056Berlin.qxd 21.11.2000 16:51 Uhr Seite 56

Berlin - Alternative to the X Window System

DOWN WITH THE X-WALL!

TOBIAS HUNGER

Linux users are getting more and more spoilt – calls for features such as font-anti-aliasing or alpha-blending are getting louder all the time. And at the same time, of course, the system has to be high-performance – the X Window system can help only to a limited extent. The Berlin Project, with its modular, object-orientated architecture, is attempting to make a new start.

056Berlin .qxd 21.11.2000 16:51 Uhr Seite 57

More than a decade has now gone by since the development of X-Window. It is not least for this reason that this display system is very robust. Continual new demands from users can be fulfilled with the aid of expansions of the X-protocol. Despite the far-sightedness of its creators though, the X Window system is gradually coming up against its limits. 3D applications can only be used effectively with a complicated GLX/DRI mix. Smoothed character sets are still awaited. X also allows only one ToolKit-related configuration. If you play in a KDE theme, the appearance of the Gimp does not alter for a long time.

New start

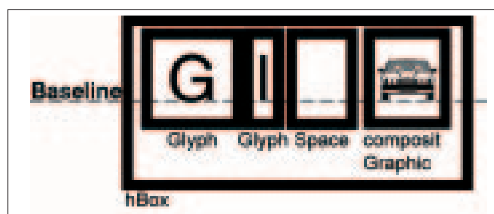
Berlin offers a central configuration of its appearance. This means that applications present a consistent "Look and Feel" at all times.

Gaming fanatics will love the rotating and transparent windows. For programmers the most interesting feature must surely be the device-independent graphics output. The best possible quality is always used, regardless of whether a monitor or a printer is employed for output. As a result, there is no need for the print routines necessary for X, or to adapt a program to different colour depths and screen resolutions.

Goals

Berlin's goal is to create an alternative to X which can cope better with modern hardware and can be more easily adapted to new input and output devices. The whole system is meant to be modular, so that the actual server is scaled by the selection of suitable modules. Thus the same server should run on both a small PDA and a large 3D graphics

Fig. 1: Whether rotated or enlarged: TrueType- and Bitmap fonts are improved in quality as the result of edge smoothing.



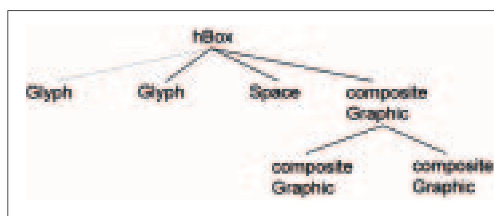
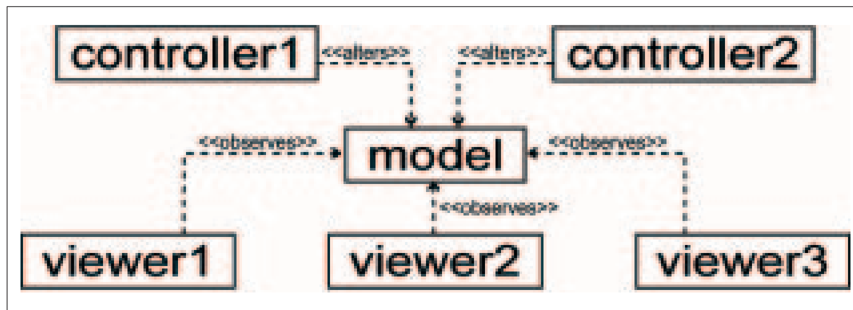


Fig. 2: Example of a structured graphic (Left scenegraph, right result)

Table 1: History - From Interviews to Berlin	
1988	InterViews, Stanford University (Linton, Calder, Vlissides) InterViews was a fruitful attempt to combine structured graphics and object-orientation.
1994-98	Fresco, X Consortium Fresco was the successor to InterViews, extending many of its ideas and integrating the individual libraries into a modular framework. The main focus of the development was a combination of distributed objects and "distributed graphical embedding". The internal use of an ORB even accelerated the development of COBRA. The Fresco project no longer exists, however, Berlin now being used to manage the remaining documents and archive (see later).
1997	GGI Project The GGI Project was established in order to obtain a general protocol for writing graphics drivers. By separating the Linux kernel from the programmer interface, it enables access to all the functions of modern graphics hardware in complete safety.
1997	Berlin Project By building on GGI, Berlin was an attempt to produce a very easy, powerful windowing system. Portability was considered important, but speed and efficiency were the biggest priorities.
since 1998	Berlin Project, the second generation The establishment of standards such as OpenGL, COBRA and Unicode caused a change in the original Berlin focus, which is now portability and efficient through the use of common standards. In the search for inspiration we stumbled across Fresco, from which we have now taken large parts of the source texts. Indeed, the Berlin of today now has more in common with Fresco than with its own code from before 1998!

