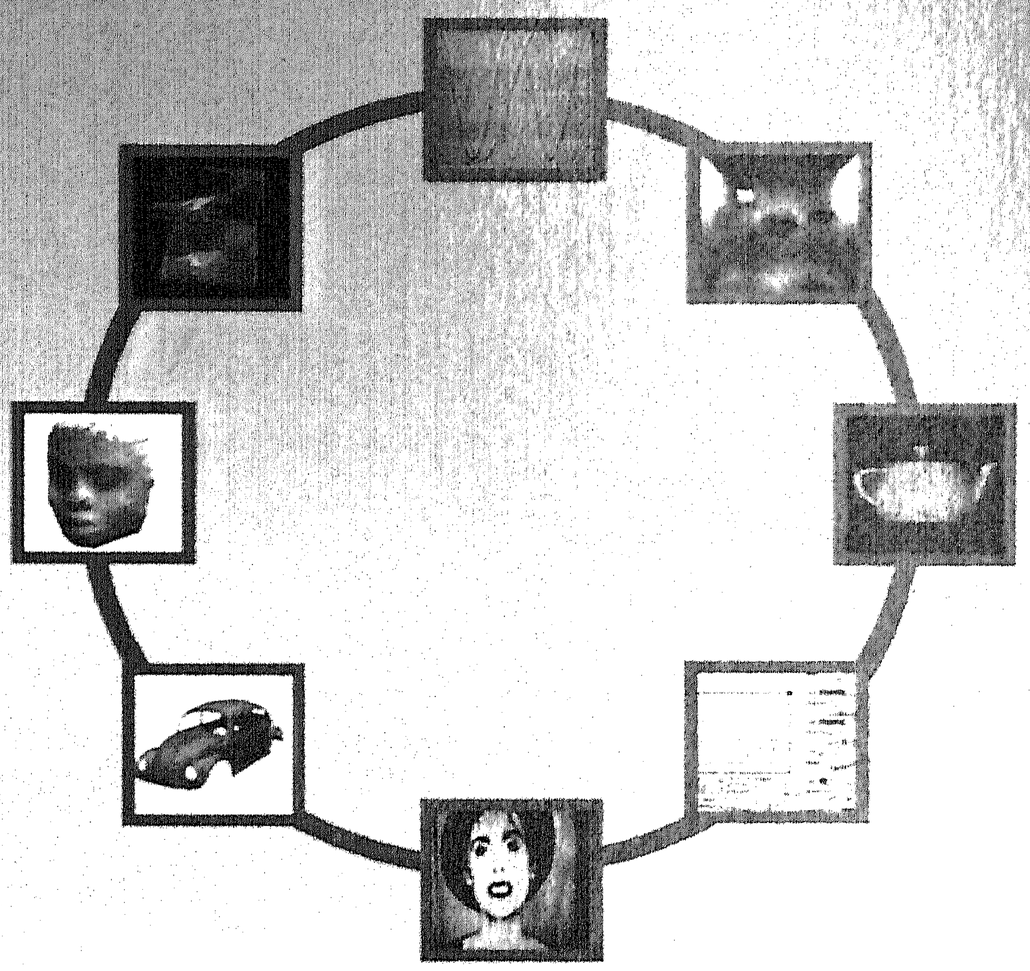
Interactive Systems
1985 – 1996



Interactive Systems
1985-1996

Edited by Robert van Liere

Preface

This book is a compilation of papers by members of the Department of Interactive Systems at the Centre for Mathematics and Computer Science in Amsterdam. Interactive Systems was initiated by Paul ten Hagen in April 1985, who remained the head of the department until December 1996. During these years, the department was active in computer graphics research, in building and evaluating experimental systems, and in various standardization efforts. This collection of articles is a tribute to all those who took part in these activities.
Throughout the years *Presentation, Interaction, and Systems* were themes that have dominated the research activities. The papers in this collection are organized into three parts:

- Part 1 is on presentation. The activities around GKS laid the foundations for the work which later led to research on realistic rendering and on techniques which incorporated lip-synchronized animation techniques. GKS is an example of the active role the department has taken in international standardization activities.

- Part 2 is on interaction. Research on formal specifications of interaction models led to work on higher level interaction models. Notable is the work done on constraint programming, logic programming and computational steering.

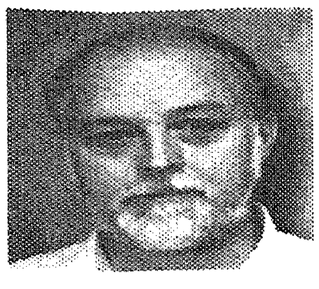
- Part 3 is on building experimental software and hardware systems. It includes papers on IIICAD, an intelligent CAD system, and on Manifolds, a specification language for parallel programming. The MADE system is a programming environment for multimedia applications. Finally, a paper on a hardware prototype, the dataflow machine, is given.

All papers have previously been published in journals, or conference or workshop proceedings. Most papers have been formatted differently, but remained essentially as they were originally written. I am grateful to all those colleagues who have provided the materials in a timely manner. Special thanks to Ivan Herman

who took the time to proofread many papers and Bèhr de Ruiter who helped me in my never ending struggle with UNIX formatting packages.

Robert van Liere
January 1997

Department Members



Paul ten Hagen

Nikola Aguirre, Varol Akman, Krzysztof Apt, Farhad Arbab, Miente Bakker, Richard van Bavel, Peter Bernus, Hanko Bicknese, Edwin Blake, Kees Blom, Peter Booyen, Pascal Bouvry, Freek Burger, Mieke Bruné, Iñaki Diaz de Etura, Marisa de Diego, Markus van Dijk, Valentijn van Dijk, Vincent Disselkoen, Stijn van Dongen, Kamran Eftekhari Shahroudi, Jan van Eijck, Wim Eshuis, Kees Everaars, Patricia Griffin, Annius Groenink, Michael Guravage, Michal Haindl, Frans Heeman, Stelios Haritakis, Jan Harkes, Marja Hegt, Ivan Herman, Hans van Hintum, Robert-Jan Honing, Hans Hoogendoorn, Jaap Kaandorp, Richard Kelleners, Liesbeth van Klarenbosch, Fons Kuijk, Wim de Leeuw, Robert van Liere, Patrick Marais, Elena Marchiori, Simone Marzola, Ingmar Maurice, Monique Megens, Eric Montfroy, Jurriaan Mulder, Hans de Nivelle, Han Noot, Daan Otten, George Papadopoulos, Ravic Pieters Kwiers, Jeroen van de Poll, Femke van Raamsdonk, Graham Reynolds, Tanja van Rij, Jan Rogier, Bert Rouwhorst, Bèhr de Ruiter, Eric Rutten, Zsofia Ruttkay, Vassilis Sakas, Henk Schouten, Anco Smit, Dirk Soede, Per Spilling, Tapio Takala, Sylvie Thiébaux, Tetsuo Tomiyama, Toos Trienekens, Anne Troelstra, Loes Vasmel-Kaarsemaker, Paul Veerkamp, Jantien van der Vegt, Michael in 't Veld, Remco Veltkamp, Dejuan Wang, Eric Weijers, Jack van Wijk, Eric Willemsen, Charles Wütrich, Yasushi Yamaguchi

Contents

Presentation

Interaction

Systems

Bibliography

[1] P.J.W. ten Hagen and M.M. de Ruiter. Segment Grouping, an Extension to the Graphical Kernel System. Technical Report CS-R8623, CWI, 1986.

[2] P.J.W. ten Hagen and H. Schouten. Parallel Graphical Output from Dialogue Cells. In G. Maréchal, editor, *Proceedings Eurographics '87*, pages 101–111. North-Holland, 1987.

[3] A.A.M. Kuijk and E.H. Blake. Faster Phong Shading via Angular Interpolation. *Computer Graphics Forum*, 8(4):315–324, 1989.

[4] R. van Liere. Divide and Conquer Radiosity. In P. Brunet and F.W. Jansen, editors, *Photorealistic Rendering in Computer Graphics*, pages 191–197. Springer Verlag, 1992.

[5] P. Griffin and H. Noot. The FERSA Project for Lip-Sync Animation. In *Proceedings of IMAGE'COM 93*, pages 111–120, 1993.

[6] D.A. Duce, P.J.W. ten Hagen, and R. van Liere. Components, Frameworks and GKS Input. In W. Hansmann, F.R.A. Hopgood, and W. Straßer, editors, *Proceedings Eurographics '89*, pages 87–103. North-Holland, 1989.

[7] R.C. Veltkamp and E.H. Blake. Event-based.constraints: coordinate.satisfaction → object.state. In Peter Wißkirchen, editor, *Object-Oriented and Mixed Programming Paradigms*, pages 159–169. Springer-Verlag, 1996.

[8] K.R. Apt. Arrays, bounded quantification and iteration in logic and constraint logic programming. *Science of Computer Programming*, 26:133–148, 1996.

[9] J. van Eijck. Presuppositions and information updating. In H. de Swart M. Kanazawa, C. Piñon, editor, *Quantifiers, Deduction, and Context*, pages 87–110. CSLI, Stanford, 1996.

[10] J.D. Mulder and J.J. van Wijk. 3D Computational Steering with Parameterized Geometric Objects. In G.M. Nielson and D. Silver, editors, *Proceedings IEEE Visualization '95*, pages 304–312. IEEE CS press, 1995.

[11] B. Veth. An Integrated Data Description Language for Coding Design Knowledge. In P.J.W. ten Hagen and T. Tomiyama, editors, *Intelligent CAD Systems I*, pages 295–313. Springer Verlag, 1987.

[12] I. Herman, G.J. Reynolds, and J. Davy. MADE: A Multimedia Application Development Environment. In L.A. Belady, S.M. Stevens, and R. Steinmetz, editors, *Proceedings IEEE Conference on Multimedia Computing and Systems*. IEEE CS Press, 1994.

[13] F. Arbab, C.L. Blom, F.J. Burger, and C.T.H. Everaars. Reusable Coordinator Modules for Massively Concurrent Applications. In Luc Bougé, Pierre Fraigniaud, Anne Mignotte, and Yves Robert, editors, *Proceedings of Euro-Par'96*, pages 664–677. Springer Verlag, 1996.

[14] I. Herman P.J.W. ten Hagen and J.R.G. de Vries. A Dataflow Graphics Workstation. *Computers and Graphics*, 14(1):83–93, 1990.

Segment Grouping,

an Extension to the Graphical Kernel System

P.J.W. ten Hagen, M.M. de Ruiter

Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

This paper introduces an extension to GKS, the ISO standard for 2D graphics software. Segment Grouping provides a device independent window manager functionality for application programmers. Segment Grouping allows for more efficient (given certain hardware restrictions) clipping and transformation of segments on both low and high function workstations.

1980 Math. Subject Classification : 69K32, 69K34, 69K30
Key Words & Phrases : Computer graphics, graphics systems, standardization, window management

1. Introduction

After several years of experience with GKS[1] implementations and use it is worth noticing that very few changes to GKS or extensions have been proposed. This is evidence for the fact that the generality of GKS surpasses most, if not all, other 2D packages. However two areas can be distinguished where applications fairly consistently try to add functionality which ought to be part of todays general purpose 2D graphics packages.

The first is *screen window management*.[2] This is the ability to divide the screen in rectangular areas and associate different data sets (e.g. text, pictures) with each of these windows. For alpha numeric screens, especially when they are equipped with so-called rasterop hardware facilities these window managers have been a tremendous success. For graphics devices this success has been relatively modest, mainly because of the low performance of raster devices to (re)draw pictures.

The second area is the possibilities for so-called *selective updates*. This is the possibility to minimize the number of graphical elements that need to be redrawn in order to visualize a picture change. (The not redrawn elements remain on the screen unchanged). This need is felt equally strong for high- and low performance devices, because in both cases unnecessary updates degrade performance well below expectations.

Screen window management can, among other things, provide a solution to the selective update problem. The functionality proposed in this report is termed *segment grouping*. It provides a window manager which quite naturally combines with GKS and yet supports all the facilities expected from window managers in as far as they are application program controlled (as opposed to operator controlled). Segment grouping also provides the basis for selective updates. This possibility can cause a dramatic performance improvement, especially for low cost hardware such as PC-graphics. In the next two sections the window management support and selective updates support will be explained.

Once segment grouping is introduced it is not difficult to discover many other useful functions that can be more easily supported. These however, will be discussed in a subsequent paper. This report only introduces the new concepts. The format is such that it can be seen as (another) appendix to GKS. This makes it easier to understand both in terms of added functionality and in terms of how to implement it. It may however be difficult to be understood for people who are not familiar with GKS.

Segment grouping can be implemented on any kind of workstation. It fully complies with the concept of device independence of GKS. It extends device independence by providing a device independent window manager which can be implemented using special window manager hardware support if available. The selective update facility is provided by allowing the GKS run-time system to restrict the updates to the windows that contain altered pictures.

Report CS-R8623
Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

1.1. Window management

A window manager deals with screen windows on three different levels. Two of these levels are by definition outside the scope of GKS. They are the virtual terminal level and the virtual workstation level. Terminal properties and workstation properties other than workstation dependent attributes cannot be controlled by GKS. As a result all window management functions which allow an operator to open a "new screen window" and start an application program (e.g. a GKS application) in this window are transparent to GKS. If GKS will be opened it can open one or several virtual workstations each of which will be confined to this operator selected virtual terminal screen. The window manager will have to map the virtual workstation on the appropriate portion of the actual screen, even if this portion is changed in size, position or priorities through operator manipulations. The operator has the possibility to temporarily make a virtual workstation have (almost) full screen size.
The third level of window management, being the control of the virtual workstation, is given completely to the segment grouping extension. The basic mechanism for window control is obtained by extending the set of normalization transformations by a set of so-called group normalization transformations which have a fixed viewport (in NDC). These viewports are mapped onto the screen (by the workstation transformation, as usual) to determine the lowest level of screen windows, which are not under operator control (other than by manipulating the enclosing virtual workstation viewport).
By distributing the segments over those windows (i.e. by putting them in the corresponding "group") it is possible to assure the disjointness of such segment groups. Disjointness of groups results from either group windows being non-overlapping or by a shielding capacity given to each group. In order to ensure that group segments will be confined to this window, the segments share the group normalization transformation and clip (i.e. it is global to all segments in the group and cannot be changed for those segments). However individual segment transformations remain without restrictions. In addition there is a group transformation for all the group segments which is predefined to all group segment transformations. This group transformation can be controlled by both application program and operator. In this way the group window can actually be viewed as a "window" on the group segments.

1.2. The update function

The semantics of the update function are refined for grouping by maintaining an update status per group. This means that as soon as an update is necessary only those groups that need to be updated actually will be.
There is no separate workstation independent group store. The WISS is sufficient for supporting groups as well. The collection of segments in a WISS is available for including them individually in a group. Segments contained in a group at the WISS cannot be taken out of the group through the function COPY SEGMENT TO WORKSTATION or ASSOCIATE SEGMENT WITH WORKSTATION. Instead of those there now exists COPY GROUP TO WORKSTATION and ASSOCIATE GROUP WITH WORKSTATION. However, INSERT SEGMENT can use segments from groups as well as segments outside groups.
Segment groups do not introduce any overhead for the workstation independent part of the system. They allow for a much more efficient implementation of segment handling on a per workstation basis. The updates necessary during input (e.g. providing real-time feedback) can be executed much faster when groups are used. The response time improvements are so rewarding that it seems to be appropriate to introduce a whole new class of prompt/echo typed based on group functions. This will also be further explored in the future report.

1.3. Groups for GKS-3D

The concept of segment grouping can also be added to GKS-3D. This will be discussed in a separate document.

2. CONCEPTS

2.1 *Group attributes*

A group has the following attributes:

1) Static :
- Group name, which is synonymous with its normalization transformation number.
- clipping window, which is identical to the viewport of the associated normalization transformation.

2) Dynamic :

- group transformation:	matrix
- priority:	if parts of groups overlap, the group with the higher priority will be prefered when the groups are displayed
- visibility:	VISIBLE/INVISIBLE
- detectability:	UNDETECTABLE/ DETECTABLE
- shielding:	NOSHIELDING/SHIELDING
- shielding colour:	a colourindex indicating the shield colour, which is painted as background.
- highlighting:	NORMAL/HIGHLIGHTED
- panning enabled:	DISABLED/ENABLED
- zooming enabled:	DISABLED/ENABLED
- rotating enabled:	DISABLED/ENABLED

Associated with these attributes are a number of attribute setting functions. In order to describe their effects, first the extensions of the GKS- and workstation statelists will be explained.

GKS statelist:
- holds the current values of static group attributes.
- when a group is open, the clipping window, being an entry in the global statelist, cannot be changed. (this can be enforced by forbidding normalization transformation set functions for the current normalization transformation or a more liberal approach would allow for all functions concerning transformations to be used as long as they don't have the effect of changing the clipping window in the GKS statelist)

Workstation statelist:
- has a list of all groups associated with this workstation
- may use the knowledge about grouping, in particular whether they overlap, for minimizing redrawing segments after dynamic changes.

Group statelist:
- when a group is created it gets default values for the dynamic attributes, which can be adjusted immediately following an CREATE/OPEN GROUP call, for this particular group.
- a group name is created for the group, dependent on the name of the current normalization transformation
- an empty group is made visible by its shield painted over the group window area on the screen, coloured with the background colour. depending on the setting of the shielding attribute either the viewport is solid filled or only the boundary is drawn.
- the treatment of groups is very much like the treatment of segments, but one level higher in the hierarchy.
- their is no need for an insert group function. however, an associate group function is provided as an extension of WISS for groups.

Group shielding affects the visibility of primitives laying in the clipping window area associated with a group. If the shielding attribute of a group is set to SHIELDING then a solid rectangle, equal to the group viewport, is drawn before drawing the group primitives. Thus it is possible to shield off all

primitives laying in the viewport of a specific group, and not belonging to the group. The shielding attribute can be set by the function SET GROUP SHIELDING. If shielding is set to NOSHIELDING only the boundaries of the rectangle are drawn. The colour to be used for this shielding action is indicated by the 'current shielding colour' entry in the group statelist, and can be changed by SET GROUP SHIELDING COLOUR.
Panning, zooming and rotation of a group can be realized by changing the group transformation. The pan, zoom and rotating ALLOWANCE fields in the group statelist indicates whether panning, zooming or rotating is either enabled or disabled.

2.2. Groups and segments
A group can be open when there is at least one workstation active. There can only be one open group at a time. A group cannot be opened when there is a segment open.

Primitives are not directly stored in a group, but indirectly via the segments contained in a group. A group itself does not contain any primitive. This includes that primitives outside segments will not go into a group.

Upon creating a group the set of active workstations is stored in the group statelist, thus being the set of workstations associated with the group. A workstation can be deactivated when a group is open. However it is an error to deactivate the last active workstation. By allowing a workstation to be temporarily deactivated it can skip certain segments of the group. An activate/deactivate workstation action does not affect the list of associated workstations in the group statelist. However the 'set of associated workstation' of a group can be extended by an ASSOCIATE GROUP WITH WORKSTATION call.

A close workstation action causes the workstation identifier to be deleted from the set of associated workstations in the group statelist of each group associated with the workstation. If the set of associated workstations of a group becomes empty, the group is deleted.

Workstations activated after the group has been opened are not added to the set of associated workstations, and are thus not affected by any function concerning the currently open group.

Closing a group when a segment is open is an error condition.

A closed group may be reopened. Thus it is possible to selectively store segments in the group. Segments created outside groups are not contained in a group, and cannot be affected or manipulated by any function concerning segment grouping.

On workstations not supporting grouping, a segment inside a group is treated as if there was no group open.

2.3. Group and segment attributes
The segments of a group are constrained by the group attributes. The segment priority/detectability/visibility is only defined relative to the other segments of the group. With respect to segments outside the group or other groups, the segment attributes are overruled by the group's attributes. E.g. segment detectability is valid only when the group is detectable. Likewise, segment visibility is only valid if the group is visible, etc. Segment priority is only effective in relation to other segments in the same group. The priority of segments in different groups is defined by the priority of the groups to which they belong. Segments not belonging to any group have lower priority than all group segments.

2.4. Group input priorities
Each group is associated with a normalization transformation. The connection is made via the group name which is identical to the number of one of the available transformations. During the time a group is open the associated normalization transformation must be the current transformation. To prevent unexpected visual effects on a workstation it is not allowed to change window or viewport of a normalization transformation associated with any of the existing groups.
To enforce that the creation of a locator input event inside a group viewport will return the normalization transformation belonging to the group, viewports associated with groups automatically get higher viewport input priority then viewports not associated with any group.

2.5. Deferring picture changes

GKS allows a workstation to delay for some time the actions requested by the application program. During this period the state of the display is undefined. The concept of deferral pictures refers only to a limited set of functions, namely those generating output. This set of functions must be extended with the following functions.

ASSOCIATE GROUP WITH WORKSTATION
COPY GROUP TO WORKSTATION

Depending on the type of workstation, some GKS functions can be performed immediately, while other functions lead to an implicit regeneration. Some new entries in the workstation description table indicate which changes caused by group functions require an implicit regeneration to update the workstation.
In general an implicit regeneration is equivalent to an invocation of the functions REDRAW ALL SEGMENTS ON WORKSTATION, and an invocation REDRAW GROUP ON WORKSTATION for each group on this workstation.

An implicit regeneration may be either allowed or suppressed, depending on the GKS deferral state.
Besides these cases an implicit regeneration is also made necessary if any of the following occur:

1) If the 'dynamic modification accepted' entry in the workstation description table is IRG for group priority and this workstation supports group priority:

   i) if a segment containing primitives is added to a group with a higher group priority.

   ii) if the complete execution of one of the following actions would be affected by group priority.

   DELETE GROUP
   DELETE GROUP FROM WORKSTATION
   ASSOCIATE GROUP WITH WORKSTATION
   SET GROUP TRANSFORMATION
   SET GROUP VISIBILITY
   SET GROUP PRIORITY

2) if the dynamic modification entry in the workstation description table is IRG for group transformation:

   SET GROUP TRANSFORMATION

3) if the dynamic modification entry in the workstation description table is IRG for group visibility (visible -> invisible):

   SET GROUP VISIBILITY(INVISIBLE)

4) if the dynamic modification entry in the workstation description table is IRG for group visibility (invisible -> visible):

   SET GROUP VISIBILITY(VISIBLE)

5) if the dynamic modification entry in the workstation description table is IRG for highlighting:

   SET HIGHLIGHTING

6) if the dynamic modification entry in the workstation description table is IRG for delete group:

   DELETE GROUP
   DELETE GROUP FROM WORKSTATION

Deferred actions can be made visible by the use of UPDATE WORKSTATION.

If possible the implicit regeneration can be restricted to a subset of the execution of the REDRAW ALL SEGMENTS ON WORKSTATION function and all REDRAW GROUP ON WORKSTATION functions. In the ultimate case it can be reduced to one invocation of the function REDRAW GROUP ON WORKSTATION.

2.6. Clipping
Clipping takes place after the normalization transformation, segment transformation and group transformation have been applied.

At opening a group a static clipping window is bound to the group, being the clipping rectangle from the GKS statelist. During the time a group is open this clipping rectangle may not be changed. This is enforced by forbidding to change the clipping window of the GKS statelist via a change of the current normalization transformation viewport.
At reopening an already existing group it will be an error if the current clipping window of GKS is not the same as the clipping window of the reopened group.

Clipping rectangles are not changed by a group transformation.

GKS states that a clipping rectangle is associated with each primitive in a segment, being the clipping rectangle stored in the GKS statelist. Due to the current approach each primitive in a segment of a group, inserted by a GKS output function during the time the group was open, will have the same clipping window bound to it as the clipping window of the group. There is no way to insert primitives in an open segment with a different clipping window than the current GKS clipping window. This is enforced among others by not allowing a group to be open when calling ASSOCIATE SEGMENT WITH WORKSTATION.

2.7. Workstation Independent Segment and Group Storage
In GKS one workstation Independent Segment Storage (WISS) is defined. Besides segments outside groups this workstation may also contain groups. Groups stored on this workstation can be used for the COPY GROUP TO WORKSTATION or the ASSOCIATE GROUP WITH WORKSTATION functions. None of these functions modify the contents of the groups to which they are applied.

2.8. GKS operating states
GKS always exists in one of its predefined operating states. To enable the control over the allowance of the new grouping functions two new states are introduces.
After creation or opening a group GKS will be in the state GROP. Closing the open group brings the system back into the state WSAC. Opening a segment during the state GROP brings GKS in the state GOSO. Closing the segment moves GKS back to the state GROP.

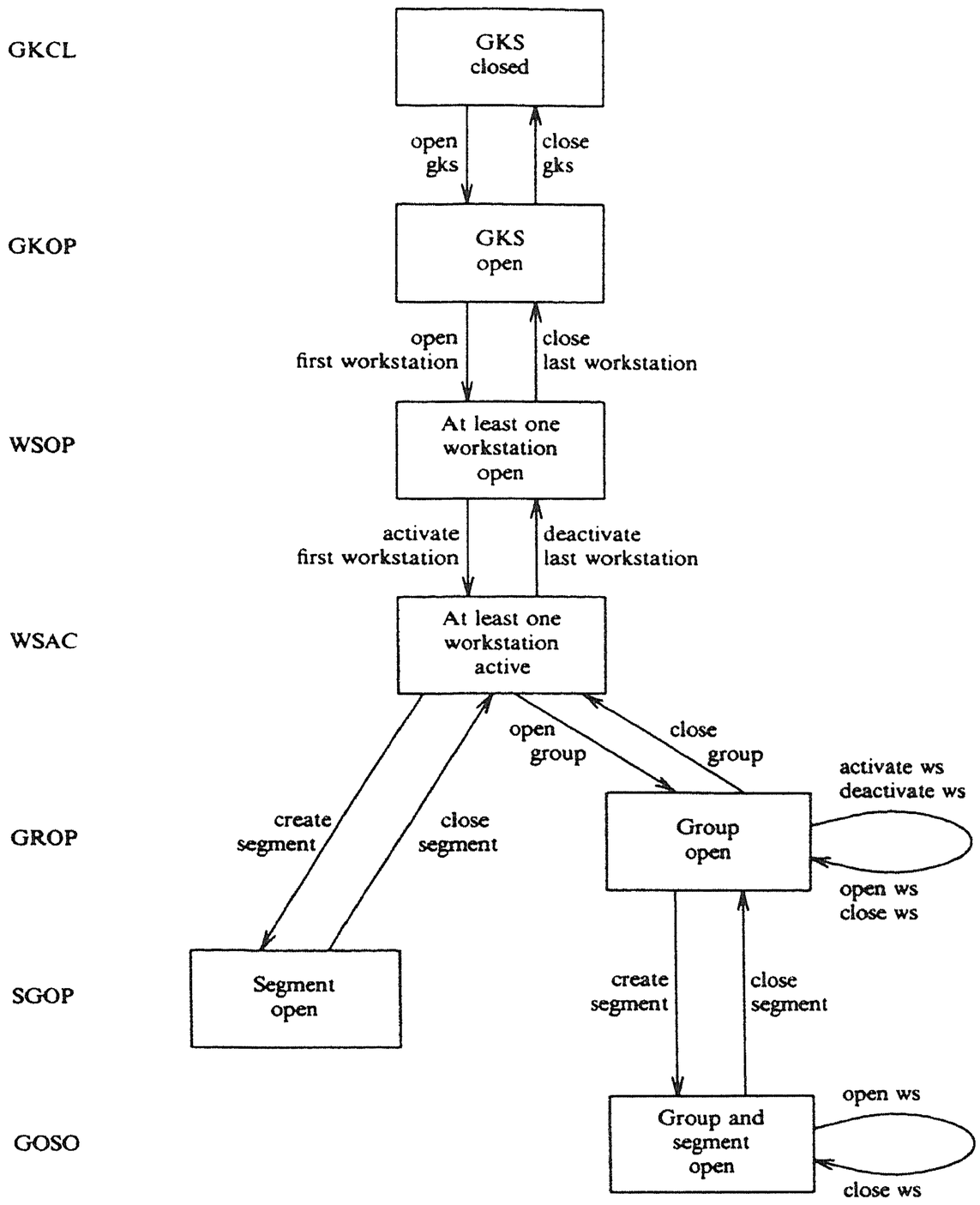


Figure 1. Some transitions between operating states

2.9. GKS 7.4 and Groups

As many standard GKS functions are also allowed in the state GROP and/or GOSO, the set of states in which each GKS function can be called must be redefined.

The functions ASSOCIATE SEGMENT WITH WORKSTATION and COPY SEGMENT TO WORKSTATION are restricted to segments outside groups.

As it is not allowed to change a normalization transformation connected to an existing group it is an error to call SET WINDOW or SET VIEWPORT for these transformations. The function SELECT CURRENT NORMALIZATION TRANSFORMATION may not be called when a group is open.

Additional information about restricted states and errorconditions is to be supplied for GKS 7.4 functions.

3. FUNCTIONS

3.1. Grouping functions

3.1.1. Grouping manipulation functions

CREATE GROUP WSAC L1a

Parameters:

Out group name N

Effect:
GKS is set into the state GROP if the current state is WSAC. The workstation statelist groupstatus entry of each active workstation, capable of handling groups, is set to GROUPOPEN. The group statelist is set up and initialized as follows:
The current normalization viewport is assigned to the viewport entry. The other entries are filled according to the default values:

visibility:	VISIBLE
shielding:	SHIELDING
shielding colour:	BACKGROUND COLOUR INDEX
highlighting:	NORMAL
detectability:	UNDETECTABLE
transformation:	IDENTITY

All subsequent output will be constrained (by clipping) to the group viewport.
A group name is created dependent on the name of the current normalization transformation.
The group name is recorded as 'the name of the open group' in the GKS state list. All subsequent segments until the next CLOSE GROUP will be collected into this group. The group name is entered into the 'set of stored groups for this workstation' in the workstation statelist of every active workstation able to handle groups. All active workstations able to handle groups are included in the 'set of associated workstations' of the group statelist of the opened group.
The group name is entered into the 'set of group names in use' in the GKS statelist. Primitive attributes are not effected.
Window and viewport of the associated normalization transformation are locked during the existence of the created group.
For further references to the created group the group name is returned to the caller of ths function.

Errors:

3	*GKS not in proper state: GKS shall be in state WSAC*
-831	*There exists already a group with group name equal to the current normalization transformation number.*
-836	*None of the active workstations able to support grouping*

OPEN GROUP WSAC L1a

Parameters:

In group name N

Effect:
GKS is set into the state GROP if the current state is WSAC. The workstation statelist entry of all active workstations, associated with the group, for the groupstatus is set to GROUPOPEN.
All subsequent output will be constrained (by clipping) to the group viewport.
The group name is recorded as 'the name of the open group' in the GKS state list. All subsequent output primitives until the next CLOSE GROUP will be collected into this group.
Primitive attributes are not effected.

Errors:

3	*GKS not in proper state: GKS shall be in state WSAC*
-830	*Specified group name is invalid*
-832	*Specified group name does not exist*
-837	*The normalization transformation associated with the specified group is not selected.*

CLOSE GROUP GROP L1a

Parameters:

none

Effect:
GKS is put into the operating state WSAC . Segments may no longer be added to the previously open group. The 'name of the open group' in the GKS statelist becomes unavailable for inquiry. The workstation statelist entry of all active workstations, associated with the group, for the groupstatus is set to NOGROUPOPEN.

Errors:
-800 *GKS not in proper state: GKS shall be in the state GROP*

DELETE GROUP WSOP, WSAC, SGOP, GROP, GOSO L1a

Parameters:

In group name		N

Effect:
The group is deleted. The group name is removed from each 'set of stored groups for this workstation' (in the workstation state lists) which contains it, and from the 'set of group names in use' in the GKS statelist. All segments in the group are deleted, i.e. an action equivalent to DELETE SEGMENT is performed for each segment in the group. The groups statelist is cancelled.

Errors:
-801 *GKS not in proper state: GKS shall be in state WSOP, WSAC, SGOP, GROP or GOSO*
-830 *Specified group name is invalid*
-832 *Specified group name does not exist*
-844 *Specified group is open*

DELETE GROUP FROM WORKSTATION WSOP, WSAC, SGOP, GROP, GOSO L1a

Parameters:

In workstation identifier	(1..n)	N
In group name		N

Effect:
The group is deleted from the specified workstation. The group name is removed from the 'set of stored groups for this workstation' in the workstation statelist. The workstation identifier is removed from the 'set of associated workstations' in the group statelist. If the 'set of associated workstations' becomes empty, the group is deleted, i.e. the DELETE GROUP function is performed.
All segments in the group are deleted at the workstation, that is, an action equivalent to DELETE SEGMENT FROM WORKSTATION is performed for each segment in the group.

Errors:
-801 *GKS not in proper state: GKS shall be in state WSOP, WSAC, SGOP, GROP or GOSO*
20 *Specified workstation is invalid*
25 *Specified workstation is not open*
33 *Specified workstation is of category MI*
35 *Specified workstation is of category INPUT*
-813 *Specified workstation does not support grouping*
-830 *Specified group name is invalid*
-833 *Specified group name does not exist on specified workstation*
-844 *Specified group is open*

CLEAR GROUP WSOP, WSAC, GROP L1a

Parameters:

In group name	N

Effect:
All segments in the group are deleted. The 'set of stored segments for this group' is set to empty.

Errors:

-806	*GKS not in proper state: GKS shall be in one of the states WSOP, WSAC or GROP*
-830	*Specified group name is invalid*
-832	*Specified group name does not exist*

REDRAW GROUP ON WORKSTATION WSOP, WSAC, GROP, SGOP, GOSO L1a

Parameters:

In workstation identifier	N
In group name	N

Effect:
The following actions are executed in the given sequence:

a) If the workstation transformation update state entry in the ws statelist is PENDING the current workstation window and current workstation viewport entries in the workstation statelist are assigned the values of the requested workstation window and requested workstation viewport entries. The workstation transformation update state entry is set to NOTPENDING.

b) If the specified group is visible the group is redisplayed. This action causes all visible segments stored in the group to be redisplayed.

Errors:

-801	*GKS not in proper state: GKS shall be in state WSOP, WSAC, SGOP, GROP or GOSO*
20	*Specified workstation identifier is invalid*
25	*Specified workstation is not open*
33	*Specified workstation is of category MI*
35	*Specified workstation is of category INPUT*
36	*Specified workstation is WISS*
-830	*Specified group name is invalid*
-832	*Specified group name does not exist*
-833	*Specified group does not exist on specified workstation.*

ASSOCIATE GROUP WITH WORKSTATION WSOP, WSAC L2a

Parameters:

In workstation identifier	N
In group name	N

Effect:
The group is sent to the specified workstation in the same way as if the workstation were active when the group was created. Clipping rectangles are copied unchanged. The group name is added to the 'set of stored groups for this workstation' in the workstation statelist. The workstation identifier is included in the 'set of associated workstations' in the group statelist, and in the 'set of associated workstations' in the segment statelists of all segments contained in the group.

Note: If the group is already associated with the specified workstation the function has no effect.

Errors:

6	*GKS not in proper state: GKS shall be in state WSOP, WSAC*
25	*Specified workstation is not open*
33	*Specified workstation is of category MI*
35	*Specified workstation is of category INPUT*

27 *Workstation Independent Segment Storage is not open*
-813 *Specified workstation does not support grouping*
-830 *Specified group name is invalid*
-834 *Specified group does not exist on Workstation Independent Segment Storage*

COPY GROUP TO WORKSTATION WSOP, WSAC L2a

Parameters:

In workstation identifier	N
In group name	N

Effect:
The primitives in the segments in the group are sent to the specified workstation after group transformation and clipping at the clipping rectangle stored with each primitive. The primitives are not stored in a segment or a group.

Errors:
6 *GKS not in proper state: GKS shall be in state WSOP, WSAC*
25 *Specified workstation is not open*
33 *Specified workstation is of category MI*
35 *Specified workstation is of category INPUT*
27 *Workstation Independent Segment Storage is not open*
36 *Specified workstation is Workstation Independent Segment Storage*
-830 *Specified group name is invalid*
-834 *Specified group does not exist on Workstation Independent Segment Storage*

3.1.2. Grouping Attributes

SET GROUP TRANSFORMATION WSOP, WSAC, SGOP, GROP, GOSO L1a

Parameters:

In group name	N
In transformation matrix	2×3×R

Effect:
The 'group transformation matrix' entry in the group statelist of the named group is set to the value specified by the parameter. When a group is displayed, the coordinates of the primitives in its segments are, before being transformed by the segment transformation, transformed by applying the following matrix multiplication to them:

$$\begin{Bmatrix} x' \\ y' \end{Bmatrix} = \begin{bmatrix} M_{11} & M_{12} & M_{13} \\ M_{21} & M_{22} & M_{23} \end{bmatrix} \times \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

This function can be used to transform a group stored on a workstation. The transformation applies to all workstations where the specified group is stored, even if they are not all active.
The group transformation (conceptually) takes place in NDC space. The group transformation will be stored in the group statelist and will not affect the contents of the group. The group transformation is not cumulative.

Errors:
-801 *GKS not in proper state: GKS shall be in state WSOP, WSAC, SGOP, GROP or GOSO*
-830 *Specified group name is invalid*
-832 *Specified group name does not exist*

SET GROUP VISIBILITY WSOP, WSAC, SGOP, GROP, GOSO L1a

Parameters:

In group name		N
In visibility	(VISIBLE, INVISIBLE)	E

Effect:
The 'visibility' entry in the group statelist is set to the value specified by the parameter.

Errors:
-801 *GKS not in proper state: GKS shall be in state WSOP, WSAC, SGOP, GROP or GOSO*
-830 *Specified group name is invalid*
-832 *Specified group name does not exist*

SET GROUP SHIELDING WSOP, WSAC, SGOP, GROP, GOSO L1a

Parameters:

In group name		N
In shielding	(NOSHIELDING, SHIELDING)	E

Effect:
The 'shielding' entry in the group statelist is set to the value specified by the parameter, thus enabling or disabling the shielding of primitives outside the specified group.

Errors:
-801 *GKS not in proper state: GKS shall be in state WSOP, WSAC, SGOP, GROP or GOSO*
-830 *Specified group name is invalid*
-832 *Specified group name does not exist*

SET GROUP SHIELDING COLOUR WSOP, WSAC, SGOP, GROP, GOSO L1a

Parameters:

In group name		N
In colour index	(0,n)	N

Effect:
The shielding colour index in the group statelist is set to the value specified by the parameter. If shielding is enabled this colour will be used as background colour.

The colour index is a pointer into the colour tables of the workstations. If the specified colour index is not present in a workstation colour table, a workstation dependent colour index is used on that workstation.

Errors:
-801 *GKS not in proper state: GKS shall be in state WSOP, WSAC, SGOP, GROP or GOSO*
92 *Colour index is less than zero*
-830 *Specified group name is invalid*
-832 *Specified group name does not exist*

SET GROUP HIGHLIGHTING WSOP, WSAC, SGOP, GROP, GOSO L1a

Parameters:

In group name		N
In highlighting	(NORMAL, HIGHLIGHTED)	E

Effect:
The 'highlighting' entry in the group statelist is set to the value specified. If the group is marked as HIGHLIGHTED and VISIBLE, the group is highlighted in an implementation dependent manner.

Errors:
-801 *GKS not in proper state: GKS shall be in state WSOP, WSAC, SGOP, GROP or GOSO*
-830 *Specified group name is invalid*
-832 *Specified group name does not exist*

Grouping attributes

SET GROUP PRIORITY	WSOP, WSAC, GROP, SGOP, GOSO	L1a
Parameters:		
In group name		N
In group priority	[0,1]	N

Effect:
The group priority entry in the group statelist of the named group is set to the value specified. Group prioirty affects the display of groups and pick input (if groups can overlap).

Errors:
-801 *GKS not in proper state: GKS shall be in state WSOP, WSAC, SGOP, GROP or GOSO*
-830 *Specified group name is invalid*
-832 *Specified group name does not exist*
-835 *Specified group priority is outside range [0, 1]*

SET GROUP DETECTABILITY	WSOP, WSAC, GROP, SGOP, GOSO	L1a
Parameters:		
In group name		N
In detectability	(UNDETECTABLE, DETECTABLE)	E

Effect:
The 'detectability' entry in the group statelist of the named group is set to the value specified. If the group is marked as DETECTABLE and VISIBLE the primitives in it are available for pick input. DETECTABLE but INVISIBLE groups cannot be picked.

Errors:
-801 *GKS not in proper state: GKS shall be in state WSOP, WSAC, SGOP, GROP or GOSO*
-830 *Specified group name is invalid*
-832 *Specified group name does not exist*

SET PAN, ZOOM AND ROTATE ALLOWANCE	WSOP, WSAC, GROP, SGOP, GOSO	L1a
Parameters:		
In group name		N
In pan allowance	(DISABLED, ENABLED)	E
In zoom allowance	(DISABLED, ENABLED)	E
In rotate allowance	(DISABLED, ENABLED)	E

Effect:
The entries for pan, zoom, and rotate allowance in the group statelist of the named group are set accordingto the value specified. Panning, zooming or rotating of the primitives in the group is either enabled or disabled, depending on the setting of the flags. The primitives in the group may be panned , zoomed or rotated by the group transformation.

Errors:
-801 *GKS not in proper state: GKS shall be in state WSOP, WSAC, SGOP, GROP or GOSO*
-830 *Specified group name is invalid*
-832 *Specified group name does not exist*

3.1.3. Grouping inquiry functions

3.1.3.1. Inquiry function for GKS description table

INQUIRE WORKST. MAX. GROUP NUMBER GKOP, WSOP, WSAC, GROP, SGOP, GOSO L1a

Parameters:

Out error indicator		I
Out max nr of workstations associated with group	(1..n)	I

Effect:

Errors:

-808 *GKS not in proper state: GKS shall be in of of the states GKOP, WSOP, WSAC, GROP, SGOP, GOSO*

3.1.3.2. Inquiry functions for GKS state list

INQUIRE NAME OF OPEN GROUP GROP, GOSO L1a

Parameters:

Out error indicator	I
Out name of open group	N

Effect:

If the inquired information is available, the error indicator is returned as 0 and values are returned in the output parameters.

Errors:

-807 *GKS not in proper state: GKS shall be in state GROP or GOSO.*

INQUIRE SET OF GROUP NAMES IN USE WSOP, WSAC, GROP, SGOP, GOSO L1a

Parameters:

Out error indicator		I
Out number of segment names	(0..n)	I
Out set of group names in use		n×N

Effect:

If the information is available, the error indicator is returned as 0 and values are returned in the output parameters.

Errors:

-801 *GKS not in proper state: GKS shall be in one of the states WSOP, WSAC, SGOP, GROP, GOSO.*

3.1.3.3. Inquiry functions for workstation statelist

INQUIRE SET OF GROUP NAMES ON WORKST. WSOP, WSAC, GROP, SGOP, GOSO L1a

Parameters:

In workstation identifier		N
Out error indicator		I
Out number of group names	(0..n)	I
Out set of stored groups for this ws		n×N

Effect:

If the information is available, the error indicator is returned as 0 and values are returned in the output parameters.

Errors:

-801 *GKS not in proper state: GKS shall be in one of states WSOP, WSAC, GROP, SGOP, GOSO*

20 ***Specified workstation identifier is invalid***

25 *Specified workstation is not open*
33 *Specified workstation is of category MI*
35 *Specified workstation is of category INPUT*
-813 *Specified workstation does not support grouping*

INQUIRE WORKSTATION GROUP STATE WSOP, WSAC, GROP, SGOP, GOSO L1a

Parameters:

Out error indicator		I
Out workstation group state	(NOGROUPOPEN, GROUPOPEN)	E

Effect:
If the information is available, the error indicator is returned as 0 and values are returned in the output parameters.

Errors:
-801 *GKS not in proper state: GKS shall be in one of states WSOP, WSAC, GROP, SGOP, GOSO*
20 *Specified workstation identifier is invalid*
25 *Specified workstation is not open*
33 *Specified workstation is of category MI*
35 *Specified workstation is of category INPUT*
-813 *Specified workstation does not support grouping*

3.1.3.4. Inquiry functions for workstation description table

INQUIRE NUMBER OF GROUP PRIORITIES SUPPORTED GKOP, WSOP, WSAC, GROP, SGOP, GOSO L1a

Parameters:

In workstation type		N
Out error indicator		I
Out number of group priorities supported	(0..n)	I

Effect:
If the inquired information is available, the error indicator is returned as 0, and values are returned in the output parameters.

Errors:
-808 *GKS not in proper state: GKS shall be in one of the states GKOP, WSOP, WSAC, GROP, SGOP, GOSO*
22 *Specified workstation type is invalid*
23 *Specified workstation type does not exist*
39 *Specified workstation is neither of category OUPUT nor of category OUTIN*
-813 *Specified workstation does not support grouping*

INQUIRE DYNAMIC MODIFICATION OF GROUP ATTRIBUTES WSOP, WSAC, GROP, SGOP, GOSO L1a

Parameters:

In workstation type		N
Out error indicator		I
Out group transformation changeable	(IRG, IMM)	E
Out visibility changeable from 'visible' to 'invisible'	(IRG, IMM)	E
Out visibility changeable from 'invisible' to 'visible'	(IRG, IMM)	E
Out shielding changeable	(IRG, IMM)	E
Out back ground colour changeable	(IRG, IMM)	E
Out highlighting changeable	(IRG, IMM)	E
Out group priority changeable	(IRG, IMM)	E

Out adding primitives to open segm in open group	(IRG, IMM)	E
Out group deletion immediately visible	(IRG, IMM)	E

Effect:
If the inquired information is available, the error indicator is returned as 0 and values are returned in the output parameters.
IRG means that implicit regeneration is necessary; IMM means that the action is performed immediately.

Errors:

-808 *GKS not in proper state; GKS shall be in one of the states GKOP, WSOP, WSAC, SGOP, GROP, GOSO*
22 *Specified workstation type is invalid*
23 *Specified workstation type does not exist*
39 *Specified workstation is neither of category OUTPUT nor of category OUTIN*
-813 *Specified workstation type does not support grouping*

INQUIRE GROUP SUPPORT ON WORKSTATION TYPE GKOP, WSOP, WSAC, GROP, SGOP, GOSO L1a

Parameters:

In workstation type	N
Out error indicator	I
Out group support available	E

Effect:
If the inquired information is available, the error indicator is returned as 0 and a value is returned in the output parameter.

Errors:

-808 *GKS not in proper state: GKS shall be in one of the states GKOP, WSOP, WSAC, GROP, GOSO, SGOP*
22 *Specified workstation type is invalid*
23 *Specified workstation type does not exist*
39 *Specified workstation is neither of category OUTPUT nor of category OUTIN.*

3.1.3.5. Inquiry functions for segment statelist

INQUIRE NAME OF GROUP CONTAINING SEGMENT WSOP, WSAC, GROP, SGOP, GOSO L1a

Parameters:

In segment name	N
Out error indicator	I
Out name of group containing segment	N

Effect:
If the inquired information is available, the error indicator is returned as 0 and the values are returned in the output parameters.

Errors:

-801 *GKS not in proper state: GKS shall be in one of the states WSOP, WSAC, GROP, SGOP, GOSO*
120 *Specified segment name is invalid*
122 *Specified segment name does not exist*
-845 *Specified segment is not contained in any group*

3.1.3.6. Inquiry functions for group statelist

INQUIRE SET OF SEGMENTS ASSOCIATED WITH A GROUP	WSOP, WSAC, GROP, SGOP, GOSO	L1a

Parameters:

In group name		N
Out error indicator		I
Out number of associated segments	(0..n)	I
Out set of associated segments		n×N

Effect:
If the inquired information is available, the error indicator is returned as 0 and the values are returned in the output parameters.

Errors:
-801 *GKS not in proper state: GKS shall be in one of the states WSOP, WSAC, GROP, SGOP, GOSO*
-830 *Specified group name is invalid*
-832 *Specified group name does not exist*

INQUIRE GROUP ATTRIBUTES	WSOP, WSAC, GROP, SGOP, GOSO	L1a

Parameters:

In group name		N
Out error indicator		I
Out group clipping window		4×R
Out group transformation matrix		2×3×R
Out group priority	[0,1]	R
Out visibility	(VISIBLE, INVISIBLE)	E
Out detectability	(UNDETECTABLE, DETECTABLE)	E
Out shielding	(NOSHIELDING, SHIELDING)	E
Out shielding colour	(0..n)	I
Out highlighting	(NORMAL, HIGHLIGHTED)	E
Out panning enabled	(DISABLED, ENABLED)	E
Out zooming enabled	(DISABLED, ENABLED)	E
Out rotating enabled	(DISABLED, ENABLED)	E

Effect:
If the information is available, the error indicator is returned as 0 and values are returned in the ouptut parameters. Among these values the current group transformation matrix, as is set and stored in the group statelist, is returned.

Errors:
-801 *GKS not in proper state: GKS shall be in state WSOP, WSAC, GROP, GOSO or SGOP*
-830 *Specified group name is invalid*
-832 *Specified group name does not exist*

INQUIRE DYNAMIC GROUP TRANSFORMATION	WSOP, WSAC, GROP, SGOP, GOSO	L1a

Parameters:

In workstation name		N
In group name		N
Out error indicator		I
Out transformation matrix		2×3×R

Effect:
The current group transformation matrix, as is realized on the given workstation, is returned.

Errors:

-801 *GKS not in proper state.*
20 *Specifed workstation identifier is invalid*
25 *Specified workstation is not open*
33 *Specified workstation is of category MI*
35 *Specified workstation is of category INPUT*
36 *Specifed group does not exist*
-813 *Specified workstation is of category WISS*
-830 *Specified group name is invalid*
-832 *Specified workstation does not support grouping*
-839 *Specified workstation not associated with specified group*

INQUIRE DYNAMIC GROUP WINDOW WSOP, WSAC, GROP, SGOP, GOSO L1a

Parameters:

In workstation name		N
In group name		N
Out error indicator		I
Out window	WC	2×P

Effect:
if the inquired information is available, the current rectangle in worldcoordinates that maps the viewport of the given group if the current normalization transformation and the current dynamic group transformation are applied, is returned.

