

XYZ GeoBench Manual V4.00

Peter Schorn, schorn@inf.ethz.ch
Informatik, ETH, CH-8092 Zurich
last revision: 11/11/24

Abstract

The XYZ GeoBench is a software system for experimental geometric computation. The User's Manual explains the Macintosh application GeoBench, an interactive front end to the XYZ Library of geometric algorithms. The Programmer's Manual specifies the interfaces of the object oriented programming environment supplied.

Contents

0 The XYZ GeoBench Software

1 User's Manual

- 1.1 General
- 1.2 File Menu
- 1.3 Edit Menu
- 1.4 Objects Menu
- 1.5 Arithmetic Menu
- 1.6 Animation Menu
- 1.7 Windows Menu
- 1.8 Operations Menu

2 Programmer's Manual

- 2.1 Organization
- 2.2 The class hierarchy
- 2.3 The common behavior of all objects: the root class 'obj'
- 2.4 Sequences, lists and dynamic arrays: the classes 'sequence', 'list', 'vector' and 'homogenVector'
- 2.5 Arithmetic independent geometric primitives: the basic geometric classes 'point2d', 'lineSegment', 'rectangle' and 'circle'
- 2.6 Common universal abstract data types: the classes 'dictionary' and 'priorityQueue'
- 2.7 Graphs: the classes 'graphEdge', 'simpleUndirectedGraph', 'undirectedGraph', 'directedGraph' and 'spanningTree'
- 2.8 Support for animation, user interaction and error checking
- 2.9 Implemented geometric algorithms

Appendix A: Syntax of the textual I/O format

Appendix B: Changes to TransSkel

0 The XYZ GeoBench Software

The XYZ GeoBench software consists of two parts: 1) the Macintosh application ‘GeoBench’ and 2) the THINK Pascal sources together with the necessary project files.

The GeoBench is a Macintosh stand-alone application and is used for demonstrating the geometric algorithms of the XYZ Library and experimenting with them. It requires a Mac II with at least 2 MB of memory and, preferably, a two-page monitor. We describe the ‘GeoBench’ user interface, which adheres closely to Macintosh standards, in the User’s Manual.

The ‘GeoBench’ and the XYZ Library are written in THINK Pascal 3.0, an object oriented extension of Pascal. Currently the system consists of 78 modules with about 1000 kB of source code. We describe the programming interfaces necessary for extending the GeoBench in the Programmer’s Manual. We assume a basic understanding of the concepts *object*, *class*, *abstract class*, *method*, *inheritance* and *overriding of methods*.

An overview of the XYZ project is given in [NB 91] while [S 91b] emphasizes the programming environment aspect of the GeoBench. A detailed description of the design and implementation of the GeoBench is given in [S 91a] together with mathematical methods for constructing provably robust geometric programs. The diploma thesis [Brü 91] describes the realization of layered objects in the GeoBench.

Acknowledgements

I thank the following people: Jürg Nievergelt leads the XYZ project. Christoph Ammann, Michele De Lorenzi and Adrian Brünger extended the GeoBench to include layered objects. Christoph Ammann implemented algorithms for splinegons, wrote the code for boolean operations on polygons and created color maps from geographic data. Michele De Lorenzi wrote the networking code. Christoph Ammann and Markus Furter improved the user interface considerably. Beat Fawer implemented two algorithms for computing the Voronoi diagram. Martin Müller wrote code for the Traveling Salesman Problem. Peter Lippuner and Peter Skrotzky wrote an earlier version of this manual and contributed a library routine for computing the contour of a set of rectangles. Björn Beeli adapted a grid file package written by Hans Hinterberger and Lise Pfau.

1.1 General

The standard menus ‘File’ and ‘Edit’ have their usual semantics. Clicking (i.e. pointing or selecting) and dragging with the mouse are supported where appropriate. The following discusses the menus available.

1.2 File Menu

New	⌘ N	Creates an empty geometry window.
Open...	⌘ O	Opens an existing geometry document containing geometric objects and displays it in a geometry window.
Close	⌘ W	Closes a geometry window. Holding down the shift key suppresses the saving dialog box.
Save	⌘ S	Saves a geometry window on the disk.
Save As...		Saves a geometry window on the disk under a new name.
Write Text...		Saves the contents of a geometry window in textual format. The grammar is given in Appendix A of the Programmer’s Manual.
Read Text...		Reads geometric objects specified in a textual format.
Quit	⌘ Q	Terminates the program. Holding down the shift key suppresses the saving dialog box.

1.3 Edit Menu

Undo	⌘ Z	Undoes the effect of the most recent operation. This works only for dangerous or destructive operations.
Cut	⌘ X	Removes the selected objects from the geometry window and puts them into the clipboard. Other applications like MacDraw can access the clipboard.
Copy	⌘ C	Copies the selected objects into the clipboard. Other applications like MacDraw can access the clipboard.
Paste	⌘ V	Pastes the geometric objects

from the clipboard into a geometry window. Objects created with MacDraw can also be pasted.

Cut Last

Like ‘Cut’ but only the object which was selected most recently is moved to the clipboard.

Copy Last

Like ‘Copy’ but only the object which was selected most recently is copied to the clipboard.

Clear All

Deletes all geometric objects in a geometry window.

Select All

⌘ A Selects all geometric objects in a geometry window.

Redraw

Draws all objects again.

Zoom In...

⌘ E Enlarges a part of the window. After choosing this command the desired rectangular part of the window is selected by dragging with the mouse.

Zoom Out

⌘ F Undoes the most recent ‘Zoom In’ operation.

Transformations

Translate... Moves the selected objects. In the dialog box the quantities Δx and Δy determine the translation vector. Note that the origin is usually in the top left corner of the geometry window. Objects can also be translated by dragging. Note however that dragging near a vertex (e.g. of a polygon) results in moving just that vertex and not the whole object. If a whole polygon is to be translated by dragging, one should grab it at an edge.

Rotate... Rotates the selected objects around a point. First the point is selected and then the objects to be rotated (while holding the Shift key down). The angle is specified using the dialog box.

Scale... The selected objects are scaled by a given factor using a user specified origin.

Reflect... Reflects the selected objects with respect to a user specified line.

3d Projection...

Opens a dialog box that lets the user specify the eye point and the projection point for the view of layer objects (see [Brü 91] for further details). In addition, the dialog box contains a check box that determines whether hidden line elimination should be performed.

3d Transformation...

Opens a dialog box that lets the user apply transformations

such as translation, rotation or scaling to the view of layer objects (see [Brü 91] for further details). For convenience this dialog box contains a check box for hidden line elimination.

1.4 Objects Menu

Select	Lets the user select objects as opposed to enter objects.
Point	Lets the user enter points by clicking with the mouse.
Line Segment	Click at the start point and drag the segment.
Poly Line	Click at the start point, drag additional segments and double click when done.
Polygon	Like ‘Poly Line’.
Convex Polygon	Like ‘Polygon’. The convex hull of the polygon entered is used if the polygon is not convex.
Rectangle	Click at a corner and drag the rectangle.
Circle & N-Gon	Click at the center and drag the circle or the regular n -gon. The number of vertices is determined by the setting of ‘N-Gon vertices...’ (see this menu below).
D Point	Create a d -dimensional point by clicking at a location. This sets the x - and the y -coordinate while the other $d-2$ coordinates (if present, see the ‘Dimension’ command) are set to zero.
Dimension...	⌘ D Specifies the dimension of subsequently created d -dimensional points.
N-Gon Vertices...	Specifies the number of vertices of subsequently created n -gons. If this number is zero (i.e. no vertices), circles are created.
Object Info...	⌘ I Allows the user to interactively edit a textual description of the selected objects.
Group	⌘ G Groups the selected objects together in an object of type ‘vector’.
Ungroup	⌘ H If the selected object is a ‘vector’ it is ungrouped. Only one level of ungrouping is performed.
Random...	⌘ R Generates a number of randomly located objects of the same type. The type is specified either using the ‘Objects’ menu or by clicking at an

icon in the palette at the left border of each geometry window.

Randomize Sets the initial value of the random number generator to some arbitrary value.

Reset Random Resets the initial value of the random number generator to the value present at program start time.

1.5 Arithmetic Menu

Real Sets the type of the coordinates of subsequently created objects to 'Real'.

Float Sets the type of the coordinates of subsequently created objects to 'Float', a floating point number system realized in software. The base and precision (number of digits in the mantissa) can be set using the two following commands.

Base... Sets the base of the 'Float' number system.

Precision... Sets the number of digits in the mantissa.

Longint Sets the type of the coordinates of subsequently created objects to 'Longint'.

Objects of the same type (e.g. point) that were created with different settings of the arithmetic menu are treated as belonging to different types. Changing the setting of the arithmetic menu affects only subsequently created objects.

1.6 Animation Menu

Algorithm animation can be selectively turned on and off. One item activates the animation of all algorithms solving the same problem (e.g. turning on the animation for 'Closest Pair' animates all algorithms solving the closest pair problem).

1.7 Windows Menu

Hide Hides the window that lies on top. By selecting its name in the 'Windows' menu it becomes visible again.

Info The 'Info' window becomes the topmost visible window.

Geo-## The specified geometry window becomes the topmost visible window.

1.8 Operations Menu

The ‘Operations’ menu contains all the operations that are applicable to the selected objects. Choosing an operation from this menu performs the operation and creates a new geometry window containing the result of the operation. For a list of implemented algorithms see section 2.9 of the Programmer’s Manual.

2 Programmer's Manual

2.1 Organization

This manual is organized as follows. Section 2.2 describes the class hierarchy. Section 2.3 describes methods applicable to all objects in the system. Section 2.4 explains the basic ways of building collections of objects. Section 2.5 introduces the geometric primitives. Section 2.6 explains the most common abstract data types, dictionary and priority queue. Section 2.7 describes the graph oriented part of the GeoBench. Section 2.8 explains animation and how to interactively obtain parameters. Section 2.9 is a reference section on the implemented algorithms. Appendix A specifies the grammar of the textual I/O format. Appendix B describes the changes that we did in TransSkel, a public domain application module written by Paul DuBois [DB 89].

