SCIENCE AND ENGINEERING RESEARCH COUNCIL
RUTHERFORD APPLETON LABORATORY

INFORMATICS DIVISION

SOFTWARE ENGINEERING GROUP NOTE 116

Dimensional Design

A Layout Algorithm for the Labelled Cuboid Model

Draft 2

Issued by
R.W.Witty and D.R.Gibson

14th April 1986

DISTRIBUTION: R W Witty
D R Gibson
M Bertran-Salvans
T Povey, DEC
C Evans, DEC
R&D/DD/DEC file

KEYWORDS: SEGN 116 Dimensional Design

History

1 TreeMeta description designed by Rob Witty, January 1986.

2 Major change to include explanatory diagrams, tables of variable name derivation, with some reworking of the original draft to reflect comments made by Rob Witty.

1. Introduction

This note is intended to document some ideas which arose during the course of technical meetings while drawing up the requirements specifications for the Dimensional Design project. The Requirements Specification may be found in SEG Note 111; several earlier drafts of the document are also available as SEG Notes.

2. The Layout of a Dimensional Design

A Dimensional Design may be represented as a tree such that each node in the tree has a number of branches; each subtree being a Dimensional Design in itself. Drawing the tree is straightforward if the Dimensional Design is limited to being a unary or binary tree, and simple rules about the placing of subtrees are enforced. For example, the first class of subtree may be drawn vertically down the page, and the second horizontally across the page. The representation of a ternary tree means drawing the third class of subtree into a further dimension. On a flat piece of paper or display screen there is clearly a problem in projecting a three dimensional object in only two dimensions; this difficulty can be overcome by using the diagonal direction to represent the third class of subtree.

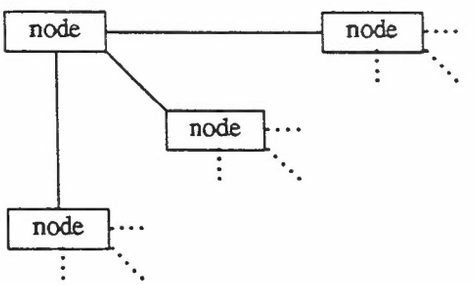

Figure 1: diagram showing how a ternary tree may be mapped onto a flat display surface, by projecting the third dimension as a diagonal.

A Dimensional Design need not be limited to having three subtrees, and in fact, there is no reason why a Dimensional Design should not have an arbitrary number of subtrees! This creates problems when attempting to draw the tree, unless some strategy is adopted which reduces the complex nature of an n-ary tree into something more manageable.

3. The Cuboid Model

In the Cuboid Model, a node which has more than three subtrees is broken down into a node having a series of groups of subtrees. Each of these groups contains three or less subtress. In order to draw the tree, the node is drawn, followed by the first group of three subtrees, using the vertical, horizontal and diagonal directions to represent each class of subtree. A box, or cuboid, is drawn so that it encloses the node and subtrees, and the next group of three subtrees is drawn around this cuboid, and so on. All of the branches are logically connected to the root node, although when drawn it may appear as though a subtree is connected to a cuboid rather than the node itself. A node which consists of several cuboids may be thought of as a stack of pieces of paper, where the size of the sheet of paper increases towards the bottom of the stack. Logically, each piece of paper shows the root node, but this is only visible on the uppermost sheet. The length of the lines connecting the logical node to each group of subtrees also increases towards the bottom of the stack, so that the subtrees of each level project out from under the piece of paper above.

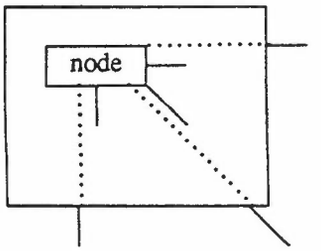

Figure 2 diagram showing how a node with six subtrees may be drawn using the cuboid model. More subtrees could be handled by adding more cuboids.

In order to maintain uniformity throughout the model, the contents of each node of a Dimensional Design may also be thought of as Dimensional Designs. This reduces the complexity of having to deal with the Dimensional Design as a whole. The contents of the nodes are treated in exactly the same way as the rest of the model, which would not be the case if the contents of a node were to be regarded as a string of text for example.

4. The Labelled Cuboid Model

During the course of the technical discussions, it was decided that it may be useful to associate a label with a cuboid, to provide a means of identifying it, or explaining how it fitted into the Dimensional Design. In order to allow a flexible system of labels, it was decided that a label could exist on any of the four sides of the cuboid. This idea soon extended to providing four possible labels for a cuboid, one on each side. To maintain uniformity once again, each label is also a Dimensional Design, and is attached to the root node of the cuboid. This means that a node may have seven, rather than just three, subtrees on each level of the stack, ie. within each cuboid.

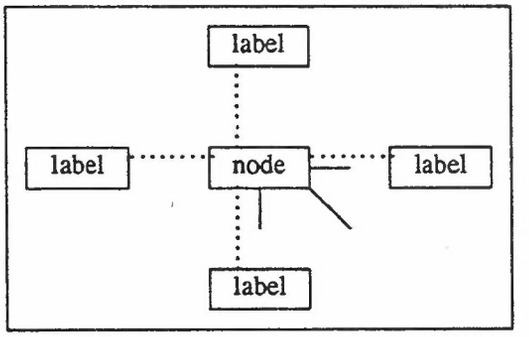

Figure 3: diagram illustrating that a cuboid with four labels contains seven and not three subtrees.

The drawing algorithm is intended to be as simple as possible. The basic scheme involves drawing the top label of the cuboid, and then the left label. The contents of the cuboid are then drawn, followed by the right and bottom labels. Since the model was chosen to be uniform, and each of the labels and the contents are themselves Dimensional Designs, it is possible to draw these by treating each as a new cuboid and recursing down the inner structures.

The algorithm is kept as general as possible, in order to leave scope for investigating various layout schemes, by parameterising as many parts of the drawing process as possible. For example, the positions of the labels relative to the root may be changed by providing a different set of parameters. These parameters will be explained in more detail later.

5. Investigation of the Model

In order to test the feasability of such a model, before considering it seriously in any higher level design stages, a means of testing it was required. A simple data format, with a regular syntax, was produced so that Designs which exhibited the features of this model could be constructed. In order to minimise time spent on building test systems, it was decided to make use of existing software.

5.1. The Use of TreeMeta

The syntax of the data format was described using BNF style rules. TreeMeta is a powerful compiler-compiler system which takes user-provided rules and produces a Pascal program which can then be compiled into a translator. This translator takes the raw input data, and transforms it according to the BNF rule set which the user supplied. As the data is processed by the translator, a tree is built in memory, and at suitable points, this tree can be traversed in order to produce the transformed style of data.

The translator which was produced takes the linear data format and builds a tree structure in memory which represents the Dimensional Design. Once the whole Design has been built up, the tree is processed according to the rules (see later), and a series of simple output primitives are produced which show how the Design may be drawn. These primitives are of the form

DrawCharacter(character, box)
DrawRectangle(box)
DrawVector(start point, end point)
DrawEnum(box)
DrawInduction(box)
DrawConstructor(box)
DrawConditional(box)

where *character* represents an ascii character enclosed in single quotes, *box* represents the set of four values for the top, left, bottom and right edges of the enclosing box, and *point* gives the x and y coordinates of a point.