Errors:

-801 *GKS not in proper state:GKS shall be in of one of the states WSOP, WSAC, SGOP, GROP, GOSO*
20 *Specifed workstation identifier is invalid*
25 *Specified workstation is not open*
33 *Specified workstation is of category MI*
35 *Specified workstation is of category INPUT*
36 *Specified workstation is of category WISS*
-813 *Specified workstation does not support grouping*
-830 *Specified group name is invalid*
-832 *Specifed group does not exist*
-839 *Specified workstation not associated with specified group*

4. GKS DATA STRUCTURES

4.1. Statelist extensions

Besides a new data structure for the Group statelist some new fields and changes are to be added to the existing GKS data structures.

4.2. Group state list

group name		N	
set of associated workstations		n×N	
(list contains all workstations active at opening the group. associated workstions may be activated or deactivated while the group is open. this action however does not affect the list. activated workstations not contained in this list are not 'seen' by this group.)			
set of stored segments		n×N	empty
clipping window	NDC	4×R	0,1,0,1
group transformation matrix		2×3×R	1,0,0 0,1,0
(the elements M_{13} , M_{23} of the transformation matrix are in NDC coordinates, the other elements are unitless)			
visibility	(VISIBLE, INVISIBLE)	E	VISIBLE
shielding	(NOSHIELDING, SHIELDING)	E	SHIELDING
shielding colour		(0, n)	0
highlighting	(NORMAL, HIGHLIGHTED)	E	NORMAL
group priority	[0, 1]	R	0
detectability	(UNDETECTABLE, DETECTABLE)	E	UNDET.
panning	(DISABLED, ENABLED)	E	ENABLED
zooming	(DISABLED, ENABLED)	E	ENABLED
rotating	(DISABLED, ENABLED)	E	ENABLED

4.3. Operating state

change			
Operating state	(GKCL, GKOP, WSOP, WSAC, SGOP)	E	GKCL
into			
Operating state	(GKCL, GKOP, WSOP, WSAC, SGOP, GROP, GOSO)	E	GKCL

4.4. GKS description table

The following must be added to the GKS description table.

max nr of workstations associated with a group	(1..n)	I	i.d.

4.5. Global statelist

The following must be added to the global statelist.

name of open group	N	undef
set of group names in use	n×N	empty
set of group state lists (one for every group)		empty

4.6. Workstation description table

The following must be added to the workstation description table:

Entries in this group do not exist for ws of type INPUT, WISS, MO and MI			
workstation is able to support grouping	(GROUPING, NOGROUPING)		ENOGROUPING
number of group priorities supported	(0..n)	I	i.d.
if ws is able to support grouping then dynamic modification accepted for:			
group transformation	(IRG, IMM)	E	i.d.
visibility(visible->invisible)	(IRG, IMM)	E	i.d.
visibility(invisible->visible)	(IRG, IMM)	E	i.d.
shielding	(IRG, IMM)	E	i.d.
back ground colour	(IRG, IMM)	E	i.d.
highlighting	(IRG, IMM)	E	i.d.
group priority	(IRG, IMM)	E	i.d.
adding primitives to open segment in open group overlapping segment in group of higher priority	(IRG, IMM)	E	i.d.
delete group	(IRG, IMM)	E	i.d.

4.7. Workstation statelist

The following must be changed in / added to the workstation statelist.

Entries in this group only exist for workstations able to support grouping			
workstation group state	(NOGROUPOPEN, GROUPOPEN)		NOGROUPOPEN
set of stored groups for this ws		n×N	empty
change			
set of stored segments for this ws			
into			
set of stored segments outside groups for this ws			

4.8. Segment statelist

The following must be added to the segment statelist.

name of group containing segment		N	undef

5. METAFILES

5.1. Generation of metafiles

The list of 'GKS functions and their effect on GKSM output workstations' (table 4 in annex E) is to be extended with the following entries:

GKS functions which apply to workstations of category MO	**GKS item created or effect**
Control functions	
REDRAW GROUP ON WORKSTATION	201
Group functions	
CREATE GROUP	211
OPEN GROUP	212
CLOSE GROUP	213
DELETE GROUP	214
DELETE GROUP FROM WORKSTATION	214
CLEAR GROUP	215
ASSOCIATE GROUP WITH WS	
COPY GROUP TO WORKSTATION	
Group attributes	
SET GROUP TRANSFORMATION	221
SET GROUP VISIBILITY	222
SET GROUP SHIELDING	223
SET GROUP SHIELDING COLOUR	224
SET GROUP HIGHLIGHTING	225
SET GROUP PRIORITY	226
SET GROUP DETECTABILITY	227
SET GROUP PAN/ZOOM/ROTATE ALLOWANCE	228

5.2. Interpretation of metafiles

E.4.9 Group manipulation

interpretation of the CREATE GROUP item causes the invocation of GKS functions SET CLIPPING INDICATOR with parameter CLIP, SET VIEWPORT with a clipping window and normalization transformation in accordance to the information stored in the item, SELECT NORMALIZATION TRANSFORMATION with the group name as parameter, and CREATE GROUP. The function SET VIEWPORT is not called if the group name is equal 0.
Item 214 causes an invocation of DELETE GROUP.
Interpretation of the other items in this class has the same effect as invocation of the corresponding GKS functions.

E.4.10 Group attributes

Interpretation of the items in this class has the same effect as invocation of the corresponding GKS functions.

5.3. Control items

The following items are to be added to the list of metafile items.

E.5 Control items

REDRAW GROUP ON WORKSTATION

'GKSM 201'	L	G

G(i): group name

requests REDRAW GROUP ON WORKSTATION on all active workstations

E.13 Items for group manipulation

CREATE GROUP

'GKSM 211'	L	G

G(i): group name

OPEN GROUP

'GKSM 212'	L	G

G(i): group name

CLOSE GROUP

'GKSM 213'	L	G

G(i): group name

DELETE GROUP

'GKSM 214'	L	G

G(i): group name

CLEAR GROUP

'GKSM 215'	L	G

G(i): group name

E.14 Items for group attributes

SET GROUP TRANSFORMATION

'GKSM 221'	L	G	M

G(i): group name
M(6r): transformation matrix:

M_{11}, M_{12}, M_{13}, M_{21}, M_{22}, M_{23}

SET GROUP VISIBILITY

'GKSM 222'	L	G	S

G(i): group name
S(i): visibility (1 = VISIBLE, 0 = INVISIBLE)

SET GROUP SHIELDING

'GKSM 223'	L	G	S

G(i): group name
S(i): shielding (0 = NOSHIELDING, 1 = SHIELDING)

SET GROUP SHIELDING COLOUR

'GKSM 224'	L	G	C

G(i): group name
C(i): colour index

SET GROUP HIGHLIGHTING

'GKSM 225'	L	G	H

G(i): group name
H(i): highlighting
(0 = NORMAL, 1 = HIGHLIGHTED)

SET GROUP PRIORITY

'GKSM 226'	L	G	P

G(i): group name
P(r): group priority

SET GROUP DETECTABILITY

'GKSM 227'	L	G	D

G(i): group name
D(i): detectability
(0 = UNDETECTABLE, 1 = DETECTABLE)

SET GROUP PAN, ZOOM AND ROTATE ALLOWANCE

'GKSM 228'	L	G	D	D	D

G(i): group name
D(i): panning
D(i): zooming
D(i): rotating
(0 = DISABLED, 1 = ENABLED)

6. Error list

The following errors are added to the error list.

B.2 States

-800 GKS not in proper state: GKS shall be in the state GROP
-801 GKS not in proper state: GKS shall be in one of the states WSOP, WSAC, SGOP, GROP or GOSO
-802 GKS not in proper state: GKS shall be in one of the states WSAC or GROP
-803 GKS not in proper state: GKS shall be in one of the states SGOP or GOSO
-804 GKS not in proper state: GKS shall be in one of the states SGOP, GROP or GOSO
-805 GKS not in proper state: GKS shall be in one of the states WSAC, SGOP, GROP or GOSO
-806 GKS not in proper state: GKS shall be in one of the states WSOP, WSAC or GROP
-807 GKS not in proper state: GKS shall be in one of the states GROP, GOSO
-808 GKS not in proper state: GKS shall be in one of the states GKOP, WSOP, WSAC, SGOP, GROP or GOSO

B.3 Workstations

-811 Specified workstation not known in group containing specified segment
-813 Specified workstation does not support grouping

B.14 Grouping

-830 Specified group name is invalid
-831 Specified group name is already in use
-832 Specified group name does not exist
-833 Specified group name does not exist on specified workstation
-834 Specified group does not exist on Workstation Independent Segment Storage
-835 Specified group priority is outside range [0, 1]
-836 None of the active workstations is able to support grouping
-837 The normalization transformation associated with the specified group is not selected.
-838 Tried to close the last ws associated with the currently open group
-839 Specified workstation not known on specified group
-840 Specified group name not equal to current normalization transformation number
-841 Specified segment not contained in currently open group
-842 Group containing specified segment must be open
-843 Tried to deactivate the last active ws associated with the currently open group
-844 Specified group is open
-845 Specified segment not contained in any group
-846 There exists already a group with group name equal to the current normalization transformation number

B.4 Transformations

-861 Contents of specified normalization transformation may not be changed, as transformation is connected to a group.
-862 Not allowed to select another normalization transformation during states GROP and GOSO.

Test implementation

The described grouping functionality is implemented and tested as an extension to C-GKS, being the GKS implementation in C, developed at the Centre for Mathematics and Computer Science.[3] An addition to our C binding of GKS[4] is defined for the grouping functions and data types. To enable C-GKS programmers to test and use the grouping extension two appendices are added. The first appendix contains the C binding for both the type definitions and the functions.
Appendix B contains the extension that is made to the di/dd interface, as stated by the structured data type 'Screen'. Via these grouping entries group control can be performed by the workstation driver.

Acknowledgement

Discussions with Miente Bakker, Kees Blom, Fons Kuyk and Bill Sass markedly improved earlier versions of the grouping specification.

References

1. ISO (1985). *Information Processing - Graphical Kernel System - Functional description,* International Standard DIS 7942.
2. F.R.A. Hopgood, D.A. Duce, E.V.C. Fielding, K. Robinson, A.S.Williams ed. (1986). *Methodology of Window Management*, Springer-Verlag, Berlin.
3. M.M.de Ruiter. *C-GKS user guide*, CWI, Amsterdam, to be published.
4. D.S.H. Rosenthal & P.J.W. ten Hagen (1982). GKS in C, *Proceedings Eurographics '82*, p. 359-369.

APPENDIX A: C BINDING

Group type definitions
The following new C-GKS typedefinitions are relevant for the GKS groups.

```
typedef Int Grpname;
typedef Real Grppri;
typedef struct {
Grpname     gr_name;   /* group name */
Wss         **gr_onws; /* pointer to list of associated ws, followed by NULL*/
Nrect       gr_clip;   /* group clipping window */
Tmat        gr_mat;    /* group transformation matrix */
Bool        gr_vis;    /* group visibility */
Bool        gr_shld;   /* shielding enabled or disabled */
Cindex      gr_shcol;  /* colour of shield */
Bool        gr_hil;    /* group highlighting */
Grppri      gr_pri;    /* group priority */
Bool        gr_det;    /* group detectability */
Bool        gr_pan;    /* panning enabled */
Bool        gr_zoom;   /* zooming enabled */
Bool        gr_rotate; /* rotating enabled */
Seginst     *gr_segs;  /* pointer to segments associated with group */
}Group;
```

Several existing typedefs contain some new, grouping special, fields.
They are listed at the end of this section.

Grouping manipulation functions

CREATE GROUP	**Group *newgroup()**

Diagnostics:
For further reference to the newly created group a pointer to a structure of type Group is returned.

OPEN GROUP GROUP NAME	**opengroup(gr)** Group *gr;
CLOSE GROUP	**closgroup()**
DELETE GROUP IN GROUP NAME	**zapgroup(grp)** Group *grp;
DELETE GROUP FROM WORKST. IN WORKSTATION IDENTIFIER IN GROUP NAME	**delgroup(ws, grp)** Wss *ws; Group *grp;
CLEAR GROUP IN GROUP NAME	**clr_group(grp)** Group *grp;
REDRAW GROUP ON WORKSTATION IN WS IDENTIFIER IN GROUP NAME	**redr_group(ws, grp)** Wss *ws; Group *grp;
ASSOCIATE GRP. WITH W.S. IN WORKSTATION IDENTIFIER IN GROUP NAME	**sendgroup(ws, grp)** Wss *ws; Group *grp;

C binding

COPY GROUP TO WORKSTATION
IN WORKSTATION IDENTIFIER
IN GROUP NAME

copygroup(ws, grp)
Wss *ws;
Group *grp;

Grouping Attributes

TRANSFORM GROUP
IN GROUP NAME
IN TRANSFORMATION MATRIX

transgroup(grp, mat)
Group *grp;
Tmat *mat;

SET GROUP VISIBILITY
IN GROUP NAME
IN VISIBILITY

grp_vis(grp, vis)
Group *grp;
Bool vis;

SET HIGHLIGHTING
IN GROUP NAME
IN HIGHLIGHTING

grp_hil(grp, hlt)
Group *grp;
Bool hlt;

SET GROUP PRIORITY
IN GROUP NAME
IN GROUP PRIORITY

grp_prio(grp, pri)
Group *grp;
Grppri pri;

SET GROUP DETECTABILITY
IN GROUP NAME
IN DETECTABILITY

grp_det(grp, det)
Group *grp;
Bool det;

Diagnostics:
A 'det' value being TRUE means detectability enabled.

SET GROUP SHIELDING
GROUPNAME
SHIELDING

grp_shield(gr, shield)
Group *gr;
Bool shield;

Diagnostics:
A 'shield' value being TRUE means shielding enabled.

SET GROUP SHIELDING COLOUR
GROUP NAME
SHIELD COLOUR

grp_shcol(gr, col)
Group *gr;
Cindex col;

SET GROUP PAN, ZOOM, ROTATE
GROUP NAME
PAN ALLOWANCE
ZOOM ALLOWANCE
ROTATE ALLOWANCE

grp_panzoom(gr, pan, zoom, rotate)
Group *gr;
Bool pan;
Bool zoom;
Bool rotate;

C inquiry functions.
In addition to the GKS inquiry functions, new inquiry functions are defined to inquire values of the various statelists concerning grouping. If the inquiry functions are available via functions the following binding is supported.

INQUIRE WS MAX GROUP NUMBER
OUT MAX GRP NUMBER

Ercode i_max_nrgrp(max)
Int *max;

Diagnostics:
On return **max** is filled with the maximum number of groups on a workstation.

INQUIRE NAME OF OPEN GROUP
OUT GROUP NAME

Ercode i_opengrpname(name)
Group **name;

Diagnostics:
On return **name** is filled with a pointer to the statelist of currently open group.

INQUIRE SET OF GROUP NAMES IN USE

Ercode i_set_grps(nrnames, grpnames, sn_sz)
Int *nrnames;
Grpname *grpnames;
Int sn_sz;

Diagnostics:
On return the user supplied integer nrnames will contain the number of group names in use, while the user supplied array **grpnames**, being of size **sn_sz**, will contain the group names. If *nrnames is larger than sn_sz, only the first sn_sz entries are filled, and error -18 is returned.

INQUIRE SET OF GROUP NAMES ON WS

Ercode i_grps_ws(ws, nrnames, sname, sn_sz)
Wss *ws;
Int *nrnames;
Grpname *sname;
Int sn_sz;

Diagnostics:
On return the user supplied parameter **nrnames** is filled with the number of group names on workstation. The user supplied array **sname** is filled with the group names. If the array is not large enough to hold all the names, only the first sn_sz names are stored and an error is returned.

INQUIRE WORKSTATION GROUP STATE
IN WORKSTATION
OUT WS GROUP STATE

Ercode i_wsgrpstate(ws, gr_open)
Wss *ws;
Bool *gr_open;

Diagnostics:
On return **gr_open** is filled with the workstation group state. If there is a group open on the workstation gr_open is TRUE, else FALSE.

INQUIRE NR OF GROUP PRIO'S SUPP.
IN WS TYPE
OUT NR OF GROUP PRIO'S SUPP

Ercode i_nr_grpprio(wsd, nr_grpprio)
Wsd *wsd;
Int *nr_grpprio;

Diagnostics:
On return the user supplied parameter **nr_grpprio** is filled with the number of group priorities supported. on a workstation.

INQ DYN MOD OF GROUP ATTS
IN WS TYPE
OUT IMPLICIT REGENERATION FLAGS

Ercode i_dm_grp_atts(wsd, irgflags)
Wsd *wsd;
Int *irgflags;

Diagnostics:
On return the user supplied parameter **irgflags** contains a mask for the dynamic modification settings for group attributes, where:

group transformation	if (*irgflags & 01)	then IMM else IRG
visible -> invisible	if (*irgflags & 02)	then IMM else IRG
invisible -> visible	if (*irgflags & 04)	then IMM else IRG
shielding	if (*irgflags & 010)	then IMM else IRG
back ground col change	if (*irgflags & 020)	then IMM else IRG
hilighting	if (*irgflags & 040)	then IMM else IRG
group priority	if (*irgflags & 0100)	then IMM else IRG
adding prims to open seg in grp	if (*irgflags & 0200)	then IMM else IRG
grp deletation immediate visible	if (*irgflags & 0400)	then IMM else IRG
grp clearance immediate visible	if (*irgflags & 01000)	then IMM else IRG

INQUIRE GROUP SUPPORT ON WS
IN WS TYPE
OUT GROUP SUPPORT

Ercode i_grp_support(wsd, support)
Wsd *wsd;
Bool *support;

Diagnostics:
On return **support** is filled with TRUE if the workstation type supports grouping, else FALSE.

C binding

INQUIRE NAME OF GROUP CONT. SEGM.	**Ercode i_grpname_seg(segname, sg_grp)**
IN SEGMENT NAME	Segname segname;
OUT GROUP NAME	Group **sg_grp;

Diagnostics:
On return **sg_grp** is filled with the group pointer of the group containing the segment with segmentname **segname**.

INQUIRE SET OF SEGMENTS IN GROUP	**Ercode i_segs_in_grp(grp, nr_segs, segs, sn_sz)**
IN GROUP NAME	Group *grp;
OUT NUMBER OF SEGMENTS IN GROUP	Int *nr_segs;
OUT SEGMENTS IN GROUP	Seg **segs;
IN SIZE OF ARRAY segs	Int sn_sz;

Diagnostics:
On return **nr_segs** is filled with the number of segments in the group. The user supplied array **segs**, being of size **sn_sz**, is filled with the segment pointers of the segments available in the group. If *nr_segs is larger than the sn_sz, only the first sn_sz entries are filled, and error -18 is returned.

INQUIRE GROUP ATTRIBUTES	**Ercode i_grpatt(grpname, group)**
IN GROUP NAME	Grpname grpname;
OUT GROUP ATTRIBUTES	Group **group;

Diagnostics:
On return the group pointer **group** will point to the group structure belonging to the group with name **grpname**. The Group structure will contain among others the information asked for. The Group type looks as follows:

```
typedef   struct {
          Grpname    gr_name;     /*group name */
          Wss        **gr_onws;   /*pointer to list of associated ws, followed by NULL*/
          Nrect      gr_clip;     /* group clip rectangle */
          Tmat       gr_mat;      /* group transformation matrix */
          Bool       gr_vis;      /* group visibility */
          Bool       gr_shld;     /* shielding enabled or disabled */
          Cindex     gr_shcol;    /* colour of shield */
          Bool       gr_hil;      /* group highlighting */
          Grppri     gr_pri;      /* group priority */
          Bool       gr_det;      /* group detectability */
          Bool       gr_pan;      /* panning enabled */
          Bool       gr_zoom;     /* zooming enabled */
          Bool       gr_rotate;   /* rotating enabled */
          Seginst    *gr_segs;    /* pointer to segments associated with group */
          } Group;
```

INQUIRE DYNAMIC GROUP TRAFO	**Ercode i_grdyntrafo(ws, gr, tmat)**
IN WORKSTATION NAME	Wss *ws;
IN GROUP NAME	Group *gr;
OUT GROUP TRANSFORMATION	Tmat *tmat;

Diagnostics:
On return **tmat** is filled with the group transformation matrix as dynamically set on the workstation.

INQUIRE DYNAMIC GROUP WINDOW	**Ercode i_grwindow(ws, gr, wc)**
IN WORKSTATION NAME	Wss *ws;
IN GROUP NAME	Group *gr;
OUT WINDOW	Wc *wc;

Diagnostics:
On return the 4 user supplied world coordinates pointed to by **wc**, are filled with the group window as dynamically set on the workstation.

C inquiry macro's

Nearly all the inquire functions can be made available via macros, producing in-line code. They all inquire a field of one of the available statelists, and return the value of that particular field. The argument 'fn' is a field name of the structure.
To enable the use of the inquiry functions via macro's producing in-line macro's, the following macro binding is defined.

INQ WS MAX GROUP NUMBER Out max nr of ws ass with grp	call: **inq_gkd(gd_nwsag)**	return val of type: Int
INQ NAME OF OPEN GROUP Out name of open group	call: **inq_gkss(gk_opgrp)**	return val of type: Group *
INQ SET OF GROUP NAMES IN USE Out set of group names in use	call: **inq_gkss(gk_grp)**	return val of type: Grpinst *
INQ SET OF GROUP NAMES ON WS Out set of stored groups for this ws	call: **inq_wss(ws, ws_grp)**	return val of type: Grpinst *
INQ WORKSTATION GROUP STATE Out workstation group state	call: **inq_wss(ws, ws_gropen)**	return val of type: Bool
INQ NR OF GROUP PRIO'S SUPP Out nr of group prio's supported	call: **inq_wsd(wsd, wd_ngprio)**	return val of type: Int
INQ DYN MOD OF GROUP ATTR	call: **inq_wsd(wsd, wd_dmag)**	return val of type: Int

The field wd_dmag is an integer. If the bits of this integer are numbered from RIGHT to LEFT then the bits stand for:
A bit being 0 means IRG, a bit being 1 means IMM.

group transf. changeable:	1th bit
visib. changeable from vis to invis:	2nd bit
visib. changeable from invis to vis:	3nd bit
shielding changeable:	4th bit
back ground colour changeable:	5th bit
highlighting changeable:	6th bit
group priority changeable:	7th bit
adding prim's to seg in open grp:	8th bit
group deletion imm. visible:	9th bit
group clearance imm. visible:	10th bit

INQ GROUP SUPP ON WS Out group support available	call: **inq_wsd(wsd, wd_grp)**	return val of type: Bool
INQ SET OF SEGMENTS ASS. WITH GROUP Out set of segments	call: **inq_grp(grp, gr_segs)**	return val of type: Seginst *
INQ GROUP ATTRIBUTES	call:	return val of type:
Out group clipping window	**inq_grp(grp, gr_clip)**	Nrect
Out group transformation matrix	**inq_grp(grp, gr_mat)**	Tmat
Out group priority	**inq_grp(grp, gr_pri)**	Grppri
Out visibility	**inq_grp(grp, gr_vis)**	Bool
Out detectability	**inq_grp(grp, gr_det)**	Bool
Out shielding	**inq_grp(grp, gr_shld)**	Bool
Out shielding colour	**inq_grp(grp, gr_shcol)**	Cindex
Out highlighting	**inq_grp(grp, gr_hil)**	Bool
Out panning enabled	**inq_grp(grp, gr_pan)**	Bool
Out zooming enabled	**inq_grp(grp, gr_zoom)**	Bool
Out rotating enabled	**inq_grp(grp, gr_rotate)**	Bool
INQ NAME OF GROUP CONTAINING SEGMENT Out name of group containing segm.	call: **inq_seg(seg, sg_grp)**	return val of type: Grpinst *

The inquiry functions INQUIRE DYNAMIC GROUP TRANSFORMATION and INQUIRE DYNAMIC GROUP WINDOW couldn't be made availabel via inline macros. They are only available as functions.

New fields in existing C types.

The following C-GKS typedefs contain some new fields. The newly added entries are printed bold.

```
/*
 * segment statelist
 */
typedef     struct {
        Segname     sg_sn;
        Wss         *sg_onws[NWSAS+1];
        Tmat        sg_mat;
        Bool        sg_vis;
        Bool        sg_hil;
        Segpri      sg_pri;
        Bool        sg_det;
        Bhead       *sg_firstbh;
        Int         *sg_local;
        Group       *sg_grp; /* pointer to group containing segment */
        }   Seg;

/*
 * global description table
 */
typedef struct {
        GksLevl     gd_lev;
        Int         gd_nrwt;
        String      *gd_avwt;
        Int         gd_nopws;
        Int         gd_nactv;
        Int         gd_nwsas;
        Int         gd_nntran;
        Int         gd_nwsag; /* maximum number of groups on a workstation */
        }   Gksd;

/*
 * workstation description table
 */
typedef     struct {
Wscat       wd_cat;
Wsrov       wd_rov;
Screen      *wd_screen;
.
.
.
Int         wd_nprio;
Int         wd_ndevs[6];
Iddescr     *wd_ddesc[6];
Bool        wd_grp; /*  workstation type supports grouping */
Int         wd_ngprio;/* number of group priorities supported */
Int         wd_dmag; /* dynamic modification accepted flags*/
}   Wsd;
```

```
/*
 * workstation statelist
 */
typedef struct {
        Wsd         *ws_wsd;
        Wsis        *ws_wsis;
        .
        .
        Int          ws_ncolr;
        Colour      *ws_colr;
        Idevice     *ws_devs[6];
        Bool        ws_gropen;  /* currently open group on workstation */
        Grpinst     *ws_grp;        /* set of groups on workstation */
        } Wss;

/*
 * GKS global statelist
 */
typedef struct {
        Wss         **gk_opws;
        Wss         **gk_actv;
        Asflags     gk_asf;
        Bindex      gk_line;
        .
        .
        .
        Seg         *gk_opsg;
        Seginst     *gk_segs;
        Quevent     *gk_queue;
        Bool        gk_more;

        Group       *gk_opgrp; /* currently open group */
        Grpinst     *gk_grp;     /* set of existing groups */
        } Gks;
```

APPENDIX B

DI/DD interface

For intelligent devices, able to handle the grouping locally, the grouping extension makes it necessary to add new entries to the di/dd interface. New entries are added at the end of the current Screen structure.

```
typedef struct {
        Ic          s_chsz;
        Int         s_chht;
        Bool        s_clipflg;
        Bool        s_hattflg;
        Bool        s_segmflg;
        Bool        s_sgtrflg;
        Bool        s_hinpflg;

        Bool        (*s_open)();
        Bool        (*s_clos)();
        .
        .
        .
        Bool        (*s_irep)();
        Bool        (*s_iidv)();
        Bool        (*s_itex)();
        Bool        (*s_res4)();
        Bool        (*s_res5)();
        Bool        (*s_gcix)();

/***GKS GROUP extension entries*****************************************/

        Bool        s_grpflg;    /* TRUE if ws can handle grouping.
                                  * This implies that the following
                                  * entries are unequal NULL
                                  */
        Bool        (*s_rdgr)(); /* redraw all groups on workstation */
        Bool        (*s_grcr)(); /* new group */
        Bool        (*s_grop)(); /* open group */
        Bool        (*s_grcl)(); /* close group */
        Bool        (*s_grdl)(); /* delete group */
        Bool        (*s_grem)();/* clear/empty group */
        Bool        (*s_grat)(); /* set group attributes */
        Bool        (*s_igrt)(); /* inquire group transformation */
} Screen;
```

The parameters of the functions that can be called via the new entries are:

```
Bool
dev_grcr(ws, grp) /* new group */
Wss     *ws;
Group *grp;

Bool
dev_grcl(ws) /* close group */
Wss         *ws;

Bool
dev_grop(ws, gr) /* open group */
Wss     *ws;
Group   *gr;

Bool
dev_grdl(ws, grpname) /* delete group */
Wss     *ws;
Group       *grpname;

Bool
dev_grem(ws, grpname) /* clear group */
Wss     *ws;
Group   *grpname;
```

```
Bool
dev_grat(ws, grpname, attsel, mat, bool, prio, shield, pan)/* set grp attributes*/
Wss       *ws;
Group     *grpname;
Int       attsel; /* which of the attributes to be set
                  * attsel = S_GR_TR: set group transform
                  * attsel = S_GR_SH: set group shielding
                  * attsel = S_GR_SC: set group shielding colour
                  * attsel = S_GR_VI: set group visibility
                  * attsel = S_GR_HI: set group highlighting
                  * attsel = S_GR_PR: set group priority
                  * attsel = S_GR_DT: set group detectability
                  * attsel = S_GR_PA: set grp pan, zoom, rotate
                  * the contents of only one of the following fields will be interpreted
                  * according to the value of 'attsel'. the other parameters can be set to 0.
                  */
Real      *mat;   /* pointer to 1th element of matrix for 'set group transformation' */
Bool      bool;   /* indicates either group shielding, visibibilty, highlighting, or detectability*/
Grppri    prio;   /* group priority */
Cindex    shield; /* shielding colour */
Bool      *pan;   /* pointer to three booleans indicating pan, zoom and rotate allowance */


Bool
dev_rdgr(ws) /* redraw a group */
Wss       *ws;


Bool
dev_igrt(ws, gr,  gr_mat)      /* inquire dynamic group transformation */
Wss       *ws;
Group     *gr;
Tmat      *gr_mat; /* contains on return the transformation matrix as it is
                    *dynamically  set on the workstation
                    */
```

Parallel Graphical Output from Dialogue Cells

P.J.W. ten Hagen and H. Schouten
Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

Abstract

A system that accepts and processes graphical output from parallel processes using a single workstation, and its use in a User Interface Management System called Dialogue Cells, is described. It allows for a programmer to easily, concisely and precisely describe pictures and operations on them in a highly interactive environment. Each process has its own graphical output environment. The description and implementation of the system is based on GKS.

Introduction

many applications, the user wants to interact with a program in a complex way. eraction used to consist of a request and answer game via keyboard and screen re- ctively. Nowadays modern workstations are equipped with a large variety of input ices, that can be operated simultaneously. Input from one of these devices usually l result in some output. As the program cannot know which input, and consequently ich output, will come first, it must be ready to produce any of them simultaneously. Contemporary graphics packages cannot deal will parallelism well. Usually the tem is in some state at any time. The result of output calls depends completely this state. However, if parallel processes want to do output they cannot make any umption on the state, because a concurrent process may change it.

This paper will be dealing with how pictures are structured, what kind of operations needed and how these operations are achieved, in a system that can manipulate eral constituents of one picture simultaneously. The description uses GKS [3], as arting point and explains the facilities in terms of GKS concepts. This implicitly gests that parallel graphical output can be realized through a layer on top of GKS. wever, more efficient realizations are possible.

The system is called the **radical system**. It was developed as a part of a project on ser interface management system, called *Dialogue Cells*, [1]. To set the context of radical system, the dialogue cell system will be briefly described first.

1.1 An overview of the dialogue cell system.

The dialogue cell system (often abbreviated to DICE) allows a programmer to specify complex graphical user interfaces in a rather easy way. A specific language, a compiler and a run time system are designed for this purpose.

A dialogue cell describes a transaction between user and system. All the support given by the system during this transaction is described in the same dialogue cell. For example, a mouse can be used as a positioning device. Before the button click given by the user to select the proper position, the system is already showing where the position will be located on the screen. For more complex selections, e.g. positioning an electrical component on a board, it will rapidly become very difficult to show in real time all consequences of selecting a certain position. In case of the component example the system might have to adjust the wiring continuously. A dialogue cell construct is capable of describing the simple mouse + cursor input as well as the complex electrical component placement as one transaction. Such complex manipulations can, on the hand, be seen a (complex) transaction, on the other hand they can also be seen as being composed of more simple transactions. Consequently, a dialogue cell can be defined as a composition of other dialogue cells, often called subcells.

A dialogue cell contains a description of the control structure of its direct subcells, called the *symbol expression*. For example, the mouse transaction might be one of the composing transactions of the component placement mentioned above. The symbol expression allows for subcells to be activated in parallel. At the bottom of the hierarchy of dialogue cells are the so called *basic cells* that perform the basic (graphical) interactions. A library of basic cells is part of the dialogue cell system [5]. What actions are to be taken when a subcell returns to it, and what result it returns itself can be described in a set of rules. Rules are more appropriate here than sequences of statements, because it is not always predictable when or in what order subcells return results. A more comprehensive introduction to the dialogue cell system can be found in [8].

In figure 1 a scheme is given for the user-system transaction model used in Dialogue Cells. It is also indicated which part of the scheme is covered explicitly by the specification. The rest of the scheme is realized implicitly in the DICE language semantics.

The four component of the DICE language part of the scheme specify the four separate parts that can be distinguished in each transaction:

- *Prompt part*, which activates all supporting processes and resources. It also does all initializations and informs the user that this transaction can take place.
- *Symbol part*, which contains the symbol expression. On the basis of these expressions available results, e.g., user inputs are selected (parsed) for further processing.
- *Value part*, which generates the internal values corresponding to the input values accepted. The value part rules describe how internal value results of subcells are to be used, checks the results, calls application procedures or sends messages to the application, updates a data base, etc [7].

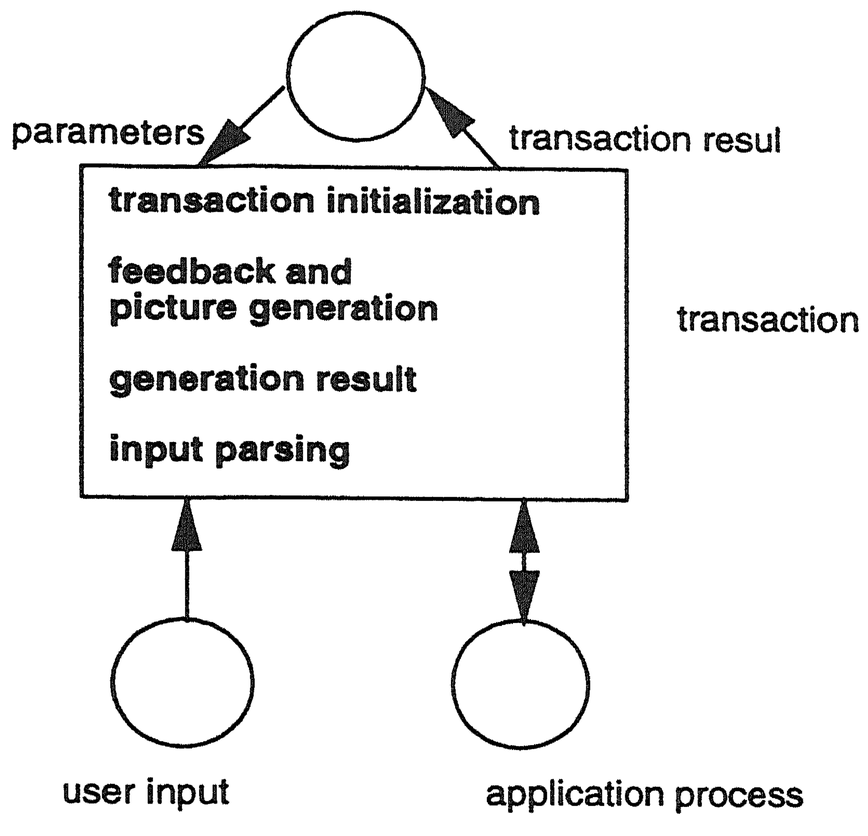


Figure 1: Transaction scheme.

- *echo part*, which generates the picture changes that visualize the new state associated with the successful input parse. As a result of such action rules new pictures may appear, existing pictures may disappear or change, either individually or in groups. Because several transactions can be in progress concurrently, the corresponding visualization steps of both prompt and echo part can be generated for several transactions concurrently. A subcell may, apart from the value result, also have an echo result consisting of ne or more pictures. These are handed over to the parent together with the internal result values. This *echo result* can subsequently be used in the echo parts of the parent.

1.2 Output on the basis of GKS.

The dialogue cell system is intended to be device independent. Although resource availability changes with workstations, by the appropriate choice of dialogue cells, a programmer can specify a dialogue that can run on several workstations. This requires that a device independent graphics system is used. We chose to use GKS, because it has many concepts that are useful to our purposes. The *grouping extension* of GKS [6] can be used to implement our window management requirements.

A group is a set of segments that share a common clipping window. The segments in a group are restricted to this window on the screen. A group window can shield off underlying pictures. A group transformation is available to transform all segments in the group as a whole. Some workstations allow the operator to change the group

transformation interactively, thus providing de facto zoom, pan and rotate facilities.

Dialogue cells can produce graphical output in basic units comparable to GKS output primitives. Such basic units can be combined into manipulable units comparable to segments. The dialogue cell system can at any time generate picture changes by manipulating that units (segments) of a group. The parallelism implies that several groups might be altered simultaneously. Such parallelism can be implemented on top of GKS by segment manipulation. However, GKS has some drawbacks, and dialogue cell requirements cannot directly be mapped onto GKS. This will be elaborated on in the sequel.

2 The Radical System

An important aspect of the graphical output from dialogue cells is that it is *parallel* in nature: there can be several dialogue cells active simultaneously, each producing their own pictures at the same time, in general unaware of the existence of the others. Therefore a system is needed that can deal with parallel output.

In a hierarchical system like DICE one can use the concept of *inheritance* to reduce the need for the specification of the environment: the environment is inherited from the parent and only (usually small) changes have to be specified. Inheritance is used in the radical system to reduce the need for explicit specification of output attributes. It can be implemented quite efficiently, not only in hierarchical graphics systems such as ILP [2] and PHIGS [4], but also in GKS. An example of the advantages of this mechanism is that if a subtree of the dialogue produces some fixed type of graphical objects, with certain attributes indicating this type to the user, the programmer does not have to specify these attributes in every dialogue cell of the subtree anew.

2.1 Radicals

The context described above poses a number of requirements on a graphical system. We will now present a system that will fulfill these requirements. It is centered around picture elements called *radicals*. The term radical is chosen to indicate that they react vividly with their environment.

Radicals are identifiable picture elements. They consist of a list of graphical output primitives. These primitives in turn consist of three components:

- *primitives* in the sense of GKS. That is, they have a type (polyline, polymarker, ...) and data that is relevant for this primitive type (coordinates, string, ...).
- *attributes*. The types of attributes are the same as those of the corresponding GKS primitive, except that aspect source flags are not used. This component is optional.
- *name*. This can be used to identify a primitive within a radical. Every primitive use a unique name.

Every radical also has attributes that apply to the radical as a whole. They correspond to the segment attributes of GKS (visibility, detectability, ...). No primitives can exist outside radicals. All graphical output will be in the form of radicals. In the following table a structure scheme for radicals is given, together with how the various components can be realized in GKS (in italics).

element	component
radical	radical attributes + primtives.
segment	*segment attributes*
primitive	name + type + geometry + attribute *pickid, output primitive, attributes*
radical attributes	visibility, highlighting, detect, transform

2.2 Local State

A radical can become visible on the screen only when it is in a certain environment. An environment consists of a *local state* and a list of radicals. The local state provides the workstation with the necessary information to display a radical. This includes:

- viewport. The geometrical data of radical primitives is interpreted with respect to the viewport in its local state.
- attributes. Attributes control the appearance of the primitives as they are displayed. Therefore, a primitive will always have attributes, either implicitly or explicitly. If a primitive in a radical has no attributes of its own, the attributes of the local state are assigned to it.

Environments can be created dynamically. A local state is always created from parent local state and within a certain viewport. It initially will contain the same set of attribute values as the parent at the moment of creation. Functions are available to alter these values, so that local states may diverge.

A viewport is assigned to an environment at creation time, either by inheritance or by explicit means. This viewport remains unaltered (static) during the lifetime of the environment. That is, the radical system cannot change it. However, the viewport may be in virtual screen space (such as Normalized Device Coordinates in GKS), which may be mapped onto the real screen under virtual terminal control. If this implies changes, then they are transparent to the radical system. In the following table an overview of the attribute structure of a local state is given. Each of the entries can be set and will be initialized automatically.

<table>
<tr><td colspan="4">environment name</td></tr>
<tr><td colspan="4">priority</td></tr>
<tr><td colspan="4">window-viewport</td></tr>
<tr><td colspan="3">group attributes</td><td>visibility
detectability
background color
shielding on/off
group transformation</td></tr>
<tr><td>polyline
attribute
bundle</td><td>polymarker
attribute
bundle</td><td>fillarea
attribute
bundle</td><td>text
attribute
bundle</td></tr>
</table>

2.3 Atomicity

Parallel access to shared facilities, e.g., picture making facilities, must be controlled in such a way that only meaningful action sequences will result. For instance, race conditions must be avoided. The environment is also providing the basis for parallel access: each dialogue cell maintains its own environment in a separate data set. Hence every change to the environment can be done at any time. By issuing the command *update environment* the new environment is make visible while extinguishing the previous version. This update function is treated by the underlying graphics system as an atomic action. However, due to the fact that the new description of the environment was completely available beforehand, this atomic action can be completed without unnecessary delays. The various update commands issued concurrently will be queued. Every update will leave the graphics system in situation that further picture making is possible without restrictions. For instance, no workstations will be deactivated or segments left open.

2.4 Radical operations

There are five types of operations on radicals and their environment.

Control functions Four functions are available to create and delete environments : two for creation and deletion of groups, and two for creation and deletion of local states. Obviously the order of calling these functions is important; a group must be created before a local state is and the local state must be deleted before the group is.

Two functions are available to move a radical from one environment to another: one to send it and one to accept it. At the next display this may cause the radical to displayed within a different viewport and with different attributes. Coordinates of primitives will be mapped to the corresponding coordinates in the new viewport automatically.

In fact the only interface with the workstation is the function *update environment*. This causes all radicals in the environment to be displayed according to their current status. No function exists to display an individual radical; if one is, all are displayed. The update mechanism is smart enough to know what needs to be updated and what

not. Display is not automatic on completion of every radical function. This allows the programmer to perform a number of manipulations, before doing a (time consuming) update of the picture.

Changing the environment A set of functions is available to change the attribute values in the local state. As a result of these operations, primitives that take their attributes from the local state may look different at the next display.

Radical manipulation Functions exist to create, delete and copy radicals. Primitives can be added to and deleted from a radical. Every primitive automatically gets a unique name, that can be used to manipulate it later on. A radical can also be concatenated to the end of another, to from a more complex picture. The concatenated radical becomes part of the other: it is not copied.

All radical attributes can be changed, too. Six functions are available to do so. For the change of the transformation matrix of a radical there are three functions available: shifting, scaling and rotation. This is because usually only one of these is done at the same time.

Primitive manipulation Primitive creation and deletion is part of the radical manipulation functions, as no primitives can exist outside radicals. Three other type of functions are available: to change the contents of a primitive (e.g. add a point to a polymarker), to change the type of primitive (e.g. from polyline into fillarea), and to change its attributes (e.g. linetype becomes DOTTED).

When a primitive is added to a radical it gets no individual attributes. Setting and changing of individual attributes is the same, i.e., a change of attributes of a primitive that has none, will cause it to have attributes from then on. Individual attributes can be removed too.

Control	Changing LS	Radical	Primitive
NEW GROUP	LINETYPE	NEW RAD	REPLACE DATA
DEL GROUP	MARKSIZE	SET VIS	CHANGE TYPE
NEW LS	FAREASTYLE	SHIFT RAD	SET LINETYPE
UPDATE		REMOVE PRIM	REMOVE ATTR
SEND RAD		ADD POLYLINE	
ACCEPT RAD			

Inquiry functions A number of inquiry functions are available to aid the programmer in manipulating the radicals. The attribute values of the local state, the list of radicals, the contents and attributes of radicals and the type, contents, and attributes of primitives can be inquired.

2.5 Use of the radical system in the DICE system.

Each dialogue cell has its own picture environment. The pictures owned by a dialogue cell can be composed of subcell picture results, pictures that it got from its parent and pictures that the cell creates itself. When a dialogue cell is activated it first gets a viewport and then a local state. Pictures can move from one environment to another, when they are part of an echo result of a dialogue cell, or when a parent gives it to a subell (by parameter). When a dialogue cell is deactivated the environment is deleted from the system. The dialogue cell run time system performs these and the update functions by calling the corresponding functions of the radical system automatically, so the programmer needs not specify these himself.

3 Implementation

3.1 Discrepancy between radicals and GKS

The radical system must multiplex parallel graphical output onto a single workstation. Every source has its own environment, that determines the interpretation of the output. A workstation has at any time only one "environment" for the output. This includes, for example, the current status of the attribute registers. This is represented in GKS in a so called *global state list*.

To ensure that the output will be shown correctly on the workstation, every local state must be mapped onto the global state each time output from the local state is done. The mapping is part of the output process of a local environment update. The atomicity of this function guarantees that the global state remains "tuned" to the local state during the time of the update.

Radicals are the same as GKS segments for the following reasons:

- segments are stored within a fixed environment. They cannot be moved to another ([6]).
- segments cannot be edited, only added to. Once closed, their contents are fixed. Also, their contents cannot be inquired. Radicals however can be edited at any time. They have no open or closed state.
- the appearance of radicals depend on their environment.

The radical system allows the programmer to specify exactly when the pictures he has created are to appear on the screen. This moment is independent of the moment other (parallel) environments want their contents to be shown. GKS does not give enough control over this. It is possible to set the deferral more to ASTI and do an explicit redraw, but two problems may arise:

- To much may be updated. GKS does not know which pictures have changed since the last update and which may remain the same. The radical system does have this knowledge.

- Updates may be to early, because GKS does not guarantee that updates will only be done when requested (for example because a buffer is full).

3.2 Implementation on top of GKS

The radical system can, and has been, implemented on top of GKS, extended with grouping. An environment consists of a data structure containing all default attribute values and a list of radicals. Within every radical the complete contents are stored. This administration is kept completely separate from GKS.

Only one radical function will actually cause GKS functions to be called: the update. All other functions merely operate on the local administration. Now radical manipulation functions can be implemented as operations on the contents of the radical data structure. Also changing local state attribute values only causes the values in the local state data structure to be changed.

When a radical is displayed, its contents are put in a segment by calling the appropriate GKS output and attribute functions. The attributes are taken from the local state or the radical itself before each primitive is output.

Not every change of a radical will cause the creation of a new segment at the next update. Changing the radical attributes will cause the corresponding segment attribute function of GKS to be called instead. Change of attribute values can be handled without creation of a new segment, because only bundled attributes are used: only the bundle representation needs to be changed. However, when the primitive contents of a radical have been changed, or geometrical text attributes have changed, a new segment must be made at the next update.

By calling GKS only when an update is required, automatically the moments updates occur is controlled. The amount of update is controlled by using the grouping function *redraw group*, that only updates the viewport of the environment. Every radical has a flag, telling weather it has changed, so that a new segment has to be created for it.

3.3 Implementation inside GKS

The implementation on top of GKS is not optimal for the following reasons:

- A considerable part of GKS is not needed when communication with the workstation is done via radicals only. This causes not only a superfluous amount of code to be linked with program, but also the execution of pieces of code that are superfluous under the conditions emerging from the radical system. For example, the radical system will always call the GKS functions properly. However, on every call GKS checks the validity.

- Some functions of the radical system cause a sequence of GKS calls that will do the job, but in a rather clumsy way. If the resulting workstation calls could be done immediately, these functions could be implemented much more efficiently.

For instance, replacing a segment by a new version causes quite a lengthly sequence of GKS calls, e.g.: replace I = delete I; create J; rename J to I; reenter contents of I.

- The functions to create, delete and update groups were outside the scope of the radical system so far, so no control over the precise result (also in time) of these functions could be exerted. In the implementation on top of GKS this caused the need for an extra layer between the radical control functions and the grouping functions.

A next implementation was therefore to incorporate the radical system inside the GKS implementation. Device independence was maintained by keeping the workstation interface the same. All workstation drivers for GKS can therefore be used for the radical system also.

For high level drivers that can handle segments itself, the mapping of radicals onto segments was maintained. Low level drivers, without segment administration, no longer needs segments. The data needed to draw a radical can be fetched from the radicals directly. In both cases GKS does not need to store the segment contents any more.

The administration of the local states was put into GKS now. Instead of letting an environment contain a viewport, the GKS group administration now contains a set of local states, so that every group can see itself what has to be updated.

4 Conclusion

4.1 Example

As a simple example we take a program that has two separate environments, that both want to do graphical output. The first is active in viewport *view1* = 0.0, 0.0, 0.6, 0.6 and the second in *view2* = 0.4, 0.4, 1.0, 1.0. The creation of a local output environment consists of two steps: first a group is created with te function *newgroup*. It is given a window, a viewport and a priority as parameters.

Next a local state can be created with the function *alloc_ls*. It is given two parameters; the group it will operate in, and the parent local state. If the local state is the first to be created, the last parameter must be a NIL, in which case the default attributes as provided by the workstation are assigned to it. Otherwise the attribute values of the parent are assigned to it. Initially no radicals will be in the local environment.

```
1. grp1  = newgroup (win1, viewp1, low_prio); /*  1st viewport
2. grp2  = newgroup (win2, viewp2, hi_prio);  /*  2nd viewport
3. ls1 = alloc_ls (NIL, grp1);                /*  1st  state *
4. ls2 = alloc_ls (ls1, grp2);                /*  2nd  state *
```

Local state ls2 has inherited the attributes of ls1. Now ls1 and ls2 can produce graphical output simultaneously and independently, for example:

```
5. rad = newrad (ls1);                          /* create radical */
6. id = radfill (3, points, rad, ls1);          /* add fillarea  */
7. s_rad_fa_is (ls1, rad, id, HATCH);           /* set  style */
8. ls_update (ls1);                             /* update */

9. s_def_pl_lt (ls2, LTDOTTED);                 /* set line type */
10. rad = newrad (ls2);                         /* create radical */
11. id = radpol (9, points, rad, ls2);          /* add polyline */
12. ls_update (ls2);                            /* update */
```

Lines 5 through 8 are part of the process that create grp1 and ls1, lines 9 through 12 are part of the other. The result will look like figure 2, no matter what time either process executes the statements.

