PENGUIN — A Portable ENvironment for a Graphical User INterface

M. Fischer
Graphics Group, Dept. of Computer Science, University of Bonn,
fischer@graphics.cs.uni-bonn.de

Abstract

Although designers of user interfaces can choose between several different user interface management systems like InterViews, Fresco, Motif, Tcl/Tk, ET^{++}, Forms, or WIN[1], none of these systems is portable between different operating systems. Most of them are fixed to the X Window system. All of them provide a large number of dialog objects, i.e. a large number of object classes like buttons or sliders. The complexity of the interfaces is further increased by the diversity of the individual objects (push buttons, check buttons, ...). As a consequence, the training period for programers becomes unacceptably long. They are not designed to be easily extendible if the functionality provided is insufficient. The reusability of their objects is small. It is very hard to integrate them into another environment, e.g. use them in combination with a special graphics package. In this situation we felt the need for a portable graphical user interface that is easy to learn and easy to use still providing an adequate functionality.

Categories and Subject Descriptions: I.3.4 [**Computer Graphics**]: Graphics Utilities — *Graphics packages; Software support*; I.3.6 [**Computer Graphics**]: Methodology and Techniques — *device independence*;

Keywords: UIMS, OOP, platform independence, high quality graphics, software productivity, extensibility, customization
General Terms: Design

1 Introduction

State-of-the-art and thus complex graphical user interfaces tend to have a large number of different dialog objects, i.e. object classes and subclasses[2]. This leads to complex interfaces most likely resulting in extensive training periods. Desired features can only be accessed by setting a large number of parameters for constructors or functions creating an object and/or by selecting and calling a few out of many possible methods or functions. Many parameters make function calls hard to read and increase the likelihood of incorrect usage. The case of more than one function that must be called to use the framework leads to the use of uninitialized (or only partially initialized) objects. The completeness of the interfaces leads to a lack of extensibility. It is very hard to integrate special dialog objects, e.g. 2D sliders or objects doing calculations during idle time, in any of the interfaces. For a comparison of the abilities and shortcomings of various user interfaces see [Fis96].

As a consequence, PENGUIN's (short for Portable ENvironment for a Graphical User INterface, [Fis96]) main design goal was a compact interface, minimal training time and comfort of usage. The compactness of PENGUIN supports beginners becoming familiar with the package and significantly reduces the time of average development cycles. This is further improved by a large number of default parameters making it possible to develop a user interface with minimal knowledge of function parameters. It is impossible to create dialog objects that are only partially initialized and hang up the system when they are used. To become started, only two methods have to be called: method *t_Penguin::instance()* and method *t_Penguin::run()*.

[1]The graphical user interface provided as part of Microsoft Windows.

[2]In this context, the word class is used with a non-object-oriented meaning for a functionality of the objects, e.g. a button, and the word subclass is used for the different graphical appearance of objects in a class, e.g. a radio button, a toggle button, ...

Even though some ideas originate from InterViews and Fresco, the interface of PENGUIN is much cleaner and more powerful. Therefore, updating to a new version of the interface is an easy task compared to the updates from InterViews versions 2.6 to 3.0 to 3.1 and finally to Fresco. During the development of PENGUIN a consistent interface has been kept across different versions.

2 Basic Concepts

The functionality of PENGUIN is divided into five parts, which are described in the following sections. Each part focuses on some basic concepts that are used throughout PENGUIN. These are the parts:

Control Elements Elements that are responsible for access to the window system, e.g. for opening windows or receiving and dispatching events, that manage hotstrings, save the context, and other global jobs.

Events Events are used to pass information from the user to the system, e.g. with mouse or keyboard events. The list of event types supported by PENGUIN is extensible. The system will react to any event derived from a predefined type. This provides an easy way for an application programmer to pass information from one section of the program to another one.

Dialog Objects These objects define the interface to the user. They can be created, modified, or deleted at any time. Derived classes implement common objects like buttons or sliders. The different dialog objects that are available provide a small but adequate functionality.

Actions Actions are called by dialog objects to perform an application specific task. Every action, that reacts to any event or is executed by any dialog object, can be connected to any character string that is entered.

Dialog Description Files These files are used to parse a dialog description at runtime. They provide rapid prototyping to PENGUIN and simplify development of dialog hierarchies.

