The Mjølner BETA System
The Bifrost Graphics System
Tutorial

Mjølner Informatics Report
MIA 91-19(1.3)
August 1996

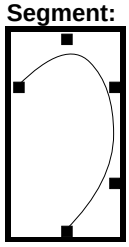
Table of Contents

Introduction

This manual is a short tutorial in the use of version 2.1 of the Bifrost Graphics System. The imaging model of the Bifrost Graphics System is based on the Stencil & Paint metaphor known from, e.g., PostScript™. That is, drawings are defined by constructing a *Stencil*—in Bifrost termed a *Shape*—describing the outline of the drawing, and filling it with a *Paint*. A composition of a Shape and a Paint is known as a *Graphical Object*. Graphical Objects can be combined into composite graphical objects, in Bifrost termed *Pictures*. This is the Bifrost abstraction of what is also known as *Graphics Modelling*.

A Graphical Object can be drawn on a *Canvas*. The Canvas is the Bifrost abstraction of the drawing device, and can be, e.g., a window in a window system. When a Graphical Object is drawn on a Canvas, the Canvas "remembers" the Graphical Object by putting it into a Picture in the Canvas. In a window system this means that the Graphical Object is automatically redrawn when needed, once it has been drawn in the Canvas. The position of a Graphical Object in the list of Graphical Objects in the Canvas Picture determines the "stacking order" of the Graphical Object in the Canvas, i.e., which objects are below and on top of the Graphical Object. The Canvas also allows interaction to be performed on the Graphical Objects, e.g. moving and reshaping as well as highlighting the Graphical Objects in various ways. Graphical Objects can even be created interactively in Canvasses.

A Spline Segment:



A pure *Shape* is composed of two different types of *Segments*: *LineSegments* and *SplineSegments*. LineSegments consist of a begin and an end point. SplineSegments are used for constructing curves. They also have a begin and an end point, but in addition more *control points* may be added. Except for the begin and end point, the spline curve does not go through the control points. Instead the intermediate control points act as small "magnets", that "pull" the curve. For instance, the spline in the margin is composed of three control points in addition to the begin and end points.

To avoid some of the tedious work of defining shapes, Bifrost includes a number of *Predefined Shapes* for the most commonly used Graphical Objects. The Graphical Objects using the Predefined Shapes are termed *Predefined Graphical Objects*. Besides relieving the user from the tedious shape defining work, the Predefined Shapes and Predefined Graphical Objects also allows Bifrost to utilize many of the devices more effectively, since most devices have predefined operations for drawing, e.g., ellipses and text. Some of the Predefined Shapes are *Strokeable Shapes*. Strokeable Shapes have the property, that when used with a Paint in a Graphical Object, instead of being *filled* with the Paint, it can be *stroked*, i.e. outlined, using the Paint and a given stroke width.

The *Paint* describes the color or raster to be pushed through the shape, when the graphical object is displayed on a Canvas. The paint concept in Bifrost supports any kind of pure colors, as well as more sophisticated features such as hatching, tiling, and sampled raster images. These various features of paint can be described in two main paint concepts: solid color and raster paint. A *solid color* fills out the entire shape with one particular color. This concept may be specialized by allowing a repeated pattern, a *tile*, to be applied to the Paint, conceptually by only allowing the paint to reach the canvas where this pattern allows it to. This is a way of obtaining various hatching effects. *Raster paint* uses a Raster to fill the shape. A *Raster* is a rectangular grid of pixel values. In order to use a Raster to fill the Shape it must be specified what to do if the Raster is too small to fill out the entire shape. Bifrost supports two approaches in this case: repeating the Raster over and over again, thus

tiling the interior of the Shape with it, or by using a solid color—called a *padding color*—to fill out any parts of the Shape not covered by the Raster, and thus not filled by the raster image.

For a complete description of the Bifrost Graphics System, including interface descriptions, see the Bifrost Reference Manual [MIA 91-13].

This tutorial presents the development of the graphical interface for a system for designing a subway railroad net for a city. The requirements for this system are as follows:

- A map of the city must be permanently shown in a window.
- Using a mouse, stations can be added by clicking at the place on the map, where it should be.
- Using the mouse, it must also be possible to connect stations with straight lines, illustrating the rails.
- By grabbing a station with the mouse, it must be possible to move the station and the rails connecting it with other stations. During the movement, feedback must be provided.

The presentation will be performed in small steps, so that only a few new major concepts are used in each step. All the programs described in this tutorial can be found on-line in the directory `~beta/bifrost/v2.1/tutorial`. In Appendix A the final program resulting from the tutorial is shown. In Appendix B a screen snapshot of the final application can be found.

1 The Program Skeleton

As described in [MIA91-13] every program using Bifrost must have ORIGIN in or INCLUDE the fragment called `Bifrost`, and execute an instance of the pattern called `bifrost`. The following program skeleton `subway1.bet` will be the starting point:

```
ORIGIN '~beta/bifrost/v2.1/Bifrost';
-- PROGRAM: descriptor --
bifrost
(# theWindow: @window
     (# myCanvas: @BifrostCanvas
          (# open::
               (# do (300, 300)->Size->theWindow.size #);
             eventhandler::
               (# onOpen:: (# (* ... *) #);
                  onMouseDown:: (# (* ... *) #);
                  onKeyDown::
                    (# do (if ch//'Q' then Terminate if) #);
               #);
          #);
        open:: (# do myCanvas.open #)
     #) (* theWindow *)
do theWindow.open;
#)
```

The ORIGIN `~beta/bifrost/v2.1/Bifrost` contains the `bifrost` fragment. As of version 2.1, Bifrost includes the graphical user interface environment Lidskjalv, see [MIA 94-27], so we can start by declaring an ordinary Lidskjalv window to contain the graphics. In the window a static Canvas[1] is declared, in which further binding of various virtuals have been prepared. In `init`, so far, the Canvas is given an arbitrary size. The `onOpen` virtual is called when the `open` procedure pattern has completed, e.g., when the window containing the Canvas has become visible in the window system. From within this virtual pattern, the drawing operations on the Canvas should be performed. The virtual pattern `onMouseDown` is called when a button on the pointing device—if any—has been pressed. So far the actions to perform in this case are unspecified. The virtual pattern `onKeyDown` is called when a keyboard key has been pressed. In this case, it is checked whether the user types ‘Q’ within the Canvas, and if so the application is stopped.

---

[1] In the current Bifrost version the pattern name BifrostCanvas is used when declaring canvasses. This is because of an overlap in names in Lidskjalv and Bifrost. The naming conflict will probably be solved in a later version of Bifrost and Lidskjalv. In the rest of this tutorial we will use the original Bifrost term "Canvas" as a synonym for "BifrostCanvas", but in the coding the latter has to be used for the moment.