In order to display the Dimensional Design in some form - it is particularly tedious to have to draw one by hand from the data - a Pascal program was written which draws the Design on a Perq screen by obeying the output primitives from the translator.

5.2. Glossary of Parameters and Variables

A large number of variables and parameters are used within the TreeMeta rules for processing the simple data format. So that the rules make some sense, they need some explanation. There is method in the naming scheme, even though it may look awkward. Where a variable name begins with three upper case characters, the first character denotes which part of an object the variable applies to, and the next two characters denote the object itself. Any variable which contains the word *gap* refers to a distance between two objects, and any which contains the word *len* refers to the length of an arc connecting two objects. The layout of the four labelled cuboid model is shown in figure 4, the next level of structure, i.e. the internals of the contents box, shown in figure 5. Tables 1 and 2 show how the variable names are derived.

The following variables are used as parametric constants within the translator, and are initialised using the input data:

CHHT and *CHWD* are the height and width of a single character within the Dimensional Design;

LLTgap is the horizontal distance between the left hand side of the cuboid and the left hand side of the top label;

TLLgap is the vertical distance between the bottom of the top label and the top of the left label;

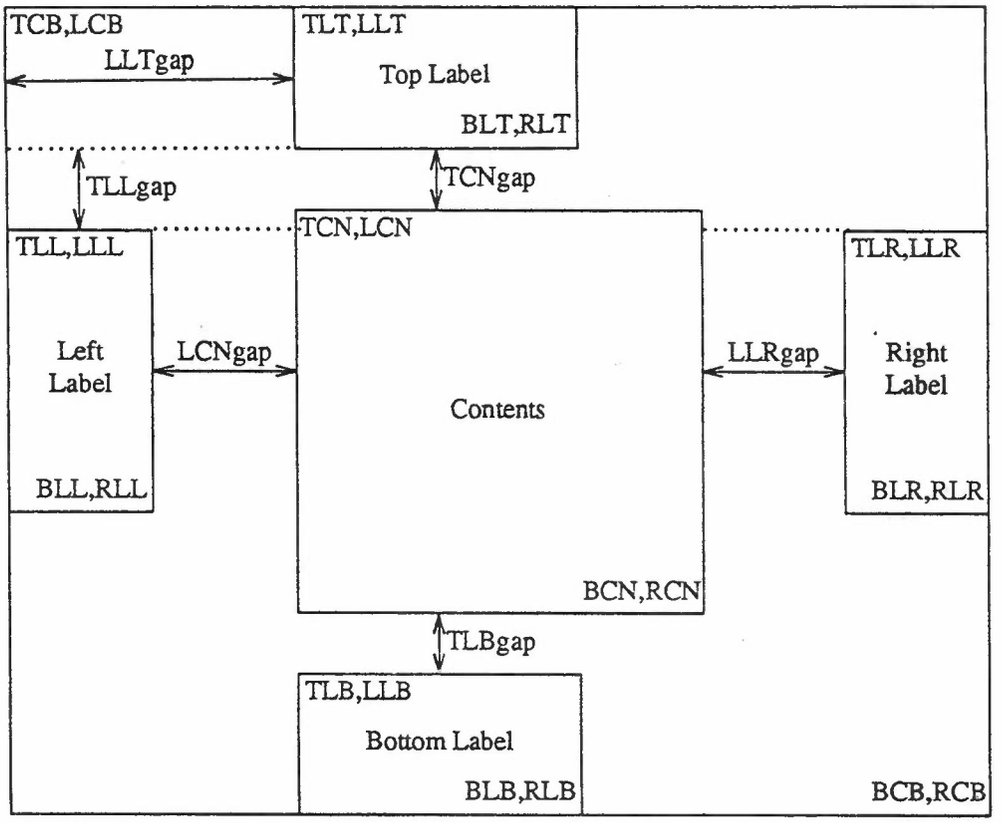

Figure 4: diagram showing the layout of a four labelled cuboid, and some of the parameters used by this particular drawing algorithm.

CHHT	=	CHaracter HeighT
CHWD	=	CHaracter WiDth
LLTgap	=	Left to Label Top gap
TLLgap	=	Top to Label Left gap
TCNgap	=	Top to CoNtents gap
LCNgap	=	Left to CoNtents gap
TLBgap	=	Top to Label Bottom gap
LLRgap	=	Left to Label Right gap
Vlen	=	Vertical length
Hlen	=	Horizontal length
Tgap	=	Top gap
Lgap	=	Left gap
ArcV	=	Arc Visibility
MaxB	=	Maximum Bottom
MaxR	=	Maximum Right

Table 1: showing the derivation of some of the parameter and variable names, as used within the translation rules.

TCNgap is the vertical distance between the bottom of the top label and the top of the contents box;

LCNgap is the horizontal distance between the right hand side of the left label and the left hand side of the contents box;

TLBgap is the vertical distance between the bottom of the contents box and the top of the bottom label;

TLT	=	Top	of Label Top	TLB	=	Top	of CuBoid
LLT	=	Left	of Label Top	LLB	=	Left	of CuBoid
BLT	=	Bottom	of Label Top	BLB	=	Bottom	of CuBoid
RLT	=	Right	of Label Top	RLB	=	Right	of CuBoid
TLL	=	Top	of Label Left	TLR	=	Top	of CoNtents
LLL	=	Left	of Label Left	LLR	=	Left	of CoNtents
BLL	=	Bottom	of Label Left	BLR	=	Bottom	of CoNtents
RLL	=	Right	of Label Left	RLR	=	Right	of CoNtents
TCB	=	Top	of Label Bottom	TRT	=	Top	of RooT
LCB	=	Left	of Label Bottom	LRT	=	Left	of RooT
BCB	=	Bottom	of Label Bottom	BRT	=	Bottom	of RooT
RCB	=	Right	of Label Bottom	RRT	=	Right	of RooT
TCN	=	Top	of Label Right	TST	=	Top	of SubTree
LCN	=	Left	of Label Right	LST	=	Left	of SubTree
BCN	=	Bottom	of Label Right	BST	=	Bottom	of SubTree
RCN	=	Right	of Label Right	RST	=	Right	of SubTree

Table 2: showing the derivation of the variable names for the coordinates of labels and other structures, as used within the translation rules.