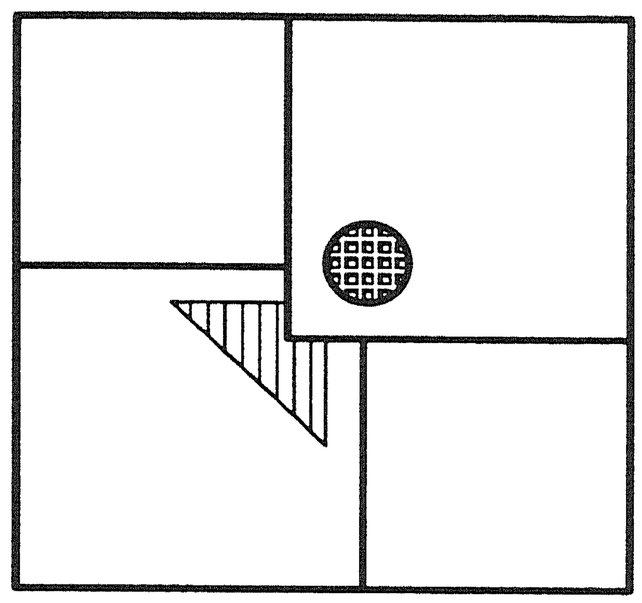


Figure 2:

Now ls2 may decide to pass the radical it created to its parent, by calling *rad_out_ls (rad, ls2)*, and passing a point to the radical to ls1. Ls1 can accept it by calling *rad_into_ls (rad, ls1)*. In between these two functions the radical is not part of any environment and thus would not be displayed at an update. The result of the move looks like figure 3.

In the context of the dialogue cell system, the initial set up of the environment is done automatically; the programmer only needs to specify the viewports that are to be used. Also, the programmer only has to specify that *rad* is to be returned to its parent. The dialogue cell run time system will then call the radical exchanging functions automatically at the proper time. Only the code in figure 5 is to be specified in the trigger rules, using a convenient syntax.

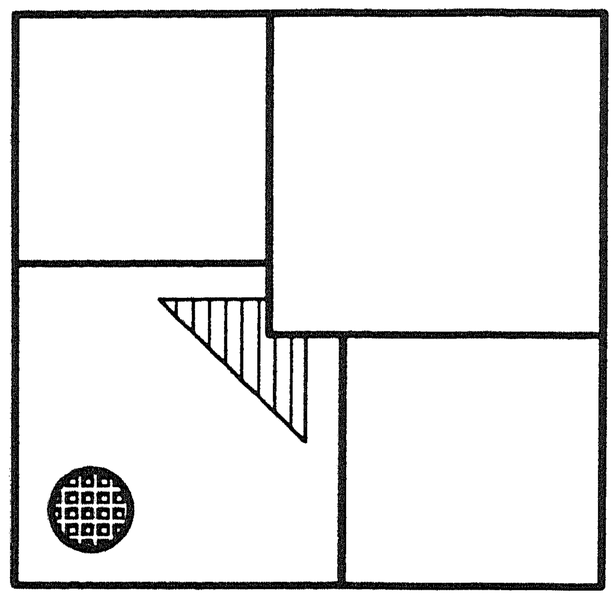


Figure 3:

4.2 Performance

We expect the radical system to run on rather advanced workstations. Even on this kind of configuration, the time to perform the radical functions is dominated by the actual drawing time, i.e. the time spent in the workstation driver.

As an example, take the program above. To reduce the effects of start up and closure of the workstation, the code of figure 5 was executed 10 times. On our installation 91% of the time to produce the pictures was workstation time. As times vary strongly with hardware, maybe the best we can do is to compare the results found with ordinary GKS program using grouping, with comparable functionality. It turns out that for the example above, the radical system was 19% faster then the GKS program, if we subtract workstation time from the total time.

As for code size: the GKS library could be reduced by 68%. The object of the small example above was 5% smaller then the GKS program. However, the source code was 68% smaller, because most of the work in the GKS program is taken over by the radical system.

4.3 Experience and future

As the dialogue cell system is still in an experimental phase, the experience with the system is limited. However, it has already become clear that the functionality of the radical system allows the programmer to describe the visual effects of a dialogue very precisely. The localization of the graphical output allows for a description of the graphical output of dialogue cells: no knowledge of the influence of a library dialogue cells on the graphics system is needed. As many aspects of the output produced are implicitly accounted for in the semantics of the radical system, the specification of pictures is very concise.

A future step we planned is not to interface with the existing GKS workstation drivers any more, but change these so the fit our purposes, to improve performance even more. Currently, the system can handle only output to one interactive workstation at once. A straightforward extension to *n* workstations will be implemented shortly. In this situation each local state and group will exist on one workstation only.

An extension to three dimensional output is envisaged also. This will require the extension of the grouping concept to 3D as well. In this event we consider building the radical system on top of PHIGS [4].

Because the radical system runs in a hierarchical environment (e.g. DICE) one may expect that the PHIGS hierarchy may be a more appropriate basis for the implementation than the linear segment organization of GKS. Also, the radical editing should be easier, given the PHIGS structure editing functions. However the difference in the semantics of the respective hierarchies as well as the parallelism greatly reduces the possibilities for taking advantage of the greater functionality of PHIGS. Nevertheless, an implementation on top of PHIGS may be expected to be slightly easier. Against the more direct implementation mentioned here it is expected that the performance will be poor (about as poor as that of GKS). Some examples may illustrate this:

- inheritance in the radical system is one time versus continuous in PHIGS, so the PHIGS hierarchy cannot be used to implement the environment hierarchy of the radical system directly.

- in PHIGS there must be one root structure per environment. Grouping like optimalizations are not within the scope of PHIGS.

- PHIGS does not provide an (obvious) function for exchanging radicals between environments.

References

[1] H.G. Borufka, H.W. Kuhlmann, and P.J.W. ten Hagen. Dialogue Cells: A Method for Defining Interactions. *IEEE Computer Graphics and Applications*, pages 25–33, July 1982.

[2] T. Hagen, P.J.W. ten Hagen, P. Klint, and H. Noot. ILP, Intermediate Language for Pictures. Technical Report IW68/1977, Mathematical Centre, Amsterdam, 1977.

[3] Internation Standard Organization, Geneva. *Information processing systems — Computer graphics — Graphical Kernel System (GKS) functional description (ISO IS 7942)*, 1985.

[4] Internation Standard Organization, Geneva. *Information processing systems — Computer graphics, Programmer's Hierarchical Interactive Graphics System (PHIGS) (ISO IS 9592)*, 1988.

[5] H.J. Schouten. Basic dialogue cells. Technical Report CS-R87XX, Centrum voor Wiskunde en Informatica, Amsterdam, 1987 (to appear).

[6] P.J.W. ten Hagen and M.M. de Ruiter. Segment grouping, an extension to the graphical kernel system. Technical Report CS-R8623, Centrum voor Wiskunde en Informatica, Amsterdam, 1986.

[7] P.J.W. ten Hagen and W. Eshuis. Value mapping in dialogue cells. Technical Report CS-R87XX, Centrum voor Wiskunde en Informatica, Amsterdam, 1987 (to appear).

[8] P.J.W. ten Hagen and R. van Liere. Introduction to dialogue cells. Technical Report CS-R8703, Centrum voor Wiskunde en Informatica, Amsterdam, 1987.

Faster Phong Shading via Angular Interpolation

*A.A.M. Kuijk, E.H. Blake**

Abstract

One of the most successful algorithms that brought realism to the world of 3D image generation is Phong shading. It is an algorithm for smooth shading meshes of planar polygons used to represent curved surfaces. The level of realism and depth perception that can be obtained by Phong shading is attractive for 3D CAD applications and related areas. However, too high per pixel computation costs and/or artifacts, introduced by some of the more efficient evaluation methods and apparent only when displaying moving objects, are major factors that blocked the common usage of Phong shading in highly interactive applications.

In this paper we present angular interpolation for Phong shading planar polygons. Angular interpolation was a method especially designed to meet requirements as imposed by special purpose hardware we developed [8], but turned out to be generally applicable. The angular interpolation method appears to be very efficient and reduces artifacts when displaying moving objects. Ideally a shading algorithm imposes no need for subdivision of patches as presented by the solid modelling system. Shading calculation via angular interpolation yields such an ideal algorithm. We will describe two alternative evaluation methods that trade off evaluation cost against level of accuracy. They both can handle light source and view point at arbitrary distances, but differ in level of accuracy. As a consequence these alternative evaluation methods do impose restrictions on the topology of patches and light sources. However, generally, the limitations imposed by these alternative shading methods are much more liberal than the limitations on patch size imposed by the geometry.

The most economic evaluation method we present can incrementally compute the colour intensity along a scanline by two additions per pixel. The methods presented are generally applicable and can easily be implemented in hardware.

Key Words & Phrases: image synthesis, shading, angular interpolation, spherical geometry, quadratic approximation, quaternions.

---

* Centre for Mathematics and Computer Science (CWI)
Department of Interactive Systems
Kruislaan 413, 1098 SJ Amsterdam, The Netherlands
Email: fons@cwi.nl edwin@cwi.nl

Introduction

Phong shading is one of the most successful algorithms for obtaining a high degree of realism in computer generated images. This shading model is often used to shade planar polygonal approximated surfaces smoothly. It not only has ambient and diffuse intensity components but incorporates a specular reflection component that produces a highlight as caused by reflection of shiny surfaces [5, 10]. The surface reflectance dependent approximation of this specular component as proposed by Phong is based on empirical observation. The Phong shading model implies that at every pixel the diffuse component $\cos \alpha$ and the reflection component $\cos^n \beta$ has to be evaluated. In general both α and β vary across the surface to be shaded.

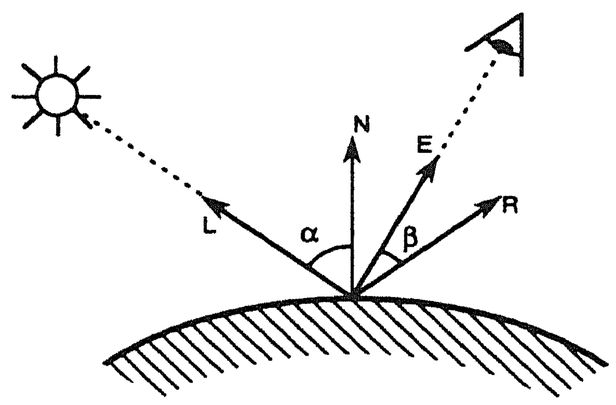


Figure 1: The Phong shading model comprises a diffuse component $\cos \alpha$ and a specular component $\cos^n \beta$. α is the angle between the surface normal $\vec{N}$ and the direction of the light source $\vec{L}$, β is the angle between the direction of reflection $\vec{R}$ and the direction of the viewpoint $\vec{E}$ and n is a coefficient depending on the reflectivity of the surface.

Phong shading would be very attractive for interactive 3D CAD applications. The level of realism that can be obtained with it will give a better depth perception and may give a better idea of the final result of the design during the design process. In spite of this, common introduction of Phong shading in highly interactive CAD applications for smooth shading of planar polygon meshes did not occur, mainly because of the computation intensiveness of the method. Another deficiency are artifacts, introduced by some of the more efficient evaluation methods [2]. These artifacts may be almost invisible for static images, but become disturbingly apparent when displaying moving objects. This may be reduced by reducing the polygon size, but this strategy would increase the computational demands and data transportation. As a result, the Phong shading model has so far been a widely accepted shading model for post design visualisation; the interactive design session –supported by a less demanding shading model– is followed by a request to show ''what it really looks like''.

In order to make Phong shading applicable for highly interactive applications, the basic problems that have to be solved are: ''how do the angles α and β vary across the polygon?'' and ''given this variation how to evaluate the intensity components in an as economic way as possible?''

In this paper we will show an efficient and high quality method for Phong shading planar polygonal meshes. In the first section, we will focus on answering the question raised above: ''how do the angles α and β vary across the polygon?''

We will discuss equi-angular interpolation for which no normalisation per pixel is needed as opposed to the traditional vector interpolation. Two alternative interpolation methods will be presented, both capable of handling light source and/or viewpoint at finite or infinite distance. The result of these methods will be an expression relating the cosines of the Phong model to one or two (depending on the method) angles incremented linearly along a scanline.

In the second section, the question: "how to evaluate the diffuse and specular components in an as economic way as possible?" will be answered.

1. Interpolation Across Polygons

For curve approximation by planar polygon meshes, information of the curvature the planar polygon has to approximate is represented by a surface normal vector $\vec{N}$ specified at each vertex of the polygon [3], as shown in Figure 2. From this, the normal along the edges and at each point inside the polygon has to be interpolated. In general, not only the normal $\vec{N}$, but also the direction of the light source $\vec{L}$, the direction of the eye $\vec{E}$ and the direction of reflected light $\vec{R}$ is dependent on the position. Since $\vec{N}$ is obtained by interpolation, these vectors usually also are found by interpolation from the values calculated at each vertex, rather than by exact calculation at each pixel.

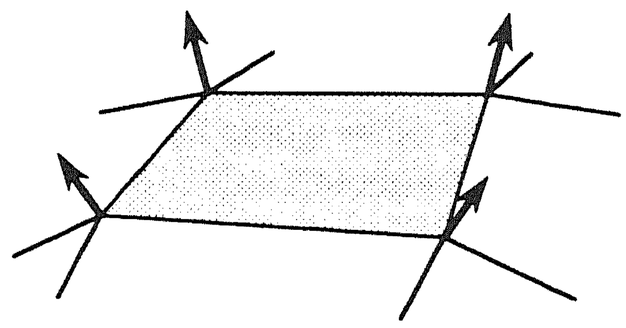


Figure 2: Approximation of curved surfaces by planar polygon meshes. For the purpose of shading calculations, the surface normal is specified at each vertex.

1.1. Vector Interpolation

The traditional vector interpolation method is based on the assumption that along a path from A to B a vector $\vec{N}_t$ at $P = A*(1-t) + B*t$ can be calculated by $\vec{N}_t = \vec{N}_A*(1-t) + \vec{N}_B*t$ with $0 \le t \le 1$ and $\vec{N}_A$ and $\vec{N}_B$ being the normals at A and B respectively. This interpolation is performed along the edges first. Using the thus obtained $\vec{N}_P$ and $\vec{N}_Q$ the normal along a scanline can be similarly interpolated, as shown in Figure 3.

To have an efficient method, it has to be possible to relate the increments directly to the evaluation of the intensity components, $\cos\alpha$ and $\cos^n\beta$. This is why vector interpolation became so widespread, the spatial increments dx, dy and dz of the

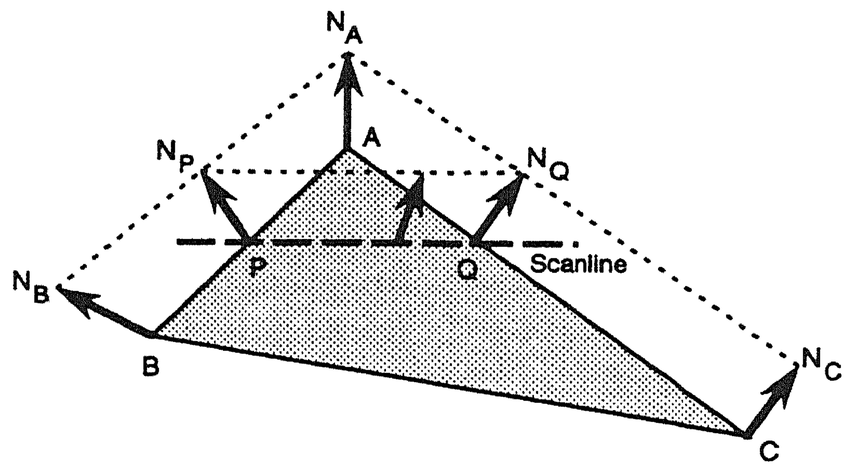


Figure 3: Interpolation of a vector across the polygon in two steps: first the vector is interpolated along the edges AB and AC, next the interpolated vectors $\vec{N}_P$ and $\vec{N}_Q$ thus obtained are used for interpolation along the scanline through PQ.

vectors along the scanline are closely related to the vector coordinates needed for evaluation of the scalar product, which makes it possible to incrementally calculate the intensity components ‡.

Figure 4 however, shows that the resulting interpolated vector has a varying length and varying angular increment. This is a result of making equal steps along a chord joining the two vectors $\vec{N}_A$ and $\vec{N}_B$. Due to the varying length, renormalisation of the interpolated vector has to be done. The varying angular increment introduces an orientation dependency of the highlight. This results in a jitter of the highlight when rotating, for example, a cylindrical object around its main axes. It also causes Mach band effects because the intensity variation is not smooth. Generally these deficiencies are avoided by putting restrictions on the maximum angular variation of the vectors across the polygon; in other words by reducing the polygon size. It may be clear that this restriction is not without cost.

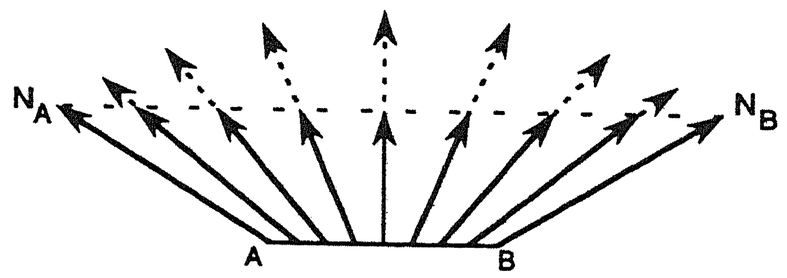


Figure 4: As a result of vector interpolation, successive vectors vary both in length and in angular increments. Renormalisation is needed when the interpolated vector is used in a scalar product.

A common assumption of most of the more efficient approaches known so far is that either the light source or the viewpoint is taken to be at infinity (sometimes even both). This means that $\vec{L}$ and/or $\vec{E}$ is taken to be a constant. This may simplify the calculations involved, but reduces the realism of the picture (e.g. if light source and eye are at infinity, planar surfaces are shaded with a constant intensity).

---

‡ $\cos\alpha = (\vec{N}\cdot\vec{L})$ and $\cos^n\beta = (\vec{R}\cdot\vec{E})^n$, provided the vectors are normalised.

Bishop and Weimer [1] published a method for Phong shading which requires a quadratic polynomial for calculating $(\vec{N}\cdot\vec{H})$ †, provided the curvature is less than 60 degrees. The $\vec{L}$ is taken to be a constant, but $\vec{E}$ is allowed to vary. Exponentiation of $(\vec{N}\cdot\vec{H})$ is done via table lookup.

An approach which uses bicubic approximations for the scalar products was published by Shantz and Sheue-Ling Lien [11]. They assumed $\vec{L}$ fixed and used lookup tables for the exponentiation.

A method also assuming $\vec{L}$ fixed was published by Deering [4]. A highly pipelined architecture interpolates and normalises $\vec{N}$ and $\vec{E}$ and lookup tables are used for square root and exponentiation functions. Of particular interest in this publication was that the approach Bishop proposed was rejected because the break-even point of the method at 10 pixels was considered too high!

1.2. Angular Interpolation

The basic idea of angular interpolation is that the angular rotation of a directional vector ($\vec{N}$, $\vec{L}$ or $\vec{H}$) is linearly related with the position along a straight line across the polygon (see Figure 5). Vectors interpolated according to this assumption have a constant length and are all in one plane; the plane spanned by the start and end vector. An elegant way of calculating these angularly incremented vectors using quaternions is described in [7, 9]. Analogous to the vector interpolation method, interpolation will be done in the two steps shown in Figure 3; first the vector is interpolated along the edges of the polygon, next the resulting vectors are used for interpolation along the scanline.

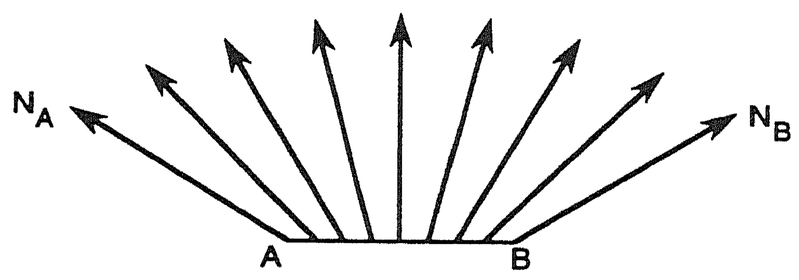


Figure 5: Angular interpolation. Successive vectors are found by equi-angular rotation using quaternions. As a consequence their length remains constant.

When observing the variation of one of the vector couples along a path across a polygon you can imagine a ''dance'' these two vectors perform. Since both vectors vary independently, the result is that they may circle around each other, get closer or move away. This ''dance'' along a straight path is completely determined by two linearly varying angles, one for each vector.

---

† $\vec{H}$ is the vector in the direction of the highlight, that is the vector halfway between $\vec{L}$ and $\vec{E}$. This vector is often introduced to replace $(\vec{E}\cdot\vec{R})^n$ by $(\vec{N}\cdot\vec{H})^n$. Note however, that the angle between $\vec{N}$ and $\vec{H}$ is about half the angle between $\vec{R}$ and $\vec{E}$, a difference discussed in [6] that can more or less be corrected by adjusting n.

We have to find a relation between these angles incremented along the scanline and the evaluation of the intensity component $\cos\alpha$ ‡ . We will show how to find this relation based on the following example of interpolation across a triangle.

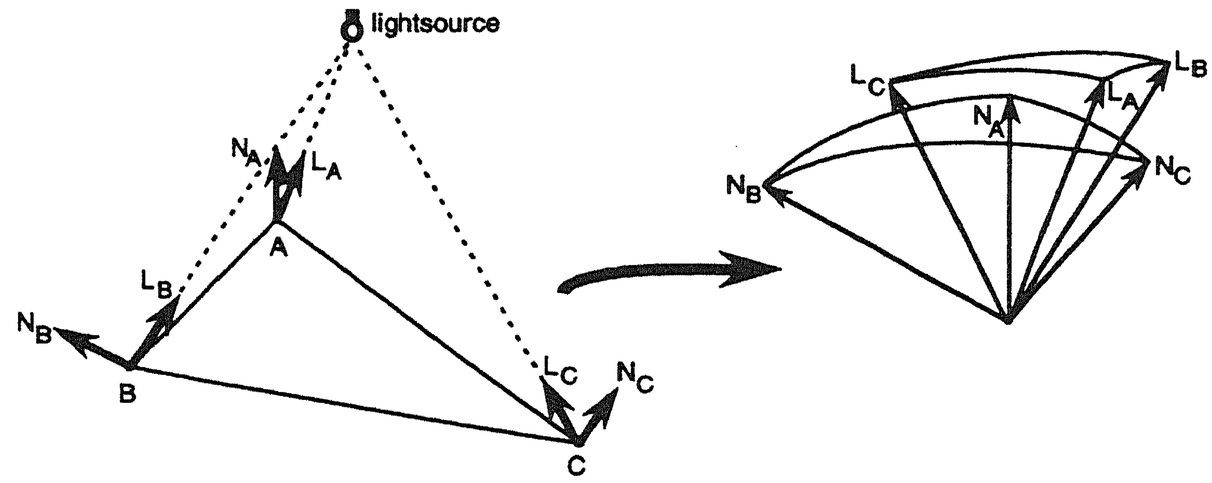


Figure 6: The range of the directional vectors $\vec{N}$ and $\vec{L}$ across the triangle ABC is indicated by two spherical triangles $N_AN_BN_C$ and $L_AL_BL_C$ on the unisphere.

The normal vector and the direction of the light source at the vertices of a triangle are shown in Figure 6. For each of the two vectors there is a mapping of the polygon on the unisphere indicating the range of that vector across the polygon. If for instance the light source is at infinity the spherical triangle $L_AL_BL_C$ is reduced to a point.

A scanline across the triangle is mapped on two circular paths, indicating the variation of the vectors $\vec{N}$ and $\vec{L}$ along this scanline (see Figure 7). These paths, from $\vec{N}_1$ to $\vec{N}_2$ and from $\vec{L}_1$ to $\vec{L}_2$, are each part of a great-circle. These two great-circles intersect at S. Let γ be the angle between the two great-circles, n be the angle between S and $\vec{N}_1$, l be the angle between S and $\vec{L}_1$. Then having t linearly varying along the scanline from t=0 at startpoint 1 and t=1 at endpoint 2, we define n_t to be t*(the angle between $\vec{N}_1$ and N_2) and l_t to be t*(the angle between $\vec{L}_1$ and $\vec{L}_2$). $\vec{N}_t$ and $\vec{L}_t$ are the vectors $\vec{N}$ and $\vec{L}$, interpolated along the scanline.

With this we have a spherical triangle SN_tL_t, dependent on t. For this triangle a standard formula, given by spherical trigonometry, leads us to the following relation between α_t (i.e. the angle between $\vec{N}_t$ and $\vec{L}_t$) and the linearly incremented angles n_t and l_t:

$$\cos\alpha_t = \cos(n+n_t)\cos(l+l_t) + \sin(n+n_t)\sin(l+l_t)\cos\gamma \qquad (1)$$

Note that n, l and γ are constant along the scanline. Here we have an expression that directly produces the diffuse intensity component. The specular component can be found similarly, but needs raising to the power of the specular exponent as well. This expression already can be calculated more efficiently than calculation of the intensity by vector interpolation, which inherently involves renormalisation. However, for our particular application it was still too complicated for efficient

---

‡ In the following we will discuss the vector couple $\vec{N}$ and $\vec{L}$ needed for the diffuse term, only; interpolation of the vector couple that produces the specular term, i.e. $\vec{N}$ and $\vec{H}$, is identical.

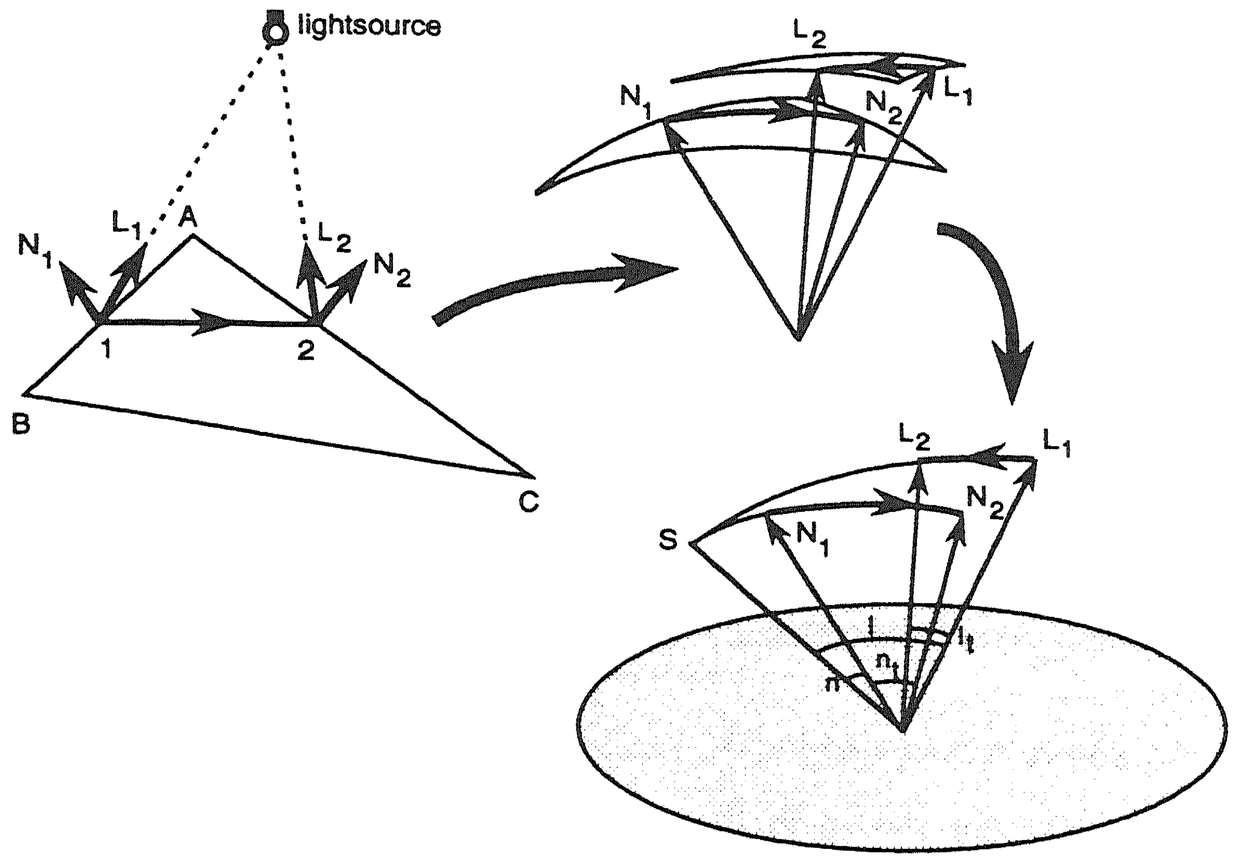


Figure 7: Interpolating $\vec{N}$ and $\vec{L}$ along a scanline is related to two paths along two great-circles. The intersection point S of these great-circles forms one point of a spherical triangle SNL. This triangle relates the two interpolated angles n_t and l_t with the intensity component cos α along the scanline.

evaluation. We developed two alternative approximation methods for this which both result in a simpler expression, and produce good results under mild restrictions. In one method the rotation of one vector is decomposed in two perpendicular components. The other method combines the two varying vectors in one.

Decomposition method.

Expression (1) will reduce to a product of two cosines, when the two great-circles are perpendicular; in that case cos γ is zero. The idea is to decompose the rotation of one vector in two rotation components, one parallel and one perpendicular to the rotation of the other vector.

Let a be the angle between the two great-circles through $\vec{L}_1$ and $\vec{L}_2$ both perpendicular to the plane in which $\vec{N}$ rotates as shown in Figure 8. Rotating $\vec{N}_2$ as well as $\vec{L}_2$ with angle a around the axis through P_N (the pole of the plane through $\vec{N}_1$ and $\vec{N}_2$), will produce the two vectors $\vec{N}'_2$ and $\vec{L}'_2$. Interpolation of $\vec{N}$ can now be done from $\vec{N}_1$ to $\vec{N}'_2$ and interpolation of $\vec{L}$ from $\vec{L}_1$ to $\vec{L}'_2$ as indicated in the figure. In doing so, we added the rotation component of $\vec{L}$ around the axis P_N to the rotation of $\vec{N}$ around that axis. As a result, for $\vec{L}$ only a rotation component perpendicular to the plane through $\vec{N}_1$ and $\vec{N}_2$ remains; $\vec{L}_1$ and $\vec{L}'_2$ are both on the same great-circle through P_N that intersects the great-circle through $\vec{N}_1$ and $\vec{N}'_2$ at S'.

Let n_1 be the angle between S' and $\vec{N}_1$, n_2 the angle between S' and $\vec{N}'_2$ and let l_1 be the angle between S' and $\vec{L}_1$ and finally l_2 the angle between S' and $\vec{L}'_2$. Then for the right angled spherical triangle $N_t S' L_t$ we can write the following equation:

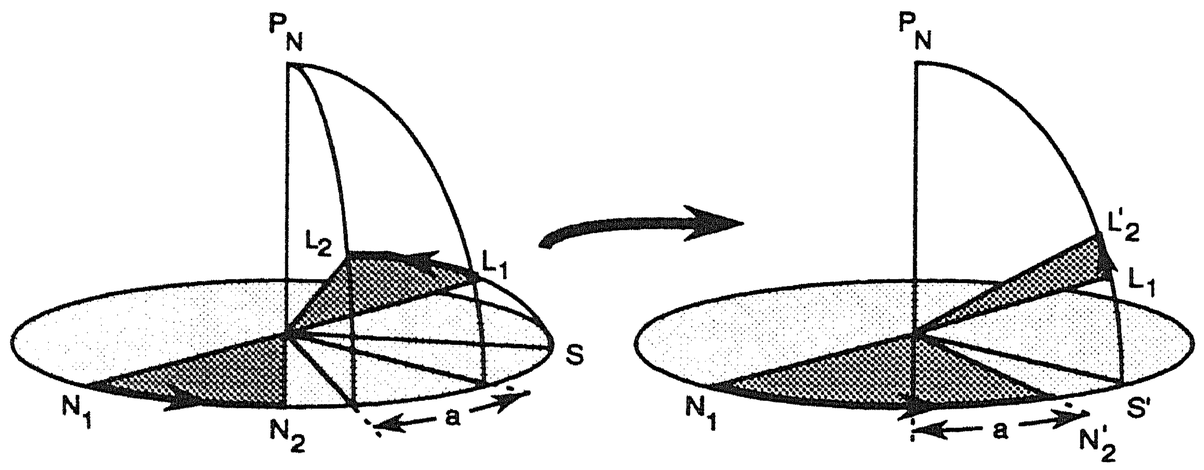


Figure 8: Decomposition of the rotation of one of the two independent rotating vectors results in two perpendicular rotated vectors.

$$\cos \alpha_t = \cos (n_1 + t (n_2 - n_1)) \cos (l_1 + t (l_2 - l_1)) \tag{2}$$

The spherical triangle shown in Figure 9 will clarify the implicit assumption we made using this method. For this spherical triangle the following equations hold:

$$\cos l_2 = \frac{\cos (l_t + l)}{\cos (a+b)} \tag{3.a}$$

$$\cos \gamma = \frac{\tan (a+b)}{\tan (l_t + l)} \tag{3.b}$$

$$\sin \gamma = \frac{\sin l_2}{\sin (l_t + l)} \tag{3.c}$$

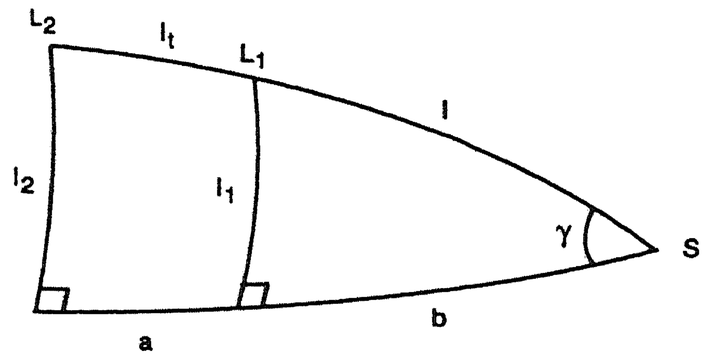


Figure 9: From this spherical triangle it can be seen that in general, linearly incrementing both a and l_2 is not equivalent to linearly incrementing l_t.

Using (3.a) and (3.b), we can prove that for t=0 and t=1, (2) will produce the same result as (1). More of interest however, are the relations between a and l_t and between l_2 and l_t as given by (3.b) and (3.c). These relations show that in general linearly incrementing the angles $n_2 - n_1$ (with a hidden in it) and $l_2 - l_1$ is not equivalent with linearly incrementing n_t and l_t. This however is an acceptable first order approximation. Only extreme situations such as an extremely nearby light source may affect the result. We could not produce a visible difference in a realistic situation up to now.

Reducing to one vector.

Instead of interpolating the two independently varying vectors $\vec{N}$ and $\vec{L}$ we would like to interpolate only one vector, without loosing the generality of the method, that is, without assuming one of the vectors fixed. Realising that only the relative position of both vectors $\vec{N}$ and $\vec{L}$ is of interest, not their absolute position, we define a vector $\vec{V}$ at each vertex of the polygon that is found by rotating $\vec{L}$ around the same axis and with the same angle as needed to rotate $\vec{N}$ to be aligned with a fixed vector $\vec{O}$ (e.g. $\vec{o}=\begin{bmatrix}0\\0\\1\end{bmatrix}$).

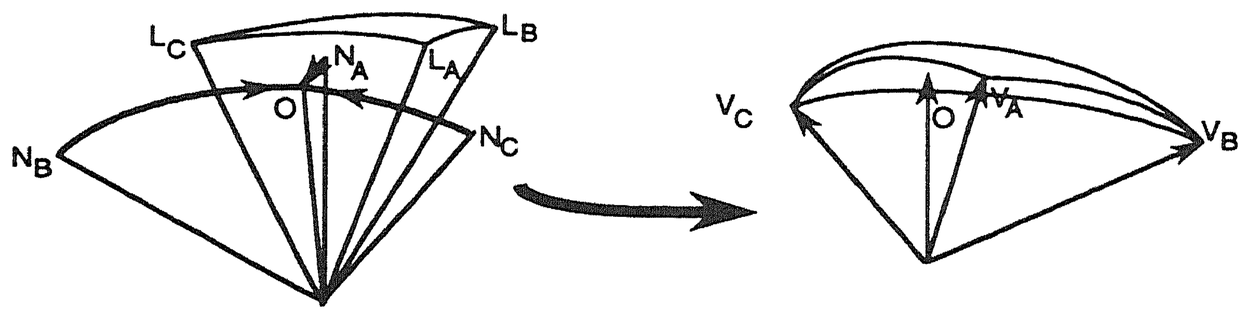


Figure 10: Interpolation of two independently varying vectors $\vec{N}$ and $\vec{L}$ can be replaced by interpolation of one vector $\vec{V}$. For each vertex, $\vec{V}$ can be found by applying the same rotation on the vector $\vec{L}$ as needed to align the corresponding $\vec{N}$ with the reference vector $\vec{O}$.

Using this method, we have specified a vector $\vec{V}$ which is the only vector that remains to be interpolated across the polygon and the quest for the angle between $\vec{N}$ and $\vec{L}$ can be replaced by the search for the angle between $\vec{V}$ and $\vec{O}$.

We do this by constructing a right angled spherical triangle V_tOS. This spherical triangle (illustrated in Figure 11) has one vertex at $\vec{O}$ and one vertex at $\vec{V}_t$ ($\vec{V}_1 \le \vec{V}_t \le \vec{V}_2$). The third vertex $\vec{S}$ lies on the great-circle through $\vec{V}_1$ and $\vec{V}_2$ such that V_tS is perpendicular to SO †. $\vec{S}$ defines two constant angles; s, the angle between $\vec{S}$ and $\vec{O}$ and v, the angle between $\vec{S}$ and $\vec{V}_1$.

Let v_t be the angle between $\vec{V}_1$ and $\vec{V}_t$, linearly incremented along the scanline. Then, α_t, the angle between $\vec{V}_t$ and $\vec{O}$ can be expressed in terms of the angle v_t with the formula:

$$\cos \alpha_t = \cos (v+v_t) \cos s \qquad (4)$$

With this expression we succeeded in relating the intensity component $\cos \alpha$ to one angle v_t, linearly varying along a scanline. By introducing the vector $\vec{V}$ we simplified the interpolation and at the same time eliminated the difference between directional and positional light sources [3].

---

† We do not have to worry about the two degenerate situations; 1) $\vec{O}$ on the great-circle through $\vec{V}_1$ and $\vec{V}_2$ and 2) $\vec{O}$ perpendicular to the plane through the great-circle because these are covered by the resulting expression equally well.

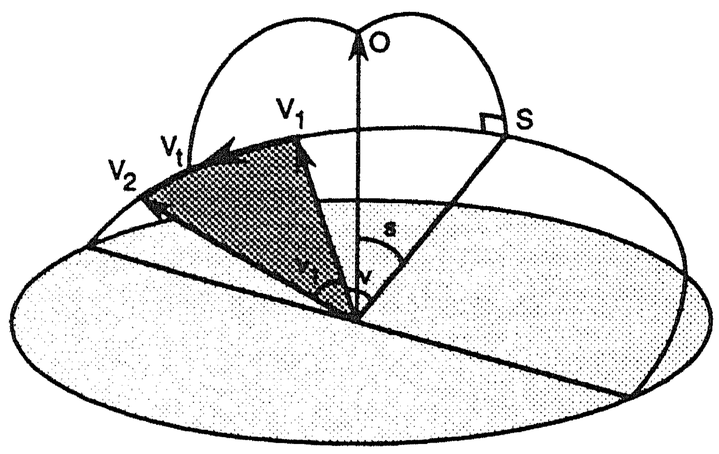


Figure 11: A right angled spherical triangle SOV relates the linearly incremented angle v_t with the angle between $\vec{V}_t$ and $\vec{O}$.

How does this method relate to the previous one? In the previous method, only the parallel component of the rotation of one vector was added to the rotation of the other vector, whereas this method adds the full rotation of one vector to the rotation of the other. As a consequence this methods imposes more restrictions on the maximum acceptable variation of the vectors across the polygon. In practice however, these limitations are easily met. Differences between the methods became visible when vector rotations on the polygon were about 90°. This is much more than what is to be expected with curved surface approximation by planar polygon meshes.

Since especially for highly reflective surfaces (i.e. a high reflectivity exponent for the specular term) the specular highlight will have a limited range, checking the range of $\vec{V}$ with respect to $\vec{O}$, can give at an early stage insight whether or not the specular term will contribute at all. This check can be performed at polygon level as well as at scanline level.

2. Evaluation of the Intensity Components

In the previous sections we presented angular interpolation of vector couples which yielded the linear expressions (1), (2) and (4) for the cosine of the angle between them along a scanline. Choosing one of these expressions, we can evaluate the diffuse component. For the specular intensity component it takes raising to the power of the specular exponent as well. Evaluation of the cosine as well as exponentiation can be done using standard function libraries or lookup tables. The latter may be the fastest solution for software implemented evaluation, but neither of these are attractive for highly parallel VLSI implementation. Lookup tables in particular would require large on chip storage (> 8K bytes for exponentiation [1]).

We had to find a more acceptable approximation to evaluate the general term $\cos^n\theta$ which is present in both (2) and (4). The approximation should meet the following criteria:

- easy to evaluate incrementally by forward differencing
- high quality, that is: no discontinuity in magnitude or slope of intensity.

An approximation by means of a second order Taylor series would be easy to evaluate. This may produce a quadratic function that correctly describes the behaviour near the centre, but the grave discontinuity in slope of the intensity where the quadratic function gets to zero will cause an unacceptable Mach band effect. Given this, we tried to find a better evaluation method.

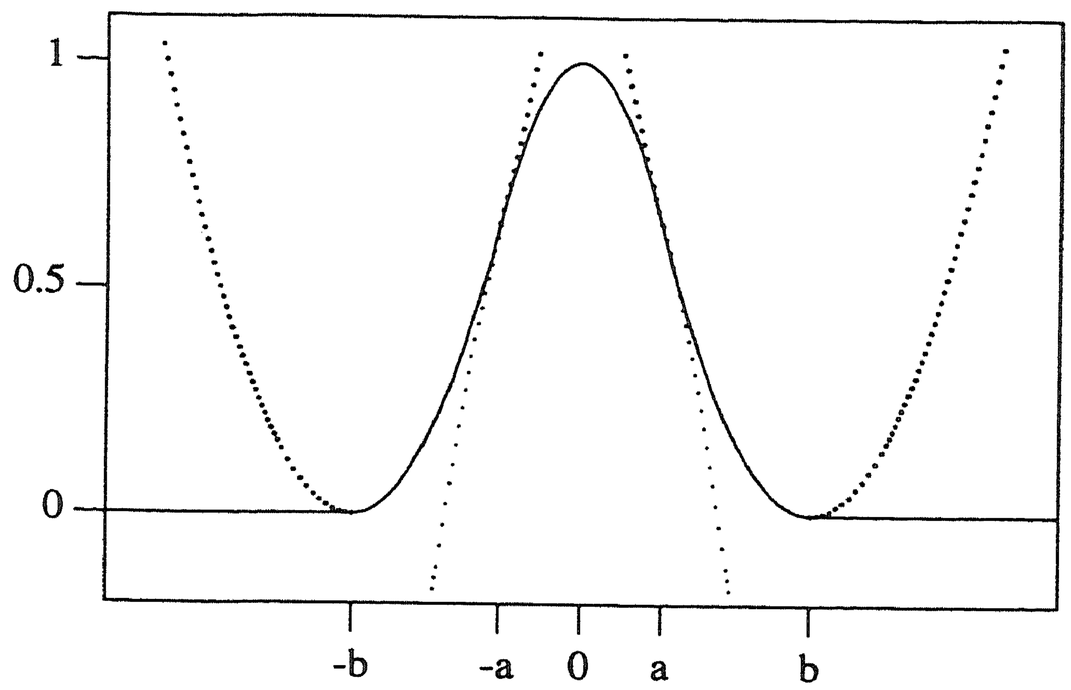


Figure 12: Three successive quadratic curves fitted on $\cos^3\theta$.

Evaluation of $\cos^n\theta$ by quadratic curve approximation.

Due to hardware limitations imposed by the technology [8], we could not allow for a higher than second order polynomial. With this in mind and looking at the shape of $\cos^n\theta$ we found that a combination of three successive quadratic curves would closely fit that shape (see Figure 12). The condition that no discontinuity in magnitude or slope of intensity is allowed is met by the following function:

$$f(\theta) = \begin{cases} 0 & \text{if } \theta \le -b \text{ or } b \le \theta \\ \dfrac{(\theta+b)^2}{b(b-a)} & \text{if } -b < \theta < -a \\ 1 - \dfrac{\theta^2}{ab} & \text{if } -a \le \theta \le a \\ \dfrac{(\theta-b)^2}{b(b-a)} & \text{if } a < \theta < b \end{cases}$$

With the two parameters a and b, this function has sufficient freedom to produce a best fit for the function $\cos^n\theta$. As can be seen in Figure 12, $-a$ and a are the points where the quadratic function changes its second derivative and $-b$ and b limit the

region for which the function is nonzero. This latter quantity is very useful; it is nice to know the range where $f(\theta) \neq 0$.

By least square fitting, for each n ($1 \leq n \leq 125$) a pair (a, b) was found. Because there are only a limited number of pairs, these values can easily be put in a small lookup table. However, looking at the values of a and b as a function of n it was clear that their relation could be described functionally as well. We found the following relations:

$$a = \frac{n+5.6}{n(0.09n+5.2)}$$

$$b = \frac{n+65.0}{5.0n+31.7}$$

Note that the values a and b are dependent on n only, so they can be considered as an attribute of the polygon, replacing the specular exponent n.

Results of this evaluation method for several values of n can be seen in Figure 13. Given the fact that the specular component as proposed by Phong is based on empirical observation, it is clear that we can be satisfied with the result. Even more so because evaluating the diffuse component with this method ($n=1$) has the pleasant side effect that it removes Mach banding. This Mach banding is caused by the sharp discontinuity in slope when the cosine reaches zero. At that point the slope changes from its maximum to zero [6]. The method proposed here does not have such a discontinuity for $n = 1$, as can be seen in Figure 13. The smooth transition of this evaluation method agrees with reality where diffraction and a finite sized light source cause a smooth transition as well.

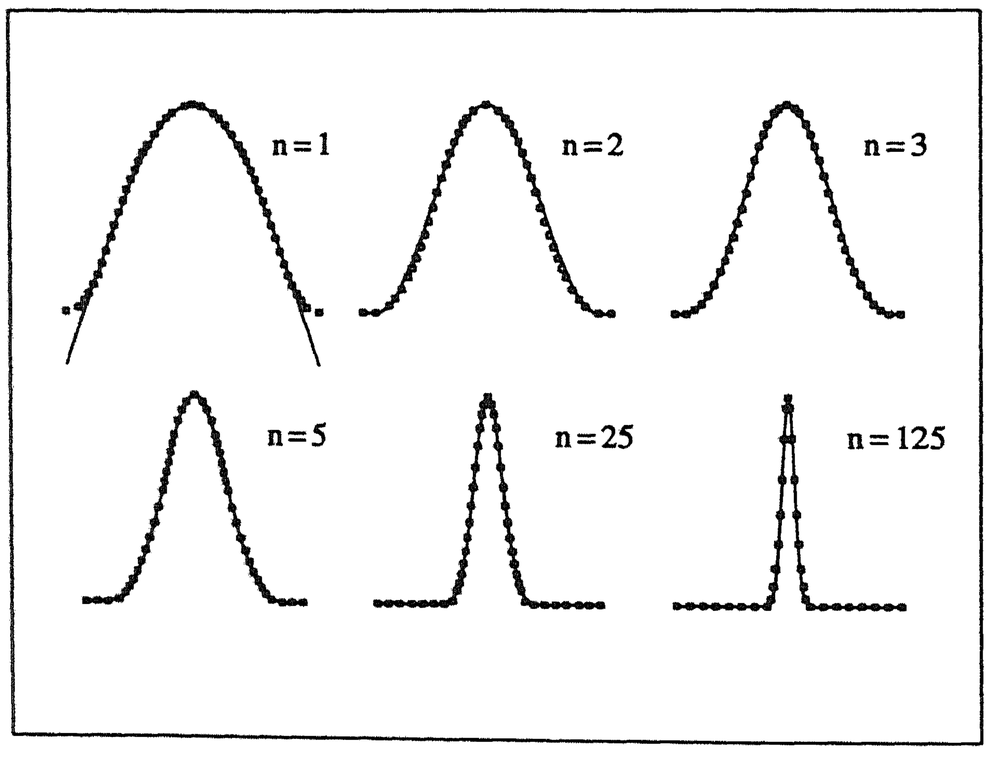


Figure 13: Results of the quadratic approximation of $\cos^n\theta$ for several values of n. The solid lines represent $\cos^n\theta$ whereas the boxes show the approximated values.

Evaluation of (4) can be done directly with this method. Per scanline and per light source for the diffuse and the specular term the constant angles s and v have to be

calculated. For each of these, along the scanline the quadratic function $f(\theta)$ can be evaluated using forward differencing. This costs two additions per pixel. At the points where the transition from one quadratic section to the next has to be made, the second order derivative has to be changed; the function value and the first order derivative are both continuous at those points.