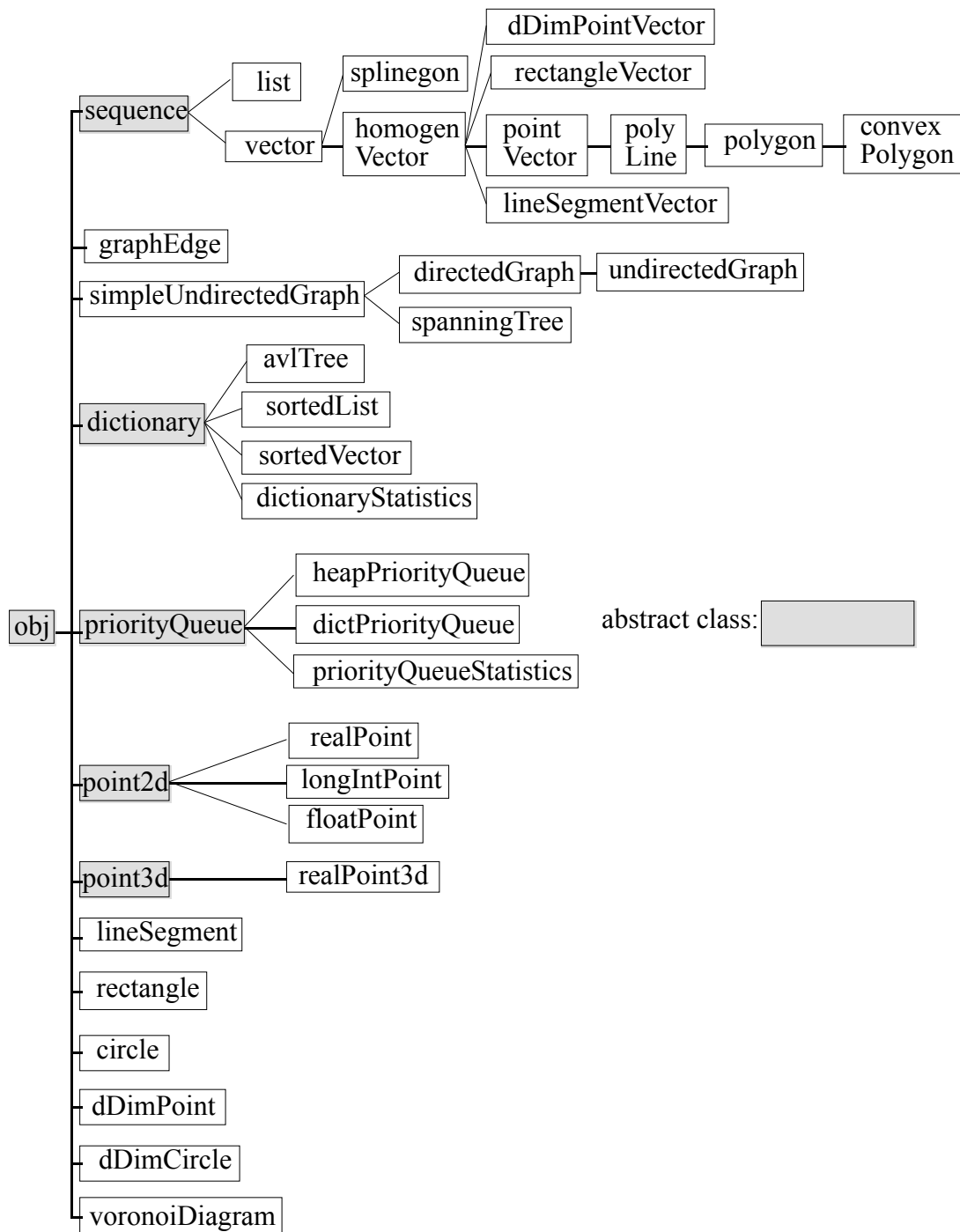
2.2 The class hierarchy

The class hierarchy is the glue holding the system together. All classes are descendants of a single abstract root class 'obj' which provides a set of universal operations applicable to all objects. The classes 'sequence', 'point2d', 'dictionary' and 'priorityQueue' are also abstract classes which should be used to postpone implementation decisions.

A geometric algorithm operating on an object of type T1 and producing an object of type T2 is a function method of class T1 producing an object of type T2.

Implementing a new geometric algorithm consists of

- 1) finding the appropriate place for the new method in the class hierarchy depending on the input data. Assume we implement a method for class T.
- 2) implementing the algorithm (with animation, if possible) using the methods from sections 2.3 – 2.9.
- 3) changing the methods 'describe' and 'execute' (see section 2.3) of class T such that the interactive front end of the XYZ GeoBench 'knows' about the newly implemented algorithm.



2.3 The common behavior of all objects: the root class ‘obj’

This section describes the interface common to all objects. When implementing a new descendant of ‘obj’, the methods described in this section must be implemented appropriately, taking the default implementation into account.

type

```

objectType = (intType, stringsType, point2dType,
lineSegmentType, rectangleType, listType, vectorType,
homogenVectorType, pointVectorType, convexPolygonType,
voronoiEdgeType, directedGraphType, undirectedGraphType,

```

```

graphEdgeType, voronoiDiagramType, polygon2dType,
markedRealPointType, simpleUndirectedGraphType,
spanningTreeType, rectangleVectorType, dDimPointType,
dDimCircleType, dDimPointVectorType, realPointType,
floatPointType, longIntPointType, sequenceType,
avlTreeType, sortedListType, sortedVectorType,
lineSegmentVectorType, dictPriorityQueueType,
heapPriorityQueueType, circleType, splinegonType,
spline2Type, straightEdgeType, polyLineType, layerType,
polyLayerType, layerVectorType, polyLayerVectorType,
QDPictureType, ColorQDPictureType, objType);

```

```

relationObj = (lessThanObj, equalObj, greaterThanObj);

```

obj = object

(* Memory management *)

```

procedure init;

```

(* Initializes the internal state of an object, i.e. all instance variables are set to a defined value. This method should be called after creating an object with 'new'. An object might *depend* on other objects, either directly like on an instance variable or indirectly like in a linked list. Dependent objects are initialized iff they belong to the internal state which is stated explicitly. 'init' may use global variables defined in 'obj' to determine types or other values. The appropriate variables are specified in the respective class interface. After 'init' has been called the object is ready to use. The default implementation does nothing. *)

```

procedure free;

```

(* Reclaims the memory belonging to the object's internal state (the reverse operation to 'init'). Dependent objects not belonging to the internal state are not destroyed. The object is removed; no dispose is allowed. The default implementation calls 'ShallowFree' which works correctly for all objects without dependent objects. *)

```

procedure freeAll;

```

(* Like 'free', but all dependent objects are also reclaimed recursively. The default implementation calls 'free' which is correct for objects that do not have dependent objects. *)

```

function duplicate: obj;

```

(* Produces a copy of the object's internal state. The default implementation calls 'ShallowDuplicate' which works correctly for all objects that do not have dependent objects. *)

```

function duplicateAll: obj;

```

(* Like 'duplicate', but all dependent objects are copied recursively. The default implementation calls 'duplicate' which is correct for objects without dependent objects. *)

procedure ShallowFree;

an object, should not be overridden. *)

(* Low level method to free

function ShallowDuplicate: obj;

duplicate an object, should not be overridden. *)

(* Low level method to

(* Interactive I/O *)

procedure interactiveOutput;

(* Draws the object.

Because of the XOR mode, an object can be removed from the window by a second call to 'interactiveOutput' which is useful in algorithm animation. The world coordinate system is used and the standard implementation raises an error condition.

Warning:

'interactiveOutput' is also used to draw into the external scrapbook, e.g., if the user selected 'Copy' from the 'Edit' menu and leaves the application. Do *not* assume that you are necessarily drawing into a window when your 'interactiveOutput' method is called. However, you may use 'thePort^.portRect' to determine the outline of the 'window' you are drawing in. If you really need to know whether you are drawing or just writing to the scrapbook, use 'thePort^.pnVis' ('true' means you are drawing in a window). See the implementation of 'voronoiDiagram.interactiveOutput' as an example. *)

procedure interactiveHighlight;

(* Draws the object in the

current window in a highlighted fashion. The world coordinate system is used. The standard implementation enlarges the pen size and calls 'interactiveOutput'. *)

procedure interactiveFlash;

(* Let the object flash in the

current window. The object is not drawn. The world coordinate system is used and the default implementation is implemented completely in terms of 'interactiveHighlight'. *)

procedure interactiveInput (px, py: integer);

(* Like 'windowInput' but

the world coordinate system is used. The default implementation is based completely on 'windowInput' and 'windowToWorld'. *)

procedure windowOutput;

(* Draws the internal state

on the current window. The window coordinate system is used and the default implementation is based completely on 'interactiveOutput' although this is inefficient in most cases. *)

procedure windowInput (px, py: integer);

(* Like 'init', but the

internal state is read using the mouse. It gives visual feedback but leaves the drawing window unchanged after completion (XOR graphics). 'px' and 'py' are the coordinates of the first mouse click. The window coordinate system is used. The default implementation of 'windowInput' is based completely on 'construct' and 'windowOutput'. *)

procedure construct (px, py, newX, newY: integer);

(* This method changes an

existing object on which 'init' has been called according to the four parameters formed by the first click location ('px', 'py') and the current click location ('newX',

newY). The window coordinate system is used. This method is only called by the default implementation of *windowInput* and need not be implemented if *windowInput* is overridden. The default implementation raises an error condition. *)

(* Binary and text file I/O *)

procedure fileOutput;

(* The whole object is

written to *currentStream* but no type identifier is written. The default
implementation raises an error condition. *)

procedure fileOutputTag;

(* Like 'fileOutput', but a type identifier is written first. The function 'fileGetType' in module 'obj' can be used to retrieve the type. The default implementation is based completely on 'getType' and 'fileOutput'. *)

procedure fileInput;