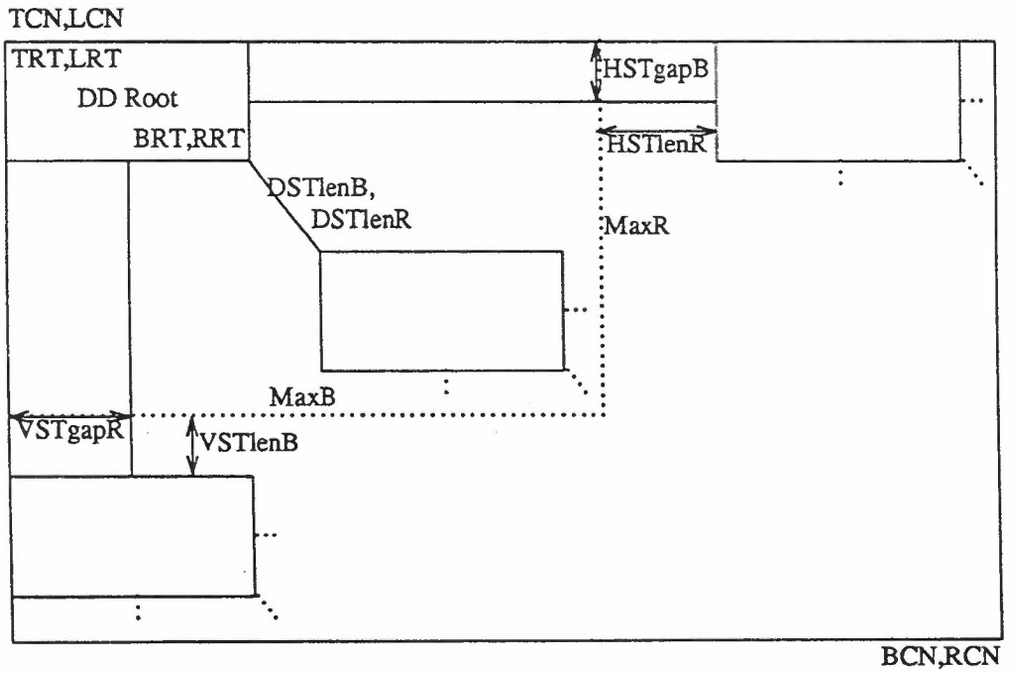

Figure 5: diagram showing the parameters used when drawing the Contents of a Cuboid (see fig 4).

LLRgap is the horizontal distance between the right hand side of the contents box and the left hand side of the right label.

VlenR#, *HlenR#*, *TgapR#* and *LgapR#*, where # represents a number from 1 to 12, go to make up a table of characteristics for drawing arcs (see later).

The rest of the variables are used within the main body of the translator. Several of the variables are used to pass values into and out of the various phases of the translator, in the same way as parameters are passed

to and from procedures. Most of the variables have to be stacked each time the drawing method recurses down the Design, and unstacked when that part of the tree has been completed.

T, L, B and *R* are the values for the top, left, bottom and right limits of the current Dimensional Design in question. The starting position of the overall design is defined by specifying the initial *T* and *L* in the parameter sequence. However, once translation has begun, these are used as local work variables, and provide a method of parameter passing within the translator.

TCB, LCB, BCB and *RCB* are the top, left, bottom and right limits of the current cuboid.

TLT, LLT, BLT and *RLT* are the top, left, bottom and right limits of the box containing the top label.

TLL, LLL, BLL and *RLL* are the top, left, bottom and right limits of the box enclosing the left label.

TLR, LLR, BLR and *RLR* are the top, left, bottom and right limits of the box containing the right label.

TLB, LLB, BLB and *RLB* are the top, left, bottom and right limits of the box enclosing the bottom label.

TCN, LCN, BCN and *RCN* are the top, left, bottom and right limits of the contents box of the current cuboid.

The following variables are used when drawing the structure within the contents box of the cuboid.

TRT, LRT, BRT and *RRT* are the limits of the box which encloses the root node of the Dimensional Design.

Vlen, Hlen, Tgap, Lgap: are the current arc characteristics, and are drawn from the table above by giving an index into the table. *Vlen* and *Hlen* are the vertical and horizontal dimensions of an arc, ie the height and width of an upright box enclosing the arc. *Tgap* and *Lgap* are vertical and horizontal displacements used to determine the starting position of an arc relative to the current position.

ArcV is the flag denoting whether the current arc is visible or invisible.

TST, LST, BST and *RST* are the top, left, bottom and right limits of the box which encloses the subtree of the root node currently being processed.

MaxB and *MaxR* give the bottom and right limits of the diagonal subtree.

M1 and *M2* are used as local variables within the MAXOF3 output rule.

In Figure 5, *DSTlenB* and *DSTlenR* are shown to illustrate the diagonal subtree parameters, *VSTlenB* and *VSTgapR* are illustrate those of the vertical subtree, and *HSTgapB* and *HSTlenR* for the horizontal subtree. These variables are a convenience only: they are not used within the TreeMeta rule system, as they are special cases which can be handled using *Vlen, Hlen, Tgap* and *Lgap*.

DSTlenB	=	Diagonal SubTree length to Bottom
DSTlenR	=	Diagonal SubTree length to Right
VSTlenB	=	Vertical SubTree length to Bottom
VSTgapR	=	Vertical SubTree gap to Right
HSTgapB	=	Horizontal SubTree gap to Bottom
HSTlenR	=	Horizontal SubTree length to Right

Table 3: showing the derivation of other (unused) variable names, as shown in figure 5.

