones it is also possible to represent more than two arcs this way. The HALO system always updates the arcs if the user has panned on the map, so arcs grow and shrink when moving around the map. [27]

A user study was performed and the results were that HALO outperformed the arrow-based navigation in almost every aspect. The subjects found the use of HALO less tedious and their error rates and task completion times were lower. HALO proved to be a more efficient tool when map navigation is performed compared to the traditional arrow based system. [22][27]

3.2.3 Focus+context displays

Focus and contextual displays use a variant of the zooming technique. The idea behind this is showing the information that the user currently focuses on in detail while more distant objects are only shown as rough sketches. This is very similar to the technique of using a ZUI complemented with a lens mentioned above. For humans, this is a very natural way of perceiving things. For example, imagine yourself sitting at a table in a seminar room with a sheet of paper in front of you. Another sheet is hanging on a pinboard on the wall and a third one can be seen on an advertising column outside the window. You can clearly read everything that is on the sheet directly in front of you while you can only read the headlines of the one on the pinboard and for the one on the advertising column, you will probably not be able to read anything but at least see that there is something written on it. A technique that utilizes this concept is the FishEye technique. [27][12]

In 1986 George Furnas experimented on this and reached the conclusion that people rate the information as most important that is either close to them or has a great a priority importance. This result led to the idea of Generalized Fisheye Views which Furnas applied to data that could be hierarchically structured in a tree graph (e.g. C-code, manuals, file systems). Furnas was the first one to implement this concept based on a so-called 'Degree Of Interest-Function' (DOI-function). This function assigns a value to each data item representing its importance. This value is based on a priori-importance of an item and its distance to the current user focus. [27][12]

The following DOI function was used to create the tree-structures:

\[ DOI(x,y) = API(x) - D(x,y) \]

This function means the following: Given the current focus y, the importance on a node x is the priori-importance of x minus the
distance to the current focus. When conducting a user study on the FishEye concept compared to the standard flat view, he discovered that the participants performance was significantly higher when using the Fisheye technique. To prevent the participants from answering on the basis of prior knowledge, the experiment consisted of navigating through a botanical taxonomy of the class of Dicotyledons. Since this introduction of the fisheye views, they have been applied to a variety of different fields and kinds of data such as a calendar application, web browser and menus. [27]

Bederson’s Fisheye Menu was created due to the fact that most people only have 35 items in the favorite list of their web browser, and he wanted to establish a way to make the selection process faster and easier. A fisheye menu is a non-hierarchical menu in which the area around the current mouse cursor position is shown in normal size while the menu entries that are further away are shown in a linearly decreasing font size. For easier navigation, the first letter of the corresponding menu entries is shown in a fixed size at a fixed position in a column left of the menu. The focus of the menu can be extended by moving the mouse cursor to the right part of the menu while it is used for navigation if it is inside the left part of the menu. [27][1]

After performing a user study on this menu technique, Bederson found that most users preferred fisheye menus for browsing tasks (i.e. the user is just looking for something he might be interested in) and hierarchical menus for goal-directed tasks (i.e. the user knows what he is looking for). When comparing the time used for goal-directed tasks in a fisheye menu and a standard hierarchical the fisheye menu was outperformed. He concluded that the fisheye menu could be useful in some scenarios and for some uses, and that it should be offered as an alternative way of navigating through menus. [27][1]

Fisheye views and similar techniques provide an interesting possibility for showing information in context and viewing large data structures. The general approach is very flexible and can be applied to a lot of different fields. For cellular phones this technique could be used in several ways, such as a way to navigate through the phone menu system or to enhance the reading experience by enlarging the currently focused text. But this technique is not useful for every application and not all users feel comfortable with it. Some users even get confused when navigating with a fisheye view. Further research will probably show where the fisheye views can be useful. [27][22]
3.2.4 Event Horizon

Limited computing power, poor screen resolution, and strict memory constraints make it difficult to utilize more advanced techniques such as zooming, scaling, or infinitely large virtual displays. Further, many of the small devices are generally intended to be manipulated with one hand only, imposing limitations on the input methods that can be used.

The Event Horizon technique is based on an infinite two dimensional space that can store all kinds of objects such as text documents, images or icons. The desktop area is circular and is centered around a black hole positioned in the middle of the screen. The only way to manipulate the desktop area is to compress or expand it. Compress means that all objects moves radially towards the circular event horizon (the black hole), and when the objects get close enough they get sucked in and disappear from the screen. The expand command achieves the opposite, objects move out from the
event horizon. Objects that already are positioned at the edge of the screen area move past the edge of the screen and are no longer visible. To be retrieved again the desktop area is compressed.

![Diagram showing desktop area being compressed](image)

Figure 3: Left: Desktop area being compressed. Right: Desktop area being expanded.

The proposed benefits of event horizon are the unlimited number of objects that can be stored, and manipulated on a virtually large but physically very limited screen. In traditional displays the user often has to scroll the display in two different directions (left/right and up/down) in order to find all the objects, while the Event Horizon easily can be handled with a one-dimensional scrolling system, thus only needing one hand to control the system. Unlike traditional one-dimensionally scrolling displays such as list views, the user still has the possibility of utilizing spatial memory and capitalizing on pragmatic and incidental cues for retrieval, such as 'I left it over there', 'it was near the memo that I wrote last week', or 'it was the topmost object on the screen'. The system can be organized in many different ways, for example, the older the file the deeper into the system.

From the implementation viewpoint an important advantage of the proposed approach is that its implementation does not require much computing power. For instance, unlike with zooming displays, there is no need for scalable fonts or other scaling operations.

This user interface concept has been implemented on a Palm. The implementation was tested by a number of users in everyday life for several months. Although there were some problems with the system the overall impression of the users and the developers was that it worked well. [22]

### 3.3 Projecting the screen

A more futuristic approach on solving the problem with a small working area consists of projecting the screen contents on a flat
object such as a wall or a desk. Using this technique, the screen contents can be filled with lots of more objects to work with and enable more than one user to view the contents of the screen. This approach is more concentrated on a higher performing hardware instead of a smarter software. Currently the hardware on mobile devices is not effective enough to manage this kind of task. It would require a lot of memory to handle a big desktop area and enough power to drive a projector. But a few years from now this might be something that the mobile devices will utilize. There are a lot of examples where this feature could have its place, like showing a bunch of friends received pictures and displaying a short movie on the wall.[6]

Projecting something from the mobile device can also be used for easier input methods. A new approach in this field is the Canesta projection keyboard. The Canesta systems projects a keyboard on a flat surface, allowing the user to type in text the same way he would with a conventional keyboard. The recognition of the input is controlled via an IR emitter and sensor. The keyboard pattern emitted is not fixed, the user can for example change the language used on the keyboard. As the components of the keyboard can be miniaturized, the size of a device using the Canesta input technology can be very small.

