THE USE OF OBJECT ORIENTED LANGUAGES IN GRAPHICS PROGRAMMING

Mark Green and Paul Philp

Unit for Computer Science
McMaster University
Hamilton, Ontario
Canada

ABSTRACT

In this paper we discuss the application of object oriented languages to the construction of graphics programs. Our experience with a particular object oriented language, called EDL, suggests that this type of language is superior to more traditional languages when it comes to the production of highly interactive graphics programs.

We present a brief description of EDL and outline our implementation of it. Language features that support graphical input and output are discussed. The use of EDL in the construction of user interfaces, and several applications of EDL are presented.

KEYWORDS: object oriented languages, graphics languages, user interfaces

1. Introduction

For several years we have been using an object oriented language, called EDL, in the production of highly interactive graphics programs. Our experience with this language suggests that this type of programming language is superior to more traditional programming languages for this particular application. In this paper we present a brief introduction to the object model of computation and the particular object oriented language we have been using. Our implementation of EDL is briefly described giving special attention to graphical input and output. We then show how EDL can be used in the construction user interfaces. The final section of this paper describes three typical applications of EDL.

In the object model of computation a program consists of a number of independent entities called objects. Each object has a data area and a set of procedures. The data in an object can only be accessed by that object. The objects in a program communicate by sending messages to each other. A message consists of the name of the object to receive the message, the name of the message, and any data associated with it. When an object receives a message the message name is used to find the procedure that processes it. The procedures in an object can change any of the object's data values, send messages, and create new objects. Conceptually, each of the objects in a program is a separate process running concurrently with the other objects in the program. This model of computation encourages the development of highly modular program with each object responsible for one well defined function.

Two well known object oriented languages are Smalltalk [Kay 1977] [BYTE 1981] and Actors [Hewitt 1977].

2. Language Description

In this section we present an brief introduction to the EDL programming language. A more detailed description of this language is contained in the EDL Programmer's Manual [Green 1982a].

2.1. Data

There are four data types in EDL; numeric, string, sequence, and code. The two simple data types are **numeric** and **string**. The numeric data type represents numeric values, either integer or real. The string data type is an arbitrary length character string. Literals of this type are represented by a series of characters enclosed in double quotes (e.g. "string").

The only structured data type in EDL is the **sequence**. A sequence is an ordered set of elements. An element can be either another sequence, or one of the simple data values. Literal sequences are made up of sequence elements enclosed in brackets. Some example sequences are:

```
[ 1 2 3 4 5 ]

[ "a string" 10 [ "another sequence" ] ]

[ 1 [ 2 [ 3 ] 4 ] 5 ]
```

Each sequence has three special positions; first, last, and current. First and last correspond to the first and last elements in the sequence. The current position can point to any element in the sequence. It is used for scanning through sequences. The current position can be set to the first element of the sequence by the use of the **reset** operator. It can be advanced to the next element by the use of **next**.

Values at any of the three positions can be examined. There are primitives to extend the sequence at either end and to insert items into the middle of it. It is possible to pop values off the front or back of a sequence. These are the only destructive operations that can be performed on sequences.

The last data type is **code**. This data type is used to represent executable program code. This is the only program representation that is maintained by the system. In order to obtain the source form of a program the operator **makestring** is used.

Variables are assigned types when they are declared. There is no static type checking in EDL. All type checking is performed at run time (this is similar to Smalltalk [Ingalls 1978]). There are operators for determining the type of a data value.

2.2. Statements

All statements in EDL have the same basic form. A statement consists of a statement body and an optional guard. A guard consists of the keyword **if** followed by a logical expression. The body can be an assignment, an operator, or a statement list. A statement list is a list of statements enclosed in braces (ie. { and }). The guard and body parts of a statement are separated by the keyword **do**. If there is no guard the do is optional. A statement is executed if its guard is true, or if it has no guard. If the guard is false the statement is skipped. Some example statements are:

```
x=x+1;

if eq(y,20) do print(x);

if ne(x,0) do {
    z=y/x;
    x=x-1;
    print(y);
};
```

2.3. Tools and Events

The objects in EDL are called tools. Each tool is capable of processing a number of different types of events. An event has a name, and a value. This value is of a specified type. Inside its defining tool an event is associated with a statement. When the event occurs this statement is executed. An event's statement can access the value of the event through the e ent's name.