5.3. The TreeMeta Rules for Data Parsing

```
.meta DimDes

" first of all, re-define the delimiters, so that
   open tm comment changes from double quote to open brace,
   close tm comment changes from double quote to close brace,
   string delimiter stays a single quote,
   inline code marker stays an at sign
"

.def delim(123,125,39,64)

" It isnt intuitive, but re-definition of default delimiters doesnt start
  until the rules are actually being parsed! Beware!
"

" Syntax Rules "

DimDes = Params
         Dd ?'Failed to match Dd '?
         : ROOT[2] * ;

{ Delimiters are re-defined now! }

{ Parameter Input }

Params = ParamA ParamB ParamC ParamD
         : PARAMS[4] ;


{ Characters }

ParamA = 'CHHT=' .num
         'CHWD=' .num
         : PARAMA[2] ;


{ Labels and Cuboids }

ParamB = 'LLTgap=' .num
         'TLLgap=' .num
         'TCNgap=' .num
         'LCNgap=' .num
         'TLBgap=' .num
         'LLRgap=' .num
         : PARAMB[6] ;


{ Sub-Tree Arc Length, Direction }

ParamC =          'Vlen' 'Hlen' 'Tgap' 'Lgap'
         'Rel-1'  .num   .num   .num   .num
         'Rel-2'  .num   .num   .num   .num
```

```
          'Rel-3'   .num    .num    .num    .num
          'Rel-4'   .num    .num    .num    .num
          'Rel-5'   .num    .num    .num    .num
          'Rel-6'   .num    .num    .num    .num
          'Rel-7'   .num    .num    .num    .num
          'Rel-8'   .num    .num    .num    .num
          'Rel-9'   .num    .num    .num    .num
          'Rel-10'  .num    .num    .num    .num
          'Rel-11'  .num    .num    .num    .num
          'Rel-12'  .num    .num    .num    .num
          : PARAMC[48] ;

ParamD = 'T=' .num
         'L=' .num
         : PARAMD[2] ;


{ Linear Form of Dimensional Design }

Dd = ( Atom / Cuboid )
     : DD[1] ;

Atom = ( SpecialSymbol / Ascii )
       : ATOM[1] ;

SpecialSymbol = '\S'
                ( '#' ^ 'Enum' /
                  '*' ^ 'Induction' /
                  '0' ^ 'Constructor' /
                  '?' ^ 'Conditional' ) ?'Failed to match SpecialSymbol '?
                : SPECIAL[1] ;

Ascii = '\A' .chr
        : ASCII[1] ;


{ Cuboid = Labels + (Root + Sub-Trees) }

Cuboid = BoxBeg
         LabTopp  ?'Failed to match LabTopp  '?
         LabLeft  ?'Failed to match LabLeft  '?
         Contents ?'Failed to match Contents '?
         LabRight ?'Failed to match LabRight '?
         LabBott  ?'Failed to match LabBott  '?
         BoxEnd   ?'Failed to match BoxEnd   '?
         : CUBOID[7] ;


BoxBeg = '['
         ( 'C' ^ 'Compressed' / .empty ^ 'Expanded' )
         ?'Failed to match BoxBeg '?
         : CUBEXP[1] ;

BoxEnd = ']'
```

```
        ( 'V' ^ 'Visible' / .empty ^ 'Invisible' )
        : CUBVIS[1] ;


LabTopp = ( 'T' Dd / .empty ^ 'nil' ) : LABEL[1] ;

LabLeft = ( 'L' Dd / .empty ^ 'nil' ) : LABEL[1] ;

LabBott = ( 'B' Dd / .empty ^ 'nil' ) : LABEL[1] ;

LabRight= ( 'R' Dd / .empty ^ 'nil' ) : LABEL[1] ;


Contents = Dd
           SubTrees ?'Failed to match SubTrees '?
           : CONTENTS[2] ;

SubTrees = Attribs
           ST1 ?'Failed to match ST1 '?
           ST2 ?'Failed to match ST2 '?
           ST3 ?'Failed to match ST3 '?
           : SUBTREES[4] ;

Attribs = Order
          RelNum ?'Failed to match RelNum '?
          ArcVis ?'Failed to match ArcVis '?
          : ATTRIBS[3] ;


Order = ( .'D1V2H3' /
          .'D1H2V3' /
          .'D1H3V2' /
          .'D1V3H2' /
          .empty    ^ 'D1V2H3' )
        : ORDER[1] ;


RelNum = (  '1-3'  ^  '1' ^  '2' ^  '3' /
            '4-6'  ^  '4' ^  '5' ^  '6' /
            '7-9'  ^  '7' ^  '8' ^  '9' /
           '10-12' ^ '10' ^ '11' ^ '12' /
           .empty  ^  '1' ^  '2' ^  '3' )
          : RELNUM[3] ;


ArcVis = ( 'DVIS' .num / .empty ^ '1' )
         ( 'VVIS' .num / .empty ^ '1' )
         ( 'HVIS' .num / .empty ^ '1' )
         : ARCVIS[3] ;


ST1 = ( '1' Dd / '#1' ^ 'nil' )
      : SUBTREE[1] ;
```

```
ST2 = ( '2' Dd / '#2' ^ 'nil' )
      : SUBTREE[1] ;

ST3 = ( '3' Dd / '#3' ^ 'nil' )
      : SUBTREE[1] ;

{ End of Syntax Rules }


{ Code Production Rules }

ROOT[-,-] =>
        *1 { Params }
        *2 { B,R <- EntireDD(T,L) }
        'Size of DD: ' %
        'Top    = ' < OUT[T] > %
        'Left   = ' < OUT[L] > %
        'Bottom= ' < OUT[B] > %
        'Right  = ' < OUT[R] > %
        ;


PARAMS[-,-,-,-] =>
        *1 *2 *3 *4 ;


PARAMA[-,-] =>
        < CHHT <- CONV[*1] >
        < CHWD <- CONV[*2] >
        .empty
        ;


PARAMB[-,-,-,-,-,-] =>
        { Labels }
        < LLTgap <- CONV[*1] >
        < TLLgap <- CONV[*2] >
        < TCNgap <- CONV[*3] >
        < LCNgap <- CONV[*4] >
        < TLBgap <- CONV[*5] >
        < LLRgap <- CONV[*6] >
       . .empty
        ;


PARAMC[-,-,-,-,-,-,-,-,-,-,-,-,-,
       -,-,-,-,-,-,-,-,-,-,-,-,-,
       -,-,-,-,-,-,-,-,-,-,-,-,-,
       -,-,-,-,-,-,-,-,-,-,-,-,-
      ] =>

        < VlenR1  <- CONV[*1]  ; HlenR1  <- CONV[*2]  ;
          TgapR1  <- CONV[*3]  ; LgapR1  <- CONV[*4]  >
```

```
        < VlenR2  <- CONV[*5]  ; HlenR2  <- CONV[*6]  ;
          TgapR2  <- CONV[*7]  ; LgapR2  <- CONV[*8]  >

        < VlenR3  <- CONV[*9]  ; HlenR3  <- CONV[*10] ;
          TgapR3  <- CONV[*11] ; LgapR3  <- CONV[*12] >

        < VlenR4  <- CONV[*13] ; HlenR4  <- CONV[*14] ;
          TgapR4  <- CONV[*15] ; LgapR4  <- CONV[*16] >

        < VlenR5  <- CONV[*17] ; HlenR5  <- CONV[*18] ;
          TgapR5  <- CONV[*19] ; LgapR5  <- CONV[*20] >

        < VlenR6  <- CONV[*21] ; HlenR6  <- CONV[*22] ;
          TgapR6  <- CONV[*23] ; LgapR6  <- CONV[*24] >

        < VlenR7  <- CONV[*25] ; HlenR7  <- CONV[*26] ;
          TgapR7  <- CONV[*27] ; LgapR7  <- CONV[*28] >

        < VlenR8  <- CONV[*29] ; HlenR8  <- CONV[*30] ;
          TgapR8  <- CONV[*31] ; LgapR8  <- CONV[*32] >

