MIA 91-13: The Bifrost Graphics System - Reference Manual

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Copyright Notice

Bifrost Graphics System

1 Bifrost Reference Manual

This report presents the library available in the Mjølner System for programming applications in the Bifrost Graphics System. Bifrost is an interactive object oriented device independent graphics system, and is the result of a master thesis work as described in [Bifrost 92].

The above mentioned report describes graphics in general using a taxonomy for graphics systems, and explains why Bifrost is designed as it is. The report also includes documentation of the implementation.

This manual starts with an introduction in Chapter 2, followed by a chapter describing necessary mathematical concepts.

Chapters 4 to 8 describe the basic concepts of the Bifrost imaging model - shape, paint, graphical object, picture and canvas, respectively.

Chapter 9 introduces a series of objects defined as an assistance for the user of Bifrost. The objects define various shapes and graphical objects in common use. After the basic concepts have been defined, chapter 10 describes how Bifrost implements interaction on display devices that support interactive input and output.

Chapter 11 describes of how to save pictures in files, and later reloading them from files.

Bifrost is currently implemented as a library for Lidskjalv; this is further explained in chapter 12.

Finally a complete set of interface descriptions for all of Bifrost is presented. This includes all patterns available, and their enter and exit parameters.

This manual is primarily a reference manual, that is, it is not recommended to try to learn how to use Bifrost from reading this manual from one end to another. Instead the reader should consult the Bifrost Tutorial [MIA 91-19], which contains a stepwise introduction to the most important parts of Bifrost.

Bifrost - Reference Manual   Mjølner Informatics

Bifrost Graphics System

2 Introduction

This chapter describes the observations of a user running a typical session with a Bifrost application. Using this strategy, we try to introduce the concepts of Bifrost, as first seen by a new user.

Imagine a graphics workstation running a window system. One of the windows is a canvas showing graphics. See Figure 1
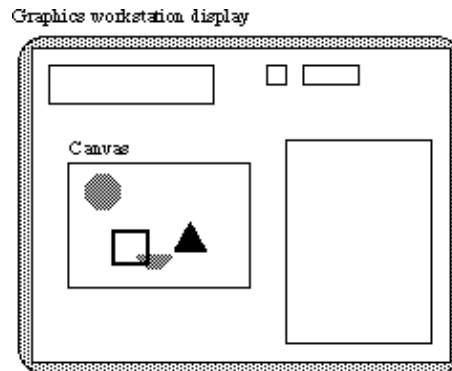
Figure 1: Illustration of a canvas in a window system displayed on a graphics workstation

The canvas picture:

Graphical object:

Picture:

The canvas is a representation of a drawing surface, and is the connection between the window system and Bifrost. The canvas contains a picture, and all graphics shown in the canvas must be in the canvas picture. The picture is a collection of graphical objects, and realize the concept of graphics modelling. The graphical object is the smallest possible entity that can be drawn, and is complete in the sense, that it contains all necessary information about how the graphical object appears on the canvas, and is therefore independent of any other graphical objects in a picture.

The graphical object concept is a composition of two concepts: shape and paint. The shape describes the outline of the object, and the paint describes the color or raster to be pushed through the object when is it displayed on the canvas. The shape of a graphical object is described by segments. A segment is either a straight line segment or a spline segment. Spline segments are

used to describe curves. The shape is analogous to the stencil in the Stencil & Paint imaging model.

The canvas picture in the margin consists of two objects: one graphical object and one picture. The gray circle graphical object is composed of a circle shape and a gray paint as illustrated in Figure 2
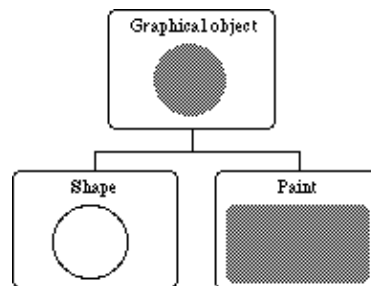
Figure 2: The graphical object is a composition of a shape and a paint

The three other graphical objects in the example above are assembled in a picture consisting of a black frame, a gray half circle, and a black triangle. The picture is shown in Figure 3. The three graphical objects are also, of course, each defined by a shape and a paint.
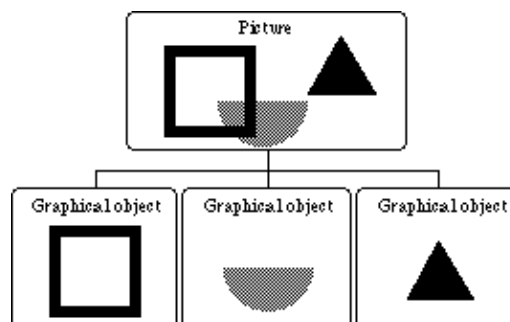
Figure 3: A picture is a composition of graphical objects

The basic imaging model of Bifrost is thus very simple: define a shape that represents the outline of the object you like to draw, select a paint as the color for the object, construct a graphical object with the shape and paint just defined, and draw the object in a canvas, i.e. insert the graphical object into the canvas picture. If the object must have different colors the object must be split into more graphical objects and assembled in a picture. The picture is itself a graphical object, and can thus be drawn in a Canvas.

An application using Bifrost to render graphics, may use many canvases and windows, but each window must have at least one associated canvas to draw graphics. More than one canvas may be associated to the same window, and the canvases in the same window may overlap.

Bifrost - Reference Manual   Mjølner Informatics

Bifrost Graphics System

3 Coordinate Systems and Transformations

Before the concepts are considered in detail, a few mathematical concepts must be defined. That is, coordinates, coordinate system, and transformation between coordinate system.

On the following pages, it is assumed that the reader is familiar with concepts like Cartesian coordinate systems and matrix operations.

Bifrost - Reference Manual

3. Coordinate Systems and Transformations

3.1 Coordinates
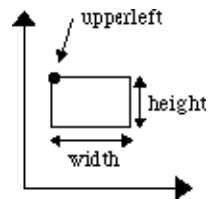
When a graphical object is to be drawn, the points that defines the outline of the graphical object must be specified in some way, i.e. where is the shape supposed to be. Bifrost uses standard Cartesian coordinates for this purpose. Standard Cartesian coordinate subtraction and addition are supported. The figure above illustrates the coordinate system used.

As shown, an axis-parallel rectangle is described by one point (upperleft) and two integers (width and height).

Bifrost - Reference Manual

Mjølner Informatics

3. Coordinate Systems and Transformations

3.2 Coordinate Systems

Output devices vary greatly in the built-in coordinate systems they use to address actual pixels within their display area. Therefore, in a device independent imaging model, there must exists at least two coordinate systems: One referring to the actual device, called the Device Coordinate System (DCS), and one coordinate system completely independent of the device coordinate system, sometimes called the world coordinate system but here called the Canvas Coordinate System (CCS) since it is related to the canvas (see chapter 8).

The implementation of Bifrost with respect to an actual device defines a transformation between these two coordinate systems. The user applications can thus draw in the device independent coordinate system, while Bifrost is making sure that the picture will be transformed into device coordinates, and that the picture can be drawn (identically) on different devices.

The transformation between the CCS and the DCS coordinate systems is not an ordinary geometric transformation. The DCS relates to the device and the device coordinates are typically integers. Bifrost does not restrict the CCS coordinates to be integer values. In cases where CCS is defined in, say, floating point coordinate values, the transformation includes, beside the normal geometric transformation, a mapping from real values to integer values. The default unit on the axes of the CCS coordinate system is currently dertermined by the pixel size of the DCS, but can be changed as needed.

In later chapters new coordinate systems will be introduced. The CCS is the world coordinate system of Bifrost, implying that all coordinate systems are initially defined to be CCS. The next section explains how to obtain geometric transformations by applying matrices to the coordinates.

Bifrost - Reference Manual

Mjølner Informatics

Bifrost Graphics System

3.3 Transformations

Transformation of coordinates from one two-dimensional coordinate system to another can be specified by means of a 3x3 transformation matrix. The matrix specifies how a point in one coordinate system is transformed into the corresponding point in another coordinate system.
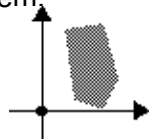
The subsequent definitions of the geometric transformations are illustrated with the example polygon in the margin.

A transformation matrix(TM) specifies a transformation of point (x,y) to point (x', y') in the following way:

$$(x', y', 1) = (x, y, 1) * TM = \begin{pmatrix} ax + cy + t_x \\ bx + dy + t_y \\ 1 \end{pmatrix}$$

$$\text{where } TM = \begin{pmatrix} a & b & 0 \\ c & d & 0 \\ t_x & t_y & 1 \end{pmatrix}$$

The common transformations: scaling, moving (translation), and rotation can easily be described by transformation matrices.
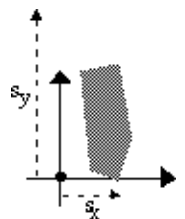
Scaling by factor s_x in the x dimension and s_y in the y dimension is accomplished by:

$$TM_{Scale} = \begin{pmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{pmatrix}$$
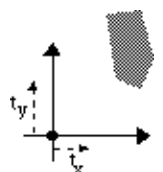
Moving (translation) by a specified displacement (t_x,t_y) is obtained by

$$TM_{Move} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ t_x & t_y & 1 \end{pmatrix}$$
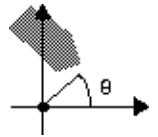
Rotating counterclockwise, around the origin of the coordinate system, by an angle θ is described by the following matrix:

$$TM_{Rotate} = \begin{pmatrix} \cos\theta & \sin\theta & 0 \\ -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

The most powerful feature of the matrix application, is that composition of geometric transformations can be expressed as multiplications of the corresponding matrices. That is, a combination of a rotate, move, and scale transformation can be combined into one matrix, and thus reduce the time of calculation of a complex transformation:

$$TM_{Rotate} \cdot TM_{Move} \cdot TM_{Scale} = \begin{pmatrix} s_x\cos\theta & s_y\sin\theta & 0 \\ -s_x\sin\theta & s_y\cos\theta & 0 \\ t_x\cos\theta - t_y\sin\theta & t_x\sin\theta + t_y\cos\theta & 1 \end{pmatrix}$$

The user must be aware of the order of transformations, since matrix multiplication is not commutative. For a more thorough explanation of matrix operations and transformations, see e.g. [Newman 81].

Bifrost - Reference Manual   Mjølner Informatics

Bifrost Graphics System

4 The Shape

The shape of a graphical object expresses the outlines of the holes in the stencil where paint can be pushed through. Shapes can be arbitrarily complicated within the Stencil & Paint model. The basic building blocks of the shape are segments and these are the subject of the following four sections.

Subsequent to the segment sections the shape concept is described. The most important properties of the shape are the shape constructing language and the ability to combine shapes and thereby e.g. making holes in shapes. Another important property of the shape is the stroke operation, which transforms the shape into a new shape.

Bifrost - Reference Manual   Mjølner Informatics

4 The Shape

4.1 Segments

One can think of a shape as the boundaries of the graphical object, where the boundaries are made of segments. Straight line boundaries are made of line segments and curved boundaries of spline segments. It is possible to combine both line and spline segments in the construction of a shape, as can be seen in in example in the margin using four line segments and two spline segments.

As can be seen in Figure 4 there are three kinds of segments: line, non-circular spline and circular spline segments.
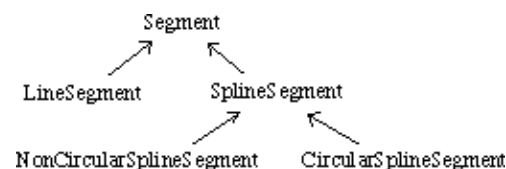
Figure 4: The segment hierarchy

A line segment is a straight line between two end points. A spline segment is spanned by at least three control points. There are two kinds of spline segments: an non-circular spline that terminates in its two extreme control points and a circular spline that do not touch any of its control points and does not have a start nor an ending point. Except for the two end points of a non-circular spline, the control points of a spline segment does not lie on the curve. Instead the control points are distant to the curve and act like 'magnets' pulling the curve. See Figure 5 , which shows examples of the three segment types. The quadratic dots are the control points defining the segments.
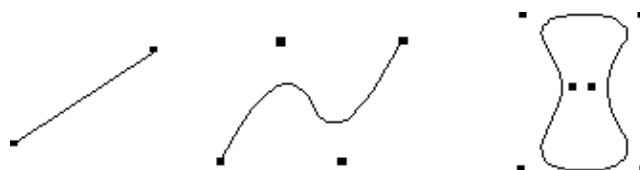
Figure 5: Examples of Segments

It is possible to construct any kind of shape using the three segment types. Any kind of shape with non-curved sides such as triangles, rectangles and polygons with an unlimited number of edges can be constructed from line segments. Circles and ellipses can be represented with circular spline segments. Even objects consisting of a combination of spline and line segments can be constructed. Since shapes represent the outlines of graphical objects, it is possible to construct any kind of graphical object as long as the object has well defined boundaries.

Bifrost - Reference Manual

4 The Shape

4.2 Orientation of Segments

A segment defines two special control points referencing the first and the last control point of the segment. These points are called FirstPoint and LastPoint, respectively. Line segments consist, of course, only of a FirstPoint and a LastPoint. Spline segments consist of at least three control points, where FirstPoint and LastPoint refers to two of the points. In the case of a circular spline FirstPoint and LastPoint are identical and refer to an arbitrary control point of the circular spline. In the case of a non-circular spline FirstPoint and LastPoint refer to the first and last point in the spline, respectively. The result of this definition is that a segment is said to have a direction from FirstPoint to LastPoint.

When segments are used in construction of a shape, the segments are connected in such a way that LastPoint of a segment is connected to FirstPoint of the next segment. In this way the shape gets an orientation. The orientation of the shape is used to determine what is inside and what is outside of the shape. It is the inside of a shape that is filled with paint when a graphical object is drawn.

Bifrost - Reference Manual

Mjølner Informatics

4 The Shape

4.3 Fill Rules

Two different approaches can be used to specify what is inside a shape: even-odd fill rule and (non-)zero winding fill rule. The following examples illustrate the two approaches.
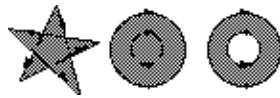
Non-Zero Winding rule   **Even-odd rule**
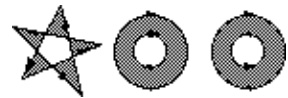
Figure 6: Fill Rules

The non-zero winding rule determines whether a given point is inside a shape by (conceptually) drawing a ray from that point to infinity in any direction and then examine the places where a segment of the shape crosses the ray. Starting with a count of zero, the count is incremented each time a segment crosses the ray from left to right and decremented each time a segment crosses from right to left.[1] After counting all the crossings, if the result is zero then the point is outside the shape, otherwise it is inside. With this rule, a simple convex shape yields inside and outside as would be expected.

Now consider a five pointed star, drawn with five connected straight line segments intersecting each other. The entire area enclosed by the star, including the pentagon in the center, is considered inside by the non-zero winding rule. For a shape composed of two concentric circles, if they are both drawn in the same direction, the areas enclosed by both circles are inside according to the rule. If they are drawn in opposite directions, only the area between the two circles is inside according to the rule; the 'hole' is outside.

The even-odd rule determines whether a given point is inside by drawing a ray from that point in an arbitrary direction and counting the number of segments that the ray crosses. If the number is odd the point is inside; if even, the point is outside. The even-odd rule yields the same results as the non-zero winding rule for simple shapes, but different results for more complex ones. For the five pointed star drawn with five intersecting lines, the even-odd rule considers the triangular parts to be inside, but the pentagon in the center to be outside. For the two concentric circles, only the area between the two circles is inside, regardless of the directions of the circles.

The non-zero winding rule is more versatile than the even-odd rule and is the default rule used by Bifrost to determine what is inside and outside of a shape. Since the even-odd rule is occasionally useful for special effects or for compatibility with other graphics systems, optionally, this rule may be used instead.

---

[1] The rule does not specify what to do if a segment coincides with or is tangent to the ray. Since any ray will do, one may simply choose a different ray that does not encounter such problem intersections.

---

4 The Shape

4.4 Shape Definition Primitives

Usually the application programmer does not have to use segments directly when defining a shape. Instead there are a few operations in the shape that can be perceived as a language for shape definition: Open, Close, LineTo, SplineTo, Stroke, in addition to several operations for combining shapes. Combining shapes is not a straightforward task and is the subject of a subsequent section.

When using these operations, the concept of shape control points is used instead of segment control points. When looking at control points of the shape, the two control points in a joining of two segments are seen as one control point of the shape.

The first four operations for shape definition are used for adding control points to the shape. Depending on which operation is used, the curve between the previously placed control point and the new control point can be either a line or a non-circular spline. The Stroke operations is a powerful way of defining shapes illustrating outlines of graphical objects. It will be presented in the next section.

Open.

Open takes one argument (a point) and defines this as the first control point of the shape. After opening the shape, it is prepared to be constructed by means of a sequence of LineTo and SplineTo messages.

Close.

This places a control point at the same position as the first point hereby closing the shape. Close does not have to be invoked on a shape to make it a legal shape, but it ensures that the shape is closed, which is necessary when it is used with a paint in a graphical object. More on this later.

LineTo.

This operation adds a line segment to the shape, using the last control point of the shape as the first control point of the line segment, and the specified point as the last control point of the line segment.
The following example illustrates the use of the LineTo and Close operations:

```
aTriangle: @Shape   (#   do ( 0, 0) -> Open;      (100,100) -> LineTo;      (1
```
Resulting triangle:

The triangle shape now consists of three line segments, is closed and could be used in a graphical object.

SplineTo.

This operation adds a control point to the non-circular spline segment under construction.
Two different cases must be considered: is there currently a non-circular spline under construction or not.
In the former case (the last operation was SplineTo) the specified point is just added as a spline control point to that spline segment.
In the latter case (the last operations was LineTo or Open) a spline segment will be

created with the ending point of the shape as the first spline control point and the specified point as the second spline control point. The following example illustrates the use of SplineTo:

```
 aShape: @Shape   (#    do (  0, 0) -> Open;         (100,50) -> LineTo;     (150,40
```

Resulting open shape:
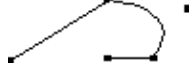
If a LineTo message follows a spline construction, LastPoint of the spline segment becomes FirstPoint of the new line segment. The shape in the example above consists of two line segments and one spline segments with three control points. Notice that the shape is not closed.

If a Close message follows a spline construction, the spline will be ended with a control point in the starting point of the shape:

```
 myShape: @Shape   (#   do (  0,  0) -> Open;        (  0, 50) -> LineTo;    ( 25,
```

Resulting closed shape:
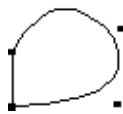
MyShape consists of two segments, one line segment and one spline segment with five control points. Circular splines can not be constructed with the SplineTo primitive. Circular splines have to be created as circular spline segments and then added to the shape.

Bifrost - Reference Manual

Mjølner Informatics

4 The Shape

4.5 Stroke

A very powerful way of defining shapes is by applying a stroke to a previously defined shape. The metaphor for stroke is that a scalpel is moved parallel to the segments of the shape definition, at a specified distance perpendicular to the segments:

```
openStroke: @Shape
   (#
   do (100, 100) -> open;
      (150,  50) -> lineTo;
      (200, 100) -> lineTo;
      (250,  50) -> lineTo;
      (300, 100) -> splineTo;
      (350,  50) -> splineTo;
      (400, 100) -> splineTo;
      (10, CapButt, JoinMiter) -> stroke;
   #);
```
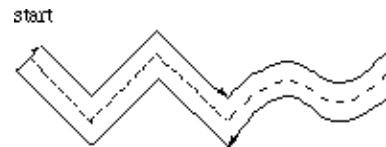
Resulting Shape:

Figure 7: Stroke operation applied to an open shape

In Figure 7 the dashed curve is the original shape which consists of three line segments and one spline segment with four control points (the control points are not shown). The resulting shape is the outline made by the scalpel. The scalpel starts (and ends) in the leftmost top corner. The orientation of the resulting shape is indicated with arrows along the segments. Notice that the shape is closed after the operation has been applied.

A closed shape is stroked likewise but with no need to make special ends:

```
closedStroke: @Shape
   (#
   do ( 50, 100) -> open;
      (200, 100) -> lineTo;
      (200,  50) -> lineTo;
      ( 50,  50) -> lineTo;
      close;
      (10, CapButt, JoinMiter) -> stroke;
   #);
```
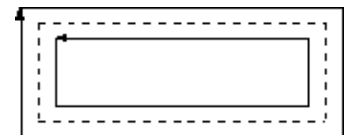
Resulting Shape:

Figure 8: The stroke operation applied to a closed shape

Notice the different orientation of the outer and inner shapes, and that the resulting shape consists of two shapes (or more precisely: the shape consists of four line segments and one shape - also consisting of four line segments)

When the stroke operation has been applied, the original shape is altered and cannot be restored but only the new shape can be manipulated (i.e. the segments of the original shape are irreversibly replaced with the segments that define the new stroked shape). This is different from traditional graphics systems like PostScript, where the original shape (or path in PostScript terms) is unchanged and the stroke only makes a temporary outline which is discharged after having been used for filling an area of the drawing surface. Notice particularly, that the stroke of a spline segment results in two parallel spline segments. This is a completely new idea, since the traditional graphics systems can avoid calculating new spline control points by approximating the spline with a polygon before calculating the temporary outline (the traditional models do not need the new splines since the temporary outline is immediately discarded without giving the user the possibility

to transform or modify the stroked spline).

The advantage of altering the shape is that it then becomes possible to further manipulate the shape, and that the shape can be used for other purposes than just drawing it, e.g. clip to the shape, detect mouse clicks within the shape etc.

4.5.1 Cap and Join Styles

The Cap parameter to the Stroke operation determines how the resulting shape looks in the part of it corresponding to the end point of the original shape. The Cap parameter is only relevant for open shapes. The Join parameter determines how the line parts of the shape are joined. In the example above for an open shape, the Cap parameter is CapButt specifying a line perpendicular to the shape and the Join parameter in both examples is JoinMiter. The alternatives are illustrated in Figure 9.

Figure 9: Cap and join styles for the Stroke operation

Bifrost - Reference Manual

4 The Shape

4.6 Hotspot

All shapes contain one special point, called hotspot. The hotspot can be set to any point in the coordinate system of the shape, but if not explicitly set, the hotspot equals the last point added to the shape. The hotspot is especially useful when working with closed splines, e.g. the hotspot could be set to the center of a circle instead of some irrelevant control point outside the circle. See the [next section](#) for an application of hotspot. Also, the hotspot is used when the shape is filled with paint involving rasters, see [chapter 5](#).

Bifrost - Reference Manual

4 The Shape

4.7 Combining Shapes

When constructing complex shapes, it is often convenient to define the shape as simpler shapes and then combining the simpler shapes into the complex shape[2]. Shape is a subclass of Segment. This makes it possible to combine simple shapes into more complex ones in Bifrost: shapes can be treated as segments. Notice, that it was not shown in the segment hierarchy figure (Figure 4) that Shape is a subclass of Segment[3].

The shape to be combined with another shape will be called the source shape, and the shape that receives the source shape will be known as the destination shape. Shapes inside another shape is referred to as subshapes. Four different semantics are possible for combining two shapes:

> ***AppendShape**.*
>
>> The source shape is automatically translated in such a way that FirstPoint comes to coincide with the LastPoint of the destination shape
>
> ***ConnectShape**.*
>
>> A transformation matrix is supplied that defines how the source shape should be transformed into the destination shape. LastPoint of the destination shape is connected to FirstPoint of the source shape with a line segment
>
> ***ConnectShapeSmooth**.*
>
>> Like ConnectShape except that the two shapes are connected with a spline segment
>
> ***CombineShape**.*
>
>> A supplied transformation matrix transforms the source shape into the coordinate system of the destination shape. The two shapes do not become connected

All operations make a copy of the source shape, and use this copy in the operation. It is important to notice that it is not all kinds of shapes that can be used as source shapes in all of the above four ways of combining shapes: the first three cannot take as argument a shape that only consist of circular spline segments. The reason is that a circular spline segment does not have a well-defined FirstPoint or a LastPoint. CombineShape cannot take an open shape as argument if it is open itself.

A shape is defined in its own coordinate system, which defaults to the CCS coordinate system. A shape has only one coordinate system, implying that all subshapes of a shape are defined in the same coordinate system as the destination shape itself. This is done by transforming the control points of the source shape into destination shape coordinates when shapes are combined.

The reason for only having one coordinate system for a shape is to limit the computing overhead and complication that would otherwise result by defining shapes with many coordinate systems within the same shape. This restriction does not reduce the power of the shape construction language, since nothing is gained by having more than one coordinate system in the same shape. Shapes in different graphical objects may each have different coordinate systems related to the graphical objects.

For each combination operation, there are four cases to consider, depending on the state of the source shape and of the destination shape:

1. Open source shape and open destination shape
2. Closed source shape and closed destination shape
3. Open source shape and closed destination shape
4. Closed source shape and open destination shape

Each case is illustrated with examples in the description of each combination operation below. The underlying philosophy of the four shape combination operations is to have consistent semantics in an operation. This can result in some combinations of shapes that do not seem useful. The most useful combinations are:

- AppendShape with open shapes
- ConnectShape and ConnectShapeSmooth with any kind of shape
- Combine with closed shapes.

4.7.1 AppendShape

The source shape is automatically translated in such a way that FirstPoint comes to coincide with the LastPoint of the destination shape. After the operation the following two statements holds:

1. FirstPoint of the source shape is equal to LastPoint of the destination shape
2. LastPoint of the resulting shape is the translated LastPoint of the source shape

In the examples below, FirstPoint of the source shape, LastPoint of the destination shape, and LastPoint of the resulting shape are marked with bullets (•).

state of source	state of dest.	source	destination	result
open	open			
closed	closed			
closed	open			
open	closed			

Notice, that in the two cases where the source shape is closed, FirstPoint and LastPoint coincide, which means that the LastPoint of the resulting shape remains unchanged (i.e. the same as LastPoint of the destination shape).

4.7.2 ConnectShape

The supplied transformation matrix transforms the source shape into the coordinate system of the destination shape. LastPoint of the source shape is connected to the FirstPoint of the destination shape with a line segment. After the operation the following statement holds:

> 1. LastPoint of the resulting shape is the transformed LastPoint of the source shape

In the examples below, FirstPoint of the source shape, LastPoint of the destination shape, and LastPoint of the resulting shape are marked with bullets. In all four cases the same transformation matrix is used. It performs rotation, scaling, and translation of the source shape.

state of source	state of dest.	source	destination	result
open	open			
closed	closed			
closed	open			
open	closed			

Notice, that in the two cases where the source shape is closed, FirstPoint and LastPoint coincide, which means that LastPoint of the resulting shape remains unchanged (i.e. the same as LastPoint of the destination shape). Notice also, that when both the source and the destination shapes are closed the resulting shape is open (FirstPoint <> LastPoint).

4.7.3 ConnectShapeSmooth

The supplied transformation matrix transforms the source shape into the coordinate system of the destination shape. LastPoint of the source shape is connected to the FirstPoint of the destination shape with a spline segment. After the operation the following statement becomes true:

> 1. LastPoint of the resulting shape is the transformed LastPoint of the source shape

In the examples below, FirstPoint of the source shape, LastPoint of the destination shape, and LastPoint of the resulting shape are marked with bullets. In all four cases the same transformation matrix is used. It performs rotation, scaling, and translation of the source shape.

state of source	state of dest.	source	destination	result
open	open			
closed	closed			
closed	open			
open	closed			

Notice, that in the cases where the destination shape is closed, two of the control points defining the spline segment, that connects the two shapes, are from the destination shape: LastPoint, and the control point prior to LastPoint. The third and last control point that defines the spline segment is FirstPoint of the source shape. Notice also, that when both the source and the destination shapes are closed the resulting shape is open (FirstPoint <> LastPoint).

