An Event Based GUI Programming Toolkit for Embedded System

Conference Paper · December 2010

DOI: 10.1109/APSCC.2010.115 · Source: DBLP

6 authors, including:

Congfeng Jiang
Hangzhou Dianzi University
65 PUBLICATIONS **264** CITATIONS

SEE PROFILE

An Event Based GUI Programming Toolkit for Embedded System

Xu Hu,Congfeng Jiang,Wei Zhang, Jilin Zhang, Ritai Yu, Changping Lv
Grid and Service Computing Technology Lab
Hangzhou Dianzi University
Hangzhou 310037, China
E-mail: dom343@gmail.com, cjiang@hdu.edu.cn, magherozhw@gmail.com, {jilin. zhang,yrt,lcp1965}@hdu.edu.cn

***Abstract*—Due to various differences in hardware architectures of devices in ubiquitous computing systems, portability and platform-independency become the main challenge for graphics programming in system design. In this paper, we propose an adaptive user interface programming toolkit for system design in ubiquitous computing environment. The toolkit leverages an existing system software infrastructure, making the application programming straightforward and platform independent. This proposed toolkit can be divided into two parts: the first part consists of open source cross-platform graphics libraries which are encapsulated into the platform dependent part of backend library for interacting with specific system. While another one, called core library, is responsible for the functions of control logics, graphics drawing and backend management. To demonstrate the practical use of this toolkit and its portability, a case study is provided for demonstration. The test results on three different embedded systems show its good adaptability on multi-platforms.**

Keywords-embedded system, graphics user interface, platform abstraction, event handling model

I. INTRODUCTION

In the ubiquitous intelligence environments, users usually need to control various devices such as TV, air conditioner, refrigerator and even electronic watch simultaneously. To obtain the services that a user needs under the specific situation, the user should know the functions and positions of the buttons in several remote controllers. Since it is difficult to manipulate them and the number of the devices that we can control increases with the system scale [1], we get to complain these designs of user interface. For even worse, there are hundreds of manufacturers and thousands of styles in the emerging ubiquitous computing environments that we cannot force all of them fit the ubiquitous design. In addition, it is inconvenient to develop application programs in an embedded system. Unlike the PC application development, we have to conduct the cross compiling process through special compile tool. It is evident that user interface should be built on some middleware. So, a new Graphic User Interface (GUI) programming toolkit should be based on some abstraction layers [20] and keep its adaption in different platform when writing applications.

At present, there are several embedded GUI toolkits that have been developed successfully for this multi-platform use. The most used one is Qt-Embedded [7], which can be ported across desktop and embedded operating systems, with integrated development tools. But the drawback of it is the inconvenience for memory-constraint system due to its large package size. wxWidget [8] is another multi-platform GUI library, which provides applications a truly native look and feel. However, lack of object-oriented consistency in structure makes it insufficient for platform-independent system design and development. FLTK [9] also provides basic functions of GUI programming; what is more, due to tailoring into lightweight one, it is more suitable for embedded system. But it cannot give enough non-English international language support.

There are also some attempts [2,3,4] to exploit multi-platform using GUI document technology. The widget tree structure can be easily translated to a description in Extensible Marking Language (XML) or other user interface languages form. These scripts are sent to possible clients who parse it and display on the user interface according to the parameters specified in GUI documents. However practicability in programming is not good enough especially when an application is extreme complicated.

To sum up all tools and libraries mentioned above, this paper proposed and worked out a multi-platform user interface programming toolkit, named huG, for embedded software development. It extends the existing work in two important ways: 1) Mature graphics libraries are used for totally new technique to retrieve low level event and operate the graphics devices; and 2) Widget subsystem is designed for object-oriented programming.