(* The complete object is read from '*currentStream*', the first byte on '*currentStream*' is the first byte of the internal state. The default implementation raises an error condition. *)

procedure textFileOutput;

(* Writes the internal state in textual form to '*currentStream*'. The first token written is always the type of the object. The grammar is given in appendix A. The default implementation raises an error condition. *)

procedure textFileInput;

(* Reads the internal state in textual form from '*currentStream*'. The first token is always the type of the object. The grammar is given in appendix A. The default implementation raises an error condition. *)

- (* **Description / execution.** Whenever objects are selected by the user, the interactive front end of the XYZ GeoBench calls for each class in the system the respective 'describe' method, thereby asking each class whether an operation of this class can be executed on the selected objects. GeoBench determines this way which operations are appropriate for which objects. Whenever a new method should be made available to the end user the pair 'describe' / 'execute' must be changed to reflect the new algorithm. *)

procedure describe;

(* Computes the (possibly hierarchical) menu entries of operations applicable to the list of objects in the global variable '*currentArguments*'. The procedures 'menuEntry' (plus 'beginSubMenu' and 'endSubMenu' for hierarchical menus) from the module 'sequence' are to be used. The default implementation does nothing. *)

procedure execute (item: integer);

(* Executes the selected method '*item*' with arguments in '*currentArguments*' and appends the result to the global variable '*currentResult*'. The default implementation raises an error condition. *)

(* **Geometric transformations** *)

procedure translate (dx, dy: extended);

(* Translates an object by ('dx','dy'). The default implementation raises an error condition. *)

procedure rotate (xOrigin, yOrigin, alpha: extended);

(* Rotates the object by

alpha (radians) around (*xOrigin*, *yOrigin*). The default implementation raises an error condition. *)

procedure scale (lambda: extended);

(* Scales the object by
lambda. The default implementation raises an error
condition. *)

procedure reflect (a, b, c: extended);

(* Mirrors the object at the

line $a \cdot x + b \cdot y = c$, $a^2 + b^2 \neq 0$. The default implementation raises an error condition.
*)

(* **Other transformations** *)

procedure randomChange;

(* Changes the internal state

randomly constrained by *currentRandomConstraint*. This global variable is usually interpreted as a rectangle describing the boundaries in which the random change takes place. The default implementation raises an error condition. *)

procedure windowToWorld;

(* Transforms the object to

world coordinates. The default implementation raises an error condition. *)

(* **Type computation and comparison** *)

function getType: objectType;

(* Get 'self's type. The

default implementation raises an error condition and returns 'objType'. *)

function getTextType: str255;

(* Computes a textual

description of 'self's type. The default implementation raises an error condition and returns the string 'Obj'. *)

function memberType (o: obj): boolean;

(* Tests whether 'o' is a

descendant of 'self'. The default implementation raises an error condition and returns 'true'. *)

function convertToType (t: pointTypeRange): obj;

(* Produces a copy of 'self'

based on type 't'. The default implementation raises an error condition and returns the result of 'duplicateAll'. *)

function equal (o: obj): boolean;

(* Determines whether

'self' is identical to 'o'. The default implementation raises an error condition and returns 'false'. *)

function genericLessThan (o: obj): boolean;

(* Determines whether 'self' is less than 'o' where 'o.getType = self.getType'. This function imposes a canonical order on the objects of a type and is used to eliminate duplicates by sorting. The default implementation raises an error condition and returns 'true' \Leftrightarrow 'o' and 'self' are different. *)

(* Selecting and dragging *)

function isSelected (where: obj): boolean;

(* Determines whether the object is selected by the given rectangle 'where'. The default implementation raises an error condition and returns 'false'. *)

function dragVertex (where: obj): boolean;

(* Tests whether the user attempted to drag a point (vertex) inside the rectangle 'where'. If a vertex is to be dragged, the function does all the dragging until the mouse button is released, updates the object and returns 'true'. The default implementation returns 'false'. *)

(* Miscellaneous *)

procedure textOutput;

(* Outputs the object in textual form in the THINK Pascal text window. Used for debugging. The default implementation raises an error condition. *)

procedure centerOfGravity (var cx, cy: extended);

(* Returns a point ('cx', 'cy') close to the object. The center is used for drawing graphs. The default implementation raises an error condition and returns the point (0, 0). *)

function pointCover (n: longInt): obj;

(* Covers the object with points, 'n' point for each atomic part. The result is a 'pointVector'. The default implementation raises an error condition and returns 'nil'. *)

end; (* obj *)

2.4 Sequences, lists and dynamic arrays:

The classes 'sequence', 'list', 'vector' and 'homogenVector'

This section describes the basic classes for building collections of objects. The abstract class 'sequence' factors out the common behavior of lists and arrays. The class 'list' realizes single linked lists whereas the class 'vector' realizes dynamic arrays. All elements in a 'homogenVector' have the same type.

type

```
seqIndex  = 1..maxN;
seqIndex0 = 0..maxN;
```


sequence = object (obj)

length: seqIndex0;

the sequence. *)

(* Number of elements in

procedure forAll (**procedure** whatToDo (x: obj));

‘whatToDo’ on all objects. *)

(* Performs the procedure

procedure append (x: obj);

the sequence. *)

(* Appends ‘x’ at the end of

procedure removeLast;

from the sequence without destroying it. *)

(* Removes the last element

procedure appendNonNil (x: obj);

*)

(* Appends ‘x’ if ‘x’ ≠ ‘nil’.

function sameType: boolean;

elements of the sequence are of the same type. *)

(* Tests whether all

function ith (i: seqIndex): obj;

of the sequence. Although this method is also available for vectors we recommend for efficiency reasons the use of ‘elements^[i]’ instead of the more expensive method call. *)

(* Produces the *i*th element

end; (* sequence *)

objArrayH = ^^ **array** [seqIndex] **of** obj;

(*

Handles (i.e. pointers to pointers) are more efficient than pointers for the Macintosh memory manager. They can be moved after they have been allocated while pointers stay fixed. *)

vector = object (sequence)

elements: objArrayH;

containing the elements. The objects in the array do not belong to the internal state but the array does. *)

(* Handle to an array

(* Memory management *)

procedure init; **override**;

(* The global variable
'currentVectorLength' determines how many elements are allocated and must be set
before 'init' is called. *)

procedure initFromList (l: list);

(* Creates a vector from the
elements in the list 'l'. No call to 'init' is necessary. *)

procedure allocateElements (number: seqIndex0);

(* Reserves space for at
most 'number' elements. This method is used to dynamically extend or shrink a
vector although it should be used cautiously because it involves copying. *)

function allocatedElements: seqIndex0;

(* Returns the number of
currently allocated elements. *)

(* Rearranging the elements *)

procedure revert;

(* Reverts the order of the
elements in the vector. *)

procedure swap (i, j: seqIndex);

(* Exchanges the 'i'th and
the 'j'th element. *)

procedure randomShuffle (n, l: seqIndex0);

(* Chooses randomly 'n'
elements from the elements in 1..'l' and places them
into 1..'n'. *)

(* Sorting and searching *)

procedure sort

(**function** lessThan (p, q: obj): boolean;
l: seqIndex; r: seqIndex0);

(* Sorts 'elements' between
'l' and 'r' using the comparison function 'lessThan'. *)

procedure sortAll

(**function** lessThan (p, q: obj): boolean);

(* Sorts 'elements' between
1 and 'length' using the comparison function 'lessThan'.*)

function binarySearch

(x: obj; **function** compare (a, b: obj): relationObj;
l: seqIndex; r: seqIndex0): seqIndex0;

(* Finds 'x' between 'l' and 'r' in the vector which is sorted in ascending order. 'binarySearch = 0' means that 'x' was not found, otherwise 'binarySearch' gives the location of 'x' in the vector. *)

function findExtreme

(**function** lessThan (p, q: obj): boolean): seqIndex0;
 (* Finds the smallest object according to the comparison function 'lessThan'. *)

(* **Heap operations** *)

procedure heapify (**function** lessThan (p, q: obj): boolean);

(* Produces a heap using 'lessThan'. The first element is the minimum. *)

procedure sift

(**function** lessThan (p, q: obj): boolean;
 l, r: seqIndex);
 (* Sifts element 'l' into the heap which ranges from 'l' + 1 to 'r'. *)

procedure nextMin (**function** lessThan (p, q: obj): boolean);

(* Creates a new heap by removing the first element. *)

procedure insert

(x: obj; **function** lessThan (p, q: obj): boolean);
 (* Inserts an element into the heap. *)

procedure heapSort

(**function** greaterThan (p, q: obj): boolean);
 (* Sorts the whole vector using heapsort into ascending order. Note that the usual lessThan function will sort the wrong way! *)

function isHeap

(root: seqIndex; **function** lessThan (p, q: obj): boolean): boolean;
 (* Tests whether the vector with the given 'root' (usually 1) fulfills the heap property. *)

(* **Miscellaneous** *)

```

procedure appendSafe (x: obj; increment: seqIndex0);
                                (* Like append, but the
                                vector is extended by ‘increment’ if there is not sufficient space. *)

procedure preset (number: seqIndex0; t: objectType);
                                (* Creates a vector of
                                ‘number’ objects of type ‘t’. No call to init is necessary. *)

procedure eliminateDuplicates;
                                (* Removes from ‘self’ all
                                duplicated elements and destroys them with ‘freeAll’. *)

procedure forAllPermutations
                                (n: seqIndex0;
procedure whatToDo (v: vector));
                                (* Executes the procedure
                                ‘whatToDo’ for all permutations of the first ‘n’ elements. *)

procedure readLength;
                                (* Reads the length of the
                                vector from ‘currentStream.’ *)

end; (* vector *)

```

homogenVector = **object**(vector)

(* All objects are of the same
type and the operations are the same as for a ‘vector’. *)

end; (* homogenVector *)

```

elementH = ^^ record
    value: obj; (* The object. *)
    next : elementH; (* The successor. *)
end; (* elementH *)