Instead of evaluating the quadratic functions per light source and for the diffuse and specular components individually it can be noted that adding these quadratic functions before performing the forward differencing along the scanline, would still result in a quadratic function, although composed of more sections. It still holds that the function value and the first order derivative are both continuous, so that only the second order derivative has to be changed at transition from one quadratic section to the next. This can be cheaper than evaluation of the diffuse and specular terms per light source individually.

Evaluation of (2) needs evaluation of two individual cosines which have to be multiplied. Dependent on the situation, in particular on the hardware available, multiplication can be done per pixel or per scanline, the latter resulting in a fourth order polynomial.

3. Conclusions & Future work

The angular interpolation presented has shown to result in a very efficient high quality evaluation of the Phong shading algorithm. Although the exact method can already be considered to be efficient as compared to vertex interpolation methods, we presented two even more efficient evaluation methods that operate satisfactorily without putting severe restrictions on viewpoint or light source distance. They differ in level of accuracy traded off against corresponding costs. The limitations on patch size and light source distance imposed by the more efficient evaluation methods due to the approximations made, generally are much more liberal than the limitations imposed by the geometry.

The quadratic evaluation presented can easily be implemented in hardware. Forward differencing would reduce the evaluation cost to two additions per pixel. Using this method for the diffuse term has the side effect of removing the Mach banding on the edge of the diffuse area as normally present when using a cosine or scalar product.

On future work: More use can be made of coherence between scanlines. This would produce a more efficient way of evaluation of the angles which are constant along a scanline.

We have a satisfactory method to evaluate a cosine raised to a power, but this is a particular case. We are currently studying how to extend this to more general situations such as the product of a linear function and $\cos^n$ and the product $\cos^n\theta\cos^m\phi$. These generalisations are needed when implementing spot light sources or anti-aliasing.

Acknowledgements

We acknowledge the national foundation STW who is funding a project that as side effect faced us with this topic. We acknowledge Kapila Jayasinghe (University of Twente) for the fruitful discussions that led to the quadratic evaluation method. Furthermore, discussions with Ute Claussen (Universitat Tubingen) made us more aware of the problems involved in angular interpolation.

References

[1] G. BISHOP AND D.M. WEIMER, "Fast Phong Shading," *ACM Computer Graphics (SIGGRAPH '86 Proceedings)*, vol. 20, no. 4, pp. 103-106, August 1986. Corrections appeared in Computer Graphics vol.21, no. 4, pp. 53. Also in the expression for T4 on pp. 105 -fjlqr should be replaced with +fjlqr and -2ffnr with -2fllnr.

[2] U. CLAUSSEN, "On Reducing the Phong Shading Method," in *EUROGRAPHICS '89*, Hamburg, Sept 1989.

[3] A. V. DAM, "PHIGS+ Functional Description, Revision 3.0," *ACM Computer Graphics*, vol. 22, no. 3, pp. 125-218, July 1988.

[4] M. DEERING, S. WINNER, B. SCHEDIWY, C. DUFFY, AND N. HUNT, "The Triangle Processor and Normal Vector Shader: A VLSI System for High Performance Graphics," *ACM Computer Graphics (SIGGRAPH '88 Proceedings)*, vol. 22, no. 4, pp. 21-30, 1988.

[5] J. FOLEY AND A. VAN DAM, *Fundamentals of Interactive Computer Graphics*, Addison-Wesley, Reading, Massachusetts, 1982.

[6] R. HALL, *Illumination and Color in Computer Generated Imagery*, Monographs in Visual Communication, Springer-Verlag, 1988.

[7] W.R. HAMILTON, *Elements of Quaternions*, Chelsea, New York, 1969.

[8] J.A.K.S. JAYASINGHE, A.A.M. KUIJK, AND L. SPAANENBURG, "A Display Controller for an Object-level Frame Store System," in *Advances in Graphics Hardware III*, ed. A.A.M. Kuijk, EurographicSeminars, pp. 141-170, Springer-Verlag, 1991.

[9] E. PERVIN AND J.A. WEBB, "Quaternions in Computer Vision and Robotics," CMU-CS-82-150, Carnegie Mellon University, 1982.

[10] B.T. PHONG, "Illumination for Computer Generated Images," *Communications of the ACM*, vol. 18, no. 6, pp. 311-317, 1975.

[11] M. SHANTZ AND SHEUE-LING LIEN, "Shading Bicubic Patches," *ACM Computer Graphics (SIGGRAPH '87 Proceedings)*, vol. 21, no. 4, pp. 189-196, 1987.

Divide and Conquer Radiosity

R. van Liere
Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

Abstract

This paper presents a coarse-grain parallel algorithm for solving the radiosity method. It describes a technique that partitions a large scene into a number of independent subscenes. The well known progressive refinement solution process will be applied to each subscene. After a fixed number of iterations each subscene will transfer accumulated energy to its neighbor.

Although we have limited ourselves to only diffuse scenes, the algorithm can easily be extended to specular scenes.

1 Introduction

During the past few years the field of image synthesis has been dominated by two trends : visual realism and interactivity. On one hand, considerable research is being done on producing increasingly realistic global illumination models. On the other hand, mainly due to the high expectations of future workstations, a trend towards interactive algorithms has been set. Within this context, novel techniques are being developed that will allow global illumination models to be solved at interactive speeds.

The radiosity method, [5], has recently become a very popular solution to a limited, but in many cases adequate, global illumination model. Originally, the radiosity method was restricted to environments consisting of only ideal diffuse reflectors and emitters. Later extensions, [6] and [8], have relaxed these restrictions somewhat so that specular and translucent reflectors and emitters can now also be taken into account.

Implementations of the initial radiosity method resulted in the so-called hemi-cube methods, [4], and its superior progressive reformulation, [3]. Subsequent extensions resulted in various two pass approaches which use hemi-cube method to determine the diffuse to diffuse surface reflections and ray-tracing techniques to take the specular surfaces of the environment into account, [9] and [8].

Recently, some researchers have also focussed their attention to parallel implementations of the radiosity method. Reker et. al., [7], use multiple workstations to solve a number of steps of the progressive refinement method in parallel. This method applies the server/client approach in implementing a coarse-grain parallel solution. It relies on future network technology to solve the high communication band-width between

clients and server. Baum et. al., [2], is an example of a finer grain parallel solution. This approach is unique in that it allows the end user to steer the radiosity calculations during the solution process. Finally, a completely alternative approach has been taken in a recent publication by Xu et. al., [10], in which a mathematical formulation is given of a method that subdivides a scene into multiple smaller scenes.

In this paper we present a parallel algorithm for solving the radiosity equations. We describe a technique to partition the complete set of radiosity equations into a number of independent smaller sets. Although we have limited ourselves to only diffuse environments, the algorithm can easily be extended to specular environments.

This paper is organized as follows: we first review the governing radiosity equations which are relevant for our environment. Next, we discuss the algorithm. Finally, we discuss convergence and error analysis of the method.

2 The Radiosity Equations

In [8], Rushmeier and Torrance extend the radiosity method to include specularly reflecting and translucent surfaces. Applying their results to surfaces that are *only* diffuse or ideal specular transmitting, we arrive at the following intensity equations:

- for ideal specular transmitting surfaces :

$$I_{o,n}(\theta_{o,n}, \phi_{o,n}) = I_{o,n(t)}(\theta_{o,n(t)}, \phi_{o,n(t)})$$

  Intuitively, this equation says that the outgoing intensity of surface n is equal to the outgoing intensity of the surface, denoted as $n(t)$, that intersects a ray specularly transmitted through surface n.

  Note that the intensities of ideal specular transmitting surfaces are direction dependent. This is indicated by the two angles, $\theta_{o,n}$ and $\phi_{o,n}$, denoting the polar and azimuthal angles.

  The equation is illustrated in figure 1, which also provides insight in the used notation.

- for diffuse surfaces :

$$I_{o,m} = I_{e,m} + \rho_m \sum_{n=1}^{N} \{I_{o,n} F_{mn} + \tau_n \sum_{q=1}^{N} I_{o,q} T_{f,nmq}\}$$

  Intuitively, this equation states that the outgoing intensity of diffuse surface m is equal to its emission plus a term due to other diffuse surfaces plus a term due to the other ideal specular transmitting surfaces.

  The term due to other diffuse surfaces is the same as in the original radiosity equations for diffuse surfaces, [3]. It includes the form-factor, denoted as F_{mn},

that specifies the fraction of the energy leaving one surface which lands on another.

The term due to the ideal specular transmitting surfaces is equal to the sum of the outgoing intensities times the so-called forward window form factor, denoted as $T_{f,nmq}$. Window form factors denote the fraction of energy leaving the transmitting surface, n, from its *back* side which impings on surface m. Window form factors are direction dependent.

The terms, ρ_n and τ_n, denotes the reflectance resp. transmittance function of surface n. Since we consider only diffuse surfaces this functions are independent of direction.

These equations are diagrammed as follows :

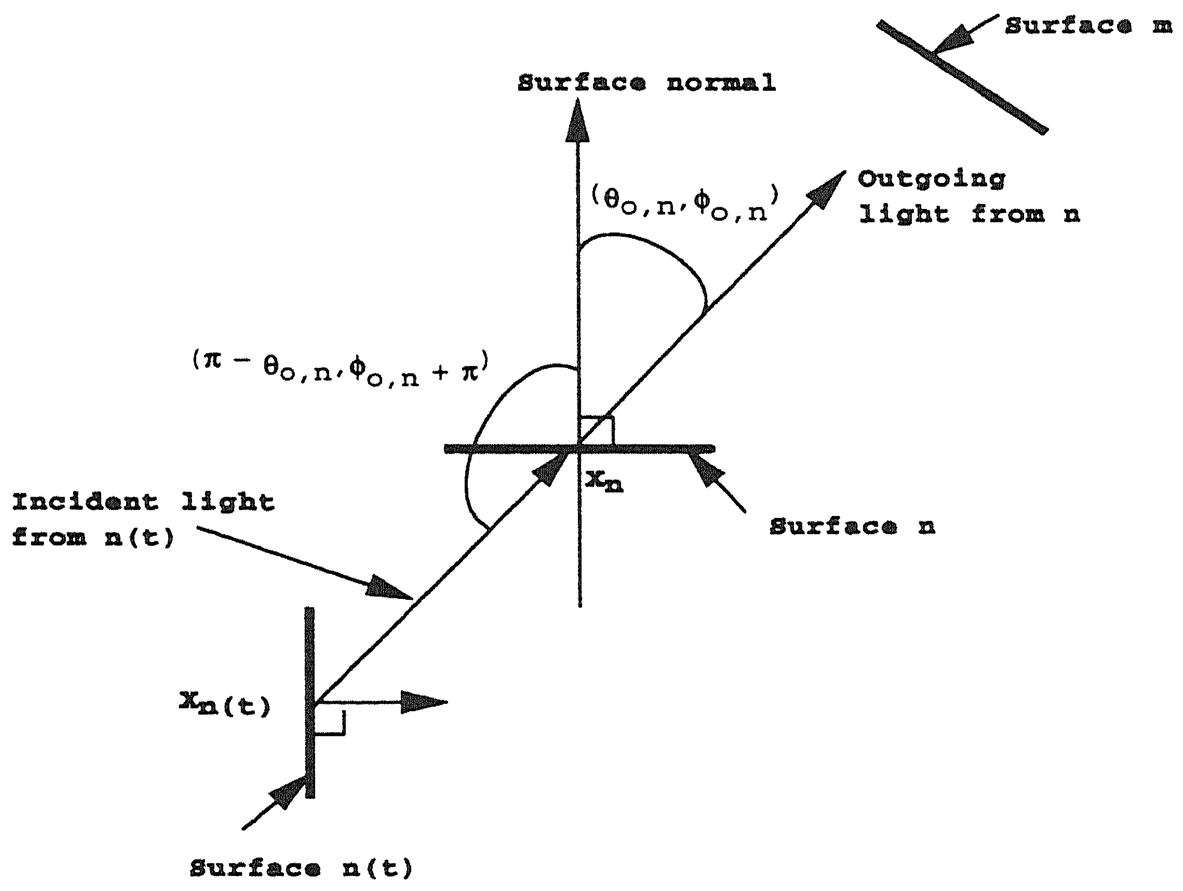


Figure 1. - Surface $n(t)$ from which light is specularly transmitted by surface n into the direction $\phi_{o,n}$ and impinging onto surface m. (Adapted from [8]).

3 The Algorithm

3.1 An intuitive view

Before going into the details, we first give a step-wise intuitive idea of how the algorithm works :

1. The algorithm starts by partitioning the scene to be rendered into a number of disjunct subscenes. Intuitively, this is done by placing a number of ideal translu-

cent surfaces into to the scene. These translucent surfaces act as boundaries for individual subscenes. Each boundary has two sides, the front and back sides.

2. Next, apply in each subscene a number of iterations of the progressive refinement solution process. All energy that impings a boundary surface is stored in the front side of the boundary surface and will not be shot back into this subscene. Both the impinging energy and impinging angle are stored in the accumulation buffer. Note that since the subscenes are independent of each other that this step can be done in parallel for each subscene.

3. Transfer the accumulated boundary energy from one subscene to the boundary in the adjacent subscene. This is done by transferring the impinging energy from one accumulation buffer (stored in the front side) to the corresponding angle in the adjacent buffer (which will be stored in the back side of the boundary). The energy stored in the back side of a boundary will subsequently be shot into the subscene.

4. Display all subscenes without displaying the boundary surfaces. Strictly speaking, this step can be done in parallel with the energy transferring step, or after each iteration of the progressive refinement solution process in each subscene. We choose to display the scenes sequentially simply for clarity reasons.

5. Go back to step two and repeat the process.

3.2 A closer look

The more detailed discussion of the algorithm can best be described through the following fragments of pseudo-code.

```
subDivideScene ()

while (notConverged)
{
     foreach subScene
        doIterations (n, subScene)

     foreach boundary
        transferEnergy (boundary)

     displayScene ();
}
```

Pseudo-code fragment 1. - The main loop.

This fragment of code contains the main loop which forks off a number of "workers" each executing the function *doIterations*. Note that during execution of *doIterations* there is no communication between workers. Once the workers have completed

their tasks, the energy that has accumulated on each boundary is passed on to the adjacent subscene, *transferEnergy*.

```
doIterations (n, subScene)
{
    while (n--)
    {
        shootPatch (selectPatch (subScene))
    }
}
```

Pseudo-code fragment 2. - The worker.

doIterations is a straightforward extension to the original sort and shooting algorithm. It selects a patch for shooting (the one with the greatest energy), and shoots its energy into the scene. The difference is that directions will have to be taken into account for boundary patches.

```
transferEnergy (boundary)
{
    foreach patch
    {
        transferEnergy (patch-> twin1, patch-> twin2)
        transferEnergy (patch-> twin2, patch-> twin1)
    }
}
```

Pseudo-code fragment 3. - The "synchronizer".

A boundary is divided in a number of patches. Each patch has two parts called *twins* . Twins are defined as : [1]

```
struct twin
{
    Radiosity incoming [SOLIDANGLES];
    Radiosity outgoing  [SOLIDANGLES];
}
```

in which *SOLIDANGLES* is a constant that determines the number of solid angles of the hemisphere. Note that the amount of memory used in a *twin* is proportional to the discretization accuracy of the hemisphere.

The following figure depicts a two dimensional view of the basic idea. The (progressively accumulated) incoming radiosity of *twin1* will be transferred to *twin2* which, in turn, will become a "shooting" candidate in its subscene.

[1] There is substantial room for memory optimization here.

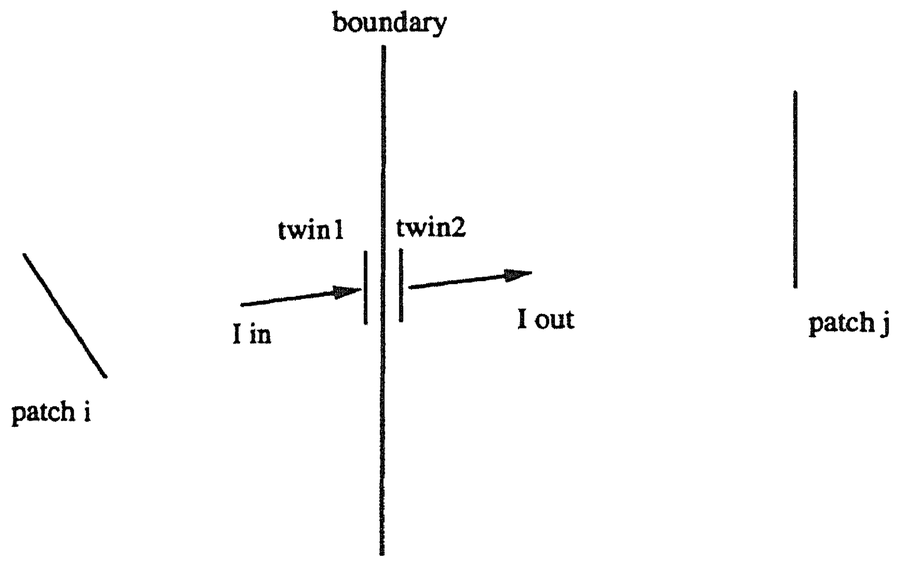


Figure 2. - Twins on a boundary.

Immel et. al., [6], have published an implementation technique that determines the radiosity in a non-diffuse environment. Aside from the convergence speed, Immel's method suffers from the vast amounts of storage needed to store directional form-factors and directional radiosity. Our method also requires a vast amount of memory. Two comments must, however, be made :

1. memory needed for the boundaries can be distributed over a number processors/workstations instead of only one central workstation.

2. additional memory is needed only for boundaries. This allows the additional memory needed to be a function of the available hardware resources and is independent of the complexity of the scene itself.

4 Convergence Analysis

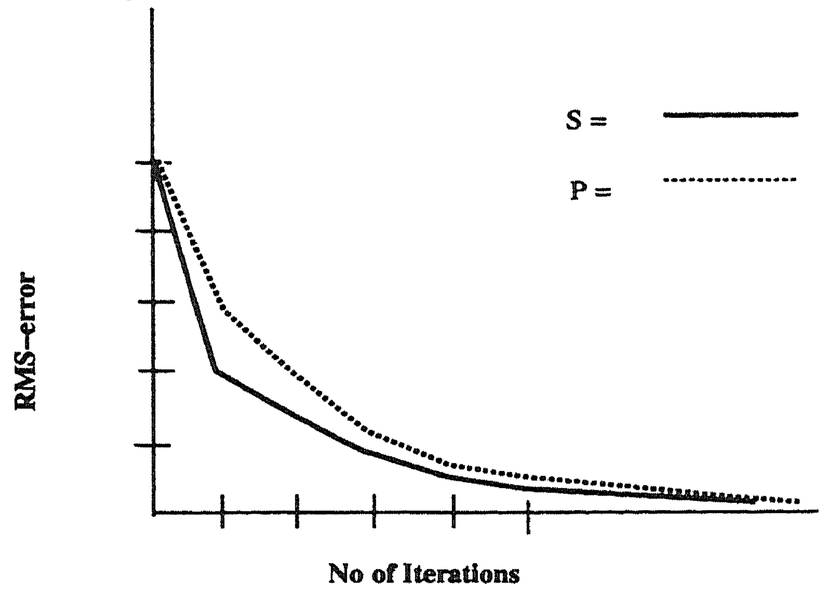
In [3] , Cohen et al. introduce a metric for comparing convergence rates of various radiosity solution methods. The given metric provides a quantitative measure of the overall radiosity inaccuracy after each iteration in the progressive refinement solution process. The square root of the area weighted mean of the square of the the individual errors (RMS error) was used as a indication of the error made at each iteration; i.e.

$$RMS\ error = \sqrt{\frac{\sum_{i \in Env} ((I_i^* - I_i)^2 A_i)}{\sum_{i \in Env} A_i}}$$

where I_i^* is the converged intensity and I_i is the intermediate intensity of a patch i in an environment Env.

The following graph shows a hypothetical graph of the normalized RMS error set out against the number of iterations during the solution process. Function S denotes the convergence curve of the progressive refinement solution process in one environment as a function of the number of iterations; i.e. the sequential case. Function P

nvergence curve of the progressive refinement solution process in mul-
›nments as a function of the number of iterations; i.e. the parallel case.
will converge to the same result.



Plot of normalized RMS error for the sequential (thick) and parallel (dotted) cases.

the following statements can be made about the behavior of S and P :

ll n : $S(n) < P(n)$; i.e. after a small number of iterations the mean **ade** by the single environment method is smaller than in the multiple **onment** case. This is due to a) extra boundary patches that are introduced **ibenvironment** method, and b) a partial ordering is used in the case of **subenvironments** to determine the shooting patch. Recall that patches **subenvironment** are sorted in increasing order. Partial orderings do not **:e** that a patch with the maximum accumulated energy is chosen as the **; patch.**

it n is the sum of the iterations in all subenvironments.

e n : $S(n) = P(n)$; i.e. both cases converge to the same mean error. **›ecause** both methods converge to the same result.

ll n and $k > 1$: $S(n) > P(kn)$ with k as the number subenvironments; **:r** n iterations in a single environment the mean error is greater than **terations** in each subenvironment in the multiple environment case. This **ise** k patches are shot simultaneously. Hence, k times the number of **is are** made.

Analysis

›n made by the divide and conquer method is that introducing additional **es** not influence the results of light energy calculations in any way. In

practice, however, additional errors are made during the form-factor calculations. The quantity of the errors depend in a complicated way on the scene being rendered. In practice, worst case errors are not very useful and "average case" errors are not well defined. We will, therefore, restrict ourselves to a qualitative analysis of errors that may occur.

Baum et al. [1] have shown that three types of errors can occur when the hemi-cube method is used to calculate form-factors :

1. errors due to the violation of the *proximity* assumption; i.e. the distance between patches i and j is small compared to the diameter of patch i. The proximity assumption is violated when, for example, i and j are adjacent patches that share a common edge. Since introduction of boundaries decreases the average distance between patches, the proximity assumption will be violated more often.

2. errors due to the violation of the *visibility* assumption; i.e. the visibility of a projected patch j does not alter across the shooting patch i. The visibility assumption is violated if another patch partially occludes the projected patch. Due to the additional boundaries, the visibility assumption will tend to be violated less frequently since there are a reduced number of patches per subenvironment.

3. errors due to the violation of the *aliasing* assumption; i.e. that patch j projects exactly onto whole pixels of patch i's hemi-cube. Introduction of boundary patches cause additional violations of the aliasing assumption. These errors can be minimized by, for example, using a larger hemi-cube.

Note that the divide and conquer algorithm does not mandate that form-factors be calculated with the hemi-cube method. Other form-factor calculation methods should be equally applicable.

6 Conclusion

We have presented a parallel algorithm for applying the progressive refinement method to the radiosity equations. The implementation of the algorithm is well suited for multiprocessor architectures because of the limited synchronization between processes.

We have also discussed convergence / error analysis of the algorithm. The convergence curve is far better than the sequential case. Additional errors are due to artifacts caused by the hemi-cube method for calculating form-factors.

The algorithm has been implemented in C++ on a network of Personal Iris workstations using SGI's graphics library for the low level rendering.

Future work will concentrate on : determining heuristics for optimal subscene division, dynamic subscene reconfiguration, minimizing memory consumption of boundaries.

Acknowledgements

Many thanks to Henk Schouten who contributed substantially to initial ideas and implementation issues. Thanks go also to Paul ten Hagen who provided me with the time and patience to work on these ideas.

References

[1] D.R. Baum, H.E. Rushmeier, and J.M. Winget. Improved radiosity solutions through the use of analytically determined form-factors. *Computer Graphics (SIGGRAPH '89 Proceedings)*, 23(3):325–334, 1989.

[2] D.R. Baum and J.M. Winget. Improving interaction with radiosity-based lighting simulation programs. *Computer Graphics (Interactive 3D Graphics)*, 24(4):51–57, 1990.

[3] M.F. Cohen, S.E. Chen, J.R. Wallace, and D.P. Greenberg. A progressive refinement approach to fast radiosity image generation. *Computer Graphics (SIGGRAPH '88 Proceedings)*, 22(4):75–84, 1988.

[4] M.F. Cohen and D.P. Greenberg. The hemi-cube : A radiosity solution for complex environments. *Computer Graphics (SIGGRAPH '85 Proceedings)*, 19(3):31–40, 1985.

[5] C.M. Goral, K.E. Torrance, D.P. Greenberg, and B. Bataille. Modelling the interaction of light between diffuse surfaces. *Computer Graphics (SIGGRAPH '84 Proceedings)*, 18(3):213–222, 1984.

[6] D.S. Immel, M.F. Cohen, and D.P. Greenberg. A radiosity method for non-diffuse environments. *Computer Graphics (SIGGRAPH '86 Proceedings)*, 20(4):133–142, 1986.

[7] R.J. Recker, D.W. George, and D.P. Greenberg. Acceleration techniques for progressive refinement radiosity. *Computer Graphics (Interactive 3D Graphics)*, 24(4):59–66, 1990.

[8] H.E. Rushmeier and K.E. Torrance. Extending the radiosity method to include specularly reflected and transluctent materials. *A.C.M. Transactions on Graphics*, 9(1):1–27, 1990.

[9] J.R. Wallace, M.F. Cohen, and D.P. Greenberg. A two-pass solution to the rendering equation: A synthesis of ray tracing and radiosity techniques. *Computer Graphics (SIGGRAPH '87 Proceedings)*, 21(4):311–328, 1987.

[10] H. Xu, Q-S. Peng, and Y-D. Liang. Accelerated radiosity method for complex environments. In W. Strasser W. Hansman, F.R.A. Hopgood, editor, *EUROGRAPHICS '89*, pages 51–61, Hamburg, 1989. North-Holland.

The FERSA Project for Lip-Sync Animation

P. Griffin

Email: patsy@cwi.nl
Department of Interactive systems,
CWI
Kruislaan 413, 1098 SJ Amsterdam, The Netherlands

H.Noot

Email: Han@cwi.nl

ABSTRACT
An approach to the development of automatic lip-synchronous animation for applications in a television post-production environment is discussed.
Key Concepts: computer vision, lip-reading, speech animation and lip-synchronization.

1. Introduction

Lip-synchronous facial animation, has had to employ time consuming, cost prohibitive, non automatic processes. The central goal of our work is to develop a means to fully automate the production of lip-synchronous animation for use in a television post-production environment.

To accomplish this we employ Computer Vision techniques to automatically process and analyze video images of a narrator and identify the necessary mouth positions needed to reconstruct an animation of the speaker. Since animations, 2-D, 3-D, claymation or film stills are more interesting if they display both lip-synchronization, as well as 'emotions', the FERSA [1)] Project proposes to, eventually, utilize not only the information from speech production but also other facial expression to drive an animation tool in real-time. However, here, we restrict ourselves to the recognition of mouth shapes during speech and limit the reconstruction to animations accessing images from a limited repertoire of stored stills. Thus, we drive an animation using mouth shape identifiers to access frames, creating an animation by means of a simple 'flip-book' method. The original audio track is simultaneously recombined with the animations.

To demonstrate the feasibility of our approach, we present for the user dependent case, a method of classifying visual speech elements from a measurement space of low dimensionality. To simplify our present task, we have temporarily lifted the 'real-time' requirement and recorded under controlled conditions.

[1)] Facial Expression Recognition as a driver for real-time lip-synchronous Speech Animation.

Simple animations have been produced using a pre-stored library of images. We have constructed the animations using 2-D graphics, 3-D computer generated still images and photographs of simple clay models.

Apart from the automation of lip-sync animation for television and CD-I, the FERSA system will contribute to the construction of talking heads for man-machine interfaces. Furthermore, the kind of data compression through facial expression recognition employed here, may also be used in videophone applications. In this paper we introduce the goals of the FERSA Project and present some initial results from a simplified test implementation.

2. Background

To place this project into a context let us examine approaches to the problem of speech animation:

1. Synthetic Speech To Image or Text-to-Visual Speech:

From a text script a phonetic score is generated and a mapping between phoneme and corresponding facial movement generates the animation [11, 25]. The problems associated with this approach include:

- The mapping used in the construction is language dependent.
- The set of synthetic sounds does not span that of human speech leading to a lack of 'naturalness'.
- Absence in rhythm and voice inflection.

2. Audio Speech Recognition to Image:

To remedy some of the problems associated with Synthetic Speech to Image, there are efforts to perform speech recognition through frequency spectrum analysis [24, 27, 33]. The problems that are encountered using this approach to construct a visual animation include:

- Given the complexity of speech recognition, it is difficult to achieve this in real time for a large range of speakers.
- One can not detect non audible mouth movement or gestures.

3. Visual Speech Recognition to Image:

An alternative approach uses Computer Vision/Image Processing techniques and anatomical analysis for purposes of full speech recognition [4, 10, 29, 36]. Problems associated with this approach include:

- User independence is difficult for a large vocabulary.
- The processing requirements here hinder real-time implementation.

Our approach hopes to remedy the above problems, however we draw from this work along with fundamental work in speech analysis, which provides us with knowledge about mouth shape/movement during speech production [1, 17, 30]. We have relied on this body of research to guide us in choosing a relevant feature space, training set and in determining the set of visual speech elements.

3. A Sketch of the Overall FERSA System

Since our goal is to drive a lip-synchronous facial speech animation within the context of a (multi-media) television post-production environment, we do not intend to recognize speech but rather facial expressions and mouth shapes of the speaker. Thus, our 'limited' recognition task is simpler than that of Visual Speech Recognition. In addition however, we will extend our recognition to gestures expressing simple non audible expressions such as a smile, smirk or grimace.

We propose to monitor the facial expressions of a narrator at the PAL rate of 25 frames/sec. Our recognition module will simultaneously analyse the frames and classify the expressions contained in them (see fig. 1). As a result, it will produce a string of identifiers [2)] that serves as input to the reconstruction module.

These identifiers are used by the reconstruction module to access the corresponding animation frames from disk or image storage unit. Thus the proper sequence of animation frames is determined and can be recombined with the audio track to form an animation. This string of identifiers is analogous to a 'musical score'. Here however, the score consists of 'visual speech' references. Possible means of image storage and real-time access are CD-ROM, DDR, DV-I [20, 25].

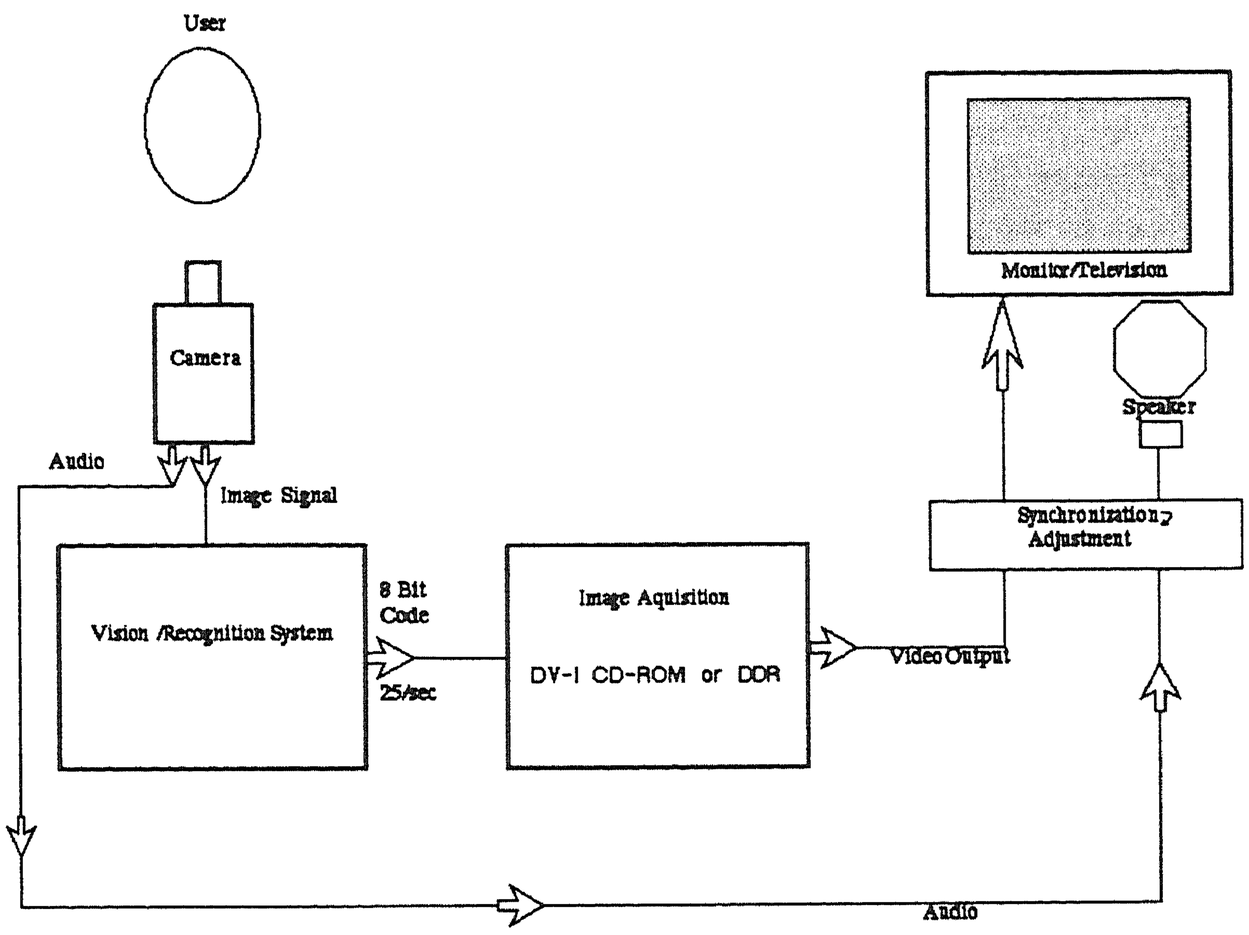


Figure 1: Sketch of the FERSA system

Previous research [3, 5], suggests that a small set of mouth shapes, approximately 20, is sufficient to represent the basic set of mouth shapes [3)]

[2)] They could also be used for off line methods of animation, such as supplying keyframe information for 3-D rendering.

necessary to reconstruct a 'convincing' speech animation (French and English). This, however, holds for reconstruction at unequal intervals. We will be sampling at an equal interval rate of 25/sec, thus, we may need additional mouth shapes in the reconstruction to accommodate for 'in-between' mouth positions.

Thus, we are faced with problems of:

- Defining a concise training set.
- Defining the reconstruction set of fundamental mouth shape classes.
- Recognizing and identifying the mouth shape classes during speech.
- Mapping an identified mouth class onto a corresponding animation frame.
- Rendering the animation.

4. Relevant Literature

Since the FERSA project is an attempt to combine various software and hardware developments into a tool which supports the creation of lip-sync animations, we discuss some relevant work.

4.1. Lip Reading

In the last 15 years, a lot of research has been published on automated lip-reading which provides us with valuable information on mouth shapes, movements and their classification during speech production.

An early paper (1983) in this field is the one by Brooke and Summerfield [6]. They discuss a procedure where a speaker's face is marked with small reflective dots and a video recording is made of the speaker pronouncing test material. Next, the recorded 'articulatory trajectories' are used to drive an elementary 2D facial model. In a 1986 paper by Montgomery and Finn [16] the variations of distances between reflective dots is used to recognize visemes [4] in VCV (Vowel-Consonant-Vowel) utterances.

In a series of interrelated papers by Brooke [7], Brooke and Petajan [8] and also Petajan, Bischoff, Bodoff and Brooke [35, 36] research is discussed in which grayscale video images are directly analysed. Various measures for describing lip shapes are discussed, among them distances, areas and direction dependent radii of the oral cavity. Features extracted from training material are used to form clusters, whose centers can then be used to classify mouth shapes occurring in test sentences.

Finally, there is recent research focusing not on mouth shape but on facial motion. This work by Pentland and Mase [29, 34] applies optical flow analysis to find classifiers.

---

[3] We will refer to this set of reconstruction mouth shapes as the set of mouth shape classes or simply, the reconstruction set.

[4] Viseme: visual speech element, analogous to the accoustic concept phoneme.

4.2. Facial Models and Synthetic Speech

Work being done in the field of facial modeling not only gives us clues and aids us in our recognition problem, but presents us with one of the means of realization. Maybe the best known facial model is the one by Parke (1982) [32]. In his paper a parametric facial wire model is described that can be covered with tissue and skin. We use this model (available in the public domain) in one of our tests.

The FACS (Facial Action Coding System) by Ekman and Friesen [14] has contributed to this area in a fundamental way. In this work, facial expressions are broken up into standardized components in such a way, that they can be described in coded form by designating their components.

Inspired by the FACS system, 3-D head-models have been constructed that can be driven by 'Action Units'. Waters [42] and Waters and Terzopoulos [41] discuss facial models based on the anatomical (muscle- and skull) structure of humans and so do Platt and Badler [37].

Other literature discusses systems that can produce Lip-Sync synthetic speech. In these systems speech is described by code for some speech synthesiser and accompanying code to drive the facial model. Examples of such systems are those of Hill, Pearce and Wyvill [24, 33], the Japanese Tron Project [21] and a system by Storey and Roberts [40]. In this last system, there is no 3-D model but a set of specially produced 2-D images corresponding to typical articulatory postures.

4.3. Acoustics Signals Driving the Animation

Another direction is taken by systems which attempt to extract the phonemes from acoustic speech signals, map them to visemes (which can be a many to one mapping). The visemes are used to create the animation.

An interesting precursor of this work is reported by Hideo Kawai and Shinichi Tamura [26]. They describe a system for generating deaf-and-mute sign language through the analysis of sound. In [28], Lewis and Parke describe a method for analyzing digitized speech based on a linear prediction model. The result of this analysis is used to drive Parke's facial model [32] lip-sync with re-synthesized speech.

Possible videophone applications have spurred interest in the kind of image compression that can potentially be achieved by classifying facial expressions in a small number of sets. Thus compact shape code can be transmitted so that an image can be synthesized at the receiving side instead of transmitting a full image. Two systems that use the acoustic signal to arrive at mouth shape codes are described in a paper by Morishima , Aizawa and Harashima [31] and in another one by Welsh, Simons, Hutchinson and Searby [43]

4.4. Performer-driven Animation

Another approach to generate the information needed to drive facial models, which is also the one we are taken, can be called 'performer-driven'. In a 1990 paper by Williams [44] where this term is used, a system is described where the face of a performer is marked with reflective dots. Movements of the performer's face are video recorded, analysed and classified. This analysis leads to code which can drive a facial model. In William's system this model is derived from a scan of a plaster casting of the same performer. An early (1981) description of this strategy can be found in a paper by Platt and Badler [37].

4.5. Image Analysis

We use standard approaches from the field of Mathematical Morphology, as pioneered by Serra [39]. Introductions to this subject can be found in a paper by Haralick, Sternberg and Zhuang [22] and in chapter 15 of a book by Pratt [38].

Finally we wish to mention the literature on image classification and clustering. A very readable introduction to clustering can be found in Everitt [15], an in depth treatment in Fukanaga [18]. Compact discussions can be found in Coleman and Andrews [12] and in Haralick and Kelly [23]. Last but not least a wealth of material is contained in Duda and Hart [13]

5. Aspects of the FERSA System

5.1. Aspects of the Initial Experimental System.

The test implementation involved the reconstruction of simple animations using experimental software, and partly ad hoc solutions.

In order to define the set of fundamental mouth shape classes needed for a successful reconstruction of the input speech, a training set was needed. Our training set consisted of simple Consonant-Vowel-Consonant (CVC) and Vowel-Consonant-Vowel (VCV) [5)] words utterances as input to the recognition module. We choose 4 consonant groups and 6 vowel groups, so we only used a subset of possible speech. From each video image of this training set, the mouth region was isolated and aspects of mouth shape or movement were measured (see fig. 2).
Analysis of the resulting data through clustering techniques lead to the selection of the mouth shape classes or the reconstruction set. This set was expected to span a sufficient subspace of visual speech.

5) CVC: aba,obob,ibi, ebe and VCV: bab,bob,bib,beb.

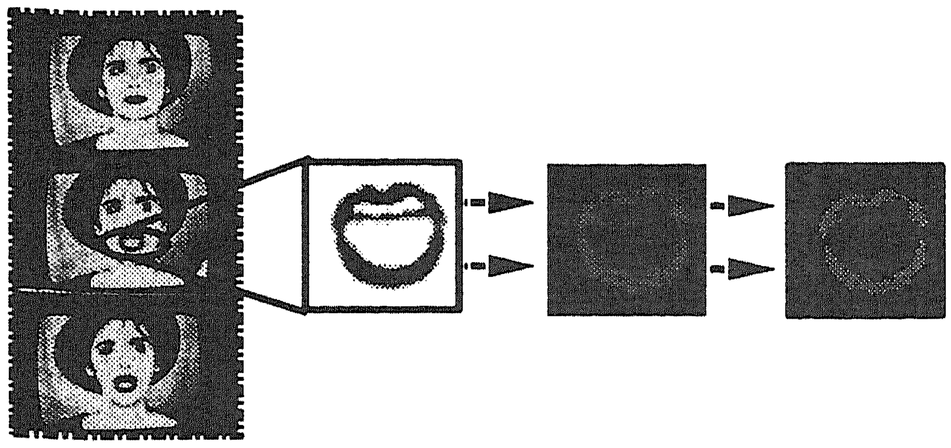


Figure 2: Analysing mouth forms.

The speaker spoke two polysyllabic words, Dutch and English, and one simple sentence. Each frame of the input video was analyzed automatically and identified as belonging to one of the mouth classes in the reconstruction set. The identified class was then mapped [6] to a corresponding animation frame and recombined with the original sound track to create the animations.

5.1.1. Image Acquisition

Our first investigation was to compare two very different approaches to the acquisition of the narrator's facial features. In the first approach, the narrator is marked up with reflective dots, in the second approach a B/W video recording is used. To explore the first approach, reflective dots, we were allowed to use the PRIMAS motion analysis system developed at Delft Technical University [19]. For the exploration of the second approach, we made recordings in our own office space, where one of us assumed the role of narrator. We also enhanced the facial features of the narrator through the use of make up.

The PRIMAS system allowed us to measure the position of 23 reflective dots surrounding the mouth area at a rate of 100 images/sec. This method produced a lot of highly accurate material which we used for initial analysis of mouth shapes but the recording seemed rather cumbersome for its intended use in a studio environment (see fig. 3).
The recorded video approach turned out to be well suited to image analysis techniques and was relatively easy to apply as far as recording is concerned. We chose for the latter approach.

Experiments performed by Brooke [9] indicate a lower limit of image detail necessary to carry out useful visual synthesis and reconstruction. Thus, we have chosen to work with binary images. To further cut down on the pre-processing time for the test system, we have used a low resolution, high contrast image where the narrator also has whitened skin and teeth and blackened lips (see fig. 4 left). The cameras aperture was also opened by one f-stop for over exposure.

For the initial experiments on training and recognition, the full frontal view of the speaker was recorded. We had 20 minutes of video which we edited down to 3.5

---

[6] This is not necessarily a one to one mapping, but depends on the type of animation.

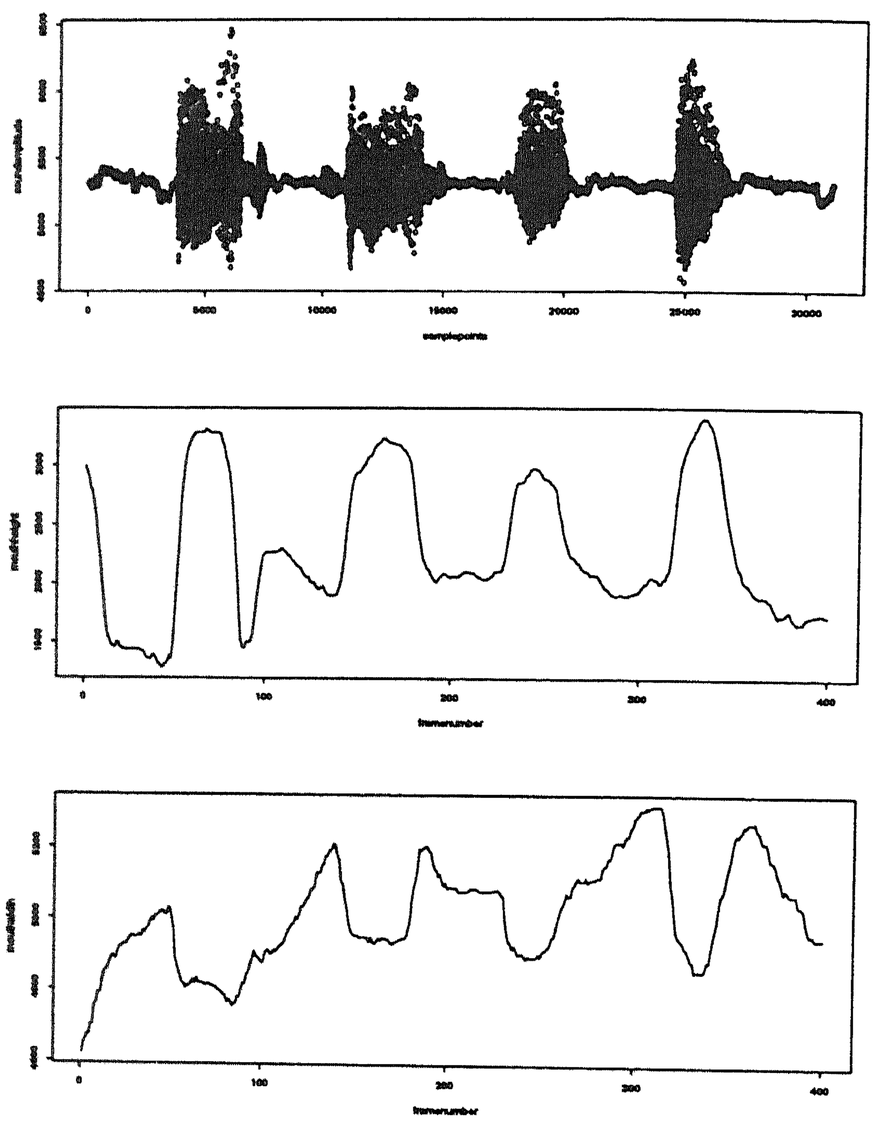


Figure 3: Measurements with the PRIMAS system for the utterance bab dad kak sas. Shown are sound amplitude versus sample points (8000/sec), mouth height versus frame number (100/sec) and mouth width versus frame number.

Figure 4: Left: Whitened face and blackened lips for recording. Right: Line drawing as a keyframe for reconstruction.

minutes. This material was written to a Sony laser disc for digitizing and enabling access of relevant images for a Silicon Graphics work stations as needed.

5.1.2. Analysis

In order to facilitate our analysis of training- and test-data and to prototype our own software, we used a powerful image processing package, Scilimage, developed by the University of Amsterdam.

During the analysis phase, our first test system performed four simple measurements on each mouth image. After normalization and scaling, the ratios of these measurements gave us points in a 2-D feature space. Scatterplots convinced us that these two numbers were fairly independent.

In this space, we performed a straightforward cluster analysis, using the hierarchical clustering procedures included in the S system for data analysis [2]. As distance measure, we employed simple euclidian distance between feature space points. After some experimentation and visual inspection of the feature space, we set the number of clusters to 30. This was in agreement with the number of mouth shapes expected from the literature. However, we take into account inbetween mouth shapes due to our regular sampling intervals..

As a final step, in each of these clusters a central point was chosen (a cluster centre). Thus forming the set of mouth shape classes.

5.1.3. Recognition

Since here we are only dealing with a subset of speech, we have chosen a small test set. The test set consists of the (video) recorded words: 'Casablanca', 'Smakelijk eten' and 'My name is Pat'. As stated, from the previous cluster analysis of the training set, we could determine a subset of images with representative mouth shapes. Using our test set as input into a simple recognition algorithm we have successfully reconstructed three animations. At this point we have performed the animation using less than 30 mouth shapes for simple drawings (see fig 4. right) or claymation stills and only eight mouth shapes for 3-D generated wireframe animation. We used eight shapes similar to the eight presented in a paper by Welsh, Simons and Hutchinson [43].

For the test animations, recognition proceeded as follows: On every image the same four measurements were made that were made during analysis of the training set. These measurements defined a point in feature space. Next the cluster centre nearest to this point (using euclidian distance) was selected and so the representational mouth shape is determined.

In the reconstruction experiments the chosen clusters proved adequate, but one unexpected problem turned up. Initially we used too many clusters, so cluster centres were lying near to each other. Because of that the recognition sometimes tended to flip between them in a a series of rather similar input images. This resulted in a somewhat jittery animation. Hence we had to conclude that there is a danger not only in having too few but also in having too many mouth shapes.

5.1.4. Reconstruction

Reconstruction can be tackled in basically two distinct ways: first there is the 'flip-book' approach and secondly we can manipulate the parameters of a facial model.

In the flip-book approach, images are flipped to screen to form an animation. This sequence of images is determined by a mapping from one of the recognized mouth classs to an animation mouth shape which corresponds to an animation frame. Depending on the type of realization, 3-D wire frame, 2-D claymation or line drawing (see fig. 4 right), the mapping of mouth class to animation frame is not necessarily one to one (see fig. 5). The number of mouth classes is maximized for purposes of full 'realism' in speech animation although in a simple line drawing or a 3-D implementation using an 'off line' keyframe method, it may only be necessary to animate with a small number of shapes. High quality animations working in this way can also be constructed off-line and set up frame by frame in a professional studio.

