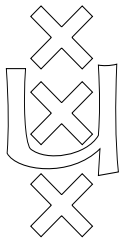

University of Amsterdam

Dept. of Social Science Informatics (SWI)
Roetersstraat 15, 1018 WB Amsterdam
The Netherlands
Tel. (+31) 20 5256786

PceDraw

An example of using PCE-4

Jan Wielemaker

jan@swi.psy.uva.nl

This document describes the design and implementation of PceDraw, a drawing tool written in PCE-4/Prolog. PceDraw exploits many of the features of PCE and is written according to our current ideas on using PCE/Prolog.

Contents

Chapter 1

Introduction

One of the aims of writing PceDraw is to provide users of PCE who have made their first steps in using the system with an example that explains how large applications can be realised using PCE/Prolog. This document motivates the decisions taken to arrive at PceDraw, both at the level of the overall design and at the level of the detailed design and implementation.

This document is part of the documentation of PCE-4. The complete documentation consists of:

- *Programming in PCE/Prolog [Wielemaker & Anjewierden, 1992b]*
  This document is an introduction to programming in PCE/Prolog. It provides the background material to understand the other documentation.
- *PCE-4 Functional Overview [Wielemaker & Anjewierden, 1992a]*
  This document provides an overview of the functionality provided by PCE. It may be used to find relevant PCE material to satisfy a particular functionality in your program.
- *PCE-4 User Defined Classes Manual [Wielemaker, 1992]*
  This document describes the definition of PCE classes from Prolog. PceDraw is implemented as a set of user-defined classes.
- *The online PCE Reference Manual*
  The paper documents are intended to provide an overview of the functionality and architecture of PCE. The online manual provides detailed descriptions of classes, methods, etc. which may be accessed from various viewpoints. [Wielemaker & Anjewierden, 1992b] describes how to use the online manual.

This document aims at PCE users who have understood the basics of PCE and have some experience with Prolog. In its final context (as an appendix) the tutorial should provide the necessary material. When new constructs are introduced in this document they are often explained. It is adviced to read chapter 2 first and proceed with the introduction and the first section of chapter 3. The remaining material may be used as a set of examples. At the end of this document is an index, indicating references to methods, predicates and files discussed.

Chapter 2 explains the overall design of PceDraw. Chapter 3 contains a brief overview of the organisation of the sources, followed by the annotated sources.

Two chapters that will be part of the tutorial have been added as appedices to this manual. The first deals with style conventions for the definition of classes and the second with using global object references (e.g. `@same_center`).

Chapter 2

Design

2.1 Functional overview

PceDraw is a drawing tool for creating structured diagrams: flow-charts, diagrams capturing architecture, etc. In this kind of diagrams there is usually a small number of reoccurring shapes that have to be linked to each other. For this reason, the editor should allow the user to create/save/load a library of prototypes. A typical example of such a prototype is a box with centered text. Lines between shapes often represent semantical relations and therefore should remain connected to the shape if the shape is moved/resized and should be destroyed when the shape is deleted.

This document aims at the software design and implementation of PceDraw and therefore the requirements analysis and functional specification is very brief. For getting a clear view on the functionality it is adviced to run PceDraw. It can be started from xpce by the command:

```
1 ?- pcedraw.
```

PceDraw has been designed from these principles. The initial tool consist of three areas: the drawing area itself, a menu with available prototypes and a general command and feedback area. Besides creating, moving, resizing, etc., the tool must be able to edit shape attributes such as the thickness of the drawing pen and the font. This functionality is dealt with by an attribute editor which can be launched in a separate toplevel window.

PceDraw provides two kinds of menu's. All commands are available through pulldown menus in the 'command area' of the tool. Frequently used commands on a single shape are also available through a popup-menu associated with each shape. This approach has several advantages. The pulldown menus provide a place where all functionality can be found (except selecting a prototype and operations performed via direct-manipulation such as selecting, moving and resizing shapes), while the popup menus allows for fast access to the commonly used commands.

The current version of PceDraw does not support keyboard accelerators. Defining accelerators should be supported by PCE's dialog primitives. This will be implemented later.

2.2 Realisation in PCE

After the functionality is specified, PCE primitives that serve as a starting point for the realisation be selected. It is hard to tell how this should be done. PCE contains a large amount of functionality that can be combined in several ways. Examples, the tutorial and the online manual (`manpce/[0-1]`) are the starting point. Below is a brief list with the main choices for PceDraw. See the various sourcefiles in chapter 3 for details.

- *Overall tool*
  A *frame* is a collection of windows and provides an ideal starting point for the overall tool.
- *Drawing area*
  A *picture* is a window indended for displaying arbitrary graphical objects.
- *Prototype menu*
  Two possibilities: 1) *dialog* + *menu* + *menu_item* or 2) *picture* + *bitmap*. See discussion in 'menu.pl'.
- *Command area*
  A *dialog* with a list of pulldown menus organised in a *menu_bar* and a *label* for feedback messages.
- *Shapes*
  Appropriate PCE graphical (box, ellipse, text, line, etc.).
- *Prototypes*
  A *device* is a collection of graphicals that can be manipulated as a single unit. The ←*klone* method can be used to create instances.
- *'Settings' (or attribute) editor*
  A *dialog* window with appropriate *dialog_items* for the various settings.
- *(Direct) manipulation of shapes*
  recognisers can be attached to the various shapes. We can start from the various standard *gestures* defined in PCE. PceDraw can operate in various modes (select, create, edit_text, etc.). A mode attribute can be attached to the drawing area, where it can easily be found from the recognisers, so they can use it as a condition.
- *Load and Save*
  Both prototypes and drawings must be saved and loaded to/from Unix files. This can be realised using PCE's behaviour 'Object →*save_in_file*' and 'File ←*object*'.

2.2.1 Creating an application

After we have selected the PCE building blocks from which to start, we have to extend them so that they fulfill our exact needs and cooperate to form the drawing tool. There are two ways to do this. The first is to regard PCE as a class/object library and extend/combine objects via 'free-style' Prolog code. In this case our entire tool is (from the outside) a collection of Prolog predicates. The second possibility is to create subclasses from the basic PCE classes. Using the latter approach, the entire tool is a class of which

an instance is created. What are the advantages of both approaches? We will look at them from an example.

Suppose we have a drawing area and displaying an object on it should change a 'modified' attribute associated with the drawing area. The PCE class picture is our starting point. Class picture does not have an instance variable 'modified', so our task is to add such a variable and provide means to display an object on it and set the modified attribute.

2.2.1.1 Using PCE as a library

When using PCE as a library, the predefined objects and classes of PCE are regarded as a library of functionality we can access via the Prolog predicates `new/2`, `send/[2-12]` and `get/[3-13]`. There are two ways to modify or extend the behaviour of an object from a standard PCE class. The first is to write Prolog predicates that perform certain operations on the object(s). The second is to use PCE's object-level programming mechanisms to extend the object. Below is the code that results from using Prolog predicates.

```
create_canvas(P) :-
        new(P, picture),
        send(P, attribute, attribute(modified, @off)).

display_canvas(P, Graphical, Point) :-
        send(P, display, Graphical, Point),
        send(P, modified, @on).
```

Although this technique does not create a new (PCE) class, it does create a new 'conceptual' kind of object: the canvas. 'Display' is a method of this new kind. Depending on whether the method is defined in the PCE class or in Prolog, the behaviour should be invoked either via `send/[2-12]` or with the Prolog predicate:

```
1 ?- send(P, selection, @nil).
2 ?- display_canvas(P, box(30,30), @default).
```

The syntactical difference makes it clear whether the action initiates a Prolog predicate —and thus a part of the application— or a method of the PCE library. A programmer using this conceptual kind of object must be aware whether the method is part of PCE or part of the extension. Calling the raw PCE method might lead to inconsistencies: if the user invokes

```
1 ?- send(P, display, box(30,30)).
```

the contents of the canvas will be modified, but the modified attribute won't change.

Extending the object

The second possibility uses programming PCE at the object level. Methods can be assigned to objects similiar to classes. The method object consists of three parts: the name or selector, the type specification and the action or message. The type specification is a

vector with the same number of arguments as expected by the method. Each element of the vector specifies the corresponding type. See the online manual, topic 'types'. While a message implementing a method is executed, `@arg1` is bound the the first argument provided, `@arg2` to the second, etc. See also 'Object →*send_method*'.

```
create_canvas(P) :-
        new(P, picture),
        send(P, attribute, attribute(modified, @off)),
        send(P, send_method,
             send_method(display, vector(graphical, '[point]'),
                         block(message(P, send_class,
                                       display, @arg1, @arg2),
                               message(P, modified, @on)))).
```

Using this solution, the user of the canvas does not need to know that the →*display* method of the raw PCE object has been redefined. The new object has a method named →*display* which not only takes care of displaying the object, but also updates the modified attribute. Remaining problems are:

- PCE object are created using `new/2`, while application objects area created via a Prolog predicate.
- From the outside one cannot tell easily whether the object is a raw PCE object or a modified one.
- If many instances are created, each of them will have method objects attached to them.
- Writing code like this requires the user to know PCE's programming classes (block, if, and, etc.).
- If the implementation cannot be handled by PCE's programming classes a message to `@prolog` is necessary. In this case the implementation will be spread over two locations.
- The code is attached to the object. If —during debugging— this code needs to be changed there is little alternative then destroying the object and recreating it.
- If the object is saved using 'Object →*save_in_file*' or kloned using 'Object ←*klone*', the code part is saved/kloned as well.
- It is difficult to read and write.

Object level programming is not used intensively in PCE, but in some situations it is the best solution.

2.2.1.2 Extending PCE

The alternative provided by PCE-4 is to create a new class for the canvas. Creating a class is done using the normal PCE interface primitives `new/2`, `send/[2-12]` and `get/[3-13]`, but a Prolog defined preprocessor based on the Edinburgh Prolog primitive `term_expansion/2`. This is our solution based on classes.

The `pce_begin_class/3` call creates class canvas as a subclass of (the predefined) class picture. Next, it asserts (using `asserta/1`) a clause for `term_expansion/2` that will convert the class declarations. The optional last argument is the summary documentation of the class. The `pce_end_class/0` call terminates the declaration by removing the clause for `term_expansion/2`.

The `variable/4` declaration is expanded to attach a new instance variable for the class. The arguments are the name, the type, the access rights and the optional summary documentation. The `:->/2` is expanded to define a send method for the class. The first argument is 'self'. The remaining arguments are of the form 'PrologVar:PceType'. The body may start with a line '"...."::', which is recorded as the summary documentation of the method. The remainder is plain Prolog code.

The method →*initialise* is called from the PCE virtual machine (VM) to initialise the instance from the arguments provided with `new/2`. It should be there if the initialisation should do something in addition to the initialisation of the super-class. When defined, the →*initialise* method should perform the initialisation of the super_class:

```
send(Self, send_super, initialise, ...)
```

In this example, the variable ⇆*modified* must be initialised to `@off`.

The →*display* method as defined below redefines the built-in method of class picture by setting the modified flag.

```
:- pce_begin_class(canvas, picture, "Drawing area").

variable(modified, bool, both, "Has diagram been modified").

initialise(C) :->
        send(C, send_super, initialise),
        send(C, modified, @off).

display(C, Gr:graphical, Pos:[point]) :->
        "Display graphical and set modified"::
        send(C, send_super, display, Gr, Pos),
        send(C, modified, @on).

:- pce_end_class.
```

After this, we can use the class as if it were a predefined PCE class:

```
        ...
        new(C, canvas),
        send(C, display, box(30,30)),
        ...
```

User defined classes is one of the three possibilities to build an application in PCE. It does not have the disadvantages of introducing 'conceptual' kinds using Prolog predicates, neither the disadvantages of using object-level programming. Complete applications however

normally consist of a large number of objects with sometimes only slightly different behaviour. Using classes for each of these categories makes it difficult to avoid large amounts of awkward classnames. For this reason, using Prolog predicates or object level programming can be a good alternative for defining a class. It is adviced to use these techniques only for local communication and use class-level programming for global communication between components of the application.

Extending vs. creating classes

PCE/Prolog allows both for extending the behaviour of existing classes and defining new ones. Extending classes implies redefining them, and should first of all be used to (temporary) overcome ommisions in the PCE system itself. Extending behaviour of existing classes may easily affect consistency of large applications, so be careful.

For one case, extending PCE classes may be considered. Suppose we have an application that creates various subclasses of the various predefined subclasses of class graphical (e.g. box, circle, line) and all these classes need to have some common method that can be implemented at the level of the PCE class graphical. In this case it might be desirable to implement the method there instead of at each subclass. If you decide to do so, it is adviced to give the method a name that clearly indicates the application for which it was introduced, so no conflicts with other applications or future PCE extensions is to be feared.[1]

Creating new classes however does not affect the consistency of the system and provides a clean way to extend PCE.

2.3 Class organisation and communication

2.3.1 Overall tool communication

The application as a whole is represented by an instance of class 'draw', which is a subclass of the PCE class frame. Class draw serves as an overall manager of the various parts of the drawing tool. Class frame forms an ideal starting point to do this:

- Any graphical object (and almost anything in such a tool is a graphical object or is closely related to one) can easily find the reference to the tool as a whole using 'Graphical $\leftarrow$*frame*'.
- Class frame can easily find all its parts using 'Frame $\leftarrow$*member*'.

For this reason, the instance of class frame is the ideal part to support communication. For example, feedback can be centralised by defining a method $\rightarrow$*feedback* on the frame. Now, any graphical object can give feedback by doing:

```
send(Myself?frame, feedback, 'I just did this').
```

[1] An alternative (and in this case better) solution to this problem would be to introduce multiple inheritance. Multiple inheritance however introduces various conceptual problems and in the current implementation of PCE unresolvable technical ones.

2.3.2 Drawing area and shapes

Picture and graphical are a communication couple. The drawing area of PceDraw is realised by class draw_canvas which is a subclass of picture. The various shapes that can be drawn are subclasses of closely related standard graphical classes (e.g. box, line). The pair canvas and shape adds responsiveness to user-events, maintenance of changes, etc. to the standard interaction between picture and graphical.

2.3.3 User Events (Shapes and gestures)

Shapes define the 'Shape →*event*' behaviour by forwarding the event to a reusable 'gesture' object. A 'gesture' is an object that allows for the management of a sequence of button-events, starting with a mouse-down and ending with the corresponding mouse-up. PCE defines several standard gestures. The file gesture.pl creates subclasses to implement the specific user-interface needed by PceDraw.

Chapter 3

The Sources

The application is subdivided into a number of files, each of which is a Prolog module file and defines a number of PCE classes that serve a similar role in the overall application. We use the Prolog module system to avoid possible name-conflicts with other packages for predicates used to support the methods. Below is an overview of the files.

- *draw.pl*
  Defines the toplevel predicates and the class 'draw', of which a drawing tool is an instance. Class draw is a subclass of the PCE class 'frame'
- *canvas.pl*
  Defines class 'draw_canvas'; a subclass of class picture. It is the drawing area of the editor.
- *shapes.pl*
  Defines the shapes that can be drawn on the canvas. These shapes are small extensions to standard PCE classes. They add handles for connections and handling user events.
- *gesture.pl*
  Defines subclasses of the PCE gesture classes. These gestures are linked to the shapes to process user events.
- *menu.pl*
  Defines the menu at the right of the drawing area and the (prototype) icons displayed on them.
- *attribute.pl*
  Defines the attribute editor that can be used to modify the attributes of graphical objects.
- *align.pl*
  Defines the automatic alignment functionality. This file is not included in the sources as it adds little to the understanding of xpce.

Conventions

Each source file is given in a section named "Source file name". The actual code is preceded by small line numbers at the left margin.

3.1 Source file "draw.pl"

```
1     /*  $Id: draw.pl,v 1.9 1993/05/06 10:12:58 jan Exp $
2         Part of XPCE
3         Designed and implemented by Anjo Anjewierden and Jan Wielemaker
4         E-mail: jan@swi.psy.uva.nl
5         Copyright (C) 1992 University of Amsterdam. All rights reserved.
6     */

7     :- module(draw,
8             [ draw/0                                % Start drawing tool
9             , draw/1                                % Start editing file
10            ]).
```

3.1.1 Linking other files

This module is the toplevel module of PceDraw. It loads the various other modules and defines class 'draw', of which the drawing tool is an instance.

PCE/Prolog modules that should run on SICStus Prolog must include the library pce, which defines the basic interface predicates. The `require/1` directive loads the requested predicates from the (PCE-)library. None of these declarations are needed for SWI-Prolog as SWI-Prolog will inherit the PCE system predicates from the module 'user' and load the other predicates using the autoloader.

```
11    :- use_module(library(pce)).
12    :- require([ concat/3
13               , send_list/3
14               ]).
```

With this declaration we load the other Prolog modules of PceDraw.

```
15    :- use_module(
16            [ gesture                               % Gestures
17            , shapes                                % Drawable shapes
18            , canvas                                % Drawing plain
19            , menu                                  % Icon Menu
20            ]).
```

The additional file declarations below are not always needed. For this reason they are defined using `pce_autoload/2`. This keeps the initial image small, reducing startup time. Whenever an attempt is made to create an instance or subclass of a class that is defined as an autoload class, PCE will activate the 'undefined_class' member of '`@pce` ←*exception_handlers*'. Using the standard interface setup, this will cause Prolog to examine the autoload declarations and load the specified file.

The library file find_file.pl defines class finder, an instance of which can be used to ask the user for a Unix file. One instance can be used for finding any file that is needed by PceDraw. For this reason we use the `pce_global/2` construct. Whenever `@finder` is passed via one of the interface predicates and `@finder` does not exist, the database of global declarations is searched.

```
21    :- pce_autoload(draw_attribute_editor, attribute).
22    :- pce_autoload(finder, library(find_file)).
23    :- pce_global(@finder, new(finder)).
```

3.1.2 Entry point

Toplevel goals:

- *draw*
  Create a drawing tool and display it.
- *draw(+File)*
  As `draw/0`, but immediately loads a file.