056Berlin.qxd 21.11.2000 16:52 Uhr Seite 58

Fig. 3: The interplay of model, views and controllers



workstation. It should also make best use of any available hardware without any changes to its source text. Berlin is trying, by means of abstract interfaces, to develop a system which allows a program to be made usable without alterations in both a truly walk-in 3D environment as well as in the classic 2D look and potentially even as an "acoustic user interface" for the visually-impaired. The whole system is being developed under the GNU Library General Public License.

Concepts

Berlin is essentially based on two concepts. The first is that of so-called "structured graphics", where simple objects such as lines, letters, but also transformational functions such as rotations or crops, are amalgamated into more complex objects. The whole procedure is similar to drafting drawings in vector graphics programs. Figure 2 shows one such (simplified) tree structure and the image created from it. In the example, several glyphs (graphical representations of letters) and a complex vector graphic are combined into a *hBox*. This element allows all subordinate graphics to be displayed side by side on a common baseline.

Model-View-Controller ("MVC") is derived from the programming language Smalltalk and is a second concept which is central for the configurability of the user interface. This means that

Fig. 4: The components of Berlin

056Berlin.qxd 21.11.2000 16:52 Uhr Seite 59

a program ought to be split into three different function groups. The *Model* contains the status of the program. That could for example be a number of binary states, a variable value in a given interval or a text. This model is processed by one or more *Controllers*. As soon as one of these controllers alters the state of the *Model*, the latter notifies this alteration to all *Views* involved. The views thus represent the graphical appearance of a program and the controllers its behaviour. In the interests of configurability by the user, Both should be as flexible as possible. The actual program functions and data stay in the model, the logistical backbone. Figure 3 shows how these three components interact. Here, the model consists of three numbers, which represent the red, green and blue parts of a colour. The colour field in the middle observes this model. If that changes, its colour changes correspondingly. The controllers here are the slide controllers. If these are moved, they change the corresponding colour values of the model.

..and their realisation

Berlin uses a so-called *scenegraph* for display. This is a structured graphic stored in the server. Here it is possible to see the strict separation of the *model* in the client and the associated *views* in the server. In the tree-like structure of the scenegraph, each client is responsible for one part of the tree. The client can insert, process or delete graphics, thus creating their own illustration. These graphics are demanded by the client from various modules in the server. The entire graphical representation of the clients is found there. Subgraphs (such as a window content) can be affected with the aid of upstream *decorator patterns*. These patterns are in fact invisible graphics, which affect the appearance of their dependents. These can be displacement, rotation, or a change of fonts or colours. The displacement of a window may be caused by its descriptive decorator pattern being inserted in front of the window subgraphs. This causes little or no communication with the client. The server has all the information it needs to redraw the whole window.

Up to this point no assumptions whatsoever have been made about the basic hardware. All details within the scenegraph are abstract. They are also independent of the pixel co-ordinates and colour codings of the hardware. For this reason, the normal application developer no longer needs to worry about colour depths, screen resolutions and print support. Berlin adapts the output accurately to the output device in use, regardless of whether this is now a monitor, a video wall or a printer. Various *DrawingKits* are used to do this. These run through the scenegraph in a so-called *draw traversal* and thereby create an output that can be processed by the basic hardware. Currently raster and OpenGL display lists are catered for.

A Postscript DrawingKit is planned.

Figure 4 and Table 2 describe individual components of Berlin.

These exclude *DrawingKits* (which has already been described) the *DesktopKit*, which contains functions for control of windows, and the *WidgetKit*, which provides frequently used graphical user interface and display elements of particular importance. Both of these kits have a similar function in Berlin to the various window managers and GUI-ToolKits in X.

Interaction

One part of the scenegraph consists of controllers. These are graphics objects which can receive,

Table 2: The components of Berlin

Display Server:	How the X-Server represents the graphical interface of a client.
libBerlin, libWarsaw	connect the client or server respectively to the Object Request Broker (ORB).Both are created by a compiler fromBerlin's CORBA interfaces.
ORB	The CORBA ORB allows communication between client and server, regardless of their respective locations in the network.
libPrague	contains classes which abstract Berlin from its underlying hardware.This library is intended to allow easy portage of Berlin.
Console	Another abstraction. The console converts events from external libraries that query the hardware into Berlin events.
Kits	Kits are dynamic loadable modules, which provide respectively defined functions. Configuration of Berlin is possible by simply exchanging kits during run time.