        < VlenR9  <- CONV[*33] ; HlenR9  <- CONV[*34] ;
          TgapR9  <- CONV[*35] ; LgapR9  <- CONV[*36] >

        < VlenR10 <- CONV[*37] ; HlenR10 <- CONV[*38] ;
          TgapR10 <- CONV[*39] ; LgapR10 <- CONV[*40] >

        < VlenR11 <- CONV[*41] ; HlenR11 <- CONV[*42] ;
          TgapR11 <- CONV[*43] ; LgapR11 <- CONV[*44] >

        < VlenR12 <- CONV[*45] ; HlenR12 <- CONV[*46] ;
          TgapR12 <- CONV[*47] ; LgapR12 <- CONV[*48] >
        .empty
        ;


PARAMD[-,-] =>
        { Starting Point }
        < T <- CONV[*1] ; L <- CONV[*2] >
        .empty
        ;

{ End of Parameters }


{ Draw Dimensional Design - Atom or Cuboid }

DD[-] =>
        *1 ;


ATOM    [ASCII[-]] =>
                < B <- T + CHHT ; R <- L + CHWD >
                'DrawCharacter('
```

```
            @39 *1:*1 @39 ','
            OUTTLBR[]

        [SPECIAL['Enum']] =>
            < B <- T + CHHT ; R <- L + CHWD >
            'DrawEnum('
            OUTTLBR[]

        [SPECIAL['Induction']] =>
            < B <- T + CHHT ; R <- L + CHWD >
            'DrawInduction('
            OUTTLBR[]

        [SPECIAL['Constructor']] =>
            < B <- T + CHHT ; R <- L + CHWD >
            'DrawConstructor('
            OUTTLBR[]

        [SPECIAL['Conditional']] =>
            < B <- T + CHHT ; R <- L + CHWD >
            'DrawConditional('
            OUTTLBR[]
        ;


OUTTLBR[] =>
        < OUT[T] > ','
        < OUT[L] > ','
        < OUT[B] > ','
        < OUT[R] > ');' %
        ;


CUBOID  [CUBEXP['Compressed'],-,-,-,-,-,-] =>
            { Do not draw Contents }
            < B <- T + CHHT ; R <- L + CHWD >
            'DrawRectangle('
            OUTTLBR[]

        [CUBEXP['Expanded'],-,-,-,-,-,-] =>
            { Draw Contents }
            STACKALL[] { to allow arithmetic to work when DD recurses }
            < TCB <- T ; LCB <- L > { input params }

            { Top Label }
            < TLT <- TCB ; LLT <- LCB + LLTgap >
            < T <- TLT ; L <- LLT ; B <- TCB ; R <- LCB >
                { ie B, R defaults if Label is nil }
            * 2 { DD or nil }
            < BLT <- B ; RLT <- R >

            { Left Label }
            < TLL <- BLT + TLLgap ; LLL <- LCB >
            < T <- TLL ; L <- LLL ; B <- BLT ; R <- LCB >
```

```
                * 3 { B, R <- DD(T,L) or nil }
                < BLL <- B ; RLL <- R >

                { Contents }
                < TCN <- BLT + TCNgap >
                < LCN <- RLL + LCNgap >
                * 4 { BCN, RCN <- Contents(TCN,LCN) }

                { Right Label }
                < TLR <- TLL > { Symmetry with left label }
                { LLR = Max(RCN,RLT)+LLRgap }
                        < PUSH[RCN] ; PUSH[RLT]; PUSH[0] >
                        MAXOF3[]
                        < LLR <- POP[0] + LLRgap >
                < T <- TLR ; L <- LLR >
                < B <- TCB ; R <- LLR > { was R <- LLR - LLRgap }
                * 5
                < BLR <- B ; RLR <- R >

                { Bottom Label }
                { TLB = Max(BLL,BCN,BLR)+TLBgap }
                        < PUSH[BLL] ; PUSH[BCN] ; PUSH[BLR] >
                        MAXOF3[]
                        < TLB <- POP[0] + TLBgap >
                < LLB <- LLT > { Symmetry with Top Label }
                < T <- TLB ; L <- LLB ; B <- TLB ; R <- LCB >
                * 6 { DD or nil }
                < BLB <- B ; RLB <- R >

                { Cuboid Overall size }
                < BCB <- BLB >
                { RCB <- Max(RLR,RLB) }
                        < PUSH[RLR] ; PUSH[RLB] ; PUSH[0] >
                        MAXOF3[]
                        < RCB <- POP[0] >

                { Cuboid Visibility - Draw a box, or not }
                * 7

                { Return Overall Size in output params }
                < T <- TCB ; L <- LCB > { restore to original value }
                < B <- BCB ; R <- RCB >
                UNSTACKALL[]
        ;


{ Label }
LABEL   ['nil'] =>
                .empty

        [-] =>
                *1 { B,R <- DD(T,L) }
        ;
```

```
{ Cuboid Visibility }
CUBVIS  ['Invisible'] =>
                .empty

        ['Visible'] =>
                'DrawRectangle('
                < OUT[TCB] > ','
                < OUT[LCB] > ','
                < OUT[BCB] > ','
                < OUT[RCB] > ');' %
        ;


CONTENTS[-,-] =>
        { Draw Root }
        < T <- TCN ; L <- LCN >
        *1 { B,R <- DD(T,L) }
        < BRT <- B ; RRT <- R >
        < MaxB <- BRT ; MaxR <- RRT > { Initialise Sub-Tree Extremities }

        { Draw Sub-Trees }
        *2 { MaxB, MaxR := ST(TCN,LCN,BRT,RRT,MaxB,MaxR) }
        < BCN <- MaxB ; RCN <- MaxR >
        ;


        { ATTRIBS(Order,RelNum,ArcVis),ST1,ST2,ST3 }
SUBTREES[ ATTRIBS[ORDER['D1V2H3'],RELNUM[-,-,-],ARCVIS[-,-,-]],-,-,-] =>
                DST[*1:*2:*1, *1:*3:*1, *2]
                VST[*1:*2:*2, *1:*3:*2, *3]
                HST[*1:*2:*3, *1:*3:*3, *4]

        [ ATTRIBS[ORDER['D1H2V3'],RELNUM[-,-,-],ARCVIS[-,-,-]],-,-,-] =>
                DST[*1:*2:*1, *1:*3:*1, *2]
                HST[*1:*2:*2, *1:*3:*2, *3]
                VST[*1:*2:*3, *1:*3:*3, *4]

        [ ATTRIBS[ORDER['D1V3H2'],RELNUM[-,-,-],ARCVIS[-,-,-]],-,-,-] =>
                DST[*1:*2:*1, *1:*3:*1, *2]
                HST[*1:*2:*2, *1:*3:*2, *4]
                VST[*1:*2:*3, *1:*3:*3, *3]

        [ ATTRIBS[ORDER['D1H3V2'],RELNUM[-,-,-],ARCVIS[-,-,-]],-,-,-] =>
                DST[*1:*2:*1, *1:*3:*1, *2]
                VST[*1:*2:*2, *1:*3:*2, *4]
                HST[*1:*2:*3, *1:*3:*3, *3]
        ;


{ Draw Diagonal Sub-Tree }
DST[-,-,-] =>
        PHYS[*1]  { New Arc Style Params }
        ARCV[*2]  { New Arc Visibility }
        *3        { Draw Subtree }
```

```
        ;