2 Adding the City Map

Having the program skeleton set up, the next issue considered, is the map of the city that is to be displayed in the Canvas. One way to display this map is to use an instance of the Predefined Graphical Object Rect, which describes a Rectangle, and then fill it with a TiledSolidColor using the BitMap for tiling. A TiledSolidColor is a solid color, e.g. black, which also has a Bitmap—a so called *tile*. When filling with a TiledSolidColor, only the bits in the tile, which are TRUE, will be painted. The tile will be replicated as many times as needed to fill out the Shape, hence the name.

Given that the bit map of the city map is in the file `~beta/bifrost/v2.1/bitmaps/Aarhus.pbm` in the PBM format discussed in [MIA 91-13, Chapter 4], the TiledSolidColor may be declared as follows in `subway2.bet`:

```
aarhus: @TiledSolidColor
      (# bits: @bitMap
            (#
            do 'Reading in map ... ' -> puttext;
               '~beta/bifrost/v2.1/bitmaps/Aarhus.pbm'
                 -> readFromPBMfile;
               'done' -> putline;
               (0,0) -> hotspot;
            #);

         init::
            (#
            do black -> name;
               bits -> thetile;
            #)
      #);
```

The BitMap is read from the PBM file using `ReadFromPBMFile`. This may take a while, and to inform the user of what is going on, some explanatory text is written out on the screen. The hotspot of the BitMap is set to (0,0). This is to control the placement of the BitMap within the Rect, that it will be used to fill out.

In the TiledSolidColor, `init` simply evaluates the BitMap, and assigns the result of the evaluation—which is a reference to the BitMap—to the `theTile` attribute of the TiledSolidColor. It also evaluates the pattern `black` and assigns the result to the `name` attribute. This will make the bits of the tile which are set appear black on the screen. The `black` pattern is declared in the fragment `~beta/bifrost/v2.1/ColorNames` which must be INCLUDEd in `subway2.bet`.

With this definition of the TiledSolidColor, the Rect may then be declared as follows:

```
map: @Rect
   (#
      init::
         (#
         do (* Make THIS(Rect) the size of the aarhus-bitmap *)
            aarhus.init;
            (0,aarhus.bits.height) -> upperleft;
            aarhus.bits.width -> width;
            aarhus.bits.height -> height;
            aarhus[] -> setpaint;
         #)
    #);
```

In the Rect, `init` first initializes the TiledSolidColor called `aarhus` which in turns reads the bitmap from the PBM file as described above. Then `upperleft`, `width` and `height` are set using the dimensions of the BitMap. Finally a reference to the TiledSolidColor is given to the `setpaint` attribute of the Rect, thus making `aarhus` be used as the paint to fill the Rect with. The shape of the Rect has a default hotspot in the upper left corner. When a graphical object is filled with a TiledSolidColor, the hotspot of `theTile` of the TiledSolidColor is placed in the hotspot of the Shape of the graphical object, and `theTile` is then replicated as needed to fill out the entire Shape. Thus having defined the hotspot of `aarhus.bits` to (0,0) and made the Rect have the exact dimensions of the BitMap, the BitMap will be shown exactly once within the Rect.

All that is needed now is to initialize and draw the Rect. It is initialized in `myCanvas.open`:

```
open::
  (#
  do (* Make THIS(BifrostCanvas) the size of the map *)
     map.init;
     (map.width, map.height)->Size->theWindow.size;
  #);
```

Instead of the arbitrary size used in `subway1.bet`, the Canvas is given the same dimensions as the Rect containing the BitMap. Now the Rect is ready to be drawn when the Canvas has become visible on the screen. Thus `myCanvas.onOpen` becomes:

```
onOpen:: (# do map[] -> draw; #);
```

The predefined graphical object Rect is defined in a separate fragment, `~beta/bifrost/v2.1/PredefinedGO`, which must be INCLUDEd in `subway2.bet`.

3 Making Stations

The next step is to create stations on top of the map in the window. First a Station pattern is declared. One way to present the stations is by a filled circle with a letter centered inside, denoting the station, as shown in the margin. Since the graphical object representing a station thus contains several other graphical objects, it is declared as a Picture in `subway3.bet`:

```
Station: Picture
  (# name: @text;
     label: @GraphicText;
     position: @point;
     Circle: Ellipse
       (# radius: @
            (# r: @integer
            enter (#
                  enter r
                  do r->horizontalradius->verticalradius;
                  #)
            exit r
            #);
       #);

     filledcircle, circleoutline: @Circle;

     init::
       (# ch: @char;
          r: @rectangle;
          c: @point;
       enter (position, ch)
       do (* Initialize filledcircle *)
          filledcircle.init;
          10->filledcircle.radius;
          position->filledcircle.center;
          fill[]->filledcircle.setPaint;

          (* Initialize circleoutline *)
          circleoutline.init;
          11->circleoutline.radius;
          position->circleoutline.center;
          true->circleoutline.theshape.stroked;
          2->circleoutline.theshape.strokewidth;
          color[]->circleoutline.setPaint;

          (* Center the label within the circles *)
          ch->name.put;
          label.init;
          (position, Times, Bold, 20, false, name[])
            -> label.inittext;
          label.getbounds->r;
          (r.x+(r.width) div 2, r.y-(r.height+1) div 2)->c;
          (circleoutline.center,c)->subpoints->label.move;
          color[]->label.setPaint;
```

```
            (* Add circles and label to THIS(Picture) *)
            filledcircle[]->add;
            circleoutline[]->add;
            label[]->add;
          #);
      #);
```

In a Station, `init` enters the position to display the station, and a character to display as a label in it. The Picture is actually made up of *two* circles, a stroked one displaying the outline of the Station and a filled one for the background. To simplify matters a Circle pattern is defined; all it does is to extend the Ellipse pattern with a `radius` attribute for setting `horizontalradius` and `verticalradius` to the same value. Notice the use of a pattern in the enter part of `radius`. This is to assure that `horizontalradius` and `verticalradius` are only changed when a value is *entered* to `radius`, not when it is only used to *exit* its value. The filled Circle uses a Paint called `fill`, and the stroked one uses a Paint called `color`. These are both SolidColors

```
color: @SolidColor;
  (* The color used for stations and rails *)
fill: @SolidColor;
  (* The color used to fill station backgrounds *)
```

and are initialized in `open` in `theWindow`, just before the Canvas is opened:

```
open::
  (#
  do (* Initialize colors for Stations and Rails *)
     color.init; IndianRed->color.name;
     fill.init;  PaleGreen->fill.name;
     (* Initialize and open the BifrostCanvas *)
     myCanvas.open
  #)
```

To display the label inside the circles, a GraphicText is used. This Predefined Graphical Object has an attribute for the text to display. Thus the first thing to do is to put the char `ch` into a text `name`. Then the GraphicText is initialized, its attributes are set using `inittext`, and it is centered within the circles by moving it a certain distance. This distance is calculated using the bounding box of the GraphicText. The `subpoints` pattern simply exits the coordinatewise difference between the two points entered. The GraphicText is given the same paint as the outline of the Station.

Finally, the three graphical objects just defined and initialized are added to the Picture, and the Station is ready to be displayed.