4.7.4 CombineShape

The supplied transformation matrix transforms the source shape into the coordinate system of the destination shape. The two shapes do not become connected. Notice, that it is an error if both source and destination shape are open, since the resulting shape would otherwise not be connected. After the operation, only one of the following statements holds:

1. LastPoint of the destination shape is LastPoint of the resulting shape
2. LastPoint of the source shape is LastPoint of the resulting shape

The latter statement only holds if the source shape is open and the destination shape is closed. The reason for this seemingly strange behavior is that the resulting shape can then be combined further in a consistent way. One can also think of this situation in terms of which shape is open after the operation - in this particular situation it is the source shape that is open.

In the examples below, FirstPoint of the source shape, LastPoint of the destination shape, and LastPoint of the resulting shape are marked with bullets.

Notice the last situation that is commented on above.

---

[2] This implements a limited form of graphics modelling. Later the concept of picture is defined as a more powerful way of doing graphics modelling.

---

[3] The discussion of the Shape inheritance hierarchy is deferred until the presentation of predefined shapes, see Chapter 9.

Bifrost - Reference Manual   Mjølner Informatics

4 The Shape

4.8 Segment Definition Primitives

Segment primitives can be used directly to construct shapes. A shape is constructed in this way by generating a number of segments, and adding these segments to the shape. Segment definition primitives are only meant for internal use in Bifrost. It is not recommended to use these primitives, except for defining circular spline segments, but instead to use the shape definition operations described in section 4.4.

4.8.1 LineSegment

A line segment is described by two control points: Begin (= FirstPoint) and End (= LastPoint). A line segment is constructed by assigning values to the Begin and End points:

```
aSeg:@LineSegment
  (#
  do (100,100) -> begin;
     (200,200) -> end;
  #)
```

Resulting line segment

4.8.2 SplineSegment

As mentioned earlier there are two different kinds of spline segments: circular and non-circular splines. A circular spline is always considered closed, and a non-circular spline is considered closed if the last operation performed is Close. Non-circular splines are typically used in construction of shapes consisting of both line and spline segments, and circular splines are useful for making circles and ellipses. Spline segments have three operations used for creating the spline:

Open.

> Takes one point as argument. The point is the first control point of the spline, and the spline is prepared to be constructed with further control points using AddControl.

AddControl.

> This operation just adds a specified control point to the spline.

Close.

> Closes a non-circular spline by adding FirstPoint to the spline definition. The operation has no effect on a circular spline. Notice, that it is still possible to add further control points to the spline definition after the Close operation.

The following examples illustrate the use of AddControl and Close for a non-circular spline:

```
aBow: @NonCircularSplineSegment
  (#
  do (  0,100) -> Open;
     (100,100) -> AddControl;
     (100,  0) -> AddControl;
     (  0,  0) -> AddControl;
  #);
```

Open non-circular spline:


```
aDrop: @NonCircularSplineSegment
  (#
  do (  0,100) -> Open;
     (100,100) -> AddControl;
     (100,  0) -> AddControl;
     (  0,  0) -> AddControl;
     close;
  #);
```

Closed non-circular spline:

The next examples illustrate the use of AddControl for circular splines:

```
aSpline: @CircularSplineSegment
  (#
  do (  0,100) -> Open;
     (100,100) -> AddControl;
     (100,  0) -> AddControl;
  #);
```

Circular spline with t


```
aCircle: @CircularSplineSegment
  (#
  do (  0,100) -> Open;
     (100,100) -> AddControl;
     (100,  0) -> AddControl;
```

Circular spline with f

```
      (  0,  0) -> AddControl;
  #);
```

Notice that a circular spline is inherently closed.

4.8.3 Adding Segments to Shapes

When the segment is defined it can be added to a shape with two shape primitives: AddLine and AddSpline.

AddLine*.*

Adds a line segment to the shape. In case where FirstPoint of the line segment does not coincide with LastPoint of the shape, the line segment is moved to the appropriate position.

AddSpline*.*

AddSpline adds a spline to the shape in construction. In case where FirstPoint of the spline segment does not coincide with LastPoint of the shape, the spline segment is moved to the appropriate position.

The following example illustrates the use of AddLine:

```
aLine: @LineSegment
  (#
  do (100,100) -> begin;
     (150,150) -> end;
  #);

anotherLine: @LineSegment
  (#
  do (200,200) -> begin;
     (200,100) -> end;
  #);

aShape: @Shape
  (#
  do aLine -> AddLine;
     anotherLine -> AddLine;
  #);
```

Two line segments

The following example illustrates the use of and AddSpline using the two splines defined previously:

```
aShape: @Shape
  (#
  do aCircle -> AddSpline;
     aDrop  -> AddSpline;
  #);
```

Resulting shape:

In the case where the spline is not closed, i.e. FirstPoint does not coincide with LastPoint of the shape, the spline is moved to the appropriate position by AddSpline:

```
aLine: @LineSegment
  (#
  do (100,100) -> begin;
     ( 50,150) -> end;
  #);

anotherLine: @LineSegment
  (#
```

Two lines

```
  do ( 50,150) -> begin;
     (150,200) -> end;
  #);
```

```
aSpline: @NonCircularSplineSegment
  (#
  do (  0,200) -> Open;
     ( 50,250) -> AddControl;
     (100,200) -> AddControl;
     (100,100) -> AddControl;
     ( 50, 50) -> AddControl;
  #);
```

A Spline

```
aShape: @Shape
  (#
  do aLine -> AddLine;
     anotherLine -> AddLine;
     aSpline -> AddSpline;
  #);
```

Resulting shape:
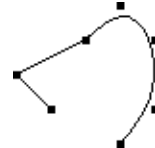
It should be clear from the examples that it is complicated to use the segment definition primitives for shape construction. Therefore, for purpose of convenience and to make the graphics system more powerful, Bifrost also includes the small shape construction language presented earlier.

Bifrost - Reference Manual

Mjølner Informatics

Bifrost Graphics System

5 The Paint

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. The concept Raster is described in section 5.1. In order to use a raster to fill the shape several things must be specified: first, the raster itself must be specified; secondly, the raster position in the shape must be supplied (by specifying the hotspot of the raster in shape coordinates); third and last, it must be specified what to do if the raster is too small to fill out the entire shape. Bifrost supports two approaches when the raster is too small to fill out the entire shape: 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.

The paint hierarchy is illustrated in Figure 10. This hierarchy may, of course, be extended if needed.
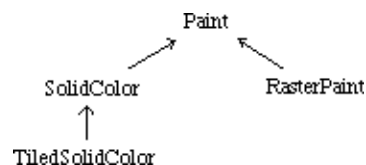
Figure 10: The Paint Hierarchy

Two operations are defined for the general paint concept:

> **FillShape**.
>
>> Takes a shape as argument and fills the shape with the paint on a canvas.
>
> **Copy**.
>
>> Makes a (deep) copy of the paint

Bifrost - Reference Manual

5.1 Rasters

As described in the previous paragraphs both tiled solid color and the raster paint concepts use some kind of rasters describing either a tile, or a raster image in the raster paint. Bifrost defines a class BitMap for using with tiling and a class PixMap to be used in the RasterPaint. In Figure 11 the hierarchy is illustrated.

Raster
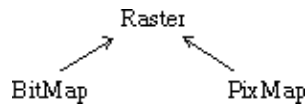
BitMap   PixMap

Figure 11: The Raster Hierarchy

The implementation of bit and pixel maps are inspired by 'portable bitmap file format (PBM)' and 'portable pixmap file format (PPM)' [Poskanzer] available on many Unix and MS/DOS installations. Since the format of the Bifrost rasters are very close to this 'standard', Bifrost can read and write PBM and PPM files, and thereby get access to a huge set of rasters in a lot of different formats.

5.1.1 Raster

The Raster class generalizes the raster concept defining the following attributes

- MagicNumber for identifying the type of the raster.
- Hotspot is a point used when the raster is used in a fill operation: the raster in positioned so that hotspot coincides with the hotspot of the shape being filled.
- Width and Height of the raster.
- Pixel is virtually declared as an Object, and must be further bound in specializations of Raster.
- Width * Height Values specifying the raster itself, starting at the top-left corner of the raster, proceeding in normal reading order.

In BETA code it could look like:
```
Raster:
(#
   MagicNumber: @Integer;
   Width,Height: @Integer;
   Pixel:< Object;
   Values: [Width*Height] @Pixel;
#)
```

Two operations are supported by all rasters:

PutPixel.

Takes an index (i,j) and a pixel value as argument and sets the pixel value into the specified position of the Values.

GetPixel.

Takes an index (i,j) as argument and returns the pixel value in the specified position of the Values.

5.1.2 BitMap

Bifrost defines a bit map in the following way:

Pixel

>is bound to a Boolean where TRUE means "set" and FALSE means "not set".

Two operations are defined to read and write BitMaps:

ReadFromPBMFile.

>Read a bit map from a PBM file into the BitMap.

WriteToPBMFile.

>Write the BitMap out on a PBM file.

5.1.3 PixMap

Bifrost defines a pixel map in the following way:

MaxVal

> The maximum color component value.

Pixel

> is bound to three decimal values between 0 and the specified maximum value. The three values for each pixel represent red, green, and blue, respectively. If it is desired to specify the pixel value relative to some other color space, e.g. HSV (cf. section 5.2), the easiest way is to instantiate a SolidColor (section 5.2), specify the HSV values to this, and then get the RGB values from the SolidColor, and use these in the Pixel.

Bifrost - Reference Manual

Mjølner Informatics

5 The Paint

5.2 SolidColor

A solid color is specified relative to some color space. In Bifrost, three color spaces are supported, namely RGB, CMY and HSV. These are the color spaces that seems to have the most widespread use in computer graphics (cf., e.g. [Bifrost 92], [Foley 90]). The HSV color space is probably the most intuitive of these, since defining a color in it resembles the way an artist do it. In order to ease the job of the programmer, colors can also be specified using a simple naming model. Of course, it will be possible to extend the color support in Bifrost for other color spaces too, if needed.

The number of colors available at the same time depends on the device used. If no more colors are available, a default color is used.

5.2.1 Defining Solid Colors

The following three operations are used to manipulate the attributes of a SolidColor. All three operations can be used to both set and get the values of the solid color in the respective color models:

RGBvalues*.*

> Enters three integers representing red, green, and blue values, and changes the SolidColor accordingly. Exits the current red, green, and blue values. Red, green, and blue values ranges from zero to the value determined by the constant MaxRGB.

HSVvalues*.*

> Like RGBvalues this operation enters three integers. These are interpreted as hue, saturation and value of the solid color. HSVvalues exits the hue, saturation, and value of the SolidColor. HSVvalues has three attributes MaxHue, MaxSat and MaxVal determining the ranges of hue, saturation and value. MaxHue, MaxSat and MaxVal is DefaultMaxHue, DefaultMaxSat, and DefaultMaxVal, respectively, by default, but may be changed by the application programmer.

CMYvalues

> This is like RGBvalues except that the three values entered and exited constitute the cyan-magenta-yellow representation of the SolidColor.

5.2.2 Examples

The following example illustrates how a column with the complete color spectrum, can be drawn using the HSV color model. This is done by stacking thin lines upon each other, having hues 1, 2, ..., MaxHue respectively (that is MaxHue number of lines in all). Full saturation and values are used.

```
(for h:DefaultMaxHue repeat
     &Line[] -> aLine[];
     aLine.init;
     1 -> aLine.width;
     ((0,h), (100,h)) -> aLine.Coordinates;

     &SolidColor[] -> aSolidColor[] -> aLine.SetPaint;
     (h, MaxSat, MaxVal) -> aSolidColor.HSVvalues;

     aLine[] -> aCanvas.draw;
for);
```

A Color Scale:

Notice that it would not suffice to instantiate just one line object and change the color of this one object, because in Bifrost each part of an image must correspond to some graphical object that is inserted into the canvas picture (by the canvas operation Draw). Changing the color of one line object would result in one line with changing colors being shown. Also notice that an instance of the predefined graphical object class Line is used. Predefined graphical objects are defined in chapter 9.

By combining the different color spaces interesting effects can be achieved. The following example is an elaboration of the previous one. The example draws the complementary color spectrum in a column adjacent to the column described in the previous example.

```
(for h:DefaultMaxHue repeat
     &Line[] -> aLine[];
     aLine.init;
     1 -> aLine.width;
     ((0,h), (100,h)) -> aLine.Coordinates;
     &SolidColor[] -> aSolidColor[] -> aLine.SetPaint;
     (h, MaxSat, MaxVal) -> aSolidColor.HSVvalues;
     aLine[] -> aCanvas.draw;

     (* draw the line with complementary color *)
     &Line[] -> aLine[];
     aLine.init;
     1 -> aLine.width;
     ((110,h), (210,h)) -> aLine.Coordinates;
     &SolidColor[] -> anotherSolidColor -> aLine.SetPaint;
     aSolidColor.RGBvalues -> anotherSolidColor.CMYvalues;
     aLine[] -> aCanvas.draw;
for);
```

The result of the last program is illustrated in Figure 12.
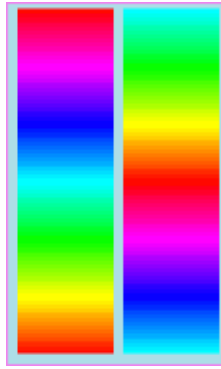
Figure 12: A Color Scale and its Complementary

5.2.3 Name Color Model

An even more intuitive 'color space' than RGB, HSV and CMY is the one used in everyday life: defining the colors simply by naming them. Bifrost support the possibility of specifying a solid color by name by means of a large number of patterns exiting RGB values corresponding to different color names. These patterns are located in the file ~beta/bifrost/ColorNames.

> ***Name**.*

>> Enters the new RGB values, and is hence just an alias for setting the color using RGBvalues. Useful when evaluating one of the color defining patterns, which exits RGB values corresponding to a given color name.

The following example illustrates how to draw a circle with center in (10,10) and radius 25 and filled with solid pink color, using the simple naming color model.

```
&Ellipse[] -> anEllipse[];
anEllipse.init;
((10,10),25,25) -> anEllipse.geometry;
&SolidColor[] -> aSolidColor[] -> anEllipse.SetPaint;
pink -> aSolidColor.Name;
anEllipse[] -> aCanvas.draw;
```

5.2.4 TiledSolidColor

A tiled solid color is a solid color extended with a BitMap. The BitMap will be tiled in the shape before the SolidColor is applied, and only where the pixel values of the BitMap is true the SolidColor will be visible. This is the normal tiling approach. As mentioned in section 5.1.1 the hotspots of the BitMap and of the shape being filled, determine the position of the BitMap within the shape.

Example:

A Shape: The Shape filled with a TiledSolidColor using the BitMap as tile:

A Bitmap:

Bifrost - Reference Manual   Mjølner Informatics

5 The Paint
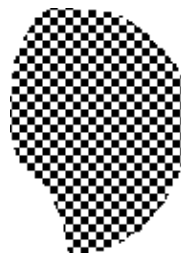
5.3 RasterPaint

In a RasterPaint a PixMap is used to describe an image, and this PixMap is positioned in the shape to be filled. This is done by positioning the hotspot of the PixMap at the hotspot of the shape. Instead of using the PixMap as weights, the pixel values of the map are used directly, and the shape to be filled determines which parts of the image in the PixMap will be shown. [4]

The application programmer should specify what to do if the PixMap given is not big enough to fill the shape: either the PixMap should just be repeated (tiled) as needed, or a solid color to use in the uncovered places of the interior of the shape should be specified. The RasterPaint has the following additional properties compared to Paint:

ThePixMap.

Refers to the PixMap.

PaddingSolidColor.

If PaddingSolidColor is none then the PixMap will be tiled, otherwise the PaddingSolidColor will be used where the PixMap does not cover.

[4] Notice the difference between RasterPaint and TiledSolidColor: In a RasterPaint, thePixMap is used directly as Paint, in a TiledSolidColor, the BitMap determines where the SolidColor should be applied. Possibly "PixmapPaint" or "ImagePaint" would be a better name than "RasterPaint".

Bifrost - Reference Manual   Mjølner Informatics

Bifrost Graphics System

6 The Graphical Object

The graphical object is the central concept of the Bifrost imaging model. The graphical object is central for two reasons. The first reason is that the graphical object is the smallest unit that can be drawn. The second reason is that the graphical object contains all necessary information to draw itself. The information necessary to draw a graphical object, is contained in the description of the shape and the paint, presented in the previous chapters. Notice, that in traditional systems the smallest unit that can be drawn does not always contain all necessary information needed to draw itself. The next section elaborates on the concept of local graphic context.
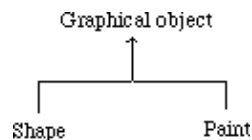
Figure 13: A graphical object is a composition of a shape and a paint

Bifrost - Reference Manual   Mjølner Informatics

6 The Graphical Object

6.1 Graphic Context

With the graphical object containing enough information to render itself independent of its surroundings, Bifrost support local graphic context. Local context is defined as the ability of graphical objects to draw themselves, [Bifrost 92] p. 4. This can very easily and elegantly be designed in an object oriented imaging model. Using local graphic context can, however, give rise to overhead if many objects with the same context are drawn successively: time is wasted by setting the (same) graphics context each time one of the graphical objects are drawn. The contrary to local context, global graphic context is not supported in Bifrost. Instead an intermediate approach between local and global context, called shared graphic context, is designed but not yet implemented [Bifrost 92] p. 80.

Bifrost - Reference Manual

Mjølner Informatics

6 The Graphical Object

6.2 Operations

Most of the operations in the graphical object manipulate, or use operations implemented in the shape or the paint. The only exceptions to this fact is manipulation of the transformation matrix and some administration (init and copy) operations.

Init.

Initializes the GraphicalObject by instantiating a shape and a transformation matrix. Init must be called as the first operation on the graphical object. If the graphical object is evaluated, init is called automatically.

To manipulate the shape and the paint of the graphical object four operations are given:

SetShape and **GetShape**.

Operations to set and get the shape.

SetPaint and **GetPaint**.

Operations to set and get the paint.

6.2.1 Geometric Transformations

Six operations support geometric transformations on graphical objects:

Move.

Enters two displacements (tx,ty) and moves the graphical object relative to its current position.

MoveTo.

Enters point and moves the hotspot of the graphical object to the point.

Scale.

Enters two scaling factors (sx,sy) and scales the graphical object relative from its current size.

Rotate.

Enters an angle (in degrees) and rotates the graphical object relative to the origin.

6.2.2 Query Operations

HitControl*.*

> Takes a point (in CCS) as argument and returns a reference to the exact point (in GraphicalObject coordinates), if it is in the neighborhood[6] of a control point of the shape of the graphical object. Otherwise returns NONE.

ContainsPoint*:*

> Takes a point (in CCS) as argument and returns true if it is inside the shape of the graphical object.

6.2.3 Interaction

InteractiveCreateShape.

Calls InteractiveCreate of the shape, see Chapter 10.

InteractiveReshape.

Calls InteractiveReshape of the shape, see Chapter 10.

InteractiveCombineShape.

Calls InteractiveCombine of the shape, see Chapter 10.

InteractiveMove.

Takes a canvas, a starting point and a modifier description as argument and interactively moves the graphical object using the interaction handler of the canvas, see Chapter 10. Calls Move to do the transformation after the interaction has ended.

Hilite.

Makes the graphical object appear highlighted by using the highlighting operation of the shape. When the graphical object is redrawn (by canvas updating) the graphical object will be drawn highlighted.

UnHilite.

Unhighlights the graphical object.

The Hilite and UnHilite operations changes the Canvas' drawing mode to be XOR, to allow for immediate feedback, and invoke an instance of a virtual drawHilite attribute. Thus the feedback may be augmented by further binding this attribute, see Chapter 10 for more details.

6.2.4 Drawing Graphical Objects

A graphical object is drawn on a canvas by calling the Draw operation of the canvas, see chapter 8. The graphical objects then becomes part of the list of graphical objects in the canvas, and the canvas asks the graphical object to draw itself on the canvas. The graphical objects uses its Draw operation to do the actual drawing in the Canvas.

When a Canvas needs to be redrawn, the Canvas knows which graphical objects are drawn in the Canvas, and can therefore ask the graphical objects in question to redraw themselves on the Canvas. Likewise a graphical object is erased by calling the Erase operation of the Canvas. See chapter 8 for a complete description of the Canvas and when it must be redrawn.

6.2.5 Transforming Graphical Objects

A graphical object can be transformed by manipulating the transformation matrix TM of the graphical object. Such a transformation will affect the appearance of the graphical object, if it is drawn in a Canvas. To simplify transformation of graphical object, the Transform attribute is present:

Transform.

Applies a matrix to the transformation matrix of the graphical object. To be precise,
```
aMatrix[] -> anAbstractGraphicalObject.transform;
```
is equivalent with
```
(anAbstractGraphicalObject.TM, aMatrix[])   -> MatrixMul -> anAbstractObj
```

Notice, that in general, only instances of Shape are guarantied to be transformable, in particular some of the Predefined Graphical Objects (see later) will not respond correctly to all transformations. Translations, i.e. linear moving, however, will work for all kinds of graphical objects. Pictures (see below) may also be transformed, but if they contain Predefined Graphical Objects, the restrictions mentioned above apply to the Picture itself too.

---

[5] Geometric transformations are described in Chapter 3.

---

[6] See section 10.3.1 for a definition of the concept neighborhood.

Bifrost - Reference Manual   Mjølner Informatics

Bifrost Graphics System

7 The Picture

The picture concept is designed to supports graphics modelling.

A picture is a collection of graphical objects and pictures and is itself a graphical object.

It is therefore possible to make hierarchies of graphical objects and pictures, leading to the required capability of doing graphics modelling.

Bifrost - Reference Manual   Mjølner Informatics

7 The Picture

7.1 The Picture List

The concept Picture is a specialization of the concept GraphicalObject. The reason for this specialization is that the picture defines a list attribute consisting of graphical objects. Due to the specialization, the picture is also a graphical object, and can be added to the list of graphical objects in another Picure. The effect of this design is that every object in the list can be treated in an uniform way, without consideration to the actual type of the object. This is a very elegant foundation for Graphics Modelling.

The graphical objects are stacked (the hexagon being the front most graphical object):

Stacking graphical objects: Resulting image:

When the graphical objects in the list are drawn in a canvas, each object is put on top of the other objects already drawn. Hence the graphical objects in the list are stacked relative to each other on the canvas with respect to their positions in the list. In other words, the last object in the list is the front most object on the canvas and the first object is the lowest object on the canvas. Objects in the end of the list may therefore cover other objects earlier in the list, depending of the positions of the objects. The position in the list is therefore important when a graphical object is manipulated interactively. The subject of interaction is discussed separately in Chapter 10.

There are several operations to manipulate the list of graphics objects and pictures. Two operations are used to add and delete objects to/from the list:

>   ***Add***.
>
>>   Takes a graphical object as argument and adds it to the end of the picture list.
>
>   ***Delete***.
>
>>   Takes a graphical object as argument and deletes it from the picture list.

Two operations support moving the graphical objects relative to the other objects in the list:

>   ***BringForward***.
>
>>   Accomplished by moving the object to the last position in the list.
>
>   ***SendBehind***.
>
>>   Takes a graphical object as argument and draws the object at the bottom of the canvas. Accomplished by moving the object to the first position in the list.

Two operations support queries to the graphical objects in the list.

FirstContaining *and* ***LastContaining****.*

> Takes a point (in CCS) as argument, and reports the first or last object containing the point or NONE.

Finally, two operations support scanning through the graphical objects of the picture.

ScanGOs

> Scans through all the graphical objects in the picture, in the order of the list.

ScanGOsReverse

> Scans through all graphical objects of the picture in the opposite order than the list, i.e. from "top" to "bottom" of the picture.

Bifrost - Reference Manual   Mjølner Informatics

7 The Picture

7.2 Selection Picture

In [Bifrost 92] pp. 78, a detailed description of how the picture is supposed to support graphics modelling can be found. Currently only one form of the graphics modelling properties are implemented, that is, a constraining picture called SelectionPicture.

The SelectionPicture specializes the Add and Delete operations. When two or more graphical objects are selected the Hilite and UnHilite operations of the graphical objects are changed from HiliteControls to HiliteOutline. Graphical objects then become highlighted by drawing the outlines of the shapes instead of highlighted control points. This effect is similar to many graphical editors, e.g. the classical Macintosh program MacDraw.

Bifrost - Reference Manual   Mjølner Informatics

7 The Picture

7.3 Picture Coordinate System

When graphical objects are composed into a picture, it is necessary to have a mapping between the coordinate system of the graphical object and the coordinate system of the picture, since each graphical object is defined in its own coordinate system and is placed somewhere in the coordinate system of the picture.

The graphical object defines a transformation from its own coordinate system to the coordinate system of the picture. Since the coordinate system of the shape is identical to the coordinate system of the graphical object it is part of, the transformation actually defines how to position the shape of the graphical object in the picture. This transformation is described in a matrix called TM, inherited from the superpattern AbstractGraphicalObject.

See also the description of the Transform attribute of graphical objects above (section 6.2.5).

Bifrost - Reference Manual

Mjølner Informatics

7 The Picture

7.4 Other Operations on Pictures

The picture defines several other operations to query and manipulate the graphical objects in the picture and the picture itself. The most important ones are listed below, see the [interface description](#) for the rest.

NoOfGOs.

Returns the number of graphical objects in the list.

IsEmpty.

Returns true if the list is empty.

IsMember.

Takes a graphical object as argument and returns true if the graphical object is in the list.

Bifrost - Reference Manual

Mjølner Informatics

Bifrost Graphics System

8 The Canvas

The canvas[7] is the drawing surface of the Bifrost graphics system and the connection between Bifrost and the display device. The display device is typically a screen or a window in a window system. In a window system the canvas is a drawing surface inside a window. How the canvas is made a part of a window (borders, scroll bars, close box, etc.) depends on the specific system.

The canvas is very similar to the canvas of an artist, but has properties not comparable to the canvas of an artists, described in the next sections.

[7] As explained in chapter 12, in the current implementation of Bifrost Lidskjalv Canvas is actually named BifrostCanvas, since there is another pattern in Lidskjalv with the name Canvas.

8.1 Drawing and Visible Area

The canvas is a potentially infinitely large drawing area. Hence everything in the canvas is not necessarily immediately visible on the display device. The canvas therefore defines a visible area, defined by means of a shape called the visible shape. The visible shape is not fixed and can be reshaped even after graphical objects have been drawn in the canvas. The visible shape is a view to the canvas. By moving the visible shape it is possible to view other parts of the canvas, hence the visible shape can be used to implement scrolling. Notice that the visible shape is not effected by the surroundings of the canvas, e.g. by overlapping windows.

In typical drawing applications where the user chooses a part of the canvas to actually see on the display device, the application programmer should not consider which part of the canvas is visible, but should regard all the graphical objects in the canvas as being visible. The canvas will only draw those graphical objects that are visible on the device. This is accomplished by clipping and updating as described in section 8.3 and 8.4.

Bifrost - Reference Manual   Mjølner Informatics

8 The Canvas

8.2 The Canvas Picture

As mentioned in chapter 6, the graphical objects drawn on the canvas are stored in a picture. When a graphical object is to be shown in a canvas, it is done by invoking the Draw operation of the canvas, whereby the graphical object is added to the picture of the canvas and displayed using the draw operation of the graphical object. It is not allowed, in the current version of Bifrost, to draw objects in the canvas that is not part of the canvas picture.*[8]*

The following two attributes operate directly on the canvas picture:

> **scanThePicture***.*
>
>> Invokes the scanGOs on the Canvas picture.
>
> **scanThePictureReverse***.*
>
>> Invokes the scanGOsReverse on the Canvas picture.