{ Draw Vertical Sub-Tree }
VST[-,-,-] =>
        PHYS[*1]
        ARCV[*2]
        *3
        ;


{ Draw Horizontal Sub-Tree }
HST[-,-,-] =>
        PHYS[*1]
        ARCV[*2]
        *3
        ;


PHYS    ['1'] =>
                < Vlen <- VlenR1 ; Hlen <- HlenR1 ;
                  Tgap <- TgapR1 ; Lgap <- LgapR1 >
                .empty
        ['2'] =>
                < Vlen <- VlenR2 ; Hlen <- HlenR2 ;
                  Tgap <- TgapR2 ; Lgap <- LgapR2 >
                .empty
        ['3'] =>
                < Vlen <- VlenR3 ; Hlen <- HlenR3 ;
                  Tgap <- TgapR3 ; Lgap <- LgapR3 >
                .empty

        ['4'] =>
                < Vlen <- VlenR4 ; Hlen <- HlenR4 ;
                  Tgap <- TgapR4 ; Lgap <- LgapR4 >
                .empty
        ['5'] =>
                < Vlen <- VlenR5 ; Hlen <- HlenR5 ;
                  Tgap <- TgapR5 ; Lgap <- LgapR5 >
                .empty
        ['6'] =>
                < Vlen <- VlenR6 ; Hlen <- HlenR6 ;
                  Tgap <- TgapR6 ; Lgap <- LgapR6 >
                .empty

        ['7'] =>
                < Vlen <- VlenR7 ; Hlen <- HlenR7 ;
                  Tgap <- TgapR7 ; Lgap <- LgapR7 >
                .empty
        ['8'] =>
                < Vlen <- VlenR8 ; Hlen <- HlenR8 ;
                  Tgap <- TgapR8 ; Lgap <- LgapR8 >
                .empty
        ['9'] =>
```

```
                < Vlen <- VlenR9 ; Hlen <- HlenR9 ;
                  Tgap <- TgapR9 ; Lgap <- LgapR9 >
                .empty

        ['10'] =>
                < Vlen <- VlenR10 ; Hlen <- HlenR10 ;
                  Tgap <- TgapR10 ; Lgap <- LgapR10 >
                .empty
        ['11'] =>
                < Vlen <- VlenR11 ; Hlen <- HlenR11 ;
                  Tgap <- TgapR11 ; Lgap <- LgapR11 >
                .empty
        ['12'] =>
                < Vlen <- VlenR12 ; Hlen <- HlenR12 ;
                  Tgap <- TgapR12 ; Lgap <- LgapR12 >
                .empty

        ;


ARCV['1'] =>
        < ArcV <- 1 > { visible }
        .empty
    ['0'] =>
        < ArcV <- 0 > { invisible }
        .empty
        ;


SUBTREE ['nil'] =>
                .empty

        [-] =>
                { Draw Arc to DD }
                { Draw Arc (TST,LST,BST,RST) according to direction of arc }
                        SETTST[]
                        SETLST[]
                        SETBST[]
                        SETRST[]
                        DRAWVECTOR[]

                { Set start of DD to end of Arc }
                        SETDDTST[]
                        SETDDLST[]

                { Draw DD }
                        < T <- TST ; L <- LST >
                        *1
                        < BST <- B ; RST <- R >

                { Overall Size of Subtree }
                        { MaxB = Max(MaxB,BST) }
                                < PUSH[MaxB] ; PUSH[BST] ; PUSH[0] >
                                MAXOF3[]
```

```
                        < MaxB <- POP[0] >
                    { MaxR = Max(MaxR,RST) }
                        < PUSH[MaxR] ; PUSH[RST] ; PUSH[0] >
                        MAXOF3[]
                        < MaxR <- POP[0] >
        ;


{ Initial start point of Sub-Tree Arc }

SETTST[] =>
        < Vlen # 0 > < TST <- BRT > / < TST <- TCN >
        ;

SETBST[] =>
        < Vlen # 0 > < BST <- MaxB > / < BST <- TCN >
        ;

SETLST[] =>
        < Hlen # 0 > < LST <- RRT > / < LST <- LCN >
        ;

SETRST[] =>
        < Hlen # 0 > < RST <- MaxR > / < RST <- LCN >
        ;


DRAWVECTOR[] =>
        < ArcV # 0 > { Visible Arc }
                'DrawVector('
                < OUT[TST + Tgap] > ','
                < OUT[LST + Lgap] > ','
                < OUT[BST + Tgap + Vlen] > ','
                < OUT[RST + Lgap + Hlen] > ');' %

        / .empty { Invisible Arc }
        ;


{ Set start of Sub-Tree's DD }

SETDDTST[] =>
        < Vlen # 0 > < TST <- BST + Vlen > / < TST <- TST >
        ;

SETDDLST[] =>
        < Hlen # 0 > < LST <- RST + Hlen > / < LST <- LST >
        ;


MAXOF3[] =>
        < M1 <- POP[0] ; M2 <- POP[0] >
        MAXM12[]
        < M1 <- POP[0] ; M2 <- POP[0] >
```

```
        MAXM12[]
        ;


MAXM12[] =>
        < M1 > M2 > < PUSH[M1] > / < PUSH[M2] >
        ;


STACKALL[] =>
        <
        PUSH[ TCB ] ; PUSH[ LCB ] ; PUSH[ BCB ] ; PUSH[ RCB ] ;
        PUSH[ TLT ] ; PUSH[ LLT ] ; PUSH[ BLT ] ; PUSH[ RLT ] ;
        PUSH[ TLL ] ; PUSH[ LLL ] ; PUSH[ BLL ] ; PUSH[ RLL ] ;
        PUSH[ TCN ] ; PUSH[ LCN ] ; PUSH[ BCN ] ; PUSH[ RCN ] ;
        PUSH[ TLB ] ; PUSH[ LLB ] ; PUSH[ BLB ] ; PUSH[ RLB ] ;
        PUSH[ TLR ] ; PUSH[ LLR ] ; PUSH[ BLR ] ; PUSH[ RLR ] ;
        PUSH[ TRT ] ; PUSH[ LRT ] ; PUSH[ BRT ] ; PUSH[ RRT ] ;
        PUSH[ TST ] ; PUSH[ LST ] ; PUSH[ BST ] ; PUSH[ RST ] ;
        PUSH[ Vlen ]; PUSH[ Hlen ]; PUSH[ Tgap ]; PUSH[ Lgap ];
        PUSH[ ArcV ]; PUSH[ MaxB ]; PUSH[ MaxR ]
        >
        .empty
        ;


UNSTACKALL[] =>
        <
        MaxR <- POP[0] ; MaxB <- POP[0] ; ArcV <- POP[0] ;
        Lgap <- POP[0] ; Tgap <- POP[0] ; Hlen <- POP[0] ; Vlen <- POP[0] ;
        RST  <- POP[0] ; BST  <- POP[0] ; LST  <- POP[0] ; TST  <- POP[0] ;
        RRT  <- POP[0] ; BRT  <- POP[0] ; LRT  <- POP[0] ; TRT  <- POP[0] ;
        RLR  <- POP[0] ; BLR  <- POP[0] ; LLR  <- POP[0] ; TLR  <- POP[0] ;
        RLB  <- POP[0] ; BLB  <- POP[0] ; LLB  <- POP[0] ; TLB  <- POP[0] ;
        RCN  <- POP[0] ; BCN  <- POP[0] ; LCN  <- POP[0] ; TCN  <- POP[0] ;
        RLL  <- POP[0] ; BLL  <- POP[0] ; LLL  <- POP[0] ; TLL  <- POP[0] ;
        RLT  <- POP[0] ; BLT  <- POP[0] ; LLT  <- POP[0] ; TLT  <- POP[0] ;
        RCB  <- POP[0] ; BCB  <- POP[0] ; LCB  <- POP[0] ; TCB  <- POP[0]
        >
        .empty
        ;

{ End of Code Production Rules }

.end
```