II. RELATED WORK

In order to unify the platforms to a simple but effective way, we compared some typical libraries for event handling and graphics operations, like the libevent [10] library, the libwm [11] library, the SDL [5] library and the OpenGL library. Eventually, we decided to use SDL for event retrieving as well as basic graphics operations and we use OpenGL[6] for 3D application building. Besides, we use a subset of STL [12] to hold data structures and use the CMake [13] to generate Makefile for project automatically.

A. Multi-platform Implementation Using SDL

SDL is a cross-platform multimedia library that presents a simple interface to various platforms' graphics, sounds and input devices. For our use, we simply wrap the application programming interfaces (APIs) into class partly.

So that we can get the events of SDL, control the frame buffer in Linux system and access the DirectX in Microsoft Windows system.

But before we use this library, in order to support a specific hardware system, we have to configure it case by case. For instance, in ARM based Linux system, it only support rendering directly to the Framebuffer [25] instead of using the X Window System [24]. After that, we can build a custom dynamic link library for the specific system. Since there are a lot of extensive libraries of SDL, we also need to configure several macro switches to use them. The USE_SDL variable in configure script is used to auto-configure the dependencies for ports which use an SDL based library like `sdl_image` and `sdl_mixer`. So, we will see that extensive libraries are linked in.

B. Multi-platform Implementation Using OpenGL

Although SDL is enough to support the platform independent layer in our toolkit, we need 3D support for some special cases. As we all know, OpenGL is the foundation for high performance graphics. We can use OpenGL to cooperate with the basic SDL, making a perfect 3D user interface. For an embedded system, we can use the subset of OpenGL that is the OpenGL ES (OpenGL for Embedded Systems). But in fact, OpenGL ES is a standard that do not have an official version implementation. In our toolkit, we choose picoGL [14] for rendering which wrote by Jim Huang. The picoGL is a lightweight OpenGL subset implementation based on OpenGL ES specification with some improvements, such as Linux Frame buffer backend, fixed-point optimizations, enhanced API, etc. Unfortunately, picoGL can only be built using cross toolkit with fixed-point support. For this reason, we have to build a cross tool chain by ourselves. We use the Crosstool [15] wrote by Dan Kegel to do these work.

C. A Subset of STL-RDESTL

Considering the consistency coding, we use a lot of STL containers to implement the GUI structures instead of using data structures wrote by ourselves. Meanwhile, STL can guarantee the stability of whole system as well as the performance. However, for a portable toolkit, considering its size, we have to choose a subset of STL. Efforts have been made on putting together a small subset of the STL changed for some game development requirements called the RDESTL[12]. It provides some basic containers, a string class, and some additional container types like intrusive containers.

D. Souce Code Management Using CMake

Since we build a toolkit for multiple platforms, it is better to choose a good tool to organize our source code. CMake is a cross-platform, open-source make system used to control the software compilation process using simple platform and compiler independent configuration files. It generates native makefiles and workspaces that can be used in the compiler environment of your choice. CMake is quite sophisticated. It is possible to support complex environments requiring system configuration, pre-processor generation, code generation, and template instantiation.

III. FRAMEWORK ANALYSIS

In this paper, we consider how GUIs can be successfully disposed, not only in multiple versions, but so that it can be ported across multiple platforms and on multiple devices. Our research has identified the structure of the problem domain, and we have filled in some of the answers. The key issues [19] are that for a given program, existing or planned, it is desirable to decouple the following elements: programming language, logical control components, hardware runtime environments or virtual machines, and operating systems.

In this way, we achieve separation of the GUI specification from the program so it can be reused in other programs, possibly written in other languages; and a portability path for the whole system onto their existing, or future, operating systems, with possibly different virtual machines.

To compare with conventional GUI, a multi-platform GUI has to be built on an abstraction layer for different platforms, while the upper components have to be modified accordingly. The Fig.1 shows a typical hierarchical relationship in multi-platform environment, where the shadow zones represents the modified GUI. In this section, we would like to analyze how to separate GUI into components which can be reused. The platform abstraction layer can be treated as a middleware that it can be very flexible for different systems. The GUI component should match the interface provided by platform abstraction layer. Last, the event handling in GUI component should extract features of different event models so that it can bypass any interface of operating systems.