![Figure 4: Canesta system projecting a keyboard.](image)

When a user study was made on the Canesta Keyboard, it was found that it was 1.7 times faster than the thumb keyboard used in many mobile devices. The only input method that proved to be faster was the classical mechanical keyboard, being 1.39 times faster
than Canesta. When measuring user satisfaction it was noticed that experienced users reported a lower satisfaction level than the more inexperienced ones. This could be explained by the fact that experienced users seldom look down on their fingers while typing, and since there are no physical borders for the keys, this makes it easy to drift off from the correct position. Another problem is that most kind of projecting techniques require a flat surface to project the pattern on. Nevertheless the Canesta keyboard enhances the text input and requires very little space in the device, making it a potential feature to establish in mobile devices such as the cellular phone. [27]

Another similar projection technique is to use a stationary projector and via a wireless network transfer the mobile device screen data to it. This approach also needs a big enhancement of both mobile devices and the projectors, mainly directed at the area of ubiquitous computing. [5][17][27]

3.4 Summary

A comparison on Fisheye, Zoom and virtually large displays using panning was done by Carl Gutwin and Chris Fedak. There are five main findings from the study that can be used in designing such systems[15]:

- even at its best, navigating on the small screen is considerably slower than on the normal screen.

- overviews of the entire interface are valuable, both because the global context allows faster navigation and because many interactions can be carried out at the overview level.

- fisheye views and two-level zoom systems are both effective techniques for navigating interfaces, particularly if targeting (in the fisheye) and viewswitching (in the two-level zoom) can be improved.

- for tasks where movements are large or events happen elsewhere on the screen, panning strategies perform poorly and are disliked by users.

- users like the two-level zoom, even when its performance is less than optimal.

The above text shows us that there are a lot of different techniques for enhancing the user experience with a graphical interface displayed on a limited screen area. The software exists and is in some cases being evaluated and used on a daily bases. Evidently, a
better GUI to manage mobile devices is needed. This statement is even more true when it comes to mobile phones that usually have a smaller physical screen area than PADs and similar wearable computers. A typical screen resolution for a mobile phone bought today can be around 200x200 pixels. In this small area the user must be able to manage files, look at pictures or movies, listen to music and write smaller documents in a convenient way. While one can say that the technique used today, with the normal GUI building blocks, works ok, this should not be a reason why new more adapted techniques are not used. This leads one to wonder why any of the above mentioned approaches are not available in the latest mobile products. One reason might be that these techniques still are too resource demanding, leaving the GUI developers with no other choice than to use the standard GUI building blocks and features. One can also assume that establishing a new way of graphical user interaction is quite a risk, most people are used to the GUI used today since it exists in the PC and it might be confusing to learn a new way to interact, making the end-user avoid these kind of techniques. But some of the above mentioned techniques will probably be seen in the future development of GUIs.

The UI-toolkits examined in this master thesis did not utilize any of these new techniques, they instead use the GUI building blocks known from the PC.
4 Platform and UI-toolkit introduction

This chapter is an introduction to the Ericsson Mobile Platform and the candidate UI-toolkits.

4.1 Getting to know the Ericsson U100 Platform

The platform software consists of different functional domains in a layered architecture. Examples of these functional domains are the network access, which provides access to and services for the radio network, and the operation services, which provides basic operating system and storage support. All platform functionality is accessed through an extensive API called Open Platform API (OPA). OPA is a part of the middleware domain that provides the interface towards the platform functionality. Figure 5 shows an overview of the system from the software perspective. [11][9]

![Platform Diagram](image)

**Figure 5: The Platform**

As seen in the figure, three different entities can be developed in the application software:

- **Application** An Application is a separate executable entity, owning at least one thread.
• **Utility** A Utility is a flexible entity that can be used to provide simpler services and to provide adaptation for legacy customer code.

• **Plug-in** A Plug-in is used to extend the OPA services while behaving like an OPA service from the client perspective.

The UI-toolkit chosen to be ported to the platform would be inserted as a utility for the application software. It would enable the use of different graphical objects without adding an extra layer to the platform. From the programmer's point of view, it would look and behave like an OPA-service. [7][8]

![Figure 6: Left: Functional (implementation) view. Right: Logical view (seen by programmer).](image)

### 4.1.1 OPA

OPA structures its services into categories and subcategories. Inside a subcategory, the services are provided by components based on ECM, the Ericsson Component Model. These components are structured in two layers, the manager and object layer.

The manager layer consists of a manager component. This component is the entry to the functionality of the sub-category and acts mainly as a creator of the underlying component instances (objects).

The object layer consists of components organized in a hierarchy. From the manager instance it is possible to create component instances, also denoted objects, according to the creation hierarchy and the object model used in this sub-category. The objects under the manager are created using dedicated methods in the parent object.

In order to clarify the concepts of the OPA structure, figure 7
shows a generic, fictional category in OPA; category A, which contains two sub-categories, X Management and Y Management. The CManagerX is the basic component in the sub-category and the

entry to the sub-category functionality. The manager acts mainly as a factory/creator for the underlying components in the object layer.

Instances of the components in the object layer are referred to as objects, in order to reflect the object-based paradigm used in OPA. Normally, an object, named [X], is implemented in the ECM component C[X], and provides the incoming interface I[X] and, possibly, the ICB[X] outgoing interface (to be implemented by the client

Figure 7: OPA services structure
through use of the \([X] \) interface). Outgoing interfaces in the OPA objects are used to deliver results from asynchronous services.

X1 and X4 objects are created by using the IManagerX interface of a ManagerX object, while X2 and X3 objects are created by using the IX1 interface of an X1 object. [9]

### 4.1.2 ECM Technology

ECM is an Ericsson Mobile Platforms standard for how the application software entities and the OPA services are implemented. ECM defines software components that interact with each other via defined interfaces. All published interfaces specify a contract and this contract specifies both the syntax (the names of methods and their parameters) and the semantics (the behavior of methods and what functionality they will provide) of the interface. In order to use a service in a component (that is, accessed through the component’s interface), a component instance has to be created. A component instance is a run-time manifestation of a component. When the component instance is to be created, a CID and IID has to be supplied. This CID and IID ensures uniqueness of the component, meaning that all parts agree on what syntax and semantics to expect from an interface.