6. Examples of Test Data

The following is the parameter section of the various sets of test data, and is reproduced here to avoid unnecessary duplication in the examples.

```
CHHT= 17
CHWD= 13

LLTgap= 0
TLLgap= 0
TCNgap= 0
LCNgap= 0
TLBgap= 0
LLRgap= 0
```

	Vlen	Hlen	Tgap	Lgap
Rel-1	17	13	0	0
Rel-2	17	0	0	7
Rel-3	0	13	9	0
Rel-4	34	26	0	0
Rel-5	34	0	0	7
Rel-6	0	26	9	0
Rel-7	51	39	0	0
Rel-8	51	0	0	7
Rel-9	0	39	9	0
Rel-10	68	52	0	0
Rel-11	89	0	0	7
Rel-12	0	52	9	0

```
T= 10
L= 10
```

The character height and width are specific to the default font on the Perq, and correspond to the number of pixels. These sizes were obtained independently from of the rest of this work. The rest of the parameters are related to the character size.

Note: The diagrams that follow are sketches only; they are not intended to mirror the exact parameters used.

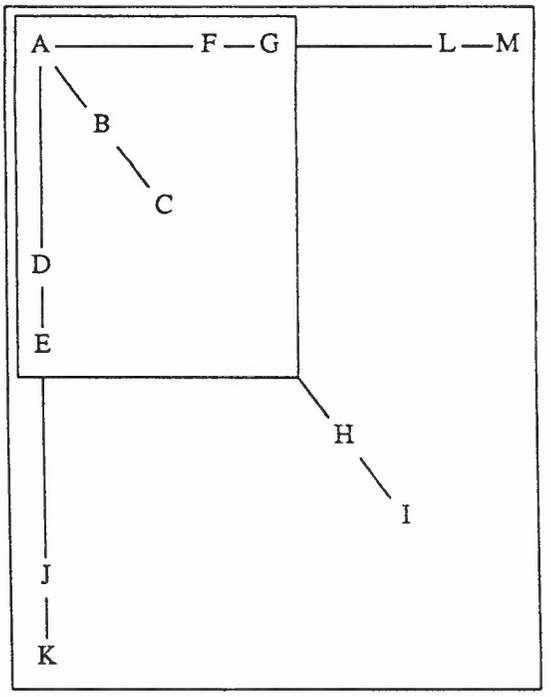

Figure 6: diagram showing the simple test case of a nested cuboid.

The Dimensional Design in figure 6 can be produced from the data (parameter section omitted):

```
[
 [ \AA 1 [ \AB 1 \AC
           #2
           #3
         ]
       2 [ \AD #1
           2 \AE
           #3
         ]
       3 [ \AF #1
           #2
           3 \AG
         ]
 ] V   1 [ \AH 1 \AI
           #2
           #3
         ]
       2 [ \AJ #1
           2 \AK
           #3
         ]
       3 [ \AL #1
           #2
           3 \AM
         ]
] V
```

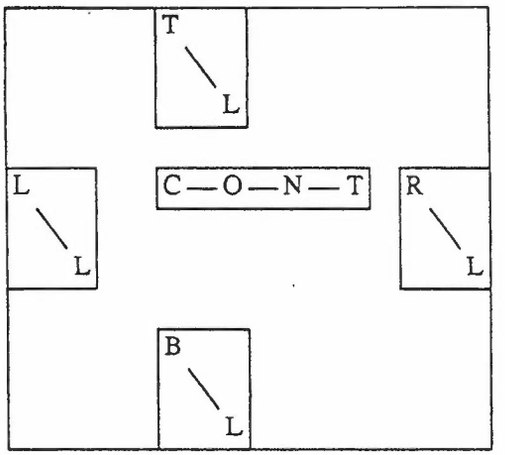

Figure 7: diagram showing a cuboid with simple labels.

The Dimensional Design shown in Figure 7 can be produced using the following data (parameter section omitted):

```
[
 T [ \AT 1 \AL #2 #3 ] V
 L [ \AL 1 \AL #2 #3 ] V
 [ \AC #1
       #2
       3 [ \AO #1
              #2
              3 [ \AN #1
                     #2
                     3 \AT
                ]
          ]
 ] #1 #2 #3
 R [ \AR 1 \AL #2 #3 ] V
 B [ \AB 1 \AL #2 #3 ] V
] V
```

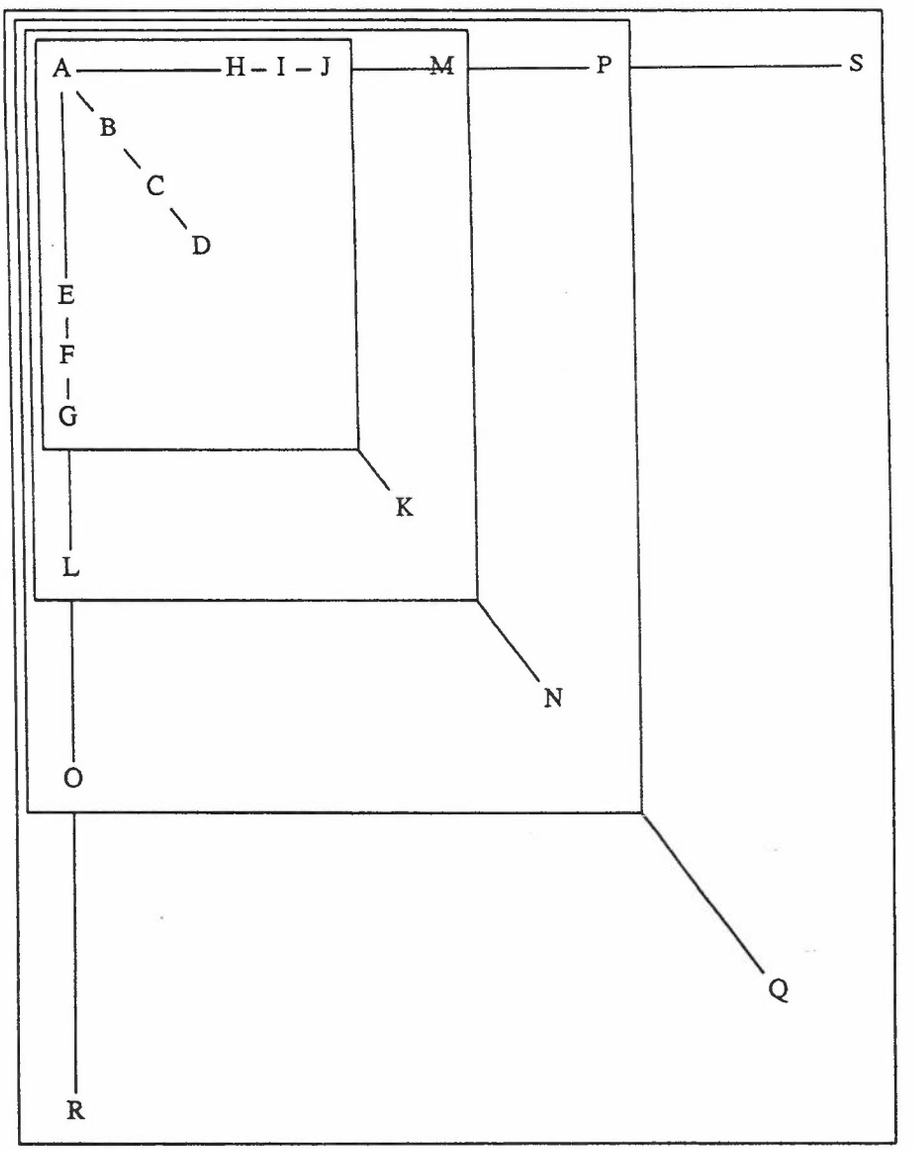

Figure 8: diagram showing nested cuboids, and selection of arcs from the parameter table.

The Dimensional Design shown in Figure 8 can be produced using the following data, but the exact arc dimensions depend on the parameter section (omitted, see earlier):

```
[
 [
  [
   [ \AA 1-3 1 [ \AB 1-3 1 [ \AC 1-3 1 \AD
                                      #2
                                      #3
                           ]
                         #2
                         #3
               ]
             2 [ \AE 1-3 #1
                         2 [ \AF 1-3 #1
                                     2 \AG
                                     #3
                           ]
                         #3
               ]
             3 [ \AH 1-3 #1
                         #2
                         3 [ \AI 1-3 #1
                                     #2
                                     3 \AJ
                           ]
               ]
    ] V 4-6 1 \AK
            2 \AL
            3 \AM
   ] V 7-9 1 \AN
           2 \AO
           3 \AP
  ] V 10-12 1 \AQ
            2 \AR
            3 \AS
] V
```

Appendix A: The Drawing Program on the Perq

The Pascal source that follows provides the bulk of the drawing program on the Perq. Several areas are Perq and PNX specific (PNX is the ICL version of UNIX† for the Perq), and in fact, this program will only run within the Window Management System, and not on the raw display. Several types have to be declared in order to provide the structures which the PNX system calls expect as parameters. ICL Pascal provides an "include" mechanism: the PNX specific types are declared in the file "PNXtypes.h". The formal declarations of the PNX specific system calls are held in "PNXproc.pas", as are the bodies of procedures which are used to hide the use of the system calls and types. The following are declared:

procedure getwindowsize(var w, h : integer); this routine returns the width and height of the window in pixels. The Window Management System must be active!

procedure gettextsize(var w, h : integer); this routine returns the width and height of a character in the default font, assuming a constant width font.

procedure clearwindow; this routine clears the contents of the window.

procedure drawline(xa, ya, xb, yb : integer); this draws a line from the point given by (xa,ya) to the point (xb,yb).

procedure drawtext(x, y : integer; s : packed array[lo..hi:integer] of char); causes the text in s to be output starting at the point (x,y).