CORBA

CORBA (Common Object Request Broker Architecture) is a standard for the development of distributed systems. In such systems the entire functionality of an application is represented by objects. In such an architecture, the distinction between client and server becomes blurred. Client-components can create objects which behave like servers.

The flexibility of distributed systems arises from the fact that all components possess a defined interface. This interface tells the components which services another component provides and how these can be used. As long as this interface remains unaltered, the implementation of a component can alter fundamentally, without the other objects involved being aware of this.

With Interface Definition Language (IDL), CORBA provides a standard mechanism for the definition of such interfaces. IDL is not an implementation language – only interfaces between objects can be defined. Source texts are created next from these definitions in the various programming languages that implement these interfaces. Only now do they need to be provided with the program logic. Berlin clients in C++, Python, Perl and Java have already been realised using these features.

In addition CORBA offers various services which make it possible to find an object. It also allows communication between different objects (including via address space) and even beyond computer limits. This communication is mediated through one or more ORBs (Object Request Brokers). On the basis of its Object Reference, the ORB can find and contact the object in the network.

056Berlin.qxd 21.11.2000 16:52 Uhr Seite 60

process and if necessary pass on events. So controllers are special *decorator patterns*, which alter not the appearance, but the behaviour of their dependents. Any graphics you desire can be converted, into a button for example, by adding a corresponding controller. All controllers are combined into another graph, the *Controlgraph*. A *pick traversal* is applied to this, to find the right recipient of an event.

Events in Berlin are kept more abstract than for example in X in order to be prepared for future input devices. This also makes events easier to synthesise. Used perhaps to simulate a mouse via the keyboard or to enable a pen-based input on a PDA.

Since controllers are in the server, many events can be carried out without any communication whatsoever with the client. The displacement of windows is another example here. The responsible controller of the DesktopKit communicates directly with the transformer of the window.

CORBA, with all its functions, seems ideal to take over this communication between server and client-objects. The Berlin interfaces defined with IDL correspond in their function to the X-Protocol. But as a rule, communication via CORBA is slower than via sockets. This frequently leads to the assumption that Berlin must be slower than the X Window system. But since communications between objects which live in the same address space are depicted in C++ as a virtual method call and in Berlin the vast majority of communication takes place in the address space of the server, slower communication between client and server is just a bad memory. Even demo-applications with a high proportion of communication between client and server are not significantly slower than those with a high proportion of communication in the server.

Kits

The separation of interface and implementation makes little sense if a programmer can only create elementary objects such as lines, letters or simple controllers. It makes much more sense to create special objects, so-called *Factory-objects*. This can make and "wire" complex trees of elementary objects in one go. In Berlin there are several such

Compatibility

Berlin and X have so few common features that the user cannot simply use his normal applications under Berlin. This is a serious drawback. Who wants to do without his favourite application? So how can Berlin be adjusted to represent existing programs?

Pixel-Level bridge

The simplest option for achieving X-compatibility is to run an X-server in a window. This option is technically relatively simple to realise and makes it possible to use all X-applications under Berlin. The drawback is that none of the advantages of Berlin come into play in this conversion. A conversion of this pixel-level bridge is already being worked on. Display is already functioning (see Figure 5), but there is as yet no passing on of events from Berlin to X.