The interfaces are designed to support an object-based, dynamically linked, interface-versioned system. As illustrated in figure 8, double indirect pointers are used to access the functionality of the component through a v-table. This double indirection is used to facilitate the dynamic linkage and the association of state data for the component instance. [7]

IRoot is a base interface for the ECM standard and is inherited by all interfaces. This interface contains methods for basic life cycle control and for retrieving further interfaces in a component. Other basic interfaces in the system are the IApplication interface, a mandatory interface for all applications, the IThread interface, used when creating sub-threads inside an application, and the IPlugin-Handler interface, used by a utility/plug-in to override the message mode of the calling entity when providing asynchronous services based on OPA. [7]

### 4.1.3 Platform communication mechanisms

The handling of messages is of central importance, both when using OPA services and for application software communication. The OPA message handling provides a powerful solution for handling messages in the system. When using OPA services, messages that an
application software entity has to deal with arise from results/events from asynchronous services. These results are delivered to the application software entity through an outgoing interface in OPA. This outgoing interface is defined by the OPA service, but is implemented in the application software entity that uses the OPA service. OPA supports two modes for handling messages:

- **Full Message Mode** Allows control of the message-loop to be taken and dedicated message handling methods are used to retrieve the messages directly.

- **Callback Mode** Hides the message-loop and provides a higher level of support, where the message information is directly routed to dedicated methods.

Each method in an outgoing interface of an OPA service defines a corresponding message. If the client uses Callback mode, the result will be delivered in a dedicated method in the outgoing interface. If the client uses Full Message mode, the result is delivered as a dedicated message. [7]
4.1.4 Platform text handling

The platform text handling consists of several plug-ins that effectively handle all text related features in the platform. These plug-ins are to be used by the UI-toolkit, thus they are a part of the porting work towards platform functionality. The text database plug-in introduces the concept of text id’s representing text strings in a database. By using the text id’s instead of hard coded strings, applications can easily change language at runtime. It is also an effective way of saving memory when the same strings are used in several different applications. The text layout plug-in contains different methods for making text layout and presenting the result on a canvas. There is also a plug-in for input methods, it contains a set of rules for translating user input to characters and words. [7][8]

4.2 The UI-toolkits

EMP had performed a marketing survey and found some interesting UI-toolkits to be examined. These UI-toolkits had to fulfill a set of requirements to be candidates for the porting part:

- **Application framework** An application framework that allows each application to run in its own thread of execution is needed. It will also have to handle the presentation of each application in a windowing system, and communication between applications through events as well as user input from different kinds of input devices.

- **Compatibility** This particularly means that the code must compile with the IAR Embedded C++ compiler. The toolkit should also have support for a MIDP 2.0 environment.

- **Memory usage** A footprint of more than one megabyte would be unacceptable.

- **Performance** This is very hard to put a number on, but any lower performance than EMP UI-toolkit will not be accepted.

- **Graphical object usage** The graphical objects used by normal C/C++ applications should, if possible, be the same as the ones used by Java applications. This would give the applications the same look-and-feel regardless of the programming language used, thus giving the end-user an easier experience with the GUI.

- **User attention time** The time between the user has made an input until some visual object notifies the user that the input has been made should be below 200 msec.
4.2.1 Trigenix

Trigenix is an end-to-end user interface solution which manages updates to the user interface of mobile devices. Trigenix is built up by 3 main components:

- **Trigplayer** is the core component of Trigenix. It runs as an application on the mobile handset displaying and managing the user interfaces called Trigs. These Trigs are customized and branded UI’s that can be downloaded over the air. Besides Trigs, different Triglets allow areas of the phone (for example ring tones, menus, images) to be replaced without effecting the rest of the Trig.

- **Trigserver** is a Java 2 server package running on a remote server and enables the downloading of Trigs and Triglets over the air.

- **Trigbuilder** is a GUI based tool for Windows that enables Trigs and Triglets to be developed and modified by UI-designers.

![Trigenix structure](image)

**Figure 9: Trigenix structure**

Trigenix uses industry-standard XML mark-up language adapted to the requirements of devices with small screen areas, limited memory.
and limited processing power. The Trigenix XML renderer’s features are "cherry picked" from less compact client languages such as HTML and JavaScript, resulting in a UI language that combines functionality with efficiency of speed, memory and power. [23][24]

**System Requirements**

Trigplayers memory consumption is barely 100KB and it uses the PNG format for efficient storage. Java MIDP 2.0 environment is required to make Trigplayer run. The Trigserver runs on any Java 2 Enterprise Edition (J2EE) VM with a BEA Weblogic application server. It also requires a connection to a JDBC-compliant database. [23][24]

**Pros And Cons**

The positive aspects of Trigenix is the low memory consumption and the dynamic GUI. Thanks to the XML written Trigs, the look-and-feel can be changed in many ways, only leaving the designer’s imagination as the limit. A dynamic GUI is something that probably will become a standard in mobile devices. The negative aspects is that Java is still slower than C/C++, but this might not matter much in the future as mobile devices will increase both CPU and memory performance. Since Trigplayer reads XML-files to render the GUI, and do not offer any external API for the GUI objects, this becomes a problem for Java applications that require a MIDP 2.0 compatible environment. The Java MIDP 2.0 environment defines a set of standard GUI objects to use, and Trigenix does not seem to offer this.

**4.2.2 QT/Embedded**

Qt/Embedded is a port of the Qt C++ API, a toolkit for GUI and application development widely used in Linux environments (such as KDE). The Qt C++ toolkit has been widely used for creating commercial applications since 1995. It is also used by enterprises as diverse as Sharp, IBM and NASA which proves its reliability and efficiency. The main difference between Qt/Embedded and Qt/X11 is that Qt/Embedded writes directly to the frame-buffer, eliminating the need for the X Window System and thus has a smaller system requirement, i.e. lower storage and memory footprints. The footprint of the library can be further reduced by compiling out unused features. Since Qt/Embedded has been around a while, it is delivered with a lot of common applications such as a phone book, calendar
and calculator. The code is designed so that adding or creating different plug-ins such as database drivers or image converters is easy. Qt Designer, a component of Qt/Embedded, is the tool for creating the user interface design in applications. It is easy to change the look-and-feel and add customized widgets. [26][25]

**System Requirements**

Qt/Embedded is available for all processors supported by Linux that have a C++ compiler. It provides over 200 configurable features, resulting in libraries varying in size between 700 KB and 5000 KB, all depending on the needs of the application. Most libraries end up between 1500 KB and 4000 KB. For an adequate performance a 100-150MHz MIPS/ARM compatible CPU is required. [26][25]

**Pros And Cons**

The positive aspects of Qt/Embedded is that it is a widely used system ensuring reliability and a GUI that supports MIDP 2.0. It is highly modular and has a built in support for touch-screen. The negative aspects is that Qt/Embedded has a big memory footprint and seems to be very fixed against Linux Embedded environment. It also takes advantage of the multiple inheritance that C++ enables, and this is not yet supported by the IAR Embedded C++ compiler.