---

[8] For purposes of interaction, it is possible to draw simple objects (lines, rectangles etc) immediately on the canvas without using the canvas picture, see Chapter 10.

Bifrost - Reference Manual   Mjølner Informatics

8 The Canvas

8.3 Clipping

Clipping is used in graphics system to restrict the area in which graphics operations have effect. For instance, when some part of a window is damaged, the application can clip to the damaged part and then draw the whole window. The result of the clipping is that the system ignores the drawing request outside the clipping area, and the speed of updating is increased significantly.

The canvas supports clipping in the situation where some of the graphical objects are totally or partly outside the visible shape of the canvas. The canvas always clips to the visible shape. In addition to this clipping area it is possible to set another clipping area, also defined by a shape - the clip shape. SetClip and GetClip can be used to set and get the clip shape. The clip shape is especially useful when damaged areas must be updated, see the next section.

Bifrost - Reference Manual   [Mjølner Informatics](#)

8 The Canvas

8.4 Updating Damaged Areas

When an area inside the visible area of the canvas has been obscured, e.g. by an overlapping window, and again becomes visible, parts of the canvas must be redrawn. The canvas supports redrawing by updating areas that have been damaged. The damaged area is handled by a damaged-list in the canvas.

Update events originating from the window system - say, a window overlapping the canvas is moved - are handled automatically by Bifrost. When Bifrost receives an update event from the window system, the damaged area is in many cases reported along with the update event. The canvas redraws the damaged area transparently to the application. In cases where the damaged areas is not reported, the whole visible area of the canvas is redrawn.

The process of updating damaged areas originating from application dependent actions - say, removing of a graphical object - is a partly application responsible process; it is not entirely automatic. In this situation the application is responsible of adding damaged areas to the damaged-list. Adding a rectangle to the list is accomplished by the operation damaged. After the application has called the operation Repair, the canvas redraws the visible area using an advanced algorithm to determine which objects must be redrawn.

The traditional way of redrawing is to draw all objects and turn the responsibility of selecting the objects inside the clipping area to the display device (or basic graphics library). Although clipping is a very efficient way of reducing the overhead of the display device in redrawing, it is still necessary to redraw all graphical objects in the canvas.

An alternative is to limit the number of graphical objects and pictures that has to be considered in the redrawing process. When applications use graphics modelling, each picture typically consists of a small set of proximate and related objects, expected to be updated collectively, e.g. by moving the picture. This means, that if a picture is completely outside the region that should be updated, then it is not necessary to further consider the graphical objects inside the picture. The following example illustrates the situation:
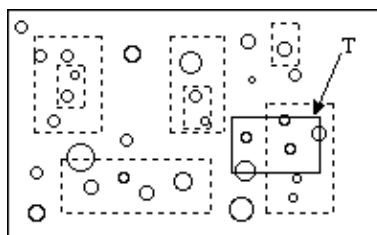
Given a clipping rectangle T and some pictures (illustrated by dashed rectangles) and graphical objects (illustrated by small circles) all the graphical objects in pictures that are completely outside T are never considered.

This approach depends on the assumption that the application is using graphics modelling, and that the graphical objects in each picture are close to each other. Consider for example two graphical objects in a picture, that are very distant from each other, the picture becomes very large and it is a more likely that the picture intersects T even though the graphical objects may be outside T. On the other hand, this update approach encourage the application programmer or user to apply graphics modelling to the drawing.

The advanced updating approach in Bifrost does not exclude the possibility to use other updating mechanisms. In cases where graphics modelling can not be used or does not make sense, the application programmer can implement a different approach, e.g. the very advanced method described by Edelsbrunner called 'dynamic rectangle intersection search' [Edelsbrunner 80].

Bifrost - Reference Manual

Mjølner Informatics

8 The Canvas

8.5 Input Control

The canvas models the input in two handlers: the event handler and the interaction handler. The canvas uses the event handler to receive general events such as window resizing and updates. Each canvas has exactly one event handler. The interaction handler is started by request, e.g., when a new graphical objects is to be created. The interaction handler is a special-purpose event handler, designed for fast interaction with the user.

The typical situation is that the event handler polls for input from the user, and when the user e.g. starts creating a new ellipse by clicking on a mouse button, the application calls InteractiveCreate for an ellipse graphical object. Interactive create is implemented using an interactionhandler.

The event handler is described in this section and the interaction handler is described in Chapter 10.

The event handler models the events originating from the basic graphics system or from user actions by six virtual operations. The application programmer may then further bind these operations in an application that uses the Bifrost graphics system.

> **OnOpen**.
>
>> Called when the canvas is shown for the first time on the display device.
>
> **OnButtonDown**.
>
>> Called when the user presses a button on the pointing device.
>
> **OnKeyDown**.
>
>> Called when the user hits a key on the keyboard.
>
> **OnRefresh**.
>
>> Called when the canvas must be redrawn, e.g. when it is exposed after is has been obscured. The refresh event is typically generated by the basic graphics system (or by the window manager).
>
> **OnActivate** and **OnDeactivate**.
>
>> Called when the canvas is (de)activated. The exact definition of activation may vary with the device. In a window system, the active window will normally have its title bar highlighted. The activation occurs when the window becomes the active window.

9 Predefined Shapes and Graphical Objects

A number of Predefined shapes and corresponding graphical objects are designed to assist the user of Bifrost. With objects such as lines, circles and text available, the user can create graphics easier and faster than from scratch. Furthermore, the predefined shapes and graphical objects may utilize the underlying graphics system and hardware/firmware operations more efficiently.

The predefined shapes currently implemented in Bifrost are described in the first seven sections of this chapter. In the last section it is outlined how the shapes are defined in Bifrost, and how to define new 'predefined' shapes. The purpose of defining new 'predefined' shapes, is mainly to utilize the underlying graphics hardware/software. Figure 14 illustrates the predefined shape inheritance hierarchy.
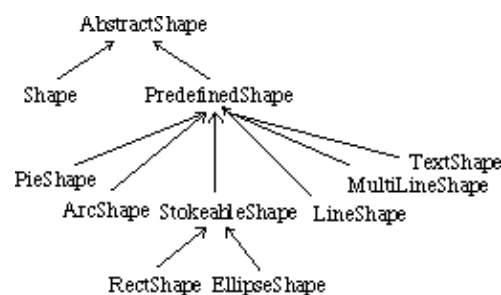
Figure 14: Predefined Shape Inheritance Hierarchy

As can be seen in Figure 14, some of the predefined shapes can be stroked. The shape will be stroked with StrokeWidth as the line width in case the attribute Stroked is true. If StrokeWidth is 0 the line width of the shape will be the smallest possible on the actual output device.

Bifrost - Reference Manual

9 Predefined Shapes and Graphical Objects

9.1 LineShape

A Line composed of a
LineShape and a
Paint:
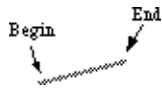
The line shape is defined by five attributes:

Begin *and* **End***.*

The beginning and ending points of the line.

Width*.*

The width of the line. If the width is 0, the line will be drawn with the smallest possible line width of the output device.

Dashes*.*

List of tuples of integers. The first integer defines the length of the first dash, the next integer defines the length of the space to the next dash and so on.
(1,2) makes a line dashed like: ▪ ▪ ▪ ▪ ▪ ▪ ▪
(4,1,1,1) makes a line dashed like: ▬ ▮ ▬ ▮ ▬ ▮ ▬

Cap*.*

Specifies how the end of the line looks. See [section 4.5.1](#).

A corresponding Line graphical object is defined with LineShape as the shape. Line uses FillLine of the Paint to draw itself (see [section 9.8.1](#)).

Bifrost - Reference Manual   [Mjølner Informatics](#)

9.2 MultiLineShape

A MultiLine composed of a
MultiLineShape and a Paint:

The multi line shape is defined by five attributes:

Points.

> PointArray defining the line. A PointArray, like the name indicates, is an array/list of Points. See the interface description for details.

Width and **Dashes**.

> Same as for Line.

Cap.

> Specifies how the ends of the line looks. See section 4.5.1.

Join.

> Specifies how the lines are joined. See section 4.5.1.

A corresponding MultiLine graphical object is defined with MultiLineShape as the shape. MultiLine uses FillMultiLine of the Paint to draw itself (see section 9.8.1).

Bifrost - Reference Manual

Mjølner Informatics

9 Predefined Shapes and Graphical Objects

9.3 TextShape

A GraphicText composed by
a
TextShape and a Paint:

Helvetica 12 point italic

The text shape can show a number of lines of text. A text shape is defined by the following attributes:

position

> Where to place the baseline of the first line of theText

theFontName

> The name of the font used: One of Courier, Times, Helvetica

theStyle

> The style of the text: Either Plain, Italic or Bold

size

> The size in points (1/72 inch) of the text drawn

underline

> Specifies if the text is to be underlined

theText

> Holds the characters of THIS(TextShape). I.e. it joins the text lines in theLines into one text, where newlines are inserted for line breaks.

theLines

> Each text element in this list corresponds to one line of text in THIS(TextShape).

numberOfLines

> Number of Texts in theLines

A corresponding text graphical object (GraphicText) is defined with TextShape as the shape. GraphicText uses FillText of the Paint to draw itself (see section 9.8.1).

9 Predefined Shapes and Graphical Objects

9.4 RectShape

A Rect composed of a
RectShape and a
Paint:
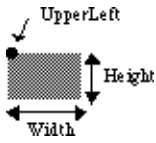
The rectangle shape is defined by the following three attributes:

UpperLeft.

Point specifying the upper left corner of the rectangle.

Width and **Height**.

The width and height of the rectangle.

A corresponding Rect graphical object is defined with RectShape as the shape. Rect uses FillRect
of the Paint to draw itself (see).

Bifrost - Reference Manual

Mjølner Informatics

9 Predefined Shapes and Graphical Objects

9.5 EllipseShape

An Ellipse composed of
an
EllipseShape and a Paint:
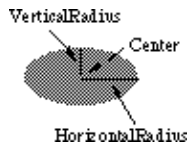
The ellipse shape is defined by the following attributes:

Center.

Point specifying the center of the ellipse.

HorizontalRadius and **VerticalRadius**.

The horizontal and vertical radius of the ellipse, respectively.

A corresponding Ellipse graphical object is defined with EllipseShape as the shape. Ellipse uses FillEllipse of the Paint to draw itself (see section 9.8.1).

Bifrost - Reference Manual

Mjølner Informatics

9 Predefined Shapes and Graphical Objects

9.6 PieShape

A Pie composed of a
PieShape and a
Paint:
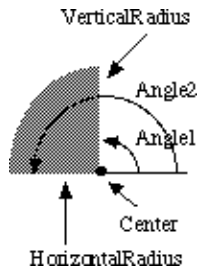
The PieSlice shape has the following attributes:

Center.

A Point specifying the center of the PieSlice shape.

HorizontalRadius and **VerticalRadius**.

The width and height of the PieSlice shape.

Angle1 and **Angle2**.

The two angles (in degrees).

A corresponding PieSlice graphical object is defined with PieShape as the shape. PieSlice uses FillPie of the Paint to draw itself (see).

Bifrost - Reference Manual

9 Predefined Shapes and Graphical Objects

9.7 ArcShape

An Arc composed of
an
ArcShape and a Paint:
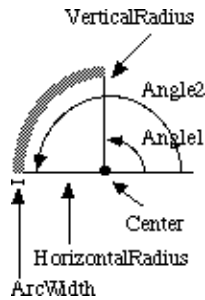
The arc shape has the following attributes:

Center*.*

A Point specifying the center of the arc shape.

HorizontalRadius *and* **VerticalRadius***.*

The width and height of the arc shape.

Angle1 *and* **Angle2***.*

The two angles (in degrees).

ArcWidth*.*

The stroke width of the arc.

A corresponding Arc graphical object is defined with ArcShape as the shape. Arc uses FillArc of the Paint to draw itself (see section 9.8.1).

Bifrost - Reference Manual

Mjølner Informatics

9 Predefined Shapes and Graphical Objects

9.8 Defining New Shapes

Predefined graphical objects are defined by describing a shape that defines the outline of the object. This is done in two steps. First, the additional attributes that specifies the shape are defined, e.g to define a line shape, the shape defines three attributes: two end points and a line width. Secondly, the operation GetShape, that calculates the actual shape in terms of line and spline segments, is defined.

In practice it could be very difficult or impossible to define the GetShape operation, e.g. for text. In case the application programmer wants to utilize special hardware/firmware operations, the Draw and Erase operations of the graphical object must be specialized together with a corresponding fill operation of the Paint (cf. section 9.8.1).

In case GetShape is not written for a particular predefined graphical object, the following operations must be further bound in the predefined shape:

> **GetBounds***.*
>
>> Return the enclosing rectangle of the shape.
>
> **ContainsPoint***.*
>
>> Determine whether the entered point is inside the shape.
>
> **Intersects***.*
>
>> Determine whether the shape of the entered graphical object intersects the shape
>
> **Within***.*
>
>> Determine whether the shape of the entered graphical object is totally inside the shape
>
> **Transform***.*
>
>> Transform all points of the shape by the transformation matrix.
>
> **HitControl***.*
>
>> Determine whether the entered point is in the neighborhood of one of the control points of the shape. See section 10.3 for a definition of neighborhood.
>
> **InteractiveCreate** *and* **InteractiveReshape***.*
>
>> Specify how to interactively create and reshape the shape. See also Chapter 10.
>
> **HiliteControls***,* **HiliteOutline***,* **HiliteBound**
>
>> Specify how to highlight the shape by using control points, the outline, or the bounding box, respectively, of the shape. See also Chapter 10.

9.8.1 Predefined Paint Operations

When a graphical object is rendered on a drawing surface, it is the responsibility of the paint of the graphical object to do the actual displaying. As stated in chapter 5, the rendering is accomplished by filling out a shape. Thus any paint has an operation FillShape that enters the shape to be filled.

To utilize the capacity of the basic graphics system a predefined shape can be drawn using a different approach. The shapes in question here could be lines with the minimal line width the output device can display, ellipses, arcs, and text.

To allow an efficient implementation of the fill operations Bifrost therefore supplies some additional filling operations for these special cases. These additional operations are only a supplement, the basic filling operation FillShape, must be able to handle all shapes. Bifrost supply the following additional fill operations:

> **FillLine** and **FillMultiLine**.
>
> > Draws the entered shape using a line drawing primitive of the basic graphics system.
>
> **FillText**.
>
> > Draws the entered TextShape using the character generator of the basic graphics system.
>
> **FillPie**, **FillArc**, **FillRect**, and **FillEllipse**.
>
> > Draws the entered shape using corresponding drawing primitives of the basic graphics system.

Bifrost - Reference Manual

Mjølner Informatics

Bifrost Graphics System

10 Interaction

This chapter explains the design and some implementation details of the interactive part of Bifrost. The chapter is mainly for the advanced user, who migth be interested in designing new interaction. As an example of designing special interaction, is when a new graphical object with a special shape is defined, as described in the previous chapter.

The reader migth prefer to skip this chapter at first reading, in case he only uses the basic graphical object, and the predefined graphical objects, and later return to the chapter when familiar with the basic usage of the Bifrost graphics system.

Bifrost - Reference Manual   Mjølner Informatics

10 Interaction

10.1 Interaction Model

Bifrost abstracts input devices used in interaction in a general interaction model. The input device is typically a pointing device like a mouse. The model is defined in pattern InteractionHandler of the canvas. InteractionHandler defines a series of virtual attributes and a general interaction loop. The usage of the interaction handler is to execute an instance of a specialization of InteractionHandler with some of the virtual attributes further bound. The attributes are:

Initialize.

Specify what to do before the interaction loop starts. Also changes the canvas' drawing mode to be XOR, to allow for immediate feedback, see below.

Motion.

Specify what to do when the user moves the pointer.

ButtonPress.

Specify what to do when the user presses a button of the pointer. ButtonInfo is a local attribute in ButtonPress that may contain device specific information, e.g. which button was pressed.

ButtonRelease.

Specify what to do when the user releases a button of the pointer.

KeyPress.

Specify what to do when the user presses a key on the keyboard.

KeyRelease.

Specify what to do when the user releases a key on the keyboard.

TerminateCondition.

Specify a condition for ending the interaction handler. Default is when the rightmost button on the pointer is released.

Terminated.

Specify what to do when the user has terminated the interaction loop. Also changes the canvas' drawing mode back to normal

In addition to these attributes the handler provides three support functions:

GetPointerLocation.

Returns the current position of the pointer.

IsModifierOn.

Returns True if the modifier entered was ON in last user action (Motion, ButtonPress, ButtonRelease, KeyPress or KeyRelease).

DoubleClick.

Returns True if the last button press was a double click.

The action part of an instance of the pattern InteractionHandler performs the following sequence of code:

```
(#
do Initialize;
   Loop and call Motion, ButtonPress, ButtonRelease, KeyPress or
   KeyRelease, depending on user action, until
   TerminateCondition returns True;
   Terminated;
#)
```

As the reader migth have noticed, the interaction handler, when excuted, temporarily replaces the event handler of the canvas and processes all events until terminated.

The InteractionHandler pattern is used to implement the interaction operations of the shapes. The following is an example of how the feedback for InteractiveCreate might be implemented for the predefined shape LineShape, using an InteractionHandler:

```
RubberLine: InteractionHandler
  (# mousePoint, anchorPoint: @Point; (* Device coords *)
     themodifier: @modifier; (* the modifier used for constrains *)
     stopinteraction: @boolean; (* stop when true *)
     x,y: @integer; (* temporary variables *)

     Initialize::
       (# do (anchorpoint, mousePoint) -> immediateLine #);
     Motion::
       (#
       do (anchorpoint, mousePoint) -> immediateLine;
          GetPointerLocation -> mousePoint;
          (if themodifier -> isModifierOn then
             (* constrain the angles *)
             (mousepoint.x-anchorpoint.x) -> abs -> x;
             (mousepoint.y-anchorpoint.y) -> abs -> y;
             (if y > x then (* constrain to vertical *)
                 anchorpoint.x -> mousepoint.x
              else (* constrain to horizontal *)
                 anchorpoint.y -> mousepoint.y
             if)
          if);
          (anchorpoint, mousePoint) -> immediateLine;
       #);
     ButtonPress:: (# do true -> stopinteraction #);
     TerminateCondition:: (# do stopinteraction -> res #);
     Terminated::
       (# do (anchorpoint, mousePoint) -> immediateLine #);
enter (anchorpoint, mousepoint, themodifier)
exit  mousepoint
#);
```

The interaction obtained by the above handler works as follows: a rubberband line is spanned between the anchorpoint point and the pointer location, following the pointer movements. In case the modifier is on (e.g. Shift is down) then the angle of the line is constrained to multiples of 90 degrees. The interaction terminates when the pointer button is pressed again.

10 Interaction

10.2 Feedback

Bifrost supports two different forms of feedback. One is the feedback generated when the user is creating or modifying a graphical object. The primives in Bifrost for this kind of feedback is presented in section 10.2.1 and 10.2.2 below.

The second kind is the feedback used, e.g., to identify a of selection of graphical objects, e.g. by highlighting of the control points or outlining the shape. This is elaborated upon in section 10.3, which also presents other interaction facilities of the Shape pattern.

Finally section 10.4 deals with the notion of modifiers, i.e. pseudo-buttons used to modify the meaning of another key or a mouse button being pressed.

10.2.1 Canvas Primitives for Feedback

Feedback drawing is done in immediate mode, that is, the feedback is not a graphical object with a shape and a paint defined. Immediate drawings are not inserted into the canvas picture.

Immediate mode drawing is supported in the Canvas by the following operations, which should typically be performed in XOR mode to allow easy erasing (by simply redrawiingthe feedback a second time). Notice, that the Hilite and Unhilite patterns of graphical objects, as well as Initialize of the InteractionHandler pattern of Canvas automatically puts the Canvas into XOR mode.

SetImmediateLineWidth.

Set the line width for subsequent immediate drawings.

ImmediateLine.

Draws a line between the two points specified as arguments.

ImmediateDot.

Takes one point as argument and draws a dot of the size of one device pixel at the point.

ImmediateSpot.

Takes one point as argument and draws a small filled square (approx. 2x2mm) around the point.

ImmediateMultiLine.

Draws a line between the points in the PointArray specified as arguments.

ImmediateRect.

Takes one point, a width, and a height as arguments and draws a rectangle with Upperleft at the point.

ImmediateText.

Enters a text (possibly with embedded newlines), a position, and the text attributes FontName, Style, Size, and UnderLine, and draws the text at the position in the specified way.

10.2.2 Segment Primitives for Feedback

The building blocks of the shape, LineSegment and SplineSegment, define an operation DrawRubberBand, constructed by the immediate primitives above, to draw feedback:

> **DrawRubberBand**.
>
> > Enters a point (NewPoint), an index into spline control points (ignored if the segment is a line segment), and a reference to the next segment (NextSeg) and draws a rubber line in either of the following two ways: **LineShapeSplineSegment**This LineShape example has four line segments where two participate in the interaction. Draws a local spline rubberband around the control point specified at control index. NextSeg is used when the control index is the last index of the spline.

THIS(Line Segment) NewPoint Control index NewPoint NextSeg
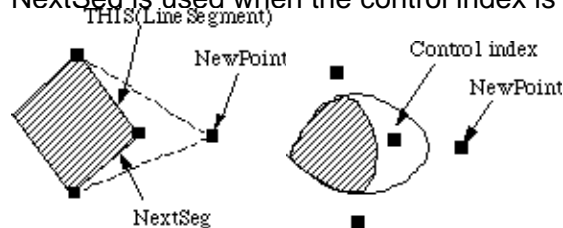
10 Interaction

10.3 Interaction Facilities in the Shape

Since the shape of a graphical object defines the outline of the object, the shape must define how to, interactively, create and modify itself. This is accomplished in the operations InteractiveCreate, InteractiveReshape and InteractiveCombine. The operations use the general InteractionHandler and feedback primitives described above.

> ### *InteractiveCreate*.
>
> > Takes a beginning point and a modifier as arguments and starts an interaction loop letting the user define the outline of a shape. When the loop is terminated the control points of the shape are set accordingly. The operation is most commonly used from InteractiveCreateShape of a graphical object.
>
> ### *InteractiveReshape*.
>
> > Takes a point as argument and starts an interaction loop letting the user reshape the shape at the control point in the neighborhood of the point (obtained by using, say, HitControl). The operation is most commonly used from InteractiveReshape of a graphical object.
>
> ### *InteractiveCombine*.
>
> > Takes a beginning point and a modifier as argument and starts an interaction loop letting the user create a shape. When the loop is terminated the new shape is combined with the original shape by using the CombineShape operation. The operation is most commonly used from InteractiveCombineShape of a graphical object.

10.3.1 Neighborhood

The concept of neighborhood is used in some of the operations presented. Neighborhood is defined as follows:

> *A point P is said to be in the neighborhood of another point Q if P is inside a square with Q as center and a given side length.*

The length of the sides defaults to 2 mm, but may be changed by the programmer.

10.3.2 Direct changing of Control Points

A shape can also be manipulated by adding a new control point to the shape, or by deleting a control point from the shape. The following two operations support manipulation of control points. They are especially useful in interaction.

Insert.

Takes two points as parameter. If the first point is in the neighborhood of an existing control point, the second point is added as a new control point between the neighbor point and the next control point of the neighbor point.

Delete.

Takes a point as parameter and, if there is one, deletes a control point in the neighborhood of the parameter point.

10.3.3 Shape Highlighting

Highlighting a graphical object is also part of interaction and interaction feedback, and is handled by the shape of the graphical object by instances of specializations of a special HiliteDesc pattern. The HiliteDesc pattern enters three parameters: The canvas to present the feedback in, a boolean indicating whether the feedback is to be drawn or erased*[9]*, and a transformation matrix, which will be applied to the feedback before it is drawn in the canvas. The following three predefined specializations of HiliteDesc define how to highlight and unhighlight the shape in three standard ways. The actual way of highlighting the shape is determined by the variable DrawHilite. It references one of the Hilite operations. The application programmer can easily extend the highlighting scheme by adding new operations.

An ellipse with highlighted
control points:
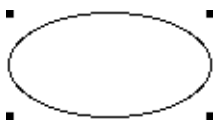
HiliteControls.

Draws small squares at the locations of the control points. The concept of control points can in this context be a bit different than used earlier. For example, the control points of an ellipse are the four corners of the bounding box of the ellipse. These corners can naturally be manipulated interactively to modify the ellipse, in contrast to the control points that are used to generate the ellipse shape in the earlier sense.

HiliteOutline.

Highlights the shape by drawing a curve along the shape. Draws by default the thinnest possible line, but another line width may be specified in the parameter HiliteWidth.

HiliteBound.

Highlights the shape by drawing a rectangle along the bounding box. Draws by default the thinnest possible line, but another line width may be specified in the parameter HiliteWidth.

DrawHilite.

Refers to one of the above Hilite operations, and is the attribute of the shape, which is invoked by the Hilite and Unhilite patterns of Graphical Objects.

10.3.4 Query Functions

Four operations are defined as query functions of the shape.

> ### *HitControl.*
>
>> Enters a point and if this point is in the neighborhood of a control point, the control point is exited. Otherwise NONE is exited.
>
> ### *ContainsPoint.*
>
>> Takes a point as argument and reports whether the point is inside the shape or not.
>
> ### *Intersects* and *Within.*
>
>> Takes a shape as argument, and reports whether it intersects or is totally inside this shape, respectively.

---

[9] You may have noticed some lack of consequence in defining how to draw and erase feedback: The Graphical Object defines both Hilite and UnHilite patterns, whereas the Shape uses a boolean to control this. Also the Canvas defines primitives that allow for XOR drawing, which means, that there is no need for distinguishing between drawing and erasing of feedback. Besides the fact, that the Graphical Object needs to know the "state" of the feedback (drawn or erased), the reasons for these different views on drawing versus erasing are purely historic.

Bifrost - Reference Manual   Mjølner Informatics

10 Interaction

10.4 Modifiers and constraints

Several of the interaction methods previously presented take a modifier as one of their arguments. This section elaborates on modifiers, and presents the constraints they impose on the interactions.

A keyboard modifier is a "pseudo-key" on the keyboard, that when pressed (in the following called ON) during a normal key press, will modify the meaning of the action. Usually there are at least three modifiers on a keyboard: The Shift key, the Control key, and the Meta key. On the keyboard, the Meta key is often labelled something other than Meta: On some Hewlett Packard keyboards it is labelled Extend Char, on some Sun workstation keyboards there are two Meta keys, labelled Left and Right, respectively, on some Sun SPARC keyboards it is labelled Alt, on most Macintosh keyboards it is labelled Alt, etc...

As mentioned a modifier key is not a normal key, e.g. it will not invoke the onKeyDown virtual of a Canvas eventhandler, if the modifier key is pressed alone. Instead the modifier changes (modifies) the meaning of the normal keys, if the modifier is held down when the normal key is pressed.

> ***Shift***
>
> > makes the character typed become upper case. Technically the 5'th bit of the 7 or 8 bit a character is represented by, is set.
>
> ***Control***
>
> > clears the 7'th bit of the numerical value of the character.
>
> ***Meta***
>
> > sets the 8'th bit of the numerical value of the character.

Modifiers can also be used during interaction with the mouse. This does not change anything directly, but is usually used to modify the feedback during the interaction. This is why the onButtonDown virtual of a Canvas eventhandler contains some booleans, indicating if the corresponding modifier was ON when the mouse button was pressed. E.g., if shiftmodified is true, it means that the shift modifier was ON when the mouse button was pressed.

