The GPE graphical programming environment

Peter Ward and Colin Parrott,
The IMP Group
University of Leeds Department of Anatomy, School of Medicine

'TOOLS 91' Technology of Object-Oriented Systems

Abstract

This paper describes the Graphical Programming Environment (GPE), a system that enables users to manipulate and program the behaviour of graphical objects via a tool-based interface and an Object-Oriented Programming Language (OOPL). The GPE was conceived and developed in a year with the aim of delivering a working program that employed an object-oriented paradigm to provide target users (authors of hypermedia applications) with tools that are genuinely easy to use and which enable the construction of machine-independent applications.

Employing C++, X11 windows and with a principal aim of portability, the system was restricted to handling text and graphics. As well as simplicity of use, high quality in the presentation of the interface and application objects was a principal aim. We believe that it is through a process of evaluation of a small, but powerful tools by target users in the construction of a variety of hypermedia applications that the further evolution of user-oriented information modelling environments can be informed. Also discussed is subsequent work building on the lessons learnt with the GPE, including the rejection of C++ in favour of Eiffel for future work in this area.

Introduction

The Graphical Programming Environment (GPE) is an experimental Hypermedia authoring system. Hypermedia may be defined as the dynamic linking and presentation of multimedia information (Meyrowitz 86; Ward 90). The GPE enables Hypermedia applications to be

interactively constructed from text and imported graphics. The GPE treats text and graphics as interactive, persistent objects that belong to a class hierarchy, respond to messages and have spatial and semantic relationships with an another. The user can program these objects to respond to selection with a pointing device such as a mouse.

The GPE was developed over twelve months in 1989/90 by the IMP group at the University of Leeds with funding from the Systems Technology Group, IBM UK Ltd.

Design Criteria

1. To design a user interface that was genuinely easy to operate. A fundamental belief of the IMP group is that providers of technology and the developers of applications need to work towards more "User-Oriented" systems.

2. To support a range of graphical object types. The system had to be able to deal with near-photographic quality colour images and limited animation as well as simple graphics and text.

3. To produce hypermedia applications that were highly portable. The applications constructed with the GPE needed to run on a range of machines, preferably without having to be mangled by translation software. This required a machine independent storage scheme.

4. To investigate a number of technical issues. We were very interested in the whole relevance of Object-Orientated design and programming to the specific problems of Hypermedia and to the problems of software engineering in general.

The Implementation

In the interests of portability we chose UNIX and its variants and the X11 windowing system (Nye 89) as the core implementation dependants. The GPE uses only low-level Xlib part of X11, this was thought sensible, at the time, due to the relative "bugginess" of higher-level toolkits and their lack of standardisation. However, we believe that the system could be re-engineered to use higher level X widgets. The C++ language (Stroustrup 86) seemed like a natural choice considering our existing experience and the fact the UNIX and X11 both have C interfaces. Smalltalk (Goldberg and Robson 86) was considered and rejected for reasons concerning portability, colour support and scale, but had an influence on much of the design. The GPE was initially implemented using X11R3 and the Glockenspiel V1.2D C++ compiler on the Sun 4/260 under SunOS 4.0 and on the IBM PC-RT under AIX. The GPE has also been ported to X11R4, version 2 of C++, SunOS 4.1 and the IBM RS-6000.

The User Interface

The GPE uses the concept of the pointer being a tool that manipulates objects beneath it on the screen. The benefits of using the tool metaphor in interface design are:

1. It provides users with a model that relates well to their experience of the real world.

2. It provides a framework of possibilities for users to explore.

3. It can be elegantly modelled using the object-oriented paradigm.

Actions by the user result in event messages being sent to the current tool, the tool can then send further messages to the object it is being applied to.

The GPE tools were specified using the following criteria:

1. Was the tool essential.
2. Was it easy to use.
3. Was its function understandable.
4. Was it useable in many different situations.
5. Was it compatible with other tools.
6. Was it unique, or could it be combined with another tool.

The objects that tools manipulate have the following features:

1. A rectangular background of solid single colour or 8bit (256) colour image.
2. Optional border of solid colour.
3. Optional overlaid text.
4. A set of intrinsic messages that they respond to.
5. Three user-programmable method scripts.

The system treats everything including the menus, tools and graphical objects in a uniform manner. All these things are "objects". Access to the object world is provided via an Object-Oriented Programming Language, in which method scripts are written. Below is a brief description of each tool: (see Fig 1).

The Rectangle tool creates coloured rectangles with borders. A left button drag defines two opposite corners of the rectangle. If a rectangle is created within another rectangle or image it will become "adopted" by it. An object that is adopted is called a child of the parent object and a sibling of any other objects sharing the same parent. A tree of spatial relationships is thus created.