4.2.3 Peg

PEG stands for Portable Embedded GUI and is a library and development tool designed specifically for embedded systems. The PEG class library provides building blocks for a big variety of different graphical user interface. In addition to the class library, PEG also provides tools needed to construct a custom graphical interface. This includes utilities for generating graphical fonts, optimizing and compressing graphical images and PegWindowBuilder, a rapid prototyping and design tool used to create PEG windows and dialogs and generate the source code needed.

PEG is written in C++ and the default appearance is almost identical to Windows look-and-feel, which makes it easy for Windows programmers to get acquainted to it. The default appearance can be enhanced or simplified at compile time. The memory footprint is small since it has a heavy reliance on C++ inheritance and uses configuration flags that minimizes the project and includes only what is necessary. [18][19]
System Requirements

The memory footprint for a typical full-featured PEG GUI requires 100KB of code space, 4KB of stack, and 8KB of dynamic memory. This can be reduced a lot by removing objects and services not used. The minimum footprint for PEG occupies 50KB of code space, 4KB of stack, and 2KB of dynamic memory. For an adequate performance is it recommended to have a a 16-bit or better CPU running at 16 MHz or higher. For applications needing very high resolution or color depth, the CPU performance will need to be increased to achieve a responsive graphical display. PEG is supported by all major CPU’s, RTOS and compilers and conforms to embedded C++ standards. [18][19]

Pros And Cons

The positive aspects of PEG is that the memory consumption is small and the scalability is high. It supports the graphical interfaces defined MIDP 2.0 and it has a mouse support that is easily converted into a touch-screen support. It is used by a variety of enterprises on different embedded systems with good results, which proves its stability, usability and efficient coding. The negative aspects is that the GUI is static and the default look-and-feel is edgy and a lot like the look-and-feel from Window 3.11. A static GUI seems a bit out of date since many new cellular phones enables a dynamic GUI that gives a more personal look-and-feel for the end user.

4.2.4 Summary

A summary of the examined toolkits is presented in the table below.

<table>
<thead>
<tr>
<th>Feature</th>
<th>PEG</th>
<th>Trigenix</th>
<th>Qt/Embedded</th>
</tr>
</thead>
<tbody>
<tr>
<td>Footprint</td>
<td>100KB</td>
<td>100KB</td>
<td>700KB</td>
</tr>
<tr>
<td>Compilable</td>
<td>Yes</td>
<td>Maybe</td>
<td>No</td>
</tr>
<tr>
<td>MIDP 2.0</td>
<td>Yes</td>
<td>Maybe</td>
<td>Yes</td>
</tr>
<tr>
<td>Touch Screen</td>
<td>Yes (via mouse)</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Dynamic GUI</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Widely Used</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>GUI Look</td>
<td>Old (Win 3.11)</td>
<td>Excellent</td>
<td>Good</td>
</tr>
</tbody>
</table>

Table 1: Toolkits summary.
5 Realization

This chapter handles the actual porting and measurement strategies.

5.1 Choosing a toolkit

Of the three examined UI-toolkits the best fit for the platform integration and evaluation is PEG. It has a low footprint, supports MIDP 2.0, has mouse support, is widely used and has a high scalability. Qt/Embedded had a very high footprint compared to PEG, but the main thing that removes it as a candidate is that Qt/Embedded uses multiple inheritance. This is not supported by the EC++ standards the IAR C++ compiler follows. To rewrite Qt/Embedded without the multiple inheritance is, if even possible, too much work for this project. Trigenix on the other hand is a very good UI-toolkit. The dynamic UI is a great feature and the footprint is small enough. The main problem with Trigenix is that it might not have external MIDP 2.0 API’s that can be used by Java applications. One possible way to solve this matter could be to make a MIDP layer that would generates XML-files usable by Trigplayer. But this would, if even possible, not be an effective solution. Several emails for more information on this matter and requests for an evaluation version was sent to Trigenix without any response, thus leaving no other choice than to abandon it.

5.2 Porting PEG

When this kind of project is to be dealt with, a good porting strategy is needed to make it efficient. There are a lot of articles and smaller books regarding porting strategies, but none of them found was directly translatable into this kind of project. They mainly dealt with strategies of porting C to C++ or C++ to Java and such. This resulted in our own developed strategy:

- Analyze the systems
- Compile and link
- Port
- Create test applications
- Benchmark
5.2.1 Analyzing the systems

To accomplish an effective port with a thin layer, it is important that a deep understanding of the system functionality is achieved. A good way to accomplish this is by reading the manuals, running sample applications and debugging the code.

PEG

PEG is delivered with a lot of Windows compatible demo applications. Many of these were tested, debugged and altered to get an understanding on how the system worked beneath the surface. PEG had three objects that were of interest for this project:

- **PegPresentationManager** PegPresentationManager keeps track of all of the windows and sub-objects present on the display. In addition, PegPresentationManager keeps track of which object has the input focus (i.e. which object should receive user input such as keyboard input), and which objects are on top of other objects. When something gets added to the PegPresentationManager, it will get drawn on the screen.

- **PegScreen** PegScreen is the PEG class that provides the drawing primitives used by the individual PEG objects to draw themselves on the display device. PegScreen will do the calculating of where objects should reside on the screen. When the actual drawing takes place, PegScreen calls a derived PegScreen object. Several templates of this object, adapted for different screen hardware, was included in PEG. It was realized that the methods in this object where the most efficient place where the actual adaption layer to OPA was to be implemented.

- **PegMessageQueue** PegMessageQueue is a simple encapsulated FIFO message queue with member functions for queue management. The messages placed in PegMessageQueue are the driving force behind the graphical interface. These messages contain notifications and commands which cause the graphical objects to redraw themselves, remove themselves from the screen, resize themselves, or perform any number of various other tasks. Messages can also be user-defined.

The customizibility of objects in PEG consists of changing size, colors and adding 3D effects. PEG also benefits from the fact that all graphical objects are derived from one base class called PegThing, making it easy to derive customized objects. The include file pconfig.hpp contain several settings for PEG. These settings determine the target environment, along with other options such as input
device drivers, screen drivers, screen resolution and a clipping algorithm, making it easy to scale down and adapt to the system in use.[19]

Making a small test application in PEG is easy and similar to working in other Windows programming environments. The following code snippet displays a raised PegTextBox with the text 'Hello World!'.

```csharp
PegTextBox *pTextBox;
PegRect Rect;
Rect.Set(0,0,176,220);
pTextBox = new PegTextBox(Rect, 0, FF-RAISED,"Hello World!", 15);
pPresent->Center(pTextBox); //Centers TextBox on screen
pPresent->Add(pTextBox);  //Adds TextBox to PegPresentationManager
```

EMP Platform

To make a simple test application for the EMP platform, five interfaces where needed:

- **Display Interface** This component is needed when creating a Display Session which is used for accessing the screen and creating a window interface.