```

list = **object**(sequence) (* linked list *)

first, last: elementH;
(* The first and the last element in the linked list.
*)

```

procedure concatenate (x: list);
                                (* Appends the list ‘x’ at the
                                end of the list. Note: No duplication is taking place and the list ‘x’ is left intact. *)

```

```

procedure flatten;
(* Appends to the list
recursively all elements which are member of a sequence which is a member of
'self'. The member sequences of 'self' are destroyed. *)

procedure push (x: obj);
(* Appends the element 'x'
at the front of the list. *)

procedure pop;
(* Removes the first
element from the list. *)

end; (* list *)

```

2.5 Arithmetic independent geometric primitives:

The basic geometric classes 'point2d', 'lineSegment', 'rectangle' and 'circle'

In this section we describe the elementary geometric objects and the geometric primitives available. We recommend to use the abstract class 'point2d' whenever possible in order to write code that is arithmetic independent.

```

type
    signType = -1..1;

```

```

point2d = object(obj)

```

```

procedure randomChange; override;
(* The global variable
'currentRandomConstraint' is interpreted as a rectangle. *)

(* Coordinate manipulation *)

function getX: extended;
(* Gets the x-coordinate. *)

procedure setX (newX: extended);
(* Sets the x-coordinate. *)

function getY: extended;
(* Gets the y-coordinate. *)

procedure setY (newY: extended);
(* Sets the y-coordinate. *)

function xLessThan (p: point2d): boolean;

```

```

*)
(* Is 'self.getX < p.getX' ?
*)
function xEqual (p: point2d): boolean;
(* Is 'self.getX = p.getX' ?
*)
function yLessThan (p: point2d): boolean;
(* Is 'self.getY < p.getY' ?
*)
function yEqual (p: point2d): boolean;
(* Is 'self.getY = p.getY' ?
*)

(* Drawing *)
procedure drawLine (p: point2d);
(* Draws a line from 'self'
to 'p'. *)
procedure labelPoint (l: str255);
(* Draws the string 'l'
below 'self'. *)

(* Geometric primitives *)
function whichSideX (p, q: point2d): extended;
(* Determines on which
side of the directed line segment from 'p' to 'q' the point
'self' lies.
< 0: 'self' lies to the left
of the directed line segment 'pq'
= 0: 'self' lies on the
directed line segment 'pq'
> 0: 'self' lies to the right
of the directed line segment 'pq'
'whichSideX' is also
twice the signed area of the triangle with vertices 'self', 'p' and 'q'. *)
function whichSideSign (p, q: point2d): signType;
(* Computes
'sign(whichSideX)' and rounds it to '0' if the exact result cannot be determined. *)
function crossProductX (p, q, r: point2d): extended;
(* Computes the cross
product ('self' - 'p') · ('r' - 'q'). *)
function distanceX (p: point2d): extended;
(* Computes the Euclidean
distance between 'self' and 'p'. *)

```

```

function squaredDistanceX (p: point2d): extended;
                                (* Computes the squared
Euclidean distance between 'self' and 'p'. *)

function squaredDxX (p: point2d): extended;
                                (* Computes the squared
difference in x-coordinates between 'self' and 'p'. *)

function squaredDyX (p: point2d): extended;
                                (* Computes the squared
difference in y-coordinates between 'self' and 'p'. *)

procedure circleCenterX
                                (p, q: point2d;
var cx, cy: extended);
                                (* Computes the point ('cx',
'cy'), the center of the circle through 'self', 'p'
                                and 'q'. *)

function intersectLineSegmentX
                                (q, r, s:
point2d;
                                left, right:
point2d): boolean;
                                (* 'true'  $\Leftrightarrow$  the segment
'self q' and 'rs' intersect. The left intersection point is 'left', the right intersection
point is 'right' (lexicographically). 'l.p' = 'l.q' is possible and common. *)

function xPlusYX: extended;
                                (* Computes 'x' + 'y'. *)

function xMinusYX: extended;
                                (* Computes 'x' - 'y'. *)

function rayEvalX
                                (p: point2d;
cosSlope, sinSlope: extended): extended;
                                (* Evaluates the positive
ray emanating from 'p' with the given slope at 'self.getX'. *)

function bisectorEvalX (p, q: point2d): extended;
                                (* Evaluates the bisector of
'p' and 'q' at 'self.getX'. *)

function intersectRayBisectorX
                                (cosSlope,
sinSlope: extended;
                                p, q,
intersection: point2d): boolean;
                                (* Tests whether the
positive ray emanating from 'self' with the given slope intersects the bisector given

```

```

by 'p' and 'q'. *)

function intersectBisectorBisectorX
    (p, q, r,
intersection: point2d): boolean;
    (* Tests whether the
bisector('self','p') intersects bisector('q','r') where  $|\{\text{'self'}, \text{'p'}, \text{'q'}, \text{'r'}\}| = 3$ . *)

function intersectRayRayX
    (cosLower,
sinLower: extended;
    p: point2d;
cosUpper, sinUpper: extended;
    intersection:
point2d): boolean;
    (* Tests whether the rays
('self', 'cosLower', 'sinLower') and ('p', 'cosUpper', 'sinUpper') intersect. *)

function middle (p: point2d): point2d;
    (* Computes the point
('self.getX + p.getX', 'self.getY + p.getY') / 2. *)

function createLineSegment (p: point2d): lineSegment;
    (* Creates a line segment
from 'self' to 'p'. *)

function createCircle (p, q: point2d): circle;
    (* Creates the circle
through 'self', 'p' and 'q' if they are not collinear. *)

end; (* point2d *)

lineSegment = object(obj)

    p, q: point2d;
    (* Points 'p' and 'q' belong
to the internal state. *)

    procedure init; override;
    (* The global variable
'currentPointType' determines which kind of points are used. *)

    procedure randomChange; override;
    (* The global variable
'currentRandomConstraint' is interpreted as a rectangle. *)

    function whichSideX (r: point2d): extended;
    (* Determines on which
side of the directed (from 'p' to 'q') line segment 'self'

```


the point ' r ' lies.
 < 0 : ' r ' lies to the left of
 $= 0$: ' r ' lies on the
 > 0 : ' r ' lies to the right of

the directed line segment 'self'
directed line segment 'self'
the directed line segment 'self' *)

'whichSideX' is also
twice the signed area of the triangle with vertices ' r ', ' $\text{self}.p$ ' and ' $\text{self}.q$ '. This
function is similar to ' $\text{point2d}.whichSideX$ ' and introduced here for convenience. *)

```

function intersectLineSegmentX (r, l: lineSegment):
boolean;
                                (* 'true'  $\Leftrightarrow$  the segment
                                'self' and ' $r$ ' intersect. The intersection point is ' $l$ ', where ' $l.p \leq l.q$ '
                                lexicographically (' $l.p = l.q$ ' is possible and common). *)

function length: real;
                                (* Computes the length of
                                the segment. *)

function collinear (l: lineSegment): boolean;
                                (* Tests whether 'self' and
                                ' $l$ ' are parallel. *)

end; (* lineSegment *)

```

circle = object(obj)

```

x, y, radius: real;

procedure randomChange; override;
                                (* The global variable
                                ' $\text{currentRandomConstraint}$ ' is interpreted as a rectangle. *)

function getRadius: markedRealPoint;
                                (* Creates a
                                'markedRealPoint' marked with the radius. *)

function inCircle (p: point2d): boolean;
                                (* Tests whether ' $p$ ' is in
                                the circle. *)

procedure makeFrom2points (p, q: point2d);
                                (* Creates the smallest
                                circle through the two points ' $p$ ' and ' $q$ '. *)

procedure makeFrom3points (p, q, r: point2d);
                                (* Creates the smallest
                                circle through the three points ' $p$ ', ' $q$ ' and ' $r$ '. *)

```

```

procedure makeDisk (l: list);
(* Creates the smallest
circle through the points in 'l'. *)

procedure makeFromPointCircle (p: point2d; c: circle);
(* Creates the smallest circle through 'p' and 'c' with 'not inCircle()' . *)

end; (* circle *)

rectangle = object(obj)

    left, bottom, right, top: real;
(* (left ≤ right) ∧ (bottom ≤
top) *)

    procedure randomChange; override;
(* The global variable
'currentRandomConstraint' is interpreted as a rectangle. *)

end; (* rectangle *)

```

(* Comparison functions for points *)

```

function pointLessThanXY (p, q: point2d): boolean;
(* Lexicographical
comparison, first the x-coordinate then the y-coordinate. *)

function pointLessThanYX (p, q: point2d): boolean;
(* Lexicographical
comparison, first the y-coordinate then the x-coordinate. *)

function pointGreaterThanXY (p, q: point2d): boolean;

function pointGreaterThanYX (p, q: point2d): boolean;

function comparePointXY (p, q: point2d): relationObj;
(* Determines which relation
holds between 'p' and 'q'. *)

function comparePointYX (p, q: point2d): relationObj;

```

2.6 Common universal abstract data types: the classes 'dictionary' and 'priorityQueue'

The most common abstract data types encountered in geometric algorithms are the dictionary and the priority queue. Both data types are implemented in the XYZ GeoBench in a general way based on the *reference* concept. A reference is the abstraction of location in a data structure and can be viewed as a pointer into the data structure. For example, when a data item is inserted

into a dictionary, the dictionary returns a reference which can be used to delete the data item or to change its value.