All interfaces provided by PENGUIN, e.g. interfaces to dialog objects, events, control elements, or actions, only offer methods using reference pointers [Fis95a][3] whenever pointers to PENGUIN objects are used. The use of reference pointers ensures that all objects live exactly as long as they are used without requiring a complex management by the application programmer.

2.1 Control Elements

All user interface systems need some global control mechanisms. It is standard that an application transfers program control to the user interface system by calling a special function after initializing the interface and defining the necessary[4] dialog objects. This function keeps control of the program, either until the program ends or until control is explicitly transferred back to the application (this is possible in some user interface systems). Inside the function events are collected (e.g. from the underlying window system) and forwarded to the dialog objects in one way or another. In PENGUIN, this functionality is completely encapsulated in class *t_Penguin*. This class contains all available control elements. It is the only class in PENGUIN that contains system dependent parts. To simplify the structure of the class, the system dependent parts have not been encapsulated inside the class or moved to a special class. The easy portability of PENGUIN is obtained by implementing all system specific methods in special files (one for each system).

Class *t_Penguin* is responsible for creating, deleting, showing, and hiding of windows. It manages lists of hotstrings. A hotstring is a sequence of characters, i.e. a character array, that is used to execute an action. It is similar to hotkeys, but the use of more than one character

[3]A special design for reference pointers guarantees that no overhead or complicated or unpleasant usage is forced on the application programmer.

[4]Either all dialog objects used throughout the application or only a subset, e.g. top level dialog objects, depending on the system.

enables direct access to all available actions, using a long name to globally access an action and a short name, e.g. only one character, to access actions in the current scope. Class *t_Penguin* redraws the windows and reinitializes the dialog objects if necessary (e.g. after resize events). It receives events from the system (e.g. events from X Window) and events sent with method *sendEvent()* and dispatches the events to the dialog objects. It offers methods to flush the event queue (i.e. to delete all outstanding events) up to an event of a special type or to deliver events only to a special dialog object. It provides access to the parser for dialog description files and registers parsing functions defined at runtime (see section 2.5). It enables applications to access windows and instances of class *t_Cgi* [FF93] opened by PENGUIN and the hierarchy of dialog objects for every window. It provides some information that is global to PENGUIN, e.g. a minimum and maximum font size requested by the user.

In every application, the first step to use PENGUIN is to instantiate class *t_Penguin*. To ensure that only one instance of class *t_Penguin* is active at any time, only private constructors are provided. The user must use the static method *instance()* to instantiate the class and/or to get a pointer to the actual instance. Afterwards the application builds up a hierarchy of dialog objects (of course, this hierarchy can be changed at any time). This can be done by either directly instantiating the dialog objects or by using dialog description files. Then method *run()* of class *t_Penguin* is called to start collection and dispatching of events. When method *run()* returns, most applications clean up memory and exit. Of course it is possible to restart event handling any number of times.

2.2 Events

Events describe an action that has taken place, e.g. that a key has been pressed on the keyboard, or an action that should take place, e.g. an event defined and sent by an application programmer can invoke a special dialog object. A common interface for all events is defined in base class *t_EventBase*. It defines two methods used by PENGUIN to dispatch the events to the dialog objects. The methods describe the window and the position inside the window where the event took place. These two pieces of information are meaningful for all events created by the user, e.g. mouse or keyboard input. They might not be meaningful for events defined in the application, though the application routine can also provide a valid window id and position in the window to speed up event distribution. The application programmer must ensure that the window and position of an event created and sent by the program are recognized by the dialog object that shall handle the event. Newly defined dialog objects are free to accept events of any type that occur inside or outside their active area. The reason to provide the window and position in the base class is that most dialog objects react to user input and are only interested in events that took place inside their active area. This is used by PENGUIN. It optimizes dispatching of the events by first delivering them to the window they occurred in and provides a default implementation that rejects events outside the active area of a dialog object in the base class for dialog objects. The base class provides another method that is intended to store the state of the modifier keys (e.g. shift, alt, and control) on the keyboard at the time the event was created.