- **Window Interface** This interface delivers asynchronous results from services in the display management sub-category in the GUI category, such as user input on the keyboard or requests for drawing on the screen. This also creates a rectangle window in which the canvas interface can draw.

- **Canvas interface** ICanvas is an interface that provides methods for drawing into an off-screen canvas (memory canvas) or a canvas connected to a window. The contents of an off screen canvas can later be displayed in a window. The methods are relatively low-level and can be used to draw lines, rectangles, ellipses, and text. Drawing parameters, such as drawing color and line width, are collected under the objects Pen, Brush, and Font. Pen is used to draw lines and outlines, Brush to fill interior areas, and Font to draw text.

- **Pen interface** Since this is to be a simple test application only then Pen interface is needed. The Pen manages drawing outlines or lines on the canvas.

- **Application Interface** This interface have methods for starting and stopping the application and is the base interface for applications.
With these five interfaces the application was ready for handling user input and drawing on the screen. Next step was to try to include the PEG UI-toolkit in the test application.

5.2.2 Compile and link

When building the U100 platform one can specify if it should be built for the test cellular called S7 (ARM CPU) or the Windows compatible phone simulator called Moses. Building for Moses gives the advantage of a (10 minutes) shorter compilation time and there is no need to upload the project to the phone. The platform also enables an easy way to temporarily add files to the project via a file called Descextra.cfg. In this file one types in a filename and path to get something included in the build. The file also enables settings for special compile options for certain files. Since all the PEG graphical objects reside in different files, this made it easy to make a minimal build that only included the required objects that makes PEG work. To allow the platform test application to get access to the PEG objects, all needed was to include the file peg.hpp. [9]

The first problem to arrive when including PEG in the project was that PEG is written in C++ and the platform is written in C. This is a problem because when compiling C and C++ code the symbols for functions and data types will differ and the C compiler does not support C++ specific coding. Thus, a way to make a bridge between these two systems was required. By adding special compile options in the descextra.cfg, the test application could be compiled with a C++ compiler. But platform specific methods and

![Figure 10: Moses with PEG running](image-url)
data types in the application needed to be compiled in C, allowing them to communicate with the platform. By using the 'extern "C"
' command and placing all platform specific include files inside the tags, C compilation for those files was achieved. The compile-errors remaining could be removed by doing some additional type casts and conversions. For a detailed porting step-by-step list, see appendix 1.

After the system was able to compile and link on Moses, it was to be tested on S7. This implies using a different compiler and linker. The platform version used at this time was named R5, and during the time spent on this project new versions up to R8 where available. The different releases does not only include a later version of the software for the platform, they also include updated compilers and linkers.

When compiling on R5 everything worked as it should, but a major error occurred during the linking. The linker for the R5 version did apparently not support the long jumps that PEG used. The only way to bypass this problem was to upgrade to a later version of the platform that could support these jumps. After some reading and discussion it was decided to try a stable R7 version. The necessary changes where made and the project for S7 compiled and linked as it should. Instead a new problem arrived when using the R7 for Moses. Apparently all the interfaces had been changed in a way so that it would not compile the C++ code in order. However, this was solved after a few days and details on this problem can be found in the porting step-by-step appendix.

5.2.3 Porting

When the compilation was successful the next step was to write a screen driver enabling drawing on the screen. Several ideas on how this was to be accomplished where thought of. The first approach was to let PEG do all the drawing in the RAM memory area. When the drawing was done, PEG was to convert the memory area into an OPA-compatible canvas that later was to be used by our application in the OnPaint event. To be able to use an OPA-compatible canvas in PEG, a rewrite of the PegScreen was done. When the PegScreen was created, a canvas created from the window interface was supplied with it. Since the customized screen driver inherited from PegScreen, this would enable the use of the canvas in both objects. Using this method would have the advantage of letting PEG do all the work, thus making it easier to compare the EMP UI-toolkit efficiency with PEG’s. But this idea had to be abandoned after some time due to conversion problems. No way to make a smooth transfer between the memory area to an OPA-compatible canvas was found.
The similar idea of letting PEG draw directly on a canvas was also abandoned due to similar reasons.

The next and final approach was to remove all the actual drawing code from the customized screen driver and replace that with OPA drawing methods. This means that PEG was to do all the calculating of where graphical objects should reside on the screen, at which coordinates to draw and what. But as soon as the actual writing to the screen took place, the OPA methods would take over. PEG calculated the raw data and OPA methods executed the drawing of it. Using this method would remove PEG code that might have proved high efficiency, but this method would also allow performance increases to take place when methods against the hardware could be upgraded to faster ones. A new rewrite of PegScreen was done to make it support OPA methods handling fonts, images and drawing objects. When the drawing of an object was done, PEG called the

![Diagram](image_url)

Figure 11: Applications use PEG that use OPA for actual drawing

window interface method InvalidateRect. This would allow usage of the PEG clipping technique since InvalidateRect messages the application to only redraw a certain part of the screen, making it much more efficient.

Since OPA did the actual drawing on the screen, not only the PEG graphical objects had to be rewritten. This also included PEG features such as text handling, image decoding and color matching.
PEG includes full support for creating and displaying different fonts. But since the platform has a set of standard fonts, they should be used instead. There was not enough time to port the entire OPA text handling to PEG. This would require a quite large rewriting of PEG’s text handling. To still be able to produce a text output, a less time demanding way was to use text output methods from the canvas object. This required a rewrite of the PegFont structure, making it use the platform fonts instead. It required that all methods in the screen driver related to printing and erasing text had to be rewritten. Since PEG still were doing all the calculation of where to put objects, OPA had to be able to give PEG information of the font used, such as character height and width.

PEG supports encoding and decoding of jpg, gif, png and bmp images. All this code can be removed since OPA shall do the image handling. The only thing PEG needed was the size of the image, pixel width and height. The PEG image structures were changed so that OPA could get all the necessary file parameters when the image should be displayed on the screen. The colors PEG use had to be passed through a conversion method before it could be used by OPA.

PEG implements an event-driven programming at the application level. This means that for the event-driven system to work as supposed, the PEG code should never stop executing, or should be deeper entangled in the system. Since none of these options will occur, the events will be manually sent and caught. This means that, for example, when selecting an option in a menu a message will be sent to the PegMessageQueue object. If the event-system would operate as normal, this message would be automatically noticed by PEG and the correct action would be taken. Instead this message is manually fetched from the PegMessageQueue and sent to the correct method executing the action. This is perhaps not the most elegant solution, but it is probably the best way to do it when PEG is not fully integrated in the platform messaging mechanism.
5.2.4 Test applications

To demonstrate the results of the porting and to present the look and feel PEG gives, demo applications were made. Developing an application for Ericsson Mobile Platforms is a bit different compared to what one might be used to. A special definition language, the EIDL, is used to specify the components and their interfaces. Based on the EIDL files, include files and implementation skeletons are generated for the application.