To create a Station and display it when the mouse is pressed in the window, the `onMouseDown` virtual is extended:

```
onMouseDown::
  (#
  do (* Transform mousepos to BifrostCanvas coordinates *)
     mousepos->devicetocanvas->mousepos;
     mousepos->makeStation;
  #);
```

Local to `onMouseDown` is a Point called `mousepos`, which is the position of the mouse in the window at the time of the button press. This position is reported in device coordinates, i.e., screen coordinates, so in this case, the first thing to do is to transform `mousepos` to Canvas coordinates using the `devicetocanvas` attribute of the Canvas pattern. Once this is done, an item called `makeStation` is invoked to create the Station.

```
makeStation: @
   (# pos: @point;
      aStation: ^station;
      ch: @char;
   enter pos
   do &station[] -> aStation[];
      (pos, ch) -> aStation.init;
      ch+1 -> ch;
      aStation[] -> draw;
   #);
```

`makeStation` is a static item because it must remember the `ch` attribute, which is used as the label of the station being created. All it does is to instantiate a Station, increment `ch` by one (such that the next Station created will be labelled with the next letter in the alphabet), and finally draw the Station just created in `myCanvas`. As mentioned in the introduction, this means that the Station will now be the front most graphical object in the Picture used by the Canvas to "remember" its Graphical Objects. To make the first Station appear with the right label, `makeStation.ch` must be initialized. An appropriate place to do this is in `open` of `myCanvas`, which then becomes

```
open::
  (#
  do (* Make THIS(BifrostCanvas) the size of the map *)
     map.init;
     (map.width, map.height)->Size->theWindow.size;
     (* The first Station will have label "A" *)
     'A'->makeStation.ch;
  #);
```

Exercise: The above solution for makeStation limits the number of Stations to the number of characters in the alphabeth. Describe a solution without this limitation.

4 Moving Stations

Having created the Stations, the next task is to make it possible to move them around. The moving of a Station is simple to do, since a Station is a Picture which in turn is a specialization of AbstractGraphicalObject. A reference to an AbstractGraphicalObject can be entered to the `interactivemove` attribute of `myCanvas`, that does exactly what is needed in this case. Thus the only thing to do is to change `onMouseDown` of `myCanvas` to handle the situation right. If the mouse is pressed outside any existing Stations, a new one must be created as above. If the mouse is pressed inside an existing Station, then an interaction for moving this station should be started. One way to find out if anything was hit is by scanning through all graphical objects in the Picture of myCanvas, and for each graphical object check whether the mouse position is contained within this graphical object. An exception has to be made for the city map, which must never be moved around. In `subway4.bet`, `onMouseDown` becomes

```
onMouseDown::
  (#
  do (* Transform mousepos to BifrostCanvas coordinates *)
     mousepos->devicetocanvas->mousepos;

     scan:
       (# (* Find out what was hit - if any *)
       do thePicture.ScanGOsReverse
          (#
          do (if (myCanvas[], mousepos)->go.containspoint then
                 (if go[]
                  // map[] then (* ignore *)
                  else
                     (* We hit a Station: Move it *)
                     (go[], mousepos, NoModifier)
                       -> interactivemove;
                     leave scan
                 if);
             if);
          #);
          (* Nothing was hit *)
          mousepos->makeStation;
       #);
  #);
```

The Picture used to hold the graphical objects of `myCanvas` is called `thePicture`. The graphical objects in `thePicture` are scanned using the control pattern `ScanGOs-Reverse`. This control pattern scans through the objects in the Picture downwards from the top (frontmost) graphical object. Using `go.containspoint` it is checked if the mouse was inside the graphical object, and if so, the scanning is stopped, except if the graphical object was the map. If a Station was "hit", a reference to it is passed to `interactivemove`. The Modifier parameter to `interactivemove` is used to specify what keyboard modifiers should be used to constrain the interaction. In this case `No-Modifier` is specified, meaning that the interaction cannot be constrained. If no Station was hit, `makeStation` is invoked as before.

5 Making Rails

As mentioned earlier, a Rail should appear as a straight line between the two Stations it connects. Thus, in `subway5.bet`, the pattern Rail is as follows

```
Rail: Line
   (# init:: (# do color[] -> setpaint; 2 -> width; #) #);
```

It is simply a Line using the same Paint as is used for the outline of Stations, and which has a width of 2 pixels.

Then comes the question of creating the rails. Again `onMouseDown` has to be changed.

```
onMouseDown::
  (#
  do mousepos->devicetocanvas->mousepos;

     scan:
       (# (* Find out what was hit - if any *)
       do thePicture.ScanGOsReverse
          (#
          do (if (myCanvas[], mousepos)->go.containspoint then
                 (if go[]
                  //map[] then (* ignore *)
                  else
                     (if go##=Station## then
                         (* We hit a station *)
                         (if shiftmodified then
                             go[]->interactiveCreateRail;
                          else
                             (go[],mousepos,NoModifier)
                               -> interactivemove;
                         if);
                         leave scan
                     if);
                 if);
             if);
          #);
          (* No Station was hit *)
          mousepos->makeStation;
       #);
  #);
```

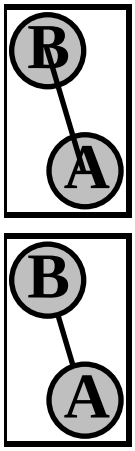
Firstly, in the scanning of graphical objects, now three possibilities exist for what can be hit: Either the map, a Station, or a Rail. Once again hitting the map is taken as if nothing was hit, but if something else was hit, it must be determined what kind of graphical object it was. This is done by comparing the structure reference of `go` with the pattern reference `Station##`. If a Station has been hit, it must be determined whether it should be moved as before, or interaction for connecting it with another Station should be started. Here it is chosen to start connecting it with another Station, if the mouse button press was modified by holding down the SHIFT key, or otherwise to move the Station as before.

Exercise: The structure comparison can be a fairly expensive operation in BETA, and some would say, that using such tests for types, is not a truly object-oriented programming style. Write a variant of `onMouseDown`, that does not use structure comparisons.
Hint: Declare a pattern `HitPicture` with a virtual `onHit`, and let both `Station` and `Rail` be specializations of `HitPicture`.

The creation of a Rail, connecting the Station with another one, is accomplished using an item called `interactiveCreateRail`:

```
interactiveCreateRail: @
   (# r: ^rail;
      hitstation, otherstation: ^Station;
      ......
   enter hitstation[]
   do &rail[] -> r[];
      r.init;
      (r[], hitstation.position, NoModifier)
        -> interactiveCreateShape;

      (* Check if r ends in another station *)
      scan: thePicture.ScanGOsReverse
         (#
         do (if (myCanvas[], r.end) -> go.containspoint then
                (if go[]
                 // map[]
                 // hitstation[] then (* ignore *)
                else
                    (if go##=Station## then
                        (* r ends in another station;
                         * connect with hitstation
                         *)
                        go[] -> otherStation[];
                        ......

                        r -> draw;
                        (* It looks better if the stations
                         * cover the ends of the rail.
                         * Instead of lowering the rail in the
                         * BifrostCanvas (which would put the rail
                         * behind the map) we raise the two
                         * stations
                         *)
                        hitstation[] -> bringForward;
                        otherstation[] -> bringforward;
                        leave scan
                     if);
                 if);
            if);
          #);
    #);
```

The first thing to do is to instantiate and initialize a Rail. Then a reference to this Rail is passed to `interactiveCreateShape`, with the position of `hitStation` as starting point, i.e. the feedback, which in this case is a "rubber line", will begin in `hitStation.position`. The feedback is ended when the user releases the mouse button.



