CODEN: LUTFD2/(TFRT–7328)/1-031/(1986)

Implementation of Graphics for HIBLIZ

Dag M. Brück

Department of Automatic Control
Lund Institute of Technology
October 1986

Department of Automatic Control **Lund Institute of Technology** P.O. Box 118 S-221 00 Lund Sweden	*Document name* Report
	Date of issue 1986-10-16
	Document Number CODEN:LUTFD2/(TFRT-7328)/1-031/(1986)
Author(s) Dag M. Brück	*Supervisor*
	Sponsoring organisation The National Swedish Board of Technical Development (STU contract 84-5069)

Title and subtitle

Implementation of Graphics for HIBLIZ

Abstract

HIBLIZ (meaning Hierarchical block diagrams with information zooming) is a simulator for dynamical systems. Hierarchical block diagrams are used for representation of structural properties: hierarchical decomposition, interconnection structure, and structure of the interfaces between the modules and their environment; information zooming is used for abstraction.

An important part of the project "New Forms of Man-Machine Interaction" was the implementation of graphics for HIBLIZ. Firstly, HIBLIZ was moved from a VAX to a Silicon Graphics IRIS 2400 workstation. Implementation of driver routines, generation of text and portability problems are discussed. Experiences are presented.

Secondly, a new graphics implementation was designed, starting with an investigation of required operations. Interesting areas such as the use of coordinate systems, availability of hardware capabilities, segment handling, rubber-band drawing and picking are discussed in more detail.

Lastly, availability and applicability of current standards (GKS and PHIGS) are discussed. The relationship to object-oriented programming, and its possible usefulness is noted.

Key words

Computer Aided Control Engineering, Man-Machine Interaction, Computer Graphics, Information zooming

Classification system and/or index terms (if any)

Supplementary bibliographical information

ISSN and key title		*ISBN*
Language English	*Number of pages* 31	*Recipient's notes*
Security classification		

To Birgitta, my better half.

It's clever, but is it art?
Rudyard Kipling

1. Introduction

The user interface is perhaps the most important part of a computing system. The design of the user interface directly influences the perceived friendliness and usefulness. The purpose of the project "New Forms of Man-Machine Interaction" was to explore different ways to design and implement user interfaces for computer-aided control engineering [Mattsson et al., 1986]. A simulator for dynamical systems was used as a test-bed during the project. The simulator is called HIBLIZ, meaning HIerarchical BLock diagrams with Information Zooming.

The topic of this report is the implementation of graphics for HIBLIZ. The report is divided into five major sections. This introduction includes a description of the HIBLIZ system in general. Section 2 deals with the first crude implementation on an IRIS 2400 workstation from Silicon Graphics, Inc. The third, and most important, section contains a discussion of required operations and the realities of implementing these. Section 4 briefly covers the area of graphics standards and object-oriented programming. The appendix contains definitions of all operations in the second implementation.

The reader is assumed to have a basic knowledge of computer graphics; otherwise, one of the standard textbooks is recommended [Foley and van Dam, 1984], [Newman and Sproull, 1979]. There is also a shorter and somewhat modernized book in swedish [Gudmundsson, 1985].

Overview of HIBLIZ

HIBLIZ is a simulator for dynamical systems. Computer graphics is used for representation of structural properties of the model: hierarchical decomposition, interconnection structure, and structure of the interfaces between modules and their environment. The decomposition and interconnection structures are described by hierarchical block diagrams. Modules at the lowest level are described by equations.

One of the key concepts introduced in HIBLIZ is information zooming. A common problem of systems of any kind is complexity; the human is unable to deal efficiently with many entities concurrently. A general way to deal with complexity is abstraction.* Using multiple levels of abstraction, the model may be viewed in more or less detail. At the first level, a name or icon may suffice. The second level may describe the usage of a module, and the third level the implementation.

A model of a thermal power plant is used as an example to visualize the concept of information zooming. In HIBLIZ, a view of the total model will only show interconnected annotated boxes (Figure 1). As the user zooms closer, the boxes will change representation and show internal structure with increasing detail (this is called "open up"). In this case, we first begin to see

* The essence of abstraction is to extract essential properties while omitting inessential details. From Grady Booch: Software Engineering with ADA, Benjamin/Cummings, 1983.

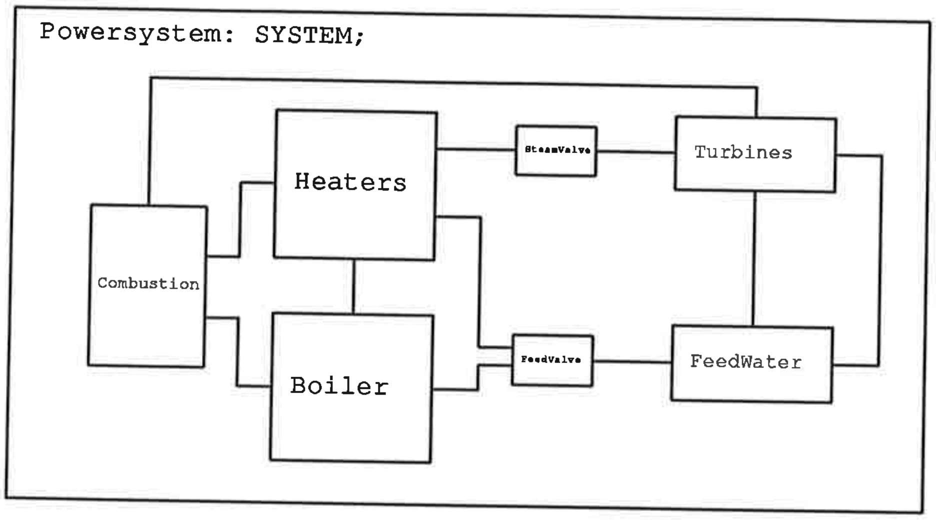


Figure 1. The power system model.

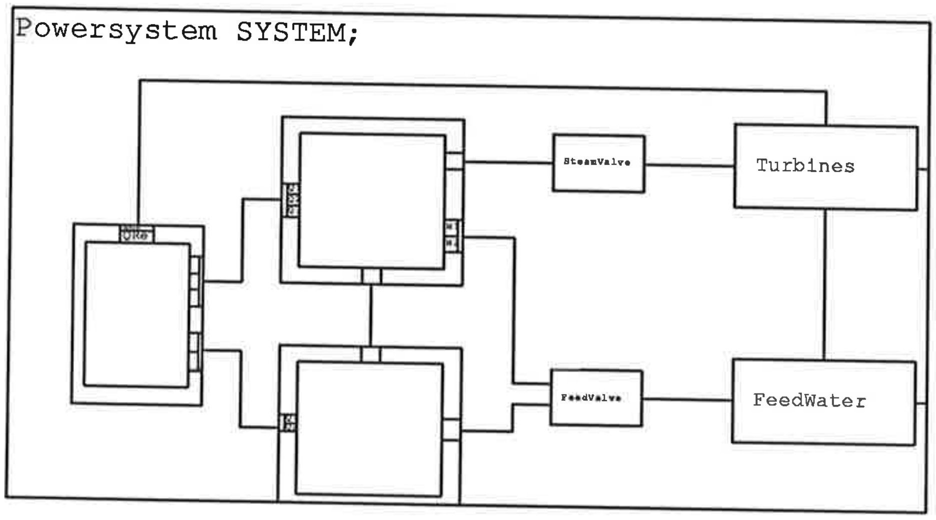


Figure 2. The power system model zoomed-in a little.

the definition of the interfaces between modules (Figure 2); a little later, the internals of the boiler are becoming visible (Figure 3). In Figure 4, we see that the boiler is represented by a new block diagram.

It turns out that a single window to the model is not enough. In particular, when block diagrams at multiple levels were used, the user too easily lost track of his work. Consequently, overlapping windows are available (Figure 5). Size and location of the windows are controlled by the user. The combination of information zooming and multiple windows at different levels of abstraction has turned out to be very powerful.

The interaction model of HIBLIZ requires powerful graphical capabilities. With a mouse, the user can scroll (move vertically), pan (move horizontally) and zoom the picture on the screen in real time. Double buffering provides instant screen update; the user does not have to watch the graphical objects appear on the screen one-by-one. In combination with fast graphics, animation

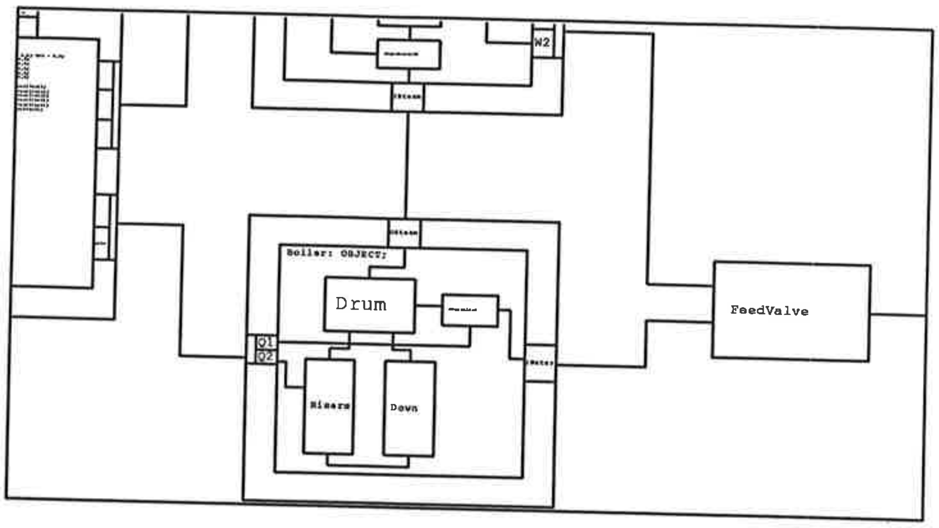


Figure 3. The power system model zoomed-in further.

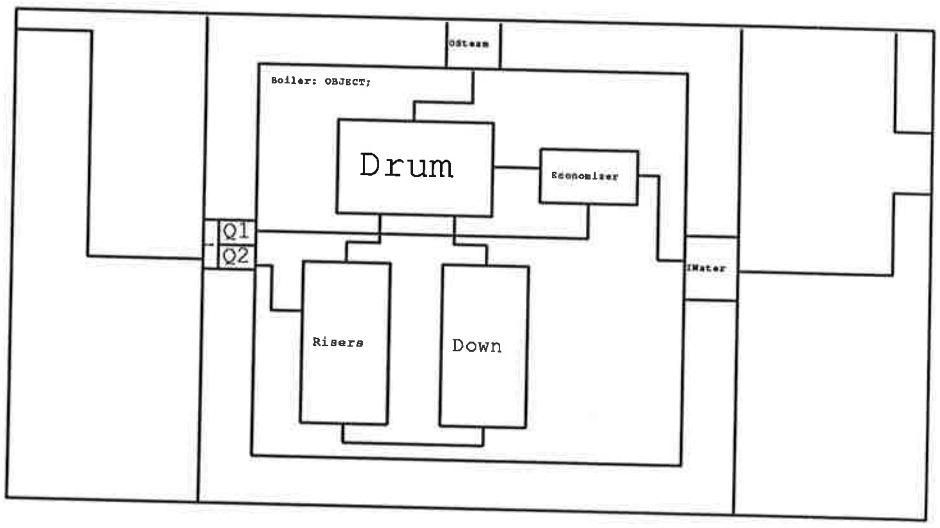
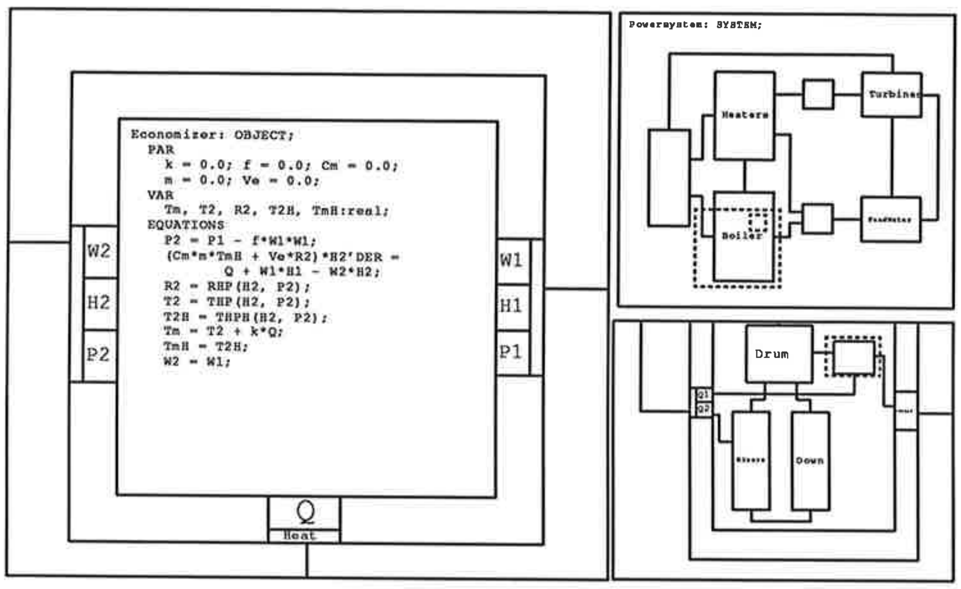


Figure 4. The internals of the boiler.

is possible. In particular, when scrolling and panning, there should be no delay between the movement of the mouse controlling the operation and the screen update.

Using the HIBLIZ system, the user can interactively draw modules and make connections between modules. Equations are edited using a built-in text editor. A special-purpose language describes the system when stored on disk. The language has constructs to describe both the equations and the graphical layout. The file can be edited outside the HIBLIZ system using any editor.

Behind the curtains is a simulator for dynamical control systems. A system is described by sets of ordinary differential equations and algebraic equations. The simulator uses the differential/algebraic system solver DASSL [Petzold, 1982]. To decrease the order and the complexity of the numerical problem, simple symbolic formula manipulation is performed [Mattsson et al., 1986].



```
Economizer: OBJECT;
  PAR
    k = 0.0; f = 0.0; Cm = 0.0;
    m = 0.0; Ve = 0.0;
  VAR
    Tm, T2, R2, T2H, TmH:real;
  EQUATIONS
    P2 = P1 - f*W1*W1;
    (Cm*m*TmH + Ve*R2)*H2'DER =
           Q + W1*H1 - W2*H2;
    R2 = RHP(H2, P2);
    T2 = THP(H2, P2);
    T2H = THPH(H2, P2);
    Tm = T2 + k*Q;
    TmH = T2H;
    W2 = W1;
```

Figure 5. Multiple windows at different levels of abstraction.

The HIBLIZ system has its roots in LICS, a Language for Implementation of Control Systems [Elmqvist, 1985]. Major parts have been reused in HIBLIZ, for example the design of the user interface. The LICS system was implemented on a VAX–11/780 with a Matrox graphics board connected to the UniBus. The need for fast graphics to facilitate scrolling, panning and zooming could not really be fulfilled with this configuration. HIBLIZ is implemented on an IRIS 2400 which has much faster graphics [Silicon Graphics Inc., 1986]. Another important extension is the simulator with routines for solving differential/algebraic equations.

2. The first implementation

This section describes how the graphics routines of LICS were moved from VAX to an IRIS workstation. This first phase gave valuable experiences which formed the basis for future developments.

Structure of the HIBLIZ program

The HIBLIZ program is large: it consists of some 28 000 lines of Pascal code. The code has been broken up into 164 modules, so that related constants, types, variables and procedures are grouped together. Since "standard" Pascal requires a fixed ordering of labels, constants, types, variables and procedures, a pre-processor called Packman is used.

Packman accepts files consisting of sections of code preceded by headings such as `.LABEL`, `.CONST`, `.TYPE` and `.VAR`. The code is sorted, and Packman outputs the contents of all sections preceded by `.CONST` to a file named CONST.SEC and so on. The main program typically just includes the files produced by Packman. There are also control directives and powerful facilities for separate compilation (given the right support by the compiler). Packman requires 5 minutes 28 seconds to pre-process the code, and the Pascal compiler

12 minutes 23 seconds to generate an executable version. This corresponds to a compilation speed of 26 lines/second.

Further, internally the program is designed like a semi-transparent onion. There are at least six conceptual layers of code, each representing some level of abstraction. Although this is not always the case, it is not uncommon that simple operations require six levels of procedure calls. The onion is semi-transparent in the sense that it has been recognized that the mapping from one level to another is often trivial; occasionally, one or more levels are by-passed (in an apparently random fashion).

At the lowest level is a rather small number of graphics driver routines. When porting HIBLIZ to a new computer, only the driver routines need to be re-written. The computer must provide a minimal set of drawing primitives, of course, but HIBLIZ has also been designed to cope with a wide range of driver capabilities. When the graphics device is opened, the capabilities are defined (using a Pascal set of pre-defined options); these are then used "above" the driver routines to minimize the work done by HIBLIZ. For example, scaling and clipping may be done either by HIBLIZ or in hardware. Multiple devices (e.g. a logging device) are supported by means of a case statement in the driver routine.

Implementation of driver routines

The first goal was to get a crude version of HIBLIZ up and running. Due to the way graphics was implemented in LICS, it was easier not to use hardware scaling and clipping. Some driver routines are not used (with our examples, anyway), so a few routines were never implemented.

The function of some routines was quite easy to comprehend. DrawLine and DrawRectangle received scaled and clipped data via a global record, which was briefly documented. Some routines used for drawing polygons and ovals, or for bit-level raster operations, were on purpose never implemented (they were not used in our examples).

Other routines were much more difficult. Due to missing (or useless) documentation, the range and meaning of many parameters are still not understood. For example, when defining colors in the look-up table, the red, blue and green components are real numbers, but the valid range was not documented. Cursor control was also difficult: unexplainable cursor positions (e.g. $-1000, -1000$) sometimes occur. The cursor now approximately follows the position maintained internally by the program, but the algorithm is probably wrong and picking can be difficult. It was often necessary to trace the execution and to look at actual parameter values using the logging device.

Other routines were still harder: in particular, initialization of the graphics device was extremely badly documented. There is a special routine DefineDevice which should be used to set device characteristics, but it was impossible to make it work. Luckily, there is a default setting available; but then some coordinate data has to be re-scaled once again in the driver routines.

Finally, hardware clipping was introduced. Speed and "smoothness" of the system improved noticeably, but some new bugs arose as well (e.g. the menu background suddenly disappeared) — a clear indication that the intended operation of the driver routines was not well understood. Implementation of hardware scaling and translation was not considered worthwhile when the extent of forthcoming problems was realized.

Text generation

To make full use of zooming in HIBLIZ, it must be possible to display text of many different sizes, from about 2×2 pixels to almost 1000×600 pixels. There are two possibilities to implement new fonts on the IRIS. New raster fonts can be defined, and the IRIS has a 32K byte memory to store fonts, patters and cursor shapes. This method is not applicable for large fonts. Larger characters have to be viewed as graphical objects consisting of straight and curved lines.

A support program to develop and generate new fonts was available from the LICS project. The program is based on the ideas given in [Knuth, 1979]. The user defines the shape of a character as a number of line segments, and the size and form (rectangular, circular, oval etc.) of the pen to be used. A line segment is defined by its start and end points, and its tangents in these points. Intermediate points on the line segment (e.g. in an 'O') are defined by a cubic spline function [Knuth, 1979, pp. 24–26]. Dr. Sven Erik Mattsson moved the font generation program to the IRIS with great difficulty. In particular, the use of splines on the IRIS was non-trivial, mainly due to incomplete documentation. It was even necessary to contact Silicon Graphics in the U. S. to get all problems sorted out.

Since the font memory is limited to 32K bytes, the only way to display large characters is to draw one line segment after the other. However, there are at least two good reasons for also having raster fonts. Firstly, when drawing a small character, quantization may deform the character so it looks ugly and is difficult to recognize. Its shape may also change when it is moved over the screen.

Secondly, it is important to make the graphics as fast as possible. When the characters are small, there may be many of them on the screen. If the characters are defined by cubic splines, this implies many calculations. To draw an 'O', four splines and at least 20 points are needed. The IRIS can at maximum generate 67 000 vectors/second, whereas it can display up to 150 000 raster characters/second.

Implementation of text output was quite easy as it was possible to use almost everything of Dr. Mattsson's work without changes. All I had to do was to construct a "switch" which could select raster fonts of correct size, or possibly drawn characters. Character strings are clipped in a strange way on the IRIS: if the first character should be clipped, all the other characters are clipped too. Therefore, characters had to be drawn/printed one-at-a-time with move operations interspersed. The effective speed is therefore lower than suggested above. Drawn characters were made faster by a pre-processor which computed the splines and issued appropriate move and draw commands.

Experiences

The implementation process would have been easier if the author had had a reasonable experience with computer graphics, the IRIS Graphics Library, or the LICS program; as it were, this was not the case.

The difficulty in maintaining, or even comprehending, a program of this size (or rather parts of it), was made all too clear. The programming style where information is abstracted and hidden as far "down" as possible, in combination with the large number of modules, has also resulted in some duplication of code; neither the division of labour between different modules, nor the differences between the many abstraction levels of the code are clear. In the LICS project, many ideas were generated and tested, and the existing code contains rudiments of such tests. Clearly, these rudiments do not make the code easier to understand. Much code is probably never executed.

Worst of all, the documentation is over-terse. When the project started, there was no documentation or report whatsoever. Later, the LICS-report [Elmqvist, 1985] arrived, but a much more specific description of the program and the design philosophy would have been much needed. Comments in the code are rare: in the 28 000 lines of Pascal code, there are about a thousand comments, most of which convey little or no information (procedure CreateView: *"Creates a new view"*). Although the code is clear and quite well written, it is nevertheless very difficult to modify it when the overall structure, operation, dependencies, etc. are undocumented.

In spite of all these deficiencies, the program was quickly moved from the VAX to the IRIS workstation. With approximately 3 man-months of work, a decent version was running. This probably implies that is was originally written by skilled programmers with portability in mind. The quality of the Pascal compiler and the UNIX system itself made life easier.

A fundamental problem with the first implementation of graphics in HIBLIZ was efficiency. The IRIS can make all transformations from local coordinates to pixel coordinates in hardware, and also support multiple hierarchical coordinate systems, but these facilities were not used. It was also considered possible to simplify the implementation while maintaining portability.

Furthermore, the use of segments in HIBLIZ (see *Segment handling* in Section 3) makes the program larger and more complex; an additional data structure has to be maintained. Because the segment handling in the IRIS Graphics Library is not used, no speed-up is achieved.

Consequently, a complete re-design of the graphics routines in HIBLIZ was considered necessary.

3. The second graphics implementation

The second phase of the project was devoted to developing new graphics routines. This section summarizes what operations are required to interactively display and manipulate the hierarchical block diagrams used in HIBLIZ. Later, some of the more interesting issues are discussed, indicating what direction the resulting implementation took. Some notes on how the new graphics was incorporated into HIBLIZ are also made.

Required operations

Returning to the example in Section 2, we see that to draw block diagrams in HIBLIZ, only straight lines, rectangles and text are needed; circles or ovals are not used (figure 1). Area fill could have been used to give modules colored backgrounds, but currently the module backgrounds are just erased.

Color is desirable: it can be used for qualitative encoding, to denote objects of different kinds or certain properties, or quantitative, e.g. to show temperature or flow rate. In HIBLIZ, modules, interfaces, connections and text have different colors; quantitative encoding is not used. Very much information can be encoded with color: the effective transfer rate from the eye to the brain is approximately 10 Mbits/second — comparable to a large disk memory. Highlighting, i. e. making an object extra "visible" by temporarily changing its color, is useful especially when picking objects.

What happens when the user zooms in on a module? When the module opens up, we see connections and other modules (Figure 3). The enclosing module defines a local coordinate system, and every object is positioned and scaled using local coordinates. Since each level in the model hierarchy defines a new coordinate system, operations on the coordinate systems are very frequent. Fast implementation is essential.

Because of the strict encapsulation by modules, it is necessary to clip the graphics inside a module at the edges. For example, very elaborate definitions on equation form are sometimes textually larger than the module. When multiple windows are used, the window manager must use clipping too, of course.

Text output has been discussed in *Text generation*, Section 2. To get smooth zooming, text must be displayed with almost continuous change in size, from about 2×2 pixels and upwards. Occasionally, much text is displayed simultaneously, so speed is very important. With small fonts, special care has to be taken to avoid malformed characters.

The most important form of user input in HIBLIZ is picking. The user points at something on the screen with the cursor, and when he/she presses one of the mouse buttons, HIBLIZ intercepts the object.

A model is often quite large, but the user has normally zoomed in on a smaller part. Knowing this, the program can avoid traversal of major parts of the internal data structure (in our case a tree), and only issue graphics commands that really update the screen. Systematically avoiding traversal of branches in the model hierarchy can be called pruning, although the tree is never destroyed. The strictly hierarchical encapsulation mechanism in HIBLIZ makes this task straightforward to implement, but it must be possible to interrogate whether a module is visible or not. Also, because the representation of modules depends on the final size when displayed on the screen, size calculations must be performed frequently.

Depending on what kind of operation the user is performing, the graphics implementation (hardware and software) must work in a number of fundamentally different ways. The implementation therefore has local state; it is always in one of five modes.

- Direct mode means that drawing commands (MoveTo, LineTo, etc.) perform an action directly on the screen. The effects of these actions are destroyed when the screen is erased. Direct mode is effective when the conditions for the other four modes are not satisfied.
- In rubber-band mode the existing picture on the screen is overlaid with new graphics commands. Rubber-band mode is like drawing on an overhead slide. This allows fast update of certain graphical objects on the screen while the background is un-touched.
- In segment mode, graphics commands are not directed to the screen, but stored in an internal data structure. The screen can be updated with all graphics commands stored in a segment in one stroke.
- Hardcopy mode directs the graphics commands to an external file. They are then processed by a separate program to obtain PostScript code which can be directly printed on an Apple LaserWriter.
- Pick mode is a form of input. The user points at something on the screen with the cursor, and the program intercepts an object suitable in the current context.

The more interesting operations like rubber-bands, segments and picking are discussed in detail below.

Coordinate systems

A module defines a master space (or master coordinate system), which is addressed using master coordinates. Several modules may be combined to form a composite module by mapping their individual master spaces to a space based on a new origin. The exception is of course the top module (the root of the hierarchy), which is located in a final master space called world space, addressed using world coordinates. The world space is defined, not by explicit transformations, but by the windowing facility, which maps an area of world space onto the screen.

In HIBLIZ, normalized master spaces are used through-out. The lower left corner is located at $(0, 0)$ and the upper-right at $(1, 1)$. To allow smooth zooming and flexible positioning, real coordinates rather than integer coordinates are used. Real coordinates are rather unusual in the kind of interactive graphics we are concerned with, mainly because of the heavy computational load. On the IRIS, all computations are normally performed by the hardware graphics pipeline, so full speed can be maintained. In HIBLIZ the Pascal program and the rasterization hardware, located at either end of the pipeline, are the limiting factors.

On a low level, the objects are mapped onto the screen, which is addressed using raster coordinates (real numbers). These are then rounded to integer pixel coordinates. The transformation from world space is a function of display resolution and physical window size. It may also be convenient to introduce a normalized space between world space and raster space. The raster space is also used for cursor position, mouse input, etc. which are normally not constrained by window limits or clip areas.

Two basic graphics primitives are Scale and Translate. By combining these primitives, more complex modelling transformations can be generated to express relationships between different parts of a complex object. The Translate command positions the master space in world space (or in an enclosing master space). Scale determines the size of master space in world space.

Projection transformations define the mapping from world space (world coordinate system) to the screen (raster space). In the two-dimensional case which is used in HIBLIZ, this is a simple scaling and translation. The main difference is in the way the projection is defined: it is controlled using the windowing facility, which for example means that the model can be viewed in multiple windows with varying magnification.

A transformation on the IRIS is expressed as a 4×4 floating point matrix. The reason is that the IRIS has been designed for three-dimensional graphics. Combined transformations can be built by concatenating a series of primitive ones, i. e. multiplying their transformation matrices. If M and P are modelling and projection transformations, then the combined transformation S that maps master space directly to raster space can be formulated

$$S = MP$$

A combined transformation matrix is formed in two steps: first the projection transformation is loaded by the windowing routines (CreateWindow and SetWindow), then the modelling transformation (Scale and Translate) is used to multiply the current transformation. Multiple modelling transformations are applied as the program descends through the graphical hierarchy. Only one projection is made in each window, before any modelling transformation. There is always a current transformation, which is applied to all coordinate data.

Virtual hardware

Although the hardware on which the graphics package is implemented should have no or minimal effect on the definition of the routines, it is nevertheless worthwhile to use one of the oldest tricks in the computing business. A hardware concept, or virtual hardware, is introduced as a means of explaining the function of the graphics routines. It is hoped that this virtual hardware can be implemented on most computers.

The hardware has two screen buffers. The image that is displayed on the screen at any moment is stored in the front buffer, while actual drawing takes place in the back buffer. A special routine is used for swapping the contents of these two buffers, so that a freshly created image can be displayed. As has been noted above, the subjective speed is very important. Double buffering means that the user does not have to see the screen being cleared and every object drawn, but only small, instant updates of the screen — provided that the total re-generation time is short.

Each buffer contains a number of bit-planes, which means that every pixel on the screen is associated with some limited number (say 0 to 63). This number is used to index a color map which determines the color of the pixel on the screen; the actual color is defined at initialization by a special routine which loads the color map. For every color index, there are actually two colors; one "normal" and one "highlighted" color. Both must be defined by the user.

When drawing, master coordinates are transformed to pixel coordinates, using the current transformation. Then the scan conversion determines which pixels make up the line; these pixels are associated with the current color by writing a color index into the bit-planes of the back buffer.

Implementation of basics

Many operations are very simple to implement on top of the IRIS Graphics Library. All drawing and screen manipulation routines, as well as mouse input, are there. The color map is initialized by specifying the normalized red, green and blue components for each color in turn.

Routines for modifying the transformation matrix (Scale and Translate) are provided in hardware. It is also possible to directly access the transformation (this is necessary when the program returns from lower levels in the graphical hierarchy). Backward transformations, for example to calculate the cursor position in master coordinates, are not available in the graphics library, but had to be implemented separately. Pixel coordinates are de-scaled, the current transformation matrix inverted, and then the equations are solved for master coordinates (x, y). As can be imagined, this is a rather resource-consuming operation.

The simplest and most well-defined operations to save and restore the current transformation are Get and Set, which just copy to/from local storage. Regrettably, Get cannot be used in segment mode, and Set produces funny results. The transformation used by Set is fixed when the segment is created, so if you Set the transformation in segment mode, it will work fine the first time, but when you change the window limits (for example when panning) a new transformation is generated. When the segment is redrawn, the old transformation based on the old window is used again.

To avoid these problems, the operations Push and Pop are provided. The transformation stack is rather small though (32 levels today), so it must be used with caution.

Text output was implemented with the routines developed for the earlier version (see *Text generation* in Section 2). Raster fonts are available in sizes from 2×2 pixels to 16×20 pixels; the program selects the nearest smaller font. The graphical font is stored using one segment per character. All splines are pre-processed, so only Move, Draw and Rectangle calls are made.

Rubber-band drawing

As outlined above, rubber-bands can be viewed as a slide that resides on-top of the currently shown image. While this may be a pleasing metaphor, it is not readily implementable on most hardwares. Therefore, our virtual hardware must use some other concept which has the same properties.

One commonly used implementation of rubber-bands is XOR-drawing; the image is drawn once to be displayed (modifying the buffer), and then once again to restore the buffer to its previous state. Regrettably, not all computers provide XOR directly on the buffers. As this operation is likely to be very difficult to implement on computers without hardware support, operations like XOR are not supported by the virtual hardware. It should be noted that XOR-drawing is less useful with color graphics as the colors often change in an apparent random way.

There are also some simple, but resource-consuming ways to implement rubber-bands: Firstly, it is possible to force the application program to redraw the entire screen for every movement of the rubber-band line. The disadvantage is mainly that a description of how to create the image must be maintained by the application, and then be interpreted either by the graphics package, or the application itself. Secondly, we may copy the front buffer bit-by-bit to a separate area. The saved image is then used to re-create the original image when the rubber-band is moved. This may be a feasible solution, unless it is too slow or requires too much storage. It is definitely the brute-force way to solve the problem.

A more sophisticated rubber-band implementation may use either the back buffer, or a surplus of bit-planes, to create the slide effect. If both the front and back buffers can be displayed simultaneously, implementation is straightforward. If there are unused bit-planes in each buffer, and if it is possible to update some bit-planes without altering the others, implementation should also be quite simple.

The conclusion must be that the properties of rubber-bands can span over a very wide range, depending on implementation style and hardware support; portability is a major concern. The only solution found so far is to severely restrict the number of allowed operations with rubber-band technique.

In HIBLIZ the supported use is as follows. The application enters rubber-band mode by calling RubberMode. Now the virtual hardware miraculously changes shape. The current image on the screen is frozen and will not be changed. The application may then issue a number of either RubberLine or RubberRectangle calls, which draws an image on the slide. The old contents of the slide is automaticly erased each time. When the user is fed up with "rubber-banding," another call to RubberMode restores the original state of the virtual hardware.

The IRIS does not support XOR-drawing, neither is it possible to display the contents of both the front buffer and the back buffer simultaneously. On the other hand, the bit-planes can be selectively masked off. In the second implementation, each buffer is further divided into two halves; one is used in direct mode for normal drawing, the other for rubber-bands. The back buffer is not used in rubber-band mode.

Segment handling

A segment contains a sequence of stored graphics commands that can be invoked more than once. By packaging these commands in a segment, much of the overhead in the binding between the programming language and the graphics implementation can be avoided. Each graphics command is "compiled," i. e. the the operation and its parameters (as defined in the current environment) are stored in the segment. Calls to a segment display the contents directly without further compilation.

In addition to significantly increasing efficiency, segments are also useful as a structuring tool, i. e. for grouping related graphics commands into a single entity. Global operations (e.g. Scale and Translate) can be applied to a segment, and segments often have attributes that apply to all graphics commands in the segment (visibility, definition of colors). From a programming point of view, it is convenient to regard graphics as a data object (a reference to a segment) belonging to some element in the internal model, rather than a procedural description or piece of code that must be executed.

One of the topics for argument is whether the segment store should be structured or un-structured (flat). Flat segments are simpler and perhaps more efficient, but in more elaborate applications the user must provide some complementary data structure himself. HIBLIZ, with its hierarchical internal structuring of the model, would lend itself naturally to an advanced segment store, preferably hierarchical. But there are some fundamental problems with segments, where the answer is not obvious:

- When the graphical data structure is traversed, it is often necessary to read auxiliary data. For example, when the program moves from one level to another in the block diagram, the current transformation must be stored, so it can be restored between multiple children modules. By using a stack of transformations, this particular problem could be circumvented, but segments do generally not provide any mechanism for reading data. See also *Implementation of basics,* above.

- It is sometimes convenient to be able to make decisions and take alternate actions while presenting the graphics. A typical example is highlighting an object that was picked by the user. A natural implementation is to traverse the graphical structure, and when it is time to draw the picked object (*CurrentObject* = *PickedObject*), the color is changed.

  This procedure means that the display operation must be temporarily suspended, and later resumed. Suspend/resume of segments is probably not available in any graphics package, but in for example GKS, it is possible to prevent a segment from being displayed without destroying the segment. One possible solution (available in C on the IRIS) is to insert calls to user-defined procedures into the segment, which then becomes a hybrid between graphical data and code.

- Implementation of information zooming combines these two problems: depending on the size of the module when it is displayed on the screen (a function of window size and zooming), the program must choose between three different representations. Except when the most detailed representation is used, modules at lower levels will not be displayed. This means that the internal graphical data structure can be pruned, leading to an early return to higher levels.

  On the IRIS, it is possible to prune a hierarchy of segments (segments may call segments in the IRIS Graphics Library), but conditional "execution" is not.

These problems have an aspect that is of general interest when designing graphics hardware. In this case, the basic difficulty of making an efficient implementation is that the graphics hardware must return a result before the application program can continue. In normal drawing of a picture, the application program and the graphics processor can work rather independently. But to read information from the graphics processor (often a pipeline architecture), the two processors must synchronize, which leads to less parallelism and decreased throughput.

For several reasons, the design of the new segment handler produced a very simple definition: segments are flat, have no global attributes, cannot be suspended and later resumed, and do not return values. Firstly, all the details of how segments are used (or can be used) in HIBLIZ were not known at design time. Secondly, a complex segment handler would be difficult to move to another computer, and possibly be difficult to map onto some existing graphics standards. Thirdly, time constraints prohibited the development of a more powerful segment handler.

Segment peculiarities

The operations for creating a segment in HIBLIZ have a peculiar syntax and semantics, which gives some practical benefits. The use of segments is best explained by an example:

```
if CreateSegment(seg) then begin
    ClearBuffer;
    SetColor(1);
    Rectangle(50, 50, 950, 950);
    CloseSegment;
end;
```

CreateSegment will check if segment number `seg` exists. If it exists when CreateSegment is called, the segment is displayed and CreateSegment returns false, i. e. the graphics commands are not executed. If not so, the graphics system enters segment mode and CreateSegment returns true, i. e. the code inside the if-statement is executed. In segment mode, the graphics commands (ClearBuffer, SetColor and Rectangle) are stored in the specified segment. CloseSegment will then exit segment mode, and finally display the segment, just as if no segment had been involved at all. Only one segment may be open at a time, i. e. it is not possible to create another segment while the program is executing between CreateSegment and CloseSegment.

In this simple way the following benefits are achieved: Firstly, the program is written almost exactly as if no segments were available; there is no special case if the segment does not exist. Secondly, re-generation of the segment is simple; the segment is deleted and the ordinary code executed again. Thirdly, the implementor may choose not to implement segments at all, in which case CreateSegment should just return true, and CloseSegment do nothing. Without segments, the code inside the if-statement is executed (in direct mode) every time, creating the desired image. This option makes a quick implementation easy. Fourthly, because segments can be "turned off" without changes in the application program, the efficiency of segments can easily be tested.

Programs that are supposed to be portable must fulfill two requirements. Firstly, the program must not assume that a segment exists, i. e. it must always have the information (context, local state, whatever) to be able to create the segment. Secondly, the segment must be deleted as soon as the information for creating the segment is changed. Otherwise, an implementation with segments will display the old (invalid) contents of the segment, while an implementation without segments will create the new image.

The implementation of pruning on the IRIS uses segments in an interesting way. The key operation in information zooming is to determine whether a module is visible on the screen or not (either outside the window, or too small).

The IRIS Graphics Library has an operation (called bbox2) specifically tailored for segments, that will abort the execution of a segment if a specified rectangle is invisible. Regrettably, the result of a bbox2 is not available to the application program. Inspired by the bbox2, a Pascal function VisibleBox was designed, which will take the master coordinates of a rectangle and a minimum feature size in pixels, and return true if it is visible and false if not.

VisibleBox will create a temporary segment with the appropriate bbox2 and a command to set the color black. Then VisibleBox will set the color white, and execute the segment. The outcome of the bbox2 can indirectly be determined by checking the current color.

Calling VisibleBox with different parameters, it is possible to choose the right representation of modules, and to prune the internal graphics structure. Using segments for implementing VisibleBox on the IRIS was significantly easier and noticeably more efficient than making backward coordinate transformations. A small cache with segments for the most common boxes improves speed considerably. The cache has an 80% hit-rate for large models.

Picking

Picking is normally the operation where the user points at an object on the screen with a cursor controlled by a mouse, and presses one of the mouse-buttons. The program then reads the cursor position and tries (very hard!) to find out what the user was interested in. It should be noted that mouse-controlled picking is only one of many possible interaction techniques [Foley, Wallace and Chan, 1984].

There are two basic problems involved in picking. Firstly, the user may not point exactly at an object, or when the object is very small, the cursor may hide the object. It is therefore common to define a "hot-zone" around the cursor, typically 10×10 pixels large. Any object inside the hot-zone is picked (this is called a pick-hit).

Secondly, if many objects are close to each other, or because of the size of the hot-zone, it is possible to pick more than one object. The program must then either conclude that one object is more likely to be picked than the others, or request the user to pick more carefully (for example, by first zooming in). If the user's intentions are known, for example after a certain operation has been selected from a menu, the program can often restrict the picking to apply only to a smaller set of objects, thereby resolving possible conflicts (cf. top-down parsing). If a "post-fix" approach is used, i. e. the user first picks an object and then selects an operation, conflict resolution becomes difficult to program. In any case, the user is very much helped if he gets instant feedback. The minimum requirement is some sort of highlighting when he has picked an object. Even better is if every object is highlighted as the cursor moves over it, before the user has committed himself. Continuous feedback virtually eliminates picking errors, at the cost of a heavy "idle" workload.

Picking can be implemented in many ways. The obvious and most portable solution is to traverse the internal graphics structure, and for every object check if it enters the hot-zone. This is in practice very cumbersome. For every module, the cursor position has to be re-calculated because the coordinate system has been changed. The shape of every object must be interpreted; for example, every side of a module rectangle must be checked separately, while an interface has an H-shaped structure. The traversal is similar to the one used for displaying graphics, but must for practical reasons be implemented as a different procedure; this leads to software maintenance problems.

The other extreme is to use a complete hardware implementation of picking. On the IRIS, the user sets a marker at appropriate points in the program, which is copied to a special "pick-buffer" each time a pick-hit occurs. Identification of a pick-hit is done by hardware. The marker consists of a variable number of 16-bit integers [Silicon Graphics Inc., 1986].

The use of hardware support has important advantages: it is easy to use and maintain, and it is efficient. The drawback is primarily the almost total lack of portability. But, if one defines picking as a mapping from mouse input to a Pascal pointer to an internal object, it is possible to use a hardware implementation in some cases, at the risk of substantial re-write in other cases.

In the second HIBLIZ implementation, the hardware is used. Pascal pointers are converted to integers and stored in the marker by calls inserted into the ordinary drawing routines; when not in pick mode, these calls are no-ops. After picking, the pick buffer is read sequentially and all markers are converted back to pointers.

Picking is likely to remain one of the least portable operations in any graphics implementation.

Integration into HIBLIZ

Because of the long editing turn-around time, the program was reduced to only a minimal set of routines. The stripped version can read a model from an external file, construct the internal data structure, and display the block diagrams; editing of block diagrams and simulations are not possible. The code was stripped with a top-down approach; major parts of the main program were subjected to conditional compilation, and other separately compiled modules were removed by trial-and-error. When only the start-up logo remained, additional routines were reinstalled one-by-one.

The parser that reads an external file and builds the internal data structure was not changed. The original system then built a parallel segment structure during a second pass, which was displayed by a third set of routines. In the new implementation, the graphics is constructed from the original data structure as it is displayed, using much of the code from the second pass; the routines to display the segment structure have been eliminated.

Due to the new syntax and semantics of segment handling, the code generating the graphics could be much simplified, concentrating on drawing and avoiding details of segment handling. A general clean-up of the code was also done, but the logic, for example to draw interfaces, remains the same. It should be noted that the original data structure has not been changed, for better and for worse.

Some parts of the second graphics implementation were never incorporated into HIBLIZ, but tested separately. Examples are rubber-band drawing, picking and multiple overlapping windows. A number of issues have not been fully explored, e.g. how the internal data structure can be simplified (by reducing the number of levels in the tree) to make generation of graphics faster. Because only zooming changes the representation of a module, state variables indicating the current representation and whether zooming has occurred would make panning and scrolling much faster.

4. Other possibilities

The current implementation uses a home-made graphics package, which is built on top of the IRIS Graphics Library [Silicon Graphics Inc., 1986]. Although a standardised graphics package was not used due to cost and possible performance penalties, current work on graphics standardisation must be considered. Much effort has also been devoted to developing new ways of programming graphics, especially in the context of object-oriented programming.

Graphics standards

The reasons for using a standardised graphics package are obvious. The major concern with standard "do-it-all" packages has been efficiency, but as computers get faster and the experience of graphics implementation grows, there will be packages sufficiently fast for most applications.

There is today one generally accepted graphics standard which is available on a number of computers: the Graphical Kernel Standard [Hopgood et al., 1983], [Enderle et al., 1984]. GKS is rather low-level, but all high-level operations in an integrated engineering environment can probably be implemented on top of GKS. It is possible to draw lines with a number of attributes, e.g. color and linetype, independent of computer or terminal type. GKS can handle multiple simultaneous output devices. There are only very primitive raster-operations, and no internal support for window management (in some implementations, each window is regarded as a separate device). Unfortunately, a very inflexible flat segment structure was adopted.

It is not clear how hierarchical graphics should be programmed using GKS. Another question is how picking is mapped back to the internal objects of the application program. Regrettably, many of today's implementations are unacceptably slow; interactive "workstation-type" graphics cannot be based on GKS.

The Programmer's Hierarchical Interactive Graphics System (PHIGS) is an ISO draft standard for computer graphics programming [Shuey et al., 1986], [SIS, 1985]. PHIGS is aiming at applications that manipulate complex displays of 2D or 3D data in an highly interactive environment.

PHIGS is in many ways similar to GKS, but there are important differences: Firstly, PHIGS provides hierarchical segments, called structures. Secondly, once a GKS segment has been created its contents cannot be modified; structures can be incrementally updated at any time. Thirdly, certain graphical attributes are bound at segment creation in GKS, but at display-time in PHIGS. This means that a structure can inherit properties from other structures dynamicly. Fourthly, GKS provides a limited modelling transformation that is applied once to a given set of data. PHIGS allows multiple, cumulative modelling transformations to be applied to objects in master space.

PHIGS seems to be better suited for the kind of application we are concerned with here. The emphasis on interactive graphics is important and hierarchical structures are probably more powerful (even in the real world) than flat segments. GKS is still valid in less demanding applications; the advantages of standardisation and the number of implementations should not be overlooked. However, PHIGS is heavily backed by IBM.

One problem area (which may eventually solve itself) is the lack of experience of using GKS or PHIGS. We do not yet know what "programming-style" best explores the strengths of e.g. GKS.

Object oriented graphics

Object-oriented programming is based on the notion of objects—entities that combines the properties of procedures and data, since they perform computations and save local state. All of the action in object-oriented programming comes from sending messages between objects; rather than calling a procedure to perform an operation on an object, one sends the object a message.

The second major idea in object oriented programming is inheritance. A certain type of objects (called a class) may inherit properties of one or more other classes. Inheritance enables the easy creation of objects with a few incremental changes. A common use is specialization of objects while maintaining common properties [Stefik and Bobrow, 1986].

Object-oriented programming lends itself very well to the way we think about many real-world problems, and indeed to the modelling of control systems. It is also natural to regard the graphical representation of of a system as just another property of an object. Different representations, either in the context of information zooming, or for example the use of block diagrams or animation, are forms of specialization.

The usefulness of object-oriented programming in computer graphics is not well documented, but graphics is often used as an example in introductory texts on object-oriented programming and Lisp machines often have powerful graphics, in particular, window managers. An important research area is to identify operations that can, or cannot, be naturally expressed with object-oriented programming.

5. Summary and conclusions

The graphics of HIBLIZ was implemented in two phases. During the first phase, existing low-level driver routines were filled in. A crude but working implementation was achieved in three man-months. This version did not use the graphics hardware of the IRIS to perform scaling and translation.

During phase two, a completely new set of graphics routines was designed and implemented. Hardware support for transformations, segments and picking was used. Most routines were incorporated into a stripped version of HIBLIZ, while other routines were tested separately.

The purpose of implementing new graphics routines was to gain general experiences of graphics programming, and to investigate common operations in this type of application. Some conclusions can luckily be made.

- Block diagrams are built from simple shapes such as straight lines and rectangles. Qualitative and quantitative color encoding is desirable.
- Hierarchical decomposition of the block diagram is powerful and quite easy to implement. Scaling and translation must be very fast, and the graphics must be clipped at module boundaries.
- Rubber-band drawing can be implemented in many different ways, depending on hardware support and required functions. Only simple operations are portable. The IRIS does not support XOR-drawing, which was a major difficulty.
- Segments significantly increase efficiency, and can be used as a structuring tool. Flat segments are useful, but the user must provide some complementary data structure himself.
- It is often necessary to read auxiliary data and to make decisions while the segments are being displayed. This is normally not well supported by existing packages.

  One basic difficulty of making an efficient implementation is that the graphics hardware must return a result before the application program can continue. The CPU and the graphics processor must synchronize, which leads to less parallelism and decreased throughput.
- One of the most important and most frequent operations in HIBLIZ is to determine the size on the screen of a module. The size controls the representation of the module. Trick-programming with segments gave a neat and relatively efficient implementation.
- Picking of graphical objects is the dominating interaction technique. Instant visual feedback is important to get a good user interface. Hardware support is desirable, but the mapping back to some object in the internal data structure is difficult.
- Emerging graphics standards such as GKS and PHIGS should be used, but are today too slow. PHIGS is the better package for advanced interactive applications.
- Object-oriented programming lends itself well to the way we think about many real-world problems. Inheritance is a very powerful structuring mechanism. An important research area is to identify operations that can, or cannot, be naturally expressed with object-oriented programming.

During the work of implementing the graphics routines, some practical notes could also be made.

- Many routines were simple to implement on-top of the IRIS Graphics Library, which is powerful and relatively easy to use.
- Information zooming requires fonts from 2×2 pixels and up. To achieve acceptable performance, raster fonts must be used as much as possible. Small graphical characters are deformed due to quantization effects.
- Smooth zoom, pan and scroll of complex block diagrams require graphics hardware in the IRIS class.

With this in mind, two developments seem natural. Firstly, to switch from a traditional programming language such as Pascal to an interactive environment. A mature LISP system is probably the best tool for experimental research. Secondly, various forms of object-oriented programming should be explored in the near future.

6. Acknowledgements

I owe my deepest gratitude to Dr. Sven Erik Mattsson, my stalwart brother-in-arms during this project. A fantastic source of inspiration is of course Professor Karl Johan Åström, whose plentiful ideas we all love seeing realized. Professor David Hill was kind enough to lend his Australian aesthetics to the manuscript. Comments by Leif Andersson, Sten Henriksson and Mats Andersson made the report much more readable.

The project "New Forms of Man-Machine Interaction" was supported by the National Swedish Board for Technical Development (STU) under contract number 84–5069.

This report was produced using TeX, a type setting program developed by Donald E. Knuth at Stanford University [Knuth, 1984]. The final result was printed on an Apple LaserWriter.

7. References

ELMQVIST, H. (1985): "LICS—Language for Implementation of Control Systems," Report no. TFRT-3179, Department of Automatic Control, Lund university, Lund, Sweden.

ENDERLE, G., K. KANSY and G. PFAFF (1984): *Computer Graphics Programming (GKS—The Graphics Standard)*, Springer-Verlag.

FOLEY, J. D. and A. VAN DAM (1984): *Fundamentals of Interactive Computer Graphics*, Addison-Wesley.

FOLEY, J. D., V. L. WALLACE and P. CHAN (1984): "The Human Factors of Computer Graphics Interaction Techniques," *IEEE Computer Graphics and Applications*, November 1984, 13–48.

GUDMUNDSSON, B. (1985): *Datorgrafik*, Studentlitteratur, Lund, Sweden.

HOPGOOD, F. R. A., D. A. DUCE, J. R. GALLOP and D. C. SUTCLIFFE (1983): *Introduction to the Graphical Kernel Standard (GKS)*, Academic Press.

KNUTH, D. E. (1979): *TeX and Metafont—New Directions in Type Setting*, American Mathematical Society and Digital Press.

KNUTH, D. E. (1984): *The TeXbook*, Addison-Wesley.

MATTSSON, S. E., H. ELMQVIST and D. M. BRÜCK (1986): "New Forms of Man-Machine Interaction," Report no. TFRT-7327, Department of Automatic Control, Lund university, Lund, Sweden.

NEWMAN, W. M. and R. F. SPROULL (1979): *Principles of Interactive Computer Graphics*, McGraw-Hill.

PETZOLD, L. R. (1982): "A Description of DASSL: A Differential/Algebraic System Solver," *Proc. of IMACS World Congress*, Montreal, Canada.

SHUEY, D., D. BAILEY and T. P. MORRISSEY (1986): "PHIGS: A Standard, Dynamic, Interactive Graphics Interface," *IEEE Computer Graphics and Applications* **6**, No. 8, August 1986, 50–57.

SILICON GRAPHICS INC. (1986): *IRIS User's Guide Version 2.1*, Silicon Graphics, Inc., Mountain View, CA, USA.

SIS (1985): "Datorgrafi—PHIGS, Programmers Hierarchical Interactive Graphics Standard," Technical report no. 306, SIS—Standardiseringskommisionen i Sverige.

STEFIK, M. and D. G. BOBROW (1986): "Object-Oriented Programming: Themes and Variations," *The AI Magazine* **6**, No. 4, Winter 1986.

Appendix. Definition of graphics operations

This appendix describes the operations available in the second implementation of graphics for HIBLIZ.

Coordinate systems, and operations on these

```
MasterCoord = real;
WorldCoord  = MasterCoord;
NormCoord   = real;                  { 0.0 .. 1.0 }
RasterCoord = real;                  { -0.5 .. 1023.5 }
PixelCoord  = integer;               { 0 .. 1023 }
Transformation = array [0..3, 0..3] of real;
```

```
procedure  GetTransform(var t : Transformation);
```

Returns current transformation in 't'. Used with SetTransform.

```
procedure  SetTransform(t : Transformation);
```

Sets current transformation to 't'. Only well-defined if 't' has been assigned using GetTransform.

```
procedure  PushTransform;
```

Pushes a copy of the current transformation onto the internal transformation stack.

```
procedure  PopTransform;
```

Pops the top element of the internal transformation stack into the current transformation. The previous current transformation is lost.

```
procedure  MasterToPixel(x, y : MasterCoord;
                         var px, py : PixelCoord);
```

Transforms the master coordinates (x, y) to pixel coordinates (px, py), using the current transformation.

```
procedure  PixelToMaster(px, py : PixelCoord;
                         var x, y : MasterCoord);
```

Transforms the pixel coordinates (px, py) to master coordinates (x, y), using the current transformation.

```
procedure  PixelSize(var x, y : MasterCoord);
```

Returns horizontal and vertical size of a pixel in master coordinates, using the current transformation.

```
procedure  NormToPixel(x, y : NormCoord;
                       var px, py : PixelCoord);
```

Transforms normalized screen coordinates (x, y) to pixel coordinates (px, py).

```
procedure  PixelToNorm(px, py : PixelCoord;
                          var x, y : NormCoord);
```

Transforms pixel coordinates (px, py) to normalized screen coordinates (x, y).

```
    procedure  TransTransform(tx, ty : MasterCoord);
```

Places the master coordinate system origin at a given point (tx, ty) in the current coordinate system, which is either world space or a superior master space.

```
    procedure  ScaleTransform(sx, sy : real);
```

Shrinks, expands or mirrors the master coordinate system. Values with magnitude greater than 1 expand the object; values less than 1 shrink it. Negative values mirrors it.

Routines for graphical output

```
    MaxColors = 15;
    ColorCode = 1..MaxColors;
```

```
    procedure  MoveTo(x, y : MasterCoord);
```

Sets current position to (x, y) without drawing a line.

```
    procedure  LineTo(x, y : MasterCoord);
```

Draws a line from current position to (x, y), which then becomes the new current position.

```
    procedure  Rectangle(Xmin, Ymin, Xmax, Ymax : MasterCoord);
```

Draws a rectangle with the specified lower-left and upper-right corners. Current position is not affected.

```
    procedure  FilledRectangle(Xmin, Ymin, Xmax, Ymax
                                        : MasterCoord);
```

Same as Rectangle, but the area will be filled using the current color. Current position is not affected.

```
    procedure  GetPosition(var x, y : MasterCoord);
```

Returns current position as set by MoveTo, LineTo or the text drawing primitives.

```
    procedure  RubberMode(on : boolean);
```

If 'on' is true, rubber-band mode is turned on. The rubber-band line will originate at the current position when calling RubberMode. Hereafter, the only well-defined routines are RubberLine, RubberRectangle and another RubberMode. If 'on' is false, rubber-band mode is turned off, and any drawing command may be used. The last rubber-band (line or rectangle) is erased.

```
    procedure  RubberLine(x1, y1, x2, y2 : MasterCoord);
```

Draws a rubber-band line from (x1, y1) to (x2, y2). The old rubber-band line is erased.

```
procedure  RubberRectangle(x1, y1, x2, y2 : MasterCoord);
```

Draws a rubber-band rectangle from (x1, y1) to (x2, y2). The old rubber-band rectangle is erased.

Attribute setting routines

```
procedure  DefineRGBcolor(col : ColorCode; r, g, b : real);
```

Defines a color in the color map. The amount of red, green and blue components is in the range [0, 1]. Values less than 0 is regarded as 0; values larger than 1 are regarded as 1.

```
procedure  SetColor(col : ColorCode);
```

Sets the current color to 'col'. Subsequent lines, rectangles and text strings will be drawn using this color.

```
procedure  SetLineWidth(width : integer);
```

Sets line width (in pixels) of subsequent lines and rectangles. Minimum line width is 1 pixel. The maximum width is implementation dependent.

```
procedure  SetLineStyle(style : StyleIndex);
```

Sets line style for subsequent lines. Currently, only solid lines are supported, so SetLineStyle is a no-op.

Miscellaneous screen handling

```
procedure  SwapBuffers;
```

Swaps front and back buffers. SwapBuffers does not erase the buffer contents, so the back buffer must be erased with ClearBuffer afterwards.

```
procedure  ClearBuffer;
```

Erases the contents of the back buffer. Not restricted by clipping boundaries.

```
procedure  SetHighlight(on : boolean);
```

If 'on' is true, everything drawn on the screen is highlighted, i. e. drawn using some alternate color.

```
procedure  GetScreenSize(var x, y : PixelCoord);
```

Returns size of screen in pixels. The upper-right corner of the screen has pixel coordinate (x–1, y–1); the lower-left corner is (0, 0).

```
procedure  InitGraphics;
```

Initializes graphics routines. Must be called once.

```
    procedure  ExitGraphics;
```
Clears the screen and does required clean-up.

Cursor and mouse i/o, logging

```
    procedure  GetCursor(var x, y : PixelCoord);
```
Returns current position of cursor (and mouse) in pixel coordinates.

```
    procedure  GetNormCursor(var x, y : real);
```
Returns normalized current position of cursor (and mouse). Range [0, 1].

```
    procedure  SetCursor(x, y : PixelCoord);
```
Sets cursor (and mouse) position in pixel coordinates.

```
    procedure  GetMouse(var left, middle, right : boolean);
```
Returns current mouse buttons. Returns true if button is pressed.

```
    function  GetButton(but : integer) : boolean;
```
Returns true if specified button is down. Buttons are numbered from 1 (leftmost) to 3 (rightmost).

```
    procedure  SetLogging(on : boolean);
```
If 'on' is true, logging is turned on. The existing log file (always called TRACE.LOG) may be deleted. If logging is turned on more than once during a single invocation of the program, output is appended to the existing log file.

Menu handling routines

```
    procedure BeginMenu(MenNo : integer; Backgrnd, Foregrnd,
                  Highlight : ColorCode; width : integer);
```
Initialize a new menu with the specified background and foreground colors. Width is the maximum number of characters in the longest menu entry. When the menu is displayed with GetMenu, selected item entry is Highlight.

```
    procedure MenuEntry(name : StringType);
```
Defines a new menu entry in the current menu. The entry will be presented by the string 'name' using the foreground color from BeginMenu.

```
    function GetMenu(MenNo : integer) : integer;
```
Show menu on screen at current cursor position. Returns an index in the range [0..max] indicating which entry was selected. Entries are numbered in the order they were defined by MenyEntry above. A selection outside the menu returns −1.

Picking routines

```
PickMarker = longint;
```

```
procedure  BeginPicking;
```

Turns on pick mode and clears the hit-list. The pick marker is emptied, i. e. nothing is added to the hit-list until SetPickMarker has been called again.

```
procedure  EndPicking(var count : integer);
```

Turns off pick mode and returns the number of picked markers.

```
procedure  SetPickMarker(item : PickMarker);
```

Sets the pick marker, which will be copied to the hit-list if a pick-hit occurs.

```
function  GetPickHit(i : integer) : PickMarker;
```

Returns the i'th pick hit in the hit-list. Hits are counted starting from 1; indexes smaller than 1 or larger than the number of hits return zero.

Segments and segment mode

```
SegmentType = Object;          { Very IRIS dependent. }
SegmentId = ^SegmentType;

NullSegment : SegmentId;
```

```
function  CreateSegment(var seg : SegmentId) : boolean;
```

If 'seg' is an existing segment, the segment is displayed and CreateSegment returns false. If not so, CreateSegment enters segment mode and returns true.

```
procedure  CloseSegment;
```

Exits segment mode and displays the segment that was open.

```
procedure  DeleteSegment(var seg : SegmentId);
```

Deletes the segment 'seg', which can be re-created by CreateSegment.

```
function  VisibleBox(Xsize, Ysize : PixelCoord;
              Xmin, Ymin, Xmax, Ymax : MasterCoord) : boolean;
```

Returns true if the box from (Xmin, Ymin) to (Xmax, Ymax) is partly or completely inside the current window, and is larger than Xsize times Ysize pixels.

Text primitives and interaction window

```
StrType = string[255];
```

```
procedure  DrawString(s : StrType);
```

Draws a string at current position with the attributes set by SetColor and SetTextHeight. The current position is updated by the size of the string, so the next time DrawString is called, the new string is drawn behind the old.

```
    procedure  SetTextHeight(h : MasterCoord);
```

Specifies size of text drawn by DrawString. The text will be drawn with the characters closest in height to 'h', but may be a little smaller. The result may depend on the number of available raster character sizes.

```
    function  GetTextLength(s : StrType) : MasterCoord;
```

Returns the length of the text string, using current size and transformation matrix.

```
    procedure  GetTextHeight(var h : MasterCoord);
```

Inquires actual height of text drawn by DrawString.

```
    procedure  WriteString(s : StrType);
```

Outputs a string to the interaction window.

```
    procedure  WriteInt(i : integer);
```

Outputs an integer number to the interaction window.

```
    procedure  WriteReal(x : real);
```

Outputs a real (floating point) number to the interaction window.

```
    procedure  BreakLine;
```

Starts a new line in the interaction window.

```
    procedure  ReadString(var s : StrType);
```

Reads a string from the keyboard.

```
    procedure  ReadInt(var i : integer);
```

Reads an integer number from the keyboard.

```
    procedure  ReadReal(var x : real);
```

Reads a real (floating point) number from the keyboard.

Window management and clipping

```
    WindowType = record
                   vp  : record      { Viewport }
                           Xmin, Ymin : PixelCoord;
                           Xmax, Ymax : PixelCoord;
                         end;
                   lim : record      { World space limits }
                           Xmin, Ymin : WorldCoord;
```

```
                             Xmax, Ymax : WorldCoord;
                           end;
                   end;
    WindowId     = ^WindowType;
```

```
    procedure  CreateWindow(var wind : WindowId;
                      Xmin, Ymin, Xmax, Ymax : PixelCoord);
```

Creates a new window that will use a viewport from (Xmin, Ymin) to (Xmax, Ymax). The default window limits in the user's world space, which are normally defined by SetWindow below, will be (0, 0) and (1, 1). Lastly, the newly created window becomes the active window.

```
    procedure  SetWindow(wind : WindowId;
                      Xmin, Ymin, Xmax, Ymax : WorldCoord);
```

Sets the window limits. A rectangular area of world space from (Xmin, Ymin) to (Xmax, Ymax) will fill the viewport. Again, 'wind' becomes the active window.

```
    procedure  SelectWindow(wind : WindowId);
```

Makes 'wind' the active window. Graphics will be drawn in the viewport specified with CreateWindow, using window limits specified with SetWindow.

```
    procedure  ClearWindow;
```

Erases the contents of the viewport of the active window; this area of the buffer is filled with some default background color. C. f. ClearBuffer.

```
    procedure SetClipArea(Xmin, Ymin, Xmax, Ymax : MasterCoord);
```

Sets clipping area. Subsequent drawing primitives will only draw inside the clipping area.