By request, the demo applications were to be a dialing application and a base application (main desktop) with menus and the opportunity to display a grid of images representing different features of the phone. It was suggested to do the applications with the Vodafone branding. Developing these application consumed a lot of time, some part for the graphics that was to be involved, but the most time was used trying to make the phone dial properly. [16]

![Figure 12: Left: Grid application Right: Calling Application.](image)

5.2.5 Benchmarking

Since the look and feel of the two toolkits is very similar, the result of the benchmarking is an important issue. If the memory consumption is higher and the time to create objects is longer for PEG, there is no reason for EMP to replace the current UI-toolkit with PEG. However, this would prove that EMP has done a good work creating their UI-toolkit. During the creation of the benchmark applications, the most difficult part was to make a test that
was fair to both UI-toolkits. While PEG objects only contain the plain graphical interfaces, the EMP UI-toolkit objects contain more intelligence such as premade actions for user input on the keyboard and support for dual displays. In order to make a fair test, it was decided to construct a PEG object that was as similar to the EMP object as possible. The object of choice was a standard list with 10 text strings, including a scrollbar and softkey buttons.

![EMP UI-Toolkit](image)

![PEG Toolkit](image)

Figure 13: Left: EMP list Right: Peg list.

The actual test was constructed so that the lists was to be drawn on to and removed from the screen several times with measure points before and after respective methods. Two different methods were used: one that created several objects and removed them after it was finished creating them all, and one that created an object and then removed it before creating the next. These two test where chosen because the first test would measure how well the toolkit handles several layers of objects (Z-order), and the other would measure the actual time to create the object. One problem was that the S7 did not have enough memory to create more than 4 EMP UI-toolkit lists at the same time, and running the test on Moses would not give the correct result, only a simulated one. But with no other choice the decision was made to run the test on S7. Simulations on Moses with more objects showed that the time used only affected PEG when removing lists. The benchmark also included measuring the size in memory for the objects.

This test shows that PEG’s memory usage for the list is a lot less then the memory usage for the EMP list. Since the difference was so
<table>
<thead>
<tr>
<th>Memory/Test</th>
<th>EMP – toolkit</th>
<th>PEG</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory Consumption</td>
<td>83000 bytes</td>
<td>13000 bytes</td>
</tr>
</tbody>
</table>

Table 2: Memory results.

Big the EMP source code was examined closer to see if the difference could be explained by something that not yet was implemented in the porting project. These studies revealed that EMP used a new canvas object for each list created, and every canvas consumed 69252 bytes of memory. PEG on the other hand is ported in a way so that each application creates a canvas and does all the drawing on that canvas, reducing the memory consumption by large. This solution by EMP is however changed in a later version of the toolkit, and uses the technique the PEG port uses. If the size of the canvas object is subtracted from the total memory consumption by EMP the result is approximately 14000 bytes, only 1000 bytes more than PEG.

The next test examines how well the UI-toolkits handled objects in a Z-order. For the EMP UI-toolkit there is a huge difference when comparing the first time with the following ones. The only explanation found for these discrepancies is that after the first list has been created, the interfaces used (text database etc.) is saved in RAM by the OS for a faster access time. When all the lists has been created the time for removing the top list and redrawing the one below only takes approximately 16 msec extra.

PEG on the other hand has a stable time for creating the objects, around 250 msec. It should be noted that when examination of the creating process took place, it was revealed that PEG spent most of the time in the text handling methods. These are, as stated above, only temporary solutions that will be much more efficient when the OPA text handling is integrated into PEG. To see how much time the text handling consumed, all text strings was removed from the lists. This resulted in an average time of 100 msec, 150 msec less than the original time. The main problem when handling Z-ordered objects in PEG arrived when it was time to remove the top objects. The times in the table below are from a simulation in Moses with 10 layered objects (without text) that are to be removed. Since

<table>
<thead>
<tr>
<th>Test</th>
<th>EMP – toolkit</th>
<th>PEG</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1: Creating Object 1</td>
<td>769 msec</td>
<td>272 msec</td>
</tr>
<tr>
<td>T1: Creating Object 2</td>
<td>89 msec</td>
<td>253 msec</td>
</tr>
<tr>
<td>T1: Creating Object 3</td>
<td>75 msec</td>
<td>247 msec</td>
</tr>
<tr>
<td>T1: Creating Object 4</td>
<td>76 msec</td>
<td>248 msec</td>
</tr>
<tr>
<td>T1: Creating Object Average</td>
<td>252 msec</td>
<td>255 msec</td>
</tr>
</tbody>
</table>

Table 3: Creating layered objects.
<table>
<thead>
<tr>
<th>Test1</th>
<th>PEG</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1: Removing Object 10</td>
<td>639 msec</td>
</tr>
<tr>
<td>T1: Removing Object 9</td>
<td>579 msec</td>
</tr>
<tr>
<td>T1: Removing Object 8</td>
<td>515 msec</td>
</tr>
<tr>
<td>T1: Removing Object 7</td>
<td>435 msec</td>
</tr>
<tr>
<td>T1: Removing Object 6</td>
<td>407 msec</td>
</tr>
<tr>
<td>T1: Removing Object 5</td>
<td>299 msec</td>
</tr>
<tr>
<td>T1: Removing Object 4</td>
<td>203 msec</td>
</tr>
<tr>
<td>T1: Removing Object 3</td>
<td>155 msec</td>
</tr>
<tr>
<td>T1: Removing Object 2</td>
<td>75 msec</td>
</tr>
<tr>
<td>T1: Removing Object 1</td>
<td>1 msec</td>
</tr>
</tbody>
</table>

Table 4: Removing layered objects in PEG.

PEG is constructed in a way to fully support a Z-order architecture, it is reasonable that the more objects an application has created, the more time it will take to traverse every object to see if it is affected by a removal of the top object. But the times presented in table 3 increase by an average of 60 msec per object, and that is too much to justify the operation PEG has to do. During this time the project was under a tight time pressure and enough time for debugging and studying the code for this procedure was not afforded. It could only be assumed that either had the developers of PEG an inefficient routine for this operation, or this was a bug that arrived while porting PEG.