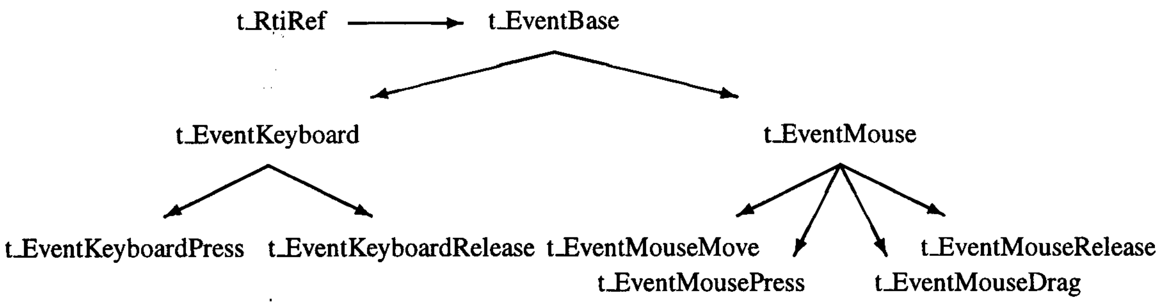

Figure 1: Class hierarchy showing classes derived from class *t_EventBase*.

Classes *t_EventMouse* and *t_EventKeyboard* are derived from class *t_EventBase*. They are designed to handle input events generated by the user (though events of these classes can also be created and sent by the application). Class *t_EventMouse* provides the information which button has most recently been changed and which status all buttons have after this change. Class *t_EventKeyboard* provides a character string describing the key that was pressed. This can either be a single character or (for special keys) a string, e.g. *"enter"*. Derived from these classes are classes *t_EventMouseMove*, *t_EventMousePress*, *t_EventMouseDrag*, *t_EventMouseRelease*, *t_EventKeyboardPress*, and *t_EventKeyboardRelease*. Depending on the type of event that occurred the corresponding derived class is instantiated. The application can access the type with runtime type information [Fis95b]. Figure 1 shows the complete class hierarchy derived from class *t_EventBase*.

2.3 Dialog Objects

Until now only elements of PENGUIN have been discussed that are invisible to the user of an application. This section focuses on the interface between the application and the user. The interface consists of objects, called *dialog objects*, which provide information to the user, e.g. a text output, and/or provide a possibility for input, e.g. a button.

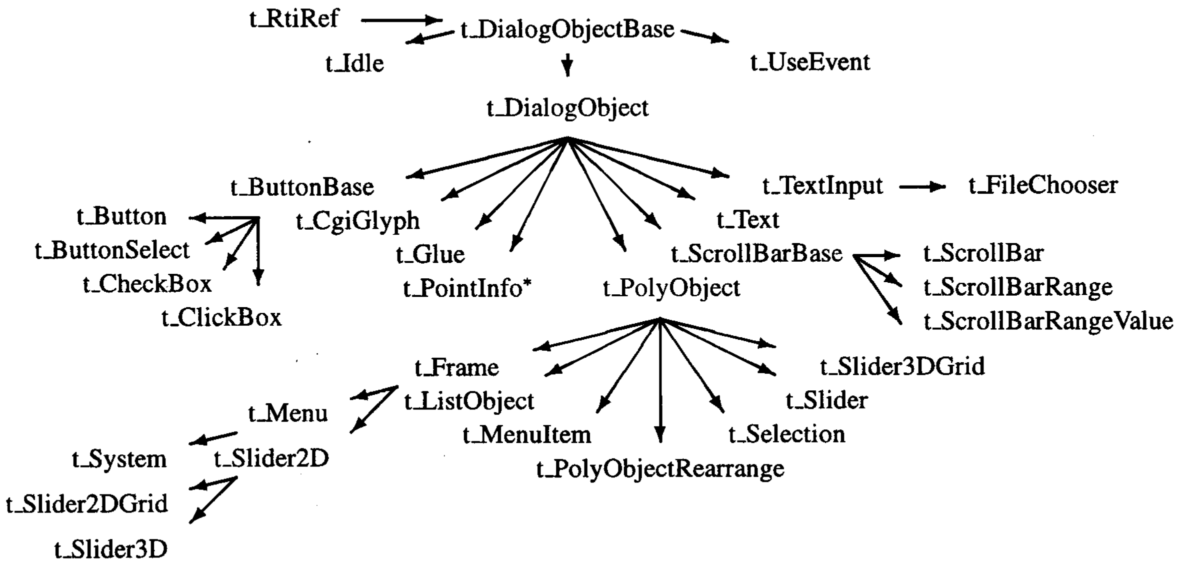

Figure 2: Class hierarchy showing classes derived from class t_DialogObjectBase. *) Class t_PointInfo is also derived from class t_Action.