The Image tool creates rectangular colour images from external 24 bit TIFF files (Aldus 90). A left button click on the desired centre position of the image produces a list of currently available image filenames. A left click on one of these filenames will load that file at the indicated position. If an image is created within another rectangle or image it will become adopted by it.

The text tool adds text editing facilities to a rectangle or image. A left button click on a rectangle or image with no text will add some initial text and put the object in editing mode, if the object already has some text then it will go straight into editing mode. Note: Text wraps automatically and will be automatically reformatted if the object is resized.

The Resize tool changes the geometry of a rectangle or image. A left button drag inside of any corner or edge of a rectangle or image will enable the user to change it's size and shape. The pointer changes to indicate which corner or edge is being resized. An image will be clipped if resized smaller and replicated if resized larger.

The Move tool changes the position of an object. A left button drag moves an object and a new descendant to a new position.

The Copy tool enables objects to be copied. A left button drag makes one or more copies of an object and it's descendants in a new position. When an object is copied all it's parameters including any programmed behaviour are also copied.

The Select tool sends *!select* messages to rectangles or images. A left button click sends the object beneath the pointer a *!select* message. Rectangles and images can be programmed to respond to these messages. The select tool is the intended "end-use" tool - used to control a finished application.

The Query tool displays an editable description of a rectangle or image. A left button click on an object pops up a large rectangle with the following controls:-

Description	**Function on left button click**
name	enter name of object class
x	enter x offset

y	enter y offset
width	enter width
height	enter height
border width	enter border width
font	enter font number
background colour	produces palette
border colour	produces palette
text colour	produces palette
!select method	enter/edit script
!deselect method	enter/edit script
!appear method	enter/edit script

The Colour tool changes the background, border or text colour of a rectangle or image. The current colour is selected by a left button click on the palette. A left button click on an object sets the colour of one of it's component to the current colour.

The Protection tool stops accidental changes being made to an object by automatically translating all tool messages (except those from the protection tool) into !select messages. Objects that are "finished" should be protected. Note: Protection is inherited by all descendants of a protected object.

The Destroy tool removes an object completely from the system. A left button click will pop up a menu offering to destroy the object.

The Exit tool simply lets the user exit the GPE program. If "save changes and exit" is clicked any changes will be saved - the user will find everything persists between this and the next session.

The Language

The GPE has a built in OOPL. The language works by modelling objects that send "messages" to each other. When an object receives a

message, it tries to interpret that message according to an internal "method". A method is a script that contains statements from the language. Underlying the system is a class hierarchy that all objects belong to. Classes are provided for objects such as numbers, strings and lists to support basic programming tasks.

A class has the following components:

Name
Super Class
Instance Part Name List
Instance Method Dictionary
Class Part Name List
Class Part List
Class Method Dictionary

Instance parts exist for each object belonging to the class. *Class parts* are shared by all objects belonging to the class. *Instance methods* are executed when an instance of a class receives a message. *Class methods* are executed when a class receives a message.

Classes are organised into the following hierarchy with the *Object* class at the root.

Object
Tool
Number
String
List
Location
Geometry
Rectangle
Image
Palette
Tool

RectangleTool
ImageTool
TextTool
ResizeTool
MoveTool
CopyTool
SelectTool
QueryTool
ColourTool
ProtectionTool
DestroyTool
ExitTool

Although many object-oriented systems present users with large number of classes, the GPE contains an absolute minimum. The temptation to create many specialised classes was resisted in order to keep the system small, understandable and flexible.

The Language syntax was designed using the following criteria:

1. Simple to understand
2. Minimum number of reserved words.
3. Infix notation and normal precedence for maths operations.
4. Contextual separation of assignment and equality test.
5. Contextual separation of reference and identity test.
6. No need for statement terminator.
7. Procedural style parameter passing.
8. Mappable to object-oriented semantics.
9. Parsable by recursive descent.

In many situations a method needs to send messages to the receiver of the message - this is accomplished by using the special object name *self*.

To teach a rectangle to turn red when selected would require programming it's !select method as follows:-

Self!colour (red)

This statement sends the message *!colour(red)* to the rectangle. To find he present colour of the rectangle we would use the following expression:-

Self?colour

This expression yields an object that is the colour of the rectangle.

The Architecture

The GPE is constructed as a virtual machine that executes high-level object-oriented code on an interpreter. This enables much of the system to be written in the internal language rather than C++. In fact, the majority of the intrinsic methods are written using the internal OOPL.