Note: The Window Management System takes care of clipping any output to the window boundary.

† UNIX is a Trademark of Bell Laboratories.

```
program draw(input, output);

const
    MAXS = 20;  (* max number of characters in a string *)

type
    string = packed array[1..MAXS] of char;

    box =   record
            left, top, right, bottom : integer;
        end;

    (* in order to keep the structure simple, keep *)
    (* the PNX specific types in a seperate file   *)

#include "PNXtype.h"

var
    ww, wh : integer;  (* width and height of window in pixels *)
    cw, ch : integer;  (* width and height of character in pixels *)

    finished : boolean;

    top, left, bottom, right : integer;

    letter : char;

    wd: string;

    sq : box;

    (* in order to keep the structure simple, declare the PNX *)
    (* specific system calls and procedures in a seperate file*)

#include "PNXproc.pas"

    (* declarations of:
        procedure getwindowsize( var w, h : integer );
        procedure gettextsize( var w, h : integer );
        procedure clearwindow;
        procedure drawline( xa, ya, xb, yb : integer);
        procedure drawtext( x, y : integer;
                  s : packed array[lo..hi:integer] of char);
    *)

    (* the rest of the routines and program should be PNX independant *)

procedure boxbuild( var b : box; bl, bt, br, bb : integer);
begin
    b.left  := bl;
    b.top   := bt;
    b.right := br;
    b.bottom:= bb;
end;
```

```
procedure drawbox( b : box );
begin
    drawline(b.left,b.top,b.right,b.top);
    drawline(b.right,b.top,b.right,b.bottom);
    drawline(b.right,b.bottom,b.left,b.bottom);
    drawline(b.left,b.bottom,b.left,b.top);
end;

procedure charinbox( b : box; c : char);
var
    stmp : packed array[1..1] of char;
begin
    stmp[1] := c;
    drawtext(b.left+3,b.bottom-1,stmp);
end;

procedure getfourcharacters(var s : string);
var
    i : integer;
begin
    for i := 1 to MAXS do s[i] := ' ';

    for i := 1 to 4 do
    begin
        if eof or eoln then
        begin
            writeln('getfourcharacters: eof or eoln');
            exit(1);
        end
        else
        begin
            read(s[i]);
        end;
    end;
end;

procedure getcharsandbracket(var s : string);
var
    i : integer;
    f : boolean;
begin
    for i := 1 to MAXS do s[i] := ' ';

    i := 1;
    f := true;

    while f do
    begin
        if eof or eoln or (i = MAXS) then
        begin
            writeln('getcharsandbracket: eof or eoln or overflow');
            exit(1);
        end
        else
```

```
        begin
            read(s[i])
            if s[i] = '(' then f := false;
            i := i + 1;
        end;
    end;
end;

procedure getcharacter(var letter : char);
var
    i : integer;
    s : string;
begin
    for i := 1 to MAXS do s[i] := ' ';

    for i := 1 to 4 do      (* quote char quote comma *)
    begin
        if eof or eoln then
        begin
            writeln('getcharacter: eof or eoln');
            exit(1);
        end
        else
        begin
            read(s[i]);
        end;
    end;

    if (s[1] <> '''') or
       (s[3] <> '''') or
       (s[4] <> ',') then
    begin
        writeln('getcharacter: missing quotes or comma');
        exit(1);
    end;

    if not (s[2] in [' '..'~']) then
    begin
        writeln('getcharacter: non-printable ascii');
        exit(1);
    end;

    letter := s[2];
end;

procedure getfourintegers(var t, l, b, r : integer);
var
    i : integer;
    n : array[1..4] of integer;
    s : string;
begin
    for i := 1 to MAXS do s[i] := ' ';

    for i := 1 to 4 do
```

```
    begin
        if eof or eoln then
        begin
            writeln('getfourintegers: eof or eoln');
            exit(1);
        end;

        read(n[i]);

        if eof or eoln then
        begin
            writeln('getfourintegers: eof or eoln');
            exit(1);
        end;

        read(s[i]);
    end;

    if (s[1] <> ',') or
       (s[2] <> ',') or
       (s[3] <> ',') or
       (s[4] <> ')') then
    begin
        writeln('getfourintegers: missing commas or bracket');
        exit(1);
    end;

    t := n[1];
    l := n[2];
    b := n[3];
    r := n[4];
end;


begin
    getwindowsize(ww, wh);
    gettextsize(cw,ch);

    cw := cw + 4;       (* add 2 pixels either side of char *)
    ch := ch + 4;       (* add 2 pixels above and below *)

    clearwindow;

    finished := false;

    while not finished do
    begin
        getfourcharacters(wd);

        (* 'Case' statement on value of wd *)

        if wd = 'Size                    ' then
        begin
            finished := true;
```

```
end
else

if wd = 'Draw                    ' then
begin
    getcharsandbracket(wd);

    (* Another 'Case' statement on value of wd *)

    if wd = 'Character(              ' then
    begin
        getcharacter(letter);
        getfourintegers(top,left,bottom,right);

        boxbuild(sq,left,top,right,bottom);
        charinbox(sq,letter);
    end
    else

    if wd = 'Enum(                   ' then
    begin
        getfourintegers(top,left,bottom,right);

        boxbuild(sq,left,top,right,bottom);
        charinbox(sq,'#');
    end
    else

    if wd = 'Induction(              ' then
    begin
        getfourintegers(top,left,bottom,right);

        boxbuild(sq,left,top,right,bottom);
        charinbox(sq,'*');
    end
    else

    if wd = 'Constructor(            ' then
    begin
        getfourintegers(top,left,bottom,right);

        boxbuild(sq,left,top,right,bottom);
        charinbox(sq,'0');
    end
    else

    if wd = 'Conditional(            ' then
    begin
        getfourintegers(top,left,bottom,right);

        boxbuild(sq,left,top,right,bottom);
        charinbox(sq,'?');
    end
    else
```

```
            if wd = 'Rectangle(           ' then
            begin
                getfourintegers(top,left,bottom,right);

                boxbuild(sq,left,top,right,bottom);
                drawbox(sq);
            end
            else

            if wd = 'Vector(              ' then
            begin
                getfourintegers(top,left,bottom,right);

                drawline(left,top,right,bottom);
            end
            else

            (* default action *)
            begin
                writeln('main loop: unknown Draw...');
                exit(1);
            end;

            (* should really check for closing semi-colon here *)

            readln;
        end
        else

        (* default action *)
        begin
            writeln('main loop: not Size... or Draw...');
            exit(1);
        end;
    end;
end.
```