Class *t_DialogObjectBase* defines the basic interface for all dialog objects. To keep the system open for improvements and extensions, this class defines only a minimal set of methods all dialog objects must support. All dialog objects focused on user input are derived from the subclass *t_DialogObject*. Direct use of class *t_DialogObjectBase* is only necessary to enlarge the functionality of PENGUIN with dialog objects that do not react to user input. It offers methods for redrawing of the dialog object, reactions to focus changes, dispatching of events, and coordinate mapping from a window position to a position inside a dialog object. Class *t_DialogObject* implements all abstract methods from class *t_DialogObjectBase*. It enhances the functionality of the base class by defining methods corresponding to all input events and by defining some colors to encourage designers of dialog objects to use only a limited number of different colors. The usage of predefined colors optimizes color management and reduces the number of similar but different colors. Figure 2 shows the class hierarchy derived from class *t_DialogObjectBase*. The underlying graphics library used for drawing is *CGI*$^{++}$ [FF93]. It provides a portable graphics interface offering the complete functionality defined in CGI

[ISO91].

A hierarchy of dialog objects is a tree where any node has an unlimited number of children, which can be nodes or leafs. The root of the tree is an instance of class *t_PolyObject* or of class *t_PolyObjectRearrange*, which is automatically created when the window for the hierarchy is created. The application programmer can choose whether class *t_PolyObject* or the derived class *t_PolyObjectRearrange* is used. Class *t_PolyObject* uses exactly the geometrical specifications of its children without respect to minimum or maximum size restrictions. Class *t_PolyObjectRearrange* adapts the specifications of its children to find a best matching graphical layout for the children. This includes resizing and moving of children relative to each other to meet size restrictions. There is no restriction on the position of dialog objects inside the tree except for objects that cannot be used as nodes. This introduces some new aspects to the design of dialog hierarchies, e.g. the possibility to insert checkboxes or sliders directly into menus. This minimizes the amount of work necessary for the user of an application to look up or change objects.

2.4 Actions

Dialog objects must access application specific functions to perform their tasks, e.g. a button labeled "quit" must somehow be connected to a function that ends the application. The interface to these functions is provided in class *t_Action*. Note that PENGUIN does not use global callback functions for this task but defines a clean interface. Class *t_Action* provides a virtual method *execute()* that is called when the action is triggered. This method is overloaded in derived classes to implement an application specific action. Depending on the application there are three basic approaches for the design of actions. The first is to derive a class from class *t_Action* for every action that is defined thus creating many different classes. This is impractical for actions that are closely connected. The second approach is to design a class hierarchy that is derived from class *t_Action*. An interface between actions that are closely connected is implemented in a base class (maybe as static members) and special actions are derived from both classes. Alternatively, method *execute()* can be used to switch between different actions. The third approach is used if one class, e.g. named *ApplicationClass*, exists that is instantiated only a few times and that defines methods that must be executed by an action. For this task, only one class is derived from class *t_Action*, which is passed a pointer to an instance of class *ApplicationClass* and a pointer to a method in class *ApplicationClass* when it is constructed. It overloads method *execute()* to call the method for the instance of class *ApplicationClass*. The class derived from class *t_Action* is instantiated once for every instance of class *ApplicationClass* and every method inside the class that must be accessed.

Method *execute()* accepts a pointer to the dialog object that invoked the action as the only parameter. This pointer is provided to access special information kept in the dialog object, e.g. the value of a slider. To access the functionality offered from the dialog object, runtime type information is used to convert the passed pointer to a class derived from class *t_DialogObjectBase*. This concept enables an action, which is part of an application, to access information available in a dialog object specially designed for that application. Class *t_Action* also provides methods to automatically register a name for the action in the parser. This is used to access the action from dialog description files (see the next section).

2.5 Dialog Description Files

To allow rapid development of applications and to loosen the connection between the graphical appearance of the user interface and the application functions that implement a specific behavior of the program a parser has been defined that allows definition of dialog objects (a complete hierarchy or only part of it) at runtime. The parser, which is based on the tool PCCTS [Par95], reads dialog description files and creates corresponding PENGUIN objects.