InteractiveCreateShape, InteractiveReshape, and InteractiveMove in the Canvas pattern, and the corresponding methods of graphical objects and shapes all have an enter parameter called theModifier, that is used to specify what modifier to use to make the interaction constrained. Thus if ShiftModifier is used, it means that if holding down the Shift key during the interaction, the interaction will be constrained in some way, see below. A pseudo modifier called NoModifier has been defined to specify that all modifiers should be ignored, i.e., the interaction should not be constrainable.

10.4.1 Default constraints in Bifrost

Bifrost contains a small graphical editor, bdraw, residing in the directory `~beta/bifrost/bdraw`. Compiling and running this allows, the interaction forms of the different graphical objects, and the constraints the modifiers impose on them can be tried in practice. In bdraw, Shift is used as the modifier.

Here is a short overview of the interaction forms when creating, moving and reshaping the different graphical objects:

InteractiveCreate:

Rect:

> The feedback is a "rubber rectangle", defined by the start point and the position of the mouse. If theModifier is ON, the Rect is constrained to be a square. The interaction stops when the mouse is clicked.

Ellipse:

> The feedback is a "rubber ellipse" defined by the start point and the mouse position. If theModifier is ON, the Ellipse is constrained to be a circle. The interaction stops when the mouse is clicked.

GraphicalObject:

> Control points are added by clicking the left mouse button. The feedback is a "rubber line" from the previous control point added to the mouse position, and another line from the start point to the mouse position. If theModifier is ON, SplineSegment control points are added, otherwise LineSegment. The interaction stops when the right mouse button is clicked. On machines with only one mouse button the interaction is stopped by double-clicking the mouse button.

PieSlice:

> The interaction has two phases: First a rectangle with an inscribed ellipse is laid out, to define what ellipse the PieSlice should be a slice of. This phase is much like InteractiveCreate of an Ellipse. The second phase is determining the two angles defining the slice. This is done using "rubber lines" from the center to the periphery of the ellipse, in direction towards the mouse position. Each of the two angles are set when the mouse is clicked. When the last angle is determined, the interaction stops. If theModifier is ON, in the first phase, the ellipse is constrained to be a circle. In the second phase, angles are constrained to be multiples of 45 degrees.

GraphicText:

> The interaction is done via the keyboard. Characters are typed in the normal way, and typing <ESCAPE> ot clicking the mouse will end the interaction. During the interaction, the end of the text being typed is marked with a vertical bar ("insertion point").

Line:

The feedback is a "rubber line" from the start point to the mouse position. If theModifier is ON, the angles of the rubber line is constrained to multiples of 45 degrees. The interaction is stopped by clicking the mouse.

MultiLine*:*

Control points are set using the left mouse button. During this phase, the interaction is a "rubber line" from the previous control point to the mouse position. The interaction is stopped by clicking the right mouse button. If theModifier is ON, the angles of the rubber line is constrained to multiples of 45 degrees. On machines with only one mouse button the interaction is stopped by double-clicking the mouse button.

Arc*:*

The interaction is like InteractiveCreate for PieSlice, except that "moving points" on the periphery are used instead of "rubber lines" from the centre to the periphery during specification of angles in the second phase.

InteractiveMove*:*

All objects*:*

The outline of the graphical object follows the movements of the mouse. If theModifier is held down, the movement is constrained to horizontal and vertical directions. The interaction stops when the mouse button is released.

InteractiveReshape*:*

All objects*:*

For all kinds of graphical objects[10] InteractiveReshape is initiated by grabbing a control point and dragging it around, thus causing the shape to be altered. In GraphicalObject, the theModifier argument of InteractiveReshape is currently ignored, but for the other object kinds, if theModifier is ON, the interaction is constrained in the same way as during InteractiveCreate. When reshaping a PieSlice or an Arc, grabbing one of the "corners" will change either the horizontal radius or the vertical radius of the object, whereas grabbing one of the two control points on the periphery defining the angles will change the corresponding angle. [10] Except for GraphicText, for which InteractiveReshape is not yet implemented
Bifrost - Reference Manual *Mjølner Informatics*

Bifrost Graphics System

11 Saving Pictures in Files

To be able to save and load graphical objects, Bifrost uses Encapsulated PostScript files with special comments. This means that a Bifrost application is able to read a PostScript file generated by it self or another Bifrost application.

Ideally we would like to be able to save a number of pictures each as a new page in the PostScript file. This is currently not possible, but the user will be able to save one Bifrost canvas per file.

Bifrost - Reference Manual

11 Saving Pictures in Files

11.1 Saving a Canvas

To save a canvas to a file all you need to do is print it to that file:

```
PSfile.openWrite;
PSfile.startEPSfile;
((0,0,595,822),true,1,PSfile[]) -> myCanvas.writeEPS;
PSfile.endEPSfile;
PSfile.close;
```

For a full-fledged example, refer to `storepicture.bet` in ~beta/demo/bifrost.

11 Saving Pictures in Files

11.2 Loading a Canvas

Loading a canvas is as easy as writing it (as long as you do not create any specializations of the standard graphical objects - see below):
```
PSfile.openRead;
PSfile.skipHeaders;
PSfile[] -> myCanvas.loadPicture -> somePicture[];
PSfile.close;
myCanvas[] -> somePicture.draw;
```

Refer to `loadpicture.bet` in ~beta/demo/bifrost for a complete example.

Bifrost - Reference Manual

11 Saving Pictures in Files

11.3 Saving and Loading Specialized Objects

It can often be useful to define specializations of the predefined graphical objects, often with additional attributes. To be able to save these kind of objects you have to tell Bifrost three things:

1. How to save the user-defined attributes
2. How to load the user-defined attributes
3. How to instantiate new objects

11.3.1 Writing user-data

A very crude method, far from the more sophisticated persistent store method, is used for storing user-defined attributes or user-data. Bifrost simply expects the user to encode the user-data in a one line text string (note: it is important that only one line is used). Furthermore the one line string should start with the PostScript comment character % if you want to be able to print the file as a regular PostScript file. Further bind the writeUserData virtual to write user-data. See `squarelib.bet` in ~beta/demo/bifrost`for an example.`

11.3.2 Reading user-data

Reading user-data is simply done by further binding the readUserData virtual. Reading user-data is the inverse of writing user-data.

11.3.3 Creating New Objects

To be able to instantiate objects of the proper type, Bifrost uses the name of the pattern as identification. This means that the user should keep the following in mind when defining specialized graphical objects:

Do not use anonymous patterns for your graphical objects. I.e. avoid
```
&Rect(# ... init::(# do ... #) ... #)[] -> draw
```

Use distinct names for all patterns you want to save in a given file.

Keep it simple: If you plan to save a very complex data structure along with your graphical objects, you should probably consider splitting your objects in to two parts: one for the graphics (some specialization of AbstractGraphicalObject) and one for your datastructure, which you then can save using a persistent store.

Bifrost - Reference Manual

Bifrost Graphics System

12 Bifrost and Lidskjalv

As has been mentioned in the previous chapters, the current implementation of Bifrost is based on the Lidskjalv User Interface Toolkit, also known as guienv, see [MIA 94-27]. As it was also mentioned in a footnote in the Canvas description, the Canvas pattern is in the current implementation named BifrostCanvas. This chapter tries to give an overview of the current situation with respect to such overlaps and inconsistencies between Lidskjalv and Bifrost.

The Lidskjalv library and the Bifrost library has been designed independently. This is the reason that there is some overlap in functionality, in the implementation of Bifrost under Lidskjalv.

Bifrost - Reference Manual   Mjølner Informatics

12 Bifrost and Lidskjalv

12.1 BifrostCanvas and Lidskjalv Canvas

Both Lidskjalv and Bifrost have a Canvas concept. The Lidskjalv Canvas is designed as a sort of "container" for WindowItems. In this respect it resembles the Bifrost Canvas, which can be thought of as a sort of "container" for graphical objects.

The current implementation of Canvas in Bifrost is named BifrostCanvas, and it is a specialization of the Canvas pattern in Lidskjalv. This means that you can combine Bifrost graphics and Lidskjalv window items in a BifrostCanvas.

It is being discussed to rename the Lidskjalv Canvas pattern to another name with a slightly less "graphical" flavor, and to re-rename the BifrostCanvas to Canvas as in the previous non-Lidskjalv based Bifrost implementations.

Bifrost - Reference Manual   Mjølner Informatics

12 Bifrost and Lidskjalv

12.2 Overlapping Data Types

The Lidskjalv fragment group graphmath defines, among other things the following patterns:

- point
  which is analog to the Bifrost pattern Point
- rectangle
  which is analog to the Bifrost pattern Rectangle
- matrix, IDmatrix, moveMatrix, scaleMatrix, rotateMatrix
  which are almost identical to the correspondingly named patterns in Bifrost (they originate from Bifrost)
- ovalAngle and circleAngle
  which are identical to the EllipseAngle and CircleAngle patterns of Bifrost (also originating from Bifrost).

These overlaps in names may sometimes lead to "Illegal Assignment" errors in compilations, and "Qualification Error" at runtime, if you mix Lidskjalv and Bifrost code. These kind of errors may most times be solved by qualifying the references with either THIS(Guienv) or THIS(Bifrost).

In a future implementation, these attributes will have been replaced by one common set of patterns.

Bifrost - Reference Manual

[Mjølner Informatics](#)

12 Bifrost and Lidskjalv

12.3 Lidskjalv Graphics and FigureItems

The Lidskjalv fragment groups graphics and figureitems contain a simple set of graphics routines to allow for some graphics in Lidskjalv. Both fragments are based on the notation of a Pen, and whereas graphics defines a procedural graphics library with "draw" and "fill" operation (but with no automatic refresh-handling like immediate drawings in the Bifrost Canvas), the figureitems resemble Bifrost predefined graphical objects somewhat. They can be thought of as a simplified "light-weight" graphical library to use as an alternative to Bifrost in Lidskjalv.

Notice, however, that the figureitems in Lidskjalv are present mostly for historical reasons, and that it is being discussed to replace them with the Bifrost equivalents.

Bifrost - Reference Manual

Mjølner Informatics

Bifrost Graphics System

Bibliography

[Edelsbrunner 80]

H. Edelsbrunner:
Dynamic Rectangle Intersection Searching.
Technische Universität Graz. February 1980.

[Foley 90]

James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes:
Computer Graphics - Principles and Practice,
Addison-Wesley, The System Programming Series, 2, 1990.

[Newman 81]

William M. Newman and Robert F. Sproull:
Principles of Interactive Computer Graphics,
McGraw-Hill Book Company, 1981.

[Poskanzer]

Jef Poskanzer:
Portable BitMap, GrayMap, and PixMap, (PBMPLUS)
Unix Manual Pages.

Bifrost - Reference Manual   Mjølner Informatics

Bifrost Interface

```
ORIGIN '~beta/guienv/guienv';
LIB_DEF 'bifrost' '../lib';
BODY 'private/Impl/BifrostImpl';
INCLUDE '~beta/containers/list';
INCLUDE '~beta/basiclib/math';

(* Bifrost - An Interactive Object Oriented Device
 * Independent Graphics System
 *
 * Refer to   DAIMI IR-100 - Internal Report
 *            Computer Science Department
 *            Aarhus University, Denmark
 *
 * COPYRIGHT
 *        Copyright Mjolner Informatics, 1990-94
 *        All rights reserved.
 *)
```

Various Simple Definitions

```
-- BifrostAttributes: attributes --
(* Specifications used to test for key and/or pointer modification *)
Modifier:
   (# m: @Integer; enter m do INNER exit m #);
NoModifier: Modifier
   (# ... #);
ShiftModifier: Modifier
   (# ... #);
ControlModifier: Modifier
   (# ... #);
LockModifier: Modifier
   (# ... #);
MetaModifier: Modifier
   (# ... #);
CommandModifier: Modifier
   (# ... #);

(* Constants used to specify fill rules *)
EvenOddRule: (# exit 0 #);
WindingRule: (# exit 1 #);

(* Cap styles *)
CapStyleDesc: (# s: @integer; enter s do INNER exit s #);

CapButt: CapStyleDesc(# ... #);
CapRounded: CapStyleDesc(# ... #);
CapSquare: CapStyleDesc(# ... #);

(* Join styles *)
JoinStyleDesc: (# s: @integer; enter s do INNER exit s #);
JoinMiter: JoinStyleDesc(# ... #);
JoinRound: JoinStyleDesc(# ... #);
JoinBevel: JoinStyleDesc(# ... #);

(* Fontnames to use in TextShape and GraphicText *)
fontName: integerObject(# do INNER #);
```

```
Courier: fontname(# ... #);
Times: fontname(# ... #);
Helvetica: fontname(# ... #);

(* Styles to use in TextShape and GraphicText *)
Style: integerObject(# do INNER #);
Plain: Style(# ... #);
Italic: Style(# ... #);
Bold: Style(# ... #);

MaxRGB: (* The upper limit for the range of RGB values *)
  (# max: @Integer
    ... (* Device dependent *)
  exit max
  #);

(* Constants specifying the range for hue, saturation and value *)
DefaultMaxHue: (# exit 360 #);
DefaultMaxSat: (# exit 32768 #); (* (2^15) *)
DefaultMaxVal: (# exit 32768 #); (* (2^15) *)

UnImplemented:
  (* Used to notify the user on features, that are not yet
   * implemented in Bifrost.
   *)
  (# feature: ^text
  enter feature[]
  ...
  #);
DebugGraphic: (# exit false #);
```

Mathematics

```
Point:
  (# x, y: @integer;
  enter (x,y)
  exit (x,y)
  #);
Vector:
  (# x,y: @Real;
  enter (x,y)
  exit (x,y)
  #);
Rectangle:
  (# x,y: @Integer (* Upper left *);
     width,height: @Integer
  enter (x,y,width,height)
  exit (x,y,width,height)
  #);
EqualPoint:
  (# p1,p2: @Point;
  enter (p1,p2)
  exit (p1.x=p2.x) and (p1.y=p2.y)
  #);
AddPoints:
  (# p1,p2: @Point;
  enter (p1,p2)
  exit (p1.x+p2.x,p1.y+p2.y)
  #);
```

```
SubPoints:
  (# p1,p2: @Point;
  enter (p1,p2)
  exit (p1.x-p2.x,p1.y-p2.y)
  #);
ExpandRectangle:
  (# r: @rectangle;
     e: @integer;
  enter (r,e)
  exit (r.x-e, r.y+e, r.width+2*e, r.height+2*e)
  #);
PointInRect:
  (# p: @Point;
     r: @Rectangle;
  enter (p,r)
  exit   ((r.x <= p.x) and (p.x <= r.x+r.width) and
     (r.y >= p.y) and (p.y >= r.y-r.height))
  #);
Matrix:
  (# a,b,c,d,tx,ty: @Real;
     (*   a  b  0
      *   c  d  0
      *   tx ty 1
      *)
     set:
       (# enter (a,b,c,d,tx,ty) #);
     invalidate:
       (* Call invalidate if you manipulate a, b, c, d, tx, or ty *)
       ...;
     transformPoint: @
       (# p,result: @Point;
       enter p
       ...
       exit result
       #);
     inverseTransformPoint: @
       (# p1,p2: @Point;
       enter p1
       ...
       exit p2
       #);
     transformRectangle: @
       (# r,result: @Rectangle;
       enter r
       do ...
       exit result
       #);
     inverseTransformRectangle:
       (# r,result: @Rectangle;
       enter r
       ...
       exit result
       #);
     getInverse: @
       (# get: @...;
          inverse: ^Matrix;
       do get;
       exit inverse[]
       #);
     private: @...;
  do INNER;
  exit (a,b,c,d,tx,ty)
  #);
IDMatrix:
  (* Exit an identity matrix *)
  (# ID: ^Matrix
```

Mathematics   110

```
  ...
  exit ID[]
  #);
MoveMatrix: Matrix   (* A matrix specifying a translation *)
  (# itx,ity: @Integer;
  enter (itx,ity)
  ...
  #);
ScaleMatrix: Matrix  (* A matrix specifying a scaling *)
  (#
  enter (a,d)
  ...
  #);
RotateMatrix: Matrix (* A matrix specifying a rotation *)
  (# theta: @Real;
  enter theta
  ...
  #);
MatrixMul: (* Multiply two matrices *)
  (# A,B,res: ^Matrix;
  enter (A[],B[])
  ...
  exit res[]
  #);
EllipseAngle:
  (* Returns the angle a (in radians) and cos(a), sin(a),
   * assuming that (x,y) is a point on the ellipse with center in
   * (cx,cy) and horizontal radius hr and verticalradius vr,
   * i.e. (x,y) = (cx,cy) + (hr*cos(a),vr*sin(a))
   *)
  (# cx, cy, hr, vr, x, y: @integer;
     a, cos_a, sin_a: @real;
     angle: @...;
  enter (cx, cy, hr, vr, x, y)
  do angle
  exit (a, cos_a, sin_a)
  #);
CircleAngle:
  (* Returns the angle a (in radians) and cos(a), sin(a),
   * assuming that (x,y) is a point on the circle with center in
   * (cx,cy) and radius r, for some r
   * i.e. (x,y) = (cx,cy) + (r*cos(a),r*sin(a))
   *)
  (# cx, cy, x, y: @integer;
     a, cos_a, sin_a: @real;
     angle: @...;
  enter (cx, cy, x, y)
  do angle
  exit (a, cos_a, sin_a)
  #);
```

Datatypes

```
PointArray: (* Array of points, extended when needed *)
  (# <<SLOT PointArrayAttributes: Attributes >>;

     npoints: @Integer
        (* Number of points currently in THIS(PointArray) *);

     initPoints: (* Must be called first *)
```

Mathematics   111

```
         (# initialsize: @integer;
         enter initialsize
         do ...;
         #);
    copy: (* Return a deep copy of THIS(PointArray) *)
         (# p: ^PointArray;
         ...
         exit p[]
         #);
    scanPoints:
         (* scan the points in THIS(PointArray).  Inx will be the index
          * of current
          *)
         (# current: ^Point; inx: @integer
           ...
         #);
    addPoint: @(* Add p as the last point in THIS(PointArray) *)
         (# p: @Point;
         enter p
         do ...;
         #);
    deletePoint: @(* delete the i'th point in THIS(PointArray) *)
         (# i: @Integer;
         enter i
         do ...;
         #);
    insertPoint: @(* insert p between the i'th and i+1'th point in THIS(PointArray) *)
         (# i: @Integer; p: @Point;
         enter (p, i)
         do ...;
         #);
    getPoint: @
         (* Return point no i in THIS(PointArray); 1<=i<=npoints *)
         (# i: @Integer;
            p: @Point;
         enter i
         ...
         exit p
         #);
    setPoint: @
         (* Change the value of point no i to p; 1<=i<=npoints *)
         (# i: @Integer;
            p: @Point;
         enter (p,i)
         do ...;
         #);
    firstPoint: @(* Return first point of THIS(PointArray) *)
         (# exitPoint: @Point;
         ...
         exit exitPoint
         #);
    lastPoint: @(* Return last point of THIS(PointArray) *)
         (# exitPoint: @Point;
         ...
         exit exitPoint
         #);

    private: @...;
  #);

TextList: list
  (# element:: Text;
     copy:<
         (# c: ^TextList;
         ...
         exit c[]
```

```
      #);
  do INNER;
  #);


IntegerList: list
  (# IntegerElement: (# i: @integer #);
     element:: IntegerElement;
  #);


PointArrayList: (* List of PointArrays, used internally *)
  (#
     private: @...;
     appendPointArray:
       (# p: ^PointArray;
       enter p[]
       ...
       #);
     scanPointArrays:
       (# p: ^PointArray;
       ...
       #);
     empty: booleanValue
       (# ... #);
  #);
```

Segment

```
Segment:
  (# <<SLOT SegmentAttributes: attributes>>;

     firstPoint:< (# p: @Point do INNER exit p #);
     lastPoint:< (# p: @Point do INNER exit p#);
     setFirstPoint:< (# p: @Point enter p do INNER #);
     setLastPoint:< (# p: @Point enter p do INNER #);
     nextToFirstPoint:< (# p: @Point do INNER exit p #);
     nextToLastPoint:< (# p: @Point do INNER exit p #);
     copy:< (* Returns a deep copy of THIS(Segment) *)
       (# aCopy: ^Segment;
       ...
       exit aCopy[]
       #);
     transform:<
       (* Transform all control points in THIS(Segment) by M *)
       (# M: ^Matrix enter M[] do INNER #);
     reverseOrientation:< object;

     (* INTERACTION *)
     drawRubberBand:<
       (* Draw an thin curve along THIS(Segment).  Useful when
        * drawing rubber feedback
        *)
       (# theCanvas: ^BifrostCanvas
            (* The BifrostCanvas to draw the rubberband on *);
          newPoint: @Point;
          theGOToDevice: ^Matrix;
          controlIndex: @Integer;
          nextSeg: ^Segment;
        enter
            (theCanvas[],theGOToDevice[],
```

```
        newPoint,controlIndex,nextSeg[])
     do INNER
     #);
  getControls:<
     (* Add all the defining points in THIS(Segment) to spots.  If
      * spots[] is NONE, a PointArray is instantiated.  canvasTM is
      * applied to all controls before they are appended to spots.
      * If canvasTM[] is NONE, IDmatrix is used.
      *)
     (# spots: ^PointArray;
        canvasTM: ^Matrix;
     enter (spots[], canvasTM[])
     ...
     exit spots[]
     #);

  (* PRIVATE, but virtual and hence cannot be in slots *)
  prepareReshape:< (* private *)
     (# theGOToDevice: ^Matrix;
        controlIndex: @Integer;
        nextSeg: ^Segment;
        movingp: @Point;
        theCanvas: ^BifrostCanvas;
     enter (theCanvas[],theGOToDevice[],controlIndex,nextSeg[])
     do INNER;
     #);
  endReshape:< (* private *)
     (# theGOToDevice: ^Matrix;
        finalPoint: @Point;
        controlIndex: @Integer;
        nextSeg: ^Segment;
        theCanvas: ^BifrostCanvas;
     enter
        (theCanvas[],theGOToDevice[],finalPoint,controlIndex,nextSeg[])
     do INNER;
     #);
  findSegments:< (* private *)
     (# p: @point;
        s1,s2: ^Segment;
        controlIndex: @Integer;
     enter p
     do INNER
     exit (s1[],s2[],controlIndex)
     #);
  calculatePoints:< (* private *)
     (# thePoints: ^PointArray;
        thePointList: ^PointArrayList;
     enter (thePoints[],thePointList[])
     do INNER;
     exit thePointList[]
     #);
  makeOffset:< (* private *)
     (# nextPoint: @Point;
        offsets: ^PointArray;
        width: @Integer;
     enter (offsets[],nextPoint)
     do INNER;
     #);
  makeSecondOffset:< (* private *)
     (# theShape: ^Shape;
        index: @Integer;
        offsets: ^PointArray;
     enter (theShape[],offsets[],index)
     do INNER;
     exit index
     #);
```

Segment   114

```
    writePS:<(# out: ^stream enter out[] do INNER #);
  do INNER;
  exit THIS(Segment)[]
  #);
```

Line- and Spline Segments

```
LineSegment: Segment
  (#
     begin,end: @Point;
     firstPoint::< (# do begin -> p #);
     lastPoint::< (# do end -> p #);
     setFirstPoint::< (# do p -> begin #);
     setLastPoint::< (# do p -> end #);
     nextToFirstPoint::< (# do end -> p; #);
     nextToLastPoint::< (# do begin -> p #);

     copy::< (# do INNER; ... #);
     transform::< (# ... #);
     reverseOrientation::< (# ... #);

     (* INTERACTION *)
     drawRubberBand::< (# ... #);
     getControls::<(# ... #);

     (* PRIVATE, but virtual and hence cannot be in slots *)
     writePS::<(# do ... #);
     prepareReshape::< (* private *)
       (# ... #);
     endReshape::< (* private *)
       (# ... #);
     findSegments::< (* private *)
       (# ... #);
     calculatePoints::< (* private *)
       (# ... #);
     makeOffset::< (* private *)
       (# do ... #);
     makeSecondOffset::< (* private *)
       (# do ... #);
  #);
```

Splinesegment

```
SplineSegment: Segment  (* abstract pattern *)
  (# <<SLOT SplineAttributes: Attributes >>;

     controls: ^PointArray;
     smoothness: @Real
       (* default 1.0 decrease to get a smoother spline increase to
        * get a coarser spline
        *);

     firstPoint::< (# ... #);
```

```
   lastPoint::< (# ... #);
   setFirstPoint::< (# ... #);
   setLastPoint::< (# ... #);
   nextToFirstPoint::<(# ... #);

   open:<
     (* Prepare THIS(SplineSegment) for adding control points *)
     (# startPoint: @Point;
     enter startPoint
     ...
     #);
   addControl:<
     (* Add p as a control point in THIS(SplineSegment) *)
     (# p: @Point;
     enter p ...
     #);
   insert:<
     (* Insert p as a control point after the control point at
      * position index
      *)
     (# p: @point;
        index: @integer;
     enter (p,index)
     do INNER
     #);
   delete:<
     (* Delete the control point at position index *)
     (# index: @integer;
     enter index
     do INNER
     #);
   copy::< (# do INNER; ... #);
   transform::< (# ... #);
   reverseOrientation::< (# do ... #);

   (* PRIVATE *)
   writePS::<(# do ... #);
   prepareReshape::< (* private *)
     (# ... #);
   endReshape::< (* private *)
     (# ... #);
   DrawRubberSplineDesc< (* private *)
     (# track: @Point;
        controlIndex: @Integer;
        theCanvas: ^BifrostCanvas;
     enter (theCanvas[],track,controlIndex)
     do INNER
     #);
   calculatePoints::< (* private *)
     (# splinePoints: ^PointArray;
     ...
     #);
   splineprivate: @...;

do INNER;
#); (* SplineSegment *)
```

CircularSplineSegment

```
CircularSplineSegment: SplineSegment
  (# nextToLastPoint::< (# ... #);
     copy::<(# do ... #);
     drawRubberBand::< (# ... #);

     (* PRIVATE *)
     writePS::<(# do ... #);
     DrawRubberSplineDesc::< (* private *)
       (# do ... #);
     findSegments::< (* private *)
       (# ... #);
     calculatePoints::< (* private *)
       (# ... #);
     getControls::< (* private *)
       (# ... #);
     makeOffset::< (* private *)
       (# do ... #);
     makeSecondOffset::< (* private *)
       (# do ... #);
  do INNER;
  #);
```

NoncircularSplineSegment

```
NonCircularSplineSegment: SplineSegment
  (# nextToLastPoint::< (# ... #);
     copy::< (# do ... #);
     close:
       (# ... #);
     isClosed: booleanValue
       (#  ... #);
     open::<
       (#
       ...
       #);
     addControl::<
       (#
       ...
       #);
     drawRubberBand::< (# ... #);

     (* PRIVATE *)
     writePS::<(# do ... #);
     private: @...;
     DrawRubberSplineDesc::< (* private *)
       (# do ... #);
     findSegments::< (* private *)
       (# ... #);
     calculatePoints::< (* private *)
       (# ... #);
     getControls::< (* private *)
       (# ... #);
     makeOffset::< (* private *)
       (# do ... #);
     makeSecondOffset::< (* private *)
       (# do ...#);
  do INNER;
  #);
```