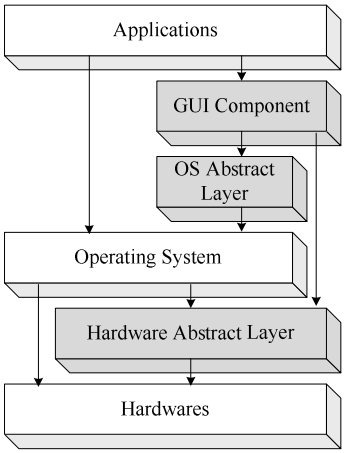

Figure 1. The hierarchical structure of multi-platform GUI

A. Platform abstraction Layer

Platform abstraction layer shields development engineers from low level hardware interfaces in order to enable the upper component to be platform independent by providing a consistent interface for interaction regardless of the operating system or data transport used. The consistent

interface is necessary given the broad variety of operating systems and data transports that exist within embedded system.

In our solution, the platform abstraction layer means operating system abstraction and hardware abstraction together. But all of abstractions refer to graphics devices and input devices. It is a very small part but related to GUI operations. Since some embedded system composed by a tiny and simple kernel and some board support package, it is necessary to control hardware related to GUI through an independent way other than operating system interfaces. However, more and more embedded operating systems have complete functions like desktop system. Thus, the operating system abstraction layer is provided for calling different system interface for hardware interactions. It is possible to organize input forms into two standard forms, i.e., mouse and keyboard because the general embedded system need not too many unusual perceptions. Nevertheless outputs of graphics are hard to unify, at least, drivers of graphics card vary from producers to products. Platform abstraction layer only creates a piece of memory space to draw that uses the simplest method to fit different systems.

Anyway, platform abstraction layer try to make the transformation as easy as possible. The whole work can be done by incept several middleware.

B. *Event Handling*

Events for GUI are the impulses that results in all of actions. Unfortunately, operating systems do not have the common realization. So an independent GUI should reorganize the event model. Actually, events can simply be divided into system event and action event while system event refers to mouse clicking, keyboard typing which generate externally, and action event follows system event for a user defined action. For the new event handling, it is possible to apply the Reactor pattern [17] which is used for I/O synchronization in network services. The reactor pattern completely separates application specific code from the reactor implementation, which means that application components can be divided into modular, reusable parts. Also, due to the synchronous calling of request handlers, it reduces blocking time to the system.

In our event model, every time the user types a character or pushes a mouse button, an event occurs and any object can be notified of the event. All it has to do is to implement the appropriate interface and to be registered as an event listener on the appropriate event source. Each event is represented by an object that gives information about the event and identifies the event source. In this context, event sources are typically components, but other kinds of objects can also be event sources. Event-handling code executes in a single thread. This ensures that each event handler will finish executing before the next executions.

C. *Widget Subsystem*

Since we pursue the lightweight and platform independent user interface, it is evident that the widget subsystem should be very succinct. Any of widgets or controls should be adaptive with all of runtime environments. In a size-constrained system, the widgets adjust their size proportionally, and even change their appearance to some planned modality. Another important problem to be addressed is what is the exact number of widgets should be customized. And what kinds of widget should be involved.

Thus, it is sufficient that a good hierarchy of this subsystem surely increases the adaption of whole system. Our design of widget inheritance structure is shown in Fig.2.

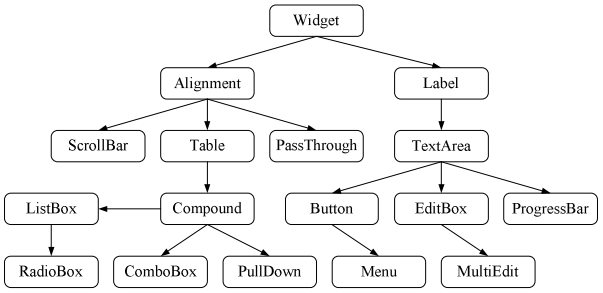

Figure 2. Widget subsystem inheritance structure

From Fig.2 we can see that the hierarchy is quite simple. When we implement these widgets, we just need five times to vertical remake the original one which is the direct parent.