For simpler applications and for testing and editing purposes we wrote a program running on Silicon Graphics workstations using SGI's graphics library and audio tools. This program can:

- Display keyframes at a rate of 25 images/second. To meet real-time requirements, the keyframes needed are kept in core at all times. So it depends on the amount of memory available on the workstation and on its further workload how big the keyframes can be and how many different ones can be used in the animation. In our situation we can have animations of arbitrary length using some 30 keyframes with 550K pixel data each.
- Play the audio together with the animation at the quality provided by SGI's standard tools.
- Interactively align the beginning of the (separately recorded) soundtrack with the first image of the animation.

In the facial model approach, we also aim at displaying 25 images a second. Once again, mouth shape classes correspond to parameter settings for the facial model, but in addition extra time varying parameters can be used to manipulate other features of the model. The facial model easily adapts itself to synthetic 'talking heads' and can be nicely used to experiment with 3-D rendering of moving objects.

5.2. Some Aspects of the Final System

In this section we discuss some implementation aspects of the final FERSA system which reflect design choices that have been made on the basis of our experiments.

5.2.1. Recording Procedures and Problems

During video recording of training set, test set and later production runs a number of technical problems have to be taken into account. In order to extract meaningful mouth shape features from recorded images and to do so at sufficient speed, it is necessary that:

- The mouth region can be easily located in the image.
- The distance and orientation of the mouth with respect to the camera remains constant. (Because of the mobility of the face, we should be more precise and say: the distance and orientation of the skull must be constant.)

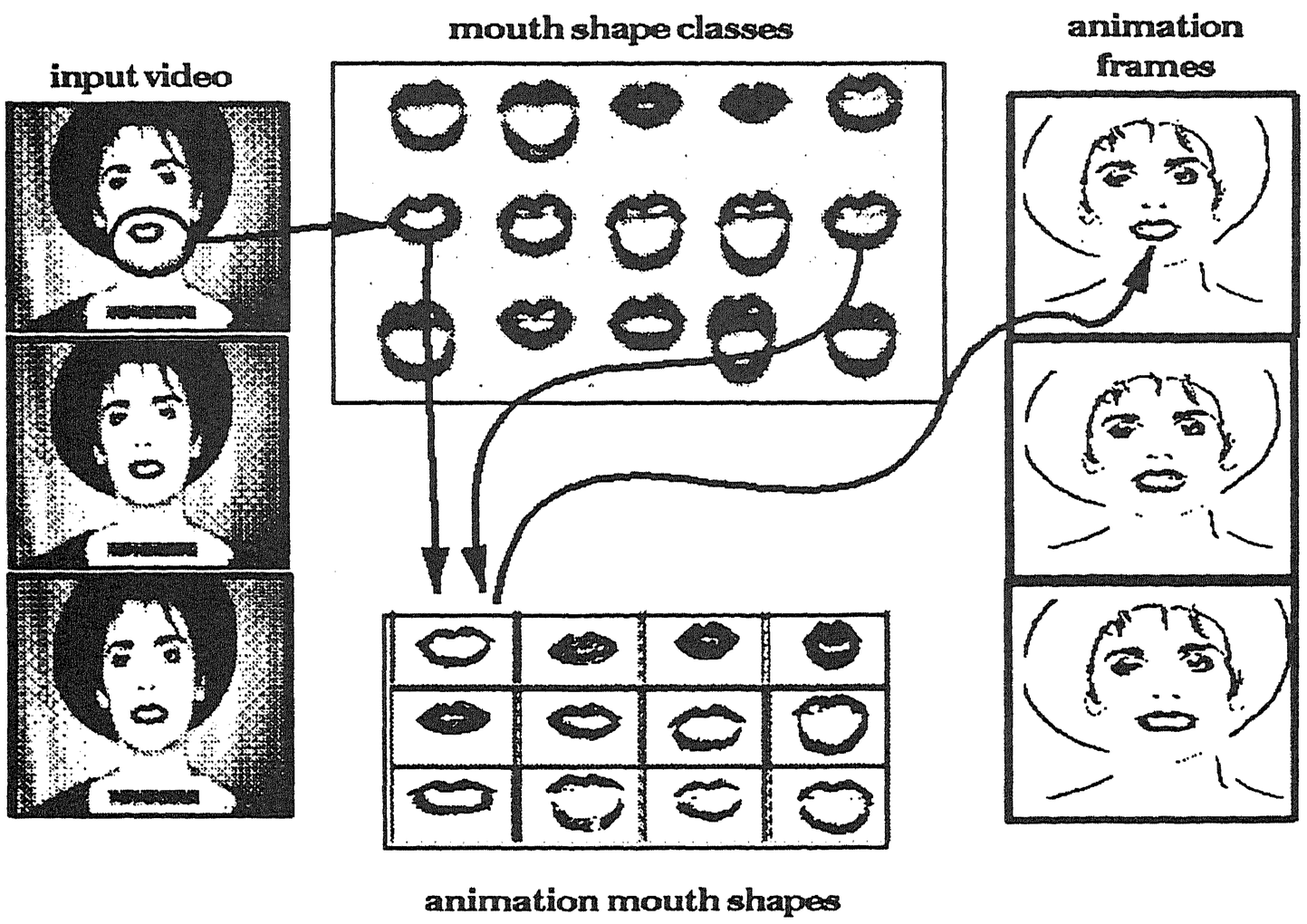


Figure 5: Figure 5:

This distance constancy is not only needed during one single recording, but the training set and later recordings have to be done at one and the same distance and orientation too. If this condition is violated mouth shape features extracted during training cannot be compared to features extracted during production runs and hence no classification is possible.

Both problems can be solved, using the idea of nostril tracking as reported by Brooke and Petajan [8]. In their paper it is noted that the nostrils, which show up in a facial image as two dark spots, have fairly constant position with respect to the facial skeleton. Furthermore, they do not move very much between successive video frames. If their position in the first frame to be analysed is somehow given, their later positions can be tracked. In every new frame we can look for them near their position in the previous frame. Finally, having a means for finding nostril positions, the mouth can be easily located by searching for it in a region at a fixed (speaker dependent) offset from those nostrils. In Petajan's system nostril tracking was done by a special purpose piece of hardware, while we intend to perform that task by software. At the time of this writing, this software is not yet complete, but experiments indicate that it can indeed do its job in real-time.

In some experimental systems the position of the speaker's head with respect to the camera is fixed by mechanical means (for instance by using a head mounted camera). This solution does not work for us; a (professional) performer simply gets to much distracted by equipment clamped to his/here head. We intend to explore another strategy (as already used by Brooke and Summerfield, see [6]). Suppose the actor is temporarily provided with "extra nostrils" for instance by painting dark

spots on more or less immobile places on the face like for instance the bridge of the nose. Nostril tracking can then be done on these objects. This gives enough information to determine the linear transformation that connects the actual head position with a standard one. This approach has a consequence though; for efficiency reasons, we have to look for classification features which can be transformed back by this linear transformation after being extracted from the untransformed image. Otherwise the whole image has to be transformed before feature extraction which is costly in computing time.

5.2.2. Special Purpose Image Analysis

Software must be developed to meet real-time requirements for feature extraction and tracking. The relevant image processing libraries from Scilimage, which we use for prototyping purposes is very fast but even faster performance can be obtained from special purpose software with build in knowledge of the type of features and images that will be encountered. Our initial work in software development for tracking and feature extraction proves promising.

6. Conclusions

The results obtained so far with the initial experimental system seem to warrant the conclusion, that the approach taken in the FERSA project will lead to a tool that can be used in the production of lip-sync animations in a production environment.

References

[1] C. ABRY AND L. J. BOE, "Laws for Lips," *Speech Communication*, vol. 5, pp. 97-104, 1986.

[2] R. A. BECKER, J. M. CHAMBERS, AND A. R. WILKS, *The S Language*, Wadsworth & Brooks/Cole, Pacific Grove, California, 1988.

[3] C. BENOIT AND T. LALLOUCHE, "Nineteen (+-19) French Visemes for visual speech synthesis," in *Proceedings of the ESCA workshop on Speech Synthesis*, september 1990.

[4] C. BENOIT, "Why Synthesize Talking Faces?," in *Proceedings of the ESCA workshop on Speech Synthesis*, pp. 55-71, september 1990.

[5] S. BRENNAN, Master's Thesis, School of Architecture and Planning., Architecture Machine Group, MIT, Cambridged, M.A., 1982.

[6] N. M. BROOKE AND Q. SUMMERFIELD, "Analysis, synthesis, and perception of visible articulatory movements," *Journal of Phonetics*, vol. 11, no. 63-76, pp. 63-76, 1983.

[7] N. M. BROOKE, "Visual Speech signals: Investigating Their Analysis, Synthesis and Perception," in *The Structure of Multimodal Dialogue*, ed. D. Bouwhuis, pp. 249-258, Elsevier, 1990.

[8] N. M. BROOKE AND E. PETAJAN, "Seeing speech: Investigations into the Synthesis and Recognition of Visible Speech Movements using Automatic Image Processing and Computer Graphics," in

Proceedings of the IEEE conference on Speech Input/Output; Techniques and Applications. Conference Publication No. 258

[9] N. M. BROOKE AND P. TEMPLETON, "Visual Speech Intelligibility of Digitally Processed Facial Images," in *Proceedings of the Institute of Acoustics*, pp. 483-490, 1990.

[10] N. M. BROOKE AND P. TEMPLETON, "Classification of Lip-Shapes and their association with acoustic speech events.," in *Proceedings of the ESCA workshop on Speech Synthesis*, pp. 245-252, september 1990.

[11] M. M. COHEN AND D. W. MASSARO, "Synthesis of Visible Speech," in *Behaviour Research Methods, Instruments and Computers*, 1990.

[12] G. B. COLEMAN AND H. C. ANDREWS, "Image Segmentation by Clustering," *Proceedings of the IEEE*, vol. 67, no. 5, pp. 773-785, May 1979.

[13] R. O. DUDA AND P. E. HART, *Pattern Classification and Scene Analysis*, John Wiley & Sons, New York, 1973.

[14] P. EKMAN AND W. FRIESEN, *Facial Action Coding System*, Consulting Psychologists Press,, Palo Alto, CA., 1987.

[15] B. EVERITT, *Cluster Analyses*, Heineman Educational Books, London, 1974.

[16] K. FINN AND A. MONTGOMERY, "The use of Visible Lip Information in Automatic Speech Recognition," in *Signal Processing III: Theories and Applications*, pp. 571-580, 1986.

[17] V. A. FROMKIN, "Lips positions in American English vowels," *Language and Speech*, vol. 7, pp. 215-225, 1964.

[18] K. FUKUNAGA, *Introduction to Statistical Pattern Recognition*, Acadamic Press, New York and London, 1972.

[19] E. FURNEE, *TV/Computer motion analysis systems*, Delft University of Technology, 1989. PhD Thesis

[20] J. GECSEI AND P. GIRARD, "Improving Computer Interfaces by Animation from Videodisc," *Visual Computer*, vol. 6, 1990.

[21] D. GROSS, "Merging Man and Machine," *Computer Graphics World*, pp. 47-50, May 1991.

[22] R. M. HARALICK, S. R. STERNBERG, AND X. ZHUANG, "Image Analysis Using mathematical Morphology," *IEEE ransactions on Pattern Analysis and Machine Intelligence*, vol. 9, no. 4, pp. 532-550, July, 1987.

[23] R. M. HARALICK AND G. L. KELLY, "Pattern Recognition with Measurement Space and Spatial Clustering for Multiple Images," *Proceedings of the IEEE*, vol. 57, pp. 654-665, April 1969.

[24] D. R. HILL, A. PEARCE, AND B. WYVILL, "Animating Speech: an Automated Approach using Speech Synthesized by Rules," *Visual Computer*, vol. 3, pp. 227-287, 1988.

[25] H. KAWAI, S. TAMURA, AND K. OKASAKI, "Sign Language Generation System Using Optical Disk Unit," *J. Inar. Telev. Eng.*, vol. 44, no. 3, pp. 303-311, Japan, 1990.

[26] H. KAWAI AND S. TAMURA, "Deaf-and-Mute Sign Language Generation System," *Pattern Recognition*, vol. 18, no. 3/4, pp. 199-205, 1985.

[27] J. LEWIS, "Automated Lip-sync: Background and Techniques," *The Journal of Visualization and Computer Animation*, vol. 2, pp. 118-122, 1991.

[28] J. LEWIS AND F. PARKE, "Automated Lip-Synch and Speech Synthesis for Character Animation," in *Proc. CHI & GI, '87 Human factors in computing systems and graphics interface*, pp. 143-147, Toronto, 1987.

[29] K. MASE AND A. PENTLAND, "Automatic Lipreading by Optical-Flow Analysis," *Systems and Computers in Japan*, vol. 22, no. 6, pp. 67-75, 1991.

[30] A. MONTGOMERY, B. K. WALDEN, AND R. A. PROSEK, "Effects of Consonantal Context on Vowel Lipreading," *Journal of Speech and Hearing Research*, vol. 30, pp. 50-59, March 1987.

[31] S. MORISHIMA, K. AIZAWA, AND H. HARASHIMA, *An Intelligent Facial Image Coding Driven by Speech and Phoneme*, pp. 1795-1798, IEEE, 1989.

[32] F. I. PARKE, "Parametrized Models for Facial Animation," *CG&A*, vol. 9, no. 2, pp. 61-68, November 1982.

[33] A. PEARCE, B. WYVILL, G. WYVILL, AND D. HILL, "Speech and Expression: A Computer Solution to Face Animation," in *Graphics Interface '86*, 1986.

[34] A. PENTLAND AND K. MASE, "Lip Reading: Automatic Visual Recognition of Spoken Words," M.I.T. Media Lab Vision Science Technical Report 117, January 15, 1989.

[35] E. PETAJAN, N. BROOKE, B. BISCHOFF, AND D. BODOFF, "Experiments in Automatic Visual Speech Recognition," in *Proceedings SPEECH '88*, pp. 1163-1170, Edinburgh, 22 - 26 August 1988.

[36] E. PETAJAN, B. BISCHOFF, AND D. BODOFF, "An Improved Automatic Lip-reading System to Enhance Speech Recognition," in *Proc. CHI & GI, 88 Human factors in computing systems and graphics interface*, pp. 19-25, Toronto, 1988.

[37] S. M. PLATT AND N. I. BADLER, "Animating Facial Expressions," *Computer Graphics*, vol. 15, no. 3, pp. 245-252, August 1981.

[38] W. K. PRATT, *Digital Image Processing*, John Wiley & Sons, New York, 1991.

[39] J. SERRA, *Image analysis and Mathematical Morphology*, Academic Press, London, 1982.

[40] D. STOREY AND M. ROBERTS, "Reading the Speech of Digital Lips: Motives and Methods for Audio-Visual Speech Synthesis," *Visible Language*, vol. XXII, no. 1, pp. 112-127.

[41] D. TERZOPOULOS AND K. WATERS, "Analysis of facial images using physical and anatomical models.," *IEEE Proceedings, ICCV conference*, pp. 727-732, Osaka, Japan.

[42] K. WATERS, "A muscle model for animation three-dimensional facial expressions.," *Proceedings of SIGGRAPH*, pp. 17-24, 1987.

[43] W. J. WELSH, A. D. SIMONS, R. A. HUTCHINSON, AND S. SEARBY, "Synthetic Face Generation for Enhancing a User Interface," in *Proceedings of the 1th Image'Com conference*, pp. 177-182, Bordeaux, November 1990.

[44] L. WILLIAMS, "Performance-Driven Facial Animation," *Computer Graphics*, vol. 24, no. 4, pp. 235-242, ACM, August 1990.

Components, Frameworks and GKS Input

D. A. Duce†, R. van Liere‡, P.J.W. ten Hagen‡

† *Rutherford Appleton Laboratory, Chilton, Didcot, OXON, U.K.*
‡ *CWI, Amsterdam, The Netherlands*

This paper was inspired by the Components/ Frameworks approach to a Reference Model for computer graphics, currently under discussion in the ISO computer graphics subject committee. The paper shows how a formal description of the GKS input model may be given in Hoare's CSP notation and explores some extensions in which some of the components in the GKS model are replaced by more interesting ones. The paper thus demonstrates some of the power and flexibility inherent in the Component/ Frameworks idea. The use of a formal notation led to a deepening of the authors' understanding of the input model and suggested some different ways of looking at the input model.

1980 Mathematics Subject Clasification : 69K32
1983 CR Categories : I.3.2
Key Words & Phrases : Graphics systems, formal descriptions, CSP, Component/Frameworks.

1. Introduction

This paper explores the application of a particular formal description technique, Hoare's Communicating Sequential Processes (CSP)[1] to the description of graphical input in the current generation of graphics standards, in the context of the Components and Frameworks [2],[3] approach to reference models for computer graphics.

This paper starts with an overview of the Components/ Frameworks idea followed by an overview of CSP. The next section describes the GKS input model in CSP and the following section gives some examples of how the model can be generalized. Although some of these ideas have been presented before in ISO working documents, the formulation given here is more general and more elegant, as a result of the structure of the formal description. The use of a formal description technique here has suggested new ways of presenting the GKS input model and has also suggested equivalences between the operating modes in GKS which were not previously apparent.

2. Overview of Components/ Frameworks

At the first plenary meeting of the new ISO/IEC subcommittee responsible for computer graphics, ISO/IEC JTC1/SC24, the need to review the work of its predecessor committee and to plan for the future development of graphics standards was recognized. SC24 authorised the formation of a Special Working Group to recommend a five-year strategic plan for organizing the work of SC24.[4] The Special Working Group met at Blakeney in the U.K. in April 1988. The major recommendation was that the next generation of computer graphics standards should be based firmly on a Reference Model which could identify demarcations and resolve disputes between standards. The Special Working Group also recommended a new approach to the development of standards, called the ''Component/ Framework Process''. Inherent in this process was a model of graphics standards which sees a graphics standard as constructed from a collection of components set in a framework. Components would include datatypes and operations. A framework is the ''glue'' which joins components together to form a system and performs management concerned with display and control. This model was seen as promoting harmonization between standards through the use of common components and frameworks. This idea is illustrated in Figure 1 below. Standards A, B and C each have their own frameworks. Some components are used by more than one standard, others by only one standard.

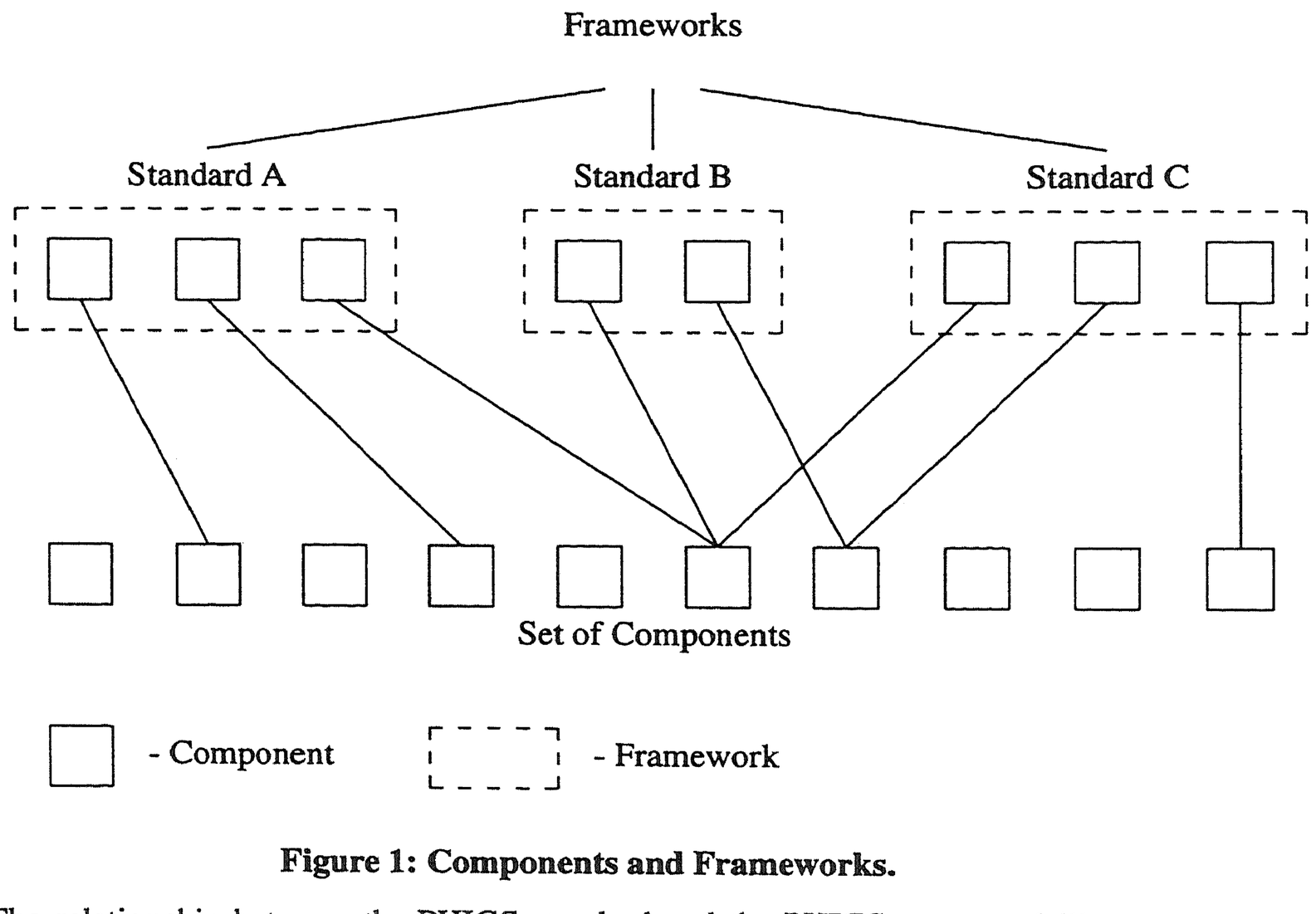


Figure 1: Components and Frameworks.

The relationship between the PHIGS standard and the PHIGS+ proposal illustrates this idea in that PHIGS and PHIGS+ share a common framework, but differ in their choice of output primitive component and attribute component. PHIGS+ uses a

richer set of components which take illumination into account. PHIGS and PHIGS+ also use the same input components.

Immediately after the Blakeney meeting, the BSI held a Reference Model meeting which produced an approach to Reference Models combining the merits of the Component/ Framework approach and an earlier BSI approach to Reference Models. The new BSI approach[2] essentially arose from the recognition of the parallels between components and abstract data types. This work drew heavily on the work of Arnold and Reynolds concerning configurable models of graphics systems and their work with Duce in the formal specification of a GKS-like output pipeline.[56] The ideas were subsequently explored further at the meeting of SC24/WG1 held in Tucson, U.S.A. in July 1988. At this meeting consideration was given to how a functional standard might be expressed in a component/ frameworks setting and how input might be treated in this way. The first (without input) proved fairly straightforward at a fairly high level of abstraction, the second proved more demanding, in part because of the lack of a suitable notation in which to describe components.

This paper is the result of work done since that meeting by the authors to explore one particular notation, which appears to offer considerable promise for expressing the components required to describe the GKS input model. The technique used is Hoare's CSP notation. A short description of this notation follows.

3. Communicating Sequential Processes

Hoare motivates CSP by a discussion of objects in the world around us which act and interact with each other in some characteristic ways.[1] The aim of CSP is to describe this characteristic behaviour. The starting point for this is to decide what kinds of events or actions will be of interest and then to choose a different name for each kind.

As an illustration, consider a simple one-place buffer. Let the event *write* correspond to the user writing a value to the buffer and *read* correspond to reading the value of the buffer. Each name actually denotes an event class, there may be many occurrences of events in a single class, separated in time.

The sets of names of events which are considered relevant to the particular description of an object is called its *alphabet*. An object cannot engage in events outside its alphabet, but the presence of a name in an object's alphabet does not imply that the object will eventually engage in that event.

In CSP occurrences of events are regarded as instantaneous or atomic actions without duration. Extended or time consuming operations can be represented by pairs of events, one denoting its start, the second its finish. During the interval between start and finish other events may occur. Time is ignored in the basic CSP model and consequently it is not meaningful to ask if one event occurs simultaneously with another. When simultaneity is important (e.g. in synchronization), it is represented as a single-event occurrence. When it is not, potentially simultaneous events are allowed to occur in any order.

CSP also does not distinguish between events initiated by an object and those initiated by some agent outside the object. This avoidance of the concept of

causality leads to considerable simplification of the theory and its application.

The word *process* is used to stand for the behaviour pattern of an object. There is a convention in CSP that events are denoted by lower case words and processes by upper case words. Let x be an event and P be a process. Then the prefix notation

$$(x \rightarrow P)$$

describes an object which first engages in event x and then behaves exactly as denoted by P. This notation can be used to describe the entire behaviour of a process that eventually stops. However, for objects which continue to act and interact with their environment for as long as they are needed, and which contain repeating patterns of behaviour, this is not a convenient notation.

Consider a simplification of the 1-place buffer, an unchanging storage location which can be read. Denote the object by B, then the alphabet of B is

$$\alpha B = \{ read \}$$

An object which behaves like B after being read once would be described by

$$(read \rightarrow B)$$

The behaviour of this object is exactly like the original object B, which suggests a formulation

$$B = (read \rightarrow B)$$

This can be regarded as an implicit definition of the behaviour of B. The potentially unbounded behaviour of B is effectively defined as

$$read \rightarrow read \rightarrow read \rightarrow read \rightarrow ...$$

This method of process description only works if the right hand side of the equation starts with at least one event prefixed to all recursive occurrences of the process name. A process description beginning with a prefix is said to be *guarded*. In the specifications following, it is sometimes necessary to refer to the process which satisfies (i.e. is the solution of) such an equation. If $F(X)$ is a guarded expression containing the process name X, then it can be shown that the equation

$$X = F(X)$$

has a unique solution. This solution is denoted by

$$\mu X: F(X)$$

X is a bound variable whose name can be changed at will. In the example above, the solution of the recursion equation for B is

$$B = \mu X: (read \rightarrow X) = \mu Y: (read \rightarrow Y)$$

Many objects, including the one-place buffer with which this discussion started, allow their behaviour to be influenced by interaction with the environment in which they are placed. If x and y are distinct events,

$$(x \rightarrow P \mid y \rightarrow Q)$$

denotes a process which initially engages in either of the events x or y. After the first event has occurred, the process behaves as P if the first event was x or as Q if the event was y.

A description of a one-place buffer which first engages in a *write* event and then subsequently in either *read* or *write* events is

$$B = (write \rightarrow \mu X: (read \rightarrow X \mid write \rightarrow X))$$

The choice of which event will actually occur can be controlled by the environment within which the process evolves. The environment of a process may itself be described as a process. The complete system is itself a process whose behaviour is definable in terms of its component processes.

When two processes are brought together to evolve concurrently, it is usually intended that they should interact with each other. These interactions can be regarded as events in which both processes participate. The one-place buffer described above might be placed in the context of an application program which will alternately write and then read the buffer. Such a program can be described by the process AP

$$AP = write \rightarrow read \rightarrow AP$$

The notation

$$AP \parallel B$$

denotes the process which behaves like the system composed of the two processes AP and B interacting in synchronization as described.

When processes P and Q with differing alphabets are combined to run concurrently, events that are in both their alphabets require simultaneous participation of P and Q. However, events in the alphabet of P which are not in the alphabet of Q are no concern of Q. Such events may occur independently of Q whenever P engages in them. Similarly, Q may engage alone in events which are in the alphabet of Q but not of P. Examples of this will be seen in the GKS input model specifications following.

CSP introduces special notation for a particular class of events called *communications*. A communication is an event that is described by a pair

$$c.v$$

where c is the name of a channel on which communication takes place and v is the value of the message which passes. The set of all messages which P can communicate on channel c is defined

$$\alpha c(P) = \{ v \mid c.v \in \alpha P \}$$

If v is a member of $\alpha c(P)$, a process which first outputs v on channel c and then behaves like P is defined

$$(c!v \rightarrow P) = (c.v \rightarrow P)$$

The only event in which this process is initially prepared to engage is the communication event $c.v$.

A process which is initially prepared to input any value x which can be communicated over the channel c, and then behave like $P(x)$ is defined

$$(c?x \rightarrow P(x)) = (y{:}\{y \mid channel(y){=}c\} \rightarrow P(message(y)))$$

In the one-place buffer example given earlier, a more complete description would be

$$\begin{aligned} AP &= write!v \rightarrow read?v \rightarrow AP \\ B &= write?v \rightarrow B_v \\ B_v &= read!v \rightarrow B_v \mid write?v \rightarrow B_v \\ B &\parallel AP \end{aligned}$$

Notice also here the use of subscripts to indicate the value of the state associated with a process.

4. The GKS Input Model

4.1. Introduction

The GKS input model is based on the concept of logical input devices, providing the application program with an interface which abstracts physical input devices from a particular hardware configuration. The paper by Rosenthal et al[7] gives a detailed exposition of the GKS input model.

Logical input devices are described in terms of a *class*, *operating modes* and *attributes*. These are described below.

Conceptually, logical input devices are explained in terms of two processes, the *measure process* and *trigger process*.

Classes.

The class of a logical input device defines the type of the input value which is returned. The six logical input data types are:

1. *LOCATOR:* a position in world coordinates and the associated number of the normalization transformation used to convert back from device coordinates via normalized device coordinates to world coordinates.
2. *STROKE:* similar to LOCATOR except it represents a sequence of world coordinate positions rather than a single position.
3. *VALUATOR:* a real number in some range.
4. *CHOICE:* an integer representing a selection from a set of choices.
5. *PICK:* the name of a selected segment and an identifier indicating which set of primitives in the segment has been picked.
6. *STRING:* a character string.

A particular measure value of a logical input device is defined to be the value of the physical input device transformed by a measure mapping function. GKS does not

place any constraints on the realization of logical input devices in terms of physical devices. A CHOICE device could be realized by a keyboard and the operator has to type in the name of the menu item to be selected. In this case the physical input value (the string) is mapped to the corresponding CHOICE device value (integer selection number) by the measure process. The following table shows a possible relationship between strings typed and the value of the CHOICE logical input device.

""	NOCHOICE
"CREATE"	1
"REDRAW"	2
"DELETE"	3
"RETURN"	4

The measure process will always contain the current measure value of the logical input device. Usually, the measure value is echoed in some way on the screen (for instance, by echoing a cursor shape in the position that corresponds with the measure value).

How the measure value is mapped onto a value returned by a logical input device is defined differently for every input class.

Operating modes.

The operating mode indicates how the input value is obtained from the logical input device. The trigger process plays an important role in this. A trigger process is an independent, active process which for certain operating modes, when *triggered* by the user, indicates that the current measure value is to be returned to the application program.

There are three different operating modes:

- *REQUEST*. Logical input devices in REQUEST mode behave rather like FORTRAN READ. A request is made by the application program for a measure to be returned from a specified device. GKS waits until the operator has set the measure to the desired value and has activated the trigger.
- *SAMPLE*. In SAMPLE mode the current measure is returned whenever requested by the application program. No triggering is involved when a logical device is sampled so that the application program will immediately continue after issuing a sample call.
- *EVENT*. A number of input devices may be active together. Each time the trigger for a particular device is activated, the current measure value and data identifying the device are added to a single queue of input events for all the devices used in event mode. The application program can interrogate the queue to retrieve the input events. It is possible to couple more than one input device to the same trigger so that multiple events can be generated from a single trigger event.

The event queue is structured as a queue of event reports. The event queue is interrogated by the GKS function AWAIT EVENT. This function removes

the event report at the top of the queue, writes it into a buffer known as the *current event report* and returns the identification of the device which produced the report (workstation identifier, input class, logical input device number) to the application program. If the queue is empty, GKS is suspended until either input arrives (in which case the function behaves as before) or a timeout period expires (in which case input class NONE is returned), whichever happens first. GKS provides a set of functions, one for each device class, which return the logical input value contained in the current event report.

Attributes.

Attributes are used to parameterize the initialization of a logical input device. Most attributes have to do with how and where input devices produce echos on the screen. Attributes include initial values, prompt / echo types, activation modes and echo areas. Data records provide the application program a means to parameterize the logical input device in a device dependent manner. For instance, an entry in the data record can specify which mouse button will be used to trigger a locator device.

4.2. Structure of the Specification

This section describes the overall structure of the CSP specification of the GKS input model. The following sections elaborate the specification in detail. Figure 2 illustrates the overall structure.

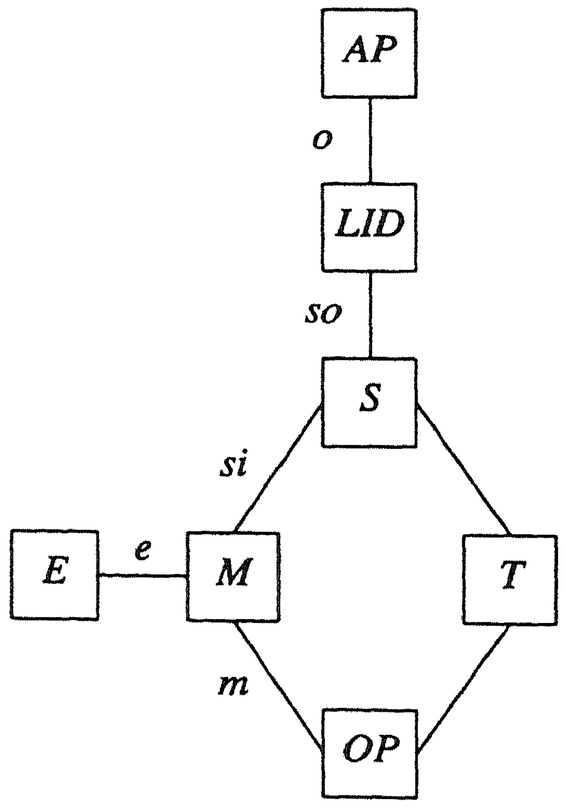


Figure 2: Structure of the Specification

The first attempt at a CSP specification of GKS input used different structures to describe each of the operating modes. By examining the resulting specifications; it was realized that each mode could be described in terms of five processes (*LID*, *S*, *E*, *M* and *T*). The following specifications are therefore presented within this framework; some of the components which populate this framework have different

descriptions in the different operating modes.

The specification starts from the realization that there are three key objects, the application program, the operator, and the logical input device which is the connection between the two. For simplicity we will only consider a single logical input device here. The application program and the operator provide the environment for the logical input device and to describe the system as a whole it is convenient to describe also what actions the operator may perform and what functions the application may invoke. These are all modelled as events in the alphabet of the CSP processes describing the application (*AP*) and the operator (*OP*).

The process *LID* describes the behaviour of a logical input device in terms of a measure process *M*, an echo process *E*, a trigger process *T* and a storage process *S*. It was the realization that each operating mode could be described in terms of processes of these kinds which led to the model presented here. In the case of SAMPLE mode the trigger process is null. It will be seen that the echo and measure processes are the same in all the operating modes. The different modes present different opportunities to the operator and application and have different storage components.

Each of the processes will now be discussed in turn for each of the operating modes. Section 4.7 discusses how this generalizes to the case of more than one device.

4.3. Application Program (AP)

The application program may set the mode of a logical input device and invoke functions appropriate to that mode. Some slight simplifications are made to the GKS model. GKS allows the application to set an initial value for a device in the appropriate workstation statelist. Here initialization is not considered as it is essentially orthogonal to the remainder of the specification. Secondly, when a device is placed in SAMPLE or EVENT modes, measure and trigger (in the case of EVENT mode only) processes are created immediately for the device and it becomes active. When a device is set into REQUEST mode, the measure and trigger processes are not created until the REQUEST <device class> function is invoked. Essentially the device is in a quiescent state until this latter function is invoked. The specification given here reflects this by explicitly introducing a *set–quiescent–mode* event.

The behaviour of the application program is described by the process:

$$
\begin{aligned}
AP = {} & set\text{–}mode\text{–}quiescent \rightarrow AP \\
& \mid (request \rightarrow REQUEST \parallel E_0 \parallel M_0 \parallel T \parallel OP^R \parallel S^R \parallel LID^R) \\
& \mid (set\text{–}mode\text{–}sample \rightarrow SAMPLE \parallel E_0 \parallel M_0 \parallel OP^S \parallel S^S \parallel LID^S) \\
& \mid (set\text{–}mode\text{–}event \rightarrow EVENT \parallel E_0 \parallel M_0 \parallel T \parallel OP^E \parallel S^E_{<>} \parallel LID^E)
\end{aligned}
$$

This specification states that the logical input device can be set into the quiescent mode, REQUEST mode, SAMPLE mode or EVENT mode. The *request* event corresponds to an invocation of the GKS REQUEST <device class> functions. From quiescent mode it can be set into any of the other modes. From the other modes, the behaviour is described by a composition of control (*REQUEST*, *SAMPLE*, *EVENT*), storage, measure, trigger echo and operator processes. Superscripts are used to denote processes which are different for the different modes. Subscripts

are used to denote the initial states of processes.

4.4. REQUEST Mode.

4.4.1. The operator process, OP^R.

The operator of a logical input device can change the value of the device's measure process or fire the trigger process. The trigger firing can be represented as an event *trigger* and setting a new measure value by outputting a value v on channel m. The behaviour of the operator is then characterized by the process OP^R defined as:

$$OP^R = (m!v \rightarrow OP^R) \mid (trigger \rightarrow STOP_{OP^R})$$

This means that the operator can choose to change the value of the measure or fire the trigger. Once the trigger has fired, the interaction with that device terminates.

4.4.2. The application program, *REQUEST*.

The application program can receive a logical input value from a channel o. The interaction with the device then terminates and the device returns to the quiescent state. The behaviour of the application program in REQUEST mode is described by the process:

$$REQUEST = o?v \rightarrow AP$$

4.4.3. The measure process, M.

The behaviour is described by:

$$M_v = (m?v' \rightarrow e!v' \rightarrow M_{v'}) \mid (get\_m \rightarrow si!v \rightarrow M_v)$$

The communication over channel m corresponds to the operator setting a new measure value, which is then transmitted to the echo process over channel e. The logical input device may request the current measure value by the event *get_m*. The value is returned along channel *si*. The special value 0 (process M_0) denotes the measure process initialized to the initial measure value recorded in the workstation state list.

Strictly speaking, the value communicated over channel m from the operator is a *physical* input value. The value associated with the state of the measure process and communicated along channel *si* is a logical input value. If f denotes the physical to logical value mapping, the first choice in the behaviour above could be written as :

$$M_v = (m?v' \rightarrow e!f\,(v') \rightarrow M_{f(v')})$$

This also makes it clear that it is the logical rather than the physical value which is echoed (see 4.4.5). The measure process is the point in the specification where the physical to logical mapping occurs. For example, if a keyboard is used to implement a CHOICE device, f might be the function:

$$f = \{\ \text{""} \rightarrow \text{NOCHOICE}, \text{"CREATE"} \rightarrow 1, \text{"REDRAW"} \rightarrow 2,$$
$$\text{"DELETE"} \rightarrow 3, \text{"RETURN"} \rightarrow 4\ \}$$

The mapping could be specified precisely using a notation such as Z^8 in

combination with CSP,[9] but that would take us rather beyond the scope of this present paper. The intention should, however, be clear.

4.4.4. The trigger process, *T*.

The behaviour of the trigger process is just :

$$T = trigger \rightarrow trigger_s \rightarrow T$$

The event *trigger* is shared by the operator and the trigger process. The event $trigger_s$ is shared by the trigger and storage processes.

4.4.5. The echo process, *E*.

The echo process can receive a value on channel *e* (from the measure process), and echo it on the display.

$$E_v = e?v' \rightarrow E_{v'}$$

The special value 0 (process E_0) denotes the echo process which echoes the initial measure value recorded in the workstation state list.

4.4.6. The logical input device, LID^R.

The logical input device reads a logical input value from the storage channel, *so*, and delivers it to the application program on channel *o*. The behaviour is defined by:

$$LID^R = get\_s \rightarrow so?v \rightarrow o!v \rightarrow LID^R$$

4.4.7. The storage process, S^R.

The storage process is initiated by the event *get_s*, awaits the trigger firing event $trigger_s$, initiates the event *get_m* to get the current value of the measure process, then reads the current value of the measure on channel *si* and transmits this value to the LID^R process on channel *so*. This behaviour is defined by:

$$S^R = (get\_s \rightarrow trigger_s \rightarrow get\_m \rightarrow si?v \rightarrow so!v \rightarrow S^R)$$

The storage process is effectively providing a one-place buffer between the measure process and the application program. The value of the measure process transmitted to the application is the value current when the trigger fires.

4.4.8. Remarks on REQUEST mode.

- The table below shows the alphabets of each of the processes in the specification. The left hand column lists all the possible event classes. An '×' underneath a process indicates that the corresponding event is in the alphabet of that process.

	OP^R	*REQUEST*	M	T	LID^R	E	S^R
m.v	×		×				
trigger	×			×			
o.v		×			×		
e.v			×			×	
get_m			×				×
si.v			×				×
$trigger_s$				×			×
get_s					×		×
so.v					×		×

- The application process, *REQUEST*, does not exhibit any choice. The choice in the system is made by the operator, who can choose when to vary the measure and when to fire the trigger. This is shown clearly in the specification.
- The specification also shows clearly that the device is put into REQUEST mode by the application. Once the trigger has fired, a logical input value is returned to the application program and interaction with the device ceases until it is put into REQUEST mode again by the application program at which point a new measure process is created.

4.5. SAMPLE Mode.

4.5.1. The operator process, OP^S.

The operator of a logical input device can only change the value of the device's measure process. The behaviour of the operator is then characterized by the process OP^S defined as:

$$OP^S = (m!v \rightarrow OP^S)$$

4.5.2. The application program, *SAMPLE*.

The application program can engage in three events, *set–mode–quiescent* which returns the device to the quiescent state, *sample* requesting a logical input value from the device, and receiving a logical input value from it on channel *o*. The behaviour of the application program is described by the process:

$$SAMPLE = (\mu X{:}sample \rightarrow o?v \rightarrow X) \mid (set\text{–}mode\text{–}quiescent \rightarrow AP)$$

Notice that once the device is in SAMPLE mode, the application program can sample the device any number of times before returning it to the quiescent state.

4.5.3. The measure process, M.

The measure process is exactly the same as for REQUEST mode. The behaviour is described by:

$$M_v = (m?v' \rightarrow e!v' \rightarrow M_{v'}) \mid (get\_m \rightarrow si!v \rightarrow M_v)$$

4.5.4. The echo process, E.

The echo process is exactly the same as for REQUEST mode. The echo process can receive a value on channel *e*, and echo it on the display.

$$E_v = e?v' \rightarrow E_{v'}$$

4.5.5. The logical input device, LID^S.

The application can sample the logical input device (*sample*), reading a logical input value from the storage process and delivering it to the application program on channel *o*. The behaviour is described by:

$$LID^S = sample \rightarrow get\_s \rightarrow so?v \rightarrow o!v \rightarrow LID^S$$

The event *sample* is also contained in the alphabet of the process *SAMPLE*. The description of SAMPLE input differs from that of REQUEST input because in the latter the device reverts to the quiescent state after one value has been returned to the application program, whereas in SAMPLE input the device remains active and may be sampled any number of times before being explicitly returned to the quiescent state.

4.5.6. The storage process, S^S.

The storage process is similar to the process S^R in REQUEST mode, except that there is no involvement of the trigger process. The value delivered is the value current when the application invokes the sample function.

$$S^S = (get\_s \rightarrow get\_m \rightarrow si?v \rightarrow so!v \rightarrow S^S)$$

4.5.7. Remarks on SAMPLE mode.

- The table below shows the alphabets of the processes in this specification. Note that the trigger process is not used in the specification.

	OP^S	*SAMPLE*	*M*	LID^S	*E*	S^S
m.v	×		×			
o.v		×		×		
e.v			×		×	
get_m			×			×
si.v			×			×
get_s				×		×
so.v				×		×
sample		×		×		
set–mode–quiescent		×				

- Figure 3 illustrates the structure of this specification. It can be seen that this is just a special case of the general case shown in Figure 2.

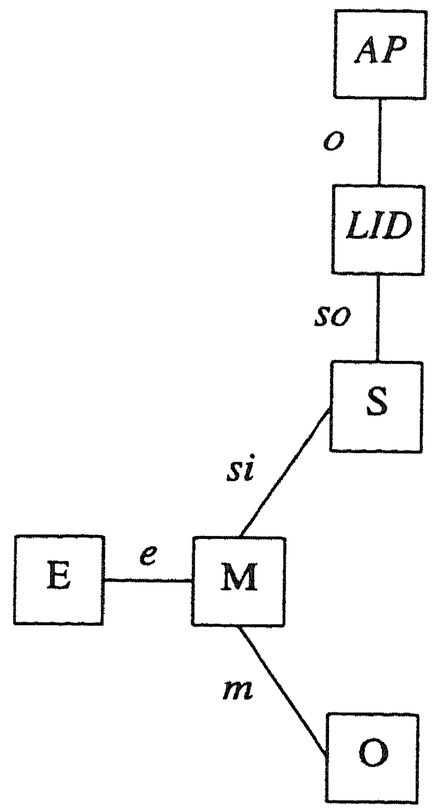


Figure 3: Structure of the SAMPLE Specification

- The application program can decide when to sample the device, that is when to invoke the SAMPLE <device> function (event *sample*). The operator can alter the value of the device's measure, but that is the only option offered to the operator.
- The device remains in SAMPLE mode until the application chooses to return it to the quiescent state. In GKS the application would set the device directly into one of the other operating modes or reinitialize the device in SAMPLE mode. Here we select a new mode in two stages, returning first to the quiescent state before selecting the new mode. This gives a tidier specification.
- The measure process in SAMPLE mode is identical to the measure process in REQUEST mode. The form of the storage processes in these two modes shows clearly the role of the trigger in REQUEST mode input.

4.6. EVENT Mode.

4.6.1. The operator process, OP^E.

The operator of a logical input device in EVENT mode can change the value of the device's measure process or fire the trigger process. The trigger firing can be represented as an event *trigger* and setting a new measure value by outputting a value v on channel m. The behaviour of the operator is then characterized by the process OP^E defined as:

$$OP^E = (m!v \rightarrow OP^E) \mid (trigger \rightarrow OP^E)$$

4.6.2. The application program, *EVENT*.

The application program can engage in three events, *set–mode–quiescent* which returns the device to the quiescent state, *await–event* requesting a logical input value from the storage and receiving a logical input value from it channel *o*. The behaviour of the application program is described by the process:

$$EVENT = \mu\, X.\ (await\_event \rightarrow o?v \rightarrow X) \mid (set\text{–}mode\text{–}quiescent \rightarrow AP)$$

4.6.3. The measure process, M.

This is identical to the measure process in SAMPLE and REQUEST modes. The behaviour is described by:

$$M_v = (m?v' \rightarrow e!v' \rightarrow M_{v'}) \mid (get\_m \rightarrow si!v \rightarrow M_v)$$

4.6.4. The trigger process, T.

This is identical to REQUEST mode.

$$T = trigger \rightarrow trigger_s \rightarrow T$$

4.6.5. The echo process, E.

The echo process can receive a value on channel *e*, and echo it on the display. This is identical to REQUEST and SAMPLE modes.

$$E_v = e?v' \rightarrow E_{v'}$$

4.6.6. The logical input device, LID^E.

The logical input device reads a logical input value from the storage channel, *so*, and delivers it to the application program on channel *o*. The behaviour is described by:

$$LID^E = await\_event \rightarrow get\_s \rightarrow so?v \rightarrow o!v \rightarrow LID^E$$

There is a similarity with the process LID^S in that the device remains active when await event has completed.

4.6.7. The storage process, S^E.

This process represents the major difference between EVENT mode and REQUEST and SAMPLE modes. In the other two modes the storage process does not retain values. In EVENT mode, trigger firing results in values being sent to the storage process. Their consumption awaits a *get_s* event from the logical input device.

In GKS, the storage discipline is a queue. Values are added to one end of the queue by *get_m* and removed from the other by *get_s*. Subscripts to the process name are used to indicate the state of the queue before and after each event which modifies the queue.

The AWAIT EVENT function in GKS returns a NONE value to the application program if the queue is empty when the function is invoked and no input is

added to the queue before a timeout period has expired. Timeout is indicated in this model by the event *time_out*. It is not further defined here.

In GKS the AWAIT EVENT function returns the identification of the device from which the event at the top of the queue originated and moves this event description to the current event report. Events are retrieved from the current event report by GET <device class> functions, one for each type of device. In this specification, the current event report is not described, it is merely stated that the logical input value is returned to the application program through channel o.

$$
\begin{aligned}
S^E_{s<v>} &= (\, get\_s \rightarrow so!v \rightarrow S^E_s) \\
S^E_s &= (\, trigger_s \rightarrow get\_m \rightarrow si?v \rightarrow S^E_{<v>s}) \\
S^E_{<>} &= (\, get\_s \rightarrow (\, time\_out \rightarrow so!NONE \rightarrow S^E_{<>} \\
&\quad \mid trigger_s \rightarrow get\_m \rightarrow si?v \rightarrow so!v \rightarrow S^E_{<>})) \\
&\quad \mid (\, trigger_s \rightarrow get\_m \rightarrow si?v \rightarrow S^E_{<v>})
\end{aligned}
$$

4.6.8. Remarks on EVENT mode.

- The table below shows the alphabets of the processes in the specification of EVENT mode input.

	OP^E	*EVENT*	*M*	*T*	LID^E	*E*	S^E
m.v	×		×				
trigger	×			×			
o.v		×			×		
e.v			×			×	
get_m			×				×
si.v			×				×
$trigger_s$				×			×
get_s					×		×
so.v					×		×
set–mode–quiescent		×					
await–event		×			×		
timeout							×

- The application process *EVENT* is very similar to the corresponding process for SAMPLE mode input. The application can decide when to ask for an input value. In the EVENT case however, the storage component does not immediately request the current value of the measure; instead it looks to see if a value is stored or if not, awaits the arrival of a new event until a timeout expires.
- The measure and trigger processes are identical to those in REQUEST mode. The operator process is similar except that in EVENT mode a trigger firing does not terminate the interaction with the device, so the operator may generate more than one value. It is an application program action which terminates the device. The difference in behaviour is accounted for by the different storage components in the two systems and the different application process behaviours.