AbstractShape

```
AbstractShape: Segment
  (# <<SLOT AShapeAttributes: attributes >>;

    copy::< (# do INNER; ... #);
    fillRule: @
      (* Rule to determine what is inside and what is outside
       * THIS(AbstractShape). Used, e.g. when filling
       * THIS(AbstractShape) with some Paint. Defaults to
       * WindingRule.
       *)
      (# r: @Integer;
         changed: @Boolean;  (* initialized as false *)
         changeRule: (# enter r do True -> changed #);
       enter changeRule
       do (if not changed then WindingRule -> r if);
       exit r
       #);
    invalidate:<
      (* invalidate THIS(AbstractShape), so it will be recalculated
       * next time used in fill or clip operation.
       *)
      (# ... #);
    invalid:
      (* Answer true if THIS(AbstractShape) has been invalidated *)
      (# b: @Boolean;
      ...
      exit b
      #);
    getBounds:<
      (* Return the bounding box of THIS(AbstractShape) *)
      (# bound: @rectangle;
      ...
      exit bound
      #);

    (* QUERY *)
    containsPoint:< booleanValue
      (* Answer whether thePoint is inside THIS(AbstractShape),
       * thePoint is assumed to be in coordinates relative to
       * theCanvas.
       *)
      (# theCanvas: ^BifrostCanvas;
         thePoint: @Point;
       enter (theCanvas[],thePoint)
      ...
      #);
    hotspot: @
      (* The default value of hotspot is firstpoint *)
      (# p: @Point;
         changed: @Boolean;  (* initialized as false *)
         changeHotspot: (# enter p do True -> changed #);
       enter changeHotspot
       do (if not changed then firstPoint -> p if);
       exit p
       #);

    (* HIGHLIGHTING *)
    hiliteDesc: (* Qualification for highlighting patterns *)
```

```
     (# doneInInner: @boolean;
        theCanvas: ^BifrostCanvas
          (* The BifrostCanvas to do the highlighting on *);
        draw: @boolean
          (* Should the feedback be drawn or erased ? *);
        TM: ^Matrix
          (* TM is applied before the feedback is drawn *);
        copy:< (* Return a deep copy of THIS(HiliteDesc) *)
          (# aCopy: ^hiliteDesc;
          ...
          exit aCopy[]
          #);
     enter (theCanvas[], draw, TM[])
     ...
     #);

  (* PREDEFINED HIGHLIGHTING PATTERNS *)

  hiliteControls:< hiliteDesc
     (* Highlight control points *)
     (# copy::< (# do INNER; ... #);
     do INNER; ... #);
  hiliteOutline:< hiliteDesc
     (* Highlight outline of THIS(AbstractShape). To be further
      * bound
      *)
     (# hiliteWidth: @integer
         (* The width of the lines used when highlighting outline.
          * 0 means as thin as possible (default).  Should be the
          * same as the corresponding hilitewidth.
          *);
        copy::< (# do ... #);
     do INNER
     #);
  hiliteBound:< hiliteDesc
     (* Highlight bounding box *)
     (# Width: @integer;
        copy::< (# ... #);
     do INNER; ...;
     #);

  (* The actual highlight patterns used.  drawhilite points to one
   * of hc, ho, hb or some user supplied specialization of
   * hilitedesc
   *)
  hc, ho, hb: ^hiliteDesc;
  drawHilite: ^hiliteDesc;

  (* DEFINITION LANGUAGE *)
  open:< (* Must be called first *)
     (# p: @Point enter p ... #);

  (* INTERACTION *)
  Interaction:
     (* Prefix for interaction patterns *)
     (# theCanvas: ^BifrostCanvas;
        theModifier: @Modifier;
        startPoint: @Point;
     enter (theCanvas[], startPoint, theModifier)
     do INNER;
     #);
  InteractiveCreate:< Interaction
     (* Provide feedback for creating THIS(AbstractShape)
      * interactively.  Make the feedback constrained if
      * theModifier is on. Start the interaction in startpoint.
      *);
```

AbstractShape   119

```
    InteractiveCombine:< Interaction
      (* Create a Shape interactively and combine that Shape with
       * THIS(AbstractShape).  Make the feedback constrained if
       * theModifier is on. Start the interaction in startpoint.
       *);
    InteractiveReshape:< Interaction
      (* Provide feedback for reshaping THIS(AbstractShape)
       * interactively.  Make the feedback constrained if
       * theModifier is on. Start the interaction in startpoint.
       *);

    transform::< (# ... #);
    getcontrols::< (# ... #);

    (* PRIVATE *)
    writePS::<
      (#
         invisible: @boolean;
      enter invisible
      do INNER
      #);
    privatePart: @...;
    calculatePoints::< (* private *)
      (# do ... #);

  do INNER;
  #); (* Abstract Shape *)
```

Shape

```
Shape: AbstractShape
  (* For making user defined objects *)
  (# <<SLOT ShapeAttributes: attributes >>;
    copy::< (# do INNER; ... #);
    getBounds::< (# ... #);
    containsPoint::< (# do ...; INNER #);
    currentPoint:< (* The last control point added *)
      (# p: @Point;
      do ...; INNER;
      exit p
      #);
    firstPoint::< (# do ...; INNER #);
    lastPoint::< (# ... #);
    nextToFirstPoint::<(# ... #);
    nextToLastPoint::< (# ... #);
    open::< (# ... #);

    (* DEFINITION LANGUAGE *)
    addSpline:
      (* Add Spline beginning at currentpoint.  Spline.lastpoint
       * becomes new currentpoint
       *)
      (# Spline: ^SplineSegment;
      enter spline[]
      do ...;
      #);
    lineTo:
      (* If currentPoint is a control point in a spline being
       * defined with splineTo, that spline is ended.  Add a
```

```
          * LineSegment beginning at currentPoint and ending at p.  p
          * becomes new currentPoint.
          *)
         (# p: @Point;
         enter p
         do ...;
         #);
    splineTo:
         (* If currentPoint is the end point in a line segment, a new
          * spline segment is opened. That spline segment becomes the
          * "current spline segment". Add currenPoint as the first
          * control point of the current spline segment.  Add p as a
          * control point to the current spline segment.  p becomes new
          * currentPoint.
          *)
         (# p: @Point;
         enter p
         ...
         #);
    close:< (* Should be called after the definition is finished *)
         (# ... #);

    (* QUERY FUNCTIONS *)
    isClosed: booleanValue
         (* NOTICE: an empty shape is considered closed!!*)
         (# ... #);
    isEmpty: booleanValue
         (# ... #);
    isFlat: booleanValue
         (* THIS(AbstractShape) is flat iff it contains no splines *)
         (# ... #);

    (* MANIPULATING THE SHAPE *)
    reverseOrientation:< (# do ...; INNER #);
    stroke:
         (* Change THIS(Shape) to be the shape obtained by stroking a
          * "pen" with the witdh W along THIS(Shape).  When stroking an
          * open Shape, the look of the "ends" of the resulting shape
          * is specified with capStyle.  At joining points the joining
          * style is specified by joinStyle.
          *)
         (# W: @Integer;
            capstyle: @capstyledesc;
            joinstyle: @joinstyledesc;
         enter (W, capstyle, joinstyle)
         do ...;
         #);
    insert: (* Not Yet Implemented *)
         (* If p1 is in the neighborhood of an existing control point,
          * P2 is added as a new control point is between the neighbor
          * point and the next point.
          *)
         (# p1, p2: @point;
         enter (p1,p2)
         ...
         #);
    delete: (* Not Yet Implemented *)
         (* If p is in the neighborhood of an existing control point,
          * this control point is deleted
          *)
         (# p: @point;
         enter p
         ...
         #);

    (* COMBINING SHAPES *)
```

```
appendShape: (* Not Yet Implemented *)
  (* Add sourceShape to THIS(Shape).  Place
   * sourceShape.firstPoint in THIS(Shape).lastPoint by
   * translating the entire sourceShape.  This is the only
   * transformation involved.  After the operation,
   * THIS(Shape).lastPoint is the translated
   * sourceShape.lastPoint.  sourceShape cannot consist of
   * circularSplines only.
   *)
  (# sourceShape: ^Shape;
  enter sourceShape[]
  ...
  #);
connectShape: (* Not Yet Implemented *)
  (* Add sourceShape to THIS(Shape).  TM is applied to
   * sourceShape before the addition.  THIS(Shape).lastpoint is
   * connected to sourceShape.firstPoint with a line segment.
   * After the operation, THIS(Shape).lastPoint is the
   * translated sourceShape.lastPoint.
   * sourceShape cannot consist of circularSplines only.
   *)
  (# TM: ^Matrix;
     sourceShape: ^Shape;
  enter (TM[],sourceShape[])
  ...
  #);
connectShapeSmooth: (* Not Yet Implemented *)
  (* Add sourceShape to THIS(Shape).  TM is applied to
   * sourceShape before the addition.  THIS(Shape).lastpoint is
   * connected to sourceShape.firstPoint with a spline segment
   * constructed from the last two points in THIS(Shape) and
   * sourceShape.firstPoint.  After the operation,
   * THIS(Shape).lastPoint is the translated
   * sourceShape.lastPoint.  sourceShape cannot consist of
   * circularSplines only.
   *)
  (# TM: ^Matrix;
     sourceShape: ^Shape;
  enter (TM[],sourceShape[])
  ...
  #);
combineShape:
  (* Add sourceShape to THIS(Shape).  TM is applied to
   * sourceShape before the addition.  sourceShape and
   * THIS(Shape) do *not* become connected.  At least one of
   * THIS(Shape) and sourceShape must be closed.  If sourceShape
   * is closed, THIS(Shape).lastPoint is unchanged.  If
   * sourceShape is open, THIS(Shape).lastPoint is
   * sourceshape.lastPoint after the operation.
   *)
  (# TM: ^Matrix;
     sourceShape: ^Shape;
  enter (TM[],sourceShape[])
  do ...;
  #);

(* HIGHLIGHTING *)
hiliteOutline::< (# ... #);

(* INTERACTION *)
InteractiveCreate::< (# do ...; INNER  #);
InteractiveCombine::< (# do ...; INNER  #);

InteractiveReshape::< (# do ...; INNER  #);
transform::< (# do ...; INNER  #);
```

```
   getControls::<(# do ... #);

   (* PRIVATE *)
   findSegments::< (* private *)
     (# do ... #);
   writePS::<(# do ... #);
do INNER;
#);  (* Shape *)
```

PredefinedShape

```
PredefinedShape: AbstractShape
  (#
     CalculateShape:<
       (* Return (approximating) Shape, if possible *)
       (# s: ^Shape
       do INNER
       exit (# ... exit s[] #)
       #);
     invalidate::<(# ... #);
     containsPoint::<(# ... #);
     transform::<(# do ...; INNER #);

     (* Patterns behaving like standard "types", but that have the
      * side-effect of invalidating THIS(PredefinedShape) when
      * changed.
      *)
     invalidatePoint:
       (# p: @Point; enter (# enter p do Invalidate #) exit p #);
     invalidateInteger: integerValue
       (# enter (# enter value do Invalidate #) #);
     invalidateReal:
       (# r: @Real; enter (# enter r do Invalidate #) exit r #);
     invalidateDash:
       (* For instance {1,2,4,2} yields '=  ====  =  ====  =' etc. *)
       (# d: ^IntegerList;
       enter (# enter d[] do invalidate #)
       exit d[]
       #);
     invalidateCapStyle:
       (# c: @CapStyleDesc;
       enter (# enter c do invalidate #)
       exit c
       #);
     invalidateJoinStyle:
       (# j: @JoinStyleDesc;
       enter (# enter j do invalidate #)
       exit j
       #);
     writePS::<(# do ... #);
     prePrivate: @...;
  do INNER;
  #);
```

LineShape

```
LineShape: PredefinedShape
  (# <<SLOT LineShapeAttributes: attributes>>;

    firstPoint::<(# do begin -> p #);

    begin: @InvalidatePoint;
    end: @InvalidatePoint;
    width: @InvalidateInteger;
    dashes: @InvalidateDash; (* Not Yet Implemented *)
    cap: @InvalidateCapStyle;

    coordinates:
      (# enter (begin, end) exit (begin, end) #);
    open::<(# ... #);
    getBounds::< (# do ...; INNER #);
    containsPoint::<(# ... #);
    getControls::<(# ... #);
    copy::< (# do INNER; ... #);

    (* HIGHLIGHTING *)
    hiliteOutline::< (# do INNER; ... #);

    (* INTERACTION *)
    interactiveCreate::<(# do ...; INNER #);
    interactiveReshape::<(# do ...; INNER #);

    writePS::<(# do ... #);
    transform::<(# ... #);
    CalculateShape::<  (* private *)
      (# ... #);

  do INNER;
  #);
```

MultilineShape

```
MultiLineShape: PredefinedShape
  (# <<SLOT MultiLineShapeAttributes: attributes>>;

    firstPoint::< (# ... #);
    lastPoint::< (# ... #);
    points: @
      (# p: ^PointArray;
      enter (# enter p[] do invalidate #)
      exit p[]
      #);
    width: @InvalidateInteger;
    dashes: @InvalidateDash; (* Not Yet Implemented *)
    cap: @InvalidateCapStyle;
    join: @InvalidateJoinStyle;

    open::< (# ... #);
    addPoint: (* Add p at the end of points *)
      (# p: @point;
      enter p
      ...
```

```
      #);
   deletePoint: (* Delete p at from points *)
      (# p: @point;
      enter p
      ...
      #);
   insertPoint: (* Insert p in points at position i *)
      (# p: @point;
         i: @integer
      enter (p,i)
      ...
      #);
   getPoint:
      (* Return point no i in THIS(MultiLineShape); 1<=i<=npoints *)
      (# i: @Integer;
         p: @Point;
      enter i
      ...
      exit p
      #);
   setPoint:
      (* Change the value of point no i to p; 1<=i<=npoints *)
      (# i: @Integer;
         p: @Point;
      enter (p,i)
      ...
      #);
   closestLineSegment:
      (# p: @point; i: @integer
      enter p
      do ...
      exit i
      #);
   getBounds::<(# do ...; INNER #);
   containsPoint::<(# ... #);
   getControls::<(# ... #);
   copy::< (# do INNER; ... #);

   (* HIGHLIGHTING *)
   hiliteOutline::< (# do INNER; ... #);

   (* INTERACTION *)
   interactiveCreate::<(# do ...; INNER #);
   interactiveReshape::<(# do ...; INNER #);
   writePS::<(# do ... #);
   transform::<(# ... #);

   calculateShape::<  (* private *)
      (# ... #);
do INNER;
#);
```

TextShape

```
TextShape: PredefinedShape
  (# <<SLOT TextShapeAttributes: attributes>>;

     firstPoint::< (# do position -> p #);
     initText: (* Specify several attributes simultaneously *)
```

```
   (#
   enter
       (position, theFontname, theStyle, size, underline, theText)
   #);
position:
   (* Where to place the baseline of the first line of theText *)
   (# p: @Point;
   enter (# enter p ... #)
   ...
   exit p
   #);
theFontName: (* one of Courier, Times, Helvetica *)
   (# nam: @fontname;
   enter (# enter nam ... #)
   ...
   exit nam
   #);
theStyle: (* Either Plain, Italic or Bold *)
   (# sty: @Style;
   enter (# enter sty ... #)
   ...
   exit sty
   #);
size: (* The size in points (1/72 inch) of the text drawn *)
   (# siz: @Integer;
   enter (# enter siz ... #)
   ...
   exit siz
   #);
underline: (* Specifies if the text is to be underlined *)
   (# ul: @Boolean;
   enter (# enter ul ... #)
   ...
   exit ul
   #);
theText:
   (* Holds the characters of THIS(TextShape). i.e. it joins
    * the text lines in theLines into one text, where newlines
    * are inserted for line breaks.
    *)
   (# t: ^Text;
   enter (# enter t[] ... #)
   ...
   exit t[]
   #);
theLines:
   (* Each text element in this list corresponds to one line
    * of text in THIS(TextShape).
    *)
   (# l: ^TextList;
   enter (# enter l[] ... #)
   ...
   exit l[]
   #);
numberOfLines: integerValue
   (* Number of Texts in theLines *)
   (# ...
   #);
getBounds::<(# do ...; INNER #);
containsPoint::<(# ... #);
getControls::<(# do ...; INNER #);
copy::< (# do INNER; ... #);

(* HIGHLIGHTING *)
hiliteOutline::< (# do INNER; ... #);
```

TextShape   126

```
   (* INTERACTION *)
   interactiveCreate::<
     (# lastCh: @char; (* Last character typed in interaction *)
     do ...; INNER
     exit lastCh
     #);
   interactiveReshape::<
     (# lastCh: @char; (* Last character typed in interaction *)
     ...
     exit lastCh
     #);

   writePS::<(# do ... #);
   transform::<(# ... #);
   TextPrivate: @ ...;
   calculateShape::< (* private *)
     (# ... #);

 do INNER;
 #);
```

PieShape

```
PieShape: PredefinedShape
  (# <<SLOT PieShapeAttributes: attributes>>;

     firstPoint::<(# do center -> p #);

     center: @InvalidatePoint;
     horizontalRadius: @InvalidateInteger;
     verticalRadius: @InvalidateInteger;
     (* Use: 0 <= angle1 <= 360 a1 <= angle2 <= 360+angle1     *)
     angle1: @InvalidateReal;
     angle2: @InvalidateReal;

     open::<(# do ...; INNER #);
     getBounds::<(# do ...; INNER #);
     containsPoint::<(# do ...; INNER #);
     getControls::< (# do ...; INNER #);
     copy::< (# do INNER; ... #);

     (* HIGHLIGHTING *)
     hiliteOutline::< (# do INNER; ... #);

     (* INTERACTION *)
     interactiveCreate::<(# do ...; INNER #);
     interactiveReshape::<(# do ...; INNER #);

     writePS::<(# do ... #);
     transform::<(# ... #);
     calculateShape::< (* private *)
       (# ... #);
  do INNER
  #);
```

ArcShape

```
ArcShape: PredefinedShape
  (# <<SLOT ArcShapeAttributes: attributes>>;

    firstPoint::<(# do center -> p #);

    center: @InvalidatePoint;
    horizontalRadius: @InvalidateInteger;
    verticalRadius: @InvalidateInteger;
    (* Use: 0 <= angle1 <= 360 a1 <= angle2 <= 360+angle1      *)
    angle1: @InvalidateReal;
    angle2: @InvalidateReal;
    arcWidth: @InvalidateInteger;

    open::<(# ... #);
    getBounds::<(# do ...; INNER #);
    containsPoint::<(# do ...; INNER #);
    getControls::<(# do ...; INNER #);
    copy::< (# do INNER; ... #);

    (* HIGHLIGHTING *)
    hiliteOutline::< (# do INNER; ... #);

    (* INTERACTION *)
    interactiveCreate::<(# do ...; INNER #);
    interactiveReshape::<(# do ...; INNER #);

    writePS::<(# do ... #);
    transform::<(# ... #);
    calculateShape::< (* private *)
      (# ... #);

  do INNER
  #);
```

StrokeableShape

```
StrokeableShape: PredefinedShape
  (# stroked: @Boolean;
    strokewidth: @Integer;

    writePS::<(# do ... #);
    getBounds::<(# ... #);
    copy ::<(# do INNER; ... #);
  do INNER
  #);
```

RectShape

```
RectShape: StrokeableShape
  (# <<SLOT RectShapeAttributes: attributes>>;
```

```
   firstPoint::<(# do upperleft -> p #);

   upperleft: @InvalidatePoint;
   width: @InvalidateInteger;
   height: @InvalidateInteger;

   corners:
     (# lowerright: @Point;
        changeCorners:
          (# enter (upperleft,lowerright)
          ...
          #);
      enter changeCorners
      exit
         (upperleft,
         ((upperleft.p.x+width),
         (upperleft.p.y+height)) )
     #);
   open::<(# ... #);
   getBounds::<(# ... #);
   containsPoint::<(# ... #);
   getControls::<(# ... #);
   copy::<(# do INNER; ... #);

   (* HIGHLIGHTING *)
   hiliteOutline::< (# do INNER; ... #);

   (* INTERACTION *)
   interactiveCreate::<(# do ...; INNER #);
   interactiveReshape::<(# do ...; INNER #);

   writePS::<(# do ... #);
   transform::<(# ... #);
   calculateShape::< (* Private *)
     (# ... #);

 do INNER;
 #);
```

EllipseShape

```
EllipseShape: StrokeableShape
  (# <<SLOT EllipseShapeAttributes: attributes>>;

   firstPoint::< (# do center -> p #);

   center: @InvalidatePoint;
   horizontalradius: @InvalidateInteger;
   verticalradius: @InvalidateInteger;

   geometry:
     (#
     enter (center, verticalradius, horizontalradius)
     exit  (center, verticalradius, horizontalradius)
     #);
   open::<(# ... #);
   getBounds::<(# ... #);
   containsPoint::<(# ... #);
```

```
  getControls::<(# do ...; INNER #);
  copy::< (# do INNER; ... #);

  (* HIGHLIGHTING *)
  hiliteOutline::< (# do INNER; ... #);

  (* INTERACTION *)
  interactiveCreate::<(# ... #);
  interactiveReshape::<(# do ...; INNER #);

  writePS::<(# do ... #);
  transform::<(# ... #);
  calculateShape::< (* private *)
     (# do ... #);

do INNER;
#);
```

Rasters

Raster:
```
  (* An abstract superpattern for all Rasters. A raster is a
   * rectangular grid of pixels.
   *)
  (# <<SLOT RasterAttributes: attributes>>;

    hotspot:
      (* When used in a filling operation hotspot is placed in
       * hotspot of the shape being filled. Defaults to (0,0).
       *)
      (# p: @Point;
      enter (# enter p ... #)
      exit  (# ... exit p #)
      #);
    pixel:< Object;

    init:<
      (# width, height: @integer;
      enter (width, height)
      ...
      #);
    copy:< (* Return a deep copy of THIS(Raster) *)
      (# aCopy: ^Raster;
      ...
      exit aCopy[]
      #);
    width: integerValue
      (* returns the width set by init or by read operations *)
      (# ... #);
    height: integerValue
      (* returns the height set by init or by read operations *)
      (# ... #);

    putPixel:<
      (# i, j: @integer; p: ^pixel;
      enter (i,j,p[])
      ...
      #);
    getPixel:<
```

```
      (# i, j: @integer; p: ^pixel;
      enter (i,j)
      ...
      exit p[]
      #);

    (* Private *)
    calculate:< (# ... #);
    RasterPrivatePart: @ ...;

  do INNER; calculate;
  exit THIS(Raster)[]
  #);

BitMap: Raster
  (* Raster in which the pixels are booleans *)
  (# <<SLOT BitmapAttributes: attributes>>;

    pixel::< (# b: @boolean enter b exit b #);
    init::<  (# do ...; INNER #);
    putPixel::< (# do ...; INNER #);
    getPixel::< (# do ...; INNER #);
    copy::< (# do INNER; ... #);
    writeToPBMfile: (* Not Yet Implemented *)
      (# pbmfilename: ^text;
         rawbits: @boolean
           (* If true, the RAWBITS format is used *);
      enter (pbmfilename[],rawbits)
      ...
      #);
    readFromPBMfile:
      (# pbmfilename: ^text;
      enter pbmfilename[]
      do ...;
      #);

    (* Private *)
    calculate::< (# ... #);
    BitMapPrivatePart: @ ...;
  do INNER;
  #);

GrayMap: Raster (* Not Yet Implemented *)
  (# <<SLOT GraymapAttributes: attributes>>;

    pixel::<(# g: @integer enter g exit g #);
    init::< (# ... #);
    putPixel::< (# ... #);
    getPixel::<(# ... #);
    copy::<(# do INNER; ... #);
    writeToPGMfile:
      (# pgmfilename: ^text;
         rawbits: @boolean
           (* If true, the RAWBITS format is used *);
      enter (pgmfilename[],rawbits)
      ...
      #);
    readFromPGMfile:
      (# pgmfilename: ^text;
      enter pgmfilename[]
      ...
      #);

    (* Private *)
    calculate::< (# ... #);
    GrayMapPrivatePart: @ ...;
```

Rasters   131

```
  do INNER;
  #);

PixMap: Raster
  (* Raster in which the pixels are RGB values *)
  (# <<SLOT PixmapAttributes: attributes>>;

    pixel::< (# r,g,b: @integer enter (r,g,b) exit (r,g,b) #);
    init::<
      (# maxVal: @integer; (* Maximum RGB value *)
      enter maxVal
      do ...; INNER
      #);
    putPixel::< (# do ...; INNER #);
    getPixel::< (# ... #);
    copy::< (# do INNER; ... #);
    writeToPPMfile:  (* Not Yet Implemented *)
      (# ppmfilename: ^text;
         rawbits: @boolean
           (* If true, the RAWBITS format is used *);
      enter (ppmfilename[],rawbits)
      ...
      #);
    readFromPPMfile:  (* Not Yet Implemented *)
      (# ppmfilename: ^text;
      enter ppmfilename[]
      ...
      #);

    (* Private *)
    calculate::< (# ... #);
    PixMapPrivatePart: @ ...;

  do INNER;
  #);
```

Paint

```
Paint: (* An abstract superpattern for all paint *)
  (# <<SLOT PaintAttributes: attributes>>;

    init:< object;

    copy:< (* Return a deep copy of THIS(Paint) *)
      (# aCopy: ^Paint;
      ...
      exit aCopy[]
      #);
    fill:
      (* Prefix for fill operations *)
      (# theCanvas: ^BifrostCanvas enter theCanvas[] do INNER #);
    fillShape:< fill
      (* Fill theShape with THIS(Paint) in theCanvas. *)
      (# theShape: ^Shape;
      enter (theShape[])
      ...
      #);
    fillLine:< fill
      (* Fill theLine with THIS(Paint) in theCanvas. *)
```

```
      (# theLine: ^LineShape;
      enter (theLine[])
      ...
      #);
   fillMultiLine:< fill
      (* Fill theMultiLine with THIS(Paint) in theCanvas.
       *)
      (# theMultiLine: ^MultiLineShape;
      enter (theMultiLine[])
      ...
      #);
   fillText:< fill
      (* Fill the specified text with THIS(Paint) in theCanvas *)
      (# theText: ^TextShape;
      enter (theText[])
      ...
      #);
   fillPie:< fill
      (* Fill thePie with THIS(Paint) in theCanvas. *)
      (# thePie: ^pieShape;
      enter (thePie[])
      ...
      #);
   fillArc:< fill
      (* Fill theArc with THIS(Paint) in theCanvas. *)
      (# theArc: ^arcShape;
      enter (theArc[])
      ...
      #);
   fillRect:< fill
      (* Fill theRect with THIS(Paint) in theCanvas. *)
      (# theRect: ^RectShape;
      enter (theRect[])
      ...
      #);
   fillEllipse:< fill
      (* Fill the theEllipse with THIS(Paint) in theCanvas *)
      (# theEllipse: ^EllipseShape;
      enter (theEllipse[])
      ...
      #);
   fillOther:< fill
      (* Used to fill other, e.g. user defined, shapes *)
      (# theShape: ^AbstractShape;
      enter theShape[]
      do INNER;
      #);

   (* PRIVATE *)
   writePS:<
      (# out: ^stream; shape_hotspot: @point
      enter (out[], shape_hotspot)
      do INNER
      #);
   paintprivate: @ ...;
   setSpecialPaint: (* Private *)
      (# theCanvas: ^BifrostCanvas;
         doneInInner: @boolean;
      enter theCanvas[]
      do INNER
      #);
   setCanvasPaint:< (* Private *) setSpecialPaint;
   setBorderPaint:< (* Private *) setSpecialPaint;
   SetBackgroundPaint:< (* Private *) setSpecialPaint;
do INNER;
exit THIS(Paint)[]
```

```
  #);
```