The parser simplifies the usage of PENGUIN. It provides the possibility to dynamically change the user interface without any modification of existing code. The dialog description files can contain complete dialog hierarchies. Any number of description files can be read in. They may add to existing dialog hierarchies or may overwrite or change them. Therefore it is simple to provide special user interfaces to different users, e.g. interfaces for professional users or for beginners, or to adapt the language of the interfaces. Any user can adapt the interface to special needs by changing the description files. The dialog objects used do not depend on objects used in the application[5], therefore the user is free to use any convenient dialog object. The only restriction is that the used dialog objects must be linked to the application. It is sufficient to create a new dialog object and link it to the application if the functionality or visible output of the existing dialog objects is not convenient for an application. Absolutely no changes in existing code are necessary. In particular, the application programmer is not forced to change any event loop to process events for or from a new object or to assign unique identifiers to any objects, neither identifiers for classes nor identifiers for instances of classes. Such identifiers are used in almost all available UIMS. The user is also free to reorder the dialog hierarchy or to insert dialog objects in more than one place in the hierarchy. This is supported by PENGUIN because the decision whether dialog hierarchies should be orthogonal or not must be made by the application programmer or by the user and not by the UIMS.

The syntax of the parser can be enlarged without modification of existing code. To use this feature, every new object must register a function that reads in the new object in the parser at startup. The underlying idea is described in detail in section 3.2.

The experiences with PENGUIN show that the instantiation of dialog hierarchies at runtime using a parser and clear text encoded description files is sufficiently fast. For small dialog hierarchies, e.g. menues, no delay is visible.

3 Enlarging the Functionality

No matter how many dialog objects a UIMS provides, there will always be a special functionality some applications desire that is not available. Therefore an easy and straightforward extensibility is a necessity for every UIMS (though most UIMS are hardly extensible).

There are several ways to enlarge the functionality of PENGUIN. The next section describes the easiest possible extension, which is to add a new optical appearance but no additional functionality to existing dialog objects. The second section describes what is necessary to register a new object in the parser.

3.1 Optical Changes

Suppose a new optical appearance for a button should be created. To achieve this goal, a new class is derived from class *t_ButtonBase*, called *t_MyButton*. Only three methods are defined for the new class: the constructor, the assignment operator, and the drawing method. The class is defined as follows[6].

```
class t_MyButton: public t_ButtonBase
{
 public:
   t_MyButton              (...);
   t_MyButton&   operator = (const t_MyButton& button);
   virtual t_Bool draw      (const t_2DVector& lower, const t_2DVector& upper);
};
```

[5]Of course, if the application requires the dialog object to provide an argument, e.g. a text string, a dialog object derived from a corresponding class must be used.

[6]The code provided is shortened. For a complete example look at the source code of PENGUIN, e.g. the definition of class *t_Button*.

The implementation of the constructor and the assignment operator is trivial. Method *draw()* defines the desired optical appearance. For this task the complete functionality of *CGI*$^{++}$ is available. The virtual device coordinates and clipping region can be adapted for each object by calling method *draw()* of its base class. Thus the call *t_ButtonBase::draw(lower, upper)* adapts the virtual device coordinates so that the lower left corner of the object has coordinates *(0,0)* and the upper right corner of the object has coordinates *(1,1)*. The clipping region is restricted to the intersection of drawing area of this object and the clipping region inherited from the dialog hierarchy to ensure that no other object is painted over.

```
t_Bool t_MyButton::draw (const t_2DVector& lower, const t_2DVector& upper)
{
   if (!t_ButtonBase::draw(lower,upper)) { return False; }
   // ... somehow draw the button ... maybe only draw the label ...
   label()->draw(cgi(),t_2DVector(0,0),t_2DVector(1,1));
   return True;
}
```

3.2 New Parser Functionality

The new class *t_MyButton*, created in the last section, can only be used if it is instantiated inside the program. It cannot be used in a dialog description file. To achieve this, some more efforts are necessary.

The functionality of the parser can be divided in two parts. Basic functionality, e.g. reading of real numbers or vectors, calculations, or definition and reference of variables, is done invisible for the application programmer. The second part is responsible for the creation of complex objects. The objects must be derived from class *t_RtiRef* since *runtime type information* [Fis95b] and *reference pointers* [Fis95a] are used to access the objects. Complex objects are created with a function call syntax. The input '*button (a, b, c, d, e);*' first creates objects *a* to *e*[7] and than creates object *button* with parameters *a* to *e*. The syntax accepted by the parser for the definition of complex objects is defined at startup and may be changed at any time by registering or removing functions (or static methods) that are able to read in new objects. This concept enables an application programmer to include new objects in an application without any modification of existing code, neither in PENGUIN nor in the application itself. This is important to simplify the definition of new objects, to protect investments in software development since no source code must be supplied, and to integrate extensions provided by different programmers since no changes must be adjusted. It is sufficient to link the application with the object files defining the new dialog objects to instantiate the new object from a dialog description file. Though the possibility to register functions in the parser was originally included in PENGUIN to provide an easy way for an application programmer to use new objects in dialog description files without enlarging the syntax accepted by the parser in an existing parser definition file, the functions can also be used to change existing objects instead of creating new objects[8]. Normally, all functions are registered at startup to be available before the first dialog description file is read in. To register a parse function, a struct must be passed to the parser that contains the following information:

- A name for the object to be matched.
- The minimum and maximum number of parameters for the parse function.
- A pointer to a function that creates (or modifies) the object from an array of parameters.
- A list of parameter types, i.e. an array of length 'maximum number of parameters' that contains runtime type information for the expected parameters.
- A list of parameter descriptions, i.e. an array of length 'maximum number of parameters' that can be used as an explanation of the parameters when dialog objects are created interactively.

[7]These objects can be any kind of object, e.g. variable references or complex objects.

[8]As an example, function *min()* sets the minimal size for a dialog object passed as a parameter and returns the object.

4 Sample Programs

4.1 A Test Application to Instanciate a Dialog Description File

The first example program reads in a dialog description file and runs the menu defined in the description file. No application specific actions are defined in the program. It reads in it's only parameter, the name of a dialog description file[9], instanciates PENGUIN, defines a parser variable (*'root'*), parses the file, adds the parser variable to the main PENGUIN polyobject, and runs the dialog.

```
#include <stdlib.h>
#include "penguin.hh"
#include "pendobj.hh"

int main (int argc, char** argv)
{
   const char* name = "mytest.pen";
   if      (argc==2) { name = *++argv; }
   else if (argc >2) { cout << "Parameters: [filename.pen]" << endl; exit(1); }

   t_Penguin* ctrl = t_Penguin::instance(t_2DintVector(400,300),"PENGUIN DEMO");
   t_DialogObjectBasePtr p = NULL;
   ctrl->defineParserVar("root",&p);
   ctrl->parse          (name);
   ctrl->mainPolyObject (ctrl->mainCgi())->addSon(p);
   ctrl->run            (False);

   t_Penguin::deleteAllInstances();
   return 0;
}
```

4.2 A Complex Description File

The second example shows an extract from a dialog description file used for the 3D rendering toolkit MRT, developed at the University of Bonn [Fel96]. Only a short extract is shown to provide an idea how complex menus can be defined. The complete code would need a description of the MRT that would go beyond the scope of this paper. One part of the menu is visible in figure 3.

The following dialog definition file creates a small part of the menu. First the dialog objects for menu *preview* are created. They are four buttons, named *wireframe*, *phong*, *gouraud*, and *flat*, two checkboxes, named *texture* and *planar*, and a slider, named *trianQual*. For example, each of the buttons is created with an offset, a size, a label, an action, and a hotstring. Note that the size for the buttons is saved in a variable *bSize* and that the offset is created by a multiplication of variable *size* and a position. These objects are inserted in menu *preview*. Menu *preview* is accessed as a submenu from menu item *previewI*. This menu item is inserted in menu *view* together with two buttons *raytrace* and *radiosity*. Menu *view* is accessed as a submenu from menu item *viewI*. *viewI* is inserted into a polyobject that is assigned to variable *mainMenu* (other objects to be inserted in this object are indicated with an ellipses notation). This variable, that is defined in the application, returns the complete menu to the application.

```
size      = 1./7.;
bSize     = (1.,0+size);
wireframe = button   ((.0,0*size),bSize,label("Wireframe")            ,Awireframe,"w");
phong     = button   ((.0,1*size),bSize,label("Phong")                ,Aphong    ,"p");
gouraud   = button   ((.0,2*size),bSize,label("Gouraud")              ,Agouraud  ,"g");
flat      = button   ((.0,3*size),bSize,label("Flat")                 ,Aflat     ,"f");
texture   = checkbox ((.0,4*size),bSize,label("Texture")              ,Atexture  ,"t");
trianQual = slider   ((.0,5*size),bSize,"Triangulation Quality",.0,0-.5,0.5,.1,AtriangQual);
```