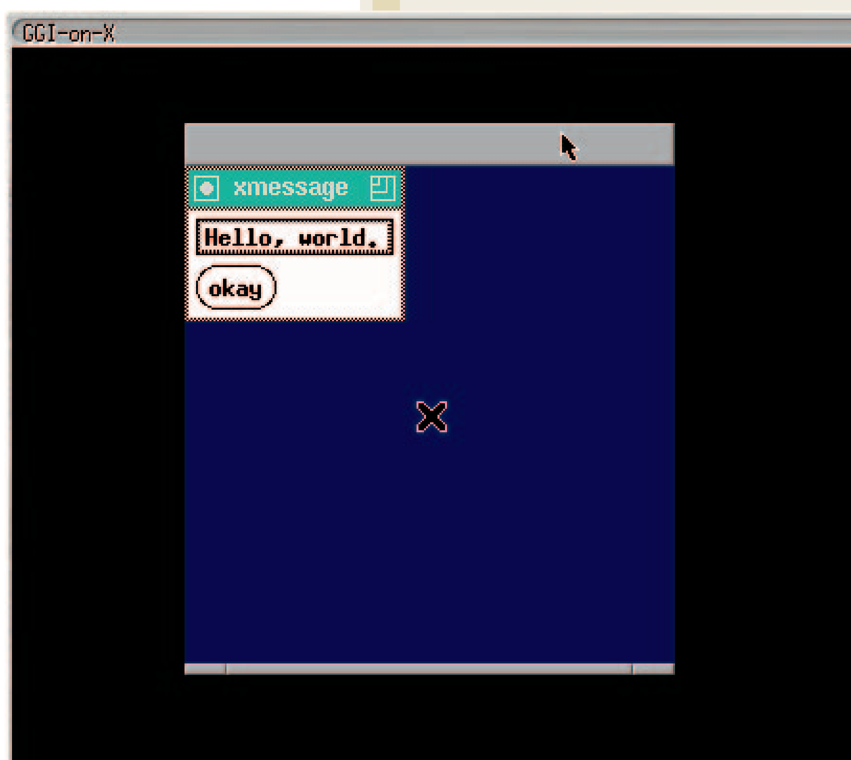
Widget bridge

In a widget bridge a wrapper library is written to convert calls to an X-based GUI-ToolKit into corresponding Berlin structures. Such a bridge has the advantage that the fundamental characteristics of Berlin (transformations, server-side object communication) are largely retained. However, realisation is very time-consuming and does not result in any additional function with respect to the X-version of the GUI-toolkit.

High-Level-Bridge

Many applications now use GUI description languages, mostly based on XML, such as Glade, XUL and Entity. With such a "global knowledge" about a user interface it may be that better connections are made between Berlin and ToolKit widgets, than is usually the case with the widget bridge.

Figure 5: X in Berlin in X



056Berlin.qxd 21.11.2000 16:52 Uhr Seite 61

Getting started...

Installing Berlin is not totally simple, as we use a few "incomplete" libraries, which have sometimes not yet made an entry in the current distributions. For this reason, there are still no Berlin packages in binary format.
The libraries needed are, in detail:

* *libGGI Version 2.0 beta*
* *Mesa Version 3.1 (with GGI-Support) or*
* *libArt, from the Gnome-CVS-Tree*
* *FreeType Version 2.0 beta 8*
* *libPNG Version 1.05*
* *libz Version 1.1.3*

All these libraries are really easy to install. Only GGI and Mesa with GGI support occasionally cause problems. The associated demos have to run – otherwise Berlin will not function either!

The next thing we need is an ORB. All ORBs with a language connection to C++ which support the POA and provide a name service, are suitable for Berlin. We would advise using omniORB Version 3.0 (or later). So far this is the only one to be "automatically" recognised. With others, manual modification of the Makefile is necessary. The installation of omniORB shouldn't give rise to any problems. After that the name service has to be configured and started. For this, the file /etc/omniORB.cfg is required. In this case, this contains the following lines:

```
ORBInitialHost C3PO
ORBInitialPort 8088
```

Please replace "C3PO" with your own computer name. When this file has been created, omniNames can be started for the first time:

```
omniNames -start 8088 \
  -logdir /var/log/ \
  -errlog /var/log/omniorb-errors
```

A good test of the configuration of the name service is to call up nameclt list. If this waits for a time-out, then something is wrong.

Finally, to compile the sources, we need a C++ compiler. The g++ version 2.95.2 is a good choice. Earlier versions can cause problems under some circumstances. You need to bear in mind that the compiler has to cope with multithreading (the compiler itself must have been converted with the option '–enable-threads'). All the more recent distributions should comply with this requirement.

All that is needed now is the source code for Berlin. Depending on how adventurous you feel, you can choose between "stable" releases or nightly snapshots. It is also possible to check out the CVS Repository. The releases are split into smaller modules. The packages "Berlin", "Prague", "Warsaw", "Server" and one of the clients will be needed for testing.

You're motoring once you've unpacked the packages. A simple make is enough. First the user will be asked some questions. The default answers should usually be appropriate in each case. The compilation will then start immediately. Installation is not yet foreseen – all programs can be executed directly from their directories.

Berlin needs a few environmental variables. These can be get using

```
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:\
  `pwd`/lib
export BERLIN_ROOT=`pwd`
```

Please take note that these commands have to be executed in the uppermost directory of the Berlin source tree. When using shells other than bash, the commands should be adjusted accordingly.

Next, the server can be started with server/server. Depending on whether you are on the console or in X, the screen will either be black, or a black window will appear. At first, nothing else will happen.

Now a client can be started. This, too, needs both of the environmental variables mentioned above. The C++ client is started with clients/C++/demo.