Then it must be determined if the mouse was released on top of another Station. This is done in the same fashion as the way it was determined if a Station was hit in `onMouseDown`. If the mouse was released on top of another Station, a reference to this Station is kept in `otherStation`. In this case, first the two Stations must be connected; this issue is considered below. Then the Rail must be drawn, and finally a few rearrangements are done to improve the appearance of the connection.

Instead of leaving the Rail on top of the two Stations after it is drawn, it looks nicer if the Rail is moved behind the Stations, see the illustration in the margin. One way of doing this is to use `SendBehind` for the Rail. However, this will put the Rail to the very back of the Picture of `myCanvas`, and would thus put the Rail behind the map too. Instead the two Stations can be brought to the top of the Picture of myCanvas using `BringForward`.

What is left is the question of how to make the two Stations know that they are connected with a Rail. At least they need to know this in order to change the Rails when the Stations are moved; this issue is considered in the next section. Since the rails are to be considered bi-directional, both Stations connected by a Rail have to know about the Rail, and where the other Station is. To accomplish this, each Station will have a list of references to Rails that connects it with other Stations. Furthermore references to the two end points of each Rail is kept in the list. These references are qualified with Rail.theShape.invalidatePoint, which is an ordinary Point, with the exception that changing it will cause some recalculations of the Shape of the Rail (the Shape is "invalidated"). This is used internally by Bifrost, e.g. to control how often the bounding box needs to be recalculated. Using the standard List from the BETA library (automatically INCLUDEd by Bifrost), this can be expressed as follows:

```
Station: Picture
   (# ......
      rails: @list
         (# element::
               (# r: ^rail;
                  mypoint,
                  otherpoint: ^r.theshape.invalidatepoint;
               #);
         #);
      ......
   #);
```

Using this data structure, the issue of connecting the two Stations in `interactiveCreateRail` can be solved as follows:

```
interactiveCreateRail: @
   (# r: ^rail;
      hitstation, otherstation: ^Station;
      e: ^hitstation.rails.element;
   enter hitstation[]
   do ... instantiate, init and interactively create r,
          as above ...
      (* Check if r ends in another station *)
      scan: thePicture.ScanGOsReverse
         (#
         do (if (myCanvas[], r.end) -> go.containspoint then
                (if go[]
                 // map[]
                 // hitstation[] then (* ignore *)
                else
                    (if go##=Station## then
                        (* r ends in another station;
                         * connect with hitstation
                         *)
                        go[] -> otherstation[];
                        otherstation.position
                          -> r.end; (* Small adjustment *)
```

```
                        (* Add r to hitstation and
                         * otherstation.
                         *)
                        &hitstation.rails.element[] -> e[];
                        r[] -> e.r[];
                        r.theshape.begin[] -> e.mypoint[];
                        r.theshape.end[]   -> e.otherpoint[];
                        e[] -> hitstation.rails.append;
                        &otherstation.rails.element[] -> e[];
                        r[] -> e.r[];
                        r.theshape.end[]   -> e.mypoint[];
                        r.theshape.begin[] -> e.otherpoint[];
                        e[] -> otherstation.rails.append;

                        ... draw r and rearrange as above ...
                        leave scan
                    if);
                if);
            if);
         #);
      #);
```

First a small adjustment is done: The mouse may have been released anywhere within `otherStation`. It looks better if the Rail goes through the center of `otherStation`, so the end point of `r` is changed to `otherStation.position`, which was initialized to the center of `otherStation` when `otherStation` was initialized. However, both `hitStation` and `otherStation` may have been moved since they were initialized, so the use of `hitStation.position` and `otherStation.position` as above is not correct if something else isn't done. What needs to be done is to update the `position` attribute each time a Station is moved. This can be done in the `move` virtual, which is called by `interactiveMove`. The easiest way to update `position` is to use the center of one of the two circles constituting the Station and applying the transformation matrix `TM`[2] of the Station. Notice that all moving, scaling and rotating of a graphical object is remembered by changing `TM`, not by changing the coordinates of the defining points of the shape of the graphical object.

```
Station: Picture
   (#
      ......
      move::
         (#
         do (* Move is called by interactiveMove *)

            (* TM describes the current transformation.
             * Make position be the *transformed* position
             *)
            circleoutline.center
              -> TM.transformpoint
              -> position;
         #);
      .....
   #);
```

Now the Stations can be connected with Rails, but one problem remains: If a Station is moved after some Rails have been added to it, the Rails will not move with the Station. In the next section this problem will be addressed.

---

2 The use of 3×3 matrices for describing geometrical transformations is described in detail in [MIA 91-13], Chapter 2.

6 Moving Rails

When moving the Stations, there are two things to do to make the Rails "stick" to the Station being moved: During the move, feedback of the Rails connecting the Station with other Stations, must be shown, and after the Station has been moved, the Rails must be physically moved too.

When a Station is moved by `InteractiveMove`, Bifrost automatically supplies feedback during the move. This is done by repeatedly executing an instance of the `HiliteOutline` virtual of the shape of the Station. When calling `HiliteOutline`, the drawing is done in XOR mode, meaning that drawings may be erased simply by drawing again. A Station is a Picture, and thus feedback for all graphical objects that have been added to this Picture is automatically provided. The Rails belonging to a Station, however, is not added to the Picture, and thus feedback for the rails is not supplied automatically. In `subway6.bet`, by further binding the `HiliteOutline` virtual of the Station pattern, the extra feedback is easily supplied:

```
Station:
   (#
      ......

      shapedesc::
         (#
            hiliteoutline::
               (#
               do rails.scan
                     (#
                     do (if TM[]//NONE then
                            (* No transformation *)
                            (current.otherpoint -> CanvasToDevice,
                             position -> CanvasToDevice)
                            -> immediateline;
                        else
                            (current.otherpoint
                               -> CanvasToDevice,
                             position -> TM.transformpoint
                               -> CanvasToDevice)
                              -> immediateline;
                        if);
                     #);
               #);
         #);
    #)
```

`HiliteOutline` enters a reference to a Matrix called `TM`. If this reference is not NONE, it describes the transformation to apply to the feedback. E.g., if the Station has been moved a distance, `TM` will describe this translation. If the reference is NONE, no transformation is to be applied.