[9]Note that the description file must define variable (*'root'*) for this simple example to work.

```
planar     = checkbox ((.0,6*size),bSize,label("No planar refinement"),Aplanar   ,"n");
preview    = menu     ((.3,.05/size),1,0,planar,trianQual,texture,flat,gouraud,phong,wireframe);

size       = 1./3.;
bSize      = (1.,0+size);
previewI   = mitem  ((.0,2*size),bSize,label("Preview"),2,preview,"P",Apre);
raytrace   = button ((.0,1*size),bSize,label("Raytrace") ,Aray ,"R");
radiosity  = button ((.0,0*size),bSize,label("Radiosity"),Aradi,"O");
view       = menu   ((.3,.05/size),1,0,previewI,raytrace,radiosity);

viewI      = mitem  ((.33,.0),(.33,1.),label("View"),1,view);
mainMenu   = polyobject ((.0,.86),(1.,.08),viewI,...);
```

5 Experiences with PENGUIN

Experiences with PENGUIN are rare at the time being. Until now it has only been used for projects inside the graphics group. Besides test and demonstration programs it has been used in a few student projects. Two applications are worth mentioning: The first is the 3D rendering toolkit whose dialog definition file was partially shown in section 4.2. A picture of it is presented in figure 3. The second is a 3D editor, which has been developed by one of the students, B. Hermes, that simplifies definition of scenes for the 3D rendering toolkit. Figure 4 shows a screen dump of the editor. The experiences with these applications show that the design goals of PENGUIN have been reached. The time needed to learn how PENGUIN is used and how its functionality can be extended (if this is necessary) is quite small. Students were able to start working on their real project within the first week of using PENGUIN.

6 Availability of the Software

PENGUIN is available for UNIX (Silicon Graphics and SUN), LINUX, Microsoft Windows NT, and Microsoft Windows 95. Details on how to get the software by anonymous ftp can be obtained from URL http://hyperg.cs.uni-bonn.de/CompGraph.

References

[Fel96] FELLNER D. W.: Extensible image synthesis. In *Object-Oriented and Mixed Programming Paradigms*, Wisskirchen P., (Ed.), Focus on Computer Graphics. Springer, Feb. 1996, pp. 7–21.

[FF93] FELLNER D. W., FISCHER M.: *CGI^{++} — A 2D Graphics Interface Based on CGI.* Tech. Rep. IAI-TR-93-9, University of Bonn, Dept. of Computer Science, Bonn, Germany, Aug. 1993.

[Fis95a] FISCHER M.: *Reference Pointers for Class Hierarchies.* Tech. Rep. IAI-TR-95-12, University of Bonn, Dept. of Computer Science, Bonn, Germany, Aug. 1995.

[Fis95b] FISCHER M.: *Runtime Type Information for Class Hierarchies.* Tech. Rep. IAI-TR-95-11, University of Bonn, Dept. of Computer Science, Bonn, Germany, July 1995.

[Fis96] FISCHER M.: *Software Architectures in Computer Graphics.* PhD thesis, University of Bonn, Dept. of Computer Science, July 1996.

[ISO91] ISO: *Information Processing Systems — Computer Graphics — Interfacing techniques for dialogues with graphical devices (CGI), Part 1-6*, IS 9636, Dec. 1991.

[Par95] PARR, JOHN TERENCE: *Language Translation Using PCCTS and C^{++} (A Reference Guide).* Parr Research Corporation, June 1995.