Test 2 is a more realistic test when it comes to normal operation of the phone. This test creates a list and then removes it before it creates the next list and so on. These times can be seen

<table>
<thead>
<tr>
<th>Test2</th>
<th>EMP – toolkit</th>
<th>PEG</th>
</tr>
</thead>
<tbody>
<tr>
<td>T2: Creating Object 1</td>
<td>824 msec</td>
<td>272 msec</td>
</tr>
<tr>
<td>T2: Creating Object 2</td>
<td>317 msec</td>
<td>253 msec</td>
</tr>
<tr>
<td>T2: Creating Object 3</td>
<td>273 msec</td>
<td>247 msec</td>
</tr>
<tr>
<td>T2: Creating Object 4</td>
<td>272 msec</td>
<td>248 msec</td>
</tr>
<tr>
<td>T2: Creating Object Average</td>
<td>421 msec</td>
<td>255 msec</td>
</tr>
<tr>
<td>T2: Removing Object 1</td>
<td>16 msec</td>
<td>1 msec</td>
</tr>
<tr>
<td>T2: Removing Object 2</td>
<td>16 msec</td>
<td>1 msec</td>
</tr>
<tr>
<td>T2: Removing Object 3</td>
<td>16 msec</td>
<td>1 msec</td>
</tr>
<tr>
<td>T2: Removing Object 4</td>
<td>16 msec</td>
<td>1 msec</td>
</tr>
<tr>
<td>T2: Removing Object Average</td>
<td>16 msec</td>
<td>1 msec</td>
</tr>
</tbody>
</table>

Table 5: Single object create/remove time.

as more realistic since a user that manages a list mostly opens one single list and finishes using it before using a second one. The EMP UI-toolkit still has a big difference when comparing the first create time to time 2,3 and 4, but the latter ones are now a lot higher compared to test 1 since the lists are removed before the next one is created. The removal time is stable at 16 msec for EMP UI-toolkit.
PEG’s creation time is similar to the first test, but is actually now less compared to the EMP UI-toolkit time. And note that these times for PEG are achieved with the inefficient text handling methods and will probably become more efficient. When it comes to removing single objects in a non-layered architecture PEG is fast and never reaches more than 1 msec.

The memory footprint in flash for each toolkit was also measured and is presented in table 5. The EMP UI-toolkit has a quite big footprint compared to PEG, though it should be noted that this includes both the standard graphical objects for normal applications and MIDP graphical objects for Java. PEG manages both these interfaces with the same building blocks, enabling a much more efficient storage. The footprint for PEG can also be reduced to an estimation of 50 Kbytes by removing unused code, hardcoded images and such. Since the footprint requirement set by EMP was a maximum of one megabyte, this leaves a lot of room for more code to enhance PEG even more.

### 5.3 Remaining integration

To make this project finalized and deliverable to customers, there are some issues that has to be solved. The integration of the OPA text handling has to be done. The text database integration will require that all hardcoded strings in PEG should be replaced with the corresponding text id from the database. The integration of the text input plug-in will require that some extra classes are created, suitable for the actual text input algorithm used. When using the eZiText algorithm for example, a list of valid text string options should be displayed for the user. The main text handling feature to integrate into PEG is the text layout plug-in. This is the plug-in that handles the actual writing to the screen. PEG needs to be given information by this plug-in to know what size to make menus and buttons containing text strings. When changing the language on the phone the string length will often change and this will require that PEG gets the new information to be able to resize the actual graphical object. The image handling of this port must be enhanced so that images are not opened and decoded both by PEG and OPA. PEG only needs the actual size of the image to be able to construct a graphical object in which the image can reside. This port of image handling only includes the GIF format. The color handling of PEG must be rewritten so that it can handle the transparency feature the
screen supports.

EMP has created a set of standard applications using the EMP UI-toolkit, and it would take a lot of unnecessary work time to reprogram all of these using the PEG toolkit. Instead an extra layer should be created. This layer will consist of all methods that the EMP UI-toolkit uses, and when a method is called, this layer will make sure that the corresponding PEG methods are called instead.

Due to the limited time and the fact that the hardware supporting touch screen could not be delivered within the time frame for this project, adding touch screen to the platform was never accomplished. The time spent with this part of the project consisted of studying the platform code and trying to figure out if and how the touch screen feature could be integrated. To make a thin port layer for PEG, the platform would have to support a callback interface for the touch screen that could handle and deliver the data required. This data should at least consist of the x and y coordinates for the actual 'touch' and 'release'. Since PEG has a built in support for mice that acts on x and y coordinates and events that signals mouse clicks, this feature should be quite easily integrated into PEG. By changing some core code the mouse support could be converted into a touch screen support, even enabling 'drag and drop' actions.

Finally there is a lot of code in PEG that is not used, such as hard-coded images and extra features not supported by OPA, that has to be removed. Also some last fine tuning and optimization of the port should be done to make PEG ready for delivery to customers.
6 Results

A major effort in this project was to achieve a port with a small footprint and a thin porting layer. The benchmarking results show that, compared to EMP UI-toolkit, a very small footprint was reached when PEG was used. This was achieved thanks to the modular way in which PEG is built, making it easy to remove unused code and avoiding duplicate functionality by replacing PEG drawing methods with the corresponding OPA methods. The porting layer between PEG and OPA is as thin as can be. When a graphical object is to be displayed on the screen, PEG will do all the calculating on where and what to draw on the screen. When the actual drawing is to take place, PEG directly calls OPA methods, thus achieving the thin layer. This layer method also allows PEG to take use of upgraded OPA methods.

EMP had also set up a list of requirements that the UI-toolkit should follow:

- **Application framework** An application framework that allows each application to run in its own thread of execution is needed. It will also have to handle the presentation of each application in a windowing system, and communication between applications through events as well as user input from different kinds of input devices.

- **Compatibility** This particularly means that the code must compile with the IAR Embedded C++ compiler. The toolkit should also have support for a MIDP 2.0 environment.

- **Memory usage** A footprint of more than one megabyte would be unacceptable.

- **Performance** This is very hard to put a number on, but any lower performance than EMP UI-toolkit will not be accepted.

- **Graphical object usage** The graphical objects used by normal C/C++ applications should, if possible, be the same used by Java applications. This would give the applications the same look-and-feel regardless of the programming language used, thus giving the end-user an easier experience with the GUI.

- **User attention time** The time between the user has made an input until some visual object notifies the user that the input has been made should be below 200 msec.

PEG supports the requirements set by the application framework. It also fulfills the compiler requirement and supports the MIDP 2.0
environment. The results from the benchmark shows that PEG easily lives up to the memory usage requirements. With a footprint of only 65KB there is a lot of free memory that can be used for enhancements and additional features. The performance results from the benchmark shows that PEG at least can perform as well as the present UI-toolkit and will probably show even better results when a full port has been reached. Since PEG supports MIDP 2.0, Java applications can use the graphical objects PEG offers, thus reaching the goal of having the same GUI for Java and native C/C++ applications. The time from user input until PEG displays a graphical object on the screen was measured to a total of 70 msec, meaning that PEG lives up to all the requirements set by EMP [10]. PEG also supports the additional feature of touch screen, giving the end-user an easier way to interact with the phone.