4.7. Multiple Devices

Although the specification given here only considers a single logical input device, multiple devices can easily be described by amending the description of the process *AP* and introducing clones of the other processes, with appropriate names (for example prefixed by the device name which is unique). If the devices are independent, then the event names need to be prefixed by the device name to make them unique. If the devices are not independent, for example if two devices share the same trigger, then events which are common have the same name. Recall that in CSP events which are in the alphabets of two processes require their simultaneous participation. Thus in this example, the trigger firing would automatically go to both devices because it is in the alphabets of the operator and device processes.

5. Extensions

5.1. Introduction.

In this section some simple extensions to the input model are discussed. Essentially these involve replacing components in the framework described here. The extensions to be discussed are logical input device types and composite devices, storage strategies and the interaction between input and output.

5.2. New device types.

As noted earlier, the GKS input model defines six classes of logical input device, each corresponding to a particular type of input value. The type of the logical input value associated with the logical input device has not featured at all in the specification given here. In fact the specification is completely independent of the type of the input value, provided that the type is consistent throughout the specification. This means that new classes of logical input device can be introduced very easily, merely by substituting components which can handle the new datatype. No change is needed to the specification to describe such systems.

In EVENT mode input, GKS allows any particular trigger to be associated with more than one measure process. Then when the trigger fires, multiple event reports are added to the queue and marked as simultaneous events. There is no analogue of this mechanism in SAMPLE or REQUEST modes. This restriction is unfortunate because this mechanism provides a nice way to accommodate devices such as the Tektronix cross-hairs and the mouse, each of which can be viewed as a composite of a LOCATOR and CHOICE device.

Such devices can be incorporated into this specification fairly easily. There are two ways to do this, the first involves generalizing the framework given here to incorporate multiple measure processes, one for each of the basic measure types which make up the device. A slightly more elegant way to do this comes from the recognition that there is one measure value associated with the device, which happens to be composed to two basic types, LOCATOR and CHOICE. We will illustrate this with the mouse device. Suppose we have a three-button mouse. The value of the device can be expressed as a LOCATOR and CHOICE value, as an ordered pair of the form (l, c), where l is of type LOCATOR, indicating the position of the

device, and c is of type CHOICE, indicating which, if any, of the buttons are depressed. The specification of such a device can be obtained from that given here by consistently substituting (l, c) for v throughout the specification.

Note that the operator now generates events of the form

$$m!(l, c)$$

so the operator has been implicitly retrained! Notice that this works perfectly well in SAMPLE and REQUEST modes as well as EVENT mode and that the notion of simultaneous events has been replaced by the simpler notion of cartesian product datatypes. All we need is a way of delivering values of this type to the application program. If the only mechanism for doing this in a particular programming language involves a notion of simultaneous events, then this should not be cluttering the specification for more flexible programming languages.

The three buttons on the mouse might also act as triggers for the device. Suppose the events $trigger_1$, $trigger_2$, $trigger_3$, denote the action of depressing the respective buttons. If each of these can trigger REQUEST or EVENT input, we can describe this by substituting a process T with the following description:

$$\begin{aligned} T = &(trigger_1 \rightarrow trigger_s \rightarrow T) \mid (trigger_2 \rightarrow trigger_s \rightarrow T) \\ &\mid (trigger_3 \rightarrow trigger_s \rightarrow T) \end{aligned}$$

5.3. Storage strategies.

The second extension we discuss here concerns storage strategies. It has been seen that the form of the process S plays an important role in determining the overall system behaviour. Interesting systems can be obtained by taking the framework given here and substituting a different storage component. A simple example will illustrate the point. In GKS EVENT mode input, the storage strategy is a queue. For whatever reason, one might want to use a last-in-first-out strategy or stack, instead. A component to do this has a very simple description:

$$\begin{aligned} S^E_{s<v>} &= (get\_s \rightarrow so!v \rightarrow S^E_s) \\ S^E_{<>} &= (get\_s \rightarrow (time\_out \rightarrow so!NONE \rightarrow S^E_{<>} \\ &\quad \mid trigger_s \rightarrow get\_m \rightarrow si?v \rightarrow so!v \rightarrow S^E_{<>})) \\ &\quad \mid (trigger_s \rightarrow get\_m \rightarrow si?v \rightarrow S^E_{<v>}) \\ S^E_s &= (trigger_s \rightarrow get\_m \rightarrow si?v \rightarrow S^E_{s<v>}) \end{aligned}$$

This is a simple example but hopefully it illustrates the point that by changing components, usefully different functionality can be obtained.

5.4. The relationship between input and output.

A workstation display surface can be represented as a process D, which can accept display events generated by the application program (through the output pipeline of the graphics system). This behaviour could be described by:

$$D_p = (display?p' \rightarrow D_{p\&p'})$$

Here the process D is receiving communications along channel *display*. The values passed are rendered output primitives, denoted by the variable p'. The primitive p'

is combined with the existing state of D to yield a new state $p\ \&p'$ which represents the display space incorporating the new output primitive. The operator '&' is not further defined here.

The echo process also generates graphical output, and the interaction between the echo and display processes could be expressed by:

$$D_p = (\ display?p' \rightarrow D_{p\&p'}\ ) \mid (\ echo?p' \rightarrow D_{p\&p'}\ )$$
$$E = (\ e?v' \rightarrow echo!r(v') \rightarrow E\ )$$

Here the function r generates the rendered primitive corresponding to the current measure value v'. In reality the operations necessary to update an echo are more complicated than those given here, but the above behaviours should give a flavour for how this approach could be used.

Following on from this, it is clear how interactions between other processes in the input model and the display process could be described. It should also be clear that if the output pipeline is described in a similar manner, then interactions between input and output can be readily described; for example the echo process might interact with the output pipeline at a higher level using facilities in the output pipeline to construct the graphical object representing the echo. It should also be clear that this approach could be used to describe systems in which values derived from input devices are used to control the graphical output, for example to determine the parameters of transformations, by introducing appropriate communications. This suggests some interesting directions for further work.

6. Conclusions

This paper has demonstrated how a Component/ Frameworks style description of the GKS input model can be given in the CSP notation. The specification clearly demonstrates how it is possible to substitute components within a framework and some simple interesting extensions to the model have been described.

The exercise has deepened the authors' understanding of the input model, and it is hoped the readers' also, by demonstrating the role of the storage component in each of the input modes and showing precisely where, and by whom, choices may be made in each of the operating modes.

7. Acknowledgements

The authors are grateful to colleagues in ISO/IEC JTC1/SSC24/WG1 for fruitful discussions on how to describe input in a component framework setting, in particular to Tom Morrissey.

References

1. C.A.R. Hoare, *Communicating Sequential Processes,* Prentice-Hall International, London (1985).
2. ISO, ''Contributions on a Computer Graphics Reference Model,'' ISO / IEC JTC1 / SC24 N4, ISO Central Secretariat (1988).
3. ISO, ''The Use of the Component / Framework Description Technique in the

Specificaiton of Computer Graphics Standards,'' ISO / IEC JTC1 / SC24 N178, ISO Central Secretariat (1988).

4. G.S. Carson, ''The Future of ISO Graphics Standards,'' *IEEE Computer Graphics and Applications*(7), pp. 82-83 (1988).

5. D.B. Arnold, G. Hall, and G.J. Reynolds, ''Proposals for Configurable Models for Graphics Systems,'' *Computer Graphics Forum* **3**(3), pp. 201-208 (1984).

6. D.B. Arnold, D.A. Duce, and G.J. Reynolds, ''An Approach to the Formal Specification of Configurable Methods of Graphics Systems,'' pp. 439-463 in *European Computer Graphics Conference and Exhibition*, ed. G. Marechal, North-Holland, Amsterdam (1987).

7. D. Rosenthal, J. C. Michener, G. Pfaff, R. Kessener, and M. Sabin, ''The detailed semantics of graphics input devices,'' *Computer Graphics* **16**(3), pp. 33-38 (July 1982).

8. I. Hayes, *Specification Case Studies,* Prentice-Hall International, London (1987).

9. J.C.P. Woodcock, ''A Strategy for the Correct Implementation of Communicating Processes,'' *Le Premier Seminaire International sur le Genie Logiciel*, Oran, Algeria (1988).

Event-based.constraints: coordinate.satisfaction -> object.state

Remco C. Veltkamp and Edwin H. Blake

Abstract

This paper is about systems support for interactive computer graphics. The aim is to integrate the two major approaches to dealing with complexity in the design and implementation of such systems, namely, constraints and object-oriented programming.

The use of constraints in managing the complexity of designing interactive graphics systems and the use of object-oriented methods for describing simulations and systems of concrete objects have been two natural methods for building large complex graphics systems. This widely acknowledged way of dealing with the complexities of modelling and interface design has had disappointingly little practical impact.

We have identified a major cause for the lack of progress in combining constraints and object-oriented methods. We believe that a proper solution to the problem requires a radical separation of the constraint system and the normal object-oriented framework. In this paper we propose a way of dealing with these problems by means of two orthogonal communication strategies for objects: events and messages.

1 Introduction

The use of constraints in managing the complexity of designing interactive graphics systems and graphical user interfaces dates back to the earliest days of interactive graphics — consider Sutherland's Sketchpad from the early sixties [15]. Object-oriented methods with their usefulness for describing simulations and systems of concrete objects have been a natural method for building large complex graphics systems. The great benefits of class inheritance in user interface design is well recognized and is finding increasing commercial application.

The desirability of combining object-oriented methods and constraints has a similar venerable and distinguished lineage — a major system from the late seventies was Borning's ThingLab [5] which was written in Smalltalk. On the whole, and rather surprisingly, this widely acknowledged way of dealing with the complexities of modelling and interface design has had disappointingly little practical impact.

If one plans to use object-oriented methods to manage complexity in building interactive computer graphics systems, and if one also wants to provide constraints as a tool to manage the complexity of analysis, design, and interaction, then constraints and objects must be combined in a harmonious and coordinated whole. However, the integration of constraints and objects leads to conflicts in programming methodologies [10], and we believe that this is one of the major causes of the lack of application of constraints and their low profile within the mainstream object-oriented approach (another major problem is the difficulty in providing powerful and general constraint solving methods).

We distinguish two incompatibilities between constraints and object-oriented concepts:

- a constraint solver looks at, and sets, the constrained objects' internal data, which conflicts with the data encapsulation concept in the object-oriented paradigm;
- object-oriented programming is imperative, while constraint programming is declarative.

2 Constraints and data encapsulation

To illustrate the problem, let us look at an example, say from a geometric figure editor. Suppose we have a circle `C` with data fields `x`, `y` and `r` representing the centre and radius, an axis parallel rectangle *R* with data fields `l`, `r`, `b`, and `t` representing the left, right, bottom, and top sides (see figure 1). Suppose further that we have the constraints that the objects touch each other and have equal area.

We could express our constraints as follows:

```
touching:          C.x+C.r=R.l
areas equal:  π×C.r×C.r = (R.t-R.b)×(R.r-R.l)
```

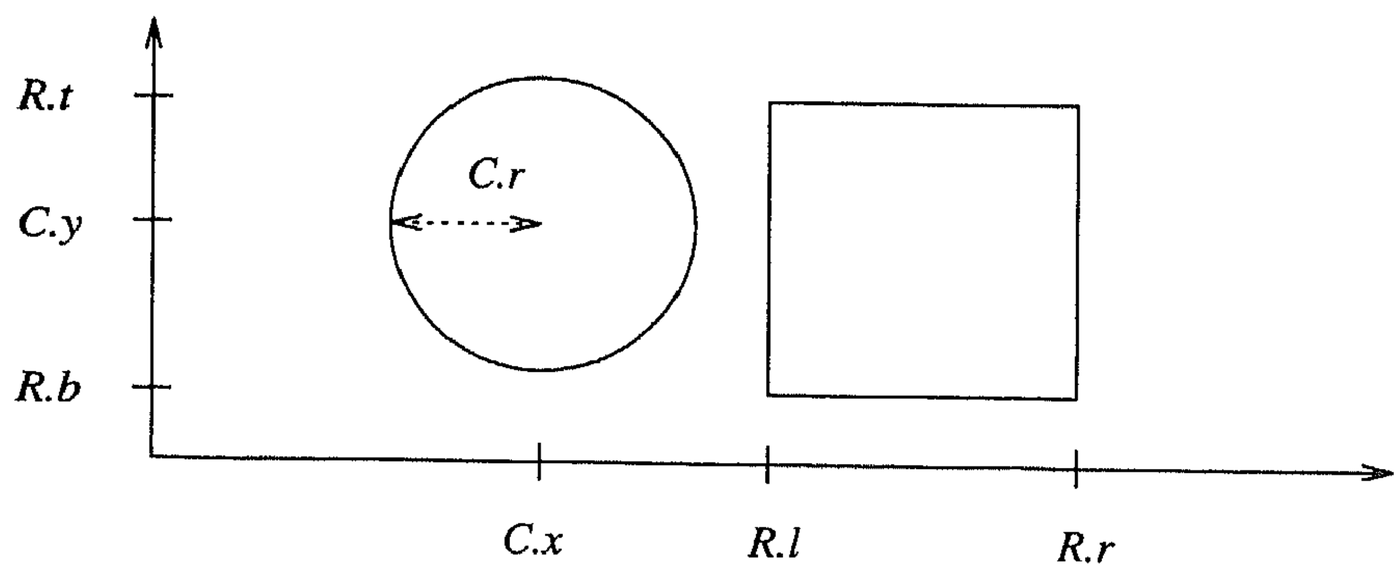


Figure 1: The circle and the rectangle must touch and must have equal area.

A constraint solver may come with the following solution (see figure 2):

```
C.x=5, C.r=1
R.l=6, R.r=7
R.b=0, R.t=π
```

Encapsulation is first violated by the constraint expressions, and then by expressing the solution. To avoid this problem, approaches based on message passing have been proposed. In [12], the methods of an object that may violate constraints are guarded by so-called propagators. The propagators send messages to other objects to maintain the constraints. This technique is similar to the pre- and postcondition facilities in Go [8] [6]. This approach is limited to constraint maintenance (i.e. truth maintenance, as opposed to starting with an inconsistent situation that is then resolved).

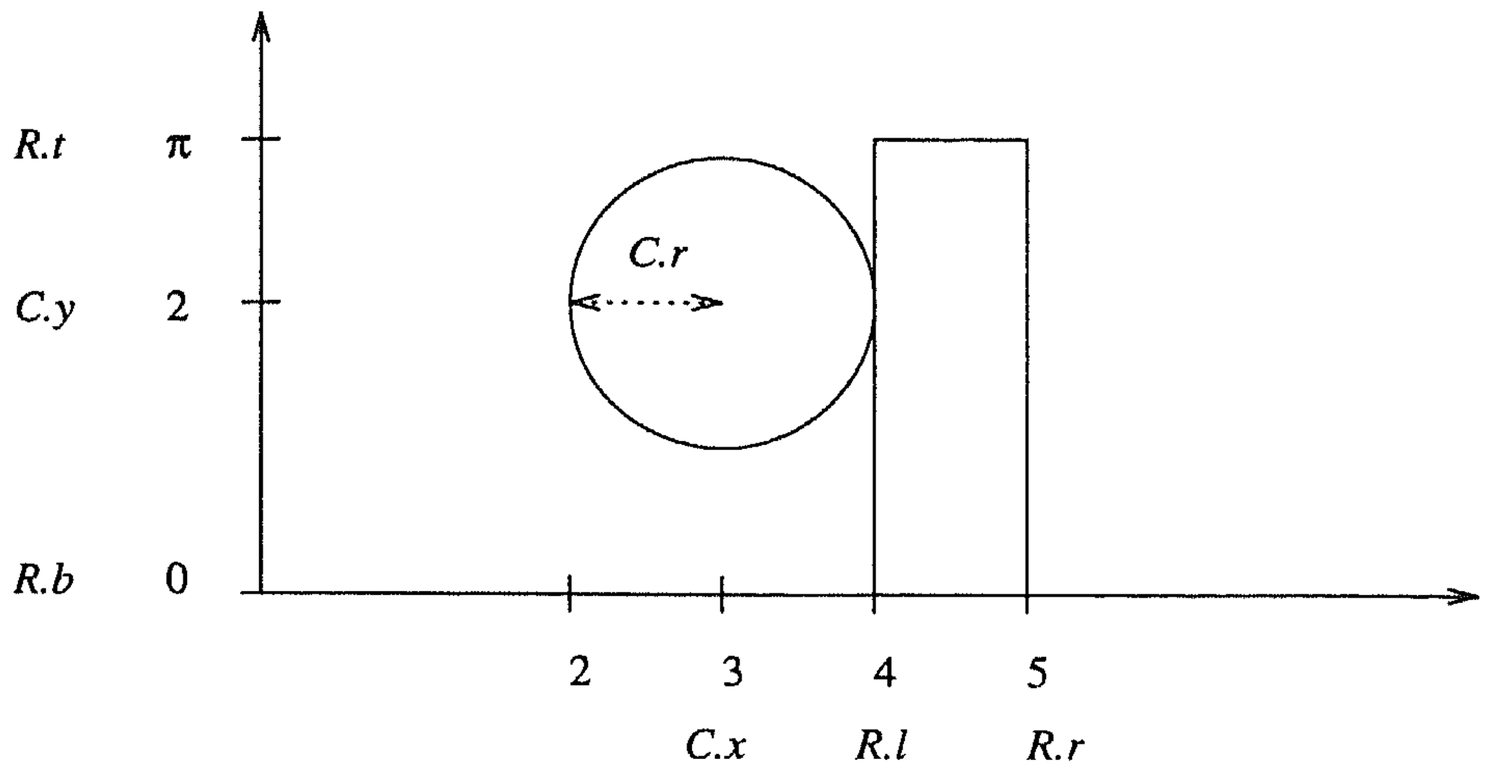


Figure 2: The circle and the rectangle touch and have equal area.

A more powerful technique is presented in [17]. The constraint solver produces a set of programs that solve constraints which are stated in the form of equations in terms of messages to the objects. It translates a declarative constraint into procedural solutions in terms of messages back to objects. This amounts to the constraint system maintaining a view on the states of the objects. The constraint solver is then able to reason about the current state of the objects and propose procedures to fulfill the constraints.

For the above example, typical constraint equations would be:

```
left(R)=right(C)
area(C)=area(R)
```

And a possible solution is:

```
scale(C,distance(R)/radius)
scale(R,area(C)/area)
```

The problem here is that the second method destroys the first constraint, which must be repaired. Doing so destroys the second constraint, etc. The real problem is the local character of the solution. More powerful solution are necessarily global in nature. The danger is that all objects need methods to get and set their internal data. This however, allows every other object to get and set these values, which is clearly against the object-oriented philosophy.

One way to restrict this, is to have an object allow value setting only when its internal constraints remain satisfied (see [14]). A constraint could be made internal by constructing a 'container object', which contains the constraint and the operant objects, but this does not solve the basic problem. In particular, the state of active objects cannot be changed without their explicit cooperation. (Active objects, or actors, conceptually have their own processor and behave autonomously, which is typical in animation and simulation.) Another approach is to limit access to private data to constraint-objects or the constraint solver-objects only. For example C++ provides the 'friend' declaration to grant functions access to the private part of objects. This is also comparable to the approach taken by [7], where special variables (slots) are accessible by constraints only. One can argue that encapsulation is still violated (and specifically that the C++ friend construct is not intended to allow changing the state of an object). Alternatively one can see constraints more as a means to manipulate information in an orderly and restricted way, than that they violate the data encapsulation principle [16], i.e. they provide controlled violation [4].

3 Relations in the Object-Oriented Paradigm

It should be pointed out that the problem of integrating constraints in the object-oriented paradigm is a sub-class of the problem of expressing relations in general in object-oriented programming. Constraints are functional relations that restrict the values which variables in an object can assume. One simple way of avoiding the encapsulation problems associated with constraints would be to include the constrained objects as part of some larger container object. It should be obvious by now that this is no real solution [3].

However, we would expect that a good approach to combining constraints and objects would provide interesting and useful pointers to dealing with problems of aggregation, parts and wholes, and inter-object relationships in general. This in turn has clear connections with object-oriented database research.

4 Imperative vs. Declarative

Object-oriented languages are imperative, and thus use a notion of state, particularly represented by objects. On the other hand, pure constraint languages are declarative, and thus specify one single timeless state: the solution to the specified problem. Both paradigms can be combined as in [9], where an imperative assignment to a variable sets a value at one moment in time, and a declarative constraint dictates a value from that moment on.

However, active objects, or actors, behave totally independently and do not by themselves need to have a notion of some sort of global time. This holds in particular in simulation and animation applications if objects are modelled as concurrent autonomous entities. One aspect of time, however, is the synchronization of objects, such as the constraint that actions of objects take place in intervals that must overlap, or have an explicit ordering. Another type of constraints on time is for modelling object behaviour during the life time of the object.

One important issue involved with constraints and time is that if the solution depends on the order in which constraints are solved, then some of the declarative semantics is destroyed.

5 Combining Objects and Constraints

The justification for combining objects and constraints derives from the fact that it addresses the problems of complexity in large interactive graphical systems which arises on two fronts. The first is the complexity inherent in specifying the behaviour of animations and interactions with many components or objects. Constraints allow the declarative modelling of the behaviour of such systems. The second front is the complexity due to the fact that we are dealing with large software systems. Sound software engineering principles, such as data encapsulation, are needed to cope with large complex software systems.

It appears that all constraint systems in an object-oriented environment infringe the data encapsulation principle to some extent. The debugging of the constraint *satisfaction* routines, which have global effects, is the responsibility of the system programmer who provides the whole interactive graphical programming environment. At least the responsibility for integrity is shifted from the constraint user to the constraint system implementor. A problem that remains is the difficulty of debugging a constraint *specification*, due to the global effects of constraints. However, these global effects should be contained within a declarative constraint programming environment where the well known techniques of declarative software engineering are applicable (e.g., provability, executable specifications).

The time complexity of constraint satisfaction depends on both the domain and the kind of constraints. For example, linear constraints over real numbers can be solved in polynomial time, discrete constraint satisfaction problems are NP-complete, a single polynomial constraint of degree higher than four does not even have an analytical

solution, and the complexity for integer polynomials of degree greater than two is still unknown. An interesting conjecture is that in the absence of global information of some kind, "interesting" constraint resolution will require exponential time [personal communication, Wilk]. It might be interesting to prove the NP completeness of an identified class of constraint resolution methods under the assumption of strict data encapsulation.

Concluding, powerful constraint solvers are global in nature and are hard to integrate with objects. Wilk's solution [17] is too complex to be really useful, and the global view on the object states does not reduce resolution complexity. Rankin's approach [14] does not allow powerful constraint solvers. The integration of Freeman-Benson [9, 10] has been taken about as far as it can in terms of efficiency. By contrast, we believe that it is worthwhile to explore a solution that keeps the paradigms distinct and does not compromise the benefits which they severally confer.

6 Event-based constraint handling

We believe that a proper solution to the problem requires a radical separation of the constraint system and the normal object-oriented framework. In this paper we propose a way of dealing with these problems by means of orthogonal communication strategies for objects. These are events and data streams on one hand, and messages on the other hand.

Events are globally broadcast communications which can be received selectively. When they are received, events cause a pre-emptive invocation of routines (interrupts). Events can be generated by state changes in objects. A *stream* is a connection between an output and an input port of processes, for example objects. Coordinators determine how these objects are interconnected by streams and how their interaction pattern changes during the execution life of the system. *Messages* are the normal communications between objects in the object-oriented sense.

For the modelling of the interaction pattern we use the Manifold model of coordination [2]. The focus of this model is on the coordination of processes and on their communication, not on the computations performed by some of the processes. These processes are considered as black boxes whose behaviour is abstracted to their input and output. The communication is supported by two mechanisms: data-flow streams and event broadcasting. The data-flow streams form a network of streams, linking input and output ports of the processes and carrying the units exchanged between them. The event broadcasting mechanism provides control on the dynamical modification of the data-flow network.

Atomic processes are external for Manifold, and atomic in the sense that they are considered as black boxes of which no internal feature or behaviour is known. At the level of Manifold, they cannot be decomposed further than their input and output channels. An atomic process can:

- raise an event,

- take a unit from a stream connected to an input port,
- put a unit out to the streams connected to an output port.

Streams carry units from the output port to the input port. There is no assumption about the contents of units, this is left to computations in atomic processes. A 'coordinator' is a process that sets up and breaks down streams between processes, i.e. a data-flow network. When an event is raised the previous network is dismantled and the new network is set up.

In the syntax of Manifold, `ev.obj` denotes the event `ev` raised by object `obj`, and `obj1.out -> obj2.in` denotes the linking of output port `out` of `obj1` to the input port `in` of `obj2` by a stream. `(a,b)` is the parallel composition of `a` and `b`, where `a` and `b` are processes or streams. The full syntax is described in [1]. (This syntax is also used in the title.)

A possible coordinator for a global solution of the above example may partially look as follows:

```
touch_coord(cir,rec,touch)
process cir, rec, touch.
{ event wait.

  start:                              do wait.

  change.cir:                    (cir->touch.in1, rec->touch.in2).
  change.rec:                      (cir->touch.in1, rec->touch.in2).
  satisfied.touch:     do wait.
  solved.touch:             (touch.out1->cir,touch.out2->rec).

  wait:                                    (cir,rec).
}
```

Event `change` from either `cir` or `rec` causes the creation of a communication network from the constraint operands to the constraint (see figure 3). In this example the constraint touch itself does the satisfaction. If it finds a solution, it raises the event `solved.touch`. Then the coordinator creates streams from the constraint to the operand objects. The coordinator only creates the communication network, all the atomic processes are responsible for actually doing something.

7 Object and constraint models

We want a change of a variable to lead to the checking of the validity of constraints on the variable. A possible approach is to have a central data base with values of the object member variables, and a data manager. Satellites processes could then subscribe to events such as changing variables. When an event occurs, the data manager notifies

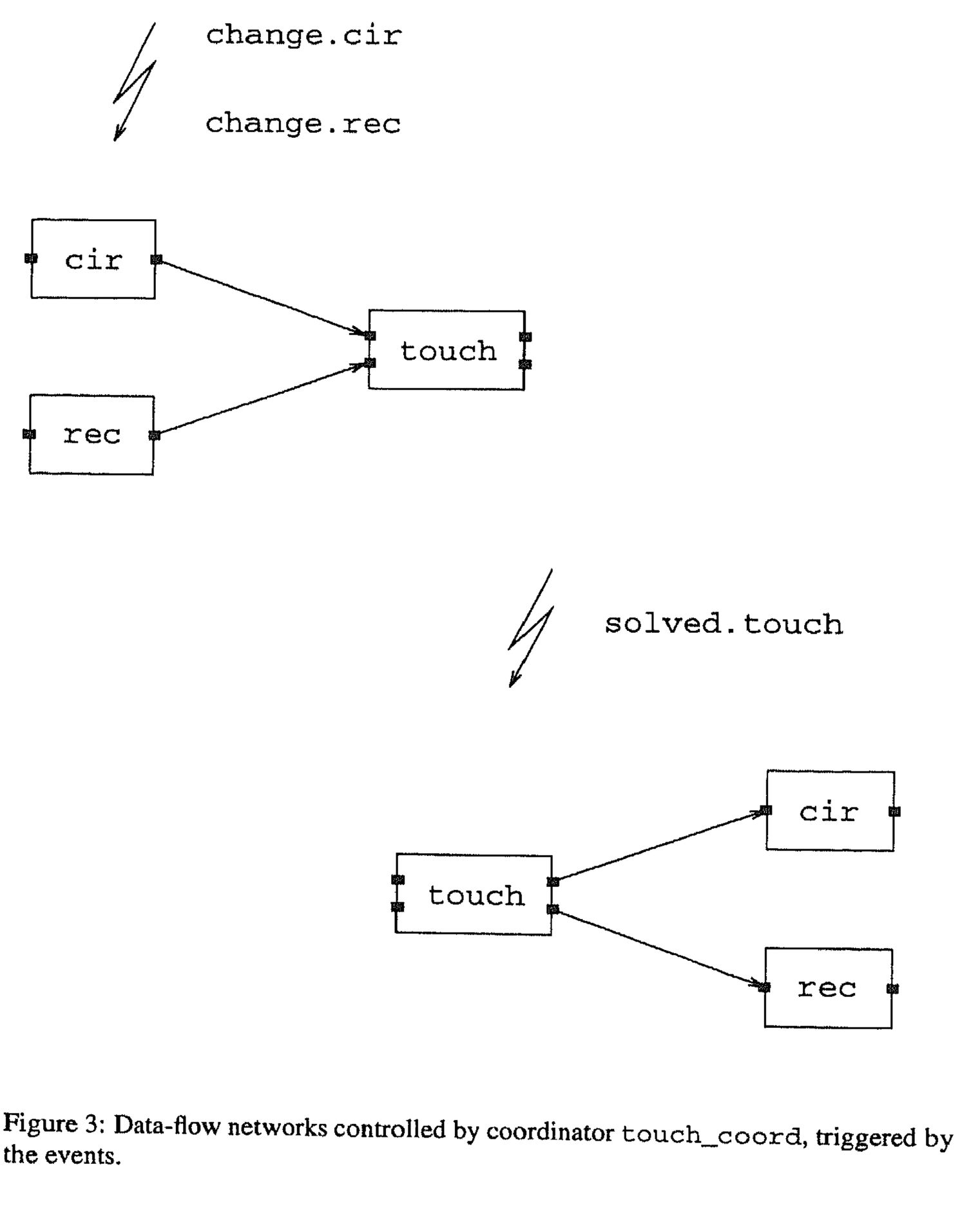


Figure 3: Data-flow networks controlled by coordinator `touch_coord`, triggered by the events.

all satellites that subscribed to that event. The concept of such a central data manager is hard to combine with object-oriented concepts such as data encapsulation.

In Manifold, all the objects conceptually are active objects. This means that every object has its own virtual processor with its own thread of control (as mentioned in section 2). When the value of an object's variable is changed, we let it raise the event `change`.

We are currently exploring two alternative approaches to modelling constraints. In the first approach, the constraints are solved and maintained in Manifold. In this way the application objects and the constraints are completely orthogonal. The communication between the objects and the constraint side is via data streams set up under Manifold control.

In the second approach, constraints are modeled as objects, just like the application objects. In this scheme, each constraint object `cstr` has an associated shadow coordinator `cstr_coord` (like `touch_coord` in the example above). The coordinator can listen to an event `change` for each of the constraint operands. The constraint coordinator can then decide to perform global or local satisfaction.

If the constraints are ordinary objects, the application programmer could create new constraint classes and new operand classes. The system should automatically generate the event raising behaviour of the objects, input and output ports for communication with Manifold, and the shadow coordinators for constraint objects. The programmer has to provide methods to write data into the output port and to read from the input port that are consistent with those at the other side of the stream, i.e. the stream between a constraint and an operand. This defines an interface between the two. In this way, the state of an object can be completely read and set, but the exact implementation of the object remains hidden. Note however that this state can only be read and set from the Manifold side, not by the other application objects.

8 Implications

We are currently exploring the implications of these two alternatives in terms of functionality, style, and ease of use. One of the implications of the separation of objects and constraints management is that several satisfaction techniques can easily be used in one system. Indeed it may be profitable to use a class of algorithms that can be used to eliminate local (node, arc, and path) inconsistencies [13] before any attempt is made to construct a complete solution. Another possibility is the combination of propagation of degrees of freedom and propagation of known states (also just called local propagation). Propagating degrees of freedom amounts to discarding all parts of the constraint network that can be satisfied easily and solving the rest by some other method. Propagation of degrees of freedom identifies a part in the network with enough degrees of freedom so that it can be changed to satisfy all its constraints. That part and all the constraints that apply to it are then removed from the network. Deletion of these constraints may give another part enough degrees of freedom so as to satisfy all its constraints. This continues until no more degrees of freedom can be propagated.

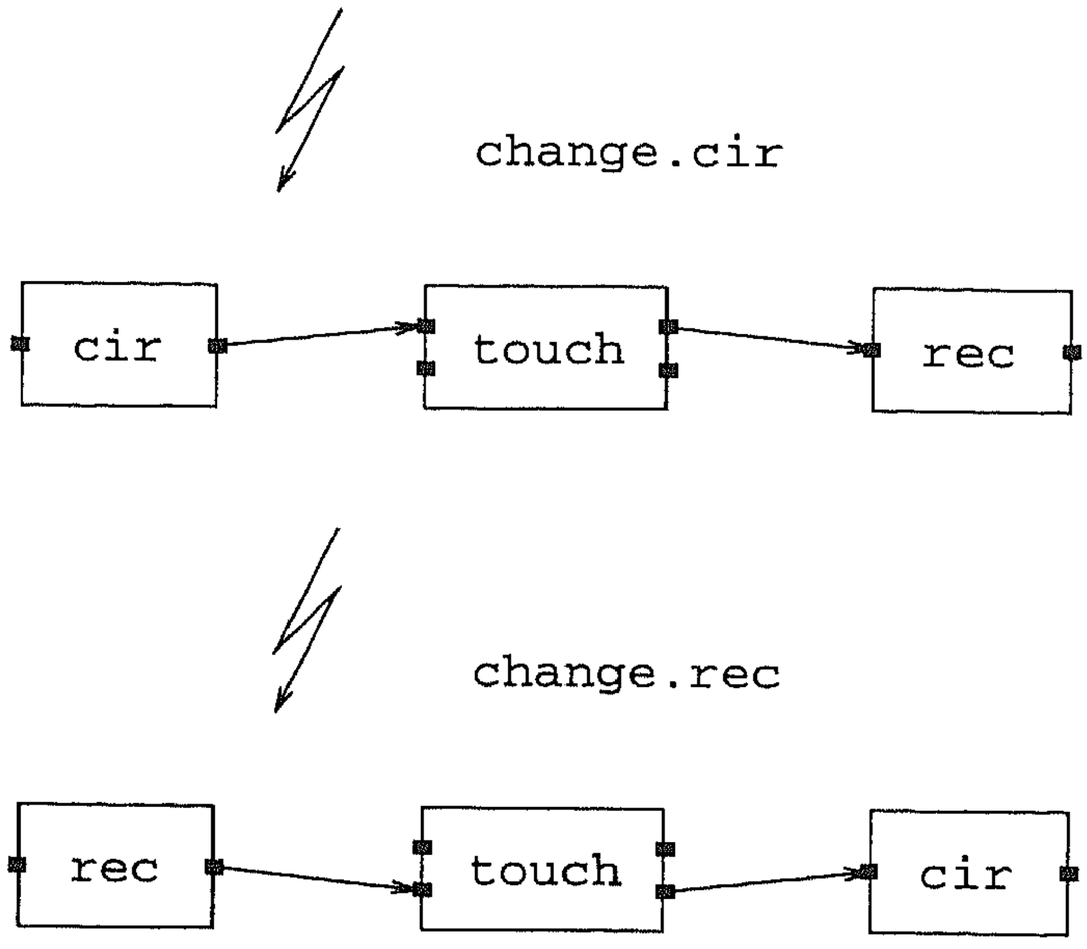


Figure 4: Local propagation networks controlled by `local_touch_coord`.

The part of the network that is left is then satisfied by some global method. The result can now be propagated towards the discarded parts, which are successively satisfied (propagation of known states).

Local propagation is easily coordinated. In our example this could be done in the following way (see figure 4):

```
local_touch_coord(cir,rec,touch)
process cir, rec, touch.
{ event wait.

  start:                              do wait.

  change.cir:                  (cir->touch.in1, touch.out2->rec).
  change.rec:                  (rec->touch.in2, touch.out1->cir).
  satisfied.touch:     do wait.

  wait:                                (cir,rec).
}
```

A change of one of the constraint operands results in the raising of an event. This causes `local_touch_coord` to create streams from the altered object to the constraint, and from the constraint to the other object. The constraint is responsible for finding a new solution for the other object.

Some situations allow an even simpler coordination. For example after a local distortion of the constraint, e.g. by a method 'translate' of `cir`. For such methods that make constraints fail, corresponding events (e.g. `translate.cir`) may trigger local propagation similar to the approach in [12]:

```
translate.cir:   (cir.out -> rec.in).
```

The above examples are by no means complete, but give a flavour of the type of solution we propose. In our approach we retain strict encapsulation for all modelling of concrete objects. Relationships between objects which cannot logically be ascribed to the internal actions of a container object are expressed in terms of constraints. These constraints may be global but the referential transparency of functional relationships allows one to reason about them and prove their correctness. The proofs of correctness will of course only apply provided the objects, which are regarded as atomic objects from the point of view of constraints, act according to specifications. All modelling of objects with states and behaviour is done in the normal object-oriented framework. In this framework correctness depends (as it always did) on correct program design, using concepts such as modularity and hierarchical decomposition.

One of our next research goals is to model the satisfaction of meta-constraints and higher-order constraints. The strict separation between coordination and functionality of constraint satisfaction provides a way to handle constraints on the satisfaction mechanism (meta-constraints), and constraints on constraints (higher-order constraints).

9 Conclusions

This paper proposes a relevant and important contribution to systems support for interactive computer graphics. This contribution, the combination of constraints and object-oriented methods, has been much heralded but has yet to arrive. We believe that we have identified a major cause for this lack of progress.

The problem is to combine two important approaches to software engineering: object-oriented and declarative programming, in casu constraint programming. The two naturally come together in computer graphics when the behaviour of active objects is partly modelled through constraints. Several approaches to integrate constraints and objects have been taken, see section 5. We have proposed a solution that keeps the object-oriented and constraint programming paradigms distinct and does not compromise the benefits which they severally confer.

The results of our research will lead to better design, analysis, and implementation of interactive graphics systems. The abstraction developed will have immediate application in graphical simulation and visualization as well as graphical user interface

management systems. More generally a way of expressing relations between objects, within a robust software engineering based approach, is urgently needed in multimedia applications and other complex interactive graphical applications.

This paper describes a research project in progress. We are currently elaborating and implementing the alternative object and constraint models with the event-based mechanism. Our next research goal is to model the satisfaction of meta-constraints and second-order constraints. This enhances the power of constraint resolution, which alleviates a second reason for the lack of impact of constraints in the object-oriented approach: the lack of powerful and general satisfaction techniques.

The example of the touch-constraint on the rectangle and the circle has been implemented in C++ and Manifold. A complete demo program is available for ftp in the directory ftp.cwi.nl:/pub/remco/EventBasedConstraints.

10 Acknowledgement

We like to thank Richard Kelleners for programming the Manifold demo. This research has been supported in part by NWO (Dutch Organization for Scientific Research) under Grants NF-51/62-514, SION-612-322-212, and SION-612-31-001.

References

[1] F. Arbab. Specification of Manifold. Technical Report CS–9220, CWI, Amsterdam, The Netherlands, 1992.

[2] F. Arbab, I. Herman, and P. Spilling. An overview of Manifold and its implementation. *Concurrency: Practice and Experience*, 5(1):23 – 70, February 1993.

[3] E. H. Blake and S. Cook. On including part hierarchies in object-oriented languages, with an implementation in smalltalk. In *Proc. ECOOP'87*, volume 276 of *Lect. Notes Comp. Sci.*, pages 41–50. Springer-Verlag, Berlin, 1987.

[4] Edwin H. Blake and Quinton Hoole. Expressing relationships between objects: Problems and solutions. In *Proceedings of the Third Eurographics Workshop on Object-Oriented Graphics,* Champery, Switzerland, 1992.

[5] Alan Borning. The programming language aspects of ThingLab, a constraint-oriented simulation laboratory. *ACM Transactions on Programming Languages and Systems*, 3(4):353 – 387, October 1981.

[6] Bull – Imaging and Office Solutions. *GoPATH 1.2.0 — A Path To Object Oriented Graphics, a public domain environment for graphical and interactive application development*, 1993.

[7] Eric Cournarie and Michel Beaudouin-Lafon. Alien: a prototype-based constraint system. In Laffra et al. [11].

[8] Jacques Davy. Go, a graphical and interactive C++ toolkit for application data presentation and editing. In *Proceedings 5th Annual Technical Conference on the X Window System*, 1991.

[9] Bjorn N. Freeman-Benson. Kaleidoscope: Mixing objects, constraints, and imperative programming. *(ECOOP/OOPSLA '90 Proceedings) SIGPLAN Notices*, 25(10):77 – 88, October 1990.

[10] Bjorn N. Freeman-Benson and Alan Borning. Integrating constraints with an object-oriented language. In O. Lehrmann Madsen, editor, *Proceedings ECOOP'92 – European Conference on Object-Oriented Programming, Utrecht, 1992*, Lecture Notes in Computer Science 615, pages 268 – 286. Springer-Verlag, 1992.

[11] Chris Laffra, Edwin Blake, Vicky de Mey, and Xavier Pintado, editors. *Advances in Object-Oriented Graphics II & III*. Springer-Verlag, 1995.

[12] Chris Laffra and Jan van den Bos. Propagators and concurrent constraints. *OOPS Messenger*, 2(2):68 – 72, April 1991.

[13] Alan K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8:99 – 118, 1977.

[14] John R. Rankin. A graphics object oriented constraint solver. In Laffra et al. [11].

[15] Ivan E. Sutherland. Sketchpad: A man-machine graphical communication system. In *Proceedings of the Spring Joint Computer Conference, Detroit, Michigan, May 21-23 1963*, pages 329 – 345. AFIPS Press, 1963.

[16] Remco C. Veltkamp. A quantum approach to geometric constraint satisfaction. In Laffra et al. [11].

[17] Michael Wilk. Equate: an object-oriented constraint solver. In *Proceedings OOPSLA'91*, pages 286 – 298, 1991.

Arrays, Bounded Quantification and Iteration in Logic and Constraint Logic Programming

Krzysztof R. Apt

CWI, P.O. Box 94079, 1090 GB Amsterdam, The Netherlands,

Dept. of Mathematics, Computer Science, Physics & Astronomy, University of Amsterdam, Plantage Muidergracht 24, 1018 TV Amsterdam, The Netherlands

Abstract

We claim that programming within the logic programming paradigm suffers from lack of attention given to iteration and arrays. To convince the reader about their merits we present several examples of logic and constraint logic programs which use iteration and arrays instead of explicit recursion and lists. These programs are substantially simpler than their counterparts written in the conventional way. They are easier to write and to understand, are guaranteed to terminate and their declarative character makes it simpler to argue about their correctness. Iteration is implemented by means of bounded quantification.

1 Introduction

Any systematic course on programming in the imperative style (say using Pascal), first concentrates on iteration constructs (say **while** or **repeat**) and only later deals with recursion. Further, the data structures are explained first by dealing with the static data structures (like arrays and records) and only later with the dynamic data structures (which are constructed by means of pointers).

In the logic programming framework the distinctions between iteration and recursion, and between static and dynamic data structures are lost. One shows that recursion is powerful enough to simulate iteration and rediscovers the latter by identifying it with tail recursion. Arrays do not exist. In contrast, records can be modelled by terms, and dynamic data structures can be defined by means of clauses, in a recursive fashion (with the exception of lists for which in Prolog there is support in the form of built-in's and a more friendly notation).

One of the side effects of this approach to programming is that one often uses a sledgehammer to cut the top of an egg. Even worse, simple problems have unnecessarily complex and clumsy solutions in which recursion is used when a much easier solution using iteration exists, is simpler to write and understand, and — perhaps even more important — is closer to the original specification.

In this paper we would like to propose an alternative approach to programming in logic and in constraint logic programming — an approach in which adequate stress is put on the use of arrays and iteration. Because iteration can be expressed by means of bounded quantification, a purely logical construct, the logic programming paradigm is not "violated". On the contrary, it is enriched, clarified and better tailored to the programming needs.

Arrays are especially natural when dealing with vectors and matrices. The use of dynamic data structures to write programs dealing with such objects is unnatural. We shall try to illustrate this point by presenting particularly simple solutions to problems such as the 8 queens problem, the knight's tour, and the map colouring problem.

Further, by adding to the language operators which allow us to express optimization, i.e. minimization and maximization, we can easily write programs for various optimization problems, like the cutting stock problem.

For pedagogical reasons we limit here our attention to programs that involve arrays, iteration and optimization constructs. Of course, recursive data types and explicit recursion have their place both in logic programming and in constraint logic programming. One of the main purposes of this paper is to illustrate how much can be achieved without them.

In the programs considered in this paper recursion is hidden in the implementation of the bounded quantifiers and this use of recursion is guaranteed to terminate. Consequently, these programs always terminate. As termination is one of the major concerns in the case of logic programming, from the correctness point of view it is better to use iteration instead of recursion, when a choice arises. Also, iteration can be implemented more efficiently than recursion (see Barklund and Bevemyr [BB93] for an explanation how to extend WAM to implement iteration in Prolog).

This work can be seen as an attempt to identify the right linguistic concepts which simplify programming in the logic programming paradigm. When presenting this view of programming within the logic programming paradigm we were very much influenced by the publications of Barklund and Millroth [BM94], Voronkov [Vor92] and Kluźniak [Klu93]. In fact, the constructs whose use we advocate, i.e. bounded quantification and arrays, were already proposed in those papers. Apart from providing further evidence for elegance of these constructs in logic programming, the only, possibly new, contribution

of this paper is a proposal to integrate these constructs into constraint logic programming.

2 Bounded Quantifiers

Bounded quantifiers in logic programming were introduced in Kluźniak [Klu91] and are thoroughly discussed in Voronkov [Vor92] (where also earlier references in Russian are given). They are also used in Kluźniak [Klu93] (see also Kluźniak and Miłkowska [KM94]) in a specification language SPILL-2 in which executable specifications can be written in the logic programming style.

Following Voronkov [Vor92] we write them as `∃X ∈ L Q` (the bounded existential quantifier) and `∀X ∈ L Q` (the bounded universal quantifier), where `L` is a list and `Q` a query, and define them as follows:

```
∃X ∈ [Y | Ys] Q ← Q{X/Y}.
∃X ∈ [Y | Ys] Q ← ∃X ∈ Ys Q.

∀X ∈ [Y | Ys] Q ← Q{X/Y}, ∀X ∈ Ys Q.
∀X ∈ [] Q.
```

To put these definitions into syntactically acceptable format, we could introduce two relations, `exists` and `forall`, and write `exists(X, L, Q)` for `∃X ∈ L Q` and `forall(X, L, Q)` for `∀X ∈ L Q`. For clarity, we shall use the original syntax.

The bounded quantifies can be easily expressed using the usual quantifiers, so the above language extension is subsumed by the proposal of Lloyd and Topor [LT84] (see also Lloyd [Llo87]) in which the queries and bodies of clauses can be arbitrary first-order formulas. Unfortunately, this modelling of bounded quantifiers yields unnecessarily complex programs, among others due to the use of negation and the introduction of new relation symbols.

Moreover, as pointed out by Barklund and Hill [BH95], this translation process introduces the possibility of incorrect use of negation which in some circumstances limits the use of the program to ground queries. This difficulty was originally pointed out by Bundy [Bun88] in the context of another form of bounded universal quantification.

Voronkov [Vor92] also discusses two other bounded quantifiers, written as `∃X ⊏ L Q` and `∀X ⊏ L Q`, where `X ⊏ L` is to be read "`X` is a suffix of `L`", which we do not consider here.

To some extent the use of bounded quantifiers allows us to introduce in some compact form the "and" and the "or" branching within the program computations. This reveals some connections with the approach of Harel [Har80], though we believe that the expressiveness and ease of programming within the logic programming paradigm makes Harel's programming proposal obsolete.

Even without the use of arrays the gain in expressiveness achieved by means of bounded quantifiers is quite spectacular. Consider for example the following problem which shows the power of the $\forall$`X` $\in$ `L` $\exists$`Y` $\in$ `M` combination.

Problem 1 Write a program which tests whether one list contains all the elements of another one.

Solution

```
subset(Xs, Ys) ← ∀X ∈ Xs ∃Y ∈ Ys X = Y.
```

Several other examples can be found in Voronkov [Vor92]. Here we content ourselves with just one more, in which we use delay declarations very much like in modern versions of Prolog (for example ECL^{i}PS^e [Agg95]) or the programming language Gödel of Hill and Lloyd [HL94]).

Problem 2 Write a program checking the satisfiability of a Boolean formula.

Solution We assume here that the input Boolean formula is written using Prolog notation, so for example `(¬ X, Y) ; Z` stands for `(¬ X ∧ Y) ∨ Z`.

```
sat(X) ← X, generate(X).
generate(X) ← vars(X, Ls), ∀Y ∈ Ls ∃Z ∈ [true, fail] Y = Z.
DELAY X UNTIL nonvar(X).
```

This remarkably short program uses meta-variables and a mild extension of the delay declarations to meta-variables. The delay declaration used here delays any call to a meta-variable until it becomes instantiated; `vars(t, Ls)` for a term `t` computes in `Ls` the list of the variables occurring in `t`. Its definition is omitted. `vars(X, Ls)` can be easily implemented using the `var(X)` and `univ` built-in's of Prolog. `true` and `fail` are Prolog's built-in's.

In Gödel the calls to negative literals are automatically delayed until they become ground. In the case of the above program such an automatic delay is not advisable as it would reduce checking for satisfiability of subformulas which begin with the negation sign to a naive generate and test method.

Even though this program shows the power of Prolog, we prefer to take another course and use types instead of exploring extensions of Prolog, which is an untyped language.

3 Arrays and Bounded Quantifiers in Logic Programming

Arrays in logic programming were introduced in Eriksson and Rayner [ER84]. Barklund and Bevemyr [BB93] proposed to extend Prolog with arrays and studied their use in conjunction with the bounded quantification. In our opinion the resulting extension (unavoidably) suffers from the fact that Prolog is an untyped language. In Kluźniak [Klu93] arrays are present, as well, where they are called indexable sequences.