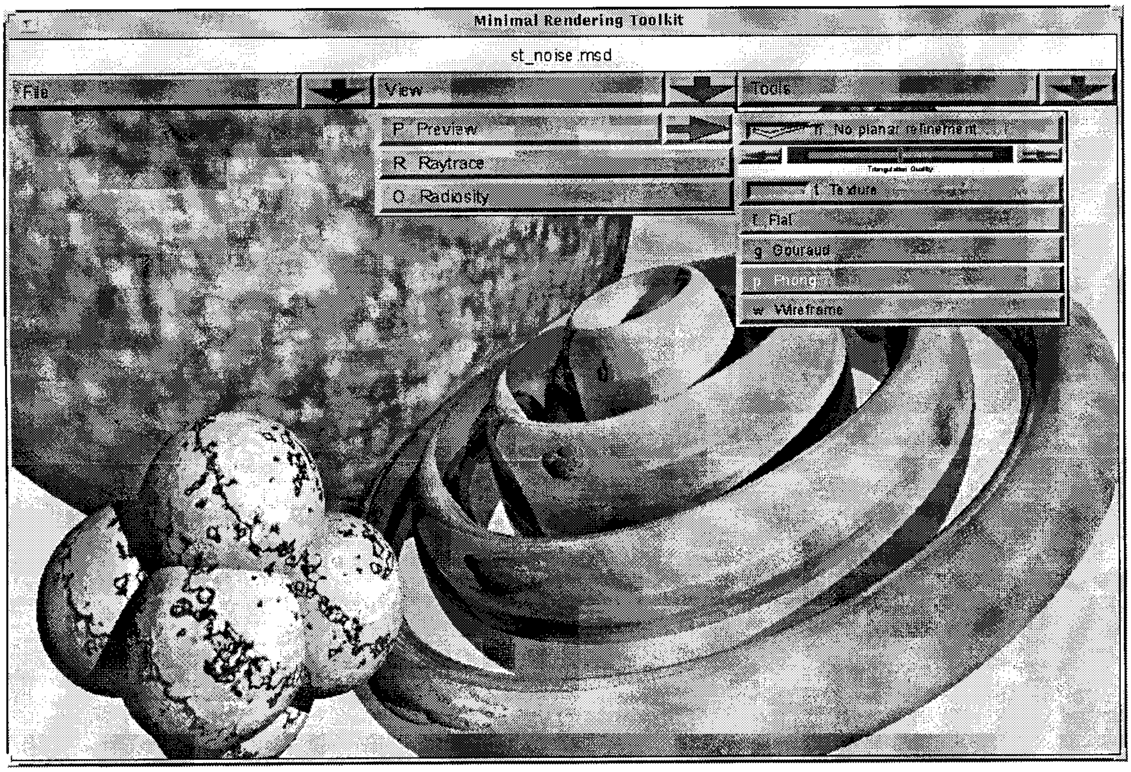

Figure 3: Screen dump of a 3D rendering toolkit using PENGUIN.

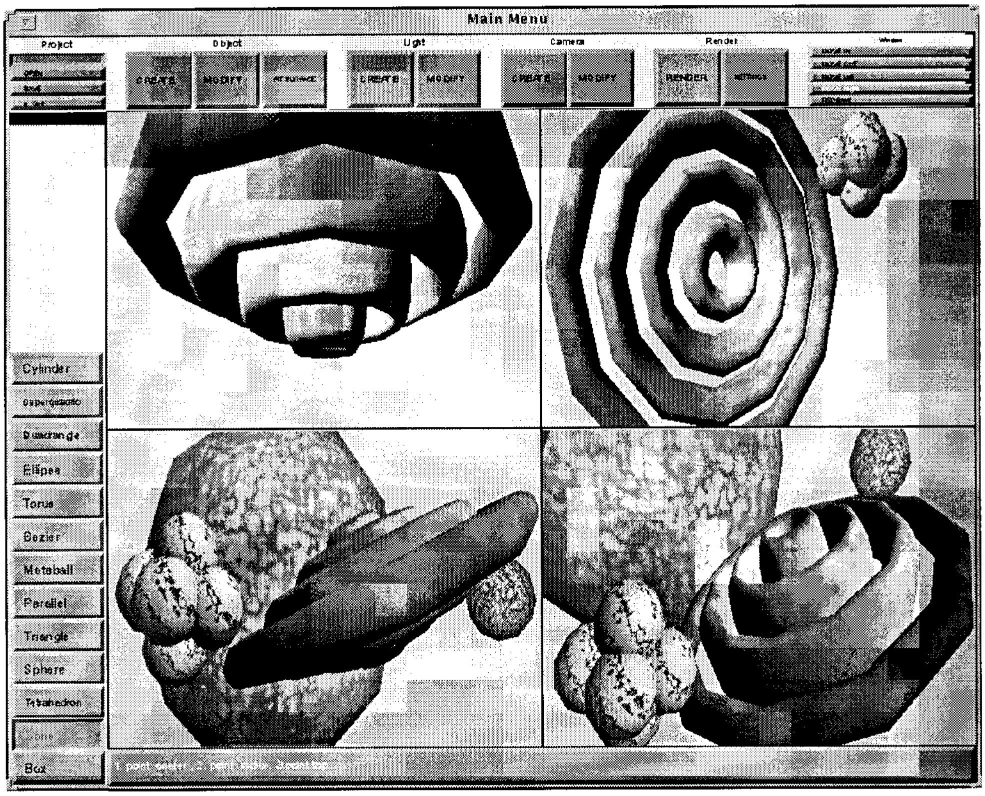

Figure 4: Screen dump of a 3D editor using PENGUIN by B. Hermes.