The fact that PEG is widely used by enterprises around the world proves that PEG is a stable UI-toolkit. Since PEG is coded in C++ it widely simplifies the usage of graphical objects, especially when it comes to constructing your own customized objects by using the inheritance feature offered by C++. Almost all of these graphical objects also offer some customization to the look and feel, such as adding 3D effects and the ability to place objects anywhere on the screen. Writing the code for the GUI is easy and someone who has programmed in a windowed environment before will find a lot of similar objects to use.

The main negative issue with PEG is the actual look and feel which can resemble to the look and feel of Windows 3.11. The GUI development in normal PCs has given users a taste of how it can look, and there should be no reason why the GUI on cellular phones could be just as good. A good look and feel is simply a reason for buying a cellular phone. The C++ code in PEG also gives a disadvantage caused by integration difficulties into the platform.
7 Conclusion

During this porting project, several things have been realized. It is not an easy task to port a toolkit like this to an existing platform. Not only does the difference of the two systems increase the complexity of the task. The EMP platform has been under development for over years by approximately 400 programmers, containing a total 25000 files. This leads to the fact that a great deal of the time has to be spent on pre studies of the platform, and in this case the commercial toolkit too. This is something that needs to be done to understand the systems and to make an efficient port. However, EMP should consider some kind of public presentation of the platform, making the learning process faster and more efficient.

Another issue that complicates the porting procedure is the request of removing all the double functionality. A removal of all methods PEG offers but OPA already has, means that the port achieves a smaller footprint and a better integration towards the platform, but increases the porting effort. The more the UI-toolkit is changed and adapted to the platform, the more effort must be spent on maintenance issues, such as updates of the UI-toolkit software. The risk of introducing new bugs increases with the amount of changed code. This is one reason to why the porting layer has to be thin and simple, and why the changes in the UI-toolkit core has to be minimized.

To meet the request of having a range of UI-toolkits to be offered to the customers, EMP should create a general API towards the applications in the platform. This API would preferably be built on the MIDP 2.0 API and then be enhanced to cover all the needs of the standard applications and demands from customers. The specification of such an API would be quite a large task. To cover all the aspects of customization and settings that the application programmer need to create efficient, reliable applications with a nice design would consume a great amount of time. This would also add an extra porting layer to the platform. However, this idea will enable EMP to offer a range of applications that can be used with any of the offered UI-toolkits, thus enhancing the platform value.

During the time spent with this thesis, many of the companies creating commercial UI-toolkits have realized that toolkits specially designed for small displays and cellular phones are needed by mobile platform developers such as EMP. This has resulted in more interesting candidates being developed or already on the market compared to when this project was started.
• **C PEG** This is an enhanced version of PEG, written in C and with a better GUI than the one ported. The fact that it is written in C will make the integration against the platform easier since two different compilers will not compile within the same file.

• **PEGASIS** Coming soon, also developed by Swell software and constructed with the goal of enabling a complete GUI framework for cellular devices.

• **Qtopia** Developed by Trolltech, this is specially designed for small mobile devices such as cellular phones. The GUI is designed to support screens with a resolution down to 176x208 pixels and is customizable by both end users and network operators.

These are a few of the new toolkits developed with a GUI mainly towards mobile devices. It is most likely that the main cellular phone development will be in a direction towards more advanced phones that can act as fully functional wearable computers. This and bigger screens will require a good GUI framework with effective graphical objects. For EMP to be a competitive force in the mobile platform business they should be able to deliver a range of different UI-toolkits to their customers. This is where PEG is recommended to be offered as a light, small footprint, alternative for customers with this demand. But for the customers with a demand of a more graphically advanced and dynamic UI, EMP is also recommended to carefully study and evaluate the new UI-toolkits that are being developed for mobile devices.

### 8 Acknowledgments

We would like to thank, in no particular order, Paul Asterland, Robert Lindh, Mattias Lundquist, Erika Prymus, Annika Hermansson, Lars-Erik Janlert, Jürgen Börstler and all the others who helped us during this thesis.
References


   Last Visited: 2003-09-10.

   Last Visited: 2003-09-10.

   Last Visited: 2003-09-10.

[27] WOLF, M., AND BARDOS, L. Interfaces for handheld devices.  
   Part of the seminar: Post-Desktop User Interfaces at Media  
A Appendix 1

This is a step by step guide on how to include PEG into the platform. It can also be used as a guideline for how to include any C++ add-ons.

1. In the file Desoxextra.cfg, include all the necessary source and header files needed for the UI-toolkit to run. The files to include for PEG can be found in appendix 2. Since the file names end with .cpp and .hpp the compiler will automatically understand that these files should be compiled with a C++ compiler.

2. To enable usage of PEG in a platform application, the file peg.hpp shall be included. This will require that the application is compiled in C++. Editing the Desoxextra.cfg, one can do this by adding ’/Tp’ for Moses or ’-ec++’ for the S7 under the source file options, ie ’TestApp.c + WIN32=(/Tp)’.

3. While the application is compiled in C++, the platform specific include files still need to be compiled in C. This can be achieved by placing all platform include files within a ’extern C’ command. A type definition of the component factory for the application must also be inside the brackets for the command.

4. Inside the application constructor, all the callback interfaces virtual table pointers has to be changed, this example is for the IWindow interface:

'pThis->myICBWindow.pVT = (void*)&Testapp_ICBWindow_vi;
changed to
'pThis->myICBWindow.pVT = &TestApp_ICBWindow_vi;
Thus removing the (void *)

5. When compiling for Moses and using a platform version R7 or later all used callback interfaces defined in the .idl file has to be manipulated in the header files for this interface. Where the actual virtual table struct for the interface is defined there is a ’const TUuid* const pIID;’ at the top of the struct. This has to be changed to ’const TUuid* pIID;’ to make the compile work.

After these 5 steps has been followed, the port of PEG’s GUI is ready to be used in any application.
B Appendix 2

This appendix contains a list of all the files used by PEG in this project.

pbitmaps.cpp
pbutton.cpp
picon.cpp
pliteral.cpp
pmenfont.cpp
pmessage.cpp
pmenu.cpp
pmsgwin.cpp
ppresent.cpp
pprompt.cpp
prect.cpp
pscreen.cpp
pscroll.cpp
psysfont.cpp
pthing.cpp
ptitle.cpp
ptxtthng.cpp
ptextbox.cpp
peditbox.cpp
pdialog.cpp
plist.cpp
pcombo.cpp
pbig5map.cpp
pbitmaps.cpp
pblight.cpp
pbmpconv.cpp
pgifconv.cpp
pimgconv.cpp
pdccwin.cpp
pdccbtn.cpp
pwindow.cpp
pal256.cpp
opascr16.cpp
wbstring.cpp