More recently, Barklund and Hill [BH95] proposed to add arrays and restricted quantification, a generalization of the bounded quantification, to Gödel, the programming language which does use types. Also Greco, Palopoli and Spadafora [GPS95] suggested to extend Datalog, a simple logic programming based database language, with arrays.

These developments should be contrasted with the early proposal of Kowalski [Kow83] to encode arrays by means of unit clauses.

In the programs below we use bounded quantification, arrays and type declarations. The use of bounded quantifiers and arrays makes them simpler, more readable and closer to specifications. We declare constants, types, variables and relations in a style borrowed from the programming language Pascal.

We begin with two introductory examples which involve search through a sequence. The first one uses the universal quantifier while the second one employs the existential quantifier.

Problem 3 Check whether a given sequence of 100 integers is ordered.

Solution

```
const n = 100.
rel ordered: array [1..n] of integer.
ordered(A) ← ∀I ∈ [1..n-1] A[I] ≤ A[I+1].
```

This example shows that the terms denoting the array subscripts should be evaluated (so that we can identify `1+1` with 2 etc.), very much like the right-hand side of the `is` built-in of Prolog. In a more general set up we could view here "+" as an external procedure in the sense of Małuszyński et al. [MBB+93]. This simple program appears originally in Kowalski [Kow83] though its procedural interpretation is not explained there.

Note that the bounded universal quantifier `∀I ∈ [1..n]` does *not* correspond to the imperative **for** `i:=1` **to** `n` loop. The former is executed as long as a failure does not arise, i.e. up to n times, whereas the latter is executed precisely n

times. The programming construct ∀I ∈ [1..n] Q actually corresponds to the construct

```
for i:=1 to n do if ¬ Q then
                    begin
                      failure := true; exit
                    end
```

which is clumsy and unnatural within the imperative programming paradigm.

(Feliks Kluźniak suggested to us the following, slightly more natural interpretation of ∀I ∈ [1..n] Q:

```
i:=1;
while i ≤ n cand Q do i:=i+1;
failure := i ≤ n,
```

where **cand** is the "conditional **and**" connective (see Gries [Gri81, pages 68-70].))

Problem 4 *Linear Search.* Check if an element is present in a given sequence of 100 integers. If yes, return its position, otherwise terminate with a failure.

```
const n = 100.
type seq: array [1..n] of integer.
rel find: (integer, seq, [1..n]).
find(E, A, J) ← ∃I ∈ [1..n] (E = A[I], J = I).
```

Here "=" is Prolog's built-in, defined by the single clause

```
X = X.
```

and called "is unifiable with". Now the query `find(e, a, J)` checks the presence of an element `e` in an array `a`. If the answer is positive, `J` is instantiated to the position of `e` in `a`. Otherwise failure results. During the execution of this query "=" is used first to compare two ground terms and then to assign a value to a variable, `J`.

It is instructive to note that the development of the corresponding solution to the linear search problem in the imperative programming style, together with the formal correctness proof, takes Sethi [Set89] three pages.

Note that in contrast to the imperative programming case, the above solution can also be used to generate all elements of `a` with their corresponding positions, by means of the query `find(E, a, J)`. In this case both uses of "=" result in assigning a value to a variable, first to `E` and then to `J`.

The bounded existential quantifier $\exists$I $\in$ [1..n] implements backtracking and has no counterpart within the imperative programming paradigm. Here the backtracking is very "shallow" and boils down to the execution of the test E = A[I] for specific values of E and A[I].

Of course, it would be preferable to use the above solution to the linear search problem for arrays of any type, not only of the type **integer**. This motivates introduction of polymorphic types in presence of arrays. Then the appropriate generalization of the above solution to an arbitrary type is obtained by using the following generalized declarations, where "*" stands for a variable denoting an unknown type:

```
type seq: array [1..n] of *.
rel find: (*, seq, [1..n]).
rel =: (*, *).
```

Of course, in general, more than one unknown type can be used in a program.

The next example shows the power of the $\forall$X $\in$ L $\exists$Y $\in$ M combination in presence of arrays and a non-trivial instance of the backtracking process.

Problem 5 Arrange three 1's, three 2's, ..., three 9's in sequence so that for all $i \in [1,9]$ there are exactly i numbers between successive occurrences of i (see Coelho and Cotta [CC88, page 193]).

Solution

```
rel sequence: array [1..27] of [1..9].
sequence(A) ← ∀I ∈ [1..9] ∃J ∈ [1..25-2I]
    (A[J] = I, A[J+I+1] = I, A[J+2I+2] = I)).
```

The range J $\in$ [1..25-2I] comes from the requirement that the indices J, J+I+1, J+2I+2 should lie within [1..27]. Thus J+2I+2 $\leq$ 27, that is J $\leq$ 25-2I.

It is useful to note here that the corresponding solution to this problem in Prolog is 15 lines long.

Next, we show the usefulness of local definitions.

Problem 6 Generate all permutations of a given sequence of 100 elements.

First we provide a solution for the case when there are no repeated elements in the sequence.

Solution 1

```
const n = 100.
rel permutation: (array [1..n] of *, array [1..n] of *).
permutation(X, Y) ← ∀I ∈ [1..n] ∃J ∈ [1..n] Y[J] = X[I].
```

Here, X is the given sequence.

Note the similarity in the structure between this program and the one that solves problem 1. This program is incorrect when the sequence contains repeated elements. For example for n = 3 and X:= [0,0,1], the array Y:= [0,1,1] is a possible answer.

To deal with the general case we use local array declarations and refine the above program.

Solution 2

```
const n = 100.
rel permutation: (array [1..n] of *, array [1..n] of *).
permutation(X, Y) ←
    var A: array [1..n] of [1..n].
    ∀I ∈ [1..n] ∃J ∈ [1..n] A[J] = I,
    ∀I ∈ [1..n] Y[I] = X[A[I]].
```

This solution states that A is an onto function from [1..n] to [1..n] and that a permution of a sequence of n elements is obtained by applying the function A to its indices.

Next, consider two well-known chess puzzles.

Problem 7 Place 8 queens on the chess board so that they do not check each other.

First, we provide a naive generate and test solution. It will be of use in the next section.

Solution 1

```
const n = 8.
type board: array [1..n] of [1..n].
rel queens, generate, safe: board.
queens(X) ← generate(X), safe(X).
generate(X) ← ∀I ∈ [1..n] ∃J ∈ [1..n] X[I] = J.
safe(X) ← ∀I ∈ [1..n] ∀J ∈ [I+1..n]
    (X[I] ≠ X[J], X[I] ≠ X[J] + (J-I), X[I] ≠ X[J] + (I-J)).
```

To improve readability `board` is explicitly declared here as a type. Declaratively, this program states the conditions which should be satisfied by the values chosen for the queens. "≠" is a built-in declared as

```
rel ≠: (*, *).
```

and defined by the single clause

```
X ≠ Y ← ¬ (X = Y).
```

In this section we use it only to compare ground terms. A more general usage of "≠" will be explained in the next section.

Next, we give a solution which involves backtracking.

Solution 2

```
const n = 8.
type board: array [1..n] of [1..n].
rel queens: board.

queens(X) ← ∀J ∈ [1..n] ∃K ∈ [1..n]
    (X[J] = K,
     ∀I ∈ [1..J-1]
        (X[I] ≠ X[J], X[I] ≠ X[J] + (J-I), X[I] ≠ X[J] + (I-J))).
```

Declaratively, this program states the conditions each possible value `K` for a queen placed in column `J` should satisfy. In its last line `X[J]` could be replaced by `K`.

Problem 8 *Knight's tour.* Find a cyclic route of a knight on the chess board so that each field is visited exactly once.

Solution We assign to each field a value between 1 and 64 and formalize the statement "from every field there is a "knight-reachable" field with the value one bigger". By symmetry we can assume that the value assigned to the field `X[1, 1]` is 1. Taking into account that the route is to be cyclic we actually get the following solution.

```
const n = 8.
type board: array [1..n, 1..n] of [1..n²].
rel knight: board.
    go_on: (board, [1..n], [1..n]).

knight(X) ← ∀I ∈ [1..n] ∀J ∈ [1..n] go_on(X, I, J), X[1, 1] = 1.

go_on(X, I, J) ← ∃I1 ∈ [1..n] ∃J1 ∈ [1..n]
    (abs((I-I1)·(J-J1)) = 2, X[I1, J1] = (X[I, J] mod n²) + 1).
```

```
DELAY go_on(X, I, J) UNTIL ground(X[I,J]).
```

Note that the equation abs(X · Y) = 2 used in the definition of go_on has exactly 8 solutions, which determine the possible directions for a knight move. Observe that each time this call to "=" is selected, both arguments of it are ground. The efficiency of go_on could be of course improved by explicitly enumerating the choices for the offsets of the new coordinates w.r.t. the old ones.

The behaviour of the above program is quite subtle. First, thanks to the delay declaration, 64 constraints of the form go_on(X, I, J) are generated. Then, thanks to the statement X[1, 1] = 1, the first of them is "triggered" which one by one activates the remaining constraints. The backtracking is carried out by choosing different values for the variables I1 and J1. The delay declaration is not needed, but without it this program would be hopelessly inefficient.

It is interesting to note that in Wirth [Wir76], a classical book on programming in Pascal, the solutions to the last two problems are given as prototypical examples of recursive programs. These solutions are based on the same principle, namely backtracking. Here recursion is implicit in the implementation of bounded quantifiers.

We conclude this section by one more program which shows the use of another type of quantifier.

Problem 9 Let m = 50 and n = 100. Determine the number of different elements in an array X:**array** [1..m, 1..n] **of integer**.

Solution

```
const m = 50.
      n = 100.
type board: array [1..m,1..n] of integer.
rel count: (board, natural).

count(X, Number) ←
    Number = m · n -
     #(I, J: I ∈ [1..m], J ∈ [1..n]:
         (∃K ∈ [1..I-1] ∃L ∈ [1..n] X[I,J] = X[K,L])
           % X[I,J] occurs in an earlier row
          ∨ (∃L ∈ [1..J-1] X[I,J] = X[I,L]).
           % X[I,J] occurs earlier in the same row
    ).
```

In this program we used the counting quantifier introduced in Gries [Gri81, page 74] and adopted in Kluźniak [Klu93] in the specification language SPILL-

2. In general, given lists `L1`, `L2`, the term `#(I, J: I ∈ L1, J ∈ L2: Q)` stands for the number of pairs `(i,j)` such that `i ∈ L1`, `j ∈ L2` and the query `Q{I/i,J/j}` succeeds. It is possible to avoid the use of the counting quantifier at the expense of introducing a local array of type `board`. This alternative program is more laborious to write.

This concludes our presentation of selected logic programs written using arrays, bounded quantifiers and some other features. Other examples, involving among others numerical computation, can be found in Barklund and Millroth [BM94].

4 Arrays and Bounded Quantifiers in Constraint Logic Programming

In this section we illustrate the use of arrays and bounded quantifiers in constraint logic programs. We assume from the reader some familiarity with the basic principles of constraint logic programming (see e.g. the survey article of Jaffar and Maher [JM94]).

The programs presented here are constraint programs with finite domains in the style of van Hentenryck [vH89], where we refer the reader for a number of unexplained notions. Each of these programs has a similar pattern: first constraints are generated, and then resolved after the possible values for variables are successively generated. We explain here briefly how individual constraints are processed, but do not discuss the strategies for constraint solving and constraint propagation. This calls for a generalization of the constraint solvers proposed in the literature to a more general situation in which subscripted variables are used.

We begin by providing here alternative solutions to two problems discussed in the previous section.

Problem 10 Solve problem 7 by means of constraints.

Solution

```
const n = 8.
type board: array [1..n] of [1..n].
rel queens, safe, generate: board.

queens(X) ← safe(X), generate(X).

safe(X) ← ∀I ∈ [1..n] ∀J ∈ [I+1..n]
    (X[I] ≠ X[J], X[I] ≠ X[J] + (J-I), X[I] ≠ X[J] + (I-J)).
```

```
generate(X) ← ∀I ∈ [1..n] ∃J ∈ dom(X[I]) X[I] = J.
```

Here `dom(X)`, for a (possibly subscripted) variable `X`, is a built-in which denotes the list of current values in the domain of `X`, say in ascending order. The value of `dom(X)` can change only by decreasing. This can happen only by executing a constraint, so in the above program an atom of the form `X` $\neq$ `t`.

The relation "$\neq$" was used in the previous section only in the case when both arguments of it were ground, so known. Here we generalizes its usage, as we now allow that one or both sides of it are not known. In fact, "$\neq$" is a built-in defined as in van Hentenryck [vH89, pages 83-84], though generalized to arbitrary non-compound types.

We require that one of the following holds:

- Both sides of "$\neq$" are known. This case is explained in the previous section.
- At most one of the sides of "$\neq$" is known and one of the sides of "$\neq$", denoted below by `X`, is either a simple variable or a subscripted variable with a known subscript.

In the second case `X` $\neq$ `t` is defined as follows, where for a term `s`, `Val(s)` stands for the set of its currently possible values:

```
if Val(X) ∩ Val(t) = ∅ then succeed
elseif Val(t) is a singleton then
% t is known, so X is not known, i.e. dom(X) has at least 2 elements
    begin dom(X):= dom(X) - Val(t);
    % remove the value of t from dom(X)
      if dom(X) = [f] then X:= f
    end .
```

If neither `Val(X)` $\cap$ `Val(t)` $= \emptyset$ nor `Val(t)` is a singleton, then the execution of `X` $\neq$ `t` is delayed. We treat `t` $\neq$ `X` as `X` $\neq$ `t`.

So for example in the program fragment

```
...
type bool: [false, true].
rel p: (bool,bool,bool).
p(A,B,C) ← A ≠ B, B ≠ C, C = true, ...
```

during the call of `p(A,B,C)` the constraints `A` $\neq$ `B` and `B` $\neq$ `C` are first delayed and then upon the execution of the atom `C = true` the variable `B` becomes `false` and subsequently `A` becomes `true`.

In turn, in the case of the solution to the 8 queens problem given above, during

the call of `safe(X)` the execution of an atom of the form `X[I] = K` for some `I,K ∈ [1..n]` can affect the domains of the variables `X[J]` for `J ∈ [I+1..n]` via the execution of a constraint of the form `X[I] ≠ X[J] + t`.

This solution to the 8 queens problem is a forward checking program (see van Hentenryck [vH89, pages 122-127]). Note the textual similarity between this program and the one given in solution 1 to problem 7. Essentially, the calls to the `safe` and `generate` relations are now reversed. The `generate` relation corresponds to the `labeling` procedure in van Hentenryck [vH89]. In the subsequent programs the definition of the `generate` relation has always the same format and is omitted.

Problem 11 Solve problem 6 by means of constraints.

Solution

```
const n = 100.
rel permutation: (array [1..n] of *, array [1..n] of *).
permutation(X, Y) ←
    type board: array [1..n] of [1..n].
    rel one_one, generate: board.

    one_one(Z) ← ∀I ∈ [1..n] ∀J ∈ [I+1..n] Z[I] ≠ Z[J].

    var A: board.
    one_one(A), generate(A),
    ∀I ∈ [1..n] Y[I] = X[A[I]].
```

In this solution, apart of the local declaration of the variable A, we also use local type and relation declarations. The efficiency w.r.t. to the logic programming solution is increased by stating, by means of the call to the `one_one` relation, that `A` is a 1-1 function. This replaces the previously used statement that `A` is an onto function. The call to `one_one` generates $n \cdot (n-1)/2 = 4950$ constraints.

We conclude this section by dealing with another classical problem — that of a map colouring. It shows the use of implication.

Problem 12 Given is a binary relation `neighbour` between countries. Colour a map in such a way that no two neighbours have the same color.

Solution

```
type color: [blue, green, red, yellow].
     countries: [austria, belgium, france, italy, ...].
rel map_color, constrain, generate: array countries of color.
    neighbour: (countries, countries).
```

```
map_color(X) ← constrain(X), generate(X).

constrain(X) ← ∀I ∈ countries ∀J ∈ countries
    neighbour(I,J) → X[I] ≠ X[J].
```

The declarative interpretion of P → Q is as follows:

```
(P → Q) ← P, Q.
(P → Q) ← ¬P.
```

So P → Q corresponds to the IF P THEN Q statement of Gödel. Obviously, an efficient implementation of P → Q should avoid the reevaluation of P. Note that in the above program, at the moment of selection of the P → Q statement, P is ground.

Thus the constrain relation generates here the constraints of the form X[I] ≠ X[J] for all I,J such that neighbour(I,J).

5 Adding Minimization and Maximization

Next, we introduce constructs allowing us to express in a compact way the requirement that we are looking for an optimal solution. To this end we introduce the *minimization operator* X = μQ which declaratively is defined as follows:

```
X = μQ ← Q, ¬(∃Y ( Y < X, Q{X/Y})).
```

We assume here that X and Y are of the type **integer**. The existential quantifier ∃X Q is defined by the clause

```
∃X Q ← Q.
```

The efficient implementation of the minimization operator should employ some specialized methods, like the branch and bound technique, in order to limit the search process during the successive attempts of finding a minimal solution to the query Q.

A dual operator, the *maximization operator* X = νQ, is defined declaratively by:

```
X = νQ ← Q, ¬(∃Y ( Y > X, Q{X/Y})).
```

To put these definitions into syntactically acceptable format, we could write min(X,Q) for X = μQ and max(X,Q) for X = νQ.

In Barklund and Hill [BH95] the minimization and the maximization operators are introduced as a form of arithmetic quantifiers, in the style of the counting quantifier introduced earlier.

We now show the use of the minimization operator.

Problem 13 *The cutting stock problem* (see van Hentenryck [vH89, pages 181-187]). There are 72 configurations, 6 kinds of shelves and 4 identical boards to be cut. Given are 3 arrays:

```
Shelves:array [1..72, 1..6] of natural,
Req:array [1..6] of natural,
Waste:array [1..72] of natural.
```

`Shelves[K,J]` denotes the number of shelves of kind `J` cut in configuration `K`, `Waste[I]` denotes the waste per board in configuration `I` and `Req[J]` the required number of shelves of kind J. The problem is to cut the required number of shelves of each kind in such a way that the total waste is minimized.

Solution We represent the chosen configurations by the array

```
Conf: array [1..4] of [1..72]
```

where `Conf[I]` denotes the configuration used to cut the board `I`.

```
rel solve: (array [1..4] of [1..72], natural).
   generate: array [1..4] of [1..72].

solve(Conf, Sol) ←
    Sol = μ
        % Sol is the minimal TCost such that:
    ∀I ∈ [1..3] Conf[I] ≤ Conf[I+1],
        % symmetry between the boards
    ∀J ∈ [1..6] Σ_{I=1}^{4} Shelves[Conf[I],J] ≥ Req[J],
        % enough shelves are cut
    Sol = Σ_{I=1}^{4} Waste[Conf[I]],
        % Sol is the total waste
    generate(Conf).
```

The constraints `Conf[I]` $\leq$ `Conf[I+1]`, for `I` $\in$ `[1..3]`, limit the number of generated solutions and (like in van Hentenryck [vH89]) are added here only for the efficiency purposes.

In this program we used as a shorthand the sum notation "$\Sigma \ldots$". In general, it is advisable to use the sum quantifier (see Gries [Gri81, page 72]), which allows us to use $\Sigma_{I=k}^{l}$ `t` as a term. The sum quantifier is adopted in SPILL-2 language of Kluźniak [Klu93]. Kluźniak's notation for this expression is:

(S I: k $\leq$ I $\leq$ l: t). The interpretation of the constraints of the form X $\leq$ t, X $\geq$ t or X = t is similar to that of X $\neq$ t and is omitted.

6 Conclusions

We have presented here several logic and constraint logic programs that use bounded quantification and arrays. We hope that these examples convinced the readers about the usefulness of these constructs. We think that this approach to programming is especially attractive when dealing with various optimization problems, as their specifications often involve arrays, bounded quantification, summation, and minimization and maximization. Constraint programming solutions to these problems can be easily written using arrays, bounded quantifiers, the sum and cardinality quantifiers, and the minimization and maximization operators. As examples let us mention the stable marriage problem, the knapsack problem and various scheduling problems.

Of course, it is not obvious whether the solutions so obtained are efficient. We expect, however, that after an addition of a small number of built-in's, like **deleteff** and **deleteffc** of van Hentenryck [vH89, pages 89-90] and specialized versions of the bounded quantifiers that allow us to alter the search order through the range, it will be possible to write simple constraint programs which will be comparable in efficiency with those written in other languages for constraint logic programming.

When introducing arrays we were quite conservative and only allowed static arrays, i.e. arrays whose bounds are determined at compile time. Of course, in a more realistic language proposal also open arrays, i.e. arrays whose bounds are determined at run-time, should be allowed. One might also envisage the use of flexible arrays, i.e. arrays whose bounds can change at run-time.

In order to make this programming proposal more realistic one should provide a smooth integration of arrays with recursive types, like lists and trees. In the language SPILL-2 of Kluźniak [Klu93] types are present but only as sets of ground terms, and polymorphism is not allowed. Barklund and Hill [BH95] proposed to add arrays to Gödel (which does support polymorphism) as a system module. We would prefer to treat arrays on equal footing with other types.

We noticed already that within the logic programming paradigm the demarkation line between iteration and recursion differs from the one in the imperative programming paradigm. In order to better understand the proposed programming style one should first clarify when to use iteration instead of recursion. In this respect it is useful to quote the opening sentence of Barklund and

Millroth [BM94]: "Programs operating on inductively defined data structures, such as lists, are naturally defined by recursive programs, while programs operating on "indexable" data structures, such as arrays, are naturally defined by iterative programs".

We do not entirely agree with this remark. For example, the "suffix" quantifiers mentioned in Section 2 allow us to write many list processing programs without explicit use of recursion (see Voronkov [Vor92]) and the `quicksort` program written in the logic programming style is more natural when written using recursion than iteration.

The single assignment property of logic programming makes certain programs that involve arrays (like Warshall's algorithm) obviously less space efficient than their imperative programming counterparts. This naturally motivates research on efficient implementation techniques of arrays within the logic programming paradigm.

Finally, a comment about the presentation. We were quite informal when explaining the meaning of the proposed language constructs. Note that the usual definition of SLD-resolution has to be appropriately modified in presence of arrays and bounded quantification. For example, the query `X[1] = 0, I = 1, X[I] = 1` fails but this fact can be deduced only when the formation of resolvents is formally explained. To this end substitution for subscripted variables needs to be properly defined. One possibility is to adopt one of the definitions used in the context of verification of imperative programs (see Apt [Apt81, pages 460-462]). We leave the task of defining a formal semantics of the constructs proposed here to another paper.

Acknowledgements I would like to thank here Jonas Barklund and Feliks Kluźniak for useful discussions on the subject of bounded quantification and Pascal van Hentenryck for encouragement at the initial stage of this work. Also, I am grateful to Feliks Kluźniak, Robert Kowalski and Andrea Schaerf for helpful comments on this paper.

References

[Agg95] A. Aggoun et al. *ECL^{i}PS^e 3.5 User Manual.* Munich, Germany, February 1995.

[Apt81] K.R. Apt. Ten years of Hoare's logic, a survey, part I. *ACM TOPLAS*, 3:431–483, 1981.

[BB93] J. Barklund and J. Bevemyr. Prolog with arrays and bounded quantifications. In Andrei Voronkov, editor, *Logic Programming and*

Automated Reasoning—Proc. 4th Intl. Conf., LNCS 698, pages 28–39, Berlin, 1993. Springer-Verlag.

[BH95] J. Barklund and P. Hill. Extending Gödel for expressing restricted quantifications and arrays. UPMAIL Tech. Rep. 102, Computer Science Department, Uppsala University, Uppsala, 1995.

[BM94] J. Barklund and H. Millroth. Providing iteration and concurrency in logic programs through bounded quantifications. UPMAIL Tech. Rep. 71, Computer Science Department, Uppsala University, Uppsala, 1994.

[Bun88] A. Bundy. A Broader Interpretation of Logic in Logic Programming. In *Proceedings of the 5th International Conference and Symposium on Logic Programming, Seattle*, pages 1624–1648, 1988.

[CC88] H. Coelho and J. C. Cotta. *Prolog by Example.* Springer-Verlag, Berlin, 1988.

[ER84] L.-H. Eriksson and M. Rayner. Incorporating mutable arrays into logic programming. In S. Å. Tarnlund, editor, *Proc. Second Int'l Conf. on Logic Programming*, pages 101–114. Uppsala University, 1984.

[Gri81] D. Gries. *The Science of Programming.* Springer-Verlag, New York, 1981.

[GPS95] S. Greco, L. Palopoli, and E. Spadafora. Extending datalog with arrays. *Data Knowledge and Engineering*, 17:31–57, 1995.

[Har80] D. Harel. And/or programs: a new approach to structured programming. *ACM Toplas*, 2(1):1–17, 1980.

[HL94] P. M. Hill and J. W. Lloyd. *The Gödel Programming Language.* The MIT Press, 1994.

[JM94] J. Jaffar and M.J. Maher. Constraint logic programming: A survey. *Journal of Logic Programming*, 19,20:503–581, 1994.

[Klu91] F. Kluźniak. Towards practical executable specifications in logic. Research report LiTH-IDA-R-91-26, Department of Computer Science, Linköping University, August 1991.

[Klu93] F. Kluźniak. SPILL-2: the language. Technical report ZMI Reports No 93-03, Institute of Informatics, Warsaw University, July 1993. A deliverable for year 1 of the BRA Esprit Project Compulog 2.

[KM94] F. Kluźniak and M. Miłkowska. Readable, runnable requirements specifications: Bridging the credibility gap. In M. Hermenegildo and J. Penjam, editors, *Programming Language Implementation and Logic Programming. Proceedings of the 6th International Symposium, PLILP'94. Madrid, September 1994*, pages 449–450. Springer-Verlag, 1994.

[Kow83] R.A. Kowalski. Logic programming. In *Proceedings IFIP'83*, pages 133–145. North-Holland, 1983.

[Llo87] J. W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, Berlin, second edition, 1987.

[LT84] J. W. Lloyd and R. W. Topor. Making PROLOG more expressive. *Journal of Logic Programming*, 1:225–240, 1984.

[MBB+93] J. Małuszyński, S. Bonnier, J. Boye, F. Kluźniak, A. Kågedal, and U. Nilsson. Logic programs with external procedures. In K.R. Apt, J.W. de Bakker, and J.J.M.M. Rutten, editors, *Current Trends in Logic Programming Languages*, pages 21–48. The MIT Press, Cambridge, Massachussets, 1993.

[Set89] R. Sethi. *Programming Languages: Concepts and Constructs.* Addison-Wesley, 1989.

[vH89] P. van Hentenryck. *Constraint Satisfaction in Logic Programming.* Logic Programming Series, The MIT Press, Cambridge, MA, 1989.

[Vor92] A. Voronkov. Logic programming with bounded quantifiers. In A. Voronkov, editor, *Logic Programming and Automated Reasoning—Proc. 2nd Russian Conference on Logic Programming*, LNCS 592, pages 486–514, Berlin, 1992. Springer-Verlag.

[Wir76] N. Wirth. *Algorithms + Data Structures = Programs.* Prentice-Hall, 1976.

Presuppositions and Information Updating

Jan van Eijck

CWI, Amsterdam and OTS, Utrecht

January 22, 1997

1 Introduction

Presupposition failures are errors occurring during the left-right processing of a computer program or natural language text. A general method for analysing such errors with dynamic logic is presented, based on the idea that sequential processing changes context dynamically and that this process of context change can be made the object of analysis in dynamic modal logic.

Updating a state of information with a statement bearing a presupposition can be viewed as a combination of two things: (i) checking whether the presupposition holds in the current context, and, provided this is the case, (ii) updating the current state of information with the informational content of the statement. In case the presupposition is not fulfilled in the current context, one might adjust the context to make it hold, and next do the further update with the informational content of the statement. These two actions are obviously *ordered*. First the context is adjusted to accommodate the presupposition, next the information state is updated with the assertion. Context and information state are used interchangeably here, and indeed, we can view the context which gets updated as a state of information.

We propose to view the study of presupposition failure from the general perspective of updating information structures ([2], [23], [14]). In this perspective, providing information is a dynamic process involving a speaker and an audience, which gets the audience from an initial information state to a new more informed state, or in case the new information is inconsistent with the current state, to the absurd information state.

It turns out that in many cases the information update relation is functional. In such cases presupposition failure can be catered for by switching to a partial update function. In case the presupposition of some update is not met in some information state, the update function yields 'undefined' for that update in that state. But information updating is not always functional. Communication mismatches do occur if the new information is too vague to yield a unique

update in the current information state. In such cases the audience will have to indicate that the update cannot be processed without further ado. Note that this is different from the communication mismatch which occurs if the new information hinges upon a wrong assumption about the current context (the cases of presupposition failure).

The point that presupposition has to do with information updating has been made again and again in the literature. Stalnaker and Karttunen come to mind as early proponents of the view that presupposition projection should be accounted for in dynamic terms. See [24, 25], [18, 19], [20], and [12]. Modern versions of this approach appear in [1], [5], [21], and [27].

Stalnaker proposes the following explanation of the rule (first stated in [18]) that the presupposition of a conjunction $A\&B$ consists of the presupposition of A conjoined with the implication $\text{ass}_A \rightarrow \text{pres}_B$.

> The explanation goes like this: ... when a speaker says something of the form *A and B*, he may take it for granted that *A* ... after he has said it. The proposition that *A* will be added to the background of common assumptions before the speaker asserts that *B*. Now suppose that *B* expresses a proposition that would, for some reason, be inappropriate to assert except in a context where A, or something entailed by *A*, is presupposed. Even if *A* is *not* presupposed initially, one may still assert *A and B* since by the time one gets to saying that *B*, the context has shifted, and it is by then presupposed that *A*.
>
> From [25], p. 211, also quoted in [13].

Modern versions of the dynamic approach to presupposition all attempt to formalize the *pragmatic* notion of presupposition, i.e., the notion of presupposition where a presupposition is a presupposition of the speaker about the context when he/she utters a sentence in that context, rather than a property of the sentence itself. On the other hand, inappropriateness of a sentence in a given context can be construed as lack of a definite truth value of that sentence in that context, and the presupposition of a sentence can always be viewed as a statement which holds in precisely those contexts where the sentence is appropriate.

We should distinguish, therefore between talking about what holds in given states of information on one hand and about updating information states on the other. It is precisely here that dynamic logic, which distinguishes between a level of static description and a level of procedural description, and which provides means of relating these two description levels, becomes an illuminating tool. The difference between the present analysis and other recent dynamic approaches to presupposition is the focus on the *link* between statics and dynamics, which relates semantic presupposition to pragmatic presupposition.

2 Information Structures

An information structure I is a pair $\langle S, \sqsubseteq \rangle$ with S a non-empty set of information states and $\sqsubseteq$ a pre-order (transitive and reflexive, but not necessarily antisymmetric) over S which is called the information order.

If L is a language for S, then an L-information model M is a triple $\langle S, \sqsubseteq, \sigma \rangle$, where $\sigma : L \to \mathcal{P}S$ is a specification function which interprets the language L in S. Classical languages can be specified by a single function σ, but for languages of partial logic one needs pairs σ^+, σ^- of such functions, and so on.

We consider the simplest case first, the case of propositional logic, where the states of information are sets of propositional valuations. Let a set of proposition letters P be given. Then the set of valuations is the set $\{0,1\}^P$. Call this set W. Members of W may be considered as '(epistemically) possible worlds'. An information state is a subset of W plus a member of W (the distinguished member plays the role of the 'actual world' or the 'current perspective of the knowing subject'); the set S of all information states is $\{\langle i, w \rangle \mid i \subseteq W, w \in W\}$. The information ordering $\sqsubseteq$ on S is given by:

$$\langle i, w \rangle \sqsubseteq \langle j, w' \rangle \text{ iff } i \supseteq j \text{ and } w = w'.$$

Note that this gives a partial order, not just a pre-order. Note also that we do not demand $w \in i$ if $\langle i, w \rangle$ is an information state. In particular, $\langle \emptyset, w \rangle$ is an information state, namely the absurd information state, viewed from perspective w. Without 'perspective worlds' it would be more awkward to modally characterize inconsistent information. With the use of a perspective world, we can simply say that $\Box\bot$ characterizes an inconsistent belief state.

Information states can be viewed as K45 models: sets of possible worlds with an 'almost' universal accessibility relation, i.e., transitive and almost reflexive and almost symmetric, for the 'perspective world' need not be accessible from itself. To be precise, K45 logic is complete for transitive and euclidean frames (a relation R is euclidean if xRy and xRz together imply that yRz),

K45 models are appropriate to talk about the kind of belief where one has complete information about one's own uncertainty (for more information about this, see [22]). An appropriate 'local' language L to talk about information states is the language of propositional modal logic:

$$\alpha \quad ::= \quad \bot \mid p \mid \neg\alpha \mid (\alpha_1 \wedge \alpha_2) \mid \Diamond\alpha.$$

We employ the usual abbreviations for $\top, \wedge, \rightarrow, \leftrightarrow, \Box$. The specification function σ for the language is given by:

$$\begin{array}{lcl}
\sigma(\bot) & = & \emptyset, \\
\sigma(p) & = & \{\langle i, w \rangle \in S \mid w(p) = 1\}, \\
\sigma(\neg\alpha) & = & S - \sigma(\alpha), \\
\sigma(\alpha_1 \wedge \alpha_2) & = & \sigma(\alpha_1) \cap \sigma(\alpha_2) \\
\sigma(\Diamond\alpha) & = & \{\langle i, w \rangle \in S \mid \exists w' \in i \text{ with } \langle i, w' \rangle \in \sigma(\alpha)\}.
\end{array}$$

We say that $s \models \alpha$ iff $s \in \sigma(\alpha)$. Note that $\langle i, w\rangle \models \Box\bot$ iff $i = \emptyset$. An information state $\langle\emptyset, w\rangle$ is absurd or inconsistent: if we are in such a state, nothing at all is compatible with what we know or believe. Any boxed formula is true in an absurd information state; indeed, $\langle i, w\rangle \neq \langle\emptyset, w\rangle$ iff there is a formula $\alpha \in L$ with $\langle i, w\rangle \not\models \Box\alpha$.

An information state $\langle W, w\rangle$ is an information state of complete ignorance, a state of having no information at all. If P is infinite, there is no formula of L which characterizes W; for finite P there is. If $P = \{p_0, \ldots, p_n\}$, let C be the (finite) set of all conjunctions of form $\Diamond((\neg)p_0 \wedge \cdots \wedge (\neg)p_n)$. Then:

$$\langle i, w\rangle \models \bigwedge C \text{ iff } i = W.$$

3 Updating Propositional Information

Updating a state of information with a new piece of information can be viewed as moving up in the information order, toward some more informed state. Propositional updates are just tests (performed in the current perspective world). Epistemic updates can shift the information state: updating the information of one's audience with $\Box F$ will get the audience in a state where F is known, i.e., a state where $\Box F$ holds. Similarly, downdating with $\Box F$ will get the audience in a state where F is not known anymore, i.e, in a state where $\Diamond\neg F$ holds.

In the more general perspective on information structures, neither updates or downdates need to be minimal (cf. [2] and [23]), but for present purposes this restriction is useful. Minimal updates are given by:

$$\begin{aligned}[\![\alpha^u]\!] \;=\; & \{\langle s, s'\rangle \in S \times S \mid s \sqsubseteq s', s' \in \sigma(\alpha),\\ & \forall s'' \in S : (s \sqsubseteq s'' \sqsubseteq s' \;\&\; s'' \in \sigma(\alpha)) \Rightarrow s' \sqsubseteq s''\}\end{aligned}$$

We will use F, G, F_1, ... as metavariables for purely propositional formulas of L. As it turns out, the minimal update relation for L is functional for updates of the form $\Box F$ or $\Diamond F$.

We may assume that knowing subjects, unless they are all-knowing, cannot distinguish the actual world from any other of their epistemic alternatives, so updates with purely propositional F are a bit silly: they can succeed only if they do not change states, and their success depends on what is true in the actual world.

Example 1 *An update with $p \vee q$ in state $\langle i, w\rangle$ checks if $i, w \models p \vee q$, and succeeds if this is the case, fails otherwise. In other words, this update succeeds iff $w(p) = 1$ or $w(q) = 1$.*

If I utter $p \vee q$, this should be understood as: 'I believe or know that $p \vee q$, and I want you to accept that information about the world, too. So, this update should be understood as having an implicit $\Box$ in front.

Example 2 *An update with* $\Box(p \vee q)$ *in state* $\langle i, w\rangle$ *causes a shift to a state* $\langle j, w\rangle$ *where* $j = \{w' \in i \mid \langle i, w'\rangle \models p \vee q\}$.

Example 3 *An update with* $\Diamond(p \vee q)$ *in state* $\langle i, w\rangle$ *does not change state in case* $\langle i, w\rangle \models \Diamond(p \vee q)$, *and otherwise fails.*

Updates with $\Diamond F$ formulas are 'consistency tests': they check whether the information F is consistent with the current information state. Let $\|\alpha\|_i$ be $\{w \in W \mid \langle i, w\rangle \in \sigma(\alpha)\}$. The following proposition holds:

Proposition 4

1. $[\![F^u]\!] = \{\langle s, s\rangle \in S \times S \mid s \models F\}$.
2. $[\![\Box F^u]\!] = \{\langle\langle i, w\rangle, \langle i \cap \|F\|_i, w\rangle\rangle \mid \langle i, w\rangle \in S\}$.
3. $[\![\Diamond F^u]\!] = \{\langle s, s\rangle \in S \times S \mid s \models \Diamond F\}$.

Example 5 *An update with* $\Box p \vee \Box q$ *in state* $\langle i, w\rangle$ *is not functional in case* i *contains a* w_1 *with* $w_1(p) = 1, w_1(q) = 0$ *and a* w_2 *with* $w_2(p) = 0, w_2(q) = 1$. *In this case there are two possible outcome states:* $s_1 = \langle\{w' \in i \mid w'(p) = 1\}, w\rangle$ *and* $s_2 = \langle\{w' \in i \mid w'(q) = 1\}, w\rangle$.

Take for example the case where $s = \langle\{p\bar{q}, \bar{p}q\}, w\rangle$ *(where* $\bar{p}$ *indicates that* p *is false). Both*

$$\langle\{p\bar{q}, \bar{p}q\}, w\rangle \mapsto \langle\{p\bar{q}\}, w\rangle$$

and

$$\langle\{p\bar{q}, \bar{p}q\}, w\rangle \mapsto \langle\{\bar{p}q\}, w\rangle$$

are minimal updates.

Modal updates are discussed in [26], but with the constraint that only modal updates of the forms $\Diamond\top \to \Diamond F$ and $\Box F$ are allowed. Our $\Diamond\top \to \Diamond F$ corresponds to Veltman's *might* F, and our $\Box F$ to his F, so the $\Box$ is left implicit in his notation. Updates of the form $\Diamond\top \to \Diamond F$ are total functions, while updates of the form $\Diamond F$ may be partial. In fact, an update with $\Diamond\top \to \Diamond F$ will effect a transition to an inconsistent state if $\Diamond F$ does not hold in the current state. Any update of the form $\Box F \wedge \bigvee\bigwedge \Diamond F_i$ (where F and all the F_i are purely propositional), is functional. An important result about K45, by the way, is that every formula has an equivalent formula of the form $\bigvee(F \wedge \Box G \wedge \bigwedge \Diamond H_i)$ (where F, G and the H_i are all purely propositional). This follows from the completeness of K45 with respect to finite 'balloon' frames, i.e., frames where the accessibility relation is transitive and euclidean; these frames have the shape of a balloon of mutually accessible worlds all accessible from a single perspective world. See e.g. [3] for more information. Disjunction over $\Box$ is the feature that 'threatens' functionality.

The functional K45 formulas are 'honest' formulas in the sense of [10]. A formula is honest if one can honestly claim that one *only* knows that formula. This is equivalent to saying that a minimal update of the state of complete ignorance with that formula is functional. Claiming that you only know $\Box p \vee \Box q$ is a cheat, because in order for that to be a true statement you either have to be in a state where all the accessible worlds are p worlds, and in that case you also know p, or you have to be in a state where all the accessible worlds are q worlds, and in that case you also know q.

A K45 formula α is persistent if $\langle i, w\rangle \models \alpha$ and $\langle i, w\rangle \sqsubseteq \langle j, w\rangle$ together imply that $\langle j, w\rangle \models \alpha$. Formulas of the form $\Box F$ are persistent, formulas of the form $\Diamond F$, with F a consistent propositional formula, are not. Conjunctions and disjunctions of persistent formulas are persistent. Every persistent formula is K45 equivalent to a disjunction of conjunctions of formulas of the forms F and $\Box F$ (F purely propositional).

Example 6 *$\Diamond p$ is true at $\langle W, w\rangle$, but false at any information state $\langle i, w\rangle$ without p worlds.*

The persistent and functional K45 formulas are precisely the formulas of the form $F_1 \wedge \Box F_2$, F_1 and F_2 purely propositional (up to K45 equivalence).

Conversely, we can look at minimal downdates to move back to a state where α does not hold anymore. Minimal downdates are given by:

$$\begin{aligned}[\![\alpha^d]\!] \;=\; & \{\langle s, s'\rangle \in S \times S \mid s' \sqsubseteq s, s' \notin \sigma(\alpha),\\ & \forall s'' \in S : (s' \sqsubseteq s'' \sqsubseteq s \;\&\; s'' \notin \sigma(\alpha)) \Rightarrow s'' \sqsubseteq s'\}.\end{aligned}$$

The minimal downdate relation is not functional for formulas of the form $\Box F$, and it is a test for formulas of the form $\Diamond F$.

Example 7 *$[\![(\Box p)^d]\!]$ will relate a state $\langle i, w\rangle$ satisfying $\langle i, w\rangle \models \Box p$ to any state $\langle j, w\rangle$ where j is of the form $i \cup \{w'\}$, with $w'(p) = 0$, and a state $\langle i, w\rangle$ not satisfying $\langle i, w, \rangle \models \Box p$ to itself.*

Downdates with formulas of the form $\Diamond F$ can only succeed in case $\Diamond F$ is inconsistent with the current information state.

Example 8 $[\![(\Diamond p)^d]\!] = \{\langle s, s\rangle \in S \times S \mid s \models \neg\Diamond p\}$.

In general we have:

Proposition 9

1. $[\![F^d]\!] = \{\langle s, s\rangle \in S \times S \mid s \models \neg F\}$.
2. $[\![\Box F^d]\!] = \{\langle s, s\rangle \in S \times S \mid s \models \neg\Box F\} \cup \{\langle\langle i, w\rangle, \langle i \cup \{w'\}, w\rangle\rangle \mid \langle i, w\rangle \in S, \langle i, w\rangle \models \Box F, w' \in W - \|F\|_i\}$.
3. $[\![\Diamond F^d]\!] = \{\langle s, s\rangle \in S \times S \mid s \models \neg\Diamond F\}$.

Over the local language L we now layer a global language L_1, which is the language of the information structure S. L has a procedural and a propositional level; the procedures are (minimal) updating, (minimal) downdating, testing, plus sequential compositions of those, the propositions of L_1 are built from the formulas of L (which act as atoms of L_1) using boolean combination and procedure projections.

$$\begin{array}{lcl} \alpha & ::= & \bot \mid p \mid \neg\alpha \mid (\alpha_1 \wedge \alpha_2) \mid \Diamond\alpha \\ \pi & ::= & \alpha^u \mid \alpha^d \mid \varphi? \mid (\pi_1;\pi_2) \\ \varphi & ::= & \alpha \mid \neg\varphi \mid (\varphi_1 \wedge \varphi_2) \mid \Diamond\varphi \mid \mathrm{dom}\,(\pi) \mid \mathrm{ran}\,(\pi) \mid \mathrm{fix}\,(\pi). \end{array}$$

The interpretation of L_1 consists of two parts: a relational interpretation for the procedures and a truth definition for the formulas. The interpretation for the procedures and formulas uses mutual recursion.

$$\begin{array}{lcl} [\![\alpha^u]\!] & = & \text{as given above.} \\ [\![\alpha^d]\!] & = & \text{as given above.} \\ [\![\varphi?]\!] & = & \{\langle s,s\rangle \in S \times S \mid S,s \models \varphi\}. \\ [\![\pi_1;\pi_2]\!] & = & [\![\pi_1]\!] \circ [\![\pi_2]\!]. \end{array}$$

In the final clause, $\circ$ denotes relational composition.

$$\begin{array}{lll} S,s \models \alpha & \text{iff} & s \models \alpha \\ S,s \models \neg\varphi & \text{iff} & S,s \not\models \varphi \\ S,s \models (\varphi_1 \wedge \varphi_2) & \text{iff} & S,s \models \varphi_1 \text{ and } S,s \models \varphi_2 \\ S,s \models \Diamond\varphi & \text{iff} & s = \langle i,w\rangle \text{ and} \\ & & \text{there is some } w' \in i \text{ with } S,\langle i,w'\rangle \models \varphi \\ S,s \models \mathrm{dom}\,(\pi) & \text{iff} & \exists s' \in S : s[\![\pi]\!]s' \\ S,s \models \mathrm{ran}\,(\pi) & \text{iff} & \exists s' \in S : s'[\![\pi]\!]s \\ S,s \models \mathrm{fix}\,(\pi) & \text{iff} & s[\![\pi]\!]s. \end{array}$$

This system is an extension of the system of update logic presented in [26] with tests, downdates and a more liberal regime concerning modal updates. Alternatively, it can be viewed as a fragment of a structured version of the Dynamic Modal Logic (DML) in [2], with structured states (in this case: K45 models) instead of unstructured propositional valuations, but with just a subset of the procedural repertoire.

We can define the perhaps more familiar dynamic logic style procedure modalities $\langle\pi\rangle$ and $[\pi]$ in terms of the projection operators and tests as follows:

$$\langle\pi\rangle\varphi \stackrel{\mathbf{def}}{=} \mathrm{dom}\,(\pi;\varphi?),$$

$$[\pi]\varphi \stackrel{\mathbf{def}}{=} \neg\mathrm{dom}\,(\pi;(\neg\varphi)?).$$

Note that it follows from these definitions that:

- $S, s \models \langle\pi\rangle\varphi$ iff $\exists s' \in S : s[\![\pi]\!]s'$ and $S, s' \models \varphi$.
- $S, s \models [\pi]\varphi$ iff $\forall s' \in S : s[\![\pi]\!]s'$ implies $S, s' \models \varphi$.

Note the following important differences:

$S, s \models \bot$		never.
$S, s \models \top$		always.
$S, s \models \Box\bot$	iff	$s = \langle\emptyset, w\rangle$.
$S, s \models \Diamond\top$	iff	$s \neq \langle\emptyset, w\rangle$.

Example 10 *We see from the above that* $[\pi]\bot$ *expresses that no* π *transition is possible, while* $[\pi]\Box\bot$ *expresses that the only possible* π *transition will get one to the absurd information state, or, in other words, that the transition* π *yields inconsistency.*

Example 11 $\langle\pi\rangle\top$ *expresses that a* π *transition is possible;* $\langle\pi\rangle\Diamond\top$ *expresses that a* consistent π *transition is possible.*

The notion of validity for L_1 is as follows:

- $\models \varphi$ iff for all $s \in S : S, s \models \varphi$.

Because we have used the full space $\mathcal{P}W$ to define the state set S, there is no need to mention S as a parameter in the validity notion: once the set of proposition letters is fixed, the set of information states is fixed. This changes when we allow S to be a proper subset of

$$\{\langle i, w\rangle \mid i \in \mathcal{P}W, w \in W\},$$

subject to certain conditions. Nothing in the notion of 'information structure' prevents us from doing this, as long as we make sure that the information ordering $\sqsubseteq$ on S remains a pre-order. We will not explore this possibility here, however.

4 Some Example Validities

We will not present a full axiomatisation of the logic of propositional up- and downdating, but merely give some of the valid principles that we can use to reason about information transitions. Next to the obvious axioms and rules of inference of propositional logic, of normal modal logic for $\Diamond$, we need the following:

P 4.1 $\Box\varphi \to \Box\Box\varphi$.

P 4.2 $\Diamond\varphi \to \Box\Diamond\varphi$.

Principles 4.1 and 4.2 are the principles of positive and negative introspection of $K45$ for $\Diamond$.

P 4.3 $\mathrm{dom}(\pi) \leftrightarrow \mathrm{dom}(\pi; \top?)$.

P 4.4 $\mathrm{ran}(\pi) \leftrightarrow \mathrm{ran}(\top?; \pi)$.

These are to make sure that we can always assume dom arguments to be of the general form $\pi; \varphi?$, and ran arguments to be of the general form $\varphi?; \pi$.

P 4.5 $[\pi](\varphi_1 \rightarrow \varphi_2) \rightarrow ([\pi]\varphi_1 \rightarrow [\pi]\varphi_2)$.

This is the K schema for π, which expresses that for every information transition procedure π, the operator $[\pi]$ is a normal modal operator.

P 4.6 $\langle \alpha^u \rangle \top \leftrightarrow \langle \alpha^u \rangle \alpha$.

This expresses that after updating with α, α will hold. The principle does not entail that updates have the property of *right seriality* (for every s there is a t with $s[\![\pi]\!]t$), for we have seen that this need not be the case for α of the forms F or $\Diamond F$.

P 4.7 $[F^u]\varphi \leftrightarrow (F \rightarrow \varphi)$.

P 4.8 $[\Diamond F^u]\varphi \leftrightarrow (\Diamond F \rightarrow \varphi)$.

These express that F and $\Diamond F$ updates are tests.

P 4.9 $\langle \Box F^u \rangle \Box G \leftrightarrow [\Box F^u] \Box G \leftrightarrow \Box(F \rightarrow G)$.