SolidColor

```
SolidColor: Paint
  (* A solid color specified relative to the RGB, HSV, or CMY color
   * spaces, or by naming the color, using one of the name patterns
   * in the fragment ColorNames.
   *)
  (# <<SLOT SolidColorAttributes: attributes>>;

     init::<(# ... #);
     copy::< (# do INNER; ... #);
     Name:
       (* Change THIS(SolidColor) to the color specified.  The color
        * names are define as descriptors in the fragment
        * 'ColorNames'.  NOTICE: This is different from earlier
        * versions of Bifrost.
        *)
       (# enter RGBvalues #);
     RGBvalues:
       (* Set or query the Red-Green-Blue values of THIS(SolidColor)
        * r, g and b all ranges from 0 to MaxRGB.
        *)
       (# r,g,b: @Integer;
          changeRGB:
            (# enter (r,g,b) ... #);
          getRGB:
            (# ... exit (r,g,b) #);
        enter changeRGB
        exit GetRGB
        #);
     HSVvalues:
       (* Set or query the Hue-Saturation-Value values of
        * THIS(SolidColor).  h, s and v are taken to range from 0 to
        * MaxHue, MaxSat and MaxVal respectively. Specializations may
        * alter the default bindings of these.
        *)
       (# h,s,v: @Integer;
          changeHSV:
            (# enter (h,s,v) do ... #);
          getHSV:
            (# do ... exit (h,s,v) #);
          MaxHue:< integerValue
            (# do DefaultMaxHue -> value; INNER #);
          MaxSat:< integerValue
            (# do DefaultMaxSat -> value; INNER #);
          MaxVal:< integerValue
            (# do DefaultMaxVal -> value; INNER #);
        enter changeHSV
        exit getHSV
        #);
     CMYvalues: (* RGB complementaries *)
       (* Set or query the Cyan-Magenta-Yellow values of
        * THIS(SolidColor).  c, m and y all ranges from 0 to MaxRGB.
        *)
       (# c,m,y: @Integer;
          changeCMY:
            (# enter (c,m,y) do ... #);
```

```
      getCMY:
          (# do ...; exit (c,m,y) #);
      enter changeCMY
      exit getCMY
      #);

   fillShape::<(# ... #);
   fillLine::<(# do INNER; ... #);
   fillMultiLine::<(# ... #);
   fillText::<(# do INNER; ... #);
   fillPie::<(# do INNER; ... #);
   fillArc::<(# do INNER; ... #);
   fillRect::<(# do INNER; ... #);
   fillEllipse::<(# do INNER; ... #);

   (* PRIVATE *)
   writePS::<(# do ... #);
   setBorderPaint::< (* Private *)
     (# ...#);
   setBackgroundPaint::< (* Private *)
     (# ...#);
   setCanvasPaint::< (* Private *)
     (# ...#);
   privatePart: @ ...;
 do INNER;
 #);
```

Predefined Graytones

```
SolidGray:
  (# g: ^SolidColor;
     percentage: @Integer;
  enter percentage
  ...
  exit g[]
  #);

SolidGrey: SolidGray (# do INNER #);
```

RasterPaint

```
RasterPaint: Paint
  (* Use thePixmap and optionally paddingSolidColor to fill out the
   * shape
   *)
  (#
     (* If paddingSolidColor[]=NONE thePixmap will be repeated when
      * filling out the shape. If not, paddingSolidColor will be used
      * to fill out any parts of the shape the pixmap doesn't cover.
      *)
     paddingSolidColor: ^SolidColor;

     thePixMap:
```

SolidColor   135

```
   (# p: ^PixMap;
    enter (# enter p[] ... #)
    exit (# ... exit p[] #)
    #);
  init::<(# ... #);
  copy::<(# do INNER; ... #);
  fillShape::<(# do INNER; ...; #);
  fillLine::<(# ... #);
  fillMultiLine::<(# ... #);
  fillText::<(# ... #);
  fillArc::<(# ... #);
  fillPie::<(# ... #);
  fillRect::<(# ... #);
  fillEllipse::<(# ... #);

  (* PRIVATE *)
  writePS::<(# do ... #);
  private: @...;
  setBorderPaint::<  (* Private *)
    (# do INNER; ... #);
  setBackgroundPaint::<  (* Private *)
    (# do INNER; ... #);
  setCanvasPaint::< (* Private *)
    (# do INNER; ... #);
do INNER;
#);
```

TiledSolidColor

```
TiledSolidColor: SolidColor
  (* A SolidColor extended with a BitMap. The BitMap will be tiled in
   * the Shape before the SolidColor is applied, and only where the
   * bits of the BitMap are true, the SolidColor will be visible.
   *)
  (#
    theTile:
      (# t: ^BitMap;
       enter (# enter t[] ... #)
       exit (# ... exit t[] #)
       #);
    init::<(# ... #);
    copy::<(# do INNER; ... #);
    fillShape::<(# ... #);
    fillLine::<(# ... #);
    fillMultiLine::<(# ... #);
    fillText::<(# ... #);
    fillArc::<(# ... #);
    fillPie::<(# ... #);
    fillRect::<(# ... #);
    fillEllipse::<(# ... #);

    (* PRIVATE *)
    writePS::<(# do ... #);
    tiledPrivate: @ ...;
    setBorderPaint::<  (* Private *)
      (# do INNER; ... #);
    setBackgroundPaint::< (* Private *)
      (# do INNER; ... #);
    setCanvasPaint::< (* Private *)
```

```
      (# do INNER; ... #);
  do INNER;
  #);
```

AbstractGraphicalObject

```
AbstractGraphicalObject: (* To be further specialized *)
  (* The graphical object is the smallest entity that can be drawn
   * in a BifrostCanvas.  It is a aggregation of a Paint and a Shape.
   * ANY graphical object MUST be initialized before used (init).
   * After a paint and a shape has been specified, it can be drawn by
   * giving the reference of it as enter parameter to the method
   * "draw" in a BifrostCanvas.  Graphical objects may also be
   * created by using InteractiveCreateShape.
   *)
(# <<SLOT AbstractGraphicalObjectAttributes: attributes>>;
  shapeDesc:< AbstractShape
    (* Specify actual shape in specializations *);
  TMDesc:<
    (# m: ^Matrix;
       transformpoint: @
         (# p: @Point enter p do p->m.transformpoint->p exit p #);
       CalcCanvasTM:<
         (# theTM: ^Matrix
          enter theTM[]
          ...
          #);
       enterTM:< (# enter m[] ... #);
       enterIt: @enterTM;
     enter enterIt
     do INNER;
     exit m[]
     #);
  (* TM describes the transformation from the coordinate system of
   * theShape (also known as GO coordinates) to the the Picture it
   * is part of, if any.
   *)
  TM: @TMDesc;

  init:< (* MUST be called first *)
    (# ... #);
  readUserData:<
    (#
       userdata: ^text;
       success: @boolean;
       parseNotification: notification
         (#
         do false -> success;
            INNER
         #);
       parseError:< parseNotification
     enter userdata[]
     do true -> success;
        INNER
     exit success
     #);
  writeUserData:<
    (#
       userdata: ^text
```

```
      do &text[] -> userdata[]; INNER
      exit userdata[]
      #);
  setPaint:<
      (* Specify the paint to use for THIS(AbstractGraphicalObject) *)
      (# enter thePaint[] do INNER #);
  getPaint:<
      (* Obtain the paint to use *)
      (# do INNER exit thePaint[] #);
  getShape:<
      (* Obtain the shape to use.  The specialization
       * PredefinedGraphicalObject returns an approximating Shape.
       * Only the specialization Shape has a corresponding SetShape.
       *)
      (# s: ^Shape
      do INNER
      exit s[]
      #);
  draw:<
      (* Draw THIS(AbstractGraphicalObject) in theCanvas.
       * Normally this is not used by the user directly.  Instead
       * THIS(AbstractGraphicalObject)[] should be given to the draw
       * method of a BifrostCanvas.
       *)
      (# doneInInner: @boolean;
         theCanvas: ^BifrostCanvas
            (* BifrostCanvas to draw THIS(AbstractGraphicalObject) on *);
      enter theCanvas[]
      ...
      #);
  erase:<
      (* Erase THIS(AbstractGraphicalObject) from theCanvas.
       * Normally this is not used by the user directly.  Instead
       * THIS(AbstractGraphicalObject)[] should be given to the erase
       * method of a BifrostCanvas.
       *)
      (# doneInInner: @boolean;
         theCanvas: ^BifrostCanvas
            (* BifrostCanvas to erase THIS(AbstractGraphicalObject) from *);
      enter theCanvas[]
      ...
      #);
  copy:< (* Return a deep copy of THIS(AbstractGraphicalObject) *)
      (# aCopy: ^AbstractGraphicalObject;
      ...
      exit aCopy[]
      #);
  getBounds:<
      (* Exit a Rectangle containing the bounding box of
       * THIS(AbstractGraphicalObject) in BifrostCanvas coordinates.
       *)
      (# r: @rectangle;
         doneInInner: @boolean;
      ...
      exit r
      #);
  hilite:< (* Highlight THIS(AbstractGraphicalObject) *)
      (# doneInInner: @boolean;
         theCanvas: ^BifrostCanvas
            (* The BifrostCanvas to do the highlighting on *)
      enter theCanvas[]
      ...
      #);
  unHilite:< (* Unhighlight THIS(AbstractGraphicalObject) *)
      (# doneInInner: @boolean;
         theCanvas: ^BifrostCanvas
```

AbstractGraphicalObject   138

```
             (* The BifrostCanvas to do the unhighlighting on *)
      enter theCanvas[]
      ...
      #);

  (* INTERACTION *)
  hitControl:<
    (* Answer whether thePoint is inside a 2x2mm box around a
     * control point of THIS(AbstractGraphicalObject).  thePoint is
     * in BifrostCanvas coordinates.  Exits reference to exact point
     * if hit, NONE otherwise.
     *)
    (# thePoint: @Point;
       res: ^Point;
    enter thePoint
    do ...;
       INNER;
    exit res[]
    #);
  interaction:
    (* Prefix for interactive operations *)
    (# theCanvas: ^BifrostCanvas
         (* The BifrostCanvas to show feedback in *);
       startPoint: @Point;
       theModifier: @Modifier;
       doneInInner: @boolean;
    enter (theCanvas[], startPoint, theModifier)
    do INNER;
    #);
  interactiveCreateShape:< interaction
    (* Initialize the shape of THIS(AbstractGraphicalObject) by
     * providing feedback in a BifrostCanvas.  Normally this is not
     * used by the user directly.  Instead
     * THIS(AbstractGraphicalObject)[] should be given to the
     * interactiveCreateShape method of a BifrostCanvas.
     *)
    (# ... #);
  interactiveCombineShape:< interaction
    (* Combine a shape with the shape of
     * THIS(AbstractGraphicalObject) by providing feedback for
     * creating the new shape in a BifrostCanvas, and then combining
     * the shape of THIS(AbstractGraphicalObject) with the obtained
     * shape.  Normally this is not used by the user directly.
     * Instead THIS(AbstractGraphicalObject)[] should be given to
     * the interactiveCombineShape method of a BifrostCanvas.
     *)
    (# ... #);
  interactiveReshape:< interaction
    (* Change the shape of THIS(AbstractGraphicalObject) by
     * providing feedback in a BifrostCanvas.  Normally this is not
     * used by the user directly.  Instead
     * THIS(AbstractGraphicalObject)[] should be given to the
     * interactiveReShape method of a BifrostCanvas.
     *)
    (# do ... #);
  interactiveMove:< interaction
    (* Move the shape of THIS(AbstractGraphicalObject) using
     * theshape.(un)hiliteoutline for feedback in the BifrostCanvas
     * THIS(AbstractGraphicalObject) is drawn in.  Calls "move" to
     * do the transformation.  Normally this is not used by the user
     * directly.  Instead THIS(AbstractGraphicalObject)[] should be
     * given to the interactiveMove method of a BifrostCanvas.
     *)
    (#
    do INNER; ...;
    #);
```

AbstractGraphicalObject   139

```
interactiveScale:< interaction (* Not Yet Implemented *)
  (* Scale the shape of THIS(AbstractGraphicalObject) using
   * theshape.(un)hiliteoutline for feedback in the BifrostCanvas
   * THIS(AbstractGraphicalObject) is drawn in.  Calls "scale" to
   * do the transformation.  Normally this is not used by the user
   * directly.  Instead THIS(AbstractGraphicalObject)[] should be
   * given to the interactiveScale method of a BifrostCanvas.
   *)
  (# ... #);
interactiveRotate:< interaction (* Not Yet Implemented *)
  (* Rotate the shape of THIS(AbstractGraphicalObject) using
   * theshape.(un)hiliteoutline for feedback in the BifrostCanvas
   * THIS(AbstractGraphicalObject) is drawn in.  Calls "rotate" to
   * do the transformation.  Normally this is not used by the user
   * directly.  Instead THIS(AbstractGraphicalObject)[] should be
   * given to the interactiveRotate method of a BifrostCanvas.
   *)
  (# ... #);

(* TRANSFORMATIONS *)
transform:<
  (* Transform THIS(AbstractGraphicalObject) by M, by multiplying
   * THIS(AbstractGraphicalObject).TM with M
   *)
  (# M: ^Matrix;
  enter M[]
  ...
  #);
move:< (* Translate THIS(AbstractGraphicalObject) by offset *)
  (# offset: @Point;
  enter offset
  do ...; INNER;
  #);
moveTo:<
  (* Move THIS(AbstractGraphicalObject).theShape.hotSpot to pos *)
  (# pos: @Point;
  enter pos
  do ...; INNER;
  #);
scale:< (* Scale THIS(AbstractGraphicalObject) by factor *)
  (# factor: @Vector; (* Real point *)
  enter factor
  do ...; INNER;
  #);
rotate:<
  (* Rotate THIS(AbstractGraphicalObject) by angle (degrees) *)
  (# angle: @Real;
  enter angle
  do ...; INNER;
  #);

(* QUERY *)
containsPoint:< booleanValue
  (* Answer if thePoint is inside the shape of
   * THIS(AbstractGraphicalObject).  thePoint is assumed to be in
   * coordinates relative to theCanvas.
   *)
  (# theCanvas: ^BifrostCanvas;
     thePoint: @Point;
     doneInInner: @boolean;
  enter (theCanvas[], thePoint)
  ...
  #);

(* The aggregation parts *)
theShape: ^ShapeDesc;
```

```
    thePaint: ^Paint;

   (* PRIVATE *)
   writePS:<
      (# out: ^stream; invisible: @boolean
      enter out[]
      do ...
      #);
   private: @ ...;
   recalculateShape:< (* private *)
      (# theCanvas: ^BifrostCanvas enter theCanvas[] do INNER #);
do INNER;
exit THIS(AbstractGraphicalObject)[]
#);
```

GraphicalObject

```
GraphicalObject: AbstractGraphicalObject
  (#
     shapeDesc::< (* The real shape with lines and splines *)
       Shape;
     setShape: (* Set the Shape of THIS(GraphicalObject) *)
       (# enter theShape[] #);
     getShape::< (* Get the Shape of THIS(GraphicalObject) *)
       (# do theShape[] -> s[] #);
     copy::< (# ... #);
     draw::< (# ... #);
     writePS::<(# do ... #);
     hilite::< (# ... #);
     unHilite::< (# ... #);
     recalculateShape::< (* private *)
       (# ... #);
  do INNER
  #);
```

PictureShape

```
PictureShape: AbstractShape (* To be further specialized *)
  (# <<SLOT PictureShapeAttributes: attributes>>;

     firstpoint::< (# ... #);
     copy::< (# do INNER; ... #);
     getBounds::< (# do ... #);
     containsPoint::<(# ... #);
     getControls::<(# ... #);
     hiliteControls::<   (# ... #);
     hiliteOutline::< (# ... #);
     transform::<(# do ...; INNER #);

     (* Private *)
     writePS::<(# do ... #);
     pictureprivate: @...;
  do INNER
```

```
          #);
```

Picture

```
Picture: AbstractGraphicalObject
  (* A collection of graphical objects *)
  (# <<SLOT PictureAttributes: attributes >>;

     shapeDesc::< PictureShape;
     TMDesc::<(# CalcCanvasTM::<(# do ...; INNER #);
          enterTM::< (# do ...; INNER #);
       do INNER;
       #);
     init::< (# ... #);
     add:<
       (* Add go to THIS(Picture) *)
       (# go: ^AbstractGraphicalObject;
       enter go[]
       ...
       #);
     delete:<
       (* Delete go from THIS(Picture) *)
       (# go: ^AbstractGraphicalObject;
       enter go[]
       ...
       #);
     drawOnPixmap:  (* Not Yet Implemented *)
       (* Draw THIS(Picture) on pm *)
       (# pm: ^Pixmap;
       enter pm[]
       do ...;
       #);
     draw::< (# ... #);
     erase::< (# ... #);
     copy::< (# do INNER; ... #);
     setPaint::<
       (* Specify the paint to use for all AbstractGraphicalObjects
        * in THIS(Picture). If they are shown on the Canvas, their
        * visual appearance is changed instantly.
        *)
       (# theCanvas: ^BifrostCanvas;
       enter theCanvas[]
       ...
       #);
     getBounds::< (# ... #);
     hilite::< (# ... #);
     unHilite::< (# ... #);
     bringForward:
       (* Make aGO the last AbstractGraphicalObject of THIS(Picture)
        * aGO must already be a member of THIS(Picture)
        *)
       (# aGO: ^AbstractGraphicalObject;
       enter aGO[]
       ...
       #);
     sendBehind:
       (* Make aGO the first AbstractGraphicalObject of THIS(Picture)
        * aGO must already be a member of THIS(Picture)
        *)
```

```
      (# aGO: ^AbstractGraphicalObject;
       enter aGO[]
       ...
       #);
  scanGOs:
      (* Scan through each AbstractGraphicalObject in THIS(Picture)
       * in order from the bottommost to the frontmost one.
       *)
      (# go: ^AbstractGraphicalObject;
       ...
       #);
  scanGOsReverse:
      (* Scan through each AbstractGraphicalObject in THIS(Picture)
       * in order from the frontmost to the bottommost one.
       *)
      (# go: ^AbstractGraphicalObject;
       ...
       #);

  (* INTERACTION *)
  interactiveCreateShape :<(# ... #);
  interactiveCombineShape :<(# ... #);
  interactiveReshape :<(# ... #);

  (* QUERY *)
  lastGO:
      (* Exit reference to last AbstractGraphicalObject in
       * THIS(Picture)
       *)
      (# aGO: ^AbstractGraphicalObject;
       ...
       exit aGO[]
       #);
  firstGO:
      (* Exit reference to last AbstractGraphicalObject in
       * THIS(Picture)
       *)
      (# aGO: ^AbstractGraphicalObject;
       ...
       exit aGO[]
       #);
  noOfGOs: integerValue
      (* Exit number of AbstractGraphicalObjects in THIS(Picture) *)
      (# ... #);
  isEmpty: booleanValue
      (* True iff no graphical objects has been added to
       * THIS(Picture)
       *)
      (# ... #);
  isMember: booleanValue
      (* True iff aGO has been added to THIS(Picture) *)
      (# aGO: ^AbstractGraphicalObject;
       enter aGO[]
       ...
       #);
  containsPoint:<
      (* Answer if thePoint (canvascoordinates) is inside the shape
       * of any graphical object of THIS(Picture)
       *)
      (# ... #);
  firstContaining:<
      (* Returns reference to first AbstractGraphicalObject in
       * THIS(Abstract) that contains thePoint.
       * thePoint is assumed to be in coordinates relative to
       * theCanvas.
       *)
```

```
      (# theCanvas: ^BifrostCanvas;
         thePoint: @Point;
         aGO: ^AbstractGraphicalObject;
      enter (theCanvas[], thePoint)
      ...
      exit aGO[]
      #);
   lastContaining:<
      (* Returns reference to last AbstractGraphicalObject in
       * THIS(Picture) that contains thePoint.
       * thePoint is assumed to be in coordinates relative to
       * theCanvas.
       *)
      (# theCanvas: ^BifrostCanvas;
         thePoint: @Point;
         aGO: ^AbstractGraphicalObject;
      enter (theCanvas[], thePoint)
      ...
      exit aGO[]
      #);
   writePS::<(# do ... #);
 do INNER;
 #);  (* Picture *)
```

BifrostCanvas

```
(* The BifrostCanvas is the connection between the graphic
 * definitions and the device. Graphical objects become visible on
 * the output device when they are added to a BifrostCanvas by the
 * use of the draw-method.
 *)

BifrostCanvas: Canvas
  (# <<SLOT CanvasAttributes: attributes >>;

     thePicture:
       (* Picture holding the graphical objects *)
       ^Picture;
     visualShape:
       (* The part of THIS(BifrostCanvas) that is visible *)
       ^Shape;
     clipShape:
       (* Shape used for clipping in THIS(BifrostCanvas). Defaults to
        * visualShape
        *)
       ^Shape;
     draw:  (* Put GO on THIS(BifrostCanvas) *)
       (# GO: ^AbstractGraphicalObject
       enter GO[]
       ...
       #);
     erase: (* Erase GO from THIS(BifrostCanvas) *)
       (# aGO: ^AbstractGraphicalObject;
       enter aGO[]
       ...
       #);
     refresh: (* Mark entire BifrostCanvas as damaged and repair *)
       ...;
     scanThePicture:
```

```
  (* Scan through each AbstractGraphicalObject in thePicture in
   * order from the bottommost to the frontmost one.
   *)
  (# go: ^AbstractGraphicalObject;
  ...
  #);
scanThePictureReverse:
  (* Scan through each AbstractGraphicalObject in thePicture in
   * order from the frontmost to the bottommost one.
   *)
  (# go: ^AbstractGraphicalObject;
  ...
  #);
firstContaining:
  (* Returns reference to first AbstractGraphicalObject in
   * thePicture that contains thePoint.
   * thePoint is assumed to be in coordinates relative to
   * THIS(BifrostCanvas).
   *)
  (# thePoint: @Point;
  enter thePoint
  exit (THIS(BifrostCanvas)[],thePoint)
       ->thePicture.firstContaining
  #);
lastContaining:
  (* Returns reference to last AbstractGraphicalObject in
   * thePicture that contains thePoint.
   * thePoint is assumed to be in coordinates relative to
   * THIS(BifrostCanvas).
   *)
  (# thePoint: @Point;
  enter thePoint
  exit (THIS(BifrostCanvas)[],thePoint)
       ->thePicture.lastContaining
  #);

(* EVENT HANDLING *)
eventHandler::<
  (#
    onOpen:<
      (* Called immediately after the BifrostCanvas has been
       * made visible.
       *)
      (#
      ...
      #);
    onMouseDown::<
      (* Called when a mouse button is pressed *)
      (# mousePos: @Point
          (* the position of the mouse in device coordinates *);
        button: (# exit buttonState #);
        shiftModified: (# exit shiftKey #);
        (*lockModified: (# exit capsLock #);*)
        controlModified: (# exit controlKey #);
        metaModified: (# exit metaKey #);
        altModified: (# exit altKey #);
      ...
      #);
    onKeyDown::< (* Called when a key is pressed *)
      (# ... #);
    onRefresh::<
      (* Called when THIS(BifrostCanvas) is being refreshed *)
      (# do ... #);
    onFrameChanged::<
      (* Called when THIS(BifrostCanvas) changes its frame
       * (size).
```

```
          *)
        (# ... #);
    onActivate::<
        (* Called when the BifrostCanvas is activated, e.g. by
         * entering it with the mouse.
         *)
        (# ... #);
    onDeactivate::<
        (* Called when the BifrostCanvas is deactivated, e.g. by
         * leaving it with the mouse.
         *)
        (# ... #);
    onFatalError:< (# err: ^text; enter err[] do INNER #);
  #);
borderwidth: @
  (* The width of the border if present. Defaults to 0 *)
  (# value: @integer;
  enter (# enter value ... #)
  exit  (# ... exit value #)
  #);
borderpaint: @
  (* The Paint used to fill the border if present. Defaults to
   * black
   *)
  (# p: ^Paint;
  enter (# enter p[] ... #)
  exit  (# ... exit p[] #)
  #);
backgroundpaint: @
  (* The Paint used as background. Defaults to white *)
  (# p: ^Paint;
  enter (# enter p[] ... #)
  exit  (# ... exit p[] #)
  #);

size:
  (* The size of THIS(BifrostCanvas) *)
  (# w, h: @integer;
  enter (# enter (w, h) ... #)
  exit THIS(Canvas).size
  #);
open::<
  (* Open the BifrostCanvas, i.e. make it visible and start to
   * handle events.
   *)
  (# create::< (# ... #);
     defaultbackground: @boolean
        (* If defaultbackground is set to true,
         * THIS(BifrostCanvas) will appear with the same
         * background color as the surrounding window, otherwise
         * it will be set to white (unless otherwise specified by
         * backgroundpaint
         *);
  ...
  #);
close::<
  (* Close the BifrostCanvas, i.e. make it disappear and forget
   * all information stored in it.
   *)
  (# ... #);
writeEPS:<
  (* Write Encapsulated PostScript to the stream out *)
  (# out: ^Stream;
     pagesize: @rectangle;
     vertical: @boolean;
     noOfCopies: @integer;
```

```
    enter (pagesize, vertical, noOfCopies, out[])
    do ...
    #);
readEPS:<
    (* Reads an EPS file written with writeEPS from stream inFile *)
    (#
        inFile: ^Stream;
    enter inFile[]
    ...
    #);
setClip:
    (* Make clipShape the new clipping region in
     * THIS(BifrostCanvas)
     *)
    (#
    enter clipShape[]
    do ...;
    #);
getClip:
    (* Exit the clipping region of THIS(BifrostCanvas) *)
    (# exit clipShape[] #);
deviceToCanvas:
    (* Transform p1 from Device coordinates to BifrostCanvas
     * coordinates.
     *)
    (# p1,p2: @Point;
    enter p1
    ...
    exit p2
    #);
canvasToDevice:
    (* Transform p1 from BifrostCanvas coordinates to Device
     * coordinates.
     *)
    (# p1,p2: @Point;
    enter p1
    ...
    exit p2
    #);

canvasToDeviceRectangle:
    (* Transform r from BifrostCanvas coordinates to Device coordinates. *)
    (# r: @Rectangle;
    enter r
    do r -> TM.transformRectangle -> r;
    exit r
    #);

deviceToCanvasRectangle:
    (* Transform r from BifrostCanvas coordinates to Device coordinates. *)
    (# r: @Rectangle;
    enter r
    do r -> TM.inverseTransformRectangle -> r;
    exit r
    #);

(* DAMAGE / REPAIR *)
damaged:
    (* Inform THIS(BifrostCanvas) that r has been damaged, and
     * thus should be a part of the area redrawn upon the next
     * repair.
     *)
    (# r: @Rectangle;
    enter r
    do ...;
    #);
```

```
  repair:
    (* Redraw all damaged areas in THIS(BifrostCanvas) *)
    (# do ... #);


(* INTERACTION *)
  interactionHandler:
    (* Specialize THIS(BifrostCanvas).InteractionHandler to
     * perform an interaction.  Specialize the different virtuals
     * inside THIS(InteractionHandler) to perform actions in
     * response to various events.  Of course, using an
     * InteractionHandler only gives meaning if a pointing device
     * and/or a keyboard is connected to the actual device.
     *
     * NOTICE: At most one InteractionHandler may active at any
     * given time
     *)
    (# initialize:<
         (* Called before THIS(InteractionHandler) is started *)
         (# ... #);
       motion:<
         (* Called when the the pointing device has been moved *)
         object;
       idle:<
         (* Called repeatedly when no events are ready *)
         object;
       buttonPress:<
         (* Called when a button of the pointing device has been
          * pressed.
          *)
         (# button: @Integer enter button do INNER; #);
       buttonRelease:< object
         (* Called when a button of the pointing device has been
          * released
          *);
       keyPress:<
         (* Called when a key on the keyboard has been pressed *)
         (# ch: @Char; enter ch do INNER #);
       keyRelease:<
         (* Called when a key on the keyboard has been released *)
         (# ch: @Char; enter ch do INNER #);
       terminateCondition:< booleanObject
         (* Specifies under what condition to stop
          * THIS(InteractionHandler)
          *)
         (# ... #);
       terminated:<
         (* Called just before THIS(InteractionHandler) ends *)
         (# ... #);
       getPointerLocation: @
         (* Returns the current pointer location in device
          * coordinates
          *)
         (# thePoint: @Point;
         do ...;
         exit thePoint
         #);
       getGlobalPointerLocation: @
         (* Returns the current pointer location in screen
          * coordinates
          *)
         (# thePoint: @Point;
         do ...;
         exit thePoint
         #);
       isModifierOn: @booleanValue
         (* Tell if theModifier is currently being pressed *)
```

```
              (# theModifier: @Modifier;
                 enter theModifier
                 do ...;
                 #);
           doubleClick: @booleanValue
              (* Answer if the last button press on the pointing device
               * was a double click
               *)
              (# ... #);
        do ...;
        #);
  interactiveCreateShape:
     (* Tell GO to start an interaction for creation on
      * THIS(BifrostCanvas)
      *)
     (# GO: ^AbstractGraphicalObject;
        p: @Point (* start interaction at p *);
        theModifier: @Modifier;
     enter (GO[],p,theModifier)
     ...
     #);
  interactiveCombineShape:
     (* Tell GO to start an interaction for combination on
      * THIS(BifrostCanvas)
      *)
     (# GO: ^AbstractGraphicalObject;
        p: @Point (* start interaction at p *);
        theModifier: @Modifier;
     enter (GO[],p,theModifier)
     ...
     #);
  interactiveReshape:
     (* Tell GO to start an interaction for reshaping on
      * THIS(BifrostCanvas)
      *)
     (# GO: ^AbstractGraphicalObject;
        p: @Point (* start interaction at p *);
        theModifier: @Modifier;
     enter (GO[],p,theModifier)
     ...
     #);
  interactiveMove:
     (* Tell GO to start an interaction for motion on
      * THIS(BifrostCanvas).
      *)
     (# GO: ^AbstractGraphicalObject;
        p: @Point (* start interaction at p *);
        theModifier: @Modifier;
     enter (GO[],p,theModifier)
     ...
     #);
  interactiveRotate: (* Not Yet Implemented *)
     (* Tell pict to start an interaction for rotation on
      * THIS(BifrostCanvas)
      *)
     (# GO: ^AbstractGraphicalObject;
        p: @Point (* start interaction at p *);
        theModifier: @Modifier;
     enter (GO[],p,theModifier)
     ...
     #);
  interactiveScale: (* Not Yet Implemented *)
     (* Tell pict to start an interaction for scaling on
      * THIS(BifrostCanvas)
      *)
     (# GO: ^AbstractGraphicalObject;
```

```
      p: @Point (* start interaction at p *);
      theModifier: @Modifier;
    enter (GO[],p,theModifier)
    ...
    #);
bringForward:
  (* Bring aGO forward in THIS(BifrostCanvas).thePicture *)
  (# aGO: ^AbstractGraphicalObject;
  enter aGO[]
  ...
  #);
sendBehind:
  (* Send aGO behind in THIS(BifrostCanvas).thePicture *)
  (# aGO: ^AbstractGraphicalObject;
  enter aGO[]
  ...
  #);

hitControl:
  (* Answer whether p is within 2 mm of a control point of aGO
   * Exits exact point if hit, NONE otherwise
   *)
  (# aGO: ^AbstractGraphicalObject;
     p: @Point;
     res: ^Point;
  enter (aGO[],p)
  ...
  exit res[]
  #);
hilite:
  (* Tell GO to highlight itself on THIS(BifrostCanvas) *)
  (# GO: ^AbstractGraphicalObject
  enter GO[]
  ...
  #);
unHilite:
  (* Tell GO to unhighlight itself on THIS(BifrostCanvas) *)
  (#  GO: ^AbstractGraphicalObject
  enter GO[]
  ...
  #);

(* Primitives for immediate drawing (sometimes also known as
 * transient drawing).  For efficiency all of these use DEVICE
 * coordinates.  Nothing drawn by means of these primitives can
 * be repaired automatically by THIS(BifrostCanvas).  Uses an
 * arbitrary color, that is guarentied to be different to what
 * is underneath.  May be erased by repeating the draw-request,
 * and is thus very useful for feedback in interaction.
 *)

setImmediateLineWidth:
  (* Set the width used for immediate lines and arcs  *)
  (# lineWidth: @Integer;
  enter lineWidth
  ...
  #);
immediatespot:
  (* Draw a small filled rectangle around center *)
  (# center: @Point;
  enter (center)
  ...
  #);
immediateLine:
  (* Draw an immediate line from p1 to p2 *)
  (# p1,p2: @Point;
```

```
      enter (p1,p2)
      ...
      #);
immediateDot:
      (* Draw a dot of the size of one device-pixel at p *)
      (# p1: @Point;
      enter (p1)
      ...
      #);
immediateMultiLine:
      (* Draw an immediate multiline specified by the points in p.
       * If close is true, the multiline will be closed by a line
       * from the first point to the last point.
       *)
      (# p: ^PointArray;
         close: @Boolean;
      enter (p[], close)
      do ...;
      #);
immediateArc:
      (# cx, cy: @integer; (* Center coordinates *)
         hr, vr: @integer; (* Horizontal/vertical radius *)
         a1, a2: @integer; (* Defining angles in degrees *)
      enter (cx, cy, hr, vr, a1, a2)
      ...
      #);
immediaterect:
      (* Draw the outline of r *)
      (# r: @Rectangle;
      enter r
      ...
      #);
immediateText:
      (* Draw theString at pos, with appearance as specified with
       * theFontName, theStyle, theSize, and underline
       *)
      (# pos: @Point;
         theFontName: @FontName;
         theStyle: @Style;
         theSize: @integer;
         underline: @boolean;
         theString: ^text;
      enter (pos, theFontName, theStyle, theSize,
            underline, theString[])
      do ...
      #);

   (* Utility functions to convert between pixels and
    * milimeter.
    *)
MMToPixel: (* Exits p scaled from mm to pixels *)
      (# p: @Point;
      enter p
      do ...
      exit p
      #);
pixelToMM: (* Exits p scaled from pixels to mm *)
      (# p: @Point;
      enter p
      do ...
      exit p
      #);

   (* Utility to set zoom factor of THIS(BifrostCanvas).
    * The value 1.0 correponds to unzoomed state.
    *)
```

```
    ZoomFactor:
      (# set:
          (# z: @vector;
          enter z
          do ...
          #);
        get:
          (# z: @vector;
          ...
          exit z
          #)
      enter set
      exit get
      #);

    (* PRIVATE *)

    privatePart: @ ...;
    TM: ^Matrix
      (* Transformation from THIS(BifrostCanvas) to the actual
       * device
       *);
  #);
```