type

```
reference      = ^integer;    (* Any pointer suffices. *)
directionType = (left, right);
```

dictionary = object(obj)

```
numberOfElements: longInt;
```

```
function getObject (where: reference): obj;
(* Retrieves the object
referenced by 'where'. *)
```

```
procedure setObject (where: reference; newValue: obj);
(* Changes the object
referenced by 'where' into 'newValue'. *)
```

```
procedure sequenceToDictionary
(s: vector; l:
seqIndex1; r: seqIndex0);
(* Equivalent to inserting
the sorted vector elements between 'l' and 'r' into an empty dictionary. *)
```

```
function find
(x: obj; function
compare (a, b: obj): relationObj;
var where:
reference;
var direction:
directionType): boolean;
(* If 'find' = 'true' then
'where' points to some element 'e' with 'compare(e, x) = equalObj'. If 'find' = 'false'
then either 'where' = 'nil' (dictionary empty) or 'x' is a direct neighbor of 'where' in
direction 'direction'. *)
```

```
function member
(x: obj;
function compare
(a, b: obj): relationObj): boolean;
(* Tests whether 'x' is in the
dictionary or not. *)
```

```
function insert
(x: obj; where:
reference;
direction:
directionType): reference;
(* If 'where' = 'nil' then
```

insert 'x' into the dictionary which must be empty. If '*where*' \neq '**nil**' then insert 'x' before ('*direction*' = '*left*') or after ('*direction*' = '*right*') '*where*'. *)

```

function insertObj
    (x: obj; function
    compare (a, b: obj): relationObj;
    boolean): reference;
    (var found:
    (* Inserts 'x' into the
    dictionary. 'found' = 'false'  $\Leftrightarrow$  'x' is unique. *)

procedure insertNewObj
    (x: obj; function
    compare (a, b: obj): relationObj);
    (* Inserts 'x' if it is not a
    member of the dictionary, otherwise this is an error. *)

procedure delete (x: reference);
    (* Removes the element
    referenced by 'x' from the dictionary. *)

procedure swap (p, q: reference);
    (* Exchanges the elements
    referenced by 'p' and 'q' in the dictionary. *)

procedure rangeSwap (p, q: reference);
    (* Exchanges the elements
    between 'p' and 'q' in the dictionary ('compare(p, q) = lessThanObj' must hold and is
    not checked). *)

function next
    (x: reference;
    direction: directionType): reference;
    (* Find the predecessor
    ('direction' = 'left') or the successor ('direction' = 'right') of 'x'. *)

function extreme (direction: directionType): reference;
    (* Find the leftmost
    ('direction' = 'left') or the rightmost ('direction' = 'right') value. *)

function isEmpty: boolean;
    (* Determines whether the
    dictionary is empty. *)

procedure forAll
    (procedure
    whatToDo (x: obj); direction: directionType);
    (* Performs 'whatToDo' on
    all elements 'x' of the dictionary, starting at the location most in direction 'direction'.
    *)

procedure rangeForAll

```

```

rightBound: obj;
(a, b: obj): relationObj;
whatToDo (x: obj); direction: directionType);
(* Performs 'whatToDo' on
all elements 'x' of the dictionary that are between 'leftBound' and 'rightBound'
(including), starting at the location most in direction 'direction'. The value 'nil'
serves as  $-\infty$  for 'leftBound' and as  $+\infty$  for 'rightBound'. *)

function compareReference (p, q: reference): relationObj;
(* Determines the order of
the elements referenced by 'p' and 'q' in the dictionary (without key operations!). *)

function infoString: str255;
(* Computes information
about the dictionary (this only makes sense for a 'dictionaryStatistics'). *)

end; (* dictionary *)

```

Dictionaries come in four different flavors, differing mainly in their internal realizations and the corresponding initialization routines.

type

avlTree = object(dictionary)

```

(* Implementation as an AVL
tree with optimal insertion / deletion costs. *)

root: avlNodeH;
(* No global variables are
needed for the initialization procedure 'init'. *)

```

end; (* avlTree *)

sortedList = object(dictionary)

```

(* Implementation as a sorted
list. Insertion / deletion is expensive. *)

minimum, maximum: sortedListNodeH;
(* No global variables are
needed for the initialization procedure 'init'. *)

```

```
end; (* sortedList *)
```

```
sortedVector = object(dictionary)
```

```
    (* Implementation as a sorted  
    array. Insertion / deletion is expensive. *)
```

```
    procedure init; override;
```

```
    (* The global variable  
    ‘currentVectorLength’ determines how many elements are allocated. *)
```

```
end; (* sortedVector *)
```

```
dictionaryStatistics = object(dictionary)
```

```
    d: dictionary;  
    (* Actual dictionary that is  
    used. *)
```

```
    maxLength,  
    (* Maximal number of  
    elements in the dictionary. *)
```

```
    insertions,  
    (* Number of insertions  
    into the dictionary. *)
```

```
    deletions: longInt;  
    (* Number of deletions  
    from the dictionary. *)
```

```
    procedure init; override;
```

```
    (* The global variable  
    ‘currentDictionaryType’ determines which kind of dictionary is used for ‘d’ and  
    initializes it. Note that the initialization of ‘d’ might require the correct setting of  
    additional global variables. *)
```

```
end; (* dictionaryStatistics *)
```

Besides the abstract data type dictionary, we support the priority queue. Again, priority queue is an abstract class realized either as a heap or a dictionary. In the first case an efficient find operation is not possible while in the second case we can guarantee a logarithmic time for find.

type

priorityQueue = object(obj)

```
function getObject (where: reference): obj;  
(* Retrieves the object  
referenced by 'where'. *)  
  
function insert  
(* x: obj;  
function compare  
(a, b: obj): relationObj): reference;  
(* Inserts 'x' into the  
priority queue using 'compare' giving a reference for  
later removal. *)  
  
procedure delete  
(* x: reference;  
function compare  
(a, b: obj): relationObj);  
(* Deletes the object  
referenced by 'x' from the priority queue. 'compare' may be used for restructuring. *)  
  
function isEmpty: boolean;  
(* Tests whether the queue  
is empty. *)  
  
function minimum: reference;  
(* Retrieves the minimum  
without deleting it. *)  
  
procedure forAll (procedure whatToDo (x: obj));  
(* Performs the procedure  
'whatToDo' on all elements 'x' of the priority queue. *)  
  
function infoString: str255;  
(* Computes information  
about the dictionary (this only makes sense for a 'priorityQueueStatistics'. *)  
  
end; (* priorityQueue *)
```

The abstract class 'priorityQueue' is realized in three different ways.

type

dictPriorityQueue = object(priorityQueue)

(*Priority queue implemented with a dictionary with an efficient find and next operation. *)

d: dictionary;

procedure init; **override**;

(* The global variable
'currentDictionaryType' determines which kind of dictionary is used for 'd' and
initializes it. Note that the initialization of 'd' might require the correct setting of
additional global variables. *)

function find

(x: obj; **function**

compare (a, b: obj): relationObj;

var where:

reference;

var direction:

directionType): boolean;

(* If 'find' = 'true' then
'where' points to some element 'e' with 'compare(e, x) = equalObj'. If 'find' = 'false'
then either 'where' = 'nil' (dictionary empty) or 'x' is a direct neighbor of 'where' in
direction 'direction'. *)

function next (x: reference;

direction:

directionType): reference;

(* Finds the predecessor
('direction' = 'left') or the successor ('direction' = 'right')
of 'x' *)

end; (* dictPriorityQueue *)

heapPriorityQueue = **object**(priorityQueue)

(* Implements the priority queue as a heap. *)

procedure init; **override**;

(* The global variable
'currentVectorLength' determines how many elements
are allocated. *)

end; (* heapPriorityQueue *)

priorityQueueStatistics =
object(priorityQueue)


```

(* Instruments a priority queue. *)

currentLength,
elements in the priority queue. *)
maxLength,
elements in the priority queue. *)
insertions,
into the priority queue. *)
deletions: longInt;
from the priority queue. *)
p: priorityQueue;
that is used. *)

procedure init; override;
(* The global variable
'currentPriorityQueueType' determines which kind of dictionary is used for 'p' and
initializes it. Note that the initialization of 'p' might require the correct setting of
additional global variables. *)

end; (* priorityQueueStatistics *)

```

We describe in section 2.8 a convenient procedure that lets the user choose between different implementations of dictionaries and priority queues (function ‘getXY’).

2.7 Graphs: the classes ‘graphEdge’, ‘simpleUndirectedGraph’, ‘undirectedGraph’, ‘directedGraph’ and ‘spanningTree’

The GeoBench is primarily designed for geometric computation. Nevertheless we support graphs in a limited way.

type

```

graphEdge = object (obj)

startVertex, endVertex: seqIndex;
corresponding vector of vertices. *)
(* Pointers to the

procedure drawEdge

```

```

                                (g:
simpleUndirectedGraph; directed: boolean);
                                (* Displays the edge which
is part of graph 'g'. 'directed' specifies whether the edge should be drawn as a
directed edge or not. *)

end; (* graphEdge *)

simpleUndirectedGraph = object (obj)

    vertices: vector;
                                (* Specifies the nodes. *)

    edges: homogenVector;
                                (* A vector of 'graphEdge'.
    *)

    procedure init; override;
                                (* The global variables
'currentVertices' and 'currentEdges' determine how many vertices and edges are
allocated. *)

    procedure addEdge (e: graphEdge);
                                (* Adds the edge 'e' to the
graph. *)

    function minimumSpanningTree: spanningTree;
                                (* Computes a minimum
spanning tree under the assumption that the edges are sorted by length into ascending
order. *)

    function perimeter: real;
                                (* Computes the total
length of all edges in the graph under the assumption that the vertices are points
(member of the 'point2d' class). *)

end; (* simpleUndirectedGraph *)

```

```

spanningTree =
    object (simpleUndirectedGraph)

    function TSPERMST: vector;
                                (* Traverses the spanning

```

tree and produces in the Euclidean case as result a traveling salesman tour that is at most twice as long as the optimal tour. *)

end; (* spanningTree *)

directedGraph =

object (simpleUndirectedGraph)

adjacency: homogenVector;

(* Of list of graphEdge. *)

procedure init; **override;**

(* The global variables

'currentVertices' and 'currentEdges' determine how many vertices and edges are allocated. *)

procedure initAdjacency;

(* Constructs the adjacency

lists for the already existing graph. This method can be used to transform a simple undirected graph into a directed graph. *)

end; (* directedGraph *)

undirectedGraph = **object** (directedGraph)

(* The undirected graph has the same operations as an directed graph. *)

end; (* undirectedGraph *)

2.8 Support for animation, user interaction and error checking

Animation

Algorithm animation is primarily used for two purposes: Demonstrating algorithms and debugging them. In order to animate an algorithm the implementor chooses a graphical representation of the program state and decides when and where this information needs to be updated. The typical code looks as follows:

```
...
(* Geometric algorithm changing internal state. *)
(* $IFC AnimationEnabled AND myAlgAnim *)
if animationFlag[myAlgAnimItem] then
  (* Update graphical state information. Often: show a picture. *)
  waitForClick(animationFlag[myAlgAnimItem]);
  (* Update graphical state information. Often: hide a picture. *)
end;
```

```
(* $ENDC *)
```

```
...
```

The conditional compilation variable ‘*AnimationEnabled*’ serves as a global flag for enabling animation for the whole system while ‘*myAlgAnim*’ is a local flag enabling or disabling animation for a specific algorithm.