The soundness of Principle 4.9 follows from the functionality of $\Box F$ updates, plus the next proposition.

Proposition 12 *$s \models [\Box F^u]\Box G$ iff $s \models \Box(F \rightarrow G)$.*

Proof: $\langle i, w \rangle \models [\Box F^u]\Box G$
iff $\langle i \cap \|F\|_i, w \rangle \models \Box G$
iff $\langle i \cap \{w' \mid w' \models F\}, w \rangle \models \Box G$
iff $\forall w' \in i$: if $w' \models F$ then $w' \models G$
iff $\langle i, w \rangle \models \Box(F \rightarrow G)$. ∎

P 4.10 $\langle \Box F^u \rangle \Diamond G \leftrightarrow [\Box F^u] \Diamond G \leftrightarrow \Diamond(F \wedge G)$.

The soundness of Principle 4.10 follows from the functionality of $\Box F$ updates, plus the next proposition.

Proposition 13 *$s \models [\Box F^u]\Diamond G$ iff $s \models \Diamond(F \wedge G)$.*

Proof: $\langle i, w\rangle \models [\Box F^u]\Diamond G$
iff $\langle i \cap \|F\|_i, w\rangle \models \Diamond G$
iff $\langle i \cap \{w' \mid w' \models F\}, w\rangle \models \Diamond G$
iff $\exists w' \in i$: $w' \models F$ and $w' \models G$
iff $\langle i, w\rangle \models \Diamond(F \wedge G)$. ∎

P 4.11 $[F^d]\varphi \leftrightarrow (F \vee \varphi)$.

P 4.12 $[\Diamond F^d]\varphi \leftrightarrow (\Diamond F \vee \varphi)$.

These express that F and $\Diamond F$ downdates are tests.

The following principle is a rule rather than an axiom schema.

P 4.13 $\frac{(\bigwedge C \rightarrow \Diamond(F \wedge \neg G))}{(\text{ran}\,(\Diamond F?; \Box G^u) \leftrightarrow \Box G)}$.

Here $\bigwedge C$ is the conjunction of all formulas of the form $\Diamond((\neg)p_0 \wedge \cdots \wedge (\neg)p_n)$, where $p_0, \ldots, p_n$ are the proposition letters occurring in F, G.

P 4.14 $\text{ran}\,(\varphi?; F^u) \leftrightarrow (\varphi \wedge F)$.

P 4.15 $\text{ran}\,(\varphi?; \Diamond F^u) \leftrightarrow (\varphi \wedge \Diamond F)$.

P 4.16 $\text{ran}\,(\Box F?; \Box G^u) \leftrightarrow \Box(F \wedge G)$.

After updating a context with a persistent update, the persistent preconditions will hold in the new context.

P 4.17 $\text{ran}\,(\varphi?; F^d) \leftrightarrow (\varphi \wedge \neg F)$.

P 4.18 $\text{ran}\,(\varphi?; \Diamond F^d) \leftrightarrow (\varphi \wedge \neg\Diamond F)$.

P 4.19 $\text{ran}\,((\Diamond F_1 \wedge \cdots \wedge \Diamond F_n)?; \Box G^d) \leftrightarrow (\Diamond F_1 \wedge \cdots \wedge \Diamond F_n \wedge \Diamond\neg G)$.

The counterparts to the previous three for downdates.

P 4.20 $\langle \alpha^d\rangle \top \leftrightarrow \langle \alpha^d\rangle \neg\alpha$.

This expresses that a downdate can only succeed if after downdating with α, α does not hold anymore. Note that downdating will be impossible if the downdate is a logical validity (the present set-up is unsuitable for modelling 'unlearning' of logical truths).

P 4.21 $\text{fix}\,(\alpha^u) \leftrightarrow \alpha$.

This expresses that updating with α doesn't change the context precisely when α already holds.

P 4.22 $\text{fix}\,(\alpha^d) \leftrightarrow \neg\alpha$.

This expresses that downdates with information that is already known to be false have no effect.

P 4.23 $\text{fix}\,(\pi;\varphi?) \leftrightarrow \text{fix}\,(\varphi?;\pi) \leftrightarrow (\text{fix}\,(\pi) \wedge \varphi)$.

This is an obvious statement about fixpoints.

P 4.24 $(\text{fix}\,(\pi) \wedge \varphi) \rightarrow (\text{dom}\,(\pi;\varphi?) \wedge \text{ran}\,(\varphi?;\pi))$.

This relates fixpoint to domain and range.

P 4.25 $(\text{fix}\,(\pi_1) \wedge \text{fix}\,(\pi_2)) \rightarrow \text{fix}\,(\pi_1;\pi_2)$.

If s is a fixpoint for π_1 and π_2, then s is a fixpoint for $\pi_1;\pi_2$ (but note that this cannot be strengthened to an equivalence).

P 4.26 $\text{dom}\,(\pi_1;\pi_2;\varphi?) \leftrightarrow \text{dom}\,(\pi_1;\text{dom}\,(\pi_2;\varphi?)?)$.

This is the usual principle for sequential composition. Stated in terms of $\langle\pi\rangle$ it can also be expressed as $\langle\pi_1;\pi_2\rangle\varphi \leftrightarrow \langle\pi_1\rangle\langle\pi_2\rangle\varphi$ (familiar from Pratt-style propositional dynamic logic).

P 4.27 $\text{ran}\,(\varphi?;\pi_1;\pi_2) \leftrightarrow \text{ran}\,(\text{ran}\,(\varphi?;\pi_1)?;\pi_2)$.

This is the counterpart to the previous principle. Finally, here are three axioms about testing.

P 4.28 $\text{dom}\,(\varphi_1?;\varphi_2?) \leftrightarrow (\varphi_1 \wedge \varphi_2)$.

P 4.29 $\text{ran}\,(\varphi_1?;\varphi_2?) \leftrightarrow (\varphi_1 \wedge \varphi_2)$.

P 4.30 $\text{fix}\,(\varphi?) \leftrightarrow \varphi$.

It should be noted that some of these principles can be simplified if we extend the relational repertoire of the language. For example, if we admit procedure intersection then we can define fixpoints by means of $\text{fix}\,(\pi) \leftrightarrow \text{dom}\,(\pi \cap \top?)$. Of course, the trade-off is that now extra principles for intersection have to be added.

5 Validity and Consequence

While there is an obvious static validity notion for the logic of propositional information transitions (see above), there are several candidates for the notion of 'dynamic valitity' ([2], [26], [7]), which are all easily expressible in the present format.

An information transition π is always accepted if for every information state s, $\langle s, s\rangle \in [\![\pi]\!]$. This is the case iff for every information state s, $s \models \text{fix}\,(\pi)$.

An information transition π is always acceptable if for every information state $s \neq \langle \emptyset, w \rangle$, there is some $s' \neq \langle \emptyset, w \rangle$ with $\langle s, s' \rangle \in [\![\pi]\!]$. This is the case iff for every information state s, $s \models \Diamond\top \rightarrow \langle\pi\rangle\Diamond\top$.

Similarly, while it is clear what the 'static' notion of logical consequence should be (namely, $\Gamma \models \Delta$ iff for every $s \in S$ with $s \models \bigwedge\Gamma$ it holds that $s \models \bigvee\Delta$), there are several candidates for 'dynamic consequence' in this framework ([2]; see [17] for further analysis):

- $\pi_1 \models_1 \pi_2$ iff for all $s \in S$, $s[\![\pi_1]\!]s$ implies $s[\![\pi_2]\!]s$.
- $\pi_1 \models_2 \pi_2$ iff for all $s, s' \in S$ with $s[\![\pi_1]\!]s'$ and $s' \neq \langle \emptyset, w \rangle$ there is an $s'' \neq \langle \emptyset, w \rangle$ with $s'[\![\pi_2]\!]s''$.
- $\pi_1 \models_3 \pi_2$ iff for all $s, s' \in S$, $s[\![\pi_1]\!]s'$ implies $s'[\![\pi_2]\!]s'$.

These are readily expressed in terms of static validity, for we have:

- $\pi_1 \models_1 \pi_2$ iff $\models \mathrm{fix}\,(\pi_1) \rightarrow \mathrm{fix}\,(\pi_2)$.

And for the second one:

- $\pi_1 \models_2 \pi_2$ iff $\models [\pi_1](\Diamond\top \rightarrow \langle\pi_2\rangle\Diamond\top)$.

And the third one:

- $\pi_1 \models_3 \pi_2$ iff $\models \mathrm{ran}\,(\pi_1) \rightarrow \mathrm{fix}\,(\pi_2)$.

6 Information Conveyed by an Update

One way of 'measuring' the information conveyed by an information transition in a state satisfying φ is by means of $\mathrm{ran}\,(\varphi?;\pi)$.

Example 14 *The information conveyed by the update $(\Diamond p)^u;(\Box\neg p)^u$ (one of Veltman's key examples) in the state of complete ignorance is calculated as follows:*

$$\begin{array}{rcl} \mathrm{ran}\,(\top?;(\Diamond p)^u;(\Box\neg p)^u) & \leftrightarrow & \mathrm{ran}\,(\mathrm{ran}\,(\top?;(\Diamond p)^u)?;(\Box\neg p)^u) \\ & \leftrightarrow & \mathrm{ran}\,(\Diamond p?;\Box\neg p^u) \\ & \leftrightarrow & \Box\neg p. \end{array}$$

The second step uses Principle 4.15, the third Principle 4.13.

Example 15 *The information conveyed by the update $(\Box\neg p)^u;(\Diamond p)^u$ in the state of complete ignorance is calculated as follows:*

$$\begin{array}{rcl} \mathrm{ran}\,(\top?;(\Box\neg p)^u;(\Diamond p)^u) & \leftrightarrow & \mathrm{ran}\,(\mathrm{ran}\,(\top?;(\Box\neg p)^u)?;(\Diamond p)^u) \\ & \leftrightarrow & \mathrm{ran}\,(\mathrm{ran}\,(\Box\top?;(\Box\neg p)^u)?;(\Diamond p)^u) \\ & \leftrightarrow & \mathrm{ran}\,((\Box\neg p)?;(\Diamond p)^u) \\ & \leftrightarrow & \Box\neg p \wedge \Diamond p. \end{array}$$

This is K45 equivalent to $\bot$, which shows that this update will never succeed. (Note the use of Principle 4.16 in the third step.)

In fact, since updating with Veltman's *might* p corresponds to updating with $\Diamond\top \rightarrow \Diamond p$, the 'rational reconstruction' of Veltman's example is slightly different:

Example 16 *The information conveyed by the update*

$$(\Box\neg p)^u; (\Diamond\top \rightarrow \Diamond p)^u$$

in the state of complete ignorance is calculated as follows:

$$\begin{aligned}
&\text{ran}\,(\top?; (\Box\neg p)^u; (\Diamond\top \rightarrow \Diamond p)^u)\\
&\leftrightarrow \text{ran}\,(\text{ran}\,(\top?; (\Box\neg p)^u)?; (\Diamond\top \rightarrow \Diamond p)^u)\\
&\leftrightarrow \text{ran}\,(\text{ran}\,(\Box\top?; (\Box\neg p)^u)?; (\Diamond\top \rightarrow \Diamond p)^u)\\
&\leftrightarrow \text{ran}\,((\Box\neg p)?; (\Diamond\top \rightarrow \Diamond p)^u)\\
&\leftrightarrow \Box\neg p \wedge (\Diamond\top \rightarrow \Diamond p).
\end{aligned}$$

This is K45 equivalent to $\Box\bot$*, which shows that this update will get the audience in an inconsistent state of information.*

To see that the final step in the calculation is correct, note that the update procedure $(\Diamond\top \rightarrow \Diamond p)^u$ *is equivalent to the procedure* $(\Diamond\top?; \Diamond p^u) \cup \Box\bot^u$*, where* $\cup$ *denotes choice between procedures. An obvious principle governing choice is:*

$$\text{ran}\,(\pi_1; (\pi_2 \cup \pi_3)) \leftrightarrow (\text{ran}\,(\pi_1; \pi_2) \vee \text{ran}\,(\pi_1; \pi_3)).$$

Using this and the other principles, the final step can easily be validated.

In general, a transition π is consistent in state s iff $\langle\pi\rangle\Diamond\top?$ holds in s. This expresses that a transition from s via π is possible which does end up in a consistent state of information.

It is tempting, especially in the light of the dynamic consequence notion $\models_1$, to equate the information conveyed by an information transition π with fix (π). But note that the combination $\Diamond p^u; \Box\neg p^u$ does not have a fixpoint, while, as we have just seen, updating with that information starting from complete ignorance yields a *consistent* information state. Is there perhaps something funny about updates of the form $\Diamond F^u$?

From the present perspective, such updates are indeed strange. Recall that our intention is to model the knowledge of an *audience* addressed by a single speaker. If one assumes that $\Diamond F^u$ corresponds to an assertion by the speaker, then the update would have to correspond to an assertion about the state of knowledge of the audience: the speaker states that F is consistent with what the audience knows already. This assertion would correspond to something like 'I take it that you know that F is possible'. But this is not an assertion in the sense of 'statement influencing the state of knowledge of the audience'.

Compare this with the 'updates' of the form 'p may be the case', or 'maybe p' in [26]. Veltman renders the assertion 'maybe p' as an update with $\Diamond\top \rightarrow \Diamond p$. The big difference between Veltman's information states and ours is that

Veltman's information states model the knowledge of a single agent reporting on how he or she processes incoming information, while ours model the knowledge of the *audience* addressed by a single speaker.

In our set-up, updates of the form $\Diamond\top \rightarrow \Diamond p$ are not consistency checks of one's own knowledge, as they are for Veltman, but statements about the knowledge of the audience. Since we cannot in general assume that a speaker has complete knowledge of what his or her audience believes, such statements are rather pointless. On the other hand, checking the knowledge of the audience by means of a test $\varphi?$ may still make eminent sense, as we will see in the next section.

In the present set-up, an assertion of the form 'maybe p' should be construed as an invitation to the audience to *reconsider* the truth of p, i.e., such an assertion is a *downdate*, and it has the form $(\neg\Diamond p)^d$, or equivalently $(\Box\neg p)^d$.

If one imposes the constraint that information transitions always be compositions of basic units of the forms $\Box F^u$ and $\Box F^d$, possibly interspersed with tests, then information content can always be described in terms of fixpoints. A consistent fixpoint for $\Diamond p^u; \Box\neg p^u$ does not exist, but for $\Box\neg p^d; \Box\neg p^u$ it does; indeed, any state where $\Box\neg p$ holds is such a fixpoint.

7 Expressing Presuppositions

We have seen that realistic information transitions in our set-up have the forms $(\Box F)^u$ or $(\Box F)^d$. In case such an information transition has a presupposition, we may assume that this has the form of a test to see whether something is known in the current context, i.e., a test of the form $\Box\varphi?$ Not only updates may have presuppositions, witness (1).

(1) Maybe the king of France is eating frog legs.

If I assert (1) then I presuppose that the king of France exists, and I invite my audience to revise their belief that his majesty is doing something other than eating frog legs. (We have seen that 'maybe statements' turn up as downdates of the form $\Box F^d$ in the present framework.) So this is a downdate with presupposition.

The framework presented above has all the machinery in place to express presuppositions. We can express the requirement that updating with α has presupposition φ in state s by means of the complex update $\varphi?; \alpha^u$.

Example 17 *$\Box p?; \Box q^u$ succeeds in state s iff $s \models \Box p$, and effects a transition to a state s' with $s \sqsubseteq s'$ and $s' \models \Box q$. (And of course also $s' \models \Box p$, for $\Box p$ is a persistent formula.)*

The semantic clause for $[\![\varphi?; \alpha^u]\!]$ is given by:

$$[\![\varphi?; \alpha^u]\!] \quad = \quad [\![\varphi?]\!] \circ [\![\alpha^u]\!]$$

$$= \{\langle s, s'\rangle \in [\![\alpha^u]\!] \mid s \models \varphi\}$$
$$= [\![\alpha^u]\!] - \{\langle s, s'\rangle \in S \times S \mid s \not\models \varphi\}.$$

As the relational interpretation demonstrates, $\varphi?;\alpha^u$ is interpreted as an update with α under the presupposition that φ holds in the current context. Similarly, $\varphi?;\alpha^d$ is interpreted as a downdate with α under the presupposition that φ holds in the current context.

Calculating the presupposition of an information transition π consists in finding a specification of the information states s for which there is an s' with $\langle s, s'\rangle \in [\![\pi]\!]$. In these cases we say that the transition π *does not abort*. Conversely, the presupposition failure conditions of an information transition π consist of a specification of the information states s for which there is no s' with $\langle s, s'\rangle \in [\![\pi]\!]$. We say in these cases that transition π *aborts* in state s.

To calculate the presupposition of an update α^u, we have to check the conditions on states s under which the relation $[\![\alpha^u]\!]$ does have a successor for s. These are given by the following schemata, which are derivable from the principles in the previous section:

T 7.1 $\langle\varphi?;\alpha^u\rangle\top \leftrightarrow \varphi \wedge \langle\alpha^u\rangle\top$.

This expresses that a simplex update with presupposition can be performed if and only if the presupposition holds in the current information state and the update without presupposition is possible in the current context.

T 7.2 $\langle\varphi?;\alpha^d\rangle\top \leftrightarrow \varphi \wedge \langle\alpha^d\rangle\top$.

This expresses that a downdate under presupposition is possible iff the presupposition holds in the current information state and the downdate without presupposition is possible in that state. Note that the presupposition of an information transition is nothing but the weakest preconditions for success of that transition, in the well known computer science sense.

For a concrete example, assume that the lexical presupposition of being a bachelor consists of being male plus being adult. We do not yet look inside the basic propositions built from these predicates, so we merely say that example (2) presupposes the conjunction of (3) and (4), and asserts (5).

(2) Jan is a bachelor.

(3) Jan is male.

(4) Jan is adult.

(5) Jan is unmarried.

Basic propositions that do not themselves have presuppositions can be represented using basic proposition letters. Let us use p for (3), q for (4), and $\neg r$ for (5).

The update for *Jan is a bachelor* does have the presuppositions *Jan is male* and *Jan is adult*, and (after the update with these presuppositions) it makes the assertion *Jan is unmarried*, so it can be represented as $\Box(p \wedge q)?;\Box\neg r^u$. Let P be the set $\{p, q, r\}$. Information states for this fragment are built from valuations in $\{p, q, r\} \to \{0, 1\}$.

(6) Jan is male. Jan is a bachelor.

The meaning of the update with the sequence (6) is given by $[\![\Box p^u; \Box(p \wedge q)?; \Box\neg r^u]\!]$. We write this out to check its meaning:

$$\begin{array}{l} s[\![\Box p^u; \Box(p \wedge q)?; \Box\neg r^u]\!]s' \\ \text{iff } \exists s'' : s[\![\Box p^u]\!]s'' \text{ and } s''[\![\Box(p \wedge q)?; \Box\neg r^u]\!]s' \\ \text{iff } \exists s'' : s[\![\Box p^u]\!]s'' \text{ and } s'' \models \Box(p \wedge q) \text{ and } s''[\![\Box\neg r^u]\!]s' \\ \text{iff } \exists s'' : s[\![\Box p^u]\!]s'' \text{ and } s'' \models \Box q \text{ and } s''[\![\Box\neg r^u]\!]s'. \end{array}$$

It follows from this that the non-abort condition on s is given by:

$$\exists s' : s[\![\Box p^u]\!]s' \text{ and } s' \models \Box q.$$

This is the case iff $s \models \Box(p \to q)$. In other words, the presupposition is that it is known in the current context that if Jan is male then he is adult. We can also derive this in the calculus, as follows:

$$\begin{array}{lll} \langle\Box p^u; \Box(p \wedge q)?; \Box\neg r^u\rangle\top & \leftrightarrow & \langle\Box p^u\rangle\langle\Box(p \wedge q)?\rangle\langle\Box\neg r^u\rangle\top \\ & \leftrightarrow & \langle\Box p^u\rangle\Box(p \wedge q) \\ & \leftrightarrow & (\Box(p \wedge q))^p \\ & \leftrightarrow & \Box(p \to q). \end{array}$$

An information transition π *holds* in a context if the transition does not affect that context. For the example case, we can spell out the conditions for this as follows:

$$\begin{array}{l} s[\![\Box p^u; \Box(p \wedge q)?; \Box\neg r^u]\!]s \\ \text{iff } \exists s' : s[\![\Box p^u]\!]s' \text{ and } s'[\![\Box(p \wedge q)?; \Box\neg r^u]\!]s \\ \text{iff } s[\![\Box p^u]\!]s \text{ and } s[\![\Box(p \wedge q)?; \Box\neg r^u]\!]s \\ \text{iff } s \models \Box p \text{ and } s \models \Box(p \wedge q) \text{ and } s \models \Box\neg r \\ \text{iff } s \models \Box(p \wedge q \wedge \neg r). \end{array}$$

To end this section, note that the present perspective sheds an illuminating light on the phenomenon known as presupposition accommodation. Presupposition accommodation is the process performed by a benevolent audience in case an assertion is made with a presupposition which does not hold in the current context. In case the audience does not know that Bill is married and someone gossips that Bill's wife wants a divorce then the context is tacitly updated with the presupposition of that assertion as well. If we allow complex updates (by an obvious extension of the language), we can model this accommodation process as a shift from transition π to transition $(\langle\pi\rangle\top)^u; \pi$.

8 Embedded Presuppositions

Until now we have only considered presuppositions under sequential composition. If we assume presuppositions to have the form $\Box F?$, then a typical sequential composition of two updates under presupposition looks like this:

$$\Box F_1?; \Box G_1^u; \Box F_2?; \Box G_2^u.$$

The presupposition of this is given by:

$$\langle \Box F_1?; \Box G_1^u; \Box F_2?; \Box G_2^u \rangle \top.$$

This reduces to:

$$\Box F_1 \wedge \langle \Box G_1^u \rangle \langle \Box F_2? \rangle \top,$$

and further to:

$$\Box F_1 \wedge \Box (G_1 \rightarrow F_2),$$

with end result:

$$\Box (F_1 \wedge (G_1 \rightarrow F_2)).$$

Thus, we see that the boxed presupposition of the sequential composition of two updates is given by conjunction of the boxed presupposition of the first and the boxed implication of assertion of the first and presupposition of the second.

To consider presuppositions under negation, let us forget about 'downward' transitions π for the moment. (Note that we cannot define the negation of a downdate with α as the assertion that downdating with α itself would lead to inconsistency, for if a downdate with α is possible at all, it will never lead to inconsistency. Also, the negation of a downdate with α cannot be construed as the assertion that downdating with α is impossible, for the impossibility of a downdate with α just means that α is a logical truth.)

An *update transition* is a transition π with the property that $s[\![\pi]\!]s'$ implies $s \sqsubseteq s'$. The *presupposition* of an update transition is the set $\{s \in S \mid \exists s' \sqsupseteq s : s[\![\pi]\!]s'\}$. This set is characterized by $\langle \pi \rangle \top$. The *content* of an update transition is the set $\{s \in S \mid s[\![\pi]\!]s\}$. This set is characterized by fix (π).

Negating an update α^u can be construed as updating with the assertion that making update α itself would yield inconsistency. Thus, we can stipulate:

$$\neg(\alpha^u) = ([\alpha^u]\Box\bot)^u.$$

If we define $\varphi_1^u \Rightarrow \varphi_2^u$ as $\neg(\varphi_1^u; \neg\varphi_2^u)$ (update implication) and $\varphi_1^u \sqcup \varphi_2^u$ as $\neg(\neg\varphi_1^u; \neg\varphi_2^u)$ (update disjunction), then we can easily derive:

- $[\![\neg(\Box F^u)]\!] = [\![\Box\neg F^u]\!]$,
- $[\![\Box F^u \Rightarrow \Box G^u]\!] = [\![\Box(F \rightarrow G)^u]\!]$,
- $[\![\Box F^u \sqcup \Box G^u]\!] = [\![\Box(F \vee G)^u]\!]$.

In the general case where an update transition π may have a presupposition, we have two options: either the negation preserves the presupposition or it cancels it. We will explore the first option. Suppose π is an upward transition. Then we define $\neg\pi$ as follows:

$$\neg\pi \stackrel{\text{def}}{=} \langle\pi\rangle\top?;([\pi]\Box\bot)^u.$$

Thus, $\neg\pi$ has the same presupposition as π, but it updates to the minimal state(s) where updating with π would yield inconsistency.

As regards the second option, an obvious choice for a definition of negated updating which cancels presuppositions is $([\pi]\Box\bot)^u$. This is an update to a state where doing π itself would lead to inconsistency. Now suppose π has a presupposition, let us say $\Box p?$, and assume $\Diamond\neg p$ holds in the current state. Then $([\pi]\Box\bot)^u$ would loop in the current state, showing that $([\pi]\Box\bot)^u$ does not have $\Box p?$ as presupposition.

If we spell out the semantics for $\neg\pi$ we get this:

$$[\![\neg\pi]\!] = A - B,$$

where

$$\begin{aligned} A \;=\; & \{\langle s,s'\rangle \in S\times S \mid s \sqsubseteq s', s'[\![\pi]\!]\langle\emptyset,w\rangle \text{ (for some } w), \\ & \text{and for all } s'' \text{ with } s \sqsubseteq s'' \sqsubseteq s' \;\&\; s''[\![\pi]\!]\langle\emptyset,w\rangle \text{ (for some } w) \\ & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad s' \sqsubseteq s''\}, \end{aligned}$$

and

$$B = \{\langle s,s'\rangle \mid s \sqsubseteq s', s \models [\pi]\bot\}.$$

Note that the earlier stipulation for $\neg(\alpha^u)$ is a special case of this.

We can now define dynamic implication and dynamic disjunction for the general case of update transitions π_1 and π_2. To calculate what happens to presuppositions under 'dynamic implication', we can make use of the fact that

$$\neg(\Box F?;\Box G^u) = \Box F?;\Box\neg G^u$$

and of the fact that for all π:

$$\langle\neg\pi\rangle\top \leftrightarrow \langle\pi\rangle\top.$$

This is just a reflection of the fact that $\neg\pi$ has the same presupposition as π.

Here is the calculation for presupposition under dynamic implication:

$$\begin{aligned} & \langle(\Box F_1?;\Box G_1^u) \Rightarrow (\Box F_2?;\Box G_2^u)\rangle\top \\ \leftrightarrow\; & \langle\neg((\Box F_1?;\Box G_1^u);\neg(\Box F_2?;\Box G_2^u))\rangle\top \\ \leftrightarrow\; & \langle\neg(\Box F_1?;\Box G_1^u;\Box F_2?;\Box\neg G_2^u)\rangle\top \\ \leftrightarrow\; & \langle\Box F_1?;\Box G_1^u;\Box F_2?;\Box\neg G_2^u\rangle\top \\ \leftrightarrow\; & \Box F_1 \wedge \langle\Box G_1^u\rangle\Box F_2 \\ \leftrightarrow\; & \Box(F_1 \wedge (G_1 \rightarrow F_2)). \end{aligned}$$

For presupposition under 'dynamic disjunction' we get:

$$\begin{aligned}
&\langle(\Box F_1?;\Box G_1^u)\sqcup(\Box F_2?;\Box G_2^u)\rangle\top\\
&\leftrightarrow\langle\neg(\neg(\Box F_1?;\Box G_1^u);\neg(\Box F_2?;\Box G_2^u))\rangle\top\\
&\leftrightarrow\langle\neg(\Box F_1?;\Box\neg G_1^u;\Box F_2?;\Box\neg G_2^u)\rangle\top\\
&\leftrightarrow\Box F_1\wedge\langle\Box\neg G_1^u\rangle\Box F_2\\
&\leftrightarrow\Box(F_1\wedge(G_1\vee F_2)).
\end{aligned}$$

Thus, calculating presuppositions of complex updates in terms of assertions and presuppositions of their components gives the following table:

update procedure	presupposition
$(\Box F_1?;\Box G_1^u);(\Box F_2?;\Box G_1^u)$	$\Box(F_1\wedge(G_1\rightarrow F_2))$.
$\neg(\Box F?;\Box G^u)$	$\Box F$.
$(\Box F_1?;\Box G_1^u)\Rightarrow(\Box F_2?;\Box G_1^u)$	$\Box(F_1\wedge(G_1\rightarrow F_2))$.
$(\Box F_1?;\Box G_1^u)\sqcup(\Box F_2?;\Box G_1^u)$	$\Box(F_1\wedge(G_1\vee F_2))$.

This is a *boxed* version of Karttunen's table of presupposition projection for 'and', 'not', 'if then' and 'or'.

9 Digression: Error States

The treatment of presupposition failure in terms of error states of [4] [5] is motivated by an obvious parallel between presupposition failure in natural language and error abortion in imperative programming. Consider the program statement (7).

(7) `x := y/z`

If at the point of execution of this statement register z happens to contain the value 0 then execution will be aborted with an error statement like 'Floating point error: division by zero attempted'.

(8) `IF z <> 0 THEN x := y/z`

In the statement (8) the dangerous case of $z = 0$ is tested for in the program code, and the danger of error abortion is staved off.

This suggests analyzing presupposition failure as 'moving to an error state'. Taking error abortion into account in the semantics of deterministic imperative programming boils down to changing the semantic interpretation function for program statements into a partial function: error abortion is the case where there is no next state.

The epistemic state of a program always consists of the current memory state, so it turns out that error abortion analysis arises as a special case of the present epistemic analysis, where there are just two state sets: $\langle\{w\},w\rangle$ (the consistent state) and $\langle\emptyset,w\rangle$ (the inconsistent state). Thus we get:

Success case $w(p) = 1, w(q) = 1$:

$$\langle\{w\}, w\rangle[\![\Box p?; \Box q^u]\!]\langle\{w\}, w\rangle$$

Failure case $w(p) = 1, w(q) = 0$:

$$\langle\{w\}, w\rangle[\![\Box p?; \Box q^u]\!]\langle\emptyset, w\rangle$$

Error abortion case $w(p) = 0$:

$$\langle\{w\}, w\rangle[\![\Box p?; \Box q^u]\!] \text{ ERROR} .$$

In nondeterministic imperative programming, program statements are interpreted as relations. Taking error abortion into account here means changing the interpretation relation into a partial relation. Executing a program statement π in state s now gives three possibilities: (1) there are proper next states (and maybe the program can also make a transition to 'error'), (2) there are no next states, and (3) the program can only make a transition to the error state. Again, an error state semantics for dynamic predicate logic ([8]) in the style of [4] turns out to be a special case of the present epistemic analysis, where there is just one first order model around, and where the possible states are the assignment functions over this single model (intuitively, the states encode the interpretations for indefinite noun phrases that are 'still in the running'). In this set-up, an update with presupposition for 'the king of France is bald' could be rendered as $\Box\exists!xKx?; \Box(x :=?; Kx; Bx)^u$. This gets us into the topic of the next section.

10 Presupposition and Quantification

As an example of a presupposition of quantified expressions, we look at the case of uniqueness presuppositions of singular definite descriptions. Dynamic versions of predicate logic have been proposed to deal with growth of information about anaphoric possibilities of a piece of natural language text. The most important ones of these are file change semantics ([11]), discourse representation theory ([16]), and dynamic predicate logic ([8]). This kind of dynamics can be, but need not be, combined with the dynamics of information updating using predicate logical formulas. Here we will concentrate on 'epistemic dynamics' for purposes of exposition, and sketch a system of information updating for standard predicate logic.

To model information growth in predicate logic, the simplest possible set-up confines attention to one particular predicate logical model M for the language under consideration, and then uses sets of variable assignments for that model as information states. Thus, if $M = \langle \text{dom}\,(M), \text{int}\,(M)\rangle$ is given, and if V is the set of variables for the predicate logical language under consideration, then

$A = \mathrm{dom}\,(M)^V$ is the set of assignments, and $S = \{\langle i, a\rangle \mid i \subseteq A, a \in A\}$ is the set of information states. The relation $\sqsubseteq$ on S is given by $s \sqsubseteq s'$ iff $s = \langle i, a\rangle, s' = \langle j, a\rangle$ and $i \supseteq j$. Absurd information states are states of the form $\langle \emptyset, a\rangle$.

Fix a language L: let a set of individual constants C and a set of predicate constants P_n (where n denotes the arity of the constant) be given. Assume V is a set of individual variables. Assume $c \in C$, $v \in V$, $R \in P_n$.

$$\begin{array}{rcl} t & ::= & c \mid v \\ \varphi & ::= & \bot \mid Rt_1 \cdots t_n \mid t_1 = t_2 \mid \neg\varphi \mid (\varphi_1 \wedge \varphi_2) \mid \exists v\varphi. \end{array}$$

Let L_1 be the language that allows epistemic statements over L:

$$\psi \quad ::= \quad \varphi \mid \Diamond\varphi \mid \Box\varphi.$$

The Tarskian satisfaction relation $M \models_a \varphi$ is defined in the usual manner. In terms of this we define an interpretation for L_1 (i.e., a specification function σ) as follows:

$$\begin{array}{rcl} \sigma(\varphi) & = & \{s \in S \mid s = \langle i, a\rangle \text{ and } M \models_a \varphi\}, \\ \sigma(\Diamond\varphi) & = & \{\langle i, a\rangle \in S \mid \exists a' \in i \text{ with } M \models_{a'} \varphi\}, \\ \sigma(\Box\varphi) & = & \{\langle i, a\rangle \in S \mid \forall a' \in i : M \models_{a'} \varphi\}. \end{array}$$

The dynamically extended language now becomes:

$$\begin{array}{rcl} t & ::= & c \mid v \\ \varphi & ::= & \bot \mid Rt_1 \cdots t_n \mid t_1 = t_2 \mid \neg\varphi \mid (\varphi_1 \wedge \varphi_2) \mid \exists v\varphi \\ \psi & ::= & \varphi \mid \Diamond\varphi \mid \Box\varphi \mid \mathrm{dom}\,(\pi) \mid \mathrm{ran}\,(\pi) \mid \mathrm{fix}\,(\pi). \\ \pi & ::= & \psi^u \mid \psi? \mid (\pi_1; \pi_2). \end{array}$$

Minimal updates are defined as before. The definitions of the dynamic operators are the same as before. Again, presuppositions are given by dom (π) and assertions by fix (π). The distinction between presupposition failure and updating with a piece of information inconsistent with the current information state is given by 'no further transition possible' versus 'transition to an absurd state'.

(9) The king of France is eating a frog.

An update with the information expressed by (9) is expressed in this format as (10).

(10) $\Box\exists!xKx?; \Box\exists x(Kx \wedge \exists y(Fy \wedge Exy))^u$.

The presupposition is given by:

(11) $\langle\Box\exists!xKx?; \Box\exists x(Kx \wedge \exists y(Fy \wedge Exy))^u\rangle\top$.

This is equivalent to:

(12) $\Box\exists!xKx$.

The assertion is given by:

(13) fix $(\Box\exists!xKx?;\Box\exists x(Kx \wedge \exists y(Fy \wedge Exy))^u)$.

This is equivalent to:

(14) $\Box(\exists!xKx \wedge \exists x(Kx \wedge \exists y(Fy \wedge Exy)))$.

Note that because our epistemic states are based on a single first order model the epistemic operators $\Diamond$ and $\Box$ are not very expressive. They serve to make the distinction between being able to make an update to an absurd state (uttering a falsehood) and not being able to make a further transition at all (error abortion). Indeed, since possible worlds are variable assignments, if F is a predicate logical formula without free variables, then the difference between $\Box F$ and $\Diamond F$ shows up only in absurd information states.

Of course, the epistemic modalities become more expressive once we redefine our information states in terms of *sets* of first order models.

11 Conclusion

We have sketched a system of epistemic dynamic logic to model presupposition and presupposition failure. Lots of logical questions remain to be answered. For instance: is the logic of propositional up- and downdating decidable? We conjecture that it is. What does a complete axiomatisation of this logic look like? What are the properties of the systems one gets by imposing further conditions on the information structures? What do the obvious variations on the combination of presupposition and quantification look like? The simplest variation is to replace standard predicate logic by dynamic predicate logic. This yields the dynamic error state semantics of [4]. Another variation is to replace states based on single first order models by states based on sets of models. This gives an epistemic first order update logic. Finally, we can combine the two in various ways (see [6] and [9]). In all cases, the main thing is to get at the right definition of the information structure $\langle S, \sqsubseteq \rangle$. [15] provide a very useful starting point for this in the form of an overview of current systems of dynamic logic from the perspective of information structures.

Acknowledgements

I would like to thank Jan Jaspars for some illuminating conversations on the topic of information updating, and Makoto Kanazawa for his very helpful comments on two earlier versions of this paper.

References

[1] Beaver, D.I. 1992. The Kinematics of Presupposition. In *Proceedings of the Eighth Amsterdam Colloquium*, ed. P. Dekker and M. Stokhof, 17–36. ILLC, University of Amsterdam.

[2] Benthem, J. van. 1991. Logic and the Flow of Information. Technical Report LP-91-10. ILLC, University of Amsterdam.

[3] Chellas, B.F. 1980. *Modal Logic: An Introduction.* Cambridge University Press.

[4] Eijck, J. van. 1993. The Dynamics of Description. *Journal of Semantics* 10:239–267.

[5] Eijck, J. van. 1994. Presupposition Failure — A Comedy of Errors. *Formal Aspects of Computing* 6A:766–787.

[6] Eijck, J. van, and G. Cepparello. 1994. Dynamic Modal Predicate Logic. In *Dynamics, Polarity and Quantification*, ed. M. Kanazawa and C.J. Piñón. 251–276. CSLI, Stanford.

[7] Eijck, J. van, and F.J. de Vries. 1995. Reasoning About Update Logic. *Journal of Philosophical Logic* 24:19–45.

[8] Groenendijk, J., and M. Stokhof. 1991. Dynamic Predicate Logic. *Linguistics and Philosophy* 14:39–100.

[9] Groenendijk, J., M. Stokhof, and F. Veltman. June, 1994. Coreference and Modality. Manuscript, ILLC, Amsterdam.

[10] Halpern, J., and Y. Moses. 1985. Towards a Theory of Knowledge and Ignorance: Preliminary Report. In *Logics and Models of Concurrent Systems*, ed. K.R. Apt. 459–476. Springer.

[11] Heim, I. 1982. *The Semantics of Definite and Indefinite Noun Phrases.* Doctoral dissertation, University of Massachusetts, Amherst.

[12] Heim, I. 1983. On the Projection Problem for Presuppositions. *Proceedings of the West Coast Conference on Formal Linguistics* 2:114–126.

[13] Heim, I. 1992. Presupposition Projection and the Semantics of the Attitude Verbs. *Journal of Semantics* 9(3):183–221. Special Issue: Presupposition, Part 1.

[14] Jaspars, J. to appear. Partial Up and Down Logic. *Notre Dame Journal of Formal Logic.*

[15] Jaspars, J., and E. Krahmer. 1995. Unified Dynamics. Manuscript, CWI, Amsterdam.

[16] Kamp, H. 1981. A Theory of Truth and Semantic Representation. In *Formal Methods in the Study of Language*, ed. J. Groenendijk et al. Mathematisch Centrum, Amsterdam.

[17] Kanazawa, M. 1994. Completeness and Decidability of the Mixed Style of Inference with Composition. In *Proceedings 9th Amsterdam Colloquium*, ed. P. Dekker and M. Stokhof. 377–390. ILLC, Amsterdam.

[18] Karttunen, L. 1973. Presuppositions of Compound Sentences. *Linguistic Inquiry* 4:169–193.

[19] Karttunen, L. 1974. Presupposition and Linguistic Context. *Theoretical Linguistics* 181–194.

[20] Karttunen, L., and S. Peters. 1979. Conventional Implicature. In *Syntax and Semantics 11: Presupposition*, ed. C.-K. Oh and D. Dinneen. 1–56. Academic Press.

[21] Krahmer, E. 1994. Partiality and dynamics; theory and application. In *Proceedings 9th Amsterdam Colloquium*, ed. P. Dekker and M. Stokhof. 391–410. ILLC, Amsterdam.

[22] Moore, R.C. 1984. Possible world semantics for autoepistemic logic. In *Proceedings AAAI Workshop on Non-Monotonic Reasoning*, 344–354. New Paltz, NY.

[23] Rijke, M. de. 1994. Meeting Some Neighbours. In *Logic and Information Flow*, ed. J. van Eijck and A. Visser. 170–195. MIT Press, Cambridge, Mass.

[24] Stalnaker, R. 1972. Pragmatics. In *Semantics of Natural Language*, ed. D. Davidson and G. Harman. 380–397. Reidel.

[25] Stalnaker, R. 1974. Pragmatic Presuppositions. In *Semantics and Philosophy*, ed. M.K. Munitz and P.K. Unger. 197–213. New York University Press.

[26] Veltman, F. 1991. Defaults in Update Semantics. Technical report. Department of Philosophy, University of Amsterdam. To appear in the *Journal of Philosophical Logic.*

[27] Zeevat, H. 1992. Presupposition and Accommodation in Update Semantics. *Journal of Semantics* 9(4):379–412. Special Issue: Presupposition, Part 2.

3D Computational Steering with Parametrized Geometric Objects

Jurriaan D. Mulder
Centrum voor Wiskunde en Informatica
P.O. Box 4097,1009 AB Amsterdam, The Netherlands.

Jarke J. van Wijk
Netherlands Energy Research Foundation ECN
P.O. Box 1, 1755 ZG Petten, The Netherlands.

Abstract

Computational Steering is the ultimate goal of interactive simulation: researchers change parameters of their simulation and immediately receive feedback on the effect. We present a general and flexible graphics tool that is part of an environment for Computational Steering developed at CWI. It enables the researcher to interactively develop his own interface with the simulation. This interface is constructed with 3D Parametrized Geometric Objects. The properties of the objects are parametrized to output data and input parameters of the simulation. The objects visualize the output of the simulation while the researcher can steer the simulation by direct manipulation of the objects. Several applications of 3D Computational Steering are presented.

1 Introduction

Computational Steering can be considered as the ultimate goal of interactive computing. Computational Steering enables the researcher to change parameters of the simulation as the simulation is in progress, while viewing the simulation results simultaneously. Thus, the researcher can change variables on the fly, correct erroneous values for input, and, most important, the researcher can gain a large amount of additional insight if he can immediately observe the effect of changes in input parameters to dependent variables.

In recent years many methods, techniques, and packages have been developed for Scientific Visualization. However, most of these systems are limited to post-processing of data-sets, and therefore do not allow the user to interact with the simulation's data and input parameters. At CWI, a software environment for computational steering is

being developed [15]. The aim is to provide the researcher with a general, flexible environment in which existing and newly developed scientific simulations can easily be incorporated to build custom computational steering applications.

In this paper, we present a 3D graphics tool (the *PGO editor*) which is part of this environment. The tool allows the user to directly interact with the simulation based on its visualization. The user can construct his own user interface for the visualization of the simulation's results and input parameters. This interface is built up with 3D Parametrized Geometric Objects (PGOs). The user can then change the values of the input parameters and state variables while the simulation is running by direct manipulation of these objects.

In section 2 we briefly describe the computational steering environment developed at CWI along with related work on computational steering environments, 3D visualization, and user interface creation. In section 3 the concept of 3D parametrized geometric objects is described. In section 4 the 3D interaction with these objects by direct manipulation is discussed. Some examples of the use of the graphics tool in computational steering applications are given in section 5, followed by a discussion and conclusion in section 6.

2 Related Work

2.1 Computational Steering Environments

Although many researchers have pointed out the importance of computational steering in next generation visualization systems (see for instance [4]), only few actual applications of visualization systems for steering have been developed. One example of such an application system is given by Marshall et al., who present a system for the visualization and simulation steering of a 3D turbulence model of Lake Erie [8, 16]. Examples of more general approaches can be found in [3] (the Visualization and Application Steering environment VASE), [5] (a rudimentary steering implementation with the use of AVS), and [6] (on the problems and advantages of the integration of scientific computations and visualization into one environment). However, no general applicable computational steering environment has yet evolved from these developments.

2.2 The Computational Steering Environment at CWI

The Computational Steering Environment which is being developed at CWI is based on two major concepts: A Data Manager which takes care of centralized data storage and event notification, and other processes called *satellites*, which can connect to and communicate with the Data Manager by the use of a 'publish and subscribe' paradigm, see figure 1.

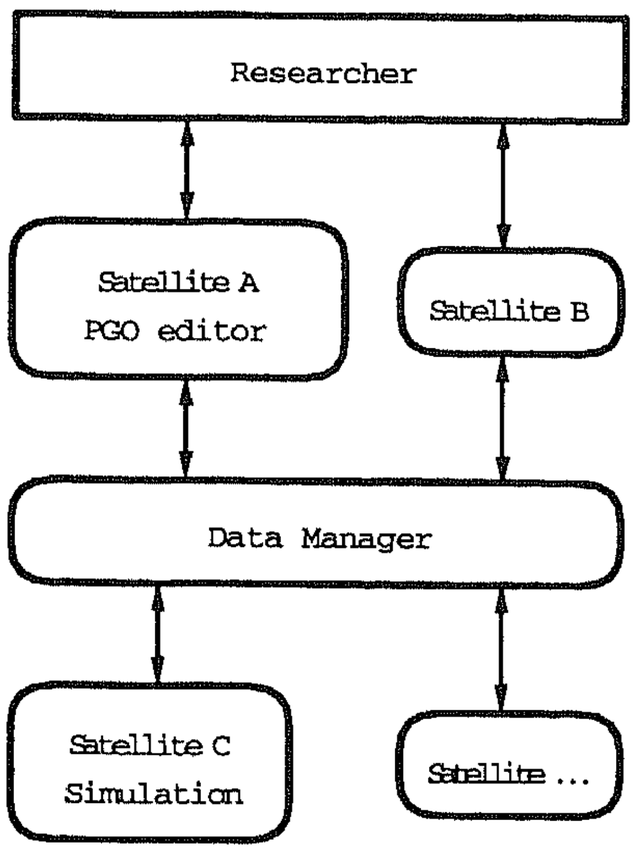


Figure 1: Architecture of the Computational Steering Environment.

The Data Manager

The central process in the CSE is the Data Manager. Other processes such as the simulation and visualization modules, can connect to and communicate with the Data Manager. The purpose of the Data Manager is twofold. A database of variables is managed, and it takes care of event notification. Satellites can create and read/write variables, and they can subscribe to events, such as notification of mutations of a particular variable. Thus, the Data Manager enables the different satellites to use the same data and to communicate with other satellites.

Satellites

Different kinds of processes can be connected to the Data Manager and thereby become satellites. The connection of satellites to the Data Manager is easily achieved with the use of a small Application Programmers Interface (API). If we connect a simulation to the Data Manager such that the output data as well as the simulation's steerable items are stored in and retrieved from the Data Manager, we are able to visualize the output data of the running simulation, and in addition steer the simulation by the use of one or more visualization and data manipulation satellites. Such satellites retrieve and manipulate the simulation's data and parameters present in the Data Manager. Several general satellites have already been developed for this purpose, such as a 2D visualization and manipulation satellite (the 2D PGO editor), a logging satellite, a calculator

satellite, and a slicing satellite.

2.3 3D Visualization and Interface Creation

Accurate visualizations and user interfaces are essential in computational steering applications. By allowing the researcher to construct his own visualization and interface with the simulation, they will be according to the researcher's wishes and demands, and can easily be adapted if the researcher's focus of interest changes.

In addition, the visualization and user interface satellite can be kept general applicable and therefore be used for different kinds of steering applications.

Building one's own custom 3D visualization and/or interface from geometric primitives is a topic of interest in both the visualization and user interface community. In [10] a tool called Glyphmaker is described, developed for data visualization/analysis. It allows users to build customized representations of multivariate data and provides interactive tools to explore the patterns in and relations between the data. With the provided primitives points, lines, spheres, cuboids, cylinders, cones, and arrows, the user can draw glyphs using the 3D Glyph Editor. Properties of the glyphs can be bound to data using the Glyph Binder. Raw (simulation) data is transcribed by the Read Module into Explorer data structures which are used by the Glyph Binder. These bindings however, are only uni-directional; only from the data to the glyphs. Therefore, Glyphmaker does not allow the user to steer the simulation by manipulating the geometric objects.

In [11] an architecture for an extensible 3D interface toolkit is presented. The toolkit can be used for construction and rapid prototyping of 3D interfaces, interactive illustrations, and 3D widgets. By direct manipulation of 3D primitives through a visual language, widgets, interface objects, and application objects are constructed whose geometry is affinely constrained. The four basic primitives of the toolkit are: the point primitive, the vector primitive, the plane primitive, and the graphical object primitive. Although the system does allow a high degree of direct manipulation to construct an interface, there remain several operations that have to be performed in an indirect manner, such as the definition of constraints between objects in a separate window with no visual feedback from the objects themselves.

Our goal when developing the 3D PGO editor was to provide a tool with the following properties:

- It has to provide accurate visual feedback of all the concepts used for the construction and editing of the visualization and interface;
- It has to provide bi-directional bindings between the objects' parameters and the simulation's parameters and state variables;
- It has to allow direct manipulation on the objects. Both when the visualization and interface are created and edited, as when the objects to steer the simulation are manipulated;
- It has to be simple and easy to use, yet remain effective and powerful.

3 3D Parametrized Geometric Objects

The 3D graphics editor uses Parametrized Geometric Objects (PGOs) for the visualization of the simulation results and the user interface. Properties of simple basic objects such as spheres, cylinders, and boxes, can be parametrized to values of variables in the Data Manager. By changing the value of a variable in the Data Manager, the objects parametrized to this variable change their visual appearance and, in the opposite direction, by manipulating an object the values of the variables to which the object is parametrized are changed in the Data Manager. Thus, there is a bi-directional symmetrical binding for input and output between the objects and the simulation's parameters.

The graphics editor has two modes: specification and application, or *edit* and *run*. In edit-mode, the