The feedback for a Rail will be a simple line from the Position of the Station in question to the other end point of the Rail. To draw the actual feedback the "immediate" drawing operations of Canvas are used. These operations draw directly into the Canvas, without adding the drawing to the "memory" of the Canvas. The operations expect device coordinates. This is the reason why the coordinates are transformed by `CanvasToDevice` before being entered to `immediateline`. Instead of using `position`

as the first end point of the immediate line, of course `current.mypoint` could have been used. Since the Station at the other end of the Rail considered is not moved, TM is only applied to the position of the Station being moved. Thus the feedback will be a "rubber line" stuck to the two Stations it connects.

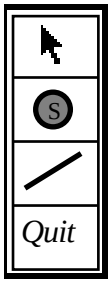
When the interaction is finished in `InteractiveMove`, as mentioned in section 5, it calls the `move` virtual to actually move the graphical object considered. Again, since the Rails are not added to the Picture, `move` will not change the Rails, when the Picture is moved by `InteractiveMove`. Instead, this may be accomplished by further binding `move`:

```
Station:
   (#
      ......

      move::
         (#
         do (* Move is called by interactiveMove.
             * Furtherbind to move the rails too
             *)

            ...

            rails.scan
               (#
               do current.r.getbounds -> damaged;
                  position -> current.mypoint;
                  (* Changes either current.r.begin or
                   * current.r.end
                   *)
                  current.r.getbounds -> damaged;
               #);
         #);

      ......
   #)
```

In `rails.scan`, `current.r` is a reference to one of the rails connecting the Station with other Stations. First the bounding box of the rail is reported to `myCanvas` as being *damaged*. Then the newly transformed `position` is entered as the new end point of the Rail, belonging to the Station considered. Of course the other end point of the Rail should not be changed – this is also the reason why the Rails are not *added* to the Picture (the Station), since all members of a Picture are moved rigidly when the Picture is moved.

After the Rail has been changed, the new bounding box of it is also reported as being damaged. After `InteractiveMove` has called `move`, it also calls `myCanvas.repair`. This will make all damaged areas of `myCanvas` be redrawn, and since the areas the Rails have covered are now marked as damaged, this will complete the moving of the rails too.

7 Adding a Palette



To illustrate the use of multiple Canvasses, the application can be elaborated a bit: Some users may find it annoying to remember to hold down the Shift modifier to be able to create rails. Except for the Shift-modification, the application as described above is "mode-less", i.e., a given user action always results in the same action in the application. As an alternative, the application can be changed, so that it may be put into different modes. To control what is the current mode, a Palette can be put into a separate window. This Palette could look like the one shown in the margin. The item with the arrow, indicates "Move-mode", where all that can be done is moving the Stations around. The second item contains a Station, and denotes "Station-mode", where all that can be done is creation of Stations. The third item contains a Rail, and denotes "Rail-Mode", where all that can be done is creation of Rails. Finally the last item is used to quit the application, using the mouse. In the Bifrost library, a pattern exists, which is intended to make such palettes out of graphical objects. The pattern is in `~beta/bifrost/v2.1/Palette`, which the application thus needs to INCLUDE. Then, in `subway7.bet`, the Palette may be specified in the following way:

```
(* Constants corresponding to palette selections *)
MoveMode:    (# exit 1 #);
StationMode: (# exit 2 #);
RailMode:    (# exit 3 #);
Quit:        (# exit 4 #);

PaletteWindow: @window
  (# Mode: @Palette
       (# changed::
            (#
            do (if selection//Quit then Terminate if);
            #);
          open::
            (# G: @GraphicalObject;
               T: @GraphicText;
               L: @myCanvas.Rail;
               S: @myCanvas.Station;
            do G.init;
               (  0, 32)->G.theShape.open;
               (  0, 10)->G.theShape.lineto;
               (  5, 15)->G.theShape.lineto;
               ( 11,  0)->G.theShape.lineto;
               ( 15,  2)->G.theShape.lineto;
               (  9, 16)->G.theShape.lineto;
               ( 16, 16)->G.theShape.lineto;
               G.theShape.close;
               blackpaint[]->G.setpaint;
               G[]->append;

               ((0,0), 'S')->S.init;
               S[]->append;

               L.init;
               ((0,0), (50,30))->L.coordinates;
               L[]->append;
```

```
                T.init;
                ((100,20),Helvetica,Italic,20,false,'Quit')
                  ->T.inittext;
                blackpaint[]->T.setpaint;
                T[]->append;

                StationMode->selection;
                (size, (4,4)) -> AddPoints -> palettewindow.size;
              #);
        #) (* Mode *);
      open:: (* PaletteWindow *)
        (# s: @point;
        do hide (* initially invisible *);
           false->paletteOpen;
           (NONE, 80, 50, true)->Mode.open;
           'Mode'->title;
           myCanvas.size->s;
           (myCanvas.position, (s.x, 0)) ->AddPoints -> position;
        #);
    #);
```

First a standard Lidskjalv window called `PaletteWindow` is created to contain the Palette. In the Palette, which is called `Mode`, `open` sets up the contents of the Palette. `Mode.open` is called from `PaletteWindow.open`. It enters a position – (10, 10) –, the size of each field in the Palette – 80 times 50 – and finally a boolean specifying if the sequence of graphical objects should be placed next to each other (a horizontal Palette), or below each other (a vertical Palette), which is chosen here. The window is given a title "Mode", and is placed next to the window containing the subway system.

In `Mode.open` the four graphical objects are specified and appended: First the arrow is made as an instance of GraphicalObject. In a GraphicalObject, the Shape may be directly manipulated. Here the shape definition language is used to add the line Segments, that define the outline of the arrow. Then `blackpaint`, which is a local attribute of Palette, is specified as the Paint, and finally the GraphicalObject is appended to the Palette. Likewise instances of Station, Rail, and GraphicText are initialized and appended. Finally the initial selection of the Palette is set. The `selection` attribute of a Palette is an integer holding the number of the currently selected item of the Palette. The number of an item is set to the current number of items after the item has been appended to the Palette. To improve the readability of the program four constant patterns have been defined, exiting the integers corresponding to the selection of the Palette.