The procedure ‘waitForClick’ stops the algorithm and lets the user choose what to do next. For the choice of ‘myAlgAnimItem’ we distinguish two cases: 1) An appropriate flag is already defined since there are already implementations solving the problem (e.g. ‘myAlgAnimItem = AconvexHullItem’, if we implement another algorithm for the convex hull) or 2) there is no such flag. In case 1) nothing needs to be done while in case 2) we define another constant, say ‘myAlgAnimItem’, in module ‘GeoBenchUtility’ and update the constant ‘maxAnimation’ accordingly. In procedure ‘initAnimation’ in module ‘GeoBenchUtility’ we add the line

```
menuEntry(myAlgAnimItem, 'My algorithm');
```

and algorithm animation is possible.

In addition to visually animating a program, one can give feedback on whether a program is executing or the machine hangs. This is done by advancing the hands of the stopwatch. The procedure ‘advanceStopWatchHands’ from the module ‘GeoBenchUtility’ should be called sufficiently often.

```
procedure advanceStopWatchHands;                                (* Advances the hands of
the stop watch. *)
```

Parameter input

Sometimes an algorithm needs additional input from the user. We provide in the module ‘getUserParameter’ three different kinds of interactive input procedures.

1) Get one to three numerical values

```
function getUserParameter1                                     (title:    str255;
default: extended; var value: extended;                        function check
(x: extended): boolean): boolean;
function getUserParameter2                                     (title1,   title2:
str255;                                                default1,
default2: extended;                                var    value1,
value2: extended;                                    function check1
(x: extended): boolean;                                function check2
```

```

(x: extended): boolean): boolean;

function getUserParameter3
                                (title1:  str255;
default1: extended;
                                var    value1:
extended;
                                function check1
(x: extended): boolean;
                                title2:  str255;
default2: extended;
                                var    value2:
extended;
                                function check2
(x: extended): boolean;
                                title3:  str255;
default3: extended;
                                var    value3:
extended;
                                function check3
(x: extended): boolean): boolean;
                                (* The string 'title' specifies
the appropriate title for the numerical value, 'default' gives the default value and
function 'check' tests whether the user supplied value is acceptable. The boolean
result 'true' indicates that the operation was performed successfully while 'false'
indicates that the operation was canceled or some numerical value was not accepted
by the 'check' function(s). *)
function noConstraint (x: extended): boolean;
                                (* This function returns
always 'true' and can be used if a numerical value is unconstrained. *)

```

2) Get one or two string values

```

function getUserString1
                                (title:  str255;
var value: str255): boolean;

function getUserString2
                                (title1,  title2:
str255;
                                var    value1,
value2: str255): boolean;
                                (* The string 'title' specifies
the appropriate title for the string value. 'value' contains at entry the default value
and at exit the user supplied value. The boolean result 'true' indicates that the
operation was performed successfully while 'false' indicates that the operation was
canceled. *)

```

3) Ask the user for the appropriate data structures

```

function getXY
                                (xLength:
seqIndex0; mustBeDictPriorityQueue: boolean;
                                var      xQueue:
priorityQueue;
                                yLength:
seqIndex0; var yTable: dictionary): boolean;
                                (* If 'xLength' > 0 the user
is asked to choose the data type of the priority queue 'xQueue'. The value of
'xLength' indicates the maximal number of entries in the priority queue. The value of
'mustBeDictPriorityQueue' specifies whether the priority queue must be realized as a
dictionary ('mustBeDictPriorityQueue' = 'true') or whether a heap implementation is
admissible ('mustBeDictPriorityQueue' = 'false'). If 'yLength' > 0 the user is asked
to choose the data type of the dictionary 'yTable'. The value of 'yLength' indicates
the maximal number of entries in the dictionary. 'getXY' = 'false'  $\Leftrightarrow$  the operation
was canceled by the user. *)

```

Display of additional information

Algorithms that need to display information (e.g. when statistics information is collected by various abstract data types) can display an info string using the procedure 'displayInfoString' from the module 'infoWindow'.

```

procedure displayInfoString (info: str255);
                                (* Displays the string 'info'
in the info window of the XYZ GeoBench. *)

```

Error checking

We advocate the use of assertions to check invariants. The module 'GeoBenchUtility' provides the procedure `assert`.

```

procedure assert (condition: boolean; t: str255);
                                (* If 'condition' = false' the
user gets the warning that the assertion 't' has failed. The user can abort the program
or continue. *)

```

2.9 Implemented geometric algorithms

This section describes the implemented geometric algorithms and their interfaces. Many algorithms contain a boolean parameter '*ask*' which determines whether the user is asked to choose how certain data structures are implemented at run time.

type

lineSegmentVector = **object**(homogenVector)

(*
segments. *)

A collection of line

(* **Intersection routines** *)

function simpleFirstIntersect: obj;
(* Computes the first
intersection using the trivial method. The result can be a 'point2d' object or a
'lineSegment' object or 'nil' if no intersection exists. *)

function planeSweepFirstIntersect (ask: boolean): obj;
(* Computes the first
intersection using a plane sweep. The result can be a 'point2d' object or a
'lineSegment' object or 'nil' if no intersection exists. *)

function simpleAllIntersect: vector;
(* Computes all pairwise
intersections with the trivial algorithm. The result is a vector of points and line
segments or 'nil' if no intersection exists. *)

function planeSweepAllIntersect (ask: boolean): vector;
(* Computes all
intersections using a plane sweep algorithm. The result is a vector of points and line
segments or 'nil' if no intersection exists. If 'ask = true' the user is asked to choose
how the x-queue and the y-table should be implemented. *)

function hvPlaneSweepAllIntersect (ask: boolean): vector;
(* Like
'planeSweepAllIntersect' but all line segments must be either horizontal or vertical.
*)

(* **Projection routines** *)

function projectOnX: lineSegmentVector;
(* Projects all line segments
on the x-axis and produces a new 'lineSegmentVector'. *)

function projectOnY: lineSegmentVector;
(* Projects all line segments
on the y-axis and produces a new 'lineSegmentVector'. *)

function projectOnXY: lineSegmentVector;
(* Projects all line segments
on the axis that results in a shorter segment and produces a new 'lineSegmentVector'.
*)

(* **Miscellaneous** *)

```

function lowerEnvelope: obj;
                                (* Computes the lower
envelope of a set of line segments. The result is in fact a 'polyLine'. *)

function isHorizontalVertical: boolean;
                                (* Tests whether all line
segments are either horizontal or vertical. *)

function eliminateZeroLengthSegments: lineSegmentVector;
                                (* Eliminates all line
segments where the start point and the end point coincide and produces a new
'lineSegmentVector'. *)

end; (* lineSegmentVector *)

pointVector = object (homogenVector)

    (* Convex hull *)

    procedure convexHull
                                (eliminate:
boolean; var hullLength: seqIndex0);
                                (* Computes in place the
convex hull of the given points using the Graham scan. The elements from 1 to
'hullLength' constitute a convex polygon whereas the elements from 'hullLength + 1'
to 'self.length' are the inner points. *)

    function convexHullDivideAndConquer: convexPolygon;
                                (* Computes the convex
hull using the divide and conquer algorithm. *)

    (* Closest pair *)

    function closestPairHeuristic: lineSegment;
                                (* Computes the closest
pair using the heuristic method which sweeps only in x-direction. The resulting
'lineSegment' is formed by the closest pair and might have zero length. The order of
the input data points might change. *)

    function closestPair (ask: boolean): lineSegment;
                                (* Computes the closest
pair using the plane sweep method. The resulting 'lineSegment' is formed by the
closest pair and might have zero length. The order of the input data points might
change. *)

    function closestPairProbabilistic
                                (ask: boolean):
lineSegment;
                                (* Computes the closest

```


pair using Rabin's probabilistic algorithm. The resulting 'lineSegment' is formed by the closest pair and might have zero length. The order of the input data points might change. 'ask = true' asks the user for the size of the hash table, otherwise the size is determined by the algorithm. This is useful for demonstration purposes where giving a large hash table causes few or no

collisions. *)

function closestPairNewHeuristic: lineSegment;
 (* Computes the closest pair using the heuristic method which sweeps simultaneously in x- and y-direction. The resulting 'lineSegment' is formed by the closest pair and might have zero length. The order of the input data points might change. *)

function closestPairDivideAndConquer: lineSegment;
 (* Computes the closest pair using the divide and conquer method. The resulting 'lineSegment' is formed by the closest pair and might have zero length. Animation is currently not supported for this algorithm. The program is adapted from 'Algorithms from P to NP, Volume I - Design & Efficiency', by B. Moret and H. Shapiro. *)

function closestPairN2: lineSegment;
 (* Computes the closest pair using the trivial quadratic method. The resulting 'lineSegment' is formed by the closest pair and might have zero length. The order of the input data points does not change. *)