IV. IMPLEMENTATION

Our implementation, named huG, consists of three modules. The core management module and widget module compose the core library, which is one of dynamic link library. And the platform abstraction module acts as another dynamic link library; the backend library. The Fig.3 shows deployment between modules of huG.

A. *Platform abstraction Module*

The platform abstraction module provides four main functions. The class `Graphics` is used for basic drawing operations, such as point, line, and color drawings, etc. It is similar to Microsoft Windows' GDI. Besides, this class holds a clipping area stack which is very important for redrawing. The clipping area in huG is a rectangle that records its position but the operations are free. The platform abstraction instance can overload these member functions. The class `Input` is used for handling mouse inputs and keyboard inputs. The class `Text` provides operations of fonts. The class `Image` and class `ImageLoader` are responsible for picture loading, rotating and displaying and so force. They are four major classes as shown in table 1.

These classes encapsulate some common interfaces of different platforms into their member functions for applications. In some degree, they are limited virtualizations from characters extracted by physical devices. They are standard interfaces to guarantee that only these member functions for application programming are used. Actually, they are empty base class indeed. A concrete class for platform interaction should give most of standard interfaces. At this version of huG, there are two native implementations have been provided by using cross-platform graphics library

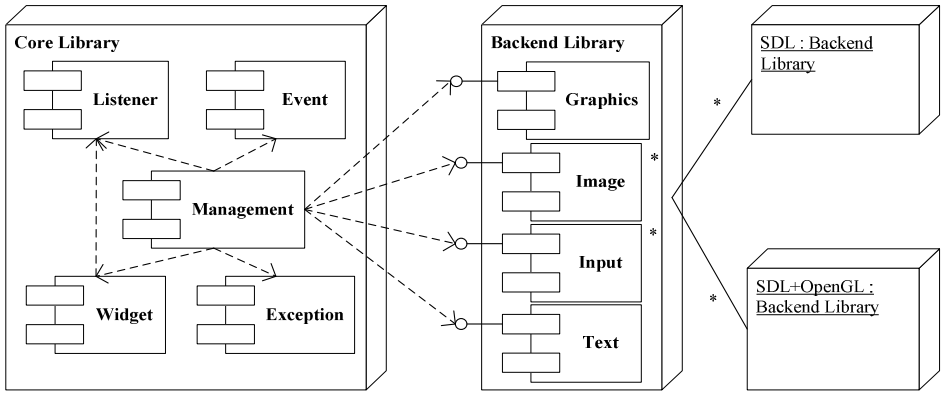

Figure 3. Deployment of core library and backend library

like SDL and OpenGL. They inherit all of base classes of platform abstraction module respectively. The SDL version is used for normal scene rendering, while SDL combined with OpenGL is used for advanced 3D scene rendering. But there are limits for some special usage cases which are not supported by SDL and OpenGL. Therefore, we have to increase additional interfaces manually.

TABLE I. MAIN FUNCTIONS OF PLATFORM ABSTRACTION

Function	Description	Belonging
Basic draw	color, draw, clip, font	Graphics
Event input	mouse event, keyboard event	Input
Image	properties, load, operate	Image
Text input	Display, delete, cursor	Text

B. Core Management Module

In fact, core management module provides not only convenience in management but also the uniform programming interfaces. These functions are shown in table 2.