The Palette must then be opened. An appropriate place to do this is in when opening the main window:

```
open::
  (#
  do (* Initialize colors for Stations and Rails *)
     color.init; IndianRed->color.name;
     fill.init;  PaleGreen->fill.name;
     (* Open the BifrostCanvas *)
     myCanvas.open;
     (* Open the Palette *)
     palettewindow.open;
    'Type ''P'' to open the Palette'->putline;
  #)
```

Notice, that because `hide` is called in `PaletteWindow.open`, the window is not immediately *shown* when opened: It should be possible to open and close the Palette when the user wants to. The application should behave like before when the Palette is not shown, i.e., Rails are created by Shift-clicking a Station and there are no other modes. But when the Palette is shown, it should define the current mode, as described above. The state of the Palette can be toggled when the user types a 'P':

```
paletteOpen: @boolean;

onKeyDown::
  (#
  do (if ch
      //'Q' then Terminate
      //'P' then
         (if not paletteOpen then
             palettewindow.show;
          else
             palettewindow.hide;
         if);
         not paletteopen -> paletteopen;
     if)
  #);
```

The current state of the Palette, i.e., shown or hidden, is determined by the boolean `paletteOpen`. This boolean is then used in `onMouseDown` to determine what to do.

```
onMouseDown::
  (#
     (* Actions for Stations *)
     StationAction: (# s: ^Station enter s[] do INNER #);
     MoveIt: StationAction
       (# do (s[],mousepos,NoModifier)->interactivemove #);
     MakeRail: StationAction
       (# do s[]->interactiveCreateRail #);

     (* Control pattern for finding a station and
      * performing an action on it.
      *)
     findStation:
       (# s: ^Station;
          action: ##StationAction;
       enter action##
       do (* Find out what was hit - if any *)
          scan: thePicture.ScanGOsReverse
            (#
            do (if (myCanvas[], mousepos)->go.containspoint then
                   (if go[]
                    // map[] then (* ignore *)
                    else
                       (if go##=Station## then
                           (* We hit a station *)
                           go[]->s[];
                           (if action##<>NONE then s[]->action if);
                           leave scan
                       if);
                    if);
                if);
            #);
       exit s[]
       #);
```

```
    hitstation: ^Station;

do mousepos->devicetocanvas->mousepos;

    (if paletteOpen then  (* Palette determines mode *)
        (if palettewindow.mode.selection
         // MoveMode then MoveIt##->findStation;
         // StationMode then mousepos->makeStation;
         // RailMode then MakeRail##->findStation;
         if);
     else (* Mode-less *)
         (if findStation->hitstation[]
          // NONE then
             mousepos->makeStation;
          else
             (* We hit a station *)
             (if shiftmodified then hitstation[]->MakeRail;
              else hitstation[]->MoveIt;
             if);
          if);
     if);
#);
```

If the Palette is open, as mentioned, `Mode.selection` determines the current mode and otherwise the behavior should be as before. Because of the more complex control structure, a slightly different approach is taken: There are several places in the code, where it should be known whether a Station was hit or not. A general control pattern `findStation` is defined. This will search for a Station that is hit, in the same way as before. If a hit Station is found, an action can be performed on it. This action is specified to `findStation` by using a pattern reference. Using `findStation`, the control structure becomes much shorter:

- In "Move-Mode" all button presses on Stations leads to an invocation of InteractiveMove, regardless of the state of the Shift-modifier.
- In "Station-Mode" all mouse presses leads to creation of a Station, regardless of if the button press was on top of another Station. There is no need to know if a Station was hit or not.
- In "Rail-Mode", all button presses on Stations start the interaction for creating a Rail, regardless of the state of the Shift-modifier.

If the Palette is not open, the behavior should be as before. As can be seen the old behavior can also be specified using `findStation` and the action patterns.

The use of pattern variables for the action in `findStation` could just as well be changed to normal object references (qualified by `StationAction`). This is mostly a matter of taste.

8 Further Readings

This completes the tutorial of Bifrost. As mentioned in the introduction, all steps of this tutorial may be found on-line in `~beta/bifrost/v2.1/tutorial/`.

Although a lot of the major concepts have been covered, many topics have not been presented. You might want to look in the files in `~beta/bifrost/v2.1/demo/` for some other examples of using Bifrost.

Also you might want to consult the Reference Manual [MIA 91-13], which contains a thorough survey of the Bifrost Graphics System, including a complete description of the programming interface and a graphical presentation of the modular design of Bifrost, i.e., the fragment structure.

The design criteria that lead to the current design of Bifrost, and some implementation details are presented in a thesis [Andersen 91].

Some of the demos, or one of the programs from the tutorial may be good starting points to do your own experiments with Bifrost.