There are two types of events in this system; external and internal. An external event occurs when a user interacts with an input device. In this case the event name has previously been associated with that input device. The event's value will be the new value of the device. An internal event is generated when a tool performs a send operation. The send operation specifies the name of a tool, the name of an event, and the value of the event. This operation has the following format

```
tool_name <- event_name data
```

If more than one data value is specified, they are collected into a sequence. This sequence is then used as the event's value. Internal events are used for communications between tools, and to implement control structures.

The definition of a simple stack tool is shown in fig. 1. A tool definition consists of four sections. The first section contains the name of the tool and the definitions of its parameters. These parameters are the same as variables, except they are given a value when the tool is created. The next section of the tool contains variable declarations. All variables used in a tool must be declared. A variable is declared by specifying its name and type. The event section of the tool definition contains event declarations. An event declaration contains the name of the event, its type, a device association (if it is an external event), and a statement. The declarations for internal and external events have the following formats.

```
    on event_name : type perform statement ;

    on event_name : type assoc device_name
      perform statement ;
```

In these declarations **on, perform,** and **assoc** are keywords and device_name is the name of an input device. The last section of the tool definition contains the initialization statement for the tool. This statement is executed when the tool is created.

The stack tool's parameters specify where an event is sent if an error occurs in the operation of the stack. The only error that could occur in this tool is an attempt to pop an item off an empty stack. The only variable declared in this tool is "store". It is used to store the values in the stack.

A stack tool can process two types of events; push and pop. The push event pushes a value onto the top of the stack. It has the following format

```
    stack <- "push" value
```

We would like our stack to be able to store values of any type. In order to handle this a special type called **any** is used. An event of this type can have any value. The pop event is used for popping values off the top of the stack. The format of this event is

```
    stack <- "pop" tool_name event_name
```

The tool_name and event_name specify where the popped value is to go. In response to this event the stack tool first checks to see if the stack is empty. If it is an error event is sent to the error handling tool. The **break** operator exits a specified number of nesting levels. In this case we exit to the end of the statement for the pop event. If there is a value on the top of the stack an event is created which sends it to the specified tool.

The initialization statement for the stack tool sets the variable "store" to a new empty sequence.

```
tool stack(error_tool : numeric;
    error_event : string);

  var
    store : sequence;

  event
    on push : any perform
      add_front(store,push);

    on pop : sequence perform {
      if null?(first(store)) do {
        error_tool <- error_event pop;
        break(2);
      };
      first(pop) <- last(pop) pop_front(store);
    };

  init
    store = newseq;

endtool;
```

fig. 1

A tool instance is created by the use of the **activate** operator. The parameters to this operator are the name of the tool and its parameters. The name of the new instance is returned as the value of this operator.

To create a stack instance called "st" the following statement is used.

```
    st = activate(stack,error,"stack_error");
```

After the activate call "st" is ready to accept push and pop events. The **deactivate** operator can be used to destroy a tool instance.

3. The Implementation of EDL

The current implementation of EDL runs on a PDP 11/23 minicomputer under the UNIX operating system. The graphical display devices supported by this implementation include a AED 512 colour graphics display, a ADM 3A with retrographics, and several standard ASCII terminals. Input devices include a Summagraphics Bit Pad with a four button cursor, a joystick, and ASCII keyboards.

The EDL system consists of a compiler and an interpreter. The compiler translates EDL source programs into byte code instructions that are executed by the interpreter. These byte codes are similar in format to the ones used in Smalltalk [Ingalls 1978].

The EDL interpreter has three major components, the byte code interpreter, the storage management system, and the input sytem. All three components are implemented as a collection of routines written in 'C'.

The largest of these components is the byte code interpreter. The byte codes are executed on a stack based machine, and they include special instructions for the sending of messages. Most of the EDL compiler has been incorporated into this part of the interpreter so statements can be interactively entered and executed.

The storage management system provides a two level storage structure for objects. When an object is created the storage manager allocates space for it in main memory. If a request for storage is made when main memory is full, the virtual memory processor is called. At this point objects are swapped out onto a disc file, on a least recently used basis, until enough space is freed. Objects remain on the disc until they must respond to a message.

One of the more interesting featu~es of EDL is the input system, which generates the external events that provide the driving force in an EDL program. At regular intervals the status of all input devices is checked. If a change in the status is detected, the current state of the input device is read, and the event associated with this device is generated. This event is sent to all objects that have external events associated with the device.

4. Graphical Output in EDL

Most EDL program rely on graphical input and output techniques to communicate with their users. Two approaches have been taken to the production of graphical output in EDL. Both of these approaches will be discussed here.

Originally the display file model [Newman 1979] was used as the basis for graphical output in EDL. In this model a global data structure, called the display file, is used to structure the graphical output produced by a program. The display file is divided into a number of segments which contain the graphical primitives generated by the program. The program can specify the segments to be displayed at any point in time by using the post and unpost operators. At any time only one segment, called the open segment, can be receiving graphical primitives from the program. This model is used in most graphics subroutine packages.

The use of the display file model in EDL caused several problems. Since only one segment can be open at a time only one object can be producing graphical output at a time. In non-concurrent programming languages this is not a problem, but in EDL where several objects can be producing graphical output at the same time this was a severe restriction. It meant that when it comes to producing graphical output all the objects in a program must synchronize themselves. This is counter to the view of each object being an independent entity. When an object produces graphical primitives it needs a segment to put them in. Segments are a global resource so before an object can use a particular segment it must check to see if any of the other objects are using the same segment. Again this is counter to the philosophy of independent objects. This experience indicates that any graphics system that depends on a global data structure will not work well in an object oriented language.

The model of graphical output currently used in EDL is much simpler than the display file model. In this model the objects in a program use a pseudo display device that is capable of displaying at least two colours (black and white). There are two types of geometrical primitives in this model; lines and text. The line graphical primitive is produced by the **drawline** operator. The parameters to this operator are the two end points of the line. The text graphical primitive is produced by the standard EDL print operators. The operator **movecursor** is used to specify where the text is to appear on the screen. A graphical primitive can be removed from the display by using the **colour** perator to set the colour to zero and redrawing the primitive.

Display organization is provided by the objects that produce the display. Each object that produces an image on the display must also be capable of erasing that image and performing any other graphical operations on it. An example of an EDL object that uses this model is shown in fig. 5. Using this model it is possible to construct a set of objects that simulate the display file model.

5. User Interfaces

One of the major applications of EDL is the production of highly interactive programs. These programs make heavy use of graphical input techniques and must have high quality user interfaces. In this section we will look at how some common interaction techniques can be implemented in EDL and an example user interface.

One of the most common interaction techniques is menu selection. In this technique the user selects a command by pointing at it on a display. Usually this display consists of command names or special icons that represent commands. Menus can be handled quite easily in EDL. A menu can be viewed as a collection of rectangular areas. In decoding a menu selection a program determines which rectangle the user pointed at. The EDL operator "in" is designed to simplify this decoding process. This operator takes three operands; a coordinate value, a lower, and an upper bound. If the coordinate value is within these bounds it returns true. We can use two "in" operators to determine if a point, (x,y), is within a rectangle in the following way

```
    if in(x,xl,xu) and in(y,yl,yu) do
        print("in the rectangle");
```

This technique can be extended to decoding the whole menu. As an example, consider the menu shown in fig. 2.

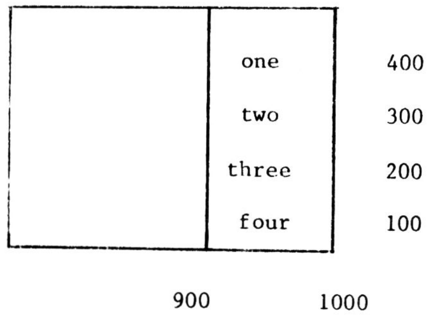

fig. 2

The positions of the commands in this menu are indicated in the figure. The input device we will use is a tablet that has a four button cursor. The button used to select a menu item is called the "z-button". An EDL tool that decodes this menu is shown in fig. 3. The menu tool's parameter is the name of the tool instance that menu hits are reported to. The first two events (xpos and ypos) defined in this tool have no statements associated with them. Their only purpose is to pick up the current position of the tablet cursor. The zaxis event does all the work in this tool. The first guard in this event guarantees that the menu is decoded only when the user presses the "z-button" and not when he releases it. The rest of the guards determine which rectangle, if any the tablet position is in.

```
tool menu(control:numeric);

  event
    on xpos:numeric assoc tabx;

    on ypos:numeric assoc taby;

    on zaxis:numeric assoc zbutton perform {
      if eq(zaxis,1) do {
        if in(xpos,900,1000) do {
          if in(ypos,50,150) do
            control <- "four";
          if in(ypos,150,250) do
            control <- "three";
          if in(ypos,250,350) do
            control <- "two";
          if in(ypos,350,450) do
            control <- "one";
        };
      };
    };
endtool;
```

fig. 3

In most applications there are several menus available to the user. The user interface contains a tool for decoding each of these menus. At any point in the dialogue only a subset of these menus will contain relevant commands. In order to prevent the user from selecting an inappropriate command only the tools decoding relevant commands are activated. A control tool is used to determine the menu decoders active at each point in the dialogue.

To illustrate the application of EDL to user interfaces consider the following simple user interface. The purpose of this user interface is to aid the user in arranging a number of simple geometrical objects on a display screen. The geometrical objects can be circles, squares, or triangles. There is a menu on the right side of the screen containing the three shapes. The user can select one of the shapes through the use of a tablet. A copy of the selected geometrical object is made and it is used as the tracking cross. The selected geometrical object can be dragged to any position in the work area and deposited there by pressing the "z-button" on the tablet. An object in the work area can be picked up again by pointing at it. It can then be dragged to another position in the work area. A picture of the display used by this program is shown in fig. 4. A formal definition of this user interface can be found in [Green 1981].

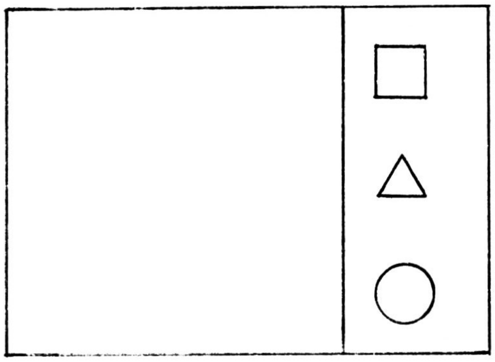

fig. 4

This user interface can be divided into four components; menu, work area, tracker, and the geometrical objects. The logical starting point for the design of this program is the geometrical object. Each geometrical object is implemented by a separate tool. A geomtrical object must be capable of performing four tasks. The first task is producing a picture of itself on the display screen. This is performed by the "draw" event. The "move" event is used to move the geometrical object around the display. The third task is detecting when the user has selected this geometrical object. This is handled by the "zaxis" event. Finally, the tool must be able to replicate itself. The "copy" event is used for this. The EDL tool definition for the triangle geometrical object is shown in fig. 5. The definitions for the other two types of geometrical objects are similar.

Defining a common communications protocol for all geometrical objects has two advantages. First, it makes it easy to add a new type of geometrical object to the program. All the details pertaining to geometrical objects are contained in the tools implementing them. Second, it makes the design of the rest of user interface easier. The rest of the user interface does not need to know how to manipulate geometrical objects or know where they are located.

```
tool triangle(owner:numeric; x:numeric;
    y:numeric; size:numeric);

  var
    seg:numeric;
    temp:numeric;
    x1 : numeric;
    y1 : numeric;
    oldx : numeric;
    oldy : numeric;

  event
    on xpos:numeric assoc tabx;

    on ypos:numeric assoc taby;

    on zaxis:numeric assoc zbutton perform
      if eq(zaxis,1) do
        if in(xpos,x,x+size) and
          in(ypos,y,y+size) do {
          owner <- "hit" self;
        };

    on copy:numeric perform {
      temp=activate(triangle,copy,x,y,size);
      owner <- "new" temp;
    }:

    on new_owner:numeric perform
      owner=new_owner;

    on move:any perform {
      self <- "draw" oldx oldy 0;
      self <- "draw" xpos ypos 1;
    };

    on draw:seq perform {
      colour(pope(draw));
      x1=first(draw);
      y1=last(draw);
      drawline(x1,y1,x1+size,y1);
      drawline(x1+size,y1,x1+size/2,y1+size);
      drawline(x1+size/2,y1+size,x1,y1);
    };

  init
    self <- "draw" x y 1;
    oldx=x;
    oldy=y;

endtool;
```

fig 5

The next component of the user interface is the menu. The tool which implements this component has two tasks. The first task is creating the geometrical objects in the menu. The other task is responding when one of the geometrical objects in the menu is selected.

This involves creating a copy of the geometrical object and passing it on to the tracker. The tool definition for the menu tool is shown in fig 6.

The tracker tool is responsible for dragging geometrical objects. It responds to two events; new_symbol, and hit. The new_symbol event occurs when an object in the menu or work area is selected by the user. The hit event is generated when the user wants to deposit the object being dragged. The tool definition for the tracker is shown in fig. 7.

```
tool menu(tracker:numeric);

  var
    circle_object:numeric;
    triangle_object:numeric;
    square_object:numeric;
    hit_object:numeric;

  event
    on hit:numeric perform {
      hit_object=hit;
      hit_object <- "copy" tracker;
    };

    on new:numeric perform {
      tracker <- "new_symbol" hit_object;
    };

  init
    circle_object=activate(circle,self,900,250,
      20);
    triangle_object=activate(triangle,self,900,
      500,20);
    square_object=activate(square,self,900,750,
      20);

endtool;
```

fig. 6

The last component of this user interface is the work area tool. The event in this tool handles hits on the geometrical objects in the work area. The tool definition for this tool is shown in fig. 8.

6. Some Applications of EDL

In this section we present three typical applications of EDL. All three of these applications involve interactive graphical input and output. These examples illustrate the range of applications that can be tackled with this language.

```
tool tracker;

  var
    work_object:numeric;
    menu_object:numeric;
    cross:numeric;
    dragged_object:numeric;

  event
    on xpos:numeric assoc tabx perform
      dragged_object <- "move";

    on ypos:numeric assoc taby perform
      dragged_object <- "move";

    on new_symbol:numeric perform {
      dragged_object=new_symbol;
      cross <- "hide";
    };

    on hit:numeric perform {
      dragged_object <- "new_owner" work_object;
      cross <- "show";
      dragged_object=cross;
    };

  init
    work_object=activate(work_area,self);
    menu_object=activate(menu,self);
    cross=activate(tracking_cross);
    dragged_object=cross;

endtool;
```

fig. 7

```
tool work_area(tracker:numeric);

  event
    on hit:numeric perform {
      hit <- "new_owner" tracker;
      tracker <- "new_symbol" hit;
    };

endtool;
```

fig. 8

6.1. A Simple Geometrical Modeling Program

The traditional approach to geometrical modeling is to use some data structure to represent the geometrical objects. A set of procedures is then produced to display and manipulate this data structure. In this section we present a program which defines a geometrical model as a collection of EDL objects. These objects represent two dimensional shapes consisting of a sequence of straight lines.

This program is implemented in EDL as a single tool. When a new object is to be added to the model an instance of this tool is created. The coordinates that define the shape of the object are stored in a sequence. Points are added to this sequence by sending the object an "add" message with the associated x and y values. When this message is received the new point is placed at the end of the sequence.

As an example consider a box with corners at (400,400), (400,600), (600,400), and (600,600). The box could be contructed by the following sequence of statements:

```
box = activate(geometric_object)
box <- "add" 400 400 ;
box <- "add" 400 600 ;
box <- "add" 600 600 ;
box <- "add" 600 400 ;
box <- "add" 400 400 ;
```

Notice that no assumptions are made about how the coordinates are being generated. They could be produced by a user sketching on a tablet, entered interactively from the keyboard, or by another object which calculates them.

Now that the box has been created we need a mechanism for displaying it on the screen. The "draw" message is used for both drawing and erasing objects. The value of the draw message is the colour that the object is to drawn in. To erase an object a "draw" message with the value zero is sent. To display the box the following message would be sent

```
box <- "draw" 1 ;
```

To erase the box we would send the message

```
box <- "draw" 0 ;
```

When the object receives the "draw" message, the object is drawn by calling the operator drawline to connect each pair of coordinates.

Transformations of the object are accomplished by sending a series of messages for scaling, rotating and translating. Each of these messages specifies the type and value of the transformation. Examples of these messages applied to the box are :

```
box <- "scaleX" factor;
box <- "move" dx dy;
box <- "rotate" theta.
```

When an object receives a transformation message each point in the sequence is transformed relative to the origin. When an object is created the origin is set to the centre of the display screen, but this may be reset with the "origin" message.

The geometric_object tool allows for quick and easy modeling of two dimensional objects. The user of this tool only needs to know which messages are needed for their application, since the sequence of coordinates is hidden inside the object.

This example program shows how objects can be used for organizing graphical data into a logical unit. The simplicity of this results from the modularity of each object.

6.2. A User Interface Prototyping System

A major application of EDL is the user interface prototyping system called PROSYS [Green 1982b]. This program provides an interactive environment for the construction and editing of user interface prototypes. In this program a prototype is construction from a set of pre-programmed building blocks. Each of these building blocks simulates a small part of the user interface. There are building blocks for most of the standard input, output, and interaction techniques. The building blocks are connected by data paths which carry the graphical information in the prototype. This approach to prototype construction makes it possible to quickly set up a prototype and change it with a minimal amount of effort.

In PROSYS each of the building blocks is implemented by a separate object. The data paths in the prototype are implemented by events. PROSYS itself is a small collection of objects that manage the objects in the prototype and provide a mechanism for creating and editing prototypes. Since the prototype is implemented as a set of objects it is possible to intermix the development and testing of a prototype. To a large extent the prototype is independent of PROSYS so special editing and testing modes are not needed. Since the building blocks are implemented as EDL tools it is quite easy to extend the set of building blocks without effecting the prototyping program.

The flexible environment provided by EDL made it possible to take this approach to prototype development.

6.3. An Interactive Tool Editor

The EDL system includes an graphical syntax based editor for tools. This editor allows the user to edit one tool at a time using any of the supported display devices and a tablet. The tool editor divides the screen into three areas called the initialization area, the event menu, and the event display. The initialization area is located in the bottom part of the screen and is used for displaying the initialization statements for the tool. The right side of the screen has a list of all the events in the tool. This list is used as a menu to select the event to be edited. The event display occupies the upper left side of the screen. This area is used for displaying the event that is currently selected. The EDL statements in this area and the initialization area are formatted by a pretty printer.

There are three basic commands in the tool editor. The first command selects the event to be edited. This command is invoked by pointing at the event name on the event menu and pressing the "z-button" on the tablet cursor. At this point a pretty printed version of the selected event will appear in the event display. The other two commands are used to edit the EDL statements in the event display and initialization area. Since the statements in these areas are pretty printed each line corresponds to a major syntactic component of a statement. The insert command is used to add lines to an EDL statement. To invoke this command the user points to the position where the new lines are to appear and presses the "z-button". He then enters the new lines through the keyboard. The delete command is used to remove lines from a statement. This command is invoked by pointing at the line to be deleted and pressing "button-1" on the tablet cursor. After these commands have been processed the new version of the statement is displayed.

The tool editor directly operates on the compiled version of the tool. This makes it possible to test the results of each editing operation without leaving the editor or recompiling the tool.

7. Conclusions

In this paper we have presented an introduction to the object oriented language EDL and have shown how it can be used in the production of highly interactive graphics programs. This language is well suited to this particular application area for the following reasons.

1) Objects are highly modular so it is possible to develop and test programs in an incremental fashion. Using the tool editor objects can be modified while the program is running.

2) The highly modular nature of tools also makes it possible to develop libraries of standard interaction techniques. This is best illustrated by the prototyping system discussed in section 6.2.

3) The concurrent nature of objects makes it easy to produce highly interactive programs that make use of combinations of devices and interaction techniques.

4) The interactive environment provided by the EDL system encourages experimentation with different interaction techniques. This stimulating environment encourages the programmer to be creative in his user interface designs.

References

[BYTE 1981] "Special Issue on Smalltalk", BYTE, vol.6, no.8, August 1981.

[Green 1981] Green M., "A Methodology For the Specification of Graphical User Interfaces", Computer Graphics, vol.15, no.3, p.99, 1981.

[Green 1982a] Green M., "The EDL Programmer's Manual", TR 82-CS-01, Unit for Computer Science, McMaster University, 1982.

[Green 1982b] Green M., "Towards a User Interfa Prototyping System", Graphics Interface'82 Proceedings, 1982.

[Hewitt 1977] Hewitt Carl, "Viewing Control Structures as Patterns of Passing Messages", Artificial Intelligence, vol.8, no.3, 1977.

[Ingalls 1978] Ingalls Daniel H. H., "The Smalltalk-76 Programming System: Design and Implementation", Proceedings of the 5th Annual ACM Symposium on Principles of Programming Languages, 1978.

[Kay 1977] Kay Allen C., "Microelectronics and the Personal Computer", Scientific American, vol.237, no.3, Sept. 1977.

[Newman 1979] Newman W.M., R.F. Sproull, Principles of Interactive Computer Graphics, 2nd Edition, McGraw-Hill, 1979.