TABLE II. MAIN FUNCTIONS OF CORE MANAGEMENT

Functions	Descriptions
Widget subsystem management	Get and set the top widget; drawing the widget subsystem
Graphics device abstract management	Get and set the graphics device abstract instance
Input abstract management	Get and set the input device abstract instance
Global status management	Maintains some global variables
Global listeners management	Listens global keyboard event
Event pre-process management	Retrieves input objects and distributes them
Exception management	Records where exceptions happened
Resource management	Manages the fonts and icons

For simplicity, the module dose not distinguish user operations and system affairs, but organize them into an integrated class, named `Gui`, for all global creations, registers and distributions. Every application has at least one class instance, and then builds specifics of GUI through its interfaces. Meanwhile, the running circle calls member functions of `Gui` for logic of widget system.

Besides, exception handling and resources can be included in this module, since `Gui` relies on these heavily. When coding an application, the `try-catch` structure has to be used. It is the simplest way to locate an error occurs. Therefore, this module has three classes now; the `Gui` class for mainly control, the `Exception` class for exception handling and the `Resource` class for resource requesting.

C. Widget Module

This module cooperates with platform abstraction module that fetches from input queue, forms into event and distributes to source widget.

First, the design of widget hierarchy is based on the Composite Pattern [16]. All types of widget are divided into container and standalone. The only difference between these two types of widget is that container distributes event received recursively. Fig.4 shows the static construction of widget in huG.

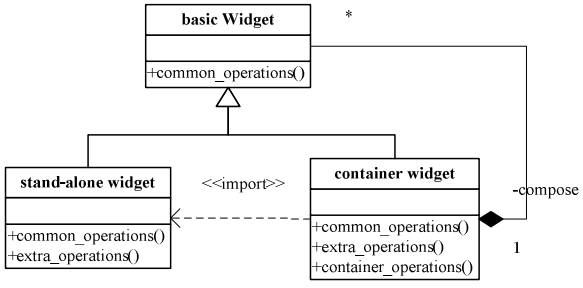

Figure 4. Static structure of widgets relationship

The huG divides events into internal ones and external ones. And external event refers to input by mouse, keyboard or other similar devices. Internal event is custom event that responses to follow external event [18]. It is important to know that every event has listeners in huG and listener also

handles event. That is, widget classes inherit one or more listeners, so that it can use member of listeners for specific

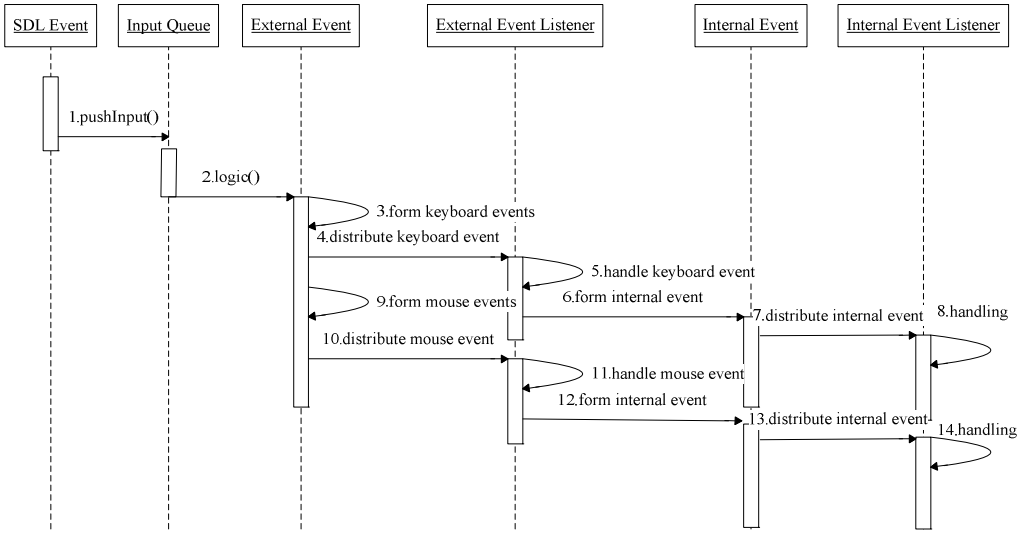

Figure 5. Time series of event handling

event handling. Therefore, it is very flexible since we can customize new listeners and add new functions.