Bifrost

```
-- LIB: attributes --
Bifrost: Guienv(# do INNER #)
```

Bifrost Interface

EPSfile Interface

EPS file

```
ORIGIN 'Bifrost';
LIB_ITEM 'bifrost';
BODY 'private/PS/EPSread'
     'private/PS/EPSmacros';
--- streamLib: attributes ---
skipEPSheaders:
  (* This functions should be called to skip over the PostScript
   * headers and check that it is a valid Bifrost PostScript file.
   *
   * PostScript headers are generated by canvas.writeEPS or if
   * startEPSfile is used directly.
   *)
  (# formatException: exception
        (* called if THIS(stream) isn't a valid Bifrost PostScript
         * file.
         *)
        (# t: ^text
        enter t[]
        ...
        #);
     formatError:< formatException;
     inFile: ^stream;
  enter inFile[]
  ...
  #);

(* The following is indented to allow the user to write to
 * PostScript files, ie. writing several pictures (pages) to one
 * file.  Currently however, the user herself will have to generate
 * some PostScript code which separates the pages and makes it valid
 * postscript code.  This is likely to change in future releases.
 *)
startEPSfile:
  (* Writes PostScript headers to THIS(stream).  This function
   * should be called exactly once for each file before you do * anything else.
   *
   * This is called automatically by canvas.writeEPS file, but can
   * be used if the user don't want to use canvas.writeEPS.
   *)
  (# ... #);
WriteMacros: ...;
endEPSfile:
  (* Write PostScript epilogue to out. This function should be
   * called exactly once, before you close this file.
   *
   * As startEPSfile, this is automatically called by
   * canvas.writeEPS, and should normally only be used if the user
   * also used startEPSfile.
   *)
  (# ... #);
eoe: booleanValue
  (* Tests for end of EPS file as marked by endEPSfile *)
  (# ... #)
```

```
--- BifrostAttributes: attributes ---
loadPicture:
  (# <<SLOT bifrostLoadPicture: attributes>>;
     parseException: exception
       (* A parse exception is generated if any parse errors occurs
        * while reading the EPS file.

        * Notice that if for instance the file is truncated, the
        * normal stream exception are generated.
        *)
       (# t: ^text;
       enter t[]
       ...
       #);
     parseError:< parseException;
     createGO:<
       (* Should be specialized to create user defined patterns *)
       (# GO: ^AbstractGraphicalObject;
          patternName: ^text;
       enter patternName[]
       ...
       exit GO[]
       #);
     in: ^stream;
     out: ^Picture;
     private: @...
  enter in[]
  ...
  exit out[]
  #)
```

EPSfile Interface

ColorNames Interface

ColorNames

```
ORIGIN '~beta/basiclib/betaenv';
-- LIB: attributes --

(* Patterns used as enter parameters for SolidColor.name *)
```

aliceblue:	(# exit (61440, 63488, 65280) #);
antiquewhite:	(# exit (64000, 60160, 55040) #);
antiquewhite1:	(# exit (65280, 61184, 56064) #);
antiquewhite2:	(# exit (60928, 57088, 52224) #);
antiquewhite3:	(# exit (52480, 49152, 45056) #);
antiquewhite4:	(# exit (35584, 33536, 30720) #);
aquamarine:	(# exit (32512, 65280, 54272) #);
aquamarine1:	(# exit (32512, 65280, 54272) #);
aquamarine2:	(# exit (30208, 60928, 50688) #);
aquamarine3:	(# exit (26112, 52480, 43520) #);
aquamarine4:	(# exit (17664, 35584, 29696) #);
azure:	(# exit (61440, 65280, 65280) #);
azure1:	(# exit (61440, 65280, 65280) #);
azure2:	(# exit (57344, 60928, 60928) #);
azure3:	(# exit (49408, 52480, 52480) #);
azure4:	(# exit (33536, 35584, 35584) #);
beige:	(# exit (62720, 62720, 56320) #);
bisque:	(# exit (65280, 58368, 50176) #);
bisque1:	(# exit (65280, 58368, 50176) #);
bisque2:	(# exit (60928, 54528, 46848) #);
bisque3:	(# exit (52480, 46848, 40448) #);
bisque4:	(# exit (35584, 32000, 27392) #);
black:	(# exit (0, 0, 0) #);
blanchedalmond:	(# exit (65280, 60160, 52480) #);
blue:	(# exit (0, 0, 65280) #);
blue1:	(# exit (0, 0, 65280) #);
blue2:	(# exit (0, 0, 60928) #);
blue3:	(# exit (0, 0, 52480) #);
blue4:	(# exit (0, 0, 35584) #);
blueviolet:	(# exit (35328, 11008, 57856) #);
brown:	(# exit (42240, 10752, 10752) #);
brown1:	(# exit (65280, 16384, 16384) #);
brown2:	(# exit (60928, 15104, 15104) #);
brown3:	(# exit (52480, 13056, 13056) #);
brown4:	(# exit (35584, 8960, 8960) #);
burlywood:	(# exit (56832, 47104, 34560) #);
burlywood1:	(# exit (65280, 54016, 39680) #);
burlywood2:	(# exit (60928, 50432, 37120) #);
burlywood3:	(# exit (52480, 43520, 32000) #);
burlywood4:	(# exit (35584, 29440, 21760) #);
cadetblue:	(# exit (24320, 40448, 40960) #);
cadetblue1:	(# exit (38912, 62720, 65280) #);
cadetblue2:	(# exit (36352, 58624, 60928) #);
cadetblue3:	(# exit (31232, 50432, 52480) #);
cadetblue4:	(# exit (21248, 34304, 35584) #);
chartreuse:	(# exit (32512, 65280, 0) #);
chartreuse1:	(# exit (32512, 65280, 0) #);
chartreuse2:	(# exit (30208, 60928, 0) #);
chartreuse3:	(# exit (26112, 52480, 0) #);

```
chartreuse4:          (# exit (17664, 35584,     0) #);
chocolate:            (# exit (53760, 26880,  7680) #);
chocolate1:           (# exit (65280, 32512,  9216) #);
chocolate2:           (# exit (60928, 30208,  8448) #);
chocolate3:           (# exit (52480, 26112,  7424) #);
chocolate4:           (# exit (35584, 17664,  4864) #);
coral:                (# exit (65280, 32512, 20480) #);
coral1:               (# exit (65280, 29184, 22016) #);
coral2:               (# exit (60928, 27136, 20480) #);
coral3:               (# exit (52480, 23296, 17664) #);
coral4:               (# exit (35584, 15872, 12032) #);
cornflowerblue:       (# exit (25600, 38144, 60672) #);
cornsilk:             (# exit (65280, 63488, 56320) #);
cornsilk1:            (# exit (65280, 63488, 56320) #);
cornsilk2:            (# exit (60928, 59392, 52480) #);
cornsilk3:            (# exit (52480, 51200, 45312) #);
cornsilk4:            (# exit (35584, 34816, 30720) #);
cyan:                 (# exit (    0, 65280, 65280) #);
cyan1:                (# exit (    0, 65280, 65280) #);
cyan2:                (# exit (    0, 60928, 60928) #);
cyan3:                (# exit (    0, 52480, 52480) #);
cyan4:                (# exit (    0, 35584, 35584) #);
darkgoldenrod:        (# exit (47104, 34304,  2816) #);
darkgoldenrod1:       (# exit (65280, 47360,  3840) #);
darkgoldenrod2:       (# exit (60928, 44288,  3584) #);
darkgoldenrod3:       (# exit (52480, 38144,  3072) #);
darkgoldenrod4:       (# exit (35584, 25856,  2048) #);
darkgreen:            (# exit (    0, 25600,     0) #);
darkkhaki:            (# exit (48384, 46848, 27392) #);
darkolivegreen:       (# exit (21760, 27392, 12032) #);
darkolivegreen1:      (# exit (51712, 65280, 28672) #);
darkolivegreen2:      (# exit (48128, 60928, 26624) #);
darkolivegreen3:      (# exit (41472, 52480, 23040) #);
darkolivegreen4:      (# exit (28160, 35584, 15616) #);
darkorange:           (# exit (65280, 35840,     0) #);
darkorange1:          (# exit (65280, 32512,     0) #);
darkorange2:          (# exit (60928, 30208,     0) #);
darkorange3:          (# exit (52480, 26112,     0) #);
darkorange4:          (# exit (35584, 17664,     0) #);
darkorchid:           (# exit (39168, 12800, 52224) #);
darkorchid1:          (# exit (48896, 15872, 65280) #);
darkorchid2:          (# exit (45568, 14848, 60928) #);
darkorchid3:          (# exit (39424, 12800, 52480) #);
darkorchid4:          (# exit (26624,  8704, 35584) #);
darksalmon:           (# exit (59648, 38400, 31232) #);
darkseagreen:         (# exit (36608, 48128, 36608) #);
darkseagreen1:        (# exit (49408, 65280, 49408) #);
darkseagreen2:        (# exit (46080, 60928, 46080) #);
darkseagreen3:        (# exit (39680, 52480, 39680) #);
darkseagreen4:        (# exit (26880, 35584, 26880) #);
darkslateblue:        (# exit (18432, 15616, 35584) #);
darkslategray:        (# exit (12032, 20224, 20224) #);
darkslategray1:       (# exit (38656, 65280, 65280) #);
darkslategray2:       (# exit (36096, 60928, 60928) #);
darkslategray3:       (# exit (30976, 52480, 52480) #);
darkslategray4:       (# exit (20992, 35584, 35584) #);
darkturquoise:        (# exit (    0, 52736, 53504) #);
darkviolet:           (# exit (37888,     0, 54016) #);
deeppink:             (# exit (65280,  5120, 37632) #);
deeppink1:            (# exit (65280,  5120, 37632) #);
deeppink2:            (# exit (60928,  4608, 35072) #);
deeppink3:            (# exit (52480,  4096, 30208) #);
deeppink4:            (# exit (35584,  2560, 20480) #);
deepskyblue:          (# exit (    0, 48896, 65280) #);
deepskyblue1:         (# exit (    0, 48896, 65280) #);
deepskyblue2:         (# exit (    0, 45568, 60928) #);
```

```
deepskyblue3:           (# exit (    0, 39424, 52480) #);
deepskyblue4:           (# exit (    0, 26624, 35584) #);
dimgray:                (# exit (26880, 26880, 26880) #);
dodgerblue:             (# exit ( 7680, 36864, 65280) #);
dodgerblue1:            (# exit ( 7680, 36864, 65280) #);
dodgerblue2:            (# exit ( 7168, 34304, 60928) #);
dodgerblue3:            (# exit ( 6144, 29696, 52480) #);
dodgerblue4:            (# exit ( 4096, 19968, 35584) #);
firebrick:              (# exit (45568,  8704,  8704) #);
firebrick1:             (# exit (65280, 12288, 12288) #);
firebrick2:             (# exit (60928, 11264, 11264) #);
firebrick3:             (# exit (52480,  9728,  9728) #);
firebrick4:             (# exit (35584,  6656,  6656) #);
floralwhite:            (# exit (65280, 64000, 61440) #);
forestgreen:            (# exit ( 8704, 35584,  8704) #);
gainsboro:              (# exit (56320, 56320, 56320) #);
ghostwhite:             (# exit (63488, 63488, 65280) #);
gold:                   (# exit (65280, 55040,     0) #);
gold1:                  (# exit (65280, 55040,     0) #);
gold2:                  (# exit (60928, 51456,     0) #);
gold3:                  (# exit (52480, 44288,     0) #);
gold4:                  (# exit (35584, 29952,     0) #);
goldenrod:              (# exit (55808, 42240,  8192) #);
goldenrod1:             (# exit (65280, 49408,  9472) #);
goldenrod2:             (# exit (60928, 46080,  8704) #);
goldenrod3:             (# exit (52480, 39680,  7424) #);
goldenrod4:             (# exit (35584, 26880,  5120) #);
gray0:                  (# exit (    0,     0,     0) #);
gray:                   (# exit (49152, 49152, 49152) #);
gray1:                  (# exit (  768,   768,   768) #);
gray2:                  (# exit ( 1280,  1280,  1280) #);
gray3:                  (# exit ( 2048,  2048,  2048) #);
gray4:                  (# exit ( 2560,  2560,  2560) #);
gray5:                  (# exit ( 3328,  3328,  3328) #);
gray6:                  (# exit ( 3840,  3840,  3840) #);
gray7:                  (# exit ( 4608,  4608,  4608) #);
gray8:                  (# exit ( 5120,  5120,  5120) #);
gray9:                  (# exit ( 5888,  5888,  5888) #);
gray10:                 (# exit ( 6656,  6656,  6656) #);
gray11:                 (# exit ( 7168,  7168,  7168) #);
gray12:                 (# exit ( 7936,  7936,  7936) #);
gray13:                 (# exit ( 8448,  8448,  8448) #);
gray14:                 (# exit ( 9216,  9216,  9216) #);
gray15:                 (# exit ( 9728,  9728,  9728) #);
gray16:                 (# exit (10496, 10496, 10496) #);
gray17:                 (# exit (11008, 11008, 11008) #);
gray18:                 (# exit (11776, 11776, 11776) #);
gray19:                 (# exit (12288, 12288, 12288) #);
gray20:                 (# exit (13056, 13056, 13056) #);
gray21:                 (# exit (13824, 13824, 13824) #);
gray22:                 (# exit (14336, 14336, 14336) #);
gray23:                 (# exit (15104, 15104, 15104) #);
gray24:                 (# exit (15616, 15616, 15616) #);
gray25:                 (# exit (16384, 16384, 16384) #);
gray26:                 (# exit (16896, 16896, 16896) #);
gray27:                 (# exit (17664, 17664, 17664) #);
gray28:                 (# exit (18176, 18176, 18176) #);
gray29:                 (# exit (18944, 18944, 18944) #);
gray30:                 (# exit (19712, 19712, 19712) #);
gray31:                 (# exit (20224, 20224, 20224) #);
gray32:                 (# exit (20992, 20992, 20992) #);
gray33:                 (# exit (21504, 21504, 21504) #);
gray34:                 (# exit (22272, 22272, 22272) #);
gray35:                 (# exit (22784, 22784, 22784) #);
gray36:                 (# exit (23552, 23552, 23552) #);
gray37:                 (# exit (24064, 24064, 24064) #);
```

```
gray38:            (# exit (24832, 24832, 24832) #);
gray39:            (# exit (25344, 25344, 25344) #);
gray40:            (# exit (26112, 26112, 26112) #);
gray41:            (# exit (26880, 26880, 26880) #);
gray42:            (# exit (27392, 27392, 27392) #);
gray43:            (# exit (28160, 28160, 28160) #);
gray44:            (# exit (28672, 28672, 28672) #);
gray45:            (# exit (29440, 29440, 29440) #);
gray46:            (# exit (29952, 29952, 29952) #);
gray47:            (# exit (30720, 30720, 30720) #);
gray48:            (# exit (31232, 31232, 31232) #);
gray49:            (# exit (32000, 32000, 32000) #);
gray50:            (# exit (32512, 32512, 32512) #);
gray51:            (# exit (33280, 33280, 33280) #);
gray52:            (# exit (34048, 34048, 34048) #);
gray53:            (# exit (34560, 34560, 34560) #);
gray54:            (# exit (35328, 35328, 35328) #);
gray55:            (# exit (35840, 35840, 35840) #);
gray56:            (# exit (36608, 36608, 36608) #);
gray57:            (# exit (37120, 37120, 37120) #);
gray58:            (# exit (37888, 37888, 37888) #);
gray59:            (# exit (38400, 38400, 38400) #);
gray60:            (# exit (39168, 39168, 39168) #);
gray61:            (# exit (39936, 39936, 39936) #);
gray62:            (# exit (40448, 40448, 40448) #);
gray63:            (# exit (41216, 41216, 41216) #);
gray64:            (# exit (41728, 41728, 41728) #);
gray65:            (# exit (42496, 42496, 42496) #);
gray66:            (# exit (43008, 43008, 43008) #);
gray67:            (# exit (43776, 43776, 43776) #);
gray68:            (# exit (44288, 44288, 44288) #);
gray69:            (# exit (45056, 45056, 45056) #);
gray70:            (# exit (45824, 45824, 45824) #);
gray71:            (# exit (46336, 46336, 46336) #);
gray72:            (# exit (47104, 47104, 47104) #);
gray73:            (# exit (47616, 47616, 47616) #);
gray74:            (# exit (48384, 48384, 48384) #);
gray75:            (# exit (48896, 48896, 48896) #);
gray76:            (# exit (49664, 49664, 49664) #);
gray77:            (# exit (50176, 50176, 50176) #);
gray78:            (# exit (50944, 50944, 50944) #);
gray79:            (# exit (51456, 51456, 51456) #);
gray80:            (# exit (52224, 52224, 52224) #);
gray81:            (# exit (52992, 52992, 52992) #);
gray82:            (# exit (53504, 53504, 53504) #);
gray83:            (# exit (54272, 54272, 54272) #);
gray84:            (# exit (54784, 54784, 54784) #);
gray85:            (# exit (55552, 55552, 55552) #);
gray86:            (# exit (56064, 56064, 56064) #);
gray87:            (# exit (56832, 56832, 56832) #);
gray88:            (# exit (57344, 57344, 57344) #);
gray89:            (# exit (58112, 58112, 58112) #);
gray90:            (# exit (58624, 58624, 58624) #);
gray91:            (# exit (59392, 59392, 59392) #);
gray92:            (# exit (60160, 60160, 60160) #);
gray93:            (# exit (60672, 60672, 60672) #);
gray94:            (# exit (61440, 61440, 61440) #);
gray95:            (# exit (61952, 61952, 61952) #);
gray96:            (# exit (62720, 62720, 62720) #);
gray97:            (# exit (63232, 63232, 63232) #);
gray98:            (# exit (64000, 64000, 64000) #);
gray99:            (# exit (64512, 64512, 64512) #);
gray100:           (# exit (65280, 65280, 65280) #);
green:             (# exit (    0, 65280,     0) #);
green1:            (# exit (    0, 65280,     0) #);
green2:            (# exit (    0, 60928,     0) #);
```

```
green3:              (# exit (    0, 52480,     0) #);
green4:              (# exit (    0, 35584,     0) #);
greenyellow:         (# exit (44288, 65280, 12032) #);
honeydew:            (# exit (61440, 65280, 61440) #);
honeydew1:           (# exit (61440, 65280, 61440) #);
honeydew2:           (# exit (57344, 60928, 57344) #);
honeydew3:           (# exit (49408, 52480, 49408) #);
honeydew4:           (# exit (33536, 35584, 33536) #);
hotpink:             (# exit (65280, 26880, 46080) #);
hotpink1:            (# exit (65280, 28160, 46080) #);
hotpink2:            (# exit (60928, 27136, 42752) #);
hotpink3:            (# exit (52480, 24576, 36864) #);
hotpink4:            (# exit (35584, 14848, 25088) #);
indianred:           (# exit (52480, 23552, 23552) #);
indianred1:          (# exit (65280, 27136, 27136) #);
indianred2:          (# exit (60928, 25344, 25344) #);
indianred3:          (# exit (52480, 21760, 21760) #);
indianred4:          (# exit (35584, 14848, 14848) #);
ivory:               (# exit (65280, 65280, 61440) #);
ivory1:              (# exit (65280, 65280, 61440) #);
ivory2:              (# exit (60928, 60928, 57344) #);
ivory3:              (# exit (52480, 52480, 49408) #);
ivory4:              (# exit (35584, 35584, 33536) #);
khaki:               (# exit (61440, 58880, 35840) #);
khaki1:              (# exit (65280, 62976, 36608) #);
khaki2:              (# exit (60928, 58880, 34048) #);
khaki3:              (# exit (52480, 50688, 29440) #);
khaki4:              (# exit (35584, 34304, 19968) #);
lavender:            (# exit (58880, 58880, 64000) #);
lavenderblush:       (# exit (65280, 61440, 62720) #);
lavenderblush1:      (# exit (65280, 61440, 62720) #);
lavenderblush2:      (# exit (60928, 57344, 58624) #);
lavenderblush3:      (# exit (52480, 49408, 50432) #);
lavenderblush4:      (# exit (35584, 33536, 34304) #);
lawngreen:           (# exit (31744, 64512,     0) #);
lemonchiffon:        (# exit (65280, 64000, 52480) #);
lemonchiffon1:       (# exit (65280, 64000, 52480) #);
lemonchiffon2:       (# exit (60928, 59648, 48896) #);
lemonchiffon3:       (# exit (52480, 51456, 42240) #);
lemonchiffon4:       (# exit (35584, 35072, 28672) #);
lightblue:           (# exit (44288, 55296, 58880) #);
lightblue1:          (# exit (48896, 61184, 65280) #);
lightblue2:          (# exit (45568, 57088, 60928) #);
lightblue3:          (# exit (39424, 49152, 52480) #);
lightblue4:          (# exit (26624, 33536, 35584) #);
lightcoral:          (# exit (61440, 32768, 32768) #);
lightcyan:           (# exit (57344, 65280, 65280) #);
lightcyan1:          (# exit (57344, 65280, 65280) #);
lightcyan2:          (# exit (53504, 60928, 60928) #);
lightcyan3:          (# exit (46080, 52480, 52480) #);
lightcyan4:          (# exit (31232, 35584, 35584) #);
lightgoldenrod:      (# exit (60928, 56576, 33280) #);
lightgoldenrod1:     (# exit (65280, 60416, 35584) #);
lightgoldenrod2:     (# exit (60928, 56320, 33280) #);
lightgoldenrod3:     (# exit (52480, 48640, 28672) #);
lightgoldenrod4:     (# exit (35584, 33024, 19456) #);
lightgoldenrodyellow:(# exit (64000, 64000, 53760) #);
lightgray:           (# exit (54016, 54016, 54016) #);
lightpink:           (# exit (65280, 46592, 49408) #);
lightpink1:          (# exit (65280, 44544, 47360) #);
lightpink2:          (# exit (60928, 41472, 44288) #);
lightpink3:          (# exit (52480, 35840, 38144) #);
lightpink4:          (# exit (35584, 24320, 25856) #);
lightsalmon:         (# exit (65280, 40960, 31232) #);
lightsalmon1:        (# exit (65280, 40960, 31232) #);
lightsalmon2:        (# exit (60928, 38144, 29184) #);
```

```
lightsalmon3:          (# exit (52480, 33024, 25088) #);
lightsalmon4:          (# exit (35584, 22272, 16896) #);
lightseagreen:         (# exit ( 8192, 45568, 43520) #);
lightskyblue:          (# exit (34560, 52736, 64000) #);
lightskyblue1:         (# exit (45056, 57856, 65280) #);
lightskyblue2:         (# exit (41984, 54016, 60928) #);
lightskyblue3:         (# exit (36096, 46592, 52480) #);
lightskyblue4:         (# exit (24576, 31488, 35584) #);
lightslateblue:        (# exit (33792, 28672, 65280) #);
lightslategray:        (# exit (30464, 34816, 39168) #);
lightsteelblue:        (# exit (45056, 50176, 56832) #);
lightsteelblue1:       (# exit (51712, 57600, 65280) #);
lightsteelblue2:       (# exit (48128, 53760, 60928) #);
lightsteelblue3:       (# exit (41472, 46336, 52480) #);
lightsteelblue4:       (# exit (28160, 31488, 35584) #);
lightyellow:           (# exit (65280, 65280, 57344) #);
lightyellow1:          (# exit (65280, 65280, 57344) #);
lightyellow2:          (# exit (60928, 60928, 53504) #);
lightyellow3:          (# exit (52480, 52480, 46080) #);
lightyellow4:          (# exit (35584, 35584, 31232) #);
limegreen:             (# exit (12800, 52480, 12800) #);
linen:                 (# exit (64000, 61440, 58880) #);
magenta:               (# exit (65280,     0, 65280) #);
magenta1:              (# exit (65280,     0, 65280) #);
magenta2:              (# exit (60928,     0, 60928) #);
magenta3:              (# exit (52480,     0, 52480) #);
magenta4:              (# exit (35584,     0, 35584) #);
maroon:                (# exit (45056, 12288, 24576) #);
maroon1:               (# exit (65280, 13312, 45824) #);
maroon2:               (# exit (60928, 12288, 42752) #);
maroon3:               (# exit (52480, 10496, 36864) #);
maroon4:               (# exit (35584,  7168, 25088) #);
mediumaquamarine:      (# exit (26112, 52480, 43520) #);
mediumblue:            (# exit (    0,     0, 52480) #);
mediumorchid:          (# exit (47616, 21760, 54016) #);
mediumorchid1:         (# exit (57344, 26112, 65280) #);
mediumorchid2:         (# exit (53504, 24320, 60928) #);
mediumorchid3:         (# exit (46080, 20992, 52480) #);
mediumorchid4:         (# exit (31232, 14080, 35584) #);
mediumpurple:          (# exit (37632, 28672, 56064) #);
mediumpurple1:         (# exit (43776, 33280, 65280) #);
mediumpurple2:         (# exit (40704, 30976, 60928) #);
mediumpurple3:         (# exit (35072, 26624, 52480) #);
mediumpurple4:         (# exit (23808, 18176, 35584) #);
mediumseagreen:        (# exit (15360, 45824, 28928) #);
mediumslateblue:       (# exit (31488, 26624, 60928) #);
mediumspringgreen:     (# exit (    0, 64000, 39424) #);
mediumturquoise:       (# exit (18432, 53504, 52224) #);
mediumvioletred:       (# exit (50944,  5376, 34048) #);
midnightblue:          (# exit ( 6400,  6400, 28672) #);
mintcream:             (# exit (62720, 65280, 64000) #);
mistyrose:             (# exit (65280, 58368, 57600) #);
mistyrose1:            (# exit (65280, 58368, 57600) #);
mistyrose2:            (# exit (60928, 54528, 53760) #);
mistyrose3:            (# exit (52480, 46848, 46336) #);
mistyrose4:            (# exit (35584, 32000, 31488) #);
moccasin:              (# exit (65280, 58368, 46336) #);
navajowhite:           (# exit (65280, 56832, 44288) #);
navajowhite1:          (# exit (65280, 56832, 44288) #);
navajowhite2:          (# exit (60928, 52992, 41216) #);
navajowhite3:          (# exit (52480, 45824, 35584) #);
navajowhite4:          (# exit (35584, 30976, 24064) #);
navy:                  (# exit (    0,     0, 32768) #);
navyblue:              (# exit (    0,     0, 32768) #);
oldlace:               (# exit (64768, 62720, 58880) #);
olivedrab:             (# exit (27392, 36352,  8960) #);
```

```
olivedrab1:          (# exit (49152, 65280, 15872) #);
olivedrab2:          (# exit (45824, 60928, 14848) #);
olivedrab3:          (# exit (39424, 52480, 12800) #);
olivedrab4:          (# exit (26880, 35584,  8704) #);
orange:              (# exit (65280, 42240,     0) #);
orange1:             (# exit (65280, 42240,     0) #);
orange2:             (# exit (60928, 39424,     0) #);
orange3:             (# exit (52480, 34048,     0) #);
orange4:             (# exit (35584, 23040,     0) #);
orangered:           (# exit (65280, 17664,     0) #);
orangered1:          (# exit (65280, 17664,     0) #);
orangered2:          (# exit (60928, 16384,     0) #);
orangered3:          (# exit (52480, 14080,     0) #);
orangered4:          (# exit (35584,  9472,     0) #);
orchid:              (# exit (55808, 28672, 54784) #);
orchid1:             (# exit (65280, 33536, 64000) #);
orchid2:             (# exit (60928, 31232, 59648) #);
orchid3:             (# exit (52480, 26880, 51456) #);
orchid4:             (# exit (35584, 18176, 35072) #);
palegoldenrod:       (# exit (60928, 59392, 43520) #);
palegreen:           (# exit (38912, 64256, 38912) #);
palegreen1:          (# exit (39424, 65280, 39424) #);
palegreen2:          (# exit (36864, 60928, 36864) #);
palegreen3:          (# exit (31744, 52480, 31744) #);
palegreen4:          (# exit (21504, 35584, 21504) #);
paleturquoise:       (# exit (44800, 60928, 60928) #);
paleturquoise1:      (# exit (47872, 65280, 65280) #);
paleturquoise2:      (# exit (44544, 60928, 60928) #);
paleturquoise3:      (# exit (38400, 52480, 52480) #);
paleturquoise4:      (# exit (26112, 35584, 35584) #);
palevioletred:       (# exit (56064, 28672, 37632) #);
palevioletred1:      (# exit (65280, 33280, 43776) #);
palevioletred2:      (# exit (60928, 30976, 40704) #);
palevioletred3:      (# exit (52480, 26624, 35072) #);
palevioletred4:      (# exit (35584, 18176, 23808) #);
papayawhip:          (# exit (65280, 61184, 54528) #);
peachpuff:           (# exit (65280, 55808, 47360) #);
peachpuff1:          (# exit (65280, 55808, 47360) #);
peachpuff2:          (# exit (60928, 51968, 44288) #);
peachpuff3:          (# exit (52480, 44800, 38144) #);
peachpuff4:          (# exit (35584, 30464, 25856) #);
peru:                (# exit (52480, 34048, 16128) #);
pink:                (# exit (65280, 49152, 51968) #);
pink1:               (# exit (65280, 46336, 50432) #);
pink2:               (# exit (60928, 43264, 47104) #);
pink3:               (# exit (52480, 37120, 40448) #);
pink4:               (# exit (35584, 25344, 27648) #);
plum:                (# exit (56576, 40960, 56576) #);
plum1:               (# exit (65280, 47872, 65280) #);
plum2:               (# exit (60928, 44544, 60928) #);
plum3:               (# exit (52480, 38400, 52480) #);
plum4:               (# exit (35584, 26112, 35584) #);
powderblue:          (# exit (45056, 57344, 58880) #);
purple:              (# exit (40960,  8192, 61440) #);
purple1:             (# exit (39680, 12288, 65280) #);
purple2:             (# exit (37120, 11264, 60928) #);
purple3:             (# exit (32000,  9728, 52480) #);
purple4:             (# exit (21760,  6656, 35584) #);
red:                 (# exit (65280,     0,     0) #);
red1:                (# exit (65280,     0,     0) #);
red2:                (# exit (60928,     0,     0) #);
red3:                (# exit (52480,     0,     0) #);
red4:                (# exit (35584,     0,     0) #);
rosybrown:           (# exit (48128, 36608, 36608) #);
rosybrown1:          (# exit (65280, 49408, 49408) #);
rosybrown2:          (# exit (60928, 46080, 46080) #);
```

```
rosybrown3:          (# exit (52480, 39680, 39680) #);
rosybrown4:          (# exit (35584, 26880, 26880) #);
royalblue:           (# exit (16640, 26880, 57600) #);
royalblue1:          (# exit (18432, 30208, 65280) #);
royalblue2:          (# exit (17152, 28160, 60928) #);
royalblue3:          (# exit (14848, 24320, 52480) #);
royalblue4:          (# exit ( 9984, 16384, 35584) #);
saddlebrown:         (# exit (35584, 17664,  4864) #);
salmon:              (# exit (64000, 32768, 29184) #);
salmon1:             (# exit (65280, 35840, 26880) #);
salmon2:             (# exit (60928, 33280, 25088) #);
salmon3:             (# exit (52480, 28672, 21504) #);
salmon4:             (# exit (35584, 19456, 14592) #);
sandybrown:          (# exit (62464, 41984, 24576) #);
seagreen:            (# exit (11776, 35584, 22272) #);
seagreen1:           (# exit (21504, 65280, 40704) #);
seagreen2:           (# exit (19968, 60928, 37888) #);
seagreen3:           (# exit (17152, 52480, 32768) #);
seagreen4:           (# exit (11776, 35584, 22272) #);
seashell:            (# exit (65280, 62720, 60928) #);
seashell1:           (# exit (65280, 62720, 60928) #);
seashell2:           (# exit (60928, 58624, 56832) #);
seashell3:           (# exit (52480, 50432, 48896) #);
seashell4:           (# exit (35584, 34304, 33280) #);
sienna:              (# exit (40960, 20992, 11520) #);
sienna1:             (# exit (65280, 33280, 18176) #);
sienna2:             (# exit (60928, 30976, 16896) #);
sienna3:             (# exit (52480, 26624, 14592) #);
sienna4:             (# exit (35584, 18176,  9728) #);
skyblue:             (# exit (34560, 52736, 60160) #);
skyblue1:            (# exit (34560, 52736, 65280) #);
skyblue2:            (# exit (32256, 49152, 60928) #);
skyblue3:            (# exit (27648, 42496, 52480) #);
skyblue4:            (# exit (18944, 28672, 35584) #);
slateblue:           (# exit (27136, 23040, 52480) #);
slateblue1:          (# exit (33536, 28416, 65280) #);
slateblue2:          (# exit (31232, 26368, 60928) #);
slateblue3:          (# exit (26880, 22784, 52480) #);
slateblue4:          (# exit (18176, 15360, 35584) #);
slategray:           (# exit (28672, 32768, 36864) #);
slategray1:          (# exit (50688, 57856, 65280) #);
slategray2:          (# exit (47360, 54016, 60928) #);
slategray3:          (# exit (40704, 46592, 52480) #);
slategray4:          (# exit (27648, 31488, 35584) #);
snow:                (# exit (65280, 64000, 64000) #);
snow1:               (# exit (65280, 64000, 64000) #);
snow2:               (# exit (60928, 59648, 59648) #);
snow3:               (# exit (52480, 51456, 51456) #);
snow4:               (# exit (35584, 35072, 35072) #);
springgreen:         (# exit (    0, 65280, 32512) #);
springgreen1:        (# exit (    0, 65280, 32512) #);
springgreen2:        (# exit (    0, 60928, 30208) #);
springgreen3:        (# exit (    0, 52480, 26112) #);
springgreen4:        (# exit (    0, 35584, 17664) #);
steelblue:           (# exit (17920, 33280, 46080) #);
steelblue1:          (# exit (25344, 47104, 65280) #);
steelblue2:          (# exit (23552, 44032, 60928) #);
steelblue3:          (# exit (20224, 37888, 52480) #);
steelblue4:          (# exit (13824, 25600, 35584) #);
tan:                 (# exit (53760, 46080, 35840) #);
tan1:                (# exit (65280, 42240, 20224) #);
tan2:                (# exit (60928, 39424, 18688) #);
tan3:                (# exit (52480, 34048, 16128) #);
tan4:                (# exit (35584, 23040, 11008) #);
thistle:             (# exit (55296, 48896, 55296) #);
thistle1:            (# exit (65280, 57600, 65280) #);
```

```
thistle2:              (# exit (60928, 53760, 60928) #);
thistle3:              (# exit (52480, 46336, 52480) #);
thistle4:              (# exit (35584, 31488, 35584) #);
tomato:                (# exit (65280, 25344, 18176) #);
tomato1:               (# exit (65280, 25344, 18176) #);
tomato2:               (# exit (60928, 23552, 16896) #);
tomato3:               (# exit (52480, 20224, 14592) #);
tomato4:               (# exit (35584, 13824,  9728) #);
turquoise:             (# exit (16384, 57344, 53248) #);
turquoise1:            (# exit (    0, 62720, 65280) #);
turquoise2:            (# exit (    0, 58624, 60928) #);
turquoise3:            (# exit (    0, 50432, 52480) #);
turquoise4:            (# exit (    0, 34304, 35584) #);
violet:                (# exit (60928, 33280, 60928) #);
violetred:             (# exit (53248,  8192, 36864) #);
violetred1:            (# exit (65280, 15872, 38400) #);
violetred2:            (# exit (60928, 14848, 35840) #);
violetred3:            (# exit (52480, 12800, 30720) #);
violetred4:            (# exit (35584,  8704, 20992) #);
wheat:                 (# exit (62720, 56832, 45824) #);
wheat1:                (# exit (65280, 59136, 47616) #);
wheat2:                (# exit (60928, 55296, 44544) #);
wheat3:                (# exit (52480, 47616, 38400) #);
wheat4:                (# exit (35584, 32256, 26112) #);
white:                 (# exit (65535, 65535, 65535) #);
whitesmoke:            (# exit (62720, 62720, 62720) #);
yellow:                (# exit (65280, 65280,     0) #);
yellow1:               (# exit (65280, 65280,     0) #);
yellow2:               (# exit (60928, 60928,     0) #);
yellow3:               (# exit (52480, 52480,     0) #);
yellow4:               (# exit (35584, 35584,     0) #);
yellowgreen:           (# exit (39424, 52480, 12800) #)
```