One could choose not to define these predicate and declare the class 'draw' to be the toplevel or public functionality. This actually might be a cleaner solution than the one choosen here.

```
24    draw :-
25            new(Draw, draw),
26            send(Draw, open).
27    draw(File) :-
28            add_extension(File, '.pd', PdFile),
29            new(Draw, draw),
30            send(Draw, open),
31            get(Draw, canvas, Canvas),
32            (   send(file(PdFile), exists)
33            ->  send(Canvas, load, PdFile, @on)
34            ;   send(Canvas, file, PdFile)
35            ).

36    add_extension(Base, Ext, Base) :-
37            concat(_, Ext, Base), !.
38    add_extension(Base, Ext, File) :-
39            concat(Base, Ext, File).
```

3.1.3 Class draw

Class 'draw' defines and manages the entire tool. Its initialisation builds the entire tool and the resulting instance provide means of communication between the various parts. The call

```
40    :- pce_begin_class(draw, frame).
```

starts the definition of a new class 'draw' that is a subclass of class frame. Classes should always be a subclass of some existing class. If there is no particular PCE class to inherit from, this should be class 'object', the root of the PCE class hierarchy.

The term `resource/4` is expanded by the PCE/Prolog class loader. A resource provides access to the X-window resource database. The PceDraw user may specify a value in /.Xdefaults:

```
Pce.Draw.auto_align_mode:       @off
```

```
41    resource(auto_align_mode,       bool,   '@on',
42             "Automatically align graphicals").
```

If the initialisation of an instance of this class differs from the initialisation of its superclass, a method called '→*initialise*' must be defined. It's task is to initialise the new instance. When PCE creates an instance (with `new/2`, `@pce` ←*instance* or otherwise), it allocates memory for it, resets all slots to `@nil` and calls the →*initialise* method. The arguments to this method may differ from the initialisation arguments of the super class. In this case, frame has three (optional) initialisation arguments, while class draw has none.

Somewhere in the initialise method, there should be a call

```
send(Self, send_super, initialise, ...)
```

To invoke the initialisation method of the superclass. The arguments should be valid arguments for the initialisation method of this superclass. The normal schema is:

1. Check the arguments and compute defaults from them.
2. send(Self, send_super, initialise, ...)
3. Do class specific initialisation.

In our case, the various windows that make up the drawing tool are created and attached to the frame.

To avoid a giant clause, a call to the sub-predicate `fill_dialog/1` is made. It is a difficult decision whether or not this should have been realised using 'send(Draw, fill_dialog, D)' and the subsequent declaration of this method. In general, use `send/[2-12]` and `get/[3-13]` for communication between classes, or communication within a class if type-checking or type-conversion associated with PCE methods is useful.

For PCE-3 users, note the use of the term `new/2` in the second and further sends to create the windows inline and get the reference. This approach is preferred over a separate `new/2` and →*append*. It is shorter but -more important- it attaches the canvas immediately to the frame, making the frame responsible for its destruction.

```
43    initialise(Draw) :->
44            send(Draw, send_super, initialise, 'PceDraw'),
45            send(Draw, done_message, message(Draw, quit)),
46            send(Draw, append, new(Canvas, draw_canvas)),
47            send(new(Menu, draw_menu), left, Canvas),
48            send(new(D, dialog), above, Menu),
49            fill_dialog(D),
50            fill_menu(Menu),
51            send(Menu, activate_select).
```

3.1.4 Command area (dialog)

Unlike the icon menu and the canvas, the dialog is just an instance of the PCE class 'dialog'. This approach is taken because the menus in the dialog can easily find the references to the various parts of PceDraw they want to activate. It is cumbersome and unnecessary to send the messages first to the dialog and from there to the appropriate part of the system.

In a sense, it would be cleaner to send the message to the overall drawing tool first and from there to the appropriate part. This would provide all the functionality of the tool menus with the tool as a whole. As a drawback, it implies the code to actually get something done will be spread over three places instead of two:

- The menu
- class draw
- The part that takes care of the actual function.

First of all, a number of obtainers and messages that can be reused in the remainder of the menu are created. This approach has two advantages over doing it 'in-place':

- By giving it a name, it becomes clear which part of the system is referred to or what function the message realises
- It exploits the reusability of messages and obtainers: only one such object is used for all the menus.

Next, the various dialog_items are attached to the dialog. Note again the use of the `new/2` construct in send to get the references. By using 'Dialog →*append*' the dialog_items are placed in a two-dimensional grid. They are given a position when the dialog is created using the 'Dialog →*layout*' method.

Finally, the (popup) menus of the menu_bar are filled. The initialisation arguments of class menu_item are:

- *Value*
  Used to refer to the item. When the ⇆*message* of the menu_item is `@default` and there is a message attached to the menu, this value is forwarded via the message as `@arg1`.
- *Message*
  This message is sent when the item is selected.
- *Label*
  This is a name or image. When `@default`, a default label will be computed from the value. See 'menu_item ←*default_label*'.
- *Condition*
  This message will be evaluated just before the menu is shown. When it succeeds the item will be active, otherwise it will be inactive (greyed). The evaluation of all condition messages in a menu should be fast for good interactive response.

```
52  fill_dialog(D) :-
53          new(Draw, D?frame),
54          new(Canvas, Draw?canvas),
```

```
55          new(Menu, Draw?menu),
56          new(Selection, Canvas?selection),
57          new(NonEmptySelection, not(message(Selection, empty))),
58          new(NonEmptyDrawing, not(message(Canvas?graphicals, empty))),
59          new(HasCurrentFile, Canvas?file \== @nil),
60          send(D, append, new(MB, menu_bar(actions))),
61          send(D, append, label(feedback, 'Welcome to PceDraw'), right),
62          send(MB, append, new(F, popup(file))),
63          send(MB, append, new(P, popup(proto))),
64          send(MB, append, new(E, popup(edit))),
65          send(MB, append, new(S, popup(settings))),
66          send_list(F, append,
67                    [ menu_item(about,
68                                message(Draw, about))
69                    , menu_item(help,
70                                message(Draw, help),
71                                @default, @on)
72                    , menu_item(load,
73                                message(Canvas, load_from))
74                    , menu_item(import,
75                                message(Canvas, import),
76                                @default, @on,
77                                NonEmptyDrawing)
78                    , menu_item(save,
79                                message(Canvas, save),
80                                @default, @off,
81                                and(NonEmptyDrawing,
82                                    Canvas?modified == @on,
83                                    HasCurrentFile))
84                    , menu_item(save_as,
85                                message(Canvas, save_as),
86                                @default, @on,
87                                NonEmptyDrawing)
88                    , menu_item(postscript,
89                                message(Canvas, postscript),
90                                @default, @off,
91                                HasCurrentFile)
92                    , menu_item(postscript_as,
93                                message(Canvas, postscript_as),
94                                @default, @off,
95                                NonEmptyDrawing)
96                    , menu_item(print,
97                                message(Canvas, print),
98                                @default, @on,
99                                NonEmptyDrawing)
100                   , menu_item(quit,
101                               message(Draw, quit),
102                               @default, @off)
103                   ]),
104         send_list(P, append,
```

```
105                     [ menu_item(create,
106                                 message(Menu, create_proto, Selection),
107                                 @default, @off,
108                                 NonEmptySelection)
109                     , menu_item(delete,
110                                 message(Menu, delete),
111                                 @default, @on,
112                                 message(Menu, can_delete))
113                     , menu_item(load,
114                                 message(Menu, load_from),
115                                 @default, @off)
116                     , menu_item(save,
117                                 message(Menu, save),
118                                 @default, @off,
119                                 Menu?modified == @on)
120                     , menu_item(save_as,
121                                 message(Menu, save_as),
122                                 @default, @on,
123                                 Menu?modified == @on)
124                     ]),
125             send_list(E, append,
126                     [ menu_item(expose,
127                                 message(Canvas, expose_selection),
128                                 @default, @off,
129                                 NonEmptySelection)
130                     , menu_item(hide,
131                                 message(Canvas, hide_selection),
132                                 @default, @on,
133                                 NonEmptySelection)
134                     , menu_item(align,
135                                 message(Canvas, align_selection),
136                                 @default, @on,
137                                 Selection?size > 1)
138                     , menu_item(edit_attributes,
139                                 message(Canvas, edit_selection),
140                                 @default, @on,
141                                 NonEmptySelection)
142                     , menu_item(duplicate,
143                                 message(Canvas, duplicate_selection),
144                                 @default, @off,
145                                 NonEmptySelection)
146                     , menu_item(cut,
147                                 message(Canvas, cut_selection),
148                                 @default, @on,
149                                 NonEmptySelection)
150                     , menu_item(import_image,
151                                 message(Canvas, import_image),
152                                 @default, @on)
153                     , menu_item(import_frame,
154                                 message(Canvas, import_frame),
155                                 @default, @on)
156                     , menu_item(clear,
```

```
157                                 message(Canvas, clear, @on),
158                                 @default, @off,
159                                 NonEmptyDrawing)
160                     ]),
161             send(S, multiple_selection, @on),
162             send(S, on_image, @mark_image),
163             send_list(S, append,
164                     [ menu_item(auto_align,
165                                 message(Canvas, auto_align_mode, @arg1))
166                     ]),
167             (   get(Draw, resource_value, auto_align_mode, @on)
168             ->  send(S, selected, auto_align, @on),
169                 send(Canvas, auto_align_mode, @on)
170             ;   true
171             ).
```

3.1.5 Initial prototypes

Fill the menu of the drawing tool with the standard options. After initialising the menu, its ⇆*modified* status is set to `@off` to indicate saving is not necessary. See the file 'menu.pl' for details.

Class draw_menu defines 'draw_menu →*proto*'. The first argument is the prototype, the second the associated mode and the third the cursor that should be displayed in this mode.

```
172   fill_menu(M) :-
173             send(M, proto, @nil,               select,         top_left_arrow),
174             send(M, proto, @nil,               edit_text,      xterm),
175             send(M, proto, draw_text(''),      create_text,    xterm),
176             send(M, proto, draw_box(0,0),      create_resize,  crosshair),
177             send(M, proto, draw_ellipse(0,0),  create_resize,  crosshair),
178             send(M, proto, draw_line(0,0,0,0), create_line,    crosshair),
179             send(M, proto, new(draw_path),     create_path,    cross),
180             send(M, proto, link(link),         connect,        plus),
181             send(M, proto, link(unique),       connect_create, plus),
182             send(M, modified, @off).
```

3.1.6 Finding parts

The methods below provide access to the various parts of the drawing tool. It makes it easier to remember how to access the parts and allows for changing the classnames without affecting too much code.

```
183   dialog(Draw, D) :<-
184             "Find the dialog of the tool"::
185             get(Draw, member, dialog, D).
186   canvas(Draw, C) :<-
187             "Find the drawing canvas"::
```

```
188           get(Draw, member, draw_canvas, C).

189   menu(Draw, C) :<-
190           "Find the icon menu"::
191           get(Draw, member, draw_menu, C).
```

3.1.7 Modes

PceDraw can operate in various modes. A mode defines what happens on a left-button-down event (ms_left_down). The various recognisers for left-button events are only sensative when the draw_canvas is in the appropriate modes.

→*mode* and →*proto* pass messages from the menu to the appropriate part of PceDraw (the canvas). As discussed above, it as ok for the dialog to send messages directly to the parts. Why is it not ok to do this from the menu? The answer is that the menu is defined in a different module of the system. It could be reusable in a different context (for example in a prototype editor), where the overall tool wants to implement mode switches differently. Note that through ←*frame* the menu has generic access to the tool it is part of.

```
192   mode(Draw, Mode:name, Cursor:cursor) :->
193           "Switch the mode"::
194           send(Draw?canvas, mode, Mode, Cursor).

195   proto(Draw, Proto:'graphical|link*') :->
196           "Switch the current prototype"::
197           send(Draw?canvas, proto, Proto).
```

3.1.8 Feedback

The method →*feedback* as defined here provides a general mechanism for any part of PceDraw to print a (short) feedback message:

```
send(MySelf?frame, feedback, string('%s: No such file', File))
```

NOTE: This mechanism should be exploited further in PCE itself by providing sensible defaults for feedback handling.

```
198   feedback(Draw, Str:string) :->
199           "Print feedback message in dialog"::
200           send(Draw?dialog?feedback_member, selection, Str).

201   about(_Draw) :->
202           "Print 'about' message"::
203           send(@display, inform, '%s\n\n%s\n%s\n%s\n',
204                'PceDraw version 1.1',
205                'By',
206                'Jan Wielemaker',
207                'E-mail: jan@swi.psy.uva.nl').
```

The →*help* method opens a view with the help-text. Currently, there are no provisions for PCE to find the help-file. Using the `library_directory/1` predicate should be considered a temporary solution.

The code below is dubious. In a larger application with various possibilities for getting help one should introduce a separate help system.

```
208   help(_Draw) :->
209           "Show window with help-text"::
210           (   library_directory(Dir),
211               concat(Dir, '/draw/draw.hlp', HelpText),
212               new(File, file(HelpText)),
213               send(File, exists)
214           ->  new(V, view('PceDraw: help')),
215               send(V, size, size(80, 40)),
216               new(D, dialog),
217               send(D, append, button(quit, message(V, free))),
218               send(D, below, V),
219               send(V, load, File),
220               (   send(File, access, write)
221               ->  send(V, editable, @on)
222               ;   send(V, editable, @off)
223               ),
224               send(V, open)
225           ;   send(@display, inform, 'Can''t find help file `draw.hlp'''')
226           ).
```

3.1.9 Quit

Quit PceDraw. This is rather simplistic. The code should both check for modifications in the prototype database and for the drawing. If one or both of them has changed a window indicating what has been modified should be displayed, allowing the user to save and/or quit PceDraw.

```
227   quit(Draw) :->
228           "Leave draw"::
229           get(Draw, canvas, Canvas),
230           (   get(Canvas, modified, @on)
231           ->  new(D, dialog),
232               send(D, transient_for, Draw),
233               send(D, append, label(message, 'Drawing has changed')),
234               send(D, append, button('Save & Quit',
235                                      message(D, return, save_and_quit))),
236               send(D, append, button(quit,
237                                      message(D, return, quit))),
238               send(D, append, button(cancel,
239                                      message(D, return, cancel))),
240               get(D, confirm_centered, Rval),
241               send(D, destroy),
242               (   Rval == save_and_quit
243               ->  send(Canvas, save),
```

```
244                     send(Draw, destroy)
245                 ;   Rval == quit
246                 ->  send(Draw, destroy)
247                 )
248             ;   (   send(@display, confirm, 'Quit PceDraw?')
249                 ->  send(Draw, destroy)
250                 ;   fail
251                 )
252             ).
253 :- pce_end_class.
```

3.2 Source file "canvas.pl"

```
1    /*  $Id: canvas.pl,v 1.12 1993/09/03 09:52:16 jan Exp $
2        Part of XPCE
3        Designed and implemented by Anjo Anjewierden and Jan Wielemaker
4        E-mail: jan@swi.psy.uva.nl
5        Copyright (C) 1992 University of Amsterdam. All rights reserved.
6    */

7    :- module(draw_canvas, []).

8    :- use_module(library(pce)).
9    :- use_module(align).
10   :- require([ chain_list/2
11              , concat/3
12              , concat_atom/2
13              , ignore/1
14              , shell/1
15              ]).
```

Class 'draw_canvas' defines the actual drawing area. Representing a collection of graphicals, the closest PCE class is 'picture'. In addition to pictures, class draw_canvas takes care of the current mode, the current prototype, the file (for loading and saving the image) and an editor for changing attributes of graphical objects.

For editing the drawing, two variables have been added: 'mode' and 'proto'. 'Mode' is an indication of the current mode. The various gestures defined in the file 'gesture' are only active in predefined modes. They can access the current mode with:

```
@event?receiver?window?mode
```

For modes that create objects, the variable 'proto' contains a prototype of the object to be created. Instances of the prototype are created using 'Object $\leftarrow$*clone*', except for links, which are instantiated by creating a connection from them.

The variables $\leftrightarrows$*file* and $\leftarrow$*modified* are used to implement $\rightarrow$*save* and $\rightarrow$*load*. [1]

The attribute_editor is a reference to an editor that allows the user to change the attributes of the graphicals in the selection. [2]

```
16   :- pce_begin_class(draw_canvas, picture).
17   resource(size, size, '500x500', 'Default size of drawing area').
18   variable(mode,             name,                          get,
19            "Current mode of operation").
20   variable(proto,            object*,                       both,
21            "Current create prototype (graphical/link)").
22   variable(file,             file*,                         get,
```

[1] Modified is a difficult issue. It should be set by all operations that change anything to the contents of the diagram. Maybe it is better to implement a modified variable at the level of window, or implement a message that allows the programmer to keep track of actions on the picture.

[2] Should we define the type of the attribute_editor to be 'draw_attribute_editor*' or rather 'object*' and just rely the attribute editor has the appropriate methods to facilitate the communication?

```
23              "Current save/load file").
24      variable(modified,          bool,                           get,
25              "Has the contents been modified?").
26      variable(auto_align_mode,   bool,                           both,
27              "Autoalign graphicals after create/resize").
28      variable(attribute_editor, draw_attribute_editor*,      both,
29              "Editor handling attributes").
```

3.2.1 Initialise

→*initialise* initialises the picture and custom slots that should not be `@nil`. It also attaches an event recogniser to the picture. Note that there are two ways to attach an event recogniser to a picture.

The first is to attach a recogniser using the 'Object →*recogniser*' method. In this case, the object is extended with an interceptor and the recogniser is attached to this interceptor. Recognisers attached to an interceptor are activated by the 'Graphical →*event*'.