(* All nearest neighbors to the left *)

function leftANN (ask: boolean): homogenVector;
 (* Computes all nearest neighbors to the left using the plane sweep method. The result is a homogeneous vector of line segments where the start point is a given point and the end point is the start point's nearest neighbor to the left. The left most point gets itself as nearest neighbor to the left. The order of the input data points might change. *)

function leftANNHeuristic: homogenVector;
 (* Computes all nearest neighbors to the left using the projection-on-x-only method. The result is a homogeneous vector of line segments where the start point is a given point and the end point is the start point's nearest neighbor to the left. The left most point gets itself as nearest neighbor to the left. The order of the input data points might change. *)

function leftANNNewHeuristic: homogenVector;
 (* Computes all nearest neighbors to the left using the projection method. The result is a homogeneous vector of line segments where the start point is a given point and the end point is the start point's nearest neighbor to the left. The left most point gets itself as nearest neighbor to the left. The order of the input data points might change. *)

```

function leftANNInSector
    (ask: boolean;
     cosLower,
     sinLower, cosUpper, sinUpper: extended):
        homogenVector;
    (* Computes all nearest
    neighbors to the left in the sector that is bounded by the rays with slope 'cosLower /
    sinLower' and 'cosUpper / sinUpper' using the plane sweep method. The result is a
    homogeneous vector of line segments where the start point is a given point and the
    end point is the start point's nearest neighbor to the left in the specified sector. The
    left most point gets itself as nearest neighbor to the left. The order of the input data
    points might change. *)

```

(* Voronoi diagram *)

```

function voronoiDiagramSweepLine
    (ask: boolean):
    voronoiDiagram;
    (* Computes the Voronoi
    diagram using the plane sweep method. The order of the input data points does not
    change. *)

function voronoiDiagramDivideAndConquer: voronoiDiagram;
    (* Computes the Voronoi
    diagram using the divide and conquer method. The order of the input data points
    does not change. *)

```

(* Euclidean minimum spanning tree *)

```

function EMST: spanningTree;
    (* Computes a Euclidean
    minimum spanning tree of the point set using a quadratic algorithm. The order of the
    input data points does not change. *)

```

(* Traveling salesman *)

```

function TSPERMST: polygon2d;
    (* Computes a tour of the
    traveling salesman through the given points with the Euclidean minimum spanning
    tree heuristic. The order of the input data points does not change. *)

```

```

function TSPNN: polygon2d;
    (* Computes a tour of the
    traveling salesman through the given points with the nearest neighbor heuristic. The
    order of the input data points does not change. *)

```

```

function TSPConvexHull: polygon2d;
    (* Computes a tour of the
    traveling salesman through the given points with the convex hull heuristic. The order
    of the input data points does not change. *)

```

(* Minimal area disk *)

function minimalDisk: circle;

(* Computes the smallest circle which contains the given points using a randomized algorithm. The order of the input data points might change. *)

function simpleMinimalDisk: circle;

(* Computes an approximation to the smallest circle which contains the given points using a heuristic algorithm. The order of the input data points might change. *)

function containedInCircle (c: circle): boolean;

(* Tests whether all the given points lie in circle 'c'. The order of the input data points does not change. *)

(* Projection *)

function projectOnX: pointVector;

(* Projects the points on the x-axis and produces a new 'pointVector'. The order of the input data points does not change. *)

function projectOnY: pointVector;

(* Projects the points on the y-axis and produces a new 'pointVector'. The order of the input data points does not change. *)

(* Test data generation *)

function createGrid (yCoordinates: pointVector):
pointVector;

(* Computes a grid G of points of size 'self.length · yCoordinates.length' where $G = \{p: \exists u \in \text{'self.elements'}: \exists v \in \text{'yCoordinates.elements'}: p_x = u_x \wedge p_y = v_y\}$. The order of the input data points does not change. *)

function createLineSegmentVector

(q: pointVector):
lineSegmentVector;

(* Creates a vector of line segments where 'self' provides the starting points and 'q' the end points. Both must have the same number of elements. The order of the input data points does not change. *)

(* Conversions *)

```

function makePolygon: polygon2d;
(* Produces a polygon from
the given points. The order of the input data points does not change. *)

function makePolyLine: polyLine;
(* Produces a poly line
from the given points. The order of the input data points does not change. *)

function makeSimplePolygon: polygon2d;
(* Produces a simple
polygon using a modification of Graham's scan. The order of the input data points
does not change. *)

function makeStarShapedPolygon: polygon2d;
(* Produces a star-shaped
polygon. A rotational sweep around the center of gravity of the first three points in
the input 'pointVector' 'self' is used. The order of the input data points does not
change. *)

end; (* pointVector *)

```

polyLine = object(pointVector)

```

function toLineSegmentVector: lineSegmentVector;
(* Computes a vector of
line segments that corresponds to the edges of the poly line or polygon. *)

function perimeter: real;
(* Computes the sum of the
length of all edges. *)

end; (* polyLine *)

```

```

polRange = 0..maxPolysM1;
polSet   = set of polRange;

```

polygon2d = object(polyLine)

```

function windingNumber
(* Computes the winding
number of 'r' around the polygon. 'onBoundary = true' ⇔ the point 'r' lies on the
polygon's boundary. *)
(r: point2d; var
onBoundary: boolean): longInt;

```

```

function TSPOptimize
    (whichN: longInt;
ask: boolean): polygon2d;
    (* Tries to shorten a given
traveling salesman tour. The parameters 'whichN' and 'ask' determine what kind of
optimization is tried. The reader is referred to the source code for their precise
meaning. *)

function clip (ls: lineSegment): polygon2d;
    (* Clips 'self' on the given
lineSegment. Everything on the right side (the line segment 'ls' itself excluded) of
the directed line segment 'ls' is assumed to be visible. Algorithm: Sutherland-
Hodgman. *)

procedure intersection
    (pols: sequence;
function zoneQ (s: polSet): boolean;
    result: list;
ask: boolean);
    (* Computes boolean
operations on polygons using the sweepline algorithm of Nievergelt and Preparata
and adds the resulting polygonal pieces to the list 'result'. 'pols' is a sequence of
polygons enumerated from 1 to 'pols.length'. The function 'zoneQ' determines
which areas belong to the result. To get the union of all polygons use 'zoneQ :=
polSet ≠ []'. To get the intersection, use 'zoneQ := polSet = [1..pols.length]'. In
particular, a single polygon can be decomposed into simple parts by using the
function 'zoneQ := polSet ≠ []' (menu entry 'Decompose'). To divide a polygon into
its 'simple hull' and zero or more simple polygons inside, use 'zoneQ := polSet = []'
(menu entry 'Simplify'). *)

function simpleSelfIntersect: vector;
    (* Computes all
intersections of edges of a polygon that do not occur at vertices using the trivial
algorithm. *)

function selfIntersect: vector;
    (* Computes all
intersections of edges of a polygon that do not occur at vertices using a boundary
traversal algorithm. This algorithm is efficient for the class of polygons that have a
left turn in each vertex. *)

end; (* polygon2d *)

convexPolygon = object (polygon2d)

    function intersect (x: convexPolygon): convexPolygon;
        (* Computes the
intersection of the two convex polygons using the boundary traversal method. *)

    procedure tangent

```

```

(c:
convexPolygon;
var upperP,
upperQ, lowerP, lowerQ: seqIndex);
(* Computes the two
common outer tangents of the two convex polygons. The lower tangent is given by
index 'lowerP' in 'self' and 'lowerQ' in 'c' and the upper tangent is given by index
'upperP' in 'self' and 'upperQ' in 'c'. *)

procedure diameter (var i, j: seqIndex0);
(* Computes the indices 'i'
and 'j' of the two points determining the diameter of the convex polygon. *)

function inside (p: point2d): boolean;
(* Tests whether point 'p' is
inside the convex polygon or not. The boundary belongs per definition to the inside.
*)

end; (* convexPolygon *)

voronoiDiagram = object (obj)

function delaunayTriangulation: simpleUndirectedGraph;
(* Computes the Delaunay
triangulation, the dual to the Voronoi diagram. *)
function EMST: spanningTree;
(* Computes a Euclidean
minimum spanning tree from the Voronoi Diagram. *)

function leftANN: homogenVector;
(* Computes a vector of
line segments such that each point of the Voronoi diagram occurs as a start point of a
segment and the end point of the segment is its nearest neighbor to the left. The
leftmost point does not occur in this collection (in contrast to the nearest neighbor to
the left algorithms that work directly on point sets. *)

end; (* voronoiDiagram *)

rectangleVector = object (homogenVector)

function boundingBox: rectangle;
(* Computes the smallest
rectangle that contains all the given rectangles. *)

function contourOfUnionOfRectangles: lineSegmentVector;
(* Computes a set of
horizontal and vertical line segments that determines the contour of the given
rectangles using a plane sweep algorithm. *)