Appendix A — The Final Program

```
This is subway7.bet:

ORIGIN '~beta/bifrost/v2.1/Bifrost';
INCLUDE '~beta/bifrost/v2.1/PredefinedGO';
INCLUDE '~beta/bifrost/v2.1/Palette';
INCLUDE '~beta/bifrost/v2.1/ColorNames';

-- PROGRAM: descriptor --

bifrost
(# theWindow: @window
     (# color: @
          (* The color used for stations and rails *)
          SolidColor;
        fill: @
          (* The color used to fill station backgrounds *)
          SolidColor;

        paletteOpen: @boolean;

        aarhus: @TiledSolidColor
          (# bits: @bitMap
               (#
               do 'Reading in map ... '->puttext;
                  '~beta/bifrost/v2.1/bitmaps/Aarhus.pbm'
                    ->readFromPBMfile;
                  'done'->putline;
                  (0,0)->hotspot;
               #);

             init::
               (#
               do black->name;
                  bits->thetile;
               #)
          #);

        (* Constants corresponding to palette selections *)
        MoveMode:    (# exit 1 #);
        StationMode: (# exit 2 #);
        RailMode:    (# exit 3 #);
        Quit:        (# exit 4 #);

        PaletteWindow: @window
          (# Mode: @Palette
               (# changed::
                    (#
                    do (if selection//Quit then Terminate if);
                    #);
                  open::
                    (# G: @GraphicalObject;
                       T: @GraphicText;
                       L: @myCanvas.Rail;
```

```
                  S: @myCanvas.Station;
               do G.init;
                  (  0, 32)->G.theShape.open;
                  (  0, 10)->G.theShape.lineto;
                  (  5, 15)->G.theShape.lineto;
                  ( 11,  0)->G.theShape.lineto;
                  ( 15,  2)->G.theShape.lineto;
                  (  9, 16)->G.theShape.lineto;
                  ( 16, 16)->G.theShape.lineto;
                  G.theShape.close;
                  blackpaint[]->G.setpaint;
                  G[]->append;

                  ((0,0), 'S')->S.init;
                  S[]->append;

                  L.init;
                  ((0,0), (50,30))->L.coordinates;
                  L[]->append;

                  T.init;
                  ((100,20),Helvetica,Italic,20,false,'Quit')
                    ->T.inittext;
                  blackpaint[]->T.setpaint;
                  T[]->append;

                  StationMode->selection;
                  (size, (4,4)) -> AddPoints
                    -> palettewindow.size;
               #);
         #) (* Mode *);
      open::
         (# s: @point;
         do hide (* initially invisible *);
            false->paletteOpen;
            (NONE, 80, 50, true)->Mode.open;
            'Mode'->title;
            myCanvas.size->s;
            (myCanvas.position, (s.x, 0)) -> AddPoints
              -> position;
         #);
   #);

myCanvas: @BifrostCanvas
   (#
      map: @Rect
         (#
            init::
              (#
              do (* Make THIS(Rect) the size of the
                  * aarhus-bitmap
                  *)
                 aarhus.init;
                 (0,aarhus.bits.height)->upperleft;
                 aarhus.bits.width->width;
                 aarhus.bits.height->height;
                 aarhus[]->setpaint;
              #)
         #);

      Rail: Line
         (# init:: (# do color[]->setpaint; 2->width; #) #);

      Station: Picture
         (# name: @text;
            label: @GraphicText;
```

```
position: @point; (* Transformed position *)
Circle: Ellipse
  (# radius: @
       (# r: @integer
       enter (#
             enter r
             do r->horizontalradius
                 ->verticalradius;
             #)
       exit r
       #);
  #);

filledcircle, circleoutline: @Circle;

rails: @list
  (#
     element::
       (# r: ^rail;
          mypoint,
          otherpoint: ^r.theshape.invalidatepoint;
       #);
  #);

move::
  (#
  do (* Move is called by interactiveMove.
      * Furtherbind to move the rails too
      *)

     (* TM describes the current
      * transformation.  Make position be the
      * *transformed* position
      *)
     circleoutline.center
       -> TM.transformpoint
       -> position;

     (* Since the rails are not members of
      * THIS(Picture), they are not updated by
      * interactivemove.  We check what areas
      * they "damage", and the call of "repair"
      * that interactivemoveperforms will take
      * care of updating these areas.
      *)
     rails.scan
     (#
     do current.r.getbounds->damaged;
        position->current.mypoint;
        (* Changes either current.r.begin or
         * current.r.end
         *)
        current.r.getbounds->damaged;
     #);
  #);

shapedesc::
  (#
     (* Picture.InteractiveMove uses
      * hiliteoutline to draw/erase the feedback
      * for all members of the Picture.  Here we
      * furtherbind the descriptor for
      * THIS(PictureShape) to erase / draw
      * feedback for the rails too
      *)
```

```
        hiliteoutline::
          (#
          do (* TM is a transformation to apply
              * before the highlighting. In this
              * case it's just a translation, and
              * this translation is only to be
              * used for the endpoint of the rail
              * belonging to THIS(Station).
              * Hiliteoutline is called multiple
              * times by InteractiveMove to draw
              * and erase the feedback. The
              * drawing is automatically performed
              * in XOR-mode, i.e., the immediate
              * line is erased simply by drawing
              * it again.  This is the reason why
              * there is no check to see if the
              * line is to be drawn or erased
              *)
             rails.scan
             (#
             do (if TM[]//NONE then
                    (* No transformation *)
                    (current.otherpoint
                      ->CanvasToDevice,
                    position->CanvasToDevice)
                      -> immediateline;
                 else
                    (current.otherpoint
                      ->CanvasToDevice,
                    position
                      ->TM.transformpoint
                      ->CanvasToDevice)
                      -> immediateline;
                if);
             #);
             (* Notice that immediateline expects
              * device coordinates
              *)
          #);
      #);

    init::
      (# ch: @char;
         r: @rectangle;
         c: @point;
      enter (position, ch)
      do (* Initialize filledcircle *)
         filledcircle.init;
         10->filledcircle.radius;
         position->filledcircle.center;
         fill[]->filledcircle.setPaint;

         (* Initialize circleoutline *)
         circleoutline.init;
         11->circleoutline.radius;
         position->circleoutline.center;
         true->circleoutline.theshape.stroked;
         2->circleoutline.theshape.strokewidth;
         color[]->circleoutline.setPaint;

         (* Center the label within the circles *)
         ch->name.put;
         label.init;
         (position, Times, Bold, 20, false, name[])
           -> label.inittext;
         label.getbounds->r;
```

```
            (r.x+(r.width) div 2,
             r.y-(r.height+1) div 2)->c;
            (circleoutline.center,c)->subpoints
              ->label.move;
            color[]->label.setPaint;

            (* Add circles and label to THIS(Picture) *)
            filledcircle[]->add;
            circleoutline[]->add;
            label[]->add;
         #);
    #);

makeStation: @
    (# pos: @point;
       aStation: ^station;
       ch: @char;
    enter pos
    do &station[]->aStation[];
       (pos, ch)->aStation.init;
       ch+1->ch;
       aStation[]->draw;
    #);

interactiveCreateRail: @
    (# r: ^rail;
       hitstation, otherstation: ^Station;
       e: ^hitstation.rails.element;
    enter hitstation[]
    do &rail[]->r[];
       r.init;
       (r[], hitstation.position, NoModifier)
         -> interactiveCreateShape;

       (* Check if r ends in another station *)
       scan: thePicture.ScanGOsReverse
         (#
         do (if (myCanvas[], r.end) -> go.containspoint
             then
                (if go[]
                 //map[]
                 //hitstation[] then (* ignore *)
                 else
                    (if go##=Station## then
                         (* r ends in another station;
                          * connect with hitstation
                          *)
                         go[]->otherstation[];
                         otherstation.position
                           -> r.end; (*Small adjustment*)

                         (* Add r to hitstation and
                          * otherstation
                          *)
                         &hitstation.rails.element[]->e[];
                         r[]->e.r[];
                         r.theshape.begin[]->e.mypoint[];
                         r.theshape.end[]->e.otherpoint[];
                         e[]->hitstation.rails.append;
                         &otherstation.rails.element[]
                           ->e[];
                         r[]->e.r[];
                         r.theshape.end[]->e.mypoint[];
                         r.theshape.begin[]
                           ->e.otherpoint[];
                         e[]->otherstation.rails.append;
```