The second is to define a method →*event*. This method may either decide to decode the events itself, or leave this to a recogniser. These approaches are used in the file shapes.pl to make shapes sensitive to user actions.

```
30      initialise(Canvas) :->
31              "Create a drawing canvas"::
32              send(Canvas, send_super, initialise, 'Canvas'),
33              send(Canvas, slot, modified, @off),
34              send(Canvas, auto_align_mode, @off),
35              send(Canvas, mode, select, @nil),
36              send(Canvas, recogniser, @draw_canvas_recogniser).
```

The recogniser itself is a reusable object (which implies other instances of draw_canvas can use the same instance of the recogniser). For this reason, it is declared using `pce_global/2`. The first time the recogniser reference is passed to PCE, the interface will trap an exception and create the object using the declaration in this file. This approach will delay the creation of the reusable object until it is really necessary and avoids conditions in the code (i.e. 'if object does not exist then create it' just before it is used). [3]

```
37      :- pce_global(@draw_canvas_recogniser,
38            new(handler_group(@draw_create_resize_gesture,
39                              @draw_create_line_gesture,
40                              @draw_create_path_gesture,
41                              @draw_create_text_recogniser,
42                              @draw_create_proto_recogniser,
43                              @draw_warp_select_gesture,
```

[3] I'm not sure whether or not it is better to a) Declare the global objects in gesture.pl and just refer to them here or b) Just declare the classes there and create instances here.

Both approaches have their advantages. The first approach guarantees maximal reuse. Actually there is only one instance of each gesture class and one may advocate it is better to use object-level programming to create this sole instance. PCE should offer

:- pce_begin_object(NewTerm). ... :- pce_end_object.

similar to defining classes.

```
44                              click_gesture(right, '', single,
45                                            message(@event?receiver?frame?menu,
46                                                    activate_select))))).
```

3.2.2 Unlink

The →*unlink* behaviour is called when an object is removed from the PCE object base, either initiated through 'Object →*free*', or through the garbage collector. 'Object →*unlink*' is responsible for unlinking the object from its environment. For example, when a window is unlinked it should inform X-windows; when a graphical is unlinked, it should inform its device. Removing an object entails the following steps:

1. Call →*unlink*
2. Reset all slots that have objects in them to `@nil`
3. Reclaim the memory

Like →*initialise*, →*unlink* should invoke the method of the super-class. Normally, it will first do its own part of the job and then starts the →*unlink* of the superclass.

```
47    unlink(Canvas) :->
48            (   get(Canvas, attribute_editor, Editor),
49                Editor \== @nil
50            ->  send(Editor, quit)
51            ;   true
52            ),
53            send(Canvas, send_super, unlink).
```

3.2.3 Modifications

```
54    modified(C) :->
55            send(C, slot, modified, @on).
```

3.2.4 Selection

Managing the selection. This is no different than for standard picture, except that we have to update the attribute-editor if it is attached.

```
56    selection(C, Sel:'graphical|chain*') :->
57            "Set the selection shape or chain"::
58            send(C, send_super, selection, Sel),
59            send(C, update_attribute_editor).

60    toggle_select(C, Shape:graphical) :->
61            "(Un)select a shape"::
62            send(Shape, toggle_selected),
63            send(C, update_attribute_editor).
```

3.2.5 Imports

Import a named X11 image (bitmap) file into the drawing. This code implements a simple modal dialog window that prompts for an image. The 'text_item →*type*' attribute describes the (PCE) type of the object requested. After the user has entered a name and type return or pressed the 'ok' button, PCE will try to convert the typed value into an PCE image object. See 'image ←*convert*' for the conversion rules.

```
64    import_image(C) :->
65            "Import an image at location (0,0)"::
66            new(D, dialog('Import Image')),
67            send(D, append, new(TI, text_item(image, ''))),
68            send(TI, type, image),
69            send(D, append, button(ok, message(D, return, TI?selection))),
70            send(D, append, button(cancel, message(D, return, @nil))),
71            send(D, default_button, ok),
72            get(D, confirm_centered, Image),
73            send(D, destroy),
74            Image \== @nil,
75            send(C, display, draw_bitmap(Image)).
```

Import another PCE frame as a bitmap. The user may select a frame to be imported by clicking on it. There are two ways to implement this. The first is to grab the mouse-focus, wait for a left-click and then locate the frame on which the user clicked. The second is to use PCE's 'inspect-handlers'. If an event happens that satisfies one of the 'display ←*inspect_handlers*', PCE will locate the graphical on which the event occurred and execute the message of the inspect-handler with `@arg1` bound to the graphical on which the event occurred. This mechanism is exploited by the 'Inspector' and 'Visual Hierarchy' tools of the manual.

```
76    import_frame(C) :->
77            "Import image of a frame"::
78            get(C, display, Display),
79            new(D, dialog('Import Frame')),
80            send(D, append,
81                 label(prompt, 'Please left-click inside PCE frame to import')),
82            send(D, append, button(cancel, message(D, return, @nil))),
83            send(Display, inspect_handler,
84                 new(G, handler(ms_left_up, message(D, return, @arg1?frame)))),
85            get(D, confirm, Frame),
86            send(Display?inspect_handlers, delete, G),
87            send(D, destroy),
88            Frame \== @nil,
89            send(C, display, draw_bitmap(Frame?image)).
```

3.2.6 Edit

Selection-edit operations. Most of them are rather trivial. Note the use of 'Chain →*for_all*' to perform operations on all members of the selection. This method is a lot faster then transferring the selection to a Prolog list and than operating on it:

```
        get(Canvas, selection, Selection),
        chain_list(Selection, List),
        forall(member(Gr, List), send(Gr, free)).
```

The 'Chain →*for_all*' operation first makes an array of objects in the chain, than invokes the message consequtively on each member of the list. Before sending the message, it validates the object still exists. This makes the method safe for cases were destroying one object destroyes related objects that may be in the chain too. Connections are an example: destroying a graphical destroys all its connections and therefore leaves 'dangling' references.

One could generalise from the code below by introducing a method →*for_selection*: message, but the advantages are very small.

```
90  edit(Canvas, Msg, Grs:'[graphical|chain]') :->
91          "Perform operation on graphicals or selection"::
92          default(Grs, Canvas?selection, G0),
93          (   send(G0, instance_of, chain)
94          ->  send(G0, for_all, Msg)
95          ;   send(Msg, forward, G0)
96          ),
97          send(Canvas, modified).

98  expose_selection(Canvas, Grs:'[graphical|chain]') :->
99          "Expose selected graphicals"::
100         send(Canvas, edit, message(@arg1, expose), Grs).

101 hide_selection(Canvas, Grs:'[graphical|chain]') :->
102         "Hide selected graphicals"::
103         send(Canvas, edit, message(@arg1, hide), Grs).

104 cut_selection(Canvas, Grs:'[graphical|chain]') :->
105         "Erase all members of the selection"::
106         send(Canvas, edit, message(@arg1, free), Grs).
```

3.2.7 Alignment

```
107 align_with_selection(Canvas, Gr:graphical) :->
108         "Align graphical (with selection)"::
109         (   get(Canvas, selection, G0),
110             send(G0, delete_all, Gr),
111             \+ send(G0, empty)
112         ->  true
113         ;   get(Canvas?graphicals, copy, G0),
114             send(G0, delete_all, Gr)
115         ),
116         get(G0, find_all, not(message(@arg1, instance_of, line)), G1),
117         chain_list(G1, L1),
118         align_graphical(Gr, L1).

119 align_selection(Canvas) :->
```

```
120         "Align all elements of the selection"::
121         send(Canvas, edit, message(Canvas, align_graphical, @arg1)).

122 align_graphical(Canvas, Gr:graphical) :->
123         "Align a single graphical"::
124         get(Canvas?graphicals, find_all,
125             not(message(@arg1, instance_of, line)), G0),
126         send(G0, delete_all, Gr),
127         chain_list(G0, L0),
128         auto_adjust(resize, Gr, L0),
129         align_graphical(Gr, L0).

130 auto_align(Canvas, Gr:graphical, How:{create,resize,move}) :->
131         "Align graphical if auto_align_mode is @on"::
132         (   get(Canvas, auto_align_mode, @on)
133         ->  ignore(auto_align(Canvas, Gr, How))
134         ;   true
135         ).

136 auto_align(Canvas, Gr, How) :-
137         get(Canvas?graphicals, find_all,
138             not(message(@arg1, instance_of, line)), G0),
139         send(G0, delete_all, Gr),
140         chain_list(G0, L0),
141         auto_adjust(How, Gr, L0),
142         align_graphical(Gr, L0).

143 auto_adjust(How, Gr, L0) :-
144         (How == create ; How == resize),
145         \+ send(Gr, instance_of, text),
146         adjust_graphical(Gr, L0), !.
147 auto_adjust(_, _, _).
```

The method below duplicates the selection and displays the duplicate at an optionally specified offset. There are various difficult operations in this predicate. The 'if-then-else' illustrates how default arguments are handled inside a method.

Next, the selection, which is a chain with the selected shapes, is cloned. 'Object $\leftarrow$*clone*' creates a recursive clone. Note that the selection as a whole is cloned rather than each member of it seperately. This guarantees proper kloning of relations inside the selection (such as connections). [4]

Finally $\rightarrow$*done* is sent to the clone of the selection chain. This indicates PCE that Prolog is no longer interrested in the object and that, if there are no references to it, it may be removed. Using 'Object $\rightarrow$*done*' is generally advocated over using $\rightarrow$*free* after Prolog has finished with the result of a get operation. Consider the following cases:

```
get(Graphical, position, Pos),
...
send(Pos, free).
```

[4] Connections to objects outside the selection are not handled properly. Kloning objects has to be based on semantics attached to slot-relations rather than classes.

and

```
        get(Graphical, area, Area),
        ...
        send(Area, free).
```

In the first example, 'Pos' is a point created by the method, but not referred to by the 'Graphical'. Using →*free* is correct. In the second case however the method ←*area* returns the ←*area* attribute of 'Graphical' and destroying this would make 'Graphical' an inconsistent object. Using →*done*, the point will be removed in the first example, but the area will remain in the second.

```
148  duplicate_selection(Canvas, Offset:[point]) :->
149          "Duplicate the selection"::
150          default(Offset, point(10, 10), Off),
151          get(Canvas?selection, clone, Duplicate),
152          send(Duplicate, for_all,
153               block(message(Canvas, display, @arg1),
154                     message(@arg1, relative_move, Off))),
155          clean_duplicates(Duplicate),
156          send(Canvas, selection, Duplicate),
157          send(Duplicate, done),
158          send(Canvas, modified).
```

The method 'object ←*clone*' makes a recursive copy of an object. If an object with connection is cloned the connections as well as the 'other side' of the connections will be cloned as well. This predicate removes all graphical objects that are related to the duplicated object but not displayed.

```
159  clean_duplicates(Chain) :-
160          new(Done, hash_table),
161          send(Chain, for_some,
162               message(@prolog, clean_duplicate_connections, @arg1, Done)),
163          send(Done, free).
164  clean_duplicate_connections(Gr, Done) :-
165          get(Done, member, Gr, @on), !.
166  clean_duplicate_connections(Gr, _) :-
167          \+ get(Gr, window, _), !,
168          send(Gr, destroy).
169  clean_duplicate_connections(Gr, Done) :-
170          send(Done, append, Gr, @on),
171          get(Gr, connections, AllConnections),
172          send(AllConnections, for_all,
173               message(@prolog, clean_duplicate_connections,
174                       ?(@arg1, opposite, Gr), Done)).
```

Start the attribute editor on the current selection. The first time, we need to create the editor. If the user hits 'quit' on the button of the editor, the editor is just removed from the display using 'Frame →*show*: `@off`' and this function can make it visible again using

→*show*: `@on`. This approach has several advantages. First of all, it is a lot faster and second, the attribute editor will be at the same location on the display as were the user left it last time.

See also →*unlink* in this class and 'draw_attribute_editor →*quit*'.

```
175   edit_selection(Canvas) :->
176           "Start attribute editor on selection"::
177           get(Canvas, attribute_editor, Editor),
178           (    Editor == @nil
179           ->   send(Canvas, slot, attribute_editor,
180                     draw_attribute_editor(Canvas)),
181                send(Canvas?attribute_editor, open)
182           ;    send(Canvas?attribute_editor, show, @on),
183                send(Canvas?attribute_editor, expose)
184           ),
185           send(Canvas?attribute_editor, client, Canvas?selection).
```

Update the setting of the attribute editor because either the selection has changed, or the attributes of members of the selection has changed. [5]

```
186   update_attribute_editor(Canvas) :->
187           "Update values in attribute editor"::
188           get(Canvas, attribute_editor, Editor),
189           (   Editor \== @nil
190           ->  send(Editor, client, Canvas?selection)
191           ;   true
192           ).

193   clear(Canvas, Confirm:[bool]) :->
194           "Clear and reset ⇆file attribute"::
195           (   Confirm == @on,
196               \+ send(Canvas?graphicals, empty)
197           ->  send(@display, confirm, 'Clear drawing?')
198           ;   true
199           ),
200           send(Canvas, send_super, clear),
201           send(Canvas, file, @nil),
202           send(Canvas, slot, modified, @off),
203           send(Canvas, update_attribute_editor).
```

3.2.8 Load/save

Saving and loading is currently performed by saving the PCE objects using PCE's binary saving algorithm. This approach has several advantages and disadvantages. The advantages:

- Using 'Object →*save_in_file*', applications whose database consists of a collection of PCE objects can easily save their data.

[5] The move and resize gestures should invoke this behaviour too.

- The PCE built-in loading and saving is fast.

The disadvantages

- Binary format. Currently no provisions for byte-order differences.
- It is difficult to control what exactly will be stored on file. See also the discussion on kloning with →*duplicate*.
- Significant changes to the representation of PCE-classes make reloading impossible. This is notably a problem for loading and storing graphical information.

An alternative is to write an application-specific save/load that is more robust against changes to PCE, but may be slow. This kind of saved version can be used to convert to later versions of PCE.

→*save_as* requests a filename and then invokes 'Object →*save*'. The filename is requested via `@finder`, an instance of the user-defined class 'finder', defined in the PCE library file 'find_file.pl'. Linking the library is declared in the file draw.pl.

This addresses the general case of asking for information using dialog-boxes. In earlier PCE applications it was normal to build a dialog-box, display it, read the information and destroy it again.

For the file-finder, the reusable object `@finder` is created using `pce_global/2` construct. Once created, `@finder` is mapped on and removed-from the display using 'Frame →*show*: `@on`/`@off`'. This approach is fast and allow us to remember status (such as the selected directory) from the last time the finder was used.

```
204   save_as(Canvas) :->
205           "Save in user-specified file"::
206           get(@finder, file, @off, '.pd', File),
207           send(Canvas, save, File).
```

Actual saving to file. The toplevel-object saved is a sheet. This way we can easily add new attributes without affecting compatibility. Future versions will probably also save the name of the file on which the prototypes were stored, so we can reload the corresponding prototypes.

```
208   save(Canvas, File:[file]) :->
209           "Save canvas in named file"::
210           (   File == @default
211           ->  get(Canvas, file, SaveFile),
212               (   SaveFile == @nil
213               ->  send(@display, inform, 'No current file'),
214                   fail
215               ;   true
216               )
217           ;   send(Canvas, file, File),
218               SaveFile = File
219           ),
220           send(SaveFile, backup),
221           new(Sheet, sheet(attribute(graphicals, Canvas?graphicals))),
222           send(Sheet, save_in_file, SaveFile),
```

```
223           send(Canvas?frame, feedback,
224                string('Saved %s', SaveFile?base_name)),
225           send(Sheet, free).

226   load_from(Canvas) :->
227           "Load from user-specified file"::
228           get(@finder, file, @on, '.pd', File),
229           send(Canvas, load, File, @on).

230   import(Canvas) :->
231           "Add contents of user-requested file"::
232           get(@finder, file, @on, '.pd', File),
233           send(Canvas, load, File, @off).
```

Load specified file and set the file attribute. The PCE object is loaded from the file using the 'File ←*object*: method. [6]

```
234   load(Canvas, File:file, Clear:[bool]) :->
235           "Load from named file and [clear]"::
236           (    Clear == @on
237           ->   send(Canvas, clear, @on)
238           ;    true
239           ),
240           get(File, object, Sheet),
241           send(Sheet?graphicals, for_all,
242                block(message(Canvas, display, @arg1),
243                      message(@arg1, selected, @off))),
244           send(Canvas, file, File),
245           send(Canvas?frame, feedback, string('Loaded %s', File?base_name)),
246           send(Sheet, done).

247   file(Canvas, File:file*) :->
248           "Set file attribute"::
249           send(Canvas, slot, file, File),
250           (   File \== @nil
251           ->  send(Canvas?frame, label,
252                    string('PceDraw: %s', File?name),
253                    string('PceDraw: %s', File?base_name))
254           ;   send(Canvas?frame, label, 'PceDraw')
255           ).
```

3.2.9 Postscript

Create a PostScript description of the contents of the picture. [7]

[6] Currently PCE provides no way for the programmer to specify what should happen on file errors. This will be fixed.