```

```
end; (* rectangleVector *)
```

```
dDimPoint = object(obj)
```

```

procedure randomChange; override;
(* The global variable
'currentRandomConstraint' is interpreted as a rectangle. *)

function distance (p: dDimPoint): extended;
(* Computes the Euclidean
distance between the  $d$ -dimensional points 'self' and ' $p$ '. *)
```

```
end; (* dDimPoint *)
```

```
dDimCircle = object(obj)
```

```

procedure randomChange; override;
(* The global variable
'currentRandomConstraint' is interpreted as a rectangle. *)

function inCircle (p: dDimPoint): boolean;
(* Tests whether the  $d$ -
dimensional point ' $p$ ' is inside the  $d$ -dimensional circle 'self'. *)

function inCircleEps
(* Tests whether the  $d$ -
dimensional point ' $p$ ' is in the  $d$ -dimensional circle 'self' whose radius is enlarged by
a factor of  $(1 + \text{'tolerance'})$ . *)
(p: dDimPoint;
tolerance: extended): boolean;

procedure makeDisk (l: list);
(* Creates the smallest disk
in  $d$ -space having the  $d$ -space points from ' $l$ ' on the boundary. *)

procedure makeFromPointCircle (p: dDimPoint; c:
dDimCircle);
(* Creates the smallest
circle through ' $p$ ' and ' $c$ ' with ' $\text{not inCircle}(p)$ '. *)
function randomPoint: dDimPoint;
(* Creates a uniformly
distributed random point inside the disk. *)
```

```
end; (* dDimCircle *)
```

dDimPointVector = **object**(homogenVector)

```
function minimalDisk
    (tolerance:
    extended; initialize: boolean): dDimCircle;
    (* Computes the minimal
    area disk that contains the given points. The radius is correct within a factor of (1 +
    'tolerance'), i.e. the radius is exact if 'tolerance = 0'. If 'initialize = true' the
    algorithm is initialized with the pair of points that have the largest coordinate
    difference in any direction. *)

function simpleMinimalDisk
    (shuffle,
    initialize: boolean): dDimCircle;
    (* Computes a disk
    containing all given points using a heuristic that is guaranteed to produce a circle
    whose radius is at most twice as large as the radius of the minimal area disk. If
    'shuffle = true' the points are randomly shuffled before the algorithm starts since it
    depends of the order of the input data points. The parameter 'tolerance' has the same
    meaning as in the previous method. *)

function worstSimpleMinimalDisk: dDimCircle;
    (* Rearranges the points in
    such a way that the worst case order for the previous enclosing disk algorithm is
    achieved. This algorithm has running time proportional to the factorial of the number
    of given points. *)

function containedInCircle (c: dDimCircle): boolean;
    (* Tests whether all given
    points lie inside the  $d$ -dimensional circle 'c'. *)

function projectOnXY: pointVector;
    (* Projects the given points
    on the x-y-plane and produces a collection of 2-dimensional points (type 'point2d').
    *)

end; (* dDimPointVector *)
```

twoDtree = **object**(binaryTree)

```
procedure insert (p: point2d);
    (* Inserts the point 'p' into
    the 2-d tree. *)

function rangeQuery (r: rectangle): pointVector;
    (* Computes all points that
    lie inside 'r' or 'nil' if none exists. *)
```


end;

Appendix A: Syntax of the textual I/O format

Axiom of the grammar is ‘List’, i.e. GeoBench expects a list of objects as textual input.

Digit	=	‘0’ ‘1’ ‘2’ ‘3’ ‘4’ ‘5’ ‘6’ ‘7’ ‘8’ ‘9’	
Natural	=	Digit { Digit }	
Integer	=	[‘-’] Natural	
Real	=	Integer [‘.’ Natural] [‘E’ [‘+’ ‘-’] Natural]	
Float	=	Natural Natural Real (* base precision value *)	
String	=	“ Char { Char } “ (* Char ≠ “ *)	
Circle	=	(‘CIR’ Real Real Real (* x y radius *))’
ConvexPolygon	=	(‘CPL’ PointList)’
ColorQuickDrawPicture	=	(‘CQP’ Real Real Real Real (* left bottom right top *))’
DDimCircle	=	(‘DCI’ CoordinateList Real (* coordinates radius *))’
DDimPoint	=	(‘DPT’ CoordinateList (* coordinates *))’
DDimPointVector	=	(‘DPV’ DDimPointList)’
DirectedGraph	=	(‘DGR’ Vector HomogenVector (* vertices edges *))’
FloatPoint	=	(‘FPT’ Float Float (* x y *))’
GraphEdge	=	(‘GED’ Integer Integer (* startVertex endVertex *))’
HomogenVector	=	(‘HVC’ HomogenObjectList)’
Int	=	(‘INT’ Integer)’
IntegerPoint	=	(‘IPT’ Integer Integer (* x y *))’
Layer	=	(‘LAY’ Real Real Vector (* z thickness objects *))’
LayerVector	=	(‘LAV’ Real Vector (* dz objects *))’
LineSegment	=	(‘LSG’ Point Point)’
LineSegmentVector	=	(‘LVC’ LineSegmentList)’
List	=	(‘LST’ ObjectList)’
MarkedRealPoint	=	(‘MRP’ Real Real Str (* x y mark *))’
Polygon	=	(‘POL’ PointList)’
PolyLayer	=	(‘PLA’ Real Real Vector (* z thickness objects *))’
PolyLayerVector	=	(‘PLV’ Real Vector (* dz objects *))’
PolyLine	=	(‘PLI’ PointList)’
PointVector	=	(‘PVC’ PointList)’
QuickDrawPicture	=	(‘QDP’ Real Real Real Real (* left bottom right top *))’

RealPoint	=	(' 'RPT'	Real Real (* x y *))'
Rectangle	=	(' 'REC'	Real Real Real Real (* left bottom right top *))'
RectangleVector	=	(' 'RVC'	RectangleList)'
SimpleUndirectedGraph	=	(' 'SUG'	Vector HomogenVector (* vertices edges *))'
SpanningTree	=	(' 'SPT'	Vector HomogenVector (* vertices edges *))'
Spline2	=	(' 'SP2'	RealPoint RealPoint)'
Splinegon	=	(' 'SPL'	ObjectList)'
Str	=	(' 'STR'	String)'
StraightEdge	=	(' 'SED'	RealPoint RealPoint)'
UndirectedGraph	=	(' 'UDG'	Vector HomogenVector (* vertices edges *))'
Vector	=	(' 'VEC'	ObjectList)'
VoronoiEdge	=	(' 'VED'	Integer Point Point [Point] [Point] (* edgeType p1 p2 v1 v2 *))'
VoronoiDiagram	=	(' 'VDG'	PointVector List List (* points edges neighbors *))'
CoordinateList	=	Integer { Integer } (* dimension coordinates *)		
DDimPointList	=	Integer { DDimPoint } (* length elements *)		
HomogenObjectList	=	Integer { Object } (* length elements *)		
LineSegmentList	=	Integer { LineSegment } (* length elements *)		
ObjectList	=	Integer { Object } (* length elements *)		
Point	=	RealPoint FloatPoint IntegerPoint		
PointList	=	Integer ({ RealPoint } { FloatPoint } { IntegerPoint }) (* length elements *)		
RectangleList	=	Integer { Rectangle } (* length elements *)		
Object	=	Circle ConvexPolygon ColorQuickDrawPicture DDimCircle DDimPoint DDimPointVector DirectedGraph FloatPoint GraphEdge HomogenVector Int IntegerPoint Layer LayerVector LineSegment LineSegmentVector List MarkedRealPoint Polygon PolyLayer PolyLayerVector PolyLine PointVector QuickDrawPicture RealPoint Rectangle RectangleVector SimpleUndirectedGraph SpanningTree Spline2 Splinegon Str StraightEdge UndirectedGraph Vector VoronoiEdge VoronoiDiagram		

Appendix B: Changes to TransSkel V2.02 in GeoBench

The Version of TransSkel V2.02 [DB 89] used by GeoBench was extended as follows:

1. Hierarchical Menus

```
function SkelMenu
(theMenu: MenuHandle;
 pSelect, pClobber: ProcPtr;
 drawBar, hierarchical: boolean): boolean;
```

If '*hierarchical*' is false, the function behaves as in the original TransSkel. If '*hierarchical*' is true, the menu is inserted in the hierarchical portion of the menu list. These menus can be used as hierarchical or pop-up menus (the '*beforeID*' parameter to '*InsertMenu*' is -1, see Inside Macintosh, Vol. V, p. 236).

2. Delayed scrap operations

```
procedure SkelApple
(aboutTitle: str255;
 aboutProc, deskAccProc, resumeProc: ProcPtr);
```

The two new procedure parameters serve to control delayed execution of the standard *Cut* and *Copy* commands of the *Edit* menu. If the two new parameters are '**nil**', the behavior is as in the original version of TransSkel. The '*deskAccProc*' of the form

```
procedure myDeskAccOpenHandler;
```

is called whenever a desk accessory is called under the old finder. The internal scrapbook should be written to the external scrapbook in this case (In older Mac OS versions this procedure is also to inform when the user switched applications in multifinder).

The '*resumeProc*' of the form

```
procedure myResumeHandle (resume: boolean);
```

is called whenever another application is activated under System 7. The '*resume*' flag tells you whether your application has been resumed or suspended. The flag '*acceptSuspendResumeEvents*' of the 'SIZE' resource should be set in order to get the corresponding events from the system. See Inside Macintosh Vol. VI, p. 5-14 and 5-19 for more information.

The procedures '*SkelSetWindResume*' and '*SkelGetWindowResume*' have been removed because they were buggy and because the suspend / resume event is actually not window-related.

3. Window growing

```
function SkelWindow
(theWind: WindowPtr;
 pMouse, pKey, pUpdate, pGrow, pActivate,
```

```
pClose, pClobber, pIdle: ProcPtr;  
frontOnly: boolean): boolean;
```

The procedure ‘*pGrow*’ is called after the user has resized the window. The boolean parameter to the ‘*pUpdate*’ procedure has been removed. **Warning:** Do *not* use update procedures which still have this parameter! Annoying, unrelated crashes are the result. Instead, use procedures of the following style:

```
procedure myUpdateHandler;  
procedure myGrowHandler;
```

In the original version of TransSkel, the whole window was redrawn after resizing a window. Now, only the part which was not visible before sizing is redrawn. To have the whole window redrawn, just insert the statement ‘*InvalRect(thePort^.portRect)*’ in your grow handler.

References

- [Brü 91] A. Brüngger: Schichtenmodelle in der XYZ GeoBench, Diploma Thesis, ETH Zürich, February 1991.
- [DB 89] TransSkel version 2.02 – Transportable application skeleton, written by Paul DuBois, Wisconsin Regional Primate Research Center, 1220 Capital Court, Madison WI 53706 USA, e-mail: dubois@rhesus.primate.wisc.edu
- [NB 91] J. Nievergelt, P. Schorn, M. De Lorenzi, C. Ammann, A. Brüngger: XYZ: A project in experimental geometric computation, to appear, May 1991.
- [S 91a] P. Schorn: Robust Algorithms in a Program Library for Geometric Computation, ETH PhD Dissertation 9519, 1991.
- [S 91b] P. Schorn: The XYZ GeoBench: A programming environment for geometric algorithms, to appear, May 1991.