TABLE III. LISTENERS IN HUG

Listeners	Relevant Events	Who uses
Action listener	Internal event	DropDown, TabbedArea
Container Listener	Widgets add in or remove from container	Container
Focus Listener	Focus event	Button, DropDown
Key Listener	Keyboard event	All widgets
Mouse Listener	Mouse event	All widgets
Selection Listener	An item be selected	DropDown, ListBox
Widget Listener	Widgets move, resize, hide etc.	None

Now, we use the mouse click event handling for example to demonstrate the whole event handling process. Fig.5 shows event handling time series.

a) Assuming that the application fetchs a SDL event of mouse click, then transfers it to mouse input object and stores it in the mouse input queue.

b) There is a circle used for querying periodically that retrieves an input object per queue.

c) Resolve the input object and assemble it to event of huG according to its type (here, we get a mouse click event).

d) Call mouse click preprocessing function to distribute the event to where it happened.

e) Traverse all the listeners belonging to the source widgets and trigger event handling functions.

f) Now, it will finish unless some internal events have been set. The internal listener also can be triggered as the same as external one.

Comparing to the callback mechanism, this method is easier to understand and debug. It simply holds several input queues which reorganize the system event to a uniform object. And then distribute events to source widget. If a proper listener has been set, the event handling will be triggered. Moreover, there is a hierarchy of event model which is good for filtering. Besides, it also benefits listeners reusing and customizing new widgets. At present, there are 7 types of listeners in huG as shown in table 3, and maybe increase in the future.

V. APPLICATION PROGRAMMING AND TEST

We use CMake to build the multi-platform project of huG. Now, we have binary dynamic link libraries on X86 Linux systems, X86 Microsoft Windows and ARM Linux systems. Table 4 shows sizes of core library for the release version. We can see that none of them is beyond 30KB, which proves huG a very small library.

Next, we provide a tiny tutorial to show how to program with huG. The example mainly focuses on widgets. Let us see main function first:

```
void main(int argc, char **argv)
{
    // register backend library
    hugsdl::init();
    // application initialization
    widgets::init();
    // application running
    hugsdl::run();
    // application halt
```

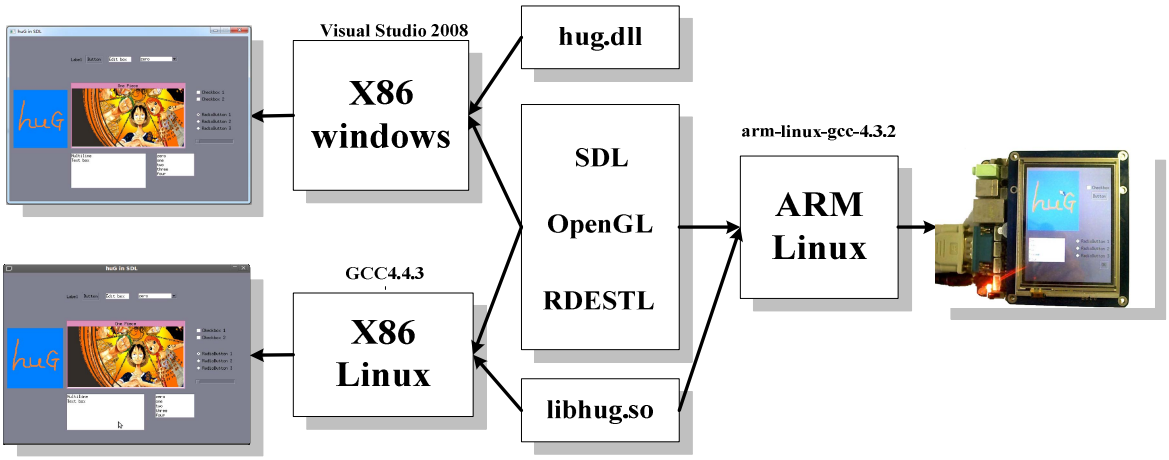

Figure 6. A case study of test process

```
    widgets::halt();
    // huG halt
    hugsdl::halt();
}
```