ColorNames Interface

Palette Interface

Palette

```
ORIGIN 'Bifrost';
BODY 'private/Impl/PaletteImpl';
INCLUDE 'PredefinedGO' ;

-- BifrostAttributes: attributes --

Palette: BifrostCanvas
  (* A canvas showing an either vertical or horizontal sequence of
   * graphical objects.  At any time exactly one of these are
   * highlighted by a black frame.  This default highlight may be
   * changed by furtherbinding "hiliteitem".  Graphical objects are
   * added to THIS(Palette) by using "append".  The number of the
   * currently selected graphical object is in "selection".
   *)
  (# eventhandler::<
       (# onOpen::<(# ... #);
          onMouseDown::< (# ... #);
       #);

     blackpaint: @SolidColor; (* A black solid color *)

     selection: @integerValue
       (* Number of currently selected item *)
       (# checknew: @
            (# ns: @integer;
            enter ns ...
            #);
          enter checknew
       #);
     noOfItems: integerValue
       (* Exit the number of graphical objects in THIS(Palette) *)
       (# ... #);
     framePaint: @
       (* The paint used in the frame and default hilite *)
       (# f: ^Paint;
          newFramePaint: @
            (#
            enter f[]
            do ...
            #)
          enter newFramePaint
          exit f[]
       #);
     goPaint: @
       (* The paint used in the AbstractGraphicalObjects *)
       (# g: ^Paint;
          newGoPaint: @
            (# b: @boolean
            enter g[]
            do ...;
            #)
          enter newGoPaint
          exit g[]
       #);
```

```
    open::<
      (* Open THIS(Palette). Place it at position.  deltax,
       * deltay gives the step-lengths in horizontal and vertical
       * direction, respectively.  If vertical is true, it will be a
       * vertical palette, otherwise a horizontal one.
       *)
      (# deltax, deltay: @integer;
         vertical: @boolean;
      enter (deltax, deltay, vertical)
      ...
      #);
    size:
      (* Hides BifrostCanvas.size. The size of THIS(Palette) should
       * not be set directly, since it will adjust itself depending
       * on the number of items.
       *)
      (# s: @point
      exit (# ... exit s #)
      #);
    close::<(# ... #);
    append:<
      (* Append go as a selectable item in THIS(Palette).  go will
       * be centered in a box with dimensions deltax and deltay (as
       * specified to init)
       *)
      (# go: ^AbstractGraphicalObject;
      enter go[]
      do ...; INNER;
      #);
    hiliteitem:<
      (* Highlight item no i instead of the currently highlighted
       * item.  Does not change the current selection
       *)
      (# doneininner: @boolean;
         i: @integer;
      enter i
      do INNER; ...
      #);
    changed:<
      (* Called when the selection changes *)
      (# ... #);

    paletteprivate: @...;
#)
```

Palette Interface

PredefinedGO Interface

PredefinedGraphicalObject

```
ORIGIN 'Bifrost';
LIB_ITEM 'bifrost';
BODY 'private/Impl/PredefinedGoImpl'

-- BifrostAttributes: attributes --

PredefinedGraphicalObject: AbstractGraphicalObject
  (# shapeDesc::<
       PredefinedShape;
     TMDesc::< (# enterTM::< (# do ...; INNER #)#);
     init::< (# ... #);
     getShape::< (# do theShape.calculateShape -> s[] #);
  do INNER;
  #);
```

Line

```
Line: PredefinedGraphicalObject
  (# shapeDesc::< LineShape;

     begin: (# enter theShape.begin  exit theShape.begin #);
     end: (# enter theShape.end     exit theShape.end #);
     width: (# enter theShape.width  exit theShape.width #);
     dashes: (# enter theShape.dashes exit theShape.dashes #);
     cap: (# enter theShape.cap     exit theShape.cap  #);

     coordinates:
       (# enter theShape.coordinates exit theShape.coordinates #);
     draw::< (# ... #);
     copy::<(# do INNER; ... #);
  do INNER;
  #);
```

Multiline

```
MultiLine: PredefinedGraphicalObject
  (# shapeDesc::< MultiLineShape;

     width: (# enter theShape.width  exit theShape.width #);
     points: (# enter theShape.points exit theShape.points #);
     addPoint:
```

```
    (# p: @point
    enter p
    do p->(TM).inversetransformpoint->theShape.addPoint
    #);
  insertPoint:
    (# i: @integer; p: @point
    enter (p,i)
    do (p->(TM).inversetransformpoint,i)->theShape.insertPoint
    #);
  deletePoint:
    (# p: @point
    enter p
    do p->(TM).inversetransformpoint->theShape.deletePoint
    #);
  getPoint:
    (# i: @Integer;
       p: @Point;
    enter i
    do i->theShape.getPoint->p
    exit p
    #);
  setPoint:
    (# i: @Integer;
       p: @Point;
    enter (i,p)
    do (p->(TM).inversetransformpoint,i)->theShape.setPoint
    #);
  closestLineSegment:
    (# p: @point; i: @integer
    enter p
    do p->(TM).inversetransformpoint->theShape.closestLineSegment->i
    exit i
    #);
  dashes: (* Not Yet Implemented *)
    (# enter theShape.dashes exit theShape.dashes #);
  cap: (# enter theShape.cap    exit theShape.cap  #);
  join: (# enter theShape.join   exit theShape.join #);
  draw::< (# ... #);
  copy::< (# do INNER; ... #);
do INNER;
#);
```

GraphicText

```
GraphicalText: GraphicText(# #); (* Alias *)

GraphicText: PredefinedGraphicalObject
  (# shapeDesc::< TextShape;

     inittext: (# enter theShape.inittext #);

     position:
       (# enter theShape.position
       exit theShape.position
       #);
     thefontname:
       (#
       enter theShape.thefontname
       exit theShape.thefontname
```

```
      #);
    theStyle:
      (#
      enter theShape.theStyle
      exit theShape.theStyle
      #);
    size:
      (#
      enter theShape.size
      exit theShape.size
      #);
    underline:
      (#
      enter theShape.underline
      exit theShape.underline
      #);
    theText:
      (#
      enter theShape.theText
      exit theShape.theText
      #);

    draw::< (# ... #);
    copy::< (# ... #);
    interactiveCreateShape::<
      (# lastCh: @Char; (* Last character typed in interaction *)
      ...
      exit lastCh
      #);
    interactiveReshape::<
      (# lastCh: @Char; (* Last character typed in interaction *)
      ...
      exit lastCh
      #);
  do INNER;
  #);
```

Arc

```
Arc: PredefinedGraphicalObject
  (# shapeDesc::< ArcShape;

    center: (# enter theShape.center   exit theShape.center #);
    horizontalRadius:
      (#
      enter theShape.horizontalradius
      exit  theShape.horizontalradius
      #);
    verticalRadius:
      (#
      enter theShape.verticalradius
      exit  theShape.verticalradius
      #);
    angle1:
      (#
      enter theShape.angle1
      exit theShape.angle1
      #);
    angle2:
```

```
      (#
      enter theShape.angle2
      exit theShape.angle2
      #);
   arcWidth:
      (#
      enter theShape.arcWidth
      exit theShape.arcWidth
      #);

   draw::< (# ... #);
   copy::< (# do INNER; ... #);
  do INNER
  #);
```

PieSlice

```
PieSlice: PredefinedGraphicalObject
  (# shapeDesc::< PieShape;

     center:
       (#
       enter theShape.center
       exit theShape.center
       #);
     horizontalRadius:
       (#
       enter theShape.horizontalradius
       exit theShape.horizontalradius
       #);
     verticalRadius:
       (#
       enter theShape.verticalradius
       exit theShape.verticalradius
       #);
     angle1:
       (#
       enter theShape.angle1
       exit theShape.angle1
       #);
     angle2:
       (#
       enter theShape.angle2
       exit theShape.angle2
       #);
     draw::< (# ... #);
     copy::< (# do INNER; ... #);
  do INNER
  #);

StrokeAblePredefinedGraphicalObject: PredefinedGraphicalObject
  (#
     shapeDesc::< StrokeableShape
  do INNER
  #);
```

Rect

```
Rect: StrokeAblePredefinedGraphicalObject
  (# shapeDesc::< RectShape;

    upperleft:
      (#
      enter theShape.upperleft
      exit theShape.upperleft
      #);
    width:
      (#
      enter theShape.width
      exit theShape.width
      #);
    height:
      (#
      enter theShape.height
      exit theShape.height
      #);
    corners:
      (#
      enter theShape.corners
      exit theShape.corners
      #);

    draw::< (# ... #);
    copy::< (# do INNER; ... #);
  do INNER;
  #);
```

Ellipse

```
Ellipse: StrokeAblePredefinedGraphicalObject
  (# shapeDesc::< EllipseShape;

    center:
      (#
      enter theShape.center
      exit theShape.center
      #);
    horizontalradius:
      (#
      enter theShape.horizontalradius
      exit theShape.horizontalradius
      #);
    verticalradius:
      (#
      enter theShape.verticalradius
      exit theShape.verticalradius
      #);
    geometry:
      (#
      enter theShape.geometry
      exit theShape.geometry
      #);

    draw::< (# ... #);
```

```
   copy::< (# do INNER; ... #);
do INNER;
#)
```

RasterGrays Interface

RasterGrays

```
ORIGIN 'Bifrost';
BODY 'private/Impl/RasterGrayImpl';

-- BifrostAttributes: Attributes --

RasterGray: TiledSolidColor
  (* Abstract superpattern for the ten patterns below: Each of these
   * use one of the bitmaps in the fragment HalftonePatterns to make
   * an illusion of a shade of gray even on a black & white device.
   * See the pattern RasterGrays below for a convenient use.
   *)
  (# do init; (0, 0, 0) -> RGBvalues; #);

RasterGray0: RasterGray
  (# init:: (# ... #)#);
RasterGray11: RasterGray
  (# init:: (# ... #)#);
RasterGray22: RasterGray
  (# init:: (# ... #)#);
RasterGray33: RasterGray
  (# init:: (# ... #)#);
RasterGray44: RasterGray
  (# init:: (# ... #)#);
RasterGray56: RasterGray
  (# init:: (# ... #)#);
RasterGray67: RasterGray
  (# init:: (# ... #)#);
RasterGray78: RasterGray
  (# init:: (# ... #)#);
RasterGray89: RasterGray
  (# init:: (# ... #)#);
RasterGray100: RasterGray
  (# init:: (# ... #)#);

RasterGrays:
  (* A convenient alternative to using the above patterns directly is
   * using an instance of RasterGrays: RasterGrays enters a
   * percentage, and exits a reference to an initialized RasterGray
   * yielding a approximating shade of gray: percentage=0 yields
   * white, percentage=100 yields black, other percentages yields one
   * of eight intermediate shades of gray.
   *)
  (# private: @...;
     percentage: @integer;
     thegray: ^RasterGray;
     init:
       (# ... #)
  enter percentage
  do ...;
  exit thegray[]
  #)
```

RasterGrays Interface

SelectionPicture Interface

SelectionPicture

```
ORIGIN 'Bifrost';
BODY 'private/Impl/SelectionPictureImpl';

-- BifrostAttributes: attributes --

SelectionPicture: Picture
  (* A picture used to hilite a group of graphical objects.
   * SelectionPicture automatically highlights the graphical objects
   * added to it.
   *)
  (# thecanvas: ^Canvas
       (* The Canvas the GOs are shown in *);
     init::< (# enter theCanvas[] ... #);
     copy::<(# do ... #);
     draw::< (# ... #);
     erase::< (# ... #);
     add::< (# ... #);
     delete::< (# ... #);
     clear:
       (* Remove all graphical objects from THIS(SelectionPicture) *)
       (# ... #);
     onOneGO:< object
       (* Called when noOfGOs becomes 1 *);
     onTwoGOs:< object
       (* Called when noOfGOs changes from 1 to 2 *);
     onEmpty:< object
       (* Called when noOfGOs becomes 0 *);
  #)
```