CODEN:LUTFD2/(TFRT-5382)/1-042/(1988)

Object-oriented Graphics for the Future Instrument Panel

Anders Nilsson

Department of Automatic Control
Lund Institute of Technology
September 1988

Department of Automatic Control
Lund Institute of Technology
P.O. Box 118
S-221 00 Lund Sweden

Document name
MASTER THESIS REPORT

Date of issue
September 1988

Document Number
CODEN:LUTFD2/(TFRT-5382)/1-042/(1988)

Author(s)
Anders Nilsson

Supervisor
Sven Erik Mattsson, Dag Brück

Sponsoring organisation

Title and subtitle
Object-oriented Graphics for the Future Instrument Panel

Abstract

This report describes the design and implementation of a simulated Future Instrument Panel (FIP). The purpose of FIP is to provide a more natural operator environment by applying new hardware and new programming methodologies.

This implementation uses a COMPAQ 386 personal computer with an additional high-performance graphics card. Object-oriented programming is used in building a library of graphical objects that make up the instrument panel. The programming language is Objective-C; the Computer Graphics Interface (CGI) is used for graphics.

Object-oriented programming has proved to be well suited for implementation of operator communication.

Key words

Classification system and/or index terms (if any)

Supplementary bibliographical information

ISSN and key title

ISBN

Language
English

Number of pages
42

Recipient's notes

Security classification

Table of Contents

Table of Objective-C classes

1. Introduction

This report describes a master thesis project conducted as a part of the FIP-project at Asea Brown Boveri (FIP stands for Future Instrument Panel). The purpose of the FIP-project is to test flat panel displays, new graphics hardware, an object-oriented language, and to build a library containing graphics objects used to represent distributed control systems on a flat panel display. The parts that the report will treat are the graphics library and some conclusions about the hardware and the object-oriented language used.

During the past several years the development of graphics applications has been fast. Computers become more powerful and calculation times decrease. This fact makes it possible for demanding graphics applications to progress. The theories used today in the graphical domain have often been well-known for a long time. An example of this is the concept of homogeneous coordinates which is used in all graphics applications when scaling, rotating and translate visible objects on the screen. This theory was well-known as early as in the 1920'ies. Homogeneous coordinates are described in Appendix A.

This chapter will give a background to the FIP-project and the master thesis project described in this report. The report will treat the hardware, the object-oriented language and the graphics used during the project. In Chapter 3 we will explain the object-oriented programming technique and how CGI (Computer Graphics Interface) works. Chapter 4 gives a description of the classes developed in the project. This description is intend to make it easier for programmers that will continue the FIP-project. Finally Chapter 5 will summarize all those experiences gained during this project, and some possible improvements will be suggested.

1.1 The FIP-project

The goal of the FIP-project is to offer a more human and intelligent way of man machine communication compared to current systems. In the future this may be used in distributed process control, to make it feasible for the operator to communicate in a more natural way with the process. The goals of the FIP-project are to:

(i) Use an object-oriented language in a graphics application

(ii) Test new fast graphics hardware

(iii) Build a presentation-object library

(iv) Test flat panel displays

The master thesis project covers items (i) and (iii).

1.2 Background

To test the new concepts a sawmill simulation was used. Following features were desired in the sawmill simulation:

- Possible to set the speed reference of the motor. Different types of dialogue primitives would support this, for example, the calculator, the slide-potentiometer or the pushbutton.
- The speed should be visible on the screen. The circular-pointer instrument would provide this quality.
- Communication between dialogue primitives and presentation primitives should be provided by pipes; the pipe concept is described in Chapter 4.7.

Figure 1.1 and Figure 1.2 give an apprehension of the concepts above. The classes in Figure 1.1 are all described in Chapter 4. Figure 1.2 gives an explanation of the symbols used in Figure 1.1.

Figure 1.1 shows the communication lines between the different classes in the sawmill simulation, more correctly the instances of the classes. In this description pushbuttons are used to set the speed reference. The workstation manager in the figure is implemented in the main program. Also the PTC Class methods were implemented in the main program. PTC stands for Presentation Type Circuit. The definition of a PTC is a composition of dialogue primitives, presentation primitives and pipes. The pipes provide the connections between the dialogue primitives and the presentation primitives and other FIP objects like the database. The pushbutton is an example of a dialogue primitive (described in Section 4.4). To the left in the figure is the circular-pointer instrument class which is an example of a presentation primitive (described in Section 4.5).

Figure 1.3 shows a picture of a flat panel display with a possible configuration. Notice that the picture of the screen is in natural size. The scroll bar menu on the top of the screen is an example of a dialogue primitive. The dotted bar to the left in the picture show a pull-down menu. The choice decides what menu you will get. The padlock in the lower left corner tells if it's possible to change the process parameters. A shut padlock means that you must have a special code to open the padlock, and by that get access to change the process parameters. When trying to open the padlock there will appear a calculator on the screen. If you know the code you just click at the buttons of the calculator using the mouse. The padlock will open if it was the right code. The padlock is a typical dialogue primitive. In the lower right bar you can see three objects. The start/stop button and the reset button are also examples of dialogue primitives. In the middle there is a display showing the time, this is a presentation primitive. The circular-pointer instrument and the %-display are together describing a Presentation Type Circuit (PTC). The blank area normally shows a dynamic process picture.

SAWMILL STRUCTURE

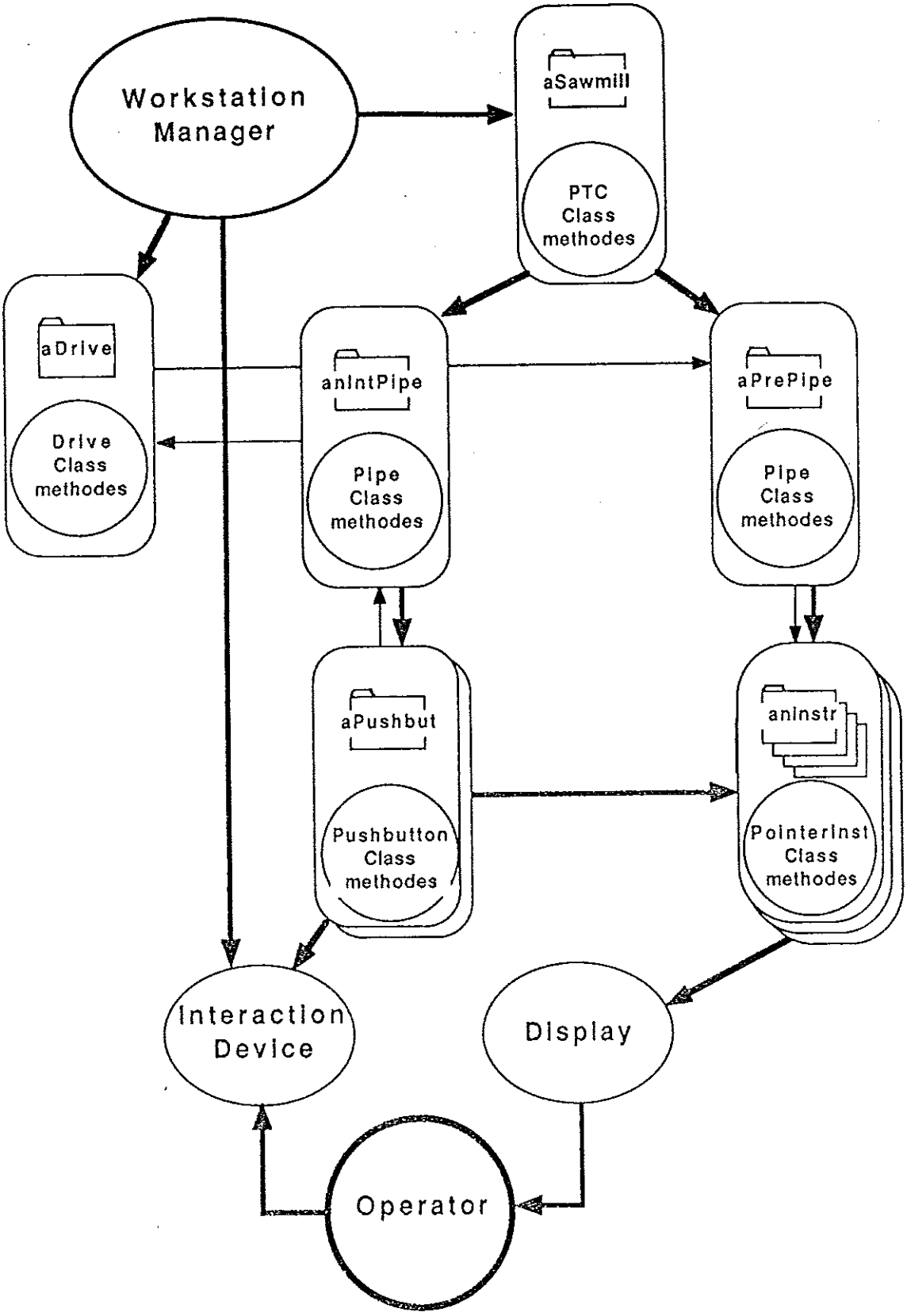

Figure 1.1 The structure of a sawmill process, simulated in the project.

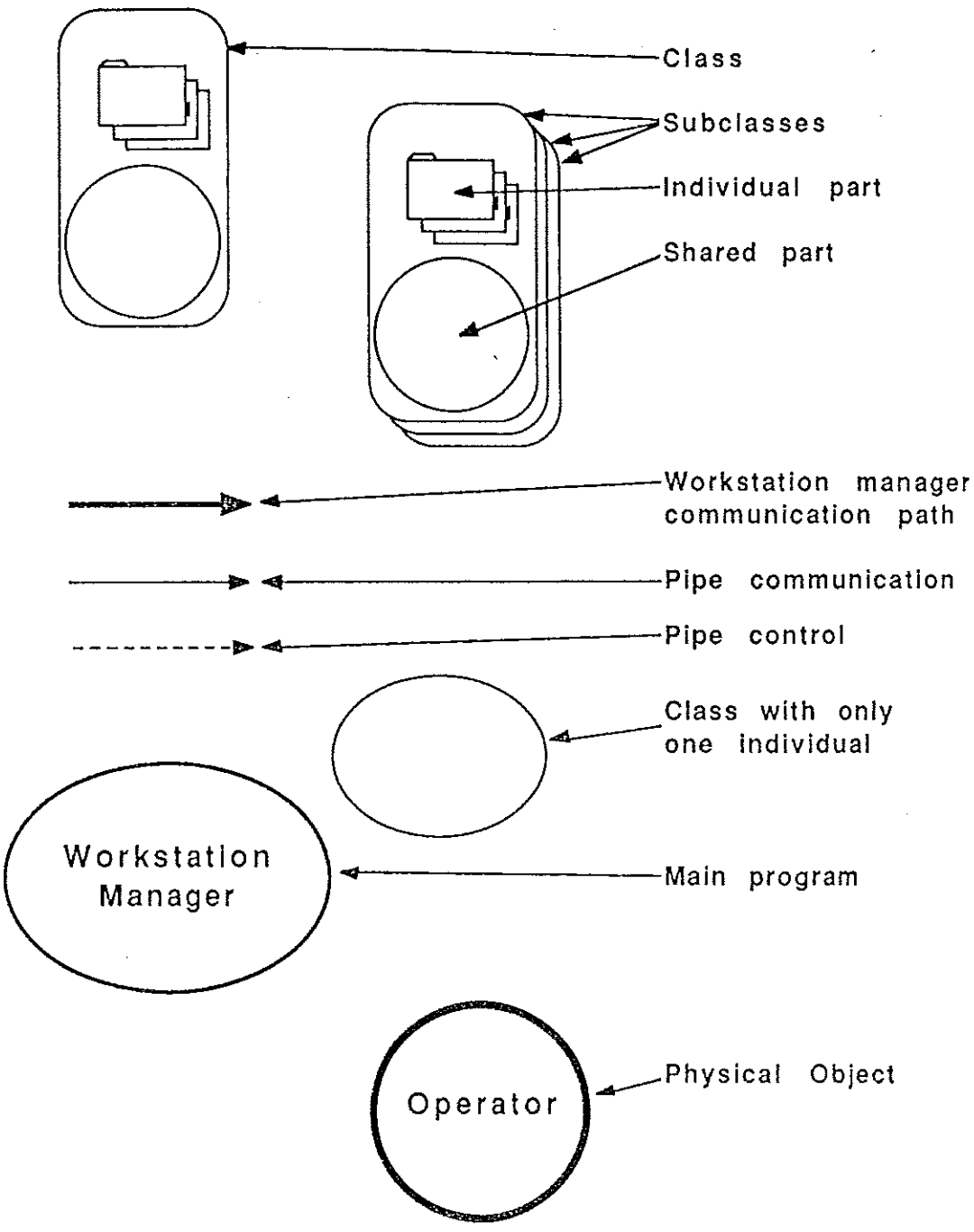

Figure 1.2 Explanations of the symbols in Figure 1.1.

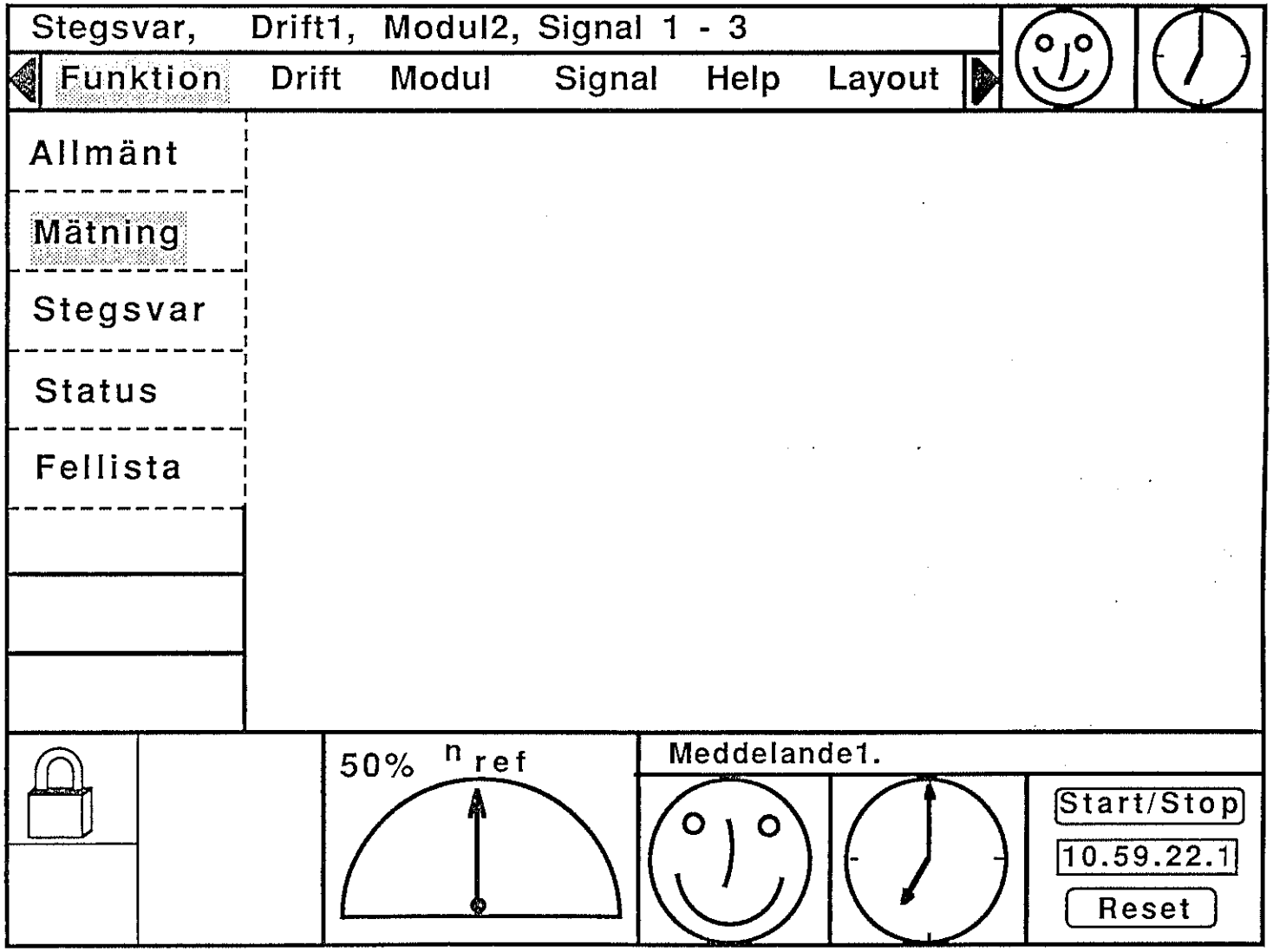

Figure 1.3 A typical configuration of the screen in natural size.

2. Project description

2.1 Development conditions

In this project an object-oriented language was used. The advantages of a program written in an object-oriented language is the natural modularity of the program, which makes it possible to replace and make changes in parts of the program without compiling all code.

It was desirable to use a 640 × 480 pixels monochrome flat panel display type electroluminescent (ELD) or LCD. Neither of these were available when the project started, so the flat panel display was simulated by using a normal CRT screen. The typical ELD yellow-brown color was used. The wish to use flat panel displays was because of their robustness and slim performance in comparison with conventional screens. It was intend to communicate with the system by touching the screen, because of that it was necessary to use a touch-screen; the touch-screen was simulated by using a mouse.

For pure graphics applications Graphics Software Systems CGI (Computer Graphics Interface) version 2.0 was used. CGI is a standard interface between device-dependent and device-independent code in a graphics environment. It makes all device drivers appear identical to the calling program.

The program was developed under MS-DOS version 3.31 using a PC. The PC was a COMPAQ 386/20 with a numeric coprocessor, and a NEC color monitor (640 × 480 pixels).

A nice graphics card called PG-641 from Matrox Canada was also used in the project. This card has a graphics processor called 34010 from Texas instruments. Almost every graphics function of this card is directly realized in hardware. For example, when drawing a circle, the card only get the information about the center-point and the radius, the rest is directly implemented in hardware. Other functions are: drawing bars by giving the coordinates of the lower left corner and the upper right corner, filling polygons with predefined patterns by giving the polygons coordinates, drawing lines and much more. The hardware implementation increases graphics performance considerably.

2.2 The goals of the master thesis project

The goals of the master thesis project were to get an apprehension about using these tools mentioned in Section 2.1 together in a graphics application. Creating an object-library containing dialogue primitives and presentation primitives. The presentation primitives would be used to present values and symbols on the screen. The dialogue primitives which would be created were the following, a calculator, a slide-potentiometer and a padlock. The dialogue primitives would all be built by using encapsulated presentation primitives (encapsulation is explained in Section 3.1). They should all be possible to translate and scale continuously, and some of them should be rotatable too. The master thesis project began with a start-project which was to create a circular-pointer instrument, this is the most important presentation primitive

in the subproject. These small building blocks would be used to simulate a sawmill-process on the screen. The communication between the operator and the process was a major part to be implemented as well. Specifically it was very interesting to see if object-oriented programming was the right tool to use in this domain. The choice of Objective-C was not very important, it was the object-oriented programming as a concept that was interesting.

Today, the most common way to translate pictures on the screen is by showing a dotted contour-picture instead of the real picture showed when the object is stationary. One of our goals was to test for the possibilities of translate objects on the screen in a continuous way showing the real picture of the object. This might be possible with a powerful hardware described in the previous section.

2.3 Realization

As mentioned before the programming language Objective-C was used in this project. Objective-C is described in Chapter 3.2. The first task was to learn object-oriented programming in general and objective-C in particular. A course called 'Thirty minutes tutorial' in the objective-C manual was a good help to learn objective-C. Before the real project we began with a smaller start-project which would result in a circular-pointer-instrument on the screen. Figure 2.1 will give an apprehension of the structure of the project. In Figure 2.1 you can see the communication lines between the CGI and the hardware. Everything above the dotted line are written in Objective-C. The ellipse containing Graphics Subroutines is an interface between the graphics routines written in Objective-C and those written in C. This part is not implemented in the master thesis project. The interface is intend to make the FIP-system more portable and independent from the system below the dotted line in the figure.

When the main project started an inheritance-structure for the classes was made. This is thoroughly treated in the Chapter 4. To make the visible objects scalable, translatable and rotatable it was necessary to build a library containing those functions. Homogeneous coordinates were used in the implementation. This concept is described in appendix A.

To form an opinion of how the objects would behave on the screen demonstration programs were used during the project. In the final demonstration program all objects which had been developed during the FIP-project were used. As a process model a program that simulate a sawmill machine was used, in which the motor was PID-controlled (see Figure 1.1).

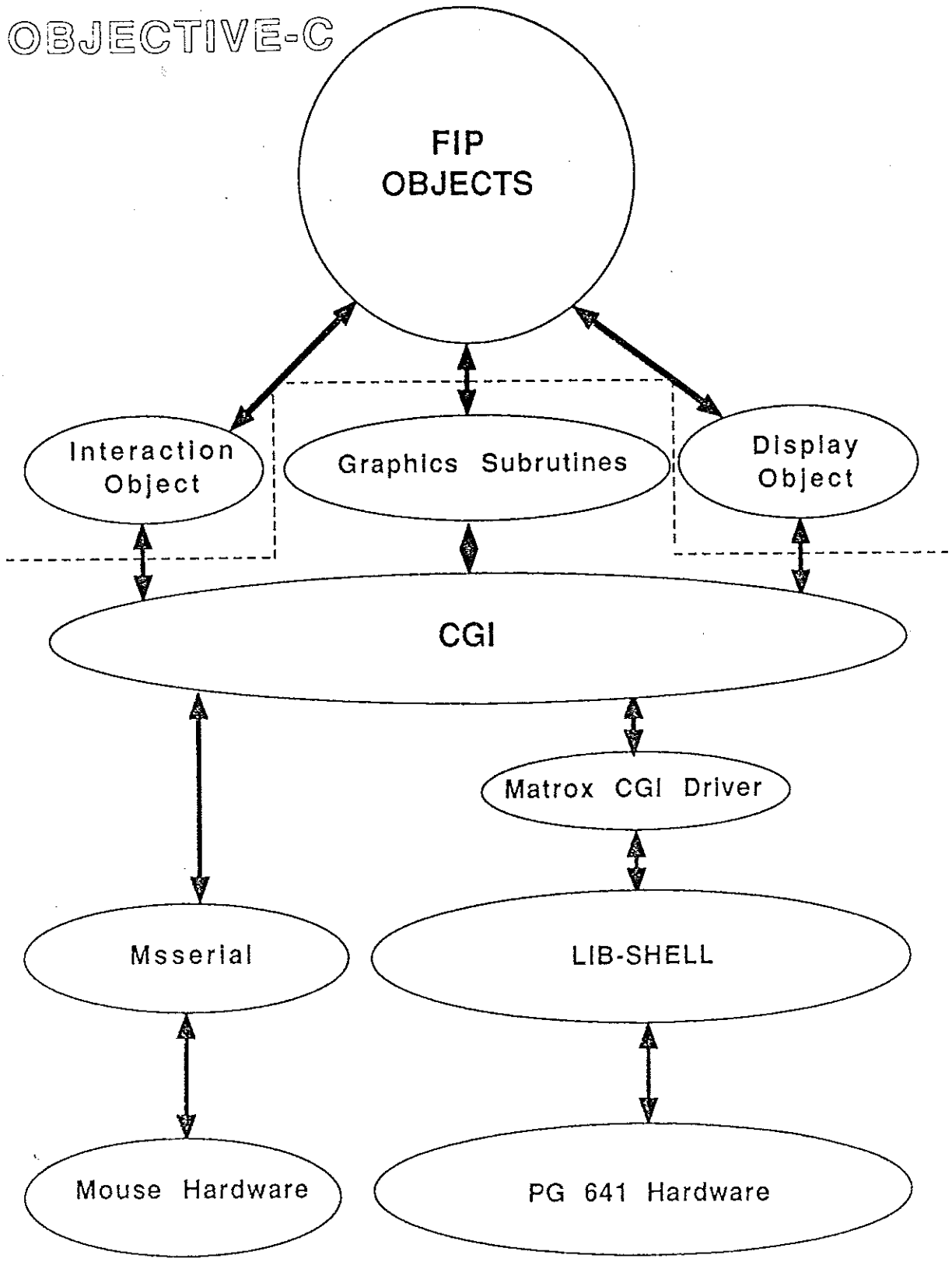

Figure 2.1 An overview of the including parts in the FIP-project.

3. Programming concepts

This chapter will give a brief description of objective-oriented programming, particularly in Objective-C, and the Computer Graphics Interface (CGI). The purpose is not to explain details of these programming tools, but to review major concepts and terminology, so that those not familiar with them can understand the following chapters. The concepts that are common to all object-oriented languages are described from the Objective-C's point of view mainly. This is done to avoid misunderstandings when meaning concepts differ between the object-oriented languages.

3.1 Concepts of Objective-C

Objects

Objects are a combination of data and procedures. Unlike conventional programming systems, which require the programmer to manage the interaction between data and procedures, object-oriented programming systems combine data and the procedures that operate on that data. The joint entity is called an object. To invoke an objects procedure, you send the object a message. In Objective-C an invocation of a method can look as below:

```
[instancename methodname: parameter].
```

Classes

In terms of general programming, classes is a good example of an abstract data-type. It is not possible to change the data of the class except using the included methods. Objects which respond in the same way to the same messages are said to be of the same class. Each object is called an instance of its class. A class is defined by describing the names and types of the data the instances should keep, as well as listing the messages to which the objects of the class should respond. The basic idea is that the class describes the structure of the data and how the messages should operate on that data, while the objects themselves contain the variable data.

Methods

All classes define functions that know how to operate on the object's data. These functions are called methods. Methods are invoked when sending a message to an object. It is the combination of data and methods that together is the definition of a class.

Inheritance

An important concept in object-oriented programming is inheritance. By letting a class inherit another class, it inherits all variables and methods defined in the ancestor class. In this way, you do not have to explicitly describe the message behaviour of each class. Classes have all the message behaviour and variable data as some other, inherited, class. This means that all classes are arranged into a tree called the inheritance hierarchy.

```
id keyboard;        /* variables in the calculator class */
id window;

keyboard = [Keyboard new];        /* method calls */
window = [Window new];
```

Listing 3.1 Example of encapsulation in Objective-C.

Encapsulation

The term encapsulation is called 'composite classes' in some other object-oriented languages. Encapsulation means to use other classes in a new class-definition. Encapsulation was quite frequently used in this project. The dialogue primitives use presentation primitives to build themselves. For example, the calculator encapsulates the window class and the keyboard class (Listing 3.1 shows a bit code of the calculator class). The keyboard class encapsulates the pushkey class. This is a very important tool in graphics applications because you very often want to use predefined building blocks your own classes.

Ordered collection

This is one of the predefined classes in the basic library of Objective-C. Instances of this class can store a collection of other objects. There is no restriction concerning what number of objects the instances can hold (except the memory capacity of course). This predefined class was used in the main program to keep track of all visible objects and the non-visible objects as pipe and drive.

3.2 The Computer Graphics Interface

The Computer Graphics Interface (CGI) is a standard interface between device dependent and device independent code in a graphics environment. It provides an interface that allows a computer to control several graphics devices simultaneously without regard for their individual characteristics.

The Virtual Device Coordinate system

The VDC system allows graphics information to be specified for all devices in an identical way, for example, the mouse or the display. The VDC area is an abstract space in which graphical functions define virtual images. With 16-bit integer coordinates, the VDC range is the set of integers in the range -32768 to 32767, in other words -32K to 32K. The visible area is machine dependent. There are four different screen modes to choose between.

0 Full screen: maps 0 through 32K in x and y to the full extent of the display surface.

1 Preserve aspect ratio: maps 0 through 32K to the full extent of the geometrically longer display surface only, it maps a sub set of VDC space to the shorter axis.

2 Device units: all coordinates are specified in physical device-dependent units.

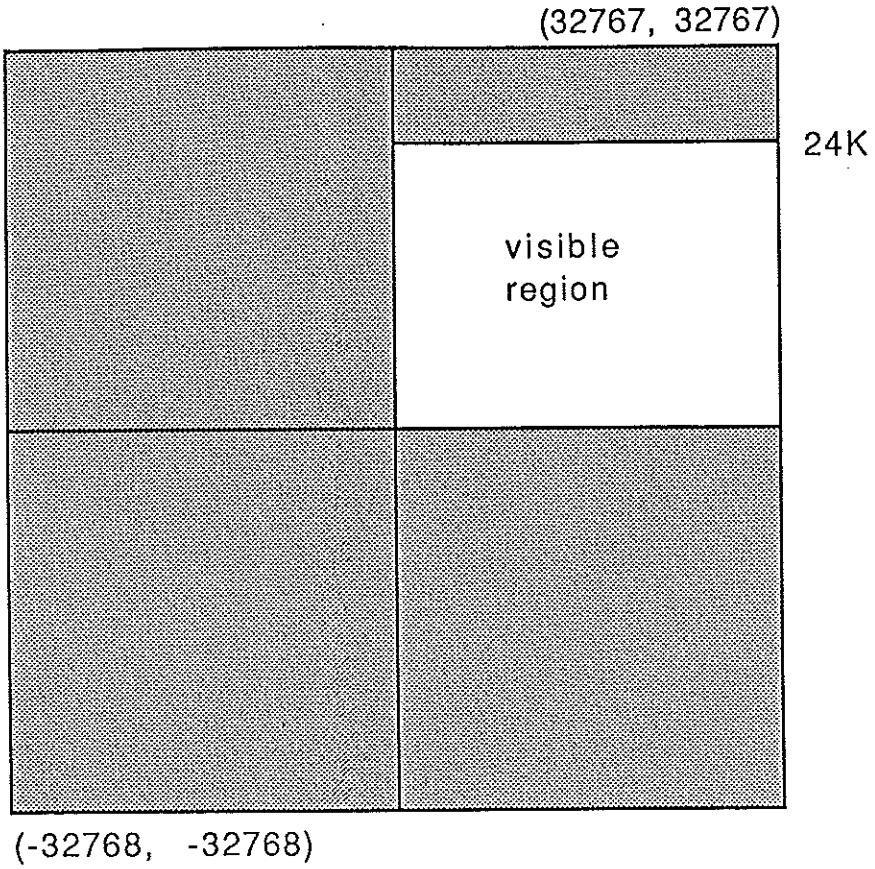

Figure 3.1 The visible region of the VDC-area when choosing mode 1.

3 Short axis: it maps 0 through 32K to a subset of the longer display surface axis in such a way as to preserve a unity aspect ratio.

In this project mode 1 was used, which result in a visible region of 0 to 32K in the horizontal plane, and about 0 to 24K in the vertical plane. See Figure 3.1.

Graphics functions

The CGI provides for the output of graphics primitives (for example: polylines, polymarkers, text, bars, circles, ellipses and arcs) and control over primitive attributes (for example: color, fill, line style). It is also capable to rotate text, it has different text fonts and have many different patterns which can be used to fill bars, polygons and circles. One of the most important functions that CGI provides is the bitmapping functions which make it possible to draw pictures in an off-screen bitmap. A bitmap can be copied to the screen in any position. When updating visible objects very frequently this is a necessity to get a flicker-free screen. In this project the screen is updated every 100 ms.

The workstation

The model defines a workstation as zero or one output devices and zero or more input devices such as a keyboard or a mouse. The CGI uses identifiers to refer to a single generic graphics device such as a display or a mouse. See [GSS, 1986a] page 3–122 and page 3–128 for more detailed information.

4. Design

In this chapter we give a brief description of the most important data structures in the code, and we describe the use of methods and their behaviour. Names of methods which begin with a '+' are factory methods, which means that they are only reachable when invoking a class, they cannot be reached by an instance call. The opposite are the instance methods which begin their names with a '−'. They are only reachable when invoking an instance of a class. Returning the receiver means that the invoked method returns its own identity. The datatype short means a 16 bits integer, the data-type double means a float point number with double precision. In this brief description we cannot give a complete description of the behaviour and appearance of the visible objects.

4.1 Overview of the inheritance structure

Inheritance is one of the basic ideas of object-oriented programming. Figures 4.1, 4.2 and 4.3 show the inheritance structure of the different objects. Notice that the library classes that Objective-C provides are not included in the figures. The sons in the inheritance structure have their local data stored in a local structured data-type. The name of the structured data-type makes it easy to understand at what level the data are inherited. Dialogue primitives are those objects that provide two-way communication, and the presentation primitives are those objects that present data on the screen.

4.2 'Full copy objects' and 'non full copy objects'

Considering the approach to presenting visible objects on the screen there are two different approaches used in this project. This causes two different types of visible objects. The reason to introduce a second type of object was that some visible objects are updated quite frequently. When updating the values frequently, the accuracy soon will deteriorate because of rounding error. After every graphic transformation there will occur a rounding error that adds to the current error. Some objects are updated every 100 ms, so it is not difficult to understand that the error grows. To avoid this to happen the 'full copy objects' were introduced. They copy their parameters into a copy before any graphics operation such as scaling translation or rotating. The copy is then used in the graphics actions. As a consequence of this, 'full copy objects' always begin their graphics actions from the state they once were defined. These classes have an extra method named copy. The 'non full copy objects' store the parameters describing the current state, they cannot remember the initial parameters.

Let us explain with an example: object A is a 'full copy object', and object B is a 'non full copy object'. Both of them have the following default values: size = 1.0, position in VDC coordinates is (x=1000, y=1000). When showing the objects on the screen they will have the size 0.2 and the position will vary between (500, 500) and (200, 100). The first action will be to shrink

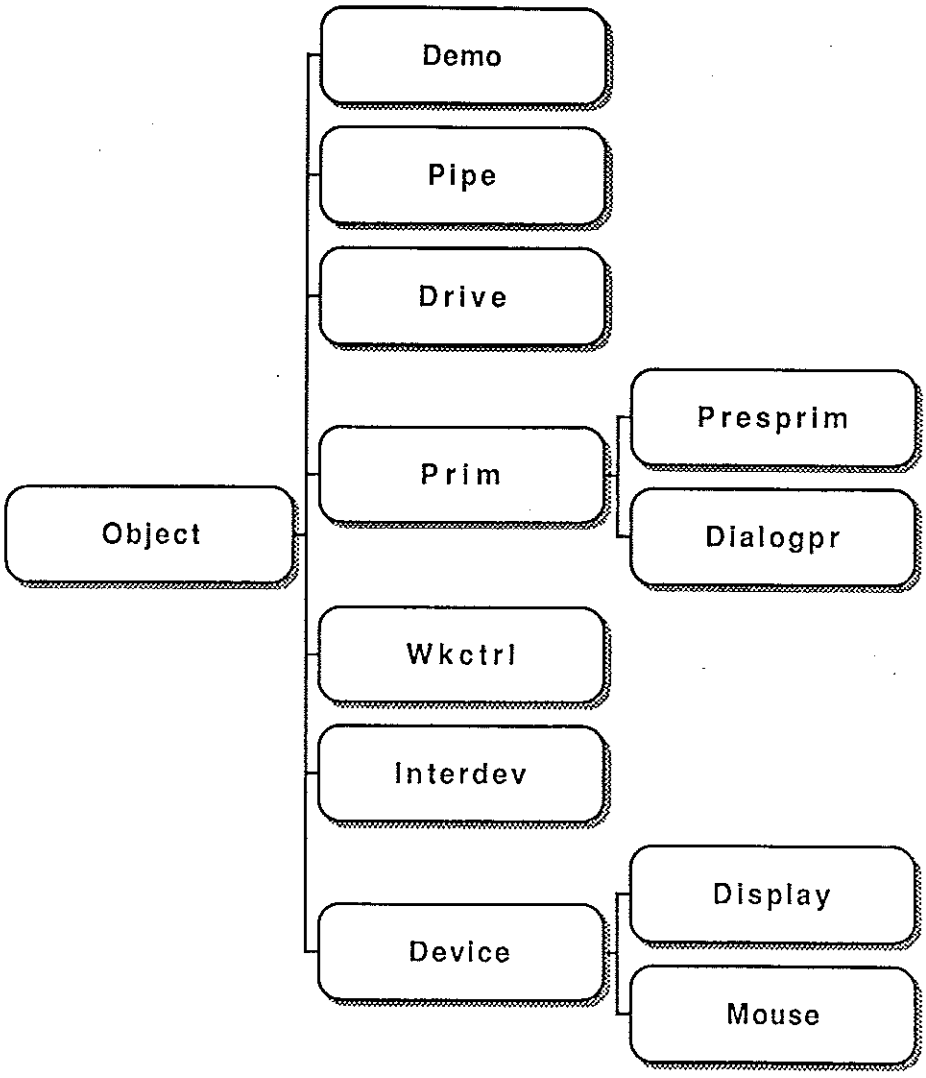

Figure 4.1 The top of the inheritance hierarchy.

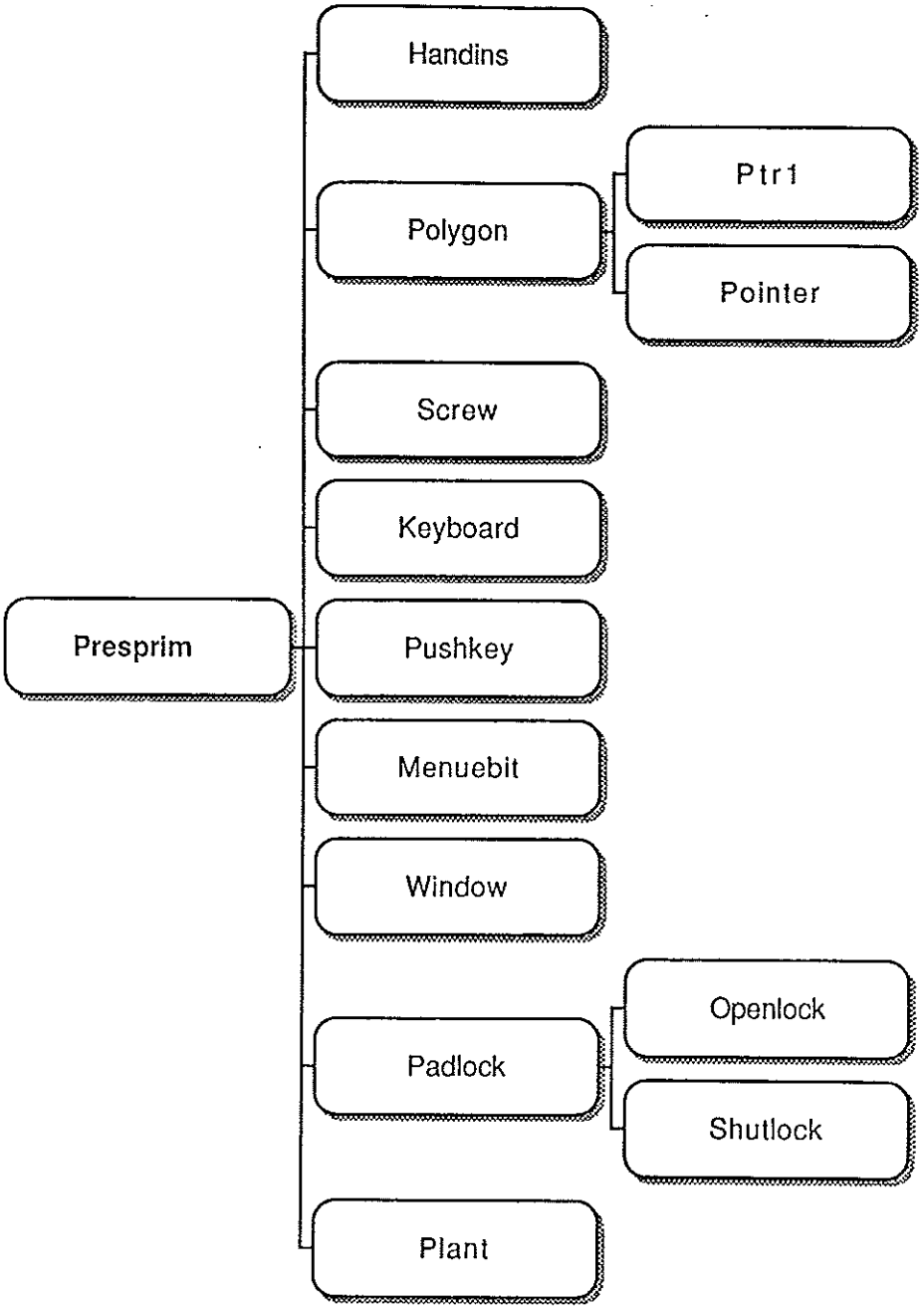

Figure 4.2 The inheritance tree of the presentation primitives.

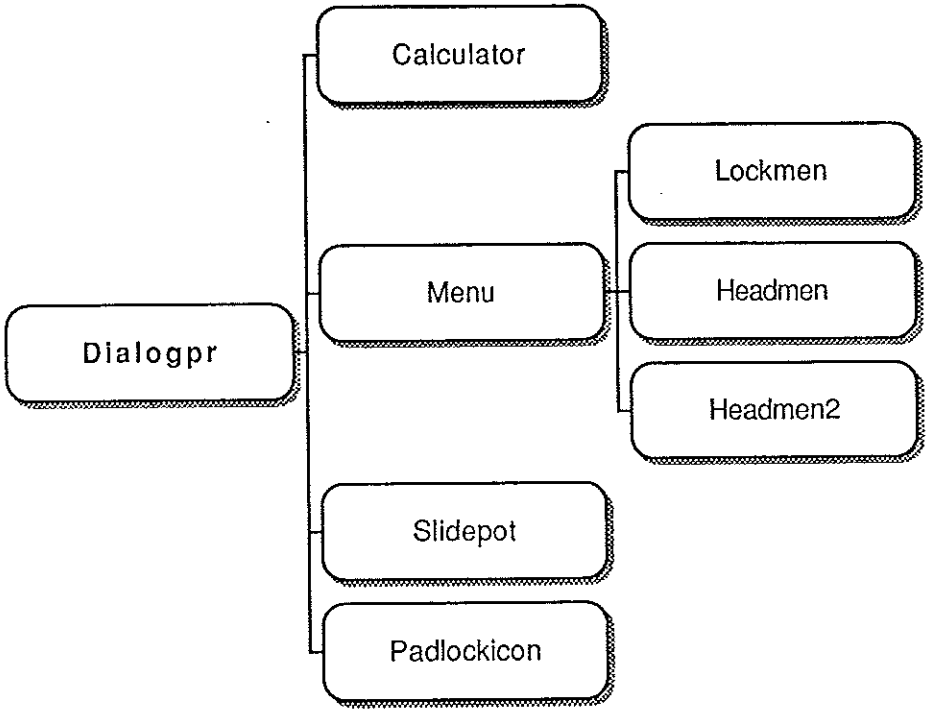

Figure 4.3 The inheritance tree of the dialogue primitives.

down to the size 0.2. Then move to position (200, 100) and then to position (500, 500). The following actions will just be translations between the two mentioned positions in this example. The local parameters of object B always contains the values defining the current state. It will always remember it's size 0.2, because that parameter will never change more. Further object B only have to do the graphic translation between the two positions. Object A just remembers the initial state and must always shrink down to size 0.2 before the translation to the actual position. In addition to the disadvantage mentioned above, the 'full copy objects' also add some overhead because they use the method copy before doing graphics actions. Those classes that are 'full copy objects' are: Polygon, Padlock and Menu. 'Non full copy objects' are for instance: Calculator, Handins and Slidepot.

Problems with rounding errors did not occur immediately in the project, this fact explains way 'full copy objects' were not used at all from the beginning. After a time those objects which were necessary to keep the accuracy were changed to 'full copy objects'. Because of the powerful hardware used in the project the advantages of the 'non full copy objects' are very small. The conclusion is that only 'full copy objects' should be used further in the project.

4.3 The base class

class Prim

Inherits from: Object

Classes used:

Source file: prim.m

Prim is the superclass of all visible objects in FIP. Prim itself cannot create a visible object. It contains a lot of methods and variables that can be used when creating new classes.

+ **new** *Method*
  Returns a new instance with all default values initiated. It invokes the methods **setDef** and **setOrig**.

- **setDef** *Method*
  Sets the default-values of the instance. It is only invoked from the the method **new**.

- **setActDev:***(short)aDevicehandler* *Method*
  Sets the workstation handler. In fact a pointer to the handler of the workstation. More detailed information about the concept workstation in Section 3.2.

- **(short)getActdev** *Method*
  Returns the identity of the workstation handler.

- **setActive:***(BOOL)aBoolean* *Method*
  Sets the boolean *active*. This parameter tells if the object is in active mode or not. This method is usually invoked from the method **hit:** which detects where the mouse-clicks hit. If the cursor hits inside the area of the object when *active* is FALSE, *active* will change to

TRUE. If *active* already is TRUE, *active* will not change. Finally when hitting the outside, *active* will change to FALSE.

- **setMovable:***(BOOL)aBoolean* *Method*
  Sets the boolean *movable*. If movable is set to TRUE, then the object is movable on the screen. The default value of *movable is TRUE.*
- **setIntDevId:***(id)anId* *Method*
  Sets the device-handler of the interaction-device. The interaction-device provides the communication between the mouse and the screen.
- **setHeadTxt:***(char *)aString* *Method*
  The variable *headTxt* assigns *aString*. The text do not have to be at the top of the object as the name says, the position is optional.
- **printHdTxt** *Method*
  Prints the head-text at the screen with start from position *headTxtPos* which contains the print-position.
- **setVal:***(double)aDouble* *Method*
  The parameter *value* assigns *aDouble*. It is used by classes like Hand-ins and Slidepot. The value can describe the normalized position of the pointer of the circular-pointer instrument, for instance. It can also contains the output value of the calculator.
- **(double)getVal** *Method*
  Returns the contents of *value*.
- **setOrig** *Method*
  Sets the instance's origin in the center of the hit-area, which defines by the variable *hitrec* (described later in this section).
- **setEncaps:(BOOL)aBoolean** *Method*
  Sets the boolean *encaps*. Tells if the instance is used by another instance.
- **moveToAbsOrig** *Method*
  Moves the visible object to the absolute origin of the VDC system.
- **setScalPR:***(struct pos *)aStructPos* *Method*
  Sets the scaling-point relative the instance origin. The scaling-point is the center in a scaling action on the object.
- **setScalPA:***(struct pos *)aStructPos* *Method*
  Sets the scaling-point relative the origin of the VDC system.
- **setRotPR:***(struct pos *)aStructPos* *Method*
  Sets the rotating-point relative the origin of the instance. This method is only used by rotatable objects.
- **setRotPA:***(struct pos *)aStructPos* *Method*
  Sets the rotating-point of rotatable objects relative the origin of the VDC system.
- **rubb:***(short)aDevicehandler* *Method*
  Clears everything inside the hit-rectangle. Is invoked from method **show:** before drawing.
- **moveA:***(struct pos *)aStructPos* *Method*
  Moves the object to position *aStructPos* relative the origin of the VDC system.
- **moveR:***(struct pos *)aStructPos* *Method*
  Moves the object to position *aStructPos* relative the old position.

- **setSizeR:***(double)aDouble* *Method*
  Sets the size relative the old size (size = size * a*Double*).
- **setSizeA:***(double)aDouble* *Method*
  Sets the size of the object.
- **(double)getSize** *Method*
  Returns the size.
- **(struct pos *)getPlace** *Method*
  Returns a pointer to the current origin-position of the object.
- **(BOOL)getMovable** *Method*
  Returns the boolean *movable* which decides if the instance is movable or not on the screen.
- **insidehit:***(struct pos *)aStructPos* *Method*
  Tells if the position a*StructPos* is inside the hit-rectangle. The hit-rectangle is the area which defines where the object is sensitive for mouse-clicks.
- **hit:***(struct pos *)aStructPos* *Method*
  At the base class level the method **hit:** is equal with the method **insidehit:**. See the description of the other classes about the method **hit:**.
- **shifthit** *Method*
  It handles the updating of the rub out area *oldrec*. It is invoked from the method **show:**, to prepare for method **rubb:** which erase everything inside *oldrec*.
- **update:***(short)aDevicehandler* *Method*
  Update all the dynamic parameters of the instance. It also updates the look of visible objects if the changed parameters are visible on the screen.

Following methods give the same result as the methods above with the same name except the capital F, they are used by the 'full copy objects'.

- **FmoveA:***(struct pos *)aStructPos* *Method*
  See explanation of **moveA:**.
- **Fshifthit** *Method*
  See explanation of **shifthit:**.
- **Fhit:***(struct pos *)aStructPos* *Method*
  See explanation of **hit:**.
- **Finsidehit:***(struct pos *)aStructPos* *Method*
  See explanation of **insidehit:**.

In this part the most important variables are mentioned, those variables that are necessary to understand the program when reading the code. The common type *struct pos* is defined in file owndef.h.

hitrec*(struct pos [2])* *Variable*
  Defines the hit-area. The hit-area is an unvisible rectangle, which defines where the object is sensitive for mouse-clicks. The instance method **insidehit:** check if the cursor is inside the hit-area.

oldrec*(struct pos [2])* *Variable*

Before an operation like scaling, rotating and translation can occur on the screen, the old picture must be erased. Method **show:** invokes **rubb:** to erase the old picture. The *oldrec* decides what area to be erased. The method *shifthit:* takes care of the updating of *oldrec*.

origin*(struct pos)* *Variable*

The origin of the object absolute the VDC system (see Section 3.2).

rotpoint*(struct pos)* *Variable*

Describes the rotating-point of rotatable objects.

scalep*(struct pos)* *Variable*

Describes the scaling-point. The scaling-point is the point which the object will shrink and grow around.

size*(double)* *Variable*

Normally this value is between 0.1 and 1.0. It is a normalized value that describes the relative size to the original size, that is 1.0.

value*(double)* *Variable*

Varying from 0.0 to 1.0. Some classes like Handins use this parameter to store the actual pointer-position for instance. The class Slidepot use it to store the current value.

encaps*(BOOL)* *Variable*

Tells if the instance is encapsulated by another instance.

movable*(BOOL)* *Variable*

Decides if the object is movable on the screen, which means that the object can be moved by using the mouse-interaction. Default value is TRUE.

active*(BOOL)* *Variable*

Tells if the instance is in active mode or not. If the instance is in active mode it can feel and 'obey' the mouse-clicks that hit the object on the screen. It is significant in method **hit:**.

4.4 Dialogue primitives

class Dialogpr ———————————————————————————

Inherits from: Prim

Classes used:

Source file: dialogpr.m

The class Dialogpr is a link between the dialogue primitives and the superclass Prim. For the moment the function is to provide a good structure in the inheritance-tree. The dialogue primitives are classes that supports two way communication. That means that the operator can control the dialogue primitive, usually by using the mouse.

class Calculator

Inherits from: Dialogpr

Classes used: Keyboard, Window, Pushkey

Source file: calculator.m

The calculator class implements the dialogue primitive that provides communication between the user and the program with an object that looks like an calculator. The calculator is intend to provide a powerful way of generate and send values in the system. The method **hit:** is quite complex in this class definition. All local variables are stored in the structured type *calcType*. See the code for more information. Only methods that differ much in performance from those in super class Prim are treated in the subsequent text.

- **hit:***(struct pos *)aStructPos* *Method*
  When the method **hit:** is invoked, different issues will occur depending on the mouse-buttons status, if *aStructPos* is inside the hit-area or not, and finally the status at the variables active and movable.
- **show:***(short)aDevicehandler* *Method*
  This method makes the instance visible on the screen and updates dynamic data.

class Menu

Inherits from: Dialogpr

Classes used: Menubit

Source file menu.m

The methods in the class Menu support drawing of menus. A menu is not capable itself to decide what will happen in each mouse-button event. The class that uses Menu have to implement these actions at local level. The local variables for the menu class is stored in the structured variable *menu*, of type *menuType*.

+ **new:***(short)aShort nrRow:(short)aShort nrColl:(short)aShort* *Method*
  Returns a new instance of *menutype*. The first parameter will tell how many characters each row in the menu can contain. The second and the third parameter describe the wanted menugrid.
- **setMenText:***(char *)aStrpointer* *Method*
  row:(short)aShort coll:(short)aColl
  This method puts a character-string into the string-matrix of the instance.
- **show:***(short)aDevicehandler* *Method*
  This method will show the specified menu at the screen. It will show a grid with text sat by the method setMenText.
- **setOrig** *Method*
  This method setOrig differs from the supermethod. Instead of setting the origin in center of the hit-rectangle it places the origin in the lower left corner of the menu.

class Slidepot

Inherits from: Dialogpr

Classes used:

Source file: slidepot.m

The slide-potentiometer class provides methods which can be used in different applications where a potentiometer is convenient. The variables that differ from the superclass is stored in the structured variable *slidepot*, of type *slidpotType*.

- setVal:*(double)aDouble* *Method*
  This method differs from **setVal** in the superclass because it also sets the vertical sledge-position, proportional to the input value.
- update:*(short)aDevicehandler* *Method*
  Updates the slide-potentiometer only if the value are changed. It draw the whole slide-potentiometer.
- hit:*(struct pos *)aPosition* *Method*
  The method **hit:** control the performance of the slide-potentiometer depending on the mouse-buttons status and the cursor position. If the slide-potentiometer is in active mode and the mouse-click is inside the hit-area the sledge will move to the actual position. The method **hit:** invokes **update:** to perform this.

class Padicon

Inherits from: Dialogpr

Classes used: Openlock, Shutlock, Lockmen, Calculator

Source file: padicon.m

The Padicon class supports an icon that can be used to control the communication between different instances of different classes. The structured type for this class is *padIconType*. The data of the instance is stored in variable padIcon of mentioned type.

- hit:*(struct pos *)aStructPos* *Method*
  When **hit:** is invoked, different issues will occur depending on the mouse-buttons status, if the position *aStructPos* is inside the hit-area or not, and finally the status of the variables *active* and *movable*.
- show:*(short)aDevicehandler* *Method*
  Shows the padlock on the screen either as an opened padlock or a closed one, depending on the status of variable *padIcon.open*.
- (BOOL)getOpen *Method*
  Returns the status at the boolean *padIcon.open* which tells if the padlock-icon is open or not.

4.5 Presentation primitives

class Presprim

Inherit from: Prim

Classes used:

Source file: presprim.m

The characteristics of presentation primitives is that they only provide one-way communication. You cannot control them by using the mouse for instance, except moving them. You can use them to build other objects, like the class Calculator use the class Keyboard and the class Window in its class definition.

class Handins

Inherits from: Presprim

Classes used: Ptr1, Screw

Source file: handins.m

Provides a circular-pointer instrument that can be used to make parameters in a process description visible. The structured variable *handIns* contains local variables of the class Handins.

- **setUnit:***(char *)aUnit* *Method*
  Put the string *aUnit* in the variable *handIns.unit*. Invoking method **printunit** which write the text on the instrument.
- **printunit:***(short)aDevicehandler* **text:***(char *)aText* *Method*
  Printing the unit on the instrument.
- **setVal:***(double)aVal* *Method*
  A value 0.0 - 1.0 corresponds to a pointer-angle of 180 - 0 degrees. The new pointer-position will be visible when the method **update:** or method **show:** is invoked.

class Polygon

Inherits from: Presprim

Classes used:

Source file: polygon.m

Supports all kinds of polygon drawing. Polygon itself cannot create visible objects. The approach is to create a subclass with the wanted look. Polygon is the only class (so far in the FIP-project) that supports rotation. Polygon and all its subclasses have their local variables stored in the structured type *polygon* of *polyType*.

- **setOrig** *Method*
  Sets the origin equal to the scaling-point of the polygon-instance.
- **rotateR:***(double)anAngle* *Method*
  Rotates the object *anAngle* degrees relative the old angle.

- **rotateA:***(double)anAngle* *Method*

  The polygon will point in a direction corresponding to the value of an*Angle* after invocation. Not visible until show has invoked.

Polygon have two subclasses named Ptr1 and Pointer, both are used as pointers by the circular-pointer instrument. They are not treated separately in this report, but the code will provide necessary information. The source files are ptr1.m and pointer.m respectively.

class Keyboard

Inherits from: Presprim

Classes used: Pushkey

Source file: keyboard.m

Keyboards are principally used by other classes like the Calculator class. The structured type *keybrdType* describes the local parameters used in the keyboard-class. There are no unique methods in this class regarding to their performance.

class Menubit

Inherits from: Presprim

Classes used:

Source file: menubit.m

Menubits are mostly used as building blocks for menus. The structured variable *menubit* of *menbitType* contains local variables not inherited.

- **setTxtLgth:***(short)aLength* *Method*

  Decides the maximum length of the text.

- **setTxtSize:***(short)aTxtSize* *Method*

  Sets the size at the characters of the text. The size varying from 1 to 100.

- **setText:***(char *)aText* *Method*

  Put the string *aText* into the instance-variable *menubit.text*. This text will be visible when **show:** is invoked.

- **(char *)getText** *Method*

  Returns a pointer to the string *menubit.text*.

class Screw

Inherits from: Presprim

Classes used:

Source file: screw.m

Provides a picture of a screw. It can be used as an adornment by other classes, as the class Handins does for instance.

class Window

Inherits from: Presprim

Classes used:

Source file: window.m

The Window class provides in fact a display. The bad chosen name be due to the name Display is already taken by another class. The windows is used by classes that need a display. The Calculator class is a good example.

- **setText:***(char *)aText* *Method*
  Put the string *aText* into the instance-variable *window.text*.
- **(char *)getText** *Method*
  Returns a pointer to the string *window.text*.

class Padlock

Inherits from: Presprim

Classes used:

Source file: padlock.m

The function of the class Padlock is as father of the subclasses Openlock and Shutlock. The classes Openlock and Shutlock offer pictures, representing an open padlock and a shut one respectively. These two classes is used by the class Padicon. The source files of these classes are openlock.m and shutlock.m.

4.6 Device classes

class Interdev

Inherits from: Object

Classes used:

Source file: interdev.m

The interaction device supports the interaction between the screeen and the mouse. The status at the mouse is decided by the variables xyOut(short[2]), pressed(short), released(short) and keystate(short). Following methods make them available.

- **update:***(short)anInhandle outHandle:(short)anOutHandle* *Method*
  This method updates the status of the mouse-device. The parameters *anInhandle* and *anOutHandle* decides what workstation you want as input station and output station respectively.
- **(struct pos *)getxyOut** *Method*
  Returns a pointer to the instance-parameter *xyOut*. This contains the current cursor-position.

- (short)getReleased *Method*

  Returns a code for which buttons released since the last invocation to the method **update:**.

  0: none

  1: button 1

  2: button 2

  3: both buttons

- (short)getPressed *Method*

  Returns a code for which buttons pressed since last invocation to **update:**.

  0: none

  1: button 1

  2: button 2

  3: both buttons

- (short)getKeystate *Method*

  Returns a code describing the pressed buttons.

  0: none

  1: button 1

  2: button 2

  3: both buttons

class Device ———————————————————————

Inherits from: Object

Classes used:

Source file: device.m

In this project two devices are used, mouse and display. Mouse and Display are subclasses of the class Device.

class Mouse ———————————————————————

Inherits from: Device

Classes used: Wkctrl

Source file: mouse.m

The Mouse class supports the workstation of the mouse. The *inHandle* is a pointer to the workstation.

- close *Method*

  Closes the workstation.

- (short)getInHandle *Method*

  Returns a pointer to the workstation of the mouse.

class Display

Inherits from: Device

Classes used: Wkctrl

Source file: display.m

The Display class supports the workstation of the display. The *outHandle* is a pointer to the workstation.

- **close** *Method*
  Closes the workstation.
- **clear** *Method*
  Clears the screen.
- **(short)getHandle** *Method*
  Returns a pointer to the workstation of the display.
- **(short)workout:***(short)anIndex* *Method*
  Returns devicedependent parameters stored in the workstation instance. More about this in the description of the class Wkctrl.

4.7 Other classes

class Pipe

Inherits from: Object

Classes used: Presprim, Dialogpr

Source file: pipe.m

Provides the communication between dialogue primitives and presentation primitives, or between two dialogue primitives. All methods implemented in the class Prim ought to be implemented in Pipe. The syntax for the usual methods can be seen below.

```
- aMethod {
    if ([aReceiver isKindOf:Presprim])
        return [aReceiver aMethod];
    else if ([aReceiver is KindOf:Dialogpr])
        return [aSender aMethod];
}
```

- **setOpen:***(BOOL)aFlag* *Method*
  Sets the boolean *open* and then returns the receiver. If open is FALSE no communication is possible. Default value is TRUE.
- **setRecei:***(id)anObject* *Method*
  Sets the receiver and returns its' identity.
- **setSend:***(id)anObject* *Method*
  Sets the sender and returns the receiver.
- **getRecei** *Method*
  Returns the receiver of the pipe.

- **getSend** *Method*
  Returns the sender of the pipe.
- **setMult:***(double)mult* *Method*
  Sets the gain-factor aMult of the pipe.
- **(double)getMult** *Method*
  Returns the gain-factor a*Mult*.

aMult*(double)* *Variable*
Is the gain-factor between the sender and the receiver in the pipe.

open*(BOOL)* *Variable*
Tells if the pipe is open or not.

aReceiver*(id)* *Variable*
A pointer to the receiver in the pipe connection.

aSender*(id)* *Variable*
A pointer to the sender in the pipe connection.

class Drive

Inherits from: Object

Classes used:

Source file: drive.m

The Drive class simulate a PID-controlled sawmill. The simulation is controlled by the system time. The sample-time is 100 ms.

- **setVal:***(double)aVal* *Method*
  Sets the speed reference and returns the receiver.
- **(double)getVal** *Method*
  Returns the speed.
- **update** *Method*
  Change to a new value if this is sat, returns the receiver.

class Demo

Inherits from: Object

Classes used:

Source file: demo.m

The Demo class provides some demonstration methods that can be useful when creating new primitives.

- **moveRD:***(id)anId devh:(short)aDev dist:(short)aDist angle:(double)anAngle* *Method*
  Move the object *anId* the distance *aDist* in the direction *anAngle*. The parameter *aDev* decides the workstation. The distance will be given in VDC-units.
- **ellipse:***(id)anId devh:(short)aDev x:(short)aX y:(short)anY* *Method*
  Move the object *anId* in an elliptical orbit. The ellipse is characterized by x and y in VDC-units.

- **rotateD:***(id)andId devh:(short)aDev* *Method*
  Rotate the object one round around its rotating-point. This method is only valid for rotatable classes as Polygon.
- **sizeUpD:***(id)anId devh:(short)aDev* *Method*
  Size up the object *anId*, and then shrink it down again.

class Wkctrl

Inherits from: Object

Classes used:

Source file: wkctrl.m

The workstation-controller provides methods that operate on the workstation. The variable *work_out(short[66])* describes the machinedependent parameters. Invoking the method **workout:** if you want information. Read more about the workstation in Section 3.3 and [GSS,1986b].

- **open** *Method*
  Opens the workstation if no error occurs.
- **close** *Method*
  Closes the workstation.
- **clear** *Method*
  Clears the workstation, in practice it clears the screen.
- **(short)workout:***(short)anIndex* *Method*
  Returns the value of instance-variable *work_out[anIndex].*
- **(BOOL)error_sign** *Method*
  Returns an error-flag describing an eventual error occured when attempt to open a workstation.

4.8 Main program

In this project the main programs that have been developed were used to test the classes. They are not relevant by themselves. The program listing includes code for maind.m and mainf.m. These are the two most important files as they are the main programs for the two demos fipview.exe and drivel.exe. The PTC is a nonimplemented class that will be implemented in FIP. This class will take over the role of the mainprogram in this version.

4.9 The include files

These files are standard in C. They have the extension h to show that they are include files. There are many standard h-files used in this project. But they are not treated here. If you want more information about the predefined h-files, see the code and [Microsoft,1986b]. However there are some h-files created in the project. A brief description of them is below.

cgi.h

Includes all those constants used in function-calls to CGI. All constants are written with capitals to distinguish them from variables.

owndef.h

Includes some useful constants and structured type-definitions used in the program. All constants are written with capitals to distinguish them from variables.

matrix.h

A very important file in this project. This file includes all those functions that support the graphics actions like scaling, rotating and translation. To understand what these functions do, read appendix A first. Notice, this file just have to be included in the main file. A brief description of the most important functions is below.

rotate*vect,n,angle,rotpoint)* *Function*
Rotate the *n* first elements in vector *vect angle* degrees around *rotpoint.*

scaling*(vect,n,size,scalep)* *Function*
Scale the *n* first element in vector *vect* with factor *size* around position *scalep.*

moveA*(vect,n,distance,origin)* *Function*
Translate the *n* first element in vector *vect* the distance *distance* relative the absolute origin.

mat3mult*(m1,m2,m3,resm)* *Function*
Multiply the matrices *m1, m2 and m3.* The result is in *resm.*

vectmult*(vect,n,M)* *Function*
Multiply vector *vect* and matrix *M.* The result is in *vect.*

5. Concluding remarks

5.1 Results and experiences

One result of this master thesis project, was the learning about all problems that occur in graphics applications. We have seen that it is important to have a strict design and choose good names for methods. Since the classes are a kind of abstract data-types, it is a good help for the next programmer if the names of the methods describe what the methods really do, without having read the code.

A special aspect regarding object-oriented language is inheritance. We have learned how to structure the inheritance tree for future project. This project suffer from some 'inheritance-bugs'. We also have learned about encapsulation. In the beginning when the objects were built, there were too much code in each class, instead of encapsulate from a few but smaller classes. A describing example was our first realization of the circular-pointer instrument. In the beginning the pointer was implemented directly in the Handins class. But in the last version of the Handins class the pointer class is encapsulated. The benefits of this action were, for example: the code got more structured, the Handins class code became smaller and gives a better survey of the code, it is now easy to shift to another pointer in the circular-pointer instrument in just a moment. Section 5.3 describes possible improvements. It has also resulted in a graphics object library which contains all those classes described in Chapter 4. A library for 2D-transformations was developed too, see Section 4.9.

Many discussions took place during the project about how to rub out old objects or parts of objects when they were updated. At the beginning the objects to be updated were erased from the screen by drawing the object in black. This approach worked well for static objects which did not change their look. The circular-pointer instrument was more difficult to update with this method, because the position of the pointer must be known exactly. The conclusion was to rub out the whole display of the instrument when updating, then it was not necessary to know the old position of the pointer. The disadvantage of this approach was that objects which were not changed must be drawn again. When erasing the whole object the approach was to fill the whole area with the background color. The result was quite a big drawback that we have not considered. Although the hardware was powerful it took long time to fill whole areas. A facility that was not planned to be included was the possibility to edit the given process picture. It became implemented simply because it was so easy to realize. The best way to form an opinion of the result is to run the demo-programs, if they are available.

5.2 Problems

During this project, there have been some problems. Most of the problems refer to the CGI software. In fact it was not the CGI itself that did not

work, it was the driver between the CGI and the hardware. The reason was that a beta-release of the CGI-package was used. The CGI driver was under development during the current work and better versions came after a time. We also had problems with the memory, which has not been sufficient for all intended applications. Some functions which were desirable to us, were bit-mapping and those functions which support output of text, but these functions still do not work very well in the latest release.

In the beginning of the project good help came from Per Sevrell who had made a graphics editor in Objective-C before. But there were a lot of differences between the new Objective-C version and the old one, also the way of compiling and linking differed a little. This fact sometimes made it difficult to use the old experiences of Objective-C and CGI. The total impression of the environment is good. All problems we have experienced are solvable.

5.3 Possible improvements

Because this project has been a way of getting more experiences about the MMC-domain, we have not done something about all improvable points during the time of this project. One point that we would have done something about was the flickering screen. When we updated the screen very frequently the screen flickered. This is due to the drawing method that was used. Instead of drawing the objects direct at the screen-map, it would be better to draw the objects in a hidden map. When ready, you map the picture into the screen. This action would have stopped the flicker. But the bitmap functions did not work, so this was not possible to implement in the given time.

When we choose the number of bits to represent coordinates, short integers (16 bits) were chosen. This was a mistake. Although the VDC-system only requires 16 bits representation it would have been better to represent the coordinates with 32 bits during the graphics calculations. At the first consideration it seems unnecessary to use 32 bits when the VDC-system only allows 16 bits. When calculating the involved matrices, the intermediate results can cause an overflow, which can be avoided with 32 bits representation. This causes some extra overhead to convert back to 16 bits representation before calling the CGI-functions.

Many methods used are quite often overridden now (overriding means that the class does not inherit the method from its father, it has its own version of the method). If we had made them more general this sometimes could have been prevented. Another aspect of this is that general methods are almost always slower because they have more tests included.

The inheritance-structure could have been better, but this partially depends on Objective-C. Sometimes one finds out that the recently created class has features in common with another class. If it would have been possible to have more than one father, this would have solved these problems (multiple inheritance is possible in some other object-oriented languages). A possible improvement worth mentioning is about the classes Display and Mouse. They now inherit from the class Device, but it might be better if they inherited from the class Wkctrl instead.

5.4 Objective-C

Although we were not very experienced in Objective-C when starting this project, our opinion is overall good about this language, except execution-time. We have done comparisons between ordinary C and Objective-C regarding execution-time. Objective-C increase the execution-time about 3 times. Another object-oriented C-dialect is C++; this language increases the time just 10 percent.

It is very important to understand the concept 'self' in objective-C before you start serious programming. Otherwise you will get some unpleasant experiences of this quite soon.

Objective-C offer a lot of predefined classes. In this project we have used the predefined class OrdCltn, that stands for Ordered Collection.

The base in Objective-C is the programming-language C. About C one can have many different opinions. We take the advantages first. It can be said that it is a very flexible language that supports a lot of good functions. You nearly have the same possibilities to bit-manipulations as in assembler-programming. Another advantage of C is that it is a very fast language. The compiler generates fast code.

Now the disadvantages. Since C offers this flexibility it is easy to fall into traps that destroy the structure in the program. A big disadvantage in C is the hierarchy of the procedures. C only allows one level at the function-declarations. If you are familiar with Pascal or another high-level language you know the benefits to have functions declared inside other functions. The possibilities of bit-manipulations can be regarded as a risk. If you are not a very experienced programmer you should not use these functions.

Another point that could be improved in C is the symbols for logical comparisons. C uses the character combination '!=' that stands for not equal. If you happened to write '=!' but '!=', the compiler will understand that as assigning NOT, because a single '!' stands for the logical NOT. This can be avoided by define your own symbols.

Some of the bad things with C can be avoided in Objective-C anyway. One disadvantage in C that can be avoided in Objective-C is the bad procedure-hierarchy. Because there is a built-in inheritance-hierarchy in Objective-C there is no greater need for procedure hierarchy.

5.5 CGI

Our overall impression of CGI is positive. The bugs in CGI (in fact the CGI driver) are only relevant for our beta-release. It is quite easy to learn using the CGI packet. The manuals are a bit confusing the first times, and they continue to be that the whole time, in other words, they could have been better. The only thing we have been lacking in the CGI are windows which are standard in VDI, which stands for Virtual Device Interface. The graphical actions like scaling, translation and rotation could have been implemented in the CGI concept as well.

Acknowledgements

This report is a documentation of the final work of my computer engineering education at Lund Institute of Technology, University of Lund, during the spring 1988. The work was done at ASEA BROWN BOVERI, department KLL at IDEON in Lund. Primary instructor was Henrik Pålsson at KLL, and secondary instructors were Dag Brück and Sven-Erik Mattsson at Department of Automatic Control, LTH.

I wish to thank all the people at ASEA BROWN BOVERI in Lund for their great support and patience during my time here. It has been a very instructive period in my education. The work has not always went the straightest way, it's probably a taste of the real world outside the safe school, with all their well defined problems. Per Sevrell and Bo Johansson were a good help for me in the beginning of the project. I also want to thank Dag Brück and Sven-Erik Mattsson for their help and comments during the writing of this report. Finally I will thank my instructor Henrik Pålsson for our great collaboration during the time of this project.

References

WAITE, MITCHEL, STEPHEN PRATA and DONALD MARTIN (1985): *Programmering i C.*

LUCIANO, DAVID, GEOFF PASCOE, JERRY PRIES AND ALAN WATT (1987): *Objective-C Reference Manual for the PC, Release 3.3.1*, Productivity Products International, Inc.

MICROSOFT (1986a): *Microsoft C Compiler for the MS-DOS Operating System, Run-Time Library Reference*, Microsoft.

MICROSOFT (1986b): *Microsoft C Compiler for the MS-DOS Operating System, User's Guide*, Microsoft.

GSS (1986a): *GSS*CGI C Language Reference Booklet*, Graphic Software Systems, Inc.

GSS (1986b): *GSS*CGI Programmer's Guide*, Graphic Software Systems, Inc.

Appendix A. 2D-transformations

Overview

The concept of transformation is to move a several primitives from one position to another. This can be done in three different ways. Figure 1 illustrates this for a rectangle.

By adding a number D_x to all x-coordinates and another number D_y to all y-coordinates, the rectangle will be moved to a new position in the x, y plane. This is translation.

By multiplying all x-coordinates with a number S_x and all y-coordinates with a number S_y, this will result in a scaling of the rectangle.

The third form of transformation is rotation, this will be possible by using trigonometric functions as sine and cosine.

Homogeneous coordinates

The three transformations can be written as follows

$$(x', y') = (x, y) + (T)$$
$$(x', y') = (x, y) * (S)$$
$$(x', y') = (x, y) * (R)$$

where T is a row-vector, S and R are 2×2 matrices. As we can see, translations means addition, but scaling and rotation means multiplication. To be able to treat all three of them as multiplication we introduce the homogeneous coordinate concept. By using this technique it is now possible to write the

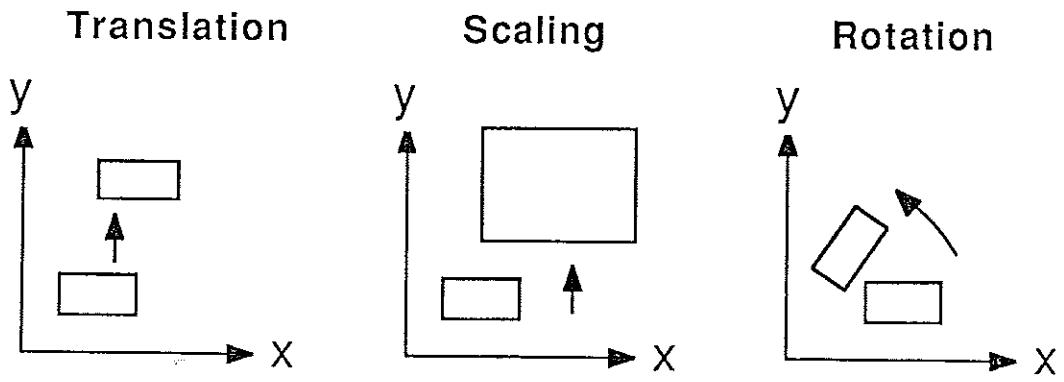

Figure 1. A rectangle transformed by translation, scaling or rotation.

$$T = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ Dx & Dy & 1 \end{bmatrix} \qquad S = \begin{bmatrix} Sx & 0 & 0 \\ 0 & Sy & 0 \\ 0 & 0 & 1 \end{bmatrix} \qquad R = \begin{bmatrix} r11 & r12 & 0 \\ r21 & r22 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Figure 2. Transformation matrices used in 2D-transformations.

equations above as follows

$$(x', y', 1) = (x, y, 1) * (T)$$
$$(x', y', 1) = (x, y, 1) * (S)$$
$$(x', y', 1) = (x, y, 1) * (R)$$

where T, R and S are all 3×3 matrices. See Figure 2 for the contents of the matrices. There are occasions when you want to do several transformations in a sequence. An example of this, move an object to the origin, change the size, then move back to the first position again. It can be proved that it is feasible to do an arbitrary number of transformations using a single 3×3 matrix M, where $M = T * R * S$.

Appendix B. Compilation

Compiling

Since Objective-C offers a natural modularity, you save a lot of time when compiling. It is enough to compile those files containing the changed data-area. There are some exceptions of this point however. If you add a new method to a class it is necessary to compile all files. But if you override a method already declared by its father you just have to compile the file containing the overriding method. Overriding means to create a new method with the same name as the the super-method (a super-method is a method inherited from the ancestor-class). The command used to compile is `objcc -c filename`. See [Luciano, 1987] and [Microsoft, 1986b] if you want more information about compiling. During this project we have had problems when compiling since the memory is not sufficient for the compiler. A necessary action is to shut off the drivers before compiling because they demand a lot of memory. This can be done by making the drivers transparent. The option /T written after the assignment of the driver in file config.sys will achieve this. When the command `drivers /R` is given before compiling, the drivers with the option T will be shut off. See also the file cmpall.bat.

Linking

All files used in the program must be linked together before running. This can be done automatically by running the makefile link.bat. If you will complete with more files, it's just to put them into the code. The file objlink.lnk contains all those files that demands for this project. See also [Luciano, 1987] and [Microsoft, 1986b] about linking.