Here, the class *hugsdl* is the major class of this application, including registering of backend instance and main circle. So look at the init function in *hugsdl* class for registering:

```
// create the core management class
globals::gui = new gcn::Gui();
// set up graphics context using
// SDLGraphics
globals::gui->setGraphics(graphics);
// set up input class using SDL_input
globals::gui->setInput(input);
```

Here, the *gcn* is the namespace of huG. And the *gui* is an instance of core management class. These operations register a backend instance to huG, including graphics part and input part. Back to the above main function, the second sentence is initialization of application. It arranges the main window and creates several widgets. We can see how the *Button* is created:

```
// create and set root widget
top = new gcn::Container();
top->setDimension(gcn::Rectangle(0, 0,
// 640, 480));
globals::gui->setTop(top);
// add a button widget
button = new gcn::Button("Button");
top->add(button, 200, 10);
```

After that, the run function is called. Let us step into this function:

```
while(running){
    // catch user input
    SDL_Event event;
    while(SDL_PollEvent(&event)){
        // get a event from SDL
        ......
        // push the event into input
        // queue
        ......
    } // end while
    // handling the event by huG
    globals::gui->logic();
    // redraw all of GUI
    globals::gui->draw();
    // switch display cache
    SDL_Flip(screen);
} // end while
```

In this part of the codes, we can see the main circle. It is noteworthy that huG does not define constant main circle form because the original system event can be retrieved in different ways. The most important part in the main circle is the logic function and the draw function. They handle inputs and perform results respectively.

Upon now, the application events have been processed. Now we can build them to executable files in different platform. The whole test process is presented in Fig.6. It is very clearly shown that it is an extremely flexible way to build the multi-platform programs in ubiquitous computing systems. It also suggests the good portability of huG.

TABLE IV. SIZES OF CORE LIBRARIE

Core library	Runtime environment	Library size
libhug.so	X86 Linux	18KB
hug.dll	X86 Windows	22KB
libhug.so	ARM Linux	7KB

VI. SUMMARY

By investigating the portability and availability problem of embedded GUI programming, in this paper we discussed how to design and implement a lightweight and multi-platform GUI programming toolkit for embedded ubiquitous systems. We proposed a GUI toolkit-huG based on some open source cross-platform libraries. Through the excellent designing of the hierarchical framework, huG provides a

very simple way for programming and effective support for various applications. A case study is also provided and it indicates that huG performs well in practical programming in embedded systems. For future work, we are considering to realize a ubiquitous service in the work environment which combines with our toolkit to achieve a real time adaption [21, 22, 23].

ACKNOWLEDGMENTS

The funding support of this work by Natural Science Fund of China (NSFC Grant No.61003077), Technology Research and Development Program of Zhejiang Province, China (Grant No. 2009C31033, No. 2009C31046), Natural Science Fund of Zhejiang Province (Grant No.Y1090940, Y1101092, Y1101104), Hangzhou Dianzi University Startup Fund (Grant No.KYS055608109, KYS 055608058, KYS055610004) are greatly appreciated.

REFERENCES

[1] Brad Myers, Scott E. Hudson, Randy Pausch, Y Pausch, "Past, present, and future of user interface software tools," ACM Transactions on Computer Human Interaction, vol.7, March 2000, pp.3-28, doi: 10.1145/344949.344959.

[2] Guido Menkhaus, Wolfgang Pree, "User interface tailoring for multi-platform service access," Proceedings of the 7th ACM international conference on Intelligent user interfaces, 2002, pp. 208-209, doi: 10.1145/502716.502760.

[3] Jan Meskens, Jo Vermeulen, Kris Luyten, Karin Coninx, "Gummy for multi-platform user interface designs: shape me, multiply me, fix me, use me," Proceedings of the working conference on Advanced visual interfaces, ACM Press, 2008, pp. 233-240, doi: 10.1145/1385569.1385607.

[4] Fabio Paterno, Carmen Santoro, Lucio Davide Spano, "MARIA: A universal, declarative, multiple abstraction-level language for service-oriented applications in ubiquitous environments," ACM Transactions on Computer-Human Interaction (TOCHI), vol. 16, 2009, pp. Article No.19, doi: 10.1145/1614390.1614394.

[5] libSDL, http://www.libsdl.org

[6] OpenGL,http://www.opengl.org/.

[7] The Nokia Qt team, "Qt for Embedded Linux", http://doc.trolltech.com/4.4/qt-embedded.html

[8] The wxWidgets team, http://www.wxwidgets.org

[9] The FLTK team, http://www.fltk.org

[10] Niels Provos, http://monkey.org/~provos/libevent/

[11] libvm , http://code.google.com/p/libwm/

[12] rdestl, http://code.google.com/p/rdestl/

[13] CMake. http://www.cmake.org

[14] PicoGL, http://jserv.sayya.org/

[15] Building and Testing gcc/glibc cross toolchains, http://www.kegel.com/crosstool/

[16] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, Design patterns: abstraction and reuse of object-oriented design, Beijing, China Machine Press, 2007.

[17] Douglas C. Schmidt, Michael Stal, Hans Rohnert, Frank Buschmann.,Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects, 2nd edition, Beijing, Publishing House of Electronics Industry, 2007.

[18] Jasmin Blanchette, Mark Summerfield, C++ GUI Programing with Qt4, Beijing, Publishing House of Electronics Industry, 2008.

[19] Alejandro Rodriguez Ascaso, Juan Fernandez, Maria Teresa, Arrendondo, Estela Conde and Carlos Sanchez, "Ubiquitous access to a multi-platform environmental control system based in Design for all principles," Proceedings of 1st International Conference On Smart homes and health Telematics, 2003,pp.38-44.

[20] Fresco R., Marucci L., Signore O, "Adaptive hypermedia for user-centered geovisualization in multiplatform environment," Proceedings of International Conference on Hypertext and Hypermedia (Hypertext 03), ACM Press,2003, pp. 20 -21.

[21] Oleg Davidyuk, Iván Sánchez, Jon Imanol Duran and Jukka Riekki, "Autonomic composition of ubiquitous multimedia applications in REACHES," Proceedings of 7th ACM International Conference on Mobile and Ubiquitous Multimedia (MUM2008), December 3-5, 2008, pp. 105-108, doi: 10.1145/1543137.1543159.

[22] Davidyuk O, Georgantas N, Issarny V and Riekki J. MEDUSA, "A middleware for end-user composition of ubiquitous applications," Mastrogiovanni F and Chong NY (ed.), Handbook of Research on Ambient Intelligence and Smart Environments: Trends and Perspectives, IGI Global, to appear in 2010.

[23] H.-S. Park, I.-J. Song and S.-B. Cho, "Context-Adaptive User Interface in Ubiquitous Home Generated by Bayesian and Action Selection Networks," Proceedings of International Conference on Ubiquitous Intelligence and Computing (UIC'08), June 2008, pp. 158-168, doi: 10.1007/978-3-540-69293-5-14.

[24] Robert W. Scheifler, Jim Gettys., "The X window system", ACM Transactions on Graphics, vol. 5, 1986, pp.79-109, doi: 10.1145/22949.24053.

[25] Framebuffer HOWTO, available at http://www.linux-tutorial.info/modules.php?name=Howto&pagename=Framebuffer-HOWTO.