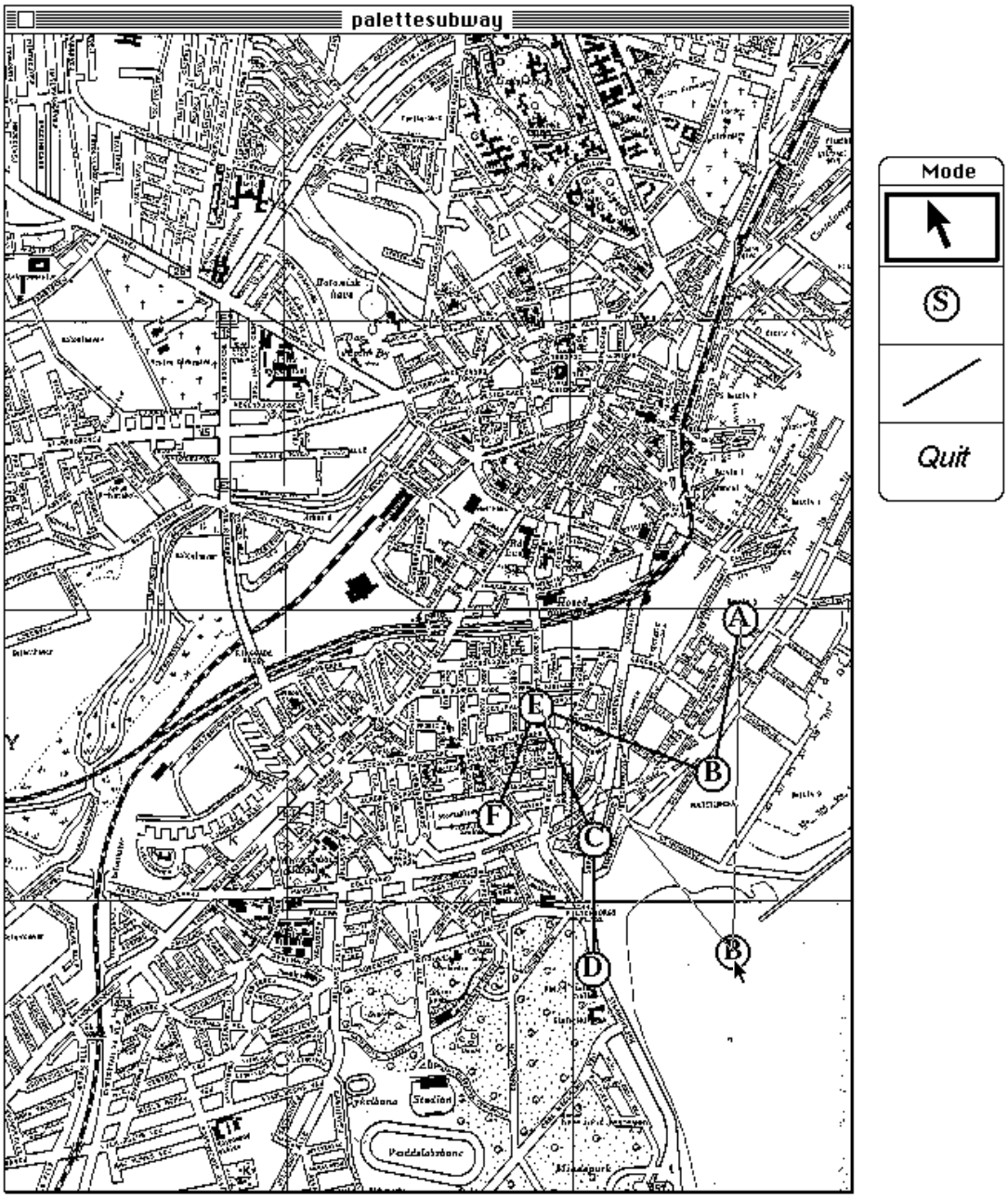
```
                                r->draw;
                                (* It looks better if the
                                 * stations cover the ends of
                                 * the rail.  Instead of
                                 * lowering the rail in the
                                 * BifrostCanvas (which would
                                 * put the rail behind the map)
                                 * we raise the two stations
                                 *)
                                hitstation[]->bringForward;
                                otherstation[]->bringforward;
                                leave scan
                            if);
                        if);
                    if);
                #);
            #);

        open::
            (#
            do (* Make THIS(BifrostCanvas) the size of the map *)
               map.init;
               (map.width, map.height)->Size->theWindow.size;
               (* The first Station will have label "A" *)
               'A'->makeStation.ch;
            #);

        eventhandler::
            (#
               onOpen:: (# do map[]->draw; #);

               onMouseDown::
                 (#
                    (* Actions for Stations *)
                    StationAction:
                      (# s: ^Station enter s[] do INNER #);
                    MoveIt: StationAction
                      (#
                      do (s[],mousepos,NoModifier)
                           ->interactivemove
                      #);
                    MakeRail: StationAction
                      (# do s[]->interactiveCreateRail #);

                    (* Control pattern for finding a station and
                     * performing an action on it.
                     *)
                    findStation:
                      (# s: ^Station;
                         action: ##StationAction;
                      enter action##
                      do (* Find out what was hit - if any *)
                         scan: thePicture.ScanGOsReverse
                           (#
                           do (if (myCanvas[], mousepos)
                                  ->go.containspoint
                              then
                                 (if go[]
                                  //map[] then (* ignore *)
                                  else
                                     (if go##=Station## then
                                         (* We hit a station *)
                                         go[]->s[];
                                         (if action##<>NONE then
                                             s[]->action;
                                         if);
```

```
                                        leave scan
                                    if);
                                if);
                            if);
                        #);
                    exit s[]
                    #);

                hitstation: ^Station;

              do mousepos->devicetocanvas->mousepos;

                 (if paletteOpen then
                     (* Palette determines mode *)
                     (if palettewindow.mode.selection
                      // MoveMode then
                         MoveIt##->findStation;
                      // StationMode then
                         mousepos->makeStation;
                      // RailMode then
                         MakeRail##->findStation;
                     if);
                  else (* Mode-less *)
                     (if findStation->hitstation[]
                      // NONE then
                         mousepos->makeStation;
                      else
                         (* We hit a station *)
                         (if shiftmodified then
                             hitstation[]->MakeRail;
                          else
                             hitstation[]->MoveIt;
                         if);
                     if);
                 if);
               #);
             onKeyDown::
               (#
               do (if ch
                   //'Q' then Terminate
                   //'P' then
                      (if not paletteOpen then
                          palettewindow.show;
                       else
                          palettewindow.hide;
                      if);
                      not paletteopen -> paletteopen;
                 if)
               #);
          #);
      #);
    open::
      (#
      do (* Initialize colors for Stations and Rails *)
         color.init; IndianRed->color.name;
         fill.init;  PaleGreen->fill.name;
         (* Initialize the BifrostCanvas *)
         (* Open the BifrostCanvas *)
         myCanvas.open;
         (* Open the Palette *)
         palettewindow.open;
         'Type ''P'' to open the Palette'->putline;
      #)
  #) (* theWindow *)
do theWindow.open;
#)
```

Appendix B — A Screen Snapshot

The screen snapshot on the following page shows an execution of the final program from Appendix A. It shows a situation where several Stations have been added, and Rails have been added to connect various of them. The Palette has been made visible by typing 'P', and in the Palette, the "Move-Mode" has been selected. In the main window, the Station marked 'B' has been clicked on with the mouse and the Station is being dragged. Notice how the Rails connecting it with other Stations stick to the Station B during the dragging.

This screen snapshot is from X Windows running on a Macintosh. On other platforms, the title bars and window borders may look different, if at all present.


palettesubway
Mode
S
Quit
A
B
C
D
E
F
B
Stadion
Paddelstrand

Bibliography

[Andersen 91] Peter Andersen, Kim Jensen Møller, and Jørgen Rask: *Bifrost—An Interactive Object Oriented Device Independent Graphics System*, Master's thesis, DAIMI Internal Report IR-100, Aarhus University, January 1991.

[MIA 90-10] Mjølner Informatics: *The Mjølner BETA System – The Macintosh Libraries*, MjølnerInformatic s Report MIA 90-10.

[MIA 91-13] Mjølner Informatics: *The Bifrost Graphics System: Reference Manual*, Mjølner Informatics Report MIA 91-13

[MIA 91-16] Mjølner Informatics: *The Mjølner BETA System—X Window System Libraries*, MjølnerInformatics Re port MIA 91-16.

[MIA 94-27] Mjølner Informatics: *The Mjølner BETA System—Lidskjalv: User Interface Framework – Reference Manual*, MjølnerInformatics Re port MIA 94-27.

Index