The virtual machine is divided into 3 major units:

A Persistent Storage Manager (PSM) looks after all memory and file management tasks for GPE objects. This enables all objects to be referred to in C++ via 32 bit handles called IDs. Unlike machine addresses these are portable, persistent and can address a larger space than the machine memory, without the use of virtual memory.

The X11 Interfaces translates incoming X events into messages to the current tool and implements graphical output primitives.

The Kernel sits between the PSM and the X11 Interface and consists of the following sub-systems:

A Script Compiler uses a recursive descent algorithm to parse script text strings into an internal (more efficient) form. The compiler has access to internal dictionaries and the call hierarchy. The compiler consists of three conceptional sections - a lexical analyser, a syntax parser and a

code generator. The latter outputs a 32 bit stream of code that is executable by the interpreter.

An Interpreter implements the sending of messages by executing compiled code and by call primitive C++ functions. It performs method dictionary lookups and superclass chaining to find methods that match receive messages. The compiled code is executed using a reverse polish scheme, a stack and a small set of primitive opcodes. The stack is employed to store temporary variables, parameters and housekeeping information.

A Global Dictionary contains string-ID associations. It is used to store the IDs of classes and global instances.

A Selector Dictionary contains string-selector associations. It is used by the compiler to convert strings representing method selectors into a compact tokenised form. The interpreter can deal with these tokens much more effectively than with text.

A set of idealised C++ objects called “things” map high-level objects onto the machine architecture. These objects represent the basic entities used by the virtual machine such as integers and arrays. These C++ objects are idealised in the sense that they are completely opaque and have sophisticated capabilities like persistence.

Conclusions

The GPE demonstrates the feasibility of using an object-oriented tool metaphor for hypermedia and prototyping applications. It provides a framework for serious evaluation and development and is a useful prototyping tool in its own right. The implementation uses a variety of novel techniques that could be transplanted into other applications.

The GPE has been used to construct several small applications. These include a realistic looking panel with push buttons, toggle switches and

sliders that animate when selected (see Fig 2). The settings of these simulated controls effect other features like the colour of a "LED" or the position of a photo in a frame. Another application displays a diagrammatic representation of a sensorimotor reflex "circuit" in the Human Nervous System. This diagram can be interrogated for information on its components and can be "Stimulated" to produce an animated simulation of the neurons firing etc. Other applications include an interactive demonstration of Human Vision in which the various components of the eye are interrogatable and responses to light and objects moving in the distance are revealed by direct manipulation.

This practical experience has confirmed the usefulness of the language and the majority of the tools. The protect tool was however not a success - an undo facility and an automatic save mechanism would have been much better. The system also needs the addition of two new tools. One for controlling reparenting operations and another for segmenting images into non-rectangular components. The technicalities behind such features are something we would like to address in future work.

We are conducting experimental evaluation exercises with various types of user (including designers of an object-oriented interface building environment employing an event handling approach in contrast to our script approach to user object programming) in order to better understand the effectiveness of various end user interface dialogue strategies. We believe that such an approach to evaluation by designers of tools intended to be used by real users in the construction of applications is very important.

We encountered several problems with C++. Many, such as lack of multiple inheritance and poor stream support have been fixed in version 2 of C++ but we still have an uncomfortable feeling about the language. In being so close to C it shares all of its problems and adds a few new ones. Basic types not being objects, lack of support for persistence, no automatic garbage collection and lack of any decent class libraries are all things that put us off using C++ for future work.

Since starting the project, the Eiffel language and environment (Meyer 88, 90) have come to our attention. The language offers advanced features like assertions, exceptions, generic classes, controllable multiple inheritance and the avoidance of pointers and global variables. The implementation offers automatic garbage collection, constant time features access (message sending) and persistence. The environment includes an extensive class library, and tools for managing compilation, browsing classes, debugging programs and generating documentation automatically.

Using Eiffel instead of C++ for the GPE could have had following results:

The program would have been far more robust due to the use of assertions and the avoidance of pointers. The PSM would have been much simpler if based on Eiffels built-in persistence mechanisms. The "Things" C++ objects would not have been needed as Eiffel has classes with similar capabilities in its library. Automatic compilation management would have meant no time wasted editing make files. Automatic documentation would mean that we had up-to-date documentation to refer to while writing this paper!

Exposure to the Eiffel language and environment have convinced us that a "fresh start" is better than a "bolt-on" approach for delivering a coherent and powerful OOPL. C++, which being a great step forward for C programming, does not offer the same potential as Eiffel.

The authors are delighted to acknowledge the support of IBM UK Hursley Park New Technologies Group.

END