[7] This should ask for options such as landscape and scaling factor, which can be applied to the Graphical ←*postscript* method.

```
256  postscript(Canvas) :->
257          "Write PostScript to default file"::
258          get(Canvas, default_psfile, File),
259          send(Canvas, generate_postscript, File).

260  postscript_as(Canvas) :->
261          "Write PostScript to file"::
262          get(Canvas, default_psfile, DefFile),
263          get(@finder, file, @off, '.ps', @default, DefFile, FileName),
264          send(Canvas, generate_postscript, FileName).

265  generate_postscript(Canvas, PsFile:file) :->
266          "Write PostScript to named file"::
267          send(PsFile, open, write),
268          send(PsFile, append, Canvas?postscript),
269          send(PsFile, close),
270          send(Canvas?frame, feedback,
271               string('Written PostScript to `%s'', PsFile?base_name)).

272  default_psfile(Canvas, DefName) :<-
273          "Default name for PostScript file"::
274          (   get(Canvas, file, File),
275              File \== @nil,
276              get(File, name, Name),
277              concat(Base, '.pd', Name)
278          ->  concat(Base, '.ps', DefName)
279          ;   DefName = 'scratch.ps'
280          ).
```

Print the image to the default printer. Also this method should be extended by resquesting additional parameters from the user.

```
281  print(Canvas) :->
282          "Send to default printer"::
283          default_printer(Printer),
284          temp_file(File),
285          new(PsFile, file(File)),
286          send(PsFile, open, write),
287          send(PsFile, append, Canvas?postscript),
288          send(PsFile, append, 'showpage\n'),
289          send(PsFile, close),
290          concat_atom(['lpr -P', Printer, ' ', File], Cmd),
291          shell(Cmd),
292          send(PsFile, remove),
293          send(PsFile, done),
294          send(Canvas?frame, feedback,
295               string('Sent to printer `%s'', Printer)).

296  default_printer(Printer) :-
297          get(@pce, environment_variable, 'PRINTER', Printer), !.
298  default_printer(postscript).

299  temp_file(Name) :-
```

```
300             get(@pce, pid, Pid),
301             concat('/tmp/xpce_', Pid, Name).
```

3.2.10 Modes

Switch the mode of the editor. The mode determines which gestures are active (see 'gesture.pl') and therefore what happens on some event. For each mode, a cursor is defined to indicate the mode to the user.

```
302   mode(Canvas, Mode:name, Cursor:cursor*) :->
303             "Set the mode of the canvas"::
304             send(Canvas, cursor, Cursor),
305             send(Canvas, slot, mode, Mode),
306             send(Canvas, keyboard_focus, @nil),
307             send(Canvas, selection, @nil).

308   :- pce_end_class.
```

3.3 Source file "shapes.pl"

```
1   /*  $Id: shapes.pl,v 1.8 1993/09/03 09:52:19 jan Exp $
2       Part of XPCE
3       Designed and implemented by Anjo Anjewierden and Jan Wielemaker
4       E-mail: jan@swi.psy.uva.nl
5       Copyright (C) 1992 University of Amsterdam. All rights reserved.
6   */
7   :- module(draw_shapes, []).
8   :- use_module(library(pce)).
9   :- require([ memberchk/2
10             ]).
```

This module defines the various shapes that can be used to construct the diagram. Most of the shapes are very close the PCE's drawing primitives. Two things have to be added for each of them: handles for connecting lines (connections) and event-handling.

Both things can be added both at the class and at the instance level. I decided to add them at the class level. As there are normally multiple instances of the classe, this approach reduces memory cost. A more important issue is kloning and saving. These operations work recursively and therefore would clone and save the object-level extensions. For saving, this has two disadvantages. The saved files would get bigger and, more important, the gestures -defining the UI of the tool- would be saved too. This leads to a bad separation of UI and the actual data manipulated.

3.3.1 Common

The various shapes are subclasses of corresponding PCE graphicals. Each of them has to be expanded with $\rightarrow$*geometry* and $\rightarrow$*attribute*. We define predicates to implement these methods and than just refer to these predicates.

```
11  geometry(Gr, X, Y, W, H) :-
12          send(Gr, send_super, geometry, X, Y, W, H),
13          modified(Gr).
14  attribute(Gr, Att, Val) :-
15          send(Gr, has_attribute, Att),
16          send(Gr, Att, Val),
17          modified(Gr).
18  modified(Gr) :-
19          get(Gr, window, Window), Window \== @nil,
20          send(Window, modified),
21          get(Gr, selected, @on),
22          send(Window, update_attribute_editor), !.
23  modified(_).
```

3.3.2 Box

Box is the most prototypical example of a graphical. Boxes in PceDraw have handles for connections in the middle of each side. Event handling is realised by the reusable object `@draw_resizable_shape_recogniser`. Note that the reference to the box need not be provided. →*event* is invoked from 'Event →*post*' and the receiver slot of the event is a reference to the box.

Note that draw_box is a subclass of box rather than an extension of class box. Extending class box might conflict with the consistency of PCE itself or other applications running in the same PCE process (never assume you are alone in the world).

The `handle/4` construct attaches a handle with specified ⇆*kind* and ⇆*name* at the specified position. The handle is attached to the class (see 'Class →*handle*') rather than to the instances (see 'Graphical →*handle*').

```
24    :- pce_begin_class(draw_box, box).
25    handle(w/2, 0,   link, north).
26    handle(w/2, h,   link, south).
27    handle(0,   h/2, link, west).
28    handle(w,   h/2, link, east).
29    event(_Box, Ev:event) :->
30            send(@draw_resizable_shape_recogniser, event, Ev).
31    geometry(B, X:[int], Y:[int], W:[int], H:[int]) :->
32            "Forward change to device"::
33            geometry(B, X, Y, W, H).
```

The →*has_attribute* method tests whether the specified attribute of the shape can be set. This is a bit of a hack. A better solution would have been to test whether the graphical has the specified method. Unfortunately att graphicals have method ⇆*pen*, but for some of them, changing the value has not effect. The same applies for some other attributes. This should be changed in PCE.

```
34    has_attribute(_B, Att:name) :->
35            "Test if object has attribute"::
36            memberchk(Att, [ pen, texture, colour, fill_pattern, radius
37                           , x, y, width, height]).
38    attribute(B, Att:name, Val:any) :->
39            attribute(B, Att, Val).
40    attribute(B, Att:name, Val) :<-
41            get(B, Att, Val).
42    :- pce_end_class.
```

3.3.3 Ellipse

```
43    :- pce_begin_class(draw_ellipse, ellipse).
44    handle(w/2, 0,   link, north).
45    handle(w/2, h,   link, south).
```

```
46    handle(0,   h/2, link, west).
47    handle(w,   h/2, link, east).
48    event(_Ellipse, Ev:event) :->
49            send(@draw_resizable_shape_recogniser, event, Ev).
50    geometry(E, X:[int], Y:[int], W:[int], H:[int]) :->
51            "Forward change to device"::
52            geometry(E, X, Y, W, H).
53    has_attribute(_E, Att:name) :->
54            "Test if object has attribute"::
55            memberchk(Att, [ pen, texture, colour, fill_pattern
56                           , x, y, width, height]).
57    attribute(E, Att:name, Val:any) :->
58            attribute(E, Att, Val).
59    attribute(E, Att:name, Val) :<-
60            get(E, Att, Val).
61    :- pce_end_class.
```

3.3.4 Text

```
62    :- pce_begin_class(draw_text, text).
63    handle(w/2, 0,   link, north).
64    handle(w/2, h,   link, south).
65    handle(0,   h/2, link, west).
66    handle(w,   h/2, link, east).
67    initialise(T, String:string, Format:[name], Font:[font]) :->
68            default(Format, center,                      Fmt),
69            default(Font,   font(helvetica, roman, 14),  Fnt),
70            send(T, send_super, initialise, String, Fmt, Fnt).
```

This method illustrates another way to define event-handling at the class level: just analyse the type of the event and perform the necessary action. For complex event-sequences gestures are to be preferred as they take care of many of the difficulties such as managing the focus, cursor and state-variables needed to parse the event sequence. For simple events all this is not necessary, so we might just as well parse them within the $\rightarrow$*event* method. [8] [9]

```
71    event(Text, Ev:event) :->
72            (   get(Text, show_caret, @on),
73                get(Ev, id, Id),
74                event(Id, Text)
75            ->  true
76            ;   send(Ev, is_a, keyboard),
```

[8] Events types will be changed shortly. Having to refer to ESC as '27' is not the right way to program. I'm not yet sure on the details.

[9] PCE will probably provided higher-level primitives such as a special subclass of recogniser to deal with most of the details of this method.

```
77              get(Text, show_caret, @on)
78          ->  send(Text, typed, Ev?id),
79              modified(Text)
80          ;   send(Ev, is_a, obtain_keyboard_focus)
81          ->  send(Text, show_caret, @on)
82          ;   send(Ev, is_a, release_keyboard_focus)
83          ->  (   get(Text?string, size, 0),
84                  send(Text?device, instance_of, draw_canvas) % HACK
85              ->  send(Text, free)
86              ;   send(Text, show_caret, @off)
87              )
88          ;   send(@draw_text_recogniser, event, Ev)
89          ).
90  event(27, Text) :-                                  % ESC
91          send(Text?window, keyboard_focus, @nil).
92  event(9, Text) :-                                   % TAB
93          send(Text?device, advance, Text).
```

Indicate to the device that this graphical is willing to accept the keyboard focus. It is interpreted by the 'Device →*advance*' method to set the keyboard focus to the next object that wants to accept keystrokes. [10]

```
94  '_wants_keyboard_focus'(_T) :->
95          "Indicate device I'm sensitive for typing"::
96          true.
97  geometry(T, X:[int], Y:[int], W:[int], H:[int]) :->
98          "Forward change to device"::
99          geometry(T, X, Y, W, H).
100 has_attribute(_T, Att:name) :->
101         "Test if object has attribute"::
102         memberchk(Att, [font, colour, transparent, x, y, width, height]).
103 attribute(T, Att:name, Val:any) :->
104         attribute(T, Att, Val).
105 attribute(T, Att:name, Val) :<-
106         get(T, Att, Val).
107 :- pce_end_class.
```

3.3.5 Line

```
108 :- pce_begin_class(draw_line, line).
109 handle(w/2, h/2, link, center).
110 handle(0,   0,   link, start).
111 handle(w,   h,   link, end).
112 event(_L, Ev:event) :->
113         send(@draw_line_recogniser, event, Ev).
```

[10] This mechanism needs some redesign and documentation.

```
114  geometry(L, X:[int], Y:[int], W:[int], H:[int]) :->
115          "Forward change to device"::
116          geometry(L, X, Y, W, H).
117  has_attribute(_L, Att:name) :->
118          "Test if object has attribute"::
119          memberchk(Att, [ pen, texture, arrows, colour, x, y, width, height]).
120  attribute(L, Att:name, Val:any) :->
121          attribute(L, Att, Val).
122  attribute(L, Att:name, Val) :<-
123          get(L, Att, Val).
124  :- pce_end_class.
```

3.3.6 Path

```
125  :- pce_begin_class(draw_path, path).
126  event(_L, Ev:event) :->
127          send(@draw_path_recogniser, event, Ev).
128  geometry(L, X:[int], Y:[int], W:[int], H:[int]) :->
129          "Forward change to device"::
130          geometry(L, X, Y, W, H).
131  has_attribute(_L, Att:name) :->
132          "Test if object has attribute"::
133          memberchk(Att,
134                    [ pen, texture, colour, fill_pattern, arrows
135                    , closed, interpolation
136                    , x, y, width, height
137                    ]).
138  attribute(L, Att:name, Val:any) :->
139          attribute(L, Att, Val).
140  attribute(L, Att:name, Val) :<-
141          get(L, Att, Val).
142  interpolation(L, N:int) :->
143          (   N == 0
144          ->  send(L, kind, poly)
145          ;   send(L, intervals, N),
146              send(L, kind, smooth)
147          ).
148  interpolation(L, N:int) :<-
149          (   get(L, kind, poly)
150          ->  N = 0
151          ;   get(L, intervals, N)
152          ).
153  :- pce_end_class.
```

3.3.7 Connections

A connection is a line between two handles on two different graphical objects. See clas handle, graphical and connection for details.

```
154  :- pce_begin_class(draw_connection, connection).
155  handle(w/2, h/2, link, center).
156  event(_C, Ev:event) :->
157          send(@draw_connection_recogniser, event, Ev).
158  has_attribute(_C, Att:name) :->
159          "Test if object has attribute"::
160          memberchk(Att, [ pen, texture, arrows, colour, x, y, width, height]).
161  attribute(C, Att:name, Val:any) :->
162          attribute(C, Att, Val).
163  attribute(C, Att:name, Val) :<-
164          get(C, Att, Val).
165  :- pce_end_class.
```

3.3.8 Bitmap

Bitmaps are used to import arbitrary images into a drawing.

```
166  :- pce_begin_class(draw_bitmap, bitmap).
167  handle(w/2, 0,   link, north).
168  handle(w/2, h,   link, south).
169  handle(0,   h/2, link, west).
170  handle(w,   h/2, link, east).
171  event(_B, Ev:event) :->
172          send(@draw_bitmap_recogniser, event, Ev).
173  has_attribute(_C, Att:name) :->
174          "Test if object has attribute"::
175          memberchk(Att, [colour, x, y]).
176  attribute(C, Att:name, Val:any) :->
177          attribute(C, Att, Val).
178  attribute(C, Att:name, Val) :<-
179          get(C, Att, Val).
180  :- pce_end_class.
```

3.3.9 Compounds

Compounds are used to realise (user-defined) prototypes that consist of more than one shape. Compound is a subclass of the PCE class 'device', that manages a collection of graphicals. In addition to devices, compounds define distribution of keyboard events to one of the text objects inside it and resizing the device.

```
:- pce_begin_class(draw_compound, device).

handle(w/2, 0,   link, north).
handle(w/2, h,   link, south).
handle(0,   h/2, link, west).
handle(w,   h/2, link, east).
```

Resizing compounds. PCE's primitives do not provide for that. However, any attempt to change to the area of the graphical via 'Graphical →*set*', 'Graphical →*x*', 'Graphical →*area*', etc. will invoke 'Graphical →*geometry*' to do the actual moving/resizing.

By default, devices will move themselve, but not resize their contents. In the method below, we first resize the contents of the device in a way very similar to resizing the selection as described in the file 'gesture.pl' and than invoke the super-behaviour to realise the move. Never try to do the move yourself: the superclass might do (and in the case of device does) additional things to changing the coordinates.

```
geometry(C, X:[int], Y:[int], W:[int], H:[int]) :->
        "Resize compound graphical"::
        resize_factor(W, C, width,  Xfactor),
        resize_factor(H, C, height, Yfactor),
        (   (Xfactor \== 1 ; Yfactor \== 1)
        ->  get(C?area, position, Origin),
            send(Origin, minus, C?position),
            send(C?graphicals, for_all,
                 message(@arg1, resize, Xfactor, Yfactor, Origin)),
            send(Origin, done)
        ;   true
        ),
        geometry(C, X, Y, W, H).

resize_factor(@default, _, _, 1) :- !.
resize_factor(W1, C, S, F) :-
        get(C, S, W0),
        F is W1 / W0.
```

The method below sets the string of all text objects. Used by the icon manager (menu.pl) and the create gesture (gesture.pl) to set the strings to 'T', resp '' (nothing).

```
string(C, Str:string) :->
        "Set string of all texts"::
        send(C?graphicals, for_all,
             if(message(@arg1, has_send_method, string),
                message(@arg1, string, Str))).

event(_C, Ev:event) :->
        send(@draw_compound_recogniser, event, Ev).
```

The method below is called from the compound_recogniser on a ms_left_down if the editor is in text_edit mode. If the down is in the area of a text, the caret is positioned as close as possible to the location of the down. Otherwise it is placed on the first text object of the compound.

First all text objects are found. Next, it tries to find the first text that overlaps with the position of the down-event. If this succeeds, the caret is placed as close as possible to the down location. Otherwise the caret is located at the end of the first text object of the compound.

```
start_text(C, Ev:[event]) :->
        "Enter typing mode"::
        get(C?graphicals, find_all,
            message(@arg1, instance_of, text), Texts),
        (   Ev \== @default,
            get(Texts, find, message(Ev, inside, @arg1), Pointed)
        ->  send(Pointed, caret, ?(Pointed, pointed,
                                   ?(Ev, position, Pointed))),
            send(C?window, keyboard_focus, Pointed)
        ;   get(Texts, head, First)
        ->  send(First, caret, @default),
            send(C?window, keyboard_focus, First)
        ),
        send(Texts, done).
```

The code below illustrates another reason for not communicating the attribute setting using $\rightarrow x$, $\rightarrow pen$, etc. For a compound, the x, y, width and height attributes should hold for the compound as a whole, while the other attributes should only hold for the parts.

```
geometry_selector(x).
geometry_selector(y).
geometry_selector(width).
geometry_selector(height).

has_attribute(C, Att:name) :->
        "Test if object has attribute"::
        (   geometry_selector(Att)
        ->  true
        ;   get(C?graphicals, find, message(@arg1, has_attribute, Att), _)
        ).

attribute(C, Att:name, Val:any) :->
        (   geometry_selector(Att)
        ->  send(C, Att, Val)
        ;   send(C?graphicals, for_some,
                 message(@arg1, attribute, Att, Val))
        ).

attribute(C, Att:name, Val) :<-
        (   geometry_selector(Att)
        ->  get(C, Att, Val)
        ;   get(C?graphicals, find, message(@arg1, has_attribute, Att), Shape),
            get(Shape, Att, Val)
        ).

:- pce_end_class.
```

3.4 Source file "gesture.pl"

```
1     /*  $Id: gesture.pl,v 1.16 1993/09/29 09:28:38 jan Exp $
2         Part of XPCE
3         Designed and implemented by Anjo Anjewierden and Jan Wielemaker
4         E-mail: jan@swi.psy.uva.nl
5         Copyright (C) 1992 University of Amsterdam. All rights reserved.
6     */
7     :- module(draw_gesture, []).
8     :- use_module(library(pce)).
9     :- require([ between/3
10                , concat/3
11                , send_list/3
12                ]).
```

This module defines event handling for the shapes. Event handling for dialog_items is predefined because the UI of dialog_items is standardised. Event handling for general purpose graphicals can be specified by defining the method 'Graphical →*event*'.

The default behaviour of →*event* (defined at the level of class graphical) is to look up the 'recognisers' slot of the attached interceptor (see 'Object →*recogniser*') and test if any of the attached interceptor is prepared to accept the event.