In the case of problems the Berlin FAQ can be of assistance, and in more difficult cases, the mailing lists will help. Links to both of these can be found on the homepage of the Berlin project.

factory objects. These are all responsible for a specified type of objects and are combined into *Kits*. These kits are the mechanism on which the whole modularity and configurability of Berlin is based. The server itself has just one function – to reload kits dynamically. It does not even need to know which interfaces a kit makes available to do this. It is enough that the application program knows.

Widgets and taskets

One of the most interesting aspects of drafting user interfaces is the interaction between man and machine. Over the course of time certain metaphors, such as icons, buttons and scrollbars, have been developed. If the architecture itself demands a separation of data and presentation (keyword MVC), it can't be that difficult to design the presentation objects in such a way that their appearance, as well as their function can be configured.

What looks to the user like a complex product, perhaps a scrollbar, is displayed as a structure in Berlin. Elsewhere, objects have defined attributes and status variables. Here these are an arrangement of graphics and controller objects, created with the aid of one or more factory objects. The creating kit is the only authority that knows the precise structure. This is the basis for the selection of different display methods. By exchanging the "WidgetKits", a different appearance is achieved.

More abstract levels can also be permitted. When one considers the function of a scrollbar, one notes that it alters a value within given limits. If one only takes these tasks, one reaches the term "Taskets". When a user wishes to find a list with

given values, he can do this in various ways ranging from menus, up to radio buttons. All perform this same task, so implementations can be one and the same tasket. In addition to the WidgetKit, Berlin also provides another more abstract "TasketKit" enabling the user to seek out his preferred selection method. This can either demand a radio-button group from the WidgetKit or instead do something completely different. The imagination has no limits. A menu is less useful than an acoustic response for the visually impaired. The application developer has the option to either impose the appearance of his/her application, by manipulating elementary graphics objects, or work as usual with Widgets. He or she can decide to work with even more abstract products, such as Taskets.

Development status and prospects

It may appear that the Berlin Project has done little in recent months. But there have been a few changes in the internal structure. First of all, many dependencies on other projects have been dissolved. Berlin no longer needs MesaGGI because of the *libArtDrawingKit*. A new hardware abstraction layer, the "console" means that even the dependency on the GGI

(General Graphics Interface) has now gone. ORB has now become exchangeable. Apart from this, the whole architecture is more robust. This is the result of improved memory management.

Berlin is now truly stable and a bit faster than in previous versions, although there is still a great deal of room for refinement. Nevertheless, Berlin is not suitable for end users. Apart from a few demos, there are still no applications. There isn't even a complete set of Widgets in order to develop these. At present, Berlin should be regarded as more of an experimentation field for developing man-machine interaction.

Since the architecture is now in large sections, work is now slowly beginning on the implementation of additional widgets. Portage to BSD and other systems is in the pipeline, together with the integration of additional libraries such as SDL and GLUT in Berlin. The developers are hoping that this will provide the long awaited "hardware support" for display. Until now, Berlin has had to manage with just software rendering. At the same time, portage onto an SGI Onyx is in the works. The project team members are hoping to be able to try out Berlin in a real, walk-in 3D environment on this machine. In preparation for this a so-called *PrimitiveKit* is being worked on. This will make it possible to insert simple three-dimensional objects such as spheres and cubes into the scenegraph. Of course, the development of the so-called *canvas widget* is not standing still, either. This could be used to embed an X-Server in Berlin. The *TextKit* is currently still in a revision phase, because it still does not provide the bi-directional textflow which would be necessary for complete representation of Unicode symbols.

It is obvious that there is still a great deal to be done before Berlin is seen as the dreamed-of alternative to X. The team of the Berlin Project would be delighted to receive any assistance in further development. ■

The author

Tobias Hunger is a student of computer science at the University of Kaiserslautern. He also runs a company dealing with questions concerning network and computer security.

Info

[1] Berlin Homepage http://www.berlin-consortium.org/
[2] Fresco Homepage http://www2.berlin-consortium.org/fresco/
[3] CORBA Information http://www.corba.org/
[4] Unicode Standard http://www.unicode.com/
[5] OpenGL Information http://www.opengl.org/
[6] Die LGPL (GNU Library General Public License) http://www.gnu.org/copyleft/lgpl.html
[7] GGI-Project Homepage http://www.ggi-project.org
[8] Mesa (free openGL-Implementation) http://www.mesa3d.org/
[9] libArt http://www.levien.com/libart/
[10] FreeType Homepage http://www.freetype.org/
[11] PNG Homepage http://www.libPNG.org/pub/png/
[12] zlib Homepage http://www.info-zip.org/pub/infozip/zlib/
[13] omniORB Homepage http://www.uk.research.att.com/omniORB/
[14] Berlin Sourcecode http://www.berlin-consortium.org/install.html