This implies there are three ways to define event parsing for graphical objects:

1. Attach a recogniser the object.
2. Write an →*event* method that parses the events.
3. Write an →*event* method that forwards the event to recognisers.

For PceDraw we chose the latter approach for shapes. See also the file canvas.pl. Provided the recognisers do not directly refer to the object for which they handle events as in

```
send(B, recogniser, click_gesture(left, '', single,
                                  message(B, inverted, @on)))
```

but, refer indirectly as in

```
send(B, recogniser, click_gesture(left, '', single,
                                  message(@receiver, inverted,
                                          @on)))
```

recognisers can be attached to any number of graphical objects. This file defines generic recognisers that are used by 'Shape →*event*'.

3.4.1 Recogniser objects

Below are the declarations of the various recognisers. Note that using `pce_global/2`, the actual creation of the recogniser is delayed to the first time an event occurs on an object that uses a specific recogniser.

```
                        /* Create shapes */
:- pce_global(@draw_create_resize_gesture,
              new(draw_create_resize_gesture)).
:- pce_global(@draw_create_line_gesture,
              new(draw_create_line_gesture)).
:- pce_global(@draw_create_path_gesture,
              new(draw_create_path_gesture)).
:- pce_global(@draw_connect_gesture,
              new(handler_group(new(draw_connect_gesture),
                                new(draw_connect_create_gesture)))).
                        /* Select shapes */
:- pce_global(@draw_shape_select_recogniser,
              make_draw_shape_select_recogniser).
:- pce_global(@draw_warp_select_gesture,
              new(draw_warp_select_gesture)).
                        /* Move/Resize shapes */
:- pce_global(@draw_move_outline_gesture,
              new(handler_group(new(draw_move_selection_gesture),
                                new(draw_move_gesture)))).
:- pce_global(@draw_resize_gesture,
              new(handler_group(new(draw_resize_selection_gesture),
                                new(draw_resize_gesture)))).
                        /* Combined shape recognisers */
:- pce_global(@draw_resizable_shape_recogniser,
              new(handler_group(@draw_shape_select_recogniser,
                                @draw_resize_gesture,
                                @draw_move_outline_gesture,
                                @draw_connect_gesture,
                                @draw_shape_popup_gesture))).
:- pce_global(@draw_text_recogniser,
              new(handler_group(@draw_shape_select_recogniser,
                                @draw_edit_text_recogniser,
                                new(draw_resize_selection_gesture),
                                @draw_move_outline_gesture,
                                @draw_connect_gesture,
                                @draw_shape_popup_gesture))).
:- pce_global(@draw_compound_recogniser,
              new(handler_group(@draw_resizable_shape_recogniser,
                                @draw_compound_draw_text_recogniser))).
:- pce_global(@draw_connection_recogniser,
              new(handler_group(@draw_shape_select_recogniser,
                                @draw_connect_gesture,
                                @draw_shape_popup_gesture))).
:- pce_global(@draw_bitmap_recogniser,
              new(handler_group(@draw_shape_select_recogniser,
                                @draw_move_outline_gesture,
                                @draw_connect_gesture,
                                @draw_shape_popup_gesture))).
:- pce_global(@draw_line_recogniser,
```

```
62                  new(handler_group(@draw_shape_select_recogniser,
63                                    @draw_connect_gesture,
64                                    @draw_shape_popup_gesture,
65                                    new(draw_change_line_gesture),
66                                    new(draw_move_selection_gesture),
67                                    new(move_gesture)))).
68      :- pce_global(@draw_path_recogniser,
69                  new(handler_group(@draw_shape_select_recogniser,
70                                    @draw_shape_popup_gesture,
71                                    new(draw_modify_path_gesture),
72                                    @draw_edit_path_gesture,
73                                    @draw_resize_gesture,
74                                    @draw_move_outline_gesture,
75                                    new(move_gesture)))).
```

3.4.2 Select

When in select mode, left-click on an object makes it the selection, shift-left-click adds or deletes it to/from the selection and left-dragging indicates an area in which all objects should be selected.

Clicking on an object is to be defined at the level of the object itself, where the drag version is to be defined at the level of the canvas. This is not very elegant as it implies we have to create two recognisers; one for the shapes and one for the canvas. The alternative would be one recogniser at the level of the canvas and find the object below the mouse on a click. It is difficult to say which of the two approaches is better.

The recogniser for shapes is defined below. It consists of a handler_group with two click_gestures. This implementation is far simpler than defining a new class. Note the definition of the obtainers before defining the gestures themselves. This method employs reusability of object and is easier to read.

```
76      make_draw_shape_select_recogniser(G) :-
77              new(Shape, @event?receiver),
78              new(Canvas, Shape?window),
79              new(SelectMode, Canvas?mode == select),
80              new(G, handler_group(click_gesture(left, '', single,
81                                                 message(Canvas, selection,
82                                                         Shape),
83                                                 SelectMode),
84                                   click_gesture(left, s, single,
85                                                 message(Canvas, toggle_select,
86                                                         Shape),
87                                                 SelectMode))).
```

The 'warp_gesture' allows the user to indicate an area by dragging a button and then selects all objects inside the indicated area. It is a rather typical example of a gesture definition. The `resource/3` declarations define the X-resources that apply: the button that activates the gesture, the modifiers required (shift, control, meta) and the cursor that indicates the gesture is active. These resource values are handled by the super-class gesture.

The variable 'outline' keeps track of the box that is used to indicate the area. It can be stored here, as only one gesture can be active at a time.

```
88    :- pce_begin_class(draw_warp_select_gesture, gesture).
89    resource(button,        button_name,    left).
90    resource(modifier,      modifier,       '').
91    resource(cursor,        cursor,         hand2).
92    variable(outline,       box,            get,
93             "Outline to 'warp' objects").
94    initialise(G, B:[button_name], M:[modifier]) :->
95            send(G, send_super, initialise, B, M),
96            send(G, slot, outline, new(Box, box(0,0))),
97            send(Box, texture, dotted).
```

The verify method is called to validate it is ok to start the gesture. In this context, this implies the canvas is in select mode and there are actually objects displayed. It is called after a button-down of the appropriate button with the appropriate modifier is detected.

```
98    verify(_G, Ev:event) :->
99            get(Ev, receiver, Canvas),
100           get(Canvas, mode, select),
101           \+ send(Canvas?graphicals, empty).
```

After 'Gesture →*verify*' succeeds 'Gesture →*initiate*' is called to start the gesture. It resizes the outline to size(0,0) using the 'Graphical →*set*' (which avoids creating a size object) and than displays it at the mouse-position.

```
102   initiate(G, Ev:event) :->
103           get(Ev, receiver, Canvas),
104           send(G?outline, set, @default, @default, 0, 0),
105           send(Canvas, display, G?outline, Ev?position).
```

On each drag-event, this method is called. It just resizes the outline.

```
106   drag(G, Ev:event) :->
107           send(G?outline, corner, Ev?position).
```

On the corresponding up-event, this method is called. It removes the outline from the device and sends 'draw_canvas →*selection*' to the canvas with a chain of all objects inside the area.

```
108   terminate(G, Ev:event) :->
109           send(G, drag, Ev),
110           get(G, outline, Outline),
111           get(Ev, receiver, Canvas),
112           send(Outline, device, @nil),
113           send(Canvas, selection, ?(Canvas, inside, Outline?area)).
114   :- pce_end_class.
```

3.4.3 Create from prototype

Prototypes have their own size, which implies creating a prototype is done using a simple click. It first displays a clone of 'draw_canvas ←*proto*' at the position of the mouse. Next it sends the →*start_text* message to the created prototype to allow the user filling the text-fields of the proto instance.

```
115  :- pce_global(@draw_create_proto_recogniser,
116                 make_create_proto_recogniser).
117  make_create_proto_recogniser(R) :-
118          new(Canvas, @event?receiver),
119          new(Proto, Canvas?proto),
120          new(R, click_gesture(left, '', single,
121                               block(assign(new(Clone, var), Proto?clone),
122                                     message(Canvas, display,
123                                             Clone, @event?position),
124                                     if(message(Clone, has_send_method,
125                                                start_text),
126                                        message(Clone, start_text))),
127                               Canvas?mode == create_proto)).
```

3.4.4 Create resizable shape

Create shapes that do not have a predefined size. The top-left-corner of the object will be at the mouse-down location, the bottom-right-corner at the mouse-up location.

```
128  :- pce_begin_class(draw_create_resize_gesture,  gesture).
129  resource(button,        button_name,    left).
130  resource(modifier,      modifier,       '').
131  resource(cursor,        cursor,         bottom_right_corner).
132  resource(minimum_size,  int,            3,
133           "Mimimum width/height of the object").
134  variable(object,        graphical*,     both,
135           "Object created").
136  verify(_G, Ev:event) :->
137          "Only active when in create_resize mode"::
138          get(Ev?receiver, mode, create_resize).
```

Display a clone of 'draw_canvas ←*proto*' and attach it to the gesture. The latter is necessary because `@event`?receiver refers to the canvas.

```
139  initiate(G, Ev:event) :->
140          "Paint the prototype"::
141          get(Ev, receiver, Canvas),
142          get(Canvas?proto, clone, Object),
143          send(G, object, Object),
144          send(Canvas, display, Object, Ev?position).
```

Drag is easy. The only non-standard thing it does is to disallow the width or height of the created object to become negative.

```
145   drag(G, Ev:event) :->
146           "Resize the object"::
147           get(Ev, position, Pos),
148           get(G, object, Obj),
149           get(Pos, x, EX), get(Pos, y, EY),
150           get(Obj, x, OX), get(Obj, y, OY),
151           max(EX, OX, CX),
152           max(EY, OY, CY),
153           send(Obj, corner, point(CX, CY)).

154   max(A, B, M) :- A >= B, !, M = A.
155   max(_, B, B).
```

Terminate checks whether the created object is too small and then deletes it. It resets the $\leftrightarrows$*object* variable of the gesture. The latter is necessary to avoid a dangling reference when the created object would be destroyed: this object does not know it is referenced by the gesture.

```
156   terminate(G, Ev:event) :->
157           "Delete the object if it is too small"::
158           send(G, drag, Ev),
159           get(G, object, Obj),
160           send(G, object, @nil),
161           get(Obj, width, W),
162           get(Obj, height, H),
163           abs(W, AbsW),
164           abs(H, AbsH),
165           get(G, resource_value, minimum_size, S),
166           (   (AbsW < S ; AbsH < S)
167           ->  send(Obj, free)
168           ;   get(Ev, receiver, Canvas),
169               send(Canvas, auto_align, Obj, create),
170               send(Canvas, modified)
171           ).
172   abs(X, Y) :-
173           (   X < 0
174           ->  Y is -X
175           ;   Y = X
176           ).
177   :- pce_end_class.
```

3.4.5 Line

Creating a line is very similar to creating a resizable shape. Only, $\rightarrow$*drag* sets the end-point rather than the corner and $\rightarrow$*terminate* should validate the length rather than the minimum of width and height.

```
178  :- pce_begin_class(draw_create_line_gesture, draw_create_resize_gesture).
179  resource(cursor,         cursor,        plus).
180  verify(_G, Ev:event) :->
181          "Only active when in create_line_mode"::
182          get(Ev?receiver, mode, create_line).
183  drag(G, Ev:event) :->
184          send(G?object, end, Ev?position).
185  terminate(G, Ev:event) :->
186          send(G, drag, Ev),
187          get(G, object, Line),
188          send(G, object, @nil),
189          get(Line, length, L),
190          get(G, resource_value, minimum_size, MS),
191          (   L < MS
192          ->  send(Line, free)
193          ;   get(Ev, receiver, Canvas),
194              send(Canvas, auto_align, Line, create)
195          ).
196  :- pce_end_class.
```

The draw_change_line_gesture does to a line what the resize_gesture does to an object that has a real area: one can drag one of the end-points.

```
197  :- pce_begin_class(draw_change_line_gesture, gesture).
198  resource(button,         button_name,    middle).
199  resource(cursor,         cursor,         plus).
200  variable(side,           name*,          both,
201           "Start or end").
```

Verify tries to find the end-point and records the result in the variable $\leftrightarrows$*side*. It fails if the event is too far away from either end of the line.

```
202  verify(G, Ev:event) :->
203          get(Ev, receiver, Line),
204          get(Ev, position, Line?device, Pos),
205          (   get(Line?start, distance, Pos, D),
206              D < 5
207          ->  send(G, side, start)
208          ;   get(Line?end, distance, Pos, D),
209              D < 5
210          ->  send(G, side, end)
211          ;   fail
212          ).
213  initiate(G, Ev:event) :->
214          get(Ev, receiver, Line),
215          send(Line?device, pointer, Line?(G?side)).
216  drag(G, Ev:event) :->
217          get(Ev, receiver, Line),
```

```
218             get(G, side, Side),
219             send(Line, Side, ?(Ev, position, Line?device)).
220     terminate(G, Ev:event) :->
221             send(G, drag, Ev).
222     :- pce_end_class.
```

3.4.6 Path

Class 'draw_create_path_gesture' is the most complicated of PceDraw's gestures because it does not yet fit in very well with the concept of 'gesture' that describes event-handling from a button-down upto the corresponding button-up. A path is created by clicking on each subsequent control-point.

```
223     :- pce_begin_class(draw_create_path_gesture, gesture).
224     resource(cursor,         cursor,        cross).
225     resource(button,         button_name,   left).
226     variable(path, path*, both, "Currently painted path").
227     variable(line, line,  get,  "Line segment for last").
228     initialise(G, Button:[button_name]) :->
229             send(G, send_super, initialise, Button),
230             send(G, slot, line, new(Line, line)),
231             send(Line, texture, dotted).
```

The →*event* method is redefined for two purposes: 1) when a path is beeing created a dotted line is displayed from the last control-point to the current mouse location (achieved by trapping the 'loc_move' events) and 2) when the user presses ESC or another mouse-button, the path is terminated.

```
232     event(G, Ev:event) :->
233             "Process an event"::
234             get(Ev?receiver, mode, create_path),
235             (   send(G, send_super, event, Ev)
236             ->  true
237             ;   get(G, path, Path), Path \== @nil,
238                 (   send(Ev, is_a, loc_move)
239                 ->  send(G, move, Ev)
240                 ;   (send(Ev, is_a, 27) ; send(Ev, is_a, button))   % terminate
241                 ->  send(Ev?window, focus, @nil),
242                     send(G, terminate_path)
243                 )
244             ).
```

→*Initiate* is called on each button-down. If there is no current path it is a 'real' initiate. If there is already a current path this method just succeeds.

```
245   initiate(G, Ev:event) :->
246           "Paint the prototype"::
247           get(G, path, CurrentPath),
248           (   CurrentPath == @nil
249           ->  get(Ev, receiver, Canvas),
250               get(Ev, position, Canvas, Pos),
251               get(Canvas?proto, clone, Path),
252               send(G, path, Path),
253               get(G, line, Line),
254               send(Line, start, Pos),
255               send(Line, end, Pos),
256               send(Canvas, display, Line),
257               send(Canvas, display, Path)
258           ;   true
259           ).
```

The method →*move* is called from →*event* when there is a current path and the mouse is moved. It replaces the →*drag* method called in normal gestures when the mouse is moved with a button pressed.

```
260   move(G, Ev:event) :->
261           get(G, line, Line),
262           get(Ev, position, Pos),
263           send(Line, end, Pos).
```

Terminate implies a button-up. This method appends the current location to the path; moves the start of the feedback line to the end of the path and invokes 'window →*focus*'. The 3-th argument of this method is the button that caused the event-focus to be grabbed. A button-up event related to this button will release the focus. By setting this button to `@nil`, the focus will not be released. See also →*event*.

```
264   terminate(G, Ev:event) :->
265           send(G, move),
266           send(G?path, append, G?line?end),
267           send(G?line, start, G?line?end),
268           send(Ev?window, focus, Ev?receiver, G, G?cursor, @nil).
```

Terminate the path. Remove the feedback-line; set the current path to `@nil` and finally remove the path if it consists of only 1 point (similar removing text objects without characters; graphicals smallers than a defined minimal size; etc.).

```
269   terminate_path(G) :->
270           get(G, path, Path),
271           send(G?line, device, @nil),
272           send(G, path, @nil),
273           (   get(Path?points, size, Size),
274               Size =< 1
275           ->  send(Path, free)
276           ;   true
277           ).
278   :- pce_end_class.
```

The 'draw_modify_path_gesture' allows the user to drag control-points with the middle-mouse button. The method 'path ←*point*' is used to find the control-point.

```
279  :- pce_begin_class(draw_modify_path_gesture, gesture).
280  resource(cursor,        cursor,         plus).
281  resource(button,        button_name,    middle).
282  variable(point,         point*,         both,   "Point to move").
283  verify(G, Ev:event) :->
284          "Start if event is close to point"::
285          get(Ev, receiver, Path),
286          get(Path, point, Ev, Point),
287          send(G, point, Point).
288  initiate(G, Ev:event) :->
289          "Move pointer to point"::
290          get(Ev, receiver, Path),
291          get(G, point, Point),
292          get(Path, offset, Offset),
293          get(Point, copy, P2),
294          send(P2, plus, Offset),
295          send(Path?device, pointer, P2).
296  drag(G, Ev:event) :->
297          "Move point to pointer"::
298          get(Ev, receiver, Path),
299          get(Path, device, Dev),
300          get(Ev, position, Dev, Pos),
301          get(Path, offset, Offset),
302          send(Pos, minus, Offset),
303          send(Path, set_point, G?point, Pos?x, Pos?y).
304  :- pce_end_class.
```

The two click-gestures below allow the user to insert/delete control-points by left-clicking on them with the control-key depressed. If the user clicks within 3 pixels from a control-point this point is deleted. Otherwise, if the user clicks close to a line-segment, a control-point is inserted between the two points that define the line-segment.

Note that the first click_gesture defines a condition. Whether or not an event is accepted by a click_gesture does not depend on the return-status of the called message. Without a condition, the first click_gesture will accept all left-clicks with the control-key helt down. The second click_gesture would never be activated.

```
305  :- pce_global(@draw_edit_path_gesture, make_draw_edit_path_gesture).
306  make_draw_edit_path_gesture(G) :-
307          new(G, handler_group),
308          send(G, append,
309               new(C1, click_gesture(left, c, single,
310                                     message(@receiver, delete,
311                                             ?(@receiver, point, @event, 3))))),
312          send(C1, condition, ?(@event?receiver, point, @event, 3)),
313          send(G, append,
```

```
314                 click_gesture(left, c, single,
315                               message(@receiver, insert,
316                                       ?(@event, position, @receiver?device),
317                                       ?(@receiver, segment, @event)))).
```

3.4.7 Text

The recognisers below define the creation of a text object and start editing a text object. Note the use of keyboard_focus; if 'Window ⇆*keyboard_focus*' is nonequal to `@nil`, all typing is tranferred to the keyboard_focus. Objects receive 'obtain_keyboard_focus' and 'release_keyboard_focus' events when they get or loose the keyboard focus.

```
318   :- pce_global(@draw_create_text_recogniser,
319                 make_draw_create_text_recogniser).
320   :- pce_global(@draw_edit_text_recogniser,
321                 make_draw_edit_text_recogniser).
322   :- pce_global(@draw_compound_draw_text_recogniser,
323                 make_draw_compound_draw_text_recogniser).
```

After 'Device →*display*' the new graphical is at the end of the 'Device ←*graphicals*' chain and thus can be found using:

```
Canvas?graphicals?tail
```

Note that the last argument of the click_gesture is the preview action, but may also be used as a condition.

```
324   make_draw_create_text_recogniser(R) :-
325           new(Canvas, @event?receiver),
326           new(Pos, @event?position),
327           new(Text, Canvas?graphicals?tail),
328           new(R, click_gesture(left, '', single,
329                                block(message(Canvas, display,
330                                              Canvas?proto?clone, Pos),
331                                      message(Canvas, keyboard_focus, Text),
332                                      message(Canvas, auto_align, Text, create)),
333                                Canvas?mode == create_text)).
334   make_draw_edit_text_recogniser(R) :-
335           new(Text, @event?receiver),
336           new(Canvas, Text?window),
337           new(Pointed, ?(Text, pointed, @event?position)),
338           new(R, click_gesture(left, '', single,
339                                block(message(Text, caret, Pointed),
340                                      message(Canvas, keyboard_focus, Text)),
341                                Canvas?mode == edit_text)).
342   make_draw_compound_draw_text_recogniser(R) :-
343           new(Compound, @event?receiver),
344           new(Canvas, Compound?window),
345           new(R, click_gesture(left, '', single,
346                                message(Compound, start_text, @event),
347                                Canvas?mode == edit_text)).
```

3.4.8 Move

The move_selection gesture is active when an object is moved that is selected and there are more objects selected. In this case all selected objects are moved by the same amount. This is indicated by showing an outline that reflects the bounding box of all objects moved.

This gesture illustrates how another gesture can be encapsulated. It is a subclass of 'move_gesture' to inherit the button and modifier resources.

```
348   :- pce_begin_class(draw_move_selection_gesture, move_gesture).
349   variable(outline,       box,    get,
350            "Box used to indicate move").
351   variable(selection,     chain*, both,
352            "Stored value of device selection").
353   variable(origin,        point,  get,
354            "Start origin of selection").
```

The gesture maintains an outline, the selection to be moved and the positon where the move orginiated. The outline itself is given a normal move_gesture to make it move on dragging. This move_gesture should operate on the same button and modifier.

```
355   initialise(G, B:[button_name], M:[modifier]) :->
356           send(G, send_super, initialise, B, M),
357           send(G, slot, outline, new(Box, box(0,0))),
358           send(G, slot, origin, point(0,0)),
359           send(Box, texture, dotted),
360           send(Box, recogniser, move_gesture(G?button, G?modifier)).
```

Verify the object is selected and there is at least one more object selected.

```
361   verify(_G, Ev:event) :->
362           get(Ev, receiver, Receiver),
363           get(Receiver, selected, @on),
364           get(Receiver?device?graphicals, find,
365               and(@arg1?selected == @on,
366                   @arg1 \== Receiver), _).
```

Initiating implies finding the device and the bounding box of all selected objects (= the 'union' of their areas). Next, the outline is displayed and all events are posted to the outline. The move_gesture of the outline ensures the outline is moved by the dragging events.

```
367   initiate(G, Ev:event) :->
368           get(Ev?receiver, device, Dev),
369           get(G, outline, Outline),
370           send(G, selection, Dev?selection),
371           get(G, selection, Selection),
372           new(Union, area(0,0,0,0)),
373           send(Selection, for_all, message(Union, union, @arg1?area)),
374           send(G?origin, copy, Union?position),
375           send(Outline, area, Union),
376           send(Union, done),
```

```
377             send(Dev, display, Outline),
378             send(Ev, post, Outline).
379     drag(G, Ev) :->
380             send(Ev, post, G?outline).
```

Terminate. First undisplay the outline. Next calculate by how much the outline has been dragged and move all objects of the selection by this amount.

```
381     terminate(G, Ev:event) :->
382             send(G, drag, Ev),
383             get(G, outline, Outline),
384             send(Outline, device, @nil),
385             get(Outline?area?position, difference, G?origin, Offset),
386             send(G?selection, for_all, message(@arg1, relative_move, Offset)),
387             send(G, selection, @nil),
388             send(Ev?receiver?window, modified).
389     :- pce_end_class.
```

3.4.9 Resize

Resizing the selection is very similar to moving it. Resizing a group of object implies finding the origin of the resize (e.i. the coordinates of the corner of the resized area that does not move) and the resize factor in both X and Y-direction. Thus, the following steps are taken:

1. On initiating, display a box indicating the bounding box of the selection and start resizing this box.
2. After resizing of the bounding box is completed, compute the static origin and the resize factors.
3. Send a →*resize* message to all the individual graphicals.

```
390     :- pce_begin_class(draw_resize_selection_gesture, resize_gesture).
391     variable(outline,       box,    get,
392              "Box used for feedback").
393     variable(selection,     chain*, both,
394              "Stored value of device selection").
395     variable(start,         area,   get,
396              "Area before resize started").
```

The outline operates the same way as the outline of the selection_move handler.

```
397     initialise(G, B:[button_name], M:[modifier]) :->
398             send(G, send_super, initialise, B, M),
399             send(G, slot, outline, new(Box, box(0,0))),
400             send(G, slot, start, area(0,0,0,0)),
401             send(Box, texture, dotted),
402             send(G, min_size, size(3, 3)),
403             send(Box, recogniser, resize_gesture(G?button, G?modifier)).
```

```
verify(G, Ev:event) :->
        get(Ev, receiver, Receiver),
        get(Receiver, selected, @on),
        send(G, send_super, verify, Ev).
```

Compute the bounding box of the selection, display the outline and post the event to the outline.

```
initiate(G, Ev:event) :->
        get(Ev?receiver, device, Dev),
        get(G, outline, Outline),
        send(G, selection, Dev?selection),
        get(G, selection, Selection),
        get(G, start, Start),
        send(Start, clear),
        send(Selection, for_all, message(Start, union, @arg1?area)),
        send(Outline, area, Start),
        send(Dev, display, Outline),
        (   send(Ev, post, Outline)     % cancel!
        ->  true
        ;   send(Outline, device, @nil),
            send(G, selection, @nil),
            fail
        ).

drag(G, Ev) :->
        send(Ev, post, G?outline).
```

Compute the resize factors and resize the contents of the selection.

```
terminate(G, Ev:event) :->
        send(G, drag, Ev),
        get(G, outline, Outline),
        send(Outline, device, @nil),
        get(G, start, A0),
        get(Outline, area, A1),
        x_resize(A0, A1, X0, Xfactor),
        y_resize(A0, A1, Y0, Yfactor),
        send(G?selection, for_all,
             message(@arg1, resize, Xfactor, Yfactor, point(X0, Y0))),
        send(G, selection, @nil),
        send(Ev?receiver?window, modified).

x_resize(A0, A1, X0, Xfactor) :-
        get(A0, left_side, Left),
        get(A1, left_side, Left), !,              % left-side has not changed
        X0 = Left,
        get(A0, width, W0),
        get(A1, width, W1),
        Xfactor is W1 / W0.
x_resize(A0, A1, X0, Xfactor) :-
        get(A0, right_side, Right),
```

```
447             X0 = Right,
448             get(A0, width, W0),
449             get(A1, width, W1),
450             Xfactor is W1 / W0.
451     y_resize(A0, A1, Y0, Yfactor) :-
452             get(A0, top_side, Top),
453             get(A1, top_side, Top), !,                % top has not changed
454             Y0 = Top,
455             get(A0, height, H0),
456             get(A1, height, H1),
457             Yfactor is H1 / H0.
458     y_resize(A0, A1, Y0, Yfactor) :-
459             get(A0, bottom_side, Bottom),
460             Y0 = Bottom,
461             get(A0, height, H0),
462             get(A1, height, H1),
463             Yfactor is H1 / H0.
464     :- pce_end_class.
465     :- pce_begin_class(draw_resize_gesture, resize_outline_gesture).
466     terminate(G, Ev:event) :->
467             "Invoke auto_align"::
468             send(G, send_super, terminate, Ev),
469             get(Ev, receiver, Shape),
470             send(Shape?device, auto_align, Shape, resize).
471     :- pce_end_class.
472     :- pce_begin_class(draw_move_gesture, move_outline_gesture).
473     terminate(G, Ev:event) :->
474             "Invoke auto_align"::
475             send(G, send_super, terminate, Ev),
476             get(Ev, receiver, Shape),
477             send(Shape?device, auto_align, Shape, move).
478     :- pce_end_class.
```

3.4.10 Connect

The code below is a refinement of the connect_gesture defined in PCE itself. It verifies the canvas is in the right mode and sets the $\leftrightarrows$*link* attribute of the gesture. This attribute will later be used to create the connection from.

The 'connect_gesture $\rightarrow$*connect*' behaviour has been redefined as well. The standard one uses a 'connection', while this one should create a 'draw_connection'.

```
479     :- pce_begin_class(draw_connect_gesture, connect_gesture).
480     verify(G, Ev:event) :->
481             "Verify canvas is in connect-mode"::
482             get(Ev?receiver, device, Dev), Dev \== @nil,
483             get(Dev, mode, connect),
```

```
484             send(G, link, Dev?proto),
485             send(G, send_super, verify, Ev).

486     connect(_G, From:graphical, To:graphical, Link:link,
487                 FH:[name], TH:[name]) :->
488             "Connect the graphicals (using a draw_connection)"::
489             new(_, draw_connection(From, To, Link, FH, TH)).

490     :- pce_end_class.
```

3.4.11 Connect create handle

```
491     :- pce_begin_class(draw_connect_create_gesture, gesture).
```

The 'draw_connect_create_gesture' is an example of a complete gesture class. It connects two graphicals at arbitrary points by attaching new handles to the graphicals and creating a connection between them.

```
492     variable(line,                  line,           get,
493              "Line indicating link").
494     variable(from_indicator,        bitmap,         get,
495              "Indicator at 'from' side").
496     variable(to_indicator,          bitmap,         get,
497              "Indicator at 'to' side").
498     variable(to,                    graphical*,     get,
499              "Graphical to connect to").

500     resource(button,                button_name,    left,
501              "Button used to connect (left)").
502     resource(modifier,              modifier,       '',
503              "Modifier used to connect").
```

Initialise the line and markers of the gesture.

```
504     initialise(G, B:[button_name], M:[modifier]) :->
505             send(G, send_super, initialise, B, M),
506             send(G, slot, line, line(0,0,0,0)),
507             send(G, slot, from_indicator, new(bitmap(@mark_handle_image))),
508             send(G, slot, to_indicator, new(bitmap(@mark_handle_image))).

509     verify(_G, Ev:event) :->
510             "Verify canvas is in connect_create-mode"::
511             get(Ev?receiver?device, mode, connect_create).
```

Indicate the start-location using the ←*from_indicator*, give the feedback-line the appropriate attributes and display it.

```
512     initiate(G, Ev:event) :->
513             "Start drawing line"::
514             get(Ev?receiver, device, Dev),
515             get(Dev, proto, Link),
516             get(Ev, position, Dev, Pos),
517             send(G?line, copy, Link?line),
```

```
518             send(G?line, texture, dotted),
519             send(G?line, start, Pos),
520             send(G?line, end, Pos),
521             send(Dev, display, G?line),
522             send(G, indicate, Ev?receiver, Pos, G?from_indicator).
```

Update the line, check whether the mouse points to a valid target and display a marker on it. Note how the target is located using the method 'Chain $\leftarrow$*find*'. This keeps everything inside PCE, avoiding interface overhead and producing far less garbage. 'Gesture $\rightarrow$*drag*' should be as fast as possible and not produce too much garbage as it will be called about 40 times per second while the mouse is dragged.

```
523   drag(G, Ev:event) :->
524       get(Ev, receiver, Receiver),
525       get(Receiver, device, Dev),
526       get(Ev, position, Dev, Pos),
527       send(G?line, end, Pos),
528       (   get(?(Dev, pointed_objects, Pos), find,
529               and(Receiver \== @arg1,
530                   G?line \== @arg1,
531                   G?from_indicator \== @arg1,
532                   G?to_indicator \== @arg1), To)
533       ->  send(G, indicate, To, Pos, G?to_indicator),
534           send(G, slot, to, To)
535       ;   send(G, slot, to, @nil),
536           send(G?to_indicator, device, @nil)
537       ).
```

If there is a target, create unique handles on both sides and link them together.

```
538   terminate(G, Ev:event) :->
539           send(G, drag, Ev),
540           send(G?line, device, @nil),
541           send(G?from_indicator, device, @nil),
542           send(G?to_indicator, device, @nil),
543           get(G, to, To),
544           (   To \== @nil
545           ->  send(G, slot, to, @nil),
546               get(Ev, receiver, Receiver),
547               get(Receiver?device, proto, Link),
548               get(G, handle, Receiver, G?from_indicator?center, Link?from, FH),
549               get(G, handle, To, G?to_indicator?center, Link?to, TH),
550               new(_, draw_connection(Receiver, To, Link, FH, TH))
551           ;   true
552           ).
```

Create a unique handle on a graphical at the indicated position. The position of the handle is taken relative to the size of the graphical.

```
553   handle(_G, Gr:graphical, Pos:point, Kind:name, Name) :<-
554           "Attach a handle at specified position and return it's name"::
555           get(Gr, x, X), get(Gr, y, Y),
556           get(Gr, width, W), get(Gr, height, H),
557           get(Pos, x, PX), get(Pos, y, PY),
558           RX is PX - X, RY is PY - Y,
559           unique_handle_name(Gr, Name),
560           send(Gr, handle, handle((RX/W) * w, (RY/H) * h, Kind, Name)).

561   unique_handle_name(Gr, Name) :-
562           between(1, 10000, N),
563           concat(c, N, Name),
564           \+ get(Gr, handle, Name, _), !.

565   indicate(_G, Gr:graphical, Pos:point, Indicator:bitmap) :->
566           "Display indication-marker for position"::
567           send(Indicator, center, Pos),
568           send(Gr?device, display, Indicator).

569   :- pce_end_class.
```

3.4.12 Shape popup

The code of this section attaches a popup-menu to the shapes. On a mouse-right-down event, the shape on which the down occurred is selected to indicate on which object the operation will take place. Next, the menu is shown.

```
570   :- pce_global(@draw_shape_popup_gesture, make_draw_shape_popup_gesture).

571   make_draw_shape_popup_gesture(G) :-
572           new(Gr, @event?receiver),
573           new(Canvas, Gr?device),

574           new(P, popup),
575           send_list(P, append,
576                     [ menu_item(align,
577                                 message(Canvas, align_with_selection, Gr),
578                                 @default, @on)
579                     , menu_item(duplicate,
580                                 block(message(Canvas, selection, Gr),
581                                       message(Canvas, duplicate_selection)))
582                     , menu_item(cut,
583                                 message(Canvas, edit,
584                                         message(@arg1, free), Gr),
585                                 @default, @on)
586                     , menu_item(edit_attributes,
587                                 block(message(Canvas, selection, Gr),
588                                       message(Canvas, edit_selection)),
589                                 @default, @on)
590                     , menu_item(hide,
591                                 message(Canvas, edit,
592                                         message(@arg1, hide), Gr))
593                     , menu_item(expose,
```

```
                                message(Canvas, edit,
                                        message(@arg1, expose), Gr),
                                @default, @on)
                    ]),
        new(G, draw_draw_shape_popup_gesture(P)).

:- pce_begin_class(draw_draw_shape_popup_gesture, popup_gesture).
variable(old_selected,  bool*,  both, "Was graphical selected").
verify(G, Ev:event) :->
        get(Ev?receiver, device, Dev),
        Dev \== @nil,
        send(Dev?class, is_a, draw_canvas),
        send(G, send_super, verify, Ev).

initiate(G, Ev:event) :->
        get(Ev, receiver, Receiver),
        send(G, old_selected, Receiver?selected),
        send(Receiver, selected, @on),
        send(G, send_super, initiate, Ev).

terminate(G, Ev:event) :->
        get(G, context, Gr),
        send(Gr, selected, G?old_selected),
        send(G, send_super, terminate, Ev).
:- pce_end_class.
```

3.5 Source file "menu.pl"

```
1     /*  $Id: menu.pl,v 1.7 1993/09/03 09:52:18 jan Exp $
2         Part of XPCE
3         Designed and implemented by Anjo Anjewierden and Jan Wielemaker
4         E-mail: jan@swi.psy.uva.nl
5         Copyright (C) 1992 University of Amsterdam. All rights reserved.
6     */
```

This module defines the mode-selection menu at the left-side of the canvas. It consists of two classes: draw_menu, which is a subclass of picture and which is responsible for communication, load/save, etc. and draw_icon, which defines the combination of a mode, a cursor and a prototype.

There are two reasonable primitives for implementing this menu. The first is to use a dialog window and a choice menu, of which the menu_items have image labels. The second is the approach taken in this file, to use a picture with a 1-column format attached to it and images for the options. Which of them is to be preferred is difficult to tell. Both approaches require about the same amount of programming. I've chosen for the latter approach, partly for 'historical' reasons and partly to illustrate how non-standard menus can be created using ordinary graphicals.

As the user can modify the menu by adding/deleting prototypes and changing prototype attributes, the contents of this menu can be saved to file.

```
7     :- module(draw_menu, []).
8     :- use_module(library(pce)).
9     :- require([ concat/3
10                , ignore/1
11                , memberchk/2
12                , send_list/3
13                ]).
```

3.5.1 Icon menu

```
14    :- pce_begin_class(draw_menu, window).
```

Variables to keep track of load/save.

```
15    variable(file,          file*,  both,
16             "File for storing prototypes").
17    variable(modified,      bool,   get,
18             "Menu has been modified").
```

Create the picture. The width of the picture is fixed using the →*hor_stretch* and →*hor_shrink* methods. Next, a 'format' object is attached to the picture. When a format is attached to a device, the graphicals are located according to the format specification. Attaching a format object to a device is a simple way to represent tabular information in PCE. [11]

[11] Formats are a rather hacky solution. There are plans to extend them with a more powerful table mechanisms.

```
19    initialise(M) :->
20            send(M, send_super, initialise, 'Icons', size(48, 200)),
21            send_list(M, [hor_stretch, hor_shrink], 0),
22            send(M, format, new(Fmt, format(horizontal, 1, @on))),
23            send(Fmt, row_sep, 0),
24            send(M, modified, @off).

25    modified(M, Value:[bool]) :->
26            default(Value, @on, Val),
27            send(M, slot, modified, Val).
```

Attach a new prototype. Note that we do not have to specify a position as the attached format object will ensure the new icon is displayed at the bottom.

```
28    proto(M, Proto:'graphical|link*', Mode:name, Cursor:cursor) :->
29            "Attach a new prototype"::
30            send(M, display, draw_icon(Proto, Mode, Cursor)),
31            send(M, modified, @on).

32    current(M, Icon) :<-
33            "Find current icon"::
34            get(M?graphicals, find, @arg1?inverted == @on, Icon).

35    activate_select(M) :->
36            "Activate icon that does select"::
37            get(M?graphicals, find, @arg1?mode == select, Icon),
38            send(Icon, activate).
```

3.5.2 Create

Create a prototype from a chain of graphicals (usually the selection; in the future this might also come from a prototype editor). If the chain has one element, no compound is needed. [12]

```
39    create_proto(M, Graphicals:chain) :->
40            "Create a prototype from a chain of graphicals"::
41            get(Graphicals, size, Size),
42            (   Size == 0
43            ->  send(@display, inform, 'No selection')
44            ;   Size == 1
45            ->  get(Graphicals?head, clone, Proto),
46                send(Proto, selected, @off)
47            ;   new(Proto, draw_compound),
48                get(Graphicals, clone, Members),
49                send(Members, for_all,
50                     and(message(Proto, display, @arg1),
51                         message(@arg1, selected, @off))),
```

[12] Due to the improper functioning of ←*clone* with regards to connections to the outside world, all connections should be internal to the chain of graphicals. We won't try to program around this problem here, but improve PCE's kloning schema later.

```
52              send(Proto, reference, @default),
53              send(Proto, string, '')
54          ),
55          send(M, proto, Proto, create_proto, dotbox).
```

3.5.3 Delete

```
56  can_delete(M) :->
57          "Test if current prototype may be deleted"::
58          get(M, current, Icon),
59          send(Icon, can_delete).

60  delete(M) :->
61          "Delete current prototype"::
62          get(M, current, Icon),
63          (   send(Icon, can_delete)
64          ->  send(M, activate_select),
65              send(Icon, free),
66              send(M, modified, @on)
67          ;   send(@display, inform, 'Can''t delete this prototype'),
68              fail
69          ).
```

3.5.4 Save/load

Saving/loading is very similar to the corresponding code in canvas.pl.

```
70  save_as(M) :->
71          "Save in user-requested file"::
72          get(@finder, file, @off, '.proto', File),
73          send(M, save, File).

74  save(M, File:[file]) :->
75          "Save prototypes to named file"::
76          (   File == @default
77          ->  get(M, file, SaveFile),
78              SaveFile \== @nil
79          ;   send(M, file, File),
80              SaveFile = File
81          ),
82          send(M?graphicals, save_in_file, SaveFile),
83          send(M, modified, @off).

84  load_from(M) :->
85          "Load from user-requested file"::
86          get(@finder, file, @on, '.proto', File),
87          send(M, load, File).

88  load(M, File:[file]) :->
89          "Load prototypes from named file"::
90          (   File == @default
```

```
91            ->  get(M, file, LoadFile),
92                LoadFile \== @nil
93            ;   send(M, file, File),
94                LoadFile = File
95            ),
96            send(M, clear),
97            get(LoadFile, object, Chain),
98            send(Chain, for_all, message(M, display, @arg1)),
99            send(M?graphicals?head, activate),
100           send(M, modified, @off).
101   :- pce_end_class.
```

3.5.5 Icons

We have chosen to specialise class 'bitmap' to represent the icon. Each icon represents a prototype, a mode and a cursor that is used by the canvas to indicate the mode. The visual representation of an icon is an outline that indicates the mode and a small version of the prototype to indicate what is drawn.

There are two reasonable choices for this job. One is to use a subclass of device and display the outline and a resized clone of the prototype. The other is to use class bitmap and draw a clone of the prototype in it. It is difficult to say which of the two is better. I finally decided that just a bitmap is cheaper to save (considering the fact that the device case holds a bitmap of the same size too). Another criterium is how difficult it is to change an argument of the prototype. For a device this is slightly simpler as we just pass the message to change the argument to the prototype and the clone of the prototype displayed in the icon. Using a bitmap, we have to recompute the contents of the bitmap. This however is not very hard.

```
102   :- pce_begin_class(draw_icon, bitmap).
103   variable(proto,         'graphical|link*',      get,
104            "Prototype represented").
105   variable(mode,          name,           both,
106            "Mode initiated by the icon").
107   variable(mode_cursor,   name,           both,
108            "Associated cursor-name").

109   initialise(I, Proto:'graphical|link*', Mode:name, Cursor:cursor) :->
110           "Create an icon for a specific mode"::
111           send(I, send_super, initialise, image(@nil, 48, 32)),
112           send(I, mode, Mode),
113           send(I, proto, Proto),
114           send(I, slot, mode_cursor, Cursor?name).

115   can_delete(I) :->
116           "Can I delete this icon?"::
117           get(I, mode, create_proto).
```

3.5.6 Prototypes

```
118   proto(I, Proto:'graphical|link*') :->
119           "Set the prototype"::
120           send(I, slot, proto, Proto),
121           send(I, paint_proto, Proto).
```

Create the image of the icon. First, we will paint the outline, indicating the mode. Next, we make a copy of the prototype (because we have to modify it and we should not change the original prototype), modify the text to 'T' and the size to fit in the icon. Finally, we draw the prototype in the icon and send 'Object →*done*' to the clone to inform PCE we have done with it.

```
122   paint_proto(I, Proto:'link|graphical*') :->
123           "Paint a small version of the prototype"::
124           send(I, paint_outline),
125           (   Proto == @nil
126           ->  true
127           ;   send(Proto, instance_of, link)
128           ->  get(Proto?line, clone, Clone),
129               send(Clone, points, 11, 10, 27, 20),
130               send(I, draw_in, Clone)
131           ;   send(Proto, instance_of, path),
132               send(Proto?points, empty)
133           ->  get(Proto, clone, Clone),
134               send(Clone, clear),
135               send(Clone, append, point(10,10)),
136               send(Clone, append, point(20,7)),
137               send(Clone, append, point(30,15)),
138               send(Clone, append, point(15,21)),
139               send(I, draw_in, Clone)
140           ;   get(Proto, clone, Clone),
141               (   send(Clone, has_send_method, string)
142               ->  send(Clone, string, 'T')
143               ;   true
144               ),
145               send(Clone, size, size(30, 14)),
146               send(Clone, center, point(22, 14)),
147               send(I, draw_in, Clone)
148           ).
```

Paint the outline in the bitmap. For each of the outlines, there is a bitmap file named 'Mode.bm' in PCE's bitmap search-path. We copy this image in the bitmap.

```
149   paint_outline(I) :->
150           "Paint the mode indicating bitmap"::
151           get(I, mode, Mode),
152           concat(Mode, '.bm', Outline),
153           send(I, copy, image(Outline)).
```

3.5.7 Attributes

These two methods from the interface to the attribute editor. See also the files 'attribute.pl' and 'shape.pl'. Note that prototypes do not have a position and therefore the 'x' and 'y' should not be regarded arguments.

```
154   has_attribute(I, Att:name) :->
155           "Test if prototype has named attribute"::
156           \+ memberchk(Att, [x, y]),
157           send(I?proto, has_attribute, Att).

158   attribute(I, Att:name, Val:any) :->
159           "Set attribute of prototype"::
160           send(I?proto, Att, Val),
161           send(I, repaint_proto),
162           send(I?window, modified, @on).
```

3.5.8 Activation

The event parsing. Currently we only define left-click to activate the icon. Activating the gesture is done via the →*event* method, so the gestures won't be saved to file.

```
163   :- pce_global(@icon_recogniser,
164                 new(handler_group(click_gesture(left, '', single,
165                                                 message(@event?receiver,
166                                                         activate))))).

167   event(_I, Ev:event) :->
168           send(@icon_recogniser, event, Ev).
```

Activate an icon. First it sets 'Graphical →*inverted*' to `@on` for only this icon in the menu. Note the use of 'Device →*for_all*' and 'if'. This is the most efficient way to reach our goals, both in terms of the amount of code we have to write as in terms of performance.

```
169   activate(I) :->
170           "Select the icon; set mode and proto"::
171           send(I?device, for_all, @default,
172                if(@arg1 == I,
173                   message(@arg1, inverted, @on),
174                   message(@arg1, inverted, @off))),
175           send(I?frame, mode, I?mode, I?mode_cursor),
176           send(I?frame, proto, I?proto).

177   :- pce_end_class.
```

3.6 Source file "attribute.pl"

```
1   /*  $Id: attribute.pl,v 1.6 1993/05/06 10:12:56 jan Exp $
2       Part of XPCE
3       Designed and implemented by Anjo Anjewierden and Jan Wielemaker
4       E-mail: jan@swi.psy.uva.nl
5       Copyright (C) 1992 University of Amsterdam. All rights reserved.
6   */

7   :- module(draw_attribute, []).
8   :- use_module(library(pce)).
9   :- require([ concat_atom/2
10              , member/2
11              , send_list/3
12              ]).
```

This module defines a separate frame that allows the user to set the values of attributes (pen, font, etc.) of shapes in the drawing. The frame contains a single dialog window, which contains dialog_items for each of the (graphical shape) attributes that can be edited.

Regardless of the shape(s) for which we are editing attributes, all dialog items are always displayed. Items that represent attributes not present in the shapes edited are greyed out to indicate such to the user. As the contents of the window changes each time the user changes the selection, non-used items are not removed from the dialog. This would change too much to the dialog, transforming the interface into a "video clip".

```
13  :- pce_begin_class(draw_attribute_editor, frame).
14  variable(editor,        object,         get,
15           "Editor I'm attached too").
16  variable(client,        chain*,         get,
17           "Objects I'm editing the attributes for").
18  %       attributes(?Label, ?Selector)
19  %
20  %       Label is the label of the menu is the dialog.  Selector is the
21  %       name of the method to be activated to change the value.   Used
22  %       both ways around and only local to  this file, Prolog is a far
23  %       easier way to store this  table.  The  alternative would be to
24  %       create  a  sheet and  attach  it to the   class.   This  needs
25  %       extensions to the preprocessor.
26  attribute(pen,          pen).
27  attribute(dash,         texture).
28  attribute(arrows,       arrows).
29  attribute(fill,         fill_pattern).
30  attribute(colour,       colour).
31  attribute(family,       font).
32  attribute(size,         font).
33  attribute(transparent,  transparent).
34  attribute(radius,       radius).
35  attribute(x,            x).
36  attribute(y,            y).
```

```
37    attribute(w,            width).
38    attribute(h,            height).
39    attribute(closed,       closed).
40    attribute(interpolation,interpolation).
```

Create the attribute window. Like the drawing-tool as a whole, the window is a subclass of the PCE class 'frame' for simple communication with its various parts. Note the use of `default/3`.

'Frame ⇆*done_message*' is activated when the frame receives a DELETE message from the window manager, normally from a 'Delete Window' entry of the window manager.

```
41    initialise(A, Draw:object, Label:[name]) :->
42            default(Label, 'Attributes', Lbl),
43            send(A, send_super, initialise, Lbl),
44            send(A, done_message, message(A, quit)),
45            send(A, append, new(D, dialog)),
46            send(A, slot, editor, Draw),
47            fill_dialog(D).
```

Fill the dialog with the various menus. We defined some generic Prolog predicates to create the various menu's.

```
48    fill_dialog(D) :-
49            new(A, D?frame),
50            send(D, append, label(feedback, '')),
51            make_line_menu(Pen,     pen,     [0,1,2,3,4,5]),
52            make_line_menu(Texture, texture, [none, dotted, dashed, dashdot]),
53            make_line_menu(Arrows,  arrows,  [none, second, first, both]),
54            make_fill_pattern_menu(FillPattern),
55            make_colour_menu(Colour),
56            make_font_family_menu(FontFamily),
57            make_font_size_menu(FontSize),
58            make_transparent_menu(Transparent),
59            make_coordinate_menu(X, x),
60            make_coordinate_menu(Y, y),
61            make_coordinate_menu(W, width),
62            make_coordinate_menu(H, height),
63            make_radius_menu(Radius),
64            make_closed_menu(Closed),
65            make_interpolation_menu(Interpolation),
66            send_list([Closed, Interpolation], align_in_column, @off),
67            send_list(D, append,
68                      [Pen, Texture, Arrows, FillPattern, Colour, Radius, Closed]),
69            send(D, append, Interpolation, right),
70            send(D, append, FontFamily),
71            send(D, append, FontSize),
72            send(D, append, Transparent),
73            send(D, append, X),
74            send(D, append, Y, right),
75            send(D, append, W, right),
```

```
76            send(D, append, H, right),
77            send(D, append, button(quit, message(A, quit, @on))).
```

3.6.1 Menu's

To create the menu's, we defined a predicate `make_proto_menu/4`. Each menu_item has as value the attribute value and as label an image with the prototype with the corresponding value set. Using this approach, the user can easily see what a specific attribute means. When the user selects a menu-item, the menu will send the value itself.

```
78    make_line_menu(Menu, Attribute, Values) :-
79            new(Proto, line(2, 8, 28, 8)),
80            make_proto_menu(Menu, Proto, Attribute, Values),
81            send(Proto, done).

82    make_fill_pattern_menu(Menu) :-
83            new(Proto, box(30, 16)),
84            make_proto_menu(Menu, Proto, fill_pattern,
85                            [ @nil
86                            , @white_image
87                            , @grey12_image
88                            , @grey25_image
89                            , @grey50_image
90                            , @grey75_image
91                            , @black_image
92                            ]),
93            send(Proto, done).
```

The colour menu. When the display is not a colour display, the only possible colours of an object are `@default` (implying the colour of the device), 'white' and 'black'. On colour displays we will show some more possibilities. For a somewhat larger set of choices, a cycle menu may be more appropriate.

Currently the only way to find out whether you are using a black-and-white or colour display is '`@display` ←*depth*'. This is the number of bits the screen uses to represent a single pixel.

Note that the colour palette is constructed from a box with `@black_image` fill pattern. The problem here is the name of `@black_image`. It does not represent the colour black, but only an image with all pixels set to 1.

```
94    colour_display :-
95            \+ get(@display, depth, 1).

96    colour(white).
97    colour(Colour) :-
98            colour_display,
99            colour_display_colour(Colour).
100   colour(black).

101   colour_display_colour(red).
102   colour_display_colour(green).
```

```
colour_display_colour(blue).
colour_display_colour(yellow).

make_colour_menu(Menu) :-
        new(Proto, box(30, 16)),
        send(Proto, fill_pattern, @black_image),
        findall(colour(Colour), colour(Colour), Colours),
        make_proto_menu(Menu, Proto, colour, [@default|Colours]),
        send(Proto, done).
```

The menu below is for the 'transparent' attribute of text. When `@on` (default), only the pixels of the font are affected. Otherwise, the bounding box of the text will be cleared first. Non-transparent text is often used to mark lines or display on top of filled areas.

```
make_transparent_menu(Menu) :-
        new(Proto, figure),
        send(Proto, display, new(B, box(30,16))),
        send(B, fill_pattern, @grey50_image),
        send(Proto, display, new(T, text('T', left,
                                         font(screen, roman, 10)))),
        send(T, center, B?center),
        send(Proto, send_method, send_method(transparent, vector(bool),
                                             message(T, transparent, @arg1))),
        make_proto_menu(Menu, Proto, transparent, [@on, @off]),
        send(Proto, done).
```

Create a menu for some prototype attribute. Each menu_item has a 'menu_item $\leftrightarrows$*value*' equal to the corresponding element of the 'Values' chain. Each label is a image with an outline-box and 'Proto' with the appropriate attribute setting drawn into it.

```
:- pce_global(@menu_proto_box, new(box(30,16))).

make_proto_menu(Menu, Proto, Attribute, Values) :-
        attribute(Label, Attribute),
        new(Menu, menu(Label, marked,
                       message(@receiver?frame, client_attribute,
                               Attribute, @arg1))),
        send(Menu, off_image, @nil),
        send(Menu, layout, horizontal),
        (   member(Value, Values),
            send(Proto, Attribute, Value),
            new(Bm, bitmap(image(@nil, 30, 16, pixmap))),
            send(Bm, draw_in, @menu_proto_box),
            send(Bm, draw_in, Proto),
            send(Menu, append, menu_item(Value, @default, Bm)),
            fail
        ;   true
        ).
```

The coordibate menu is a rather trivial text_item. Note the setting of the field-width and 'dialog_item →*auto_label_align*: `@off`'. The latter places the items just right to one another instead of vertically aligned in columns. [13]

```
139   make_coordinate_menu(Menu, Selector) :-
140           attribute(Label, Selector),
141           new(Menu, text_item(Label, 0,
142                               message(@receiver?frame, client_attribute,
143                                       Selector, @arg1))),
144           send(Menu, width, 5),
145           send(Menu, auto_label_align, @off),
146           send(Menu, align_in_column, @off).
```

The radius of a box is the radius of the circle sections (arcs) used for rounding the corners. As the user propably does not want to specify an exact number of pixels, a slider-menu is used. As a disadvantage, the range has to be specified in advance, and 100 is not the absolute limit. Note that by setting both the range and the width to 100, the slider operates 1:1.

```
147   make_radius_menu(Menu) :-
148           attribute(Label, radius),
149           new(Menu, slider(Label, 0, 100, 0,
150                            message(@receiver?frame, client_attribute,
151                                    radius, @arg1))),
152           send(Menu, drag, @on),
153           send(Menu, width, 100).

154   make_closed_menu(Menu) :-
155           attribute(Label, closed),
156           new(Menu, menu(Label, marked,
157                          message(@receiver?frame, client_attribute,
158                                  closed, @arg1))),
159           send_list(Menu, append, [@off, @on]).

160   make_interpolation_menu(Menu) :-
161           attribute(Label, interpolation),
162           new(Menu, slider(Label, 0, 10, 0,
163                            message(@receiver?frame, client_attribute,
164                                    interpolation, @arg1))),
165           send(Menu, width, 100).
```

3.6.2 Fonts

Fonts form the most difficult part of the menu's. This is because, although font is just a simple attribute of a text, it is more natural to split the menu in a font-family member and a point-size menu. These menu's have to communicate with the standard protocol, but need to communicate to each other as well.

Below is a list of the font families that can be used from the editor.

[13] We should make a subclass to allow for entering integers only. To do this properly, we should know about each keystroke in the menu rather than only the return.

```
166   font_family(helvetica,  roman).
167   font_family(helvetica,  bold).
168   font_family(helvetica,  oblique).
169   font_family(courier,    roman).
170   font_family(courier,    bold).
171   font_family(courier,    oblique).
172   font_family(times,      roman).
173   font_family(times,      bold).
174   font_family(times,      italic).

175   font_family_name(font(Fam, Style, _), Name) :-
176           concat_atom([Fam, -, Style], Name).
```

Below is an example of object-level programming. I'm aware that its sole contribution to understanding PCE may be indicating PCE is not that simple to use as its developers claim. Having decent support for class-level programming by means of Prolog's term expansion and no support for object-level programming, class level programming will propably be used there were object-level programming would have been much simpler and cheaper (in terms of memory requirements). This problem has to be dealt with.

The menu is a simple cycle menu with one additional and one redefined send method attached to it at the object level. The '→*append*: font' method appends a menu item with ⇆*value* the font and label the point-size of the font. Note that we can't use

```
... append, menu_item(@arg1, @default, @arg1?points) ...
```

As this construct would be expanded to a menu item at creation-time of the message, while we want the message to create a new menu_item instance from `@arg1` (bound to the argument font). Hence we use the '`@pce` ←*instance*' construct.

The second method redefines setting the selection. In this case, the menu items are replaced by menu_items that indicate the possible sizes of this font and the selection is set to the proper size. First of all, the font is put in a local variable 'font' because `@arg1` will be rebound in the 'Chain →*for_all*' to the subsequent member of the `@fonts` database.

The variable is declared with the variable(font,font) construct, set using '`@block` →*font*, Font' and read using '`@block` ←*font*'. `@block` is a reference to the currently executing block(-statement).

The 'Chain →*for_all*' will append all fonts with the same family to the menu. Finally, the selection of the menu is set to the font.

```
177   make_font_size_menu(Menu) :-
178           new(Menu, menu(size, cycle,
179                          message(@receiver?frame, client_attribute,
180                                  font, @arg1))),
181           send(Menu, send_method, send_method(append, vector(font),
182               message(Menu, send_class, append,
183                       ?(@pce, instance, menu_item,
184                         @arg1, @default, @arg1?points)))),
185           send(Menu, send_method, send_method(selection, vector(font),
186               block(assign(new(F, var(font)), @arg1),
187                     message(Menu, clear),
```

```
188                     message(@fonts, for_all,
189                             if(and(@arg2?family == F?family,
190                                    @arg2?style  == F?style),
191                                message(Menu, append, @arg2))),
192                     message(Menu?members, sort,
193                             @arg1?value?points < @arg2?value?points),
194                     message(Menu, send_class, selection, @arg1))))).

195   make_font_family_menu(Menu) :-
196           findall(font(Fam, Style, 14), font_family(Fam, Style), Fonts),
197           new(Menu, menu(family, cycle,
198                          message(@receiver?frame, font_family, @arg1))),
199           send(Menu, send_method, send_method(selection, vector(font),
200               block(assign(new(F, var(font)), @arg1),
201                     message(Menu, send_class, selection,
202                             ?(Menu?members, find,
203                               and(@arg1?value?family == F?family,
204                                   @arg1?value?style == F?style)))))),
205           (   member(Font, Fonts),
206               font_family_name(Font, Name),
207               send(Menu, append, new(I, menu_item(Font, @default, Name))),
208               send(I, font, Font),
209               fail
210           ;   true
211           ).

212   font_family(A, Font:font) :->
213           "Update size menu and pass new font"::
214           get(A, member, dialog, Dialog),
215           get(Dialog, member, size, SizeMenu),
216           get(SizeMenu?selection, points, Size),    % current size
217           new(NewFont, font(Font?family, Font?style, Size)),
218           send(SizeMenu, selection, NewFont),
219           send(A, client_attribute, font, NewFont).
```

3.6.3 Quit

For a secondary window like this attribute editor, it might be a useful idea not to destroy the window if the user hits 'quit', but just to unmap it from the display using 'Frame →*show*: `@off`'. In this case, it can be remapped on the display very quickly and when the window has certain status information attached to it, this will be maintained. For the case of this editor, this only concernes the coordinates of the window.

To control between actual destruction and just unmapping it, an optional boolean argument has been attached. This approach has several advantages. If the caller wants to descriminate, it can do so. For all cases where the caller does not want to discriminate, we have one central place to change the default behaviour.

```
220   quit(A, ShowOff:[bool]) :->
221           (   ShowOff == @on
222           ->  send(A, show, @off)
```

```
223             ;   send(A?editor, attribute_editor, @nil),
224                 send(A, free)
225             ).
```

3.6.4 Client communication

→*fill_items* fills and (de)activates all dialog items. The argument is a chain of shapes (normally the ←*selection* of the canvas). If one of the elements of the selection has the specified attribute, it will be activated and the →*selection* of the menu will be set accordingly.

If more than one object in the selection has some attribute, the →*selection* of the item will be the attribute value of the first object in the chain that is has the attibute. This is a rather simple way of handling this case, but what else can we do?

```
226     fill_items(A, Client) :->
227             "Fill the dialog items from chain of shapes"::
228             get(A, member, dialog, Dialog),
229             attribute(Label, Selector),
230             get(Dialog, member, Label, Menu),
231             (   get(Client, find,
232                     message(@arg1, has_attribute, Selector), Proto),
233                 get(Proto, attribute, Selector, Value)
234             ->  send(Menu, active, @on),
235                 send(Menu, selection, Value)
236             ;   send(Menu, active, @off)
237             ),
238             fail ; true.
```

Set the chain of shapes for which we are editing the attributes. Note that if the window is not shown, we won't update the contents.

```
239     client(A, Client:chain*) :->
240             "Set the graphical I'm editing"::
241             (   get(A, show, @on)
242             ->  get(A, member, dialog, Dialog),
243                 (    Client == @nil
244                 ->   send(Dialog?graphicals, for_some,
245                           message(@arg1, active, @off))
246                 ;    send(A, fill_items, Client)
247                 ),
248                 send(A, slot, client, Client)
249             ;   true
250             ).
```

Set the value of an attribute for the clients. The value is set for each shape that accepts →*has_attribute*.

```
client_attribute(A, Selector:name, Val:any) :->
        "Set attribute of client object"::
        (   get(A, client, Chain), Chain \== @nil
        ->  send(A?client, for_all,
                 if(message(@arg1, has_attribute, Selector),
                    message(@arg1, attribute, Selector, Val)))
        ;   true
        ).

:- pce_end_class.
```

Chapter 4

Conclusions

In this document we presented a medium-sized application to illustrate how applications can be designed and realised using PCE. We have tried to make the design process and design decissions explicit. No doubth it is possible critise the code and decissions made. Nevertheless, we hope the sources of PceDraw form a valuable starting point for programming in PCE/Prolog.

The drawing tool presented in this document may be used as such. It should be noted however that the functionality is incomplete. Notably editing prototypes is limited.

Appendix A

Programming Style

As O'Keefe argues in "The craft of Prolog" [OKeefe, 1990], using a 'good' programming style is not something optional. PCE/Prolog as presented in this document is definitely something different then Prolog with some additional library predicates. PceDraw as presented here is an example of what we currently believe to be good programming style.

A.1 Organisation of sourcefiles

The PCE class compiler allows for the definition of multiple classes in one file. Quintus Prolog compatible Prolog systems allow a file represent at most one Prolog module. What is the best way to organise your sources? There seem to be two reasonable solutions.

Each file either represents a Prolog module and one PCE class, or a bundle of Prolog predicates. Files of the first type generally do not export any predicates. All communication is done by sending messages to instances of the class defined in the file. Files defining normal Prolog predicates do have an export list (otherwise we can't reach their contents). These predicates can be imported as usual.

The second possibility is to define (small) classes that belong to each other or the same category in the same file (and module). Internally, these classes may communicate both using Prolog calls and by sending messages.

A.2 Organisation of a class definition

Below is a list of the various sections that make up a class definition. Except for the header and footer, all the sections are optional. Technically (currently) no ordering between the other sections is required. For clarity it is adviced to use a standard schema for all your classes.

- *Header*
  This is just the `:- pce_begin_class(Class, Super).` declaration.
- *Instance variable declarations*
  The `variable/[3-4]` declarations for additional instance variables.
- *X-resource declarations*
  The `resource/[3-4]` declarations that provide access to the X-resource database.

All aspects that are arbitrary default choices of the UI style should be declared via resources. This enhances clarity of the choices and allows the user to tailor the UI.

- *Handle declarations*
  The `handle/4` declaration to create handles for connections.
- *Initialisation method*
  The initialisation method of a class normally comes first. It is invoked by the PCE vitual machine (VM) operation new() that creates an instance. Messages and predicates that only support the initialisation method (if it is very complicated; long initialisation methods can often be found for dialog windows) are defined right below the method.
- *Unlink method*
  The unlink method is invoked from the PCE VM operation that destroyes an instance. It is normally declared right after the initialisation method.
- *Other reserved methods*
  PCE's internals call various other methods that may be redefined. Examples are 'Graphical →*geometry*', 'Graphical →*event*' and 'Gesture →*initiate*'. These are normally declared before the other methods.
- *Public functionality*
  With this, we refer to methods that facilate the communication with other parts of the application.
- *Local utilities*
  Methods and Prolog predicates that are used from various places within this class definition are placed at the bottom. The reason for this is that one is usually not interrested in these things.
- *Footer*
  The `:- pce_end_class.` declaration terminates the declaration of the class.

Section A.2.1 provides a template for the class declaration.

A.2.1 Class definition template

Italic words indicate text that should be filled in by the user. '...' denotes "more of these".

```
1   \tt\obeyspaces
2   :- pce_begin_class(\F{Class}(...\F{TermDescriptionArguments}...), \F{Super},
3                      "\F{Documentation}").
4
5   variable(\F{Name},      \F{Type},       \F{Access},  "\F{Documentation}").
6   ...
7
8   resource(\F{Name},      \F{Type},       \F{Default}, "\F{Documentation}").
9   ...
10
11  handle(\F{X_FORMULA}, \F{Y_FORMULA}, \F{Kind}, \F{Name}).
12  ...
13
```

```
14                  /*********************************
15                  *         CREATE/UNLINK         *
16                  *********************************/
17
18  initialise(\F{Self}, ...\F{Arg}:\F{Type}...) :->
19          "Initialise from \F{Arguments}"::
20          \F{CheckArguments},
21          send(\F{Self}, send_super, initialise, ...\F{SuperInitArgs}...),
22          \F{SpecificInitialisation}.
23
24  unlink(\F{Self}) :->
25          "\F{Documentation}"::
26          \F{SpecificUnlink},
27          send(Self, send_super, unlink).
28
29                  /*********************************
30                  *        RESERVED METHODS       *
31                  *********************************/
32
33  event(\F{Self}, Ev:event) :->
34          "\F{Documentation}"::
35          (   send(\F{Recogniser}, event, Ev)
36          ->  true
37          ;   send(\F{Self}, send_super, event, Ev)
38          ).
39
40  ...
41
42                  /*********************************
43                  *        PUBLIC METHODS         *
44                  *********************************/
45
46  \F{Sendmethod}(\F{Self}, ...\F{Arg}:\F{Type}...) :->
47          "\F{Documentation}"::
48          \F{Implementation}.
49
50  \F{Getmethod}(\F{Self}, ...\F{Arg}:\F{Type}..., \F{Result}) :<-
51          "\F{Documentation}"::
52          \F{Implementation}.
53
54
55                  /*********************************
56                  *           UTILITIES           *
57                  *********************************/
58
59  \F{Methods}.
60  ...
61  \F{Prologpredicates}.
62  ...
63
64  :- pce_end_class.
```

A.3 Choosing names

Apart from the Prolog predicates and variables for which any Prolog oriented naming schema applies, various other objects have to be named while using PCE/Prolog. Names for PCE objects have either global scope or local scope to the class they are associated with. Names for classes and global objects are global. Names for selectors, variables and resource are local to their class. For all these names, the following should apply:

- Names with global scope over the entire process should be short when they denote some very basic concept of the application. Otherwise they are best prefixed with some indication of the category they belong to (e.g. `draw_` for all global names related to PceDraw).
- Names with local scope (selectors, variables and resources) have meaningful names. In general they should not be abbreviations. When they denote a general operation, they should be named to this operation (e.g. 'quit', 'relate'). When they denote something very specific, something that can be used only under some non-frequently occuring situation, they should have long names.

A.4 Predicates or methods?

When writing in PCE/Prolog, there is usually the choice between writing a method and invoking this using send[2-12] or `get/[3-13]` or writing Prolog predicates and calling these directly. When using user-defined classes as the basis for structuring an application, the following rules apply:

- Communication between classes defined in different source-files is always using messages. This way the overall structure of the application is based upton one mechanism.
- Within one sourcefile Prolog based activation/calling may be used. This however should be limited to cases where:

  1. Data that is not easily converted to PCE data is to be passed as arguments.
  2. Prolog backtracking should be exploited.
  3. Communication is very time critical.
  4. It implies a private unitily.

A.5 Method arguments

Arguments to methods are determined by there location in the argument vector. PCE distinguishes between obligatory and optional arguments (i.e. arguments that may be `@default`). To avoid having to look in the manual continuously it is necessary to define some standards for argument ordering. The rules used inside PCE have never been stated explicitely and compatibility considerations sometimes leaded to non-intuitive arguments. Below is an attempt to make them explicit.

- Do not use too many obligatory arguments. If possible try to limit the number of obligatory arguments to 1 or 2. 'Name' or similar arguments in general come first. 'Values' (e.g. 'dialog_item $\leftrightarrows$*selection*') come second. If there is a sensible default or the user might not want to specify the value because it will be filled in later, make the argument optional. Example: initialising a line does not require any arguments. Start and end-points default to (0,0). This is useful as (notably for defining links), it is not unlikely the user wishes to create a line and define the start and end-point later.
- Define sensible defaults the optional and order them in decreasing 'likelyness' the user might wish to overrule the default.

A.6 Layout conventions

The considerations for layout of PCE/Prolog programs do not differ very much from those for ordinary Prolog programs. PCE/Prolog programs tend to use deeply nested complex terms, notably while specifying message objects. The normal rules for breaking long terms apply.

Bibliography

[OKeefe, 1990] R. A. OKeefe. *The Craft of Prolog.* MIT Press, Massachussetts, 1990.

[Wielemaker & Anjewierden, 1992a] J. Wielemaker and A. Anjewierden. *PCE-4 Functional Overview.* SWI, University of Amsterdam, Roetersstraat 15, 1018 WB Amsterdam, The Netherlands, 1992. E-mail: jan@swi.psy.uva.nl.

[Wielemaker & Anjewierden, 1992b] J. Wielemaker and A. Anjewierden. *Programming in PCE/Prolog.* SWI, University of Amsterdam, Roetersstraat 15, 1018 WB Amsterdam, The Netherlands, 1992. E-mail: jan@swi.psy.uva.nl.

[Wielemaker, 1992] J. Wielemaker. *PCE-4 User Defined Classes Manual.* SWI, University of Amsterdam, Roetersstraat 15, 1018 WB Amsterdam, The Netherlands, 1992. E-mail: jan@swi.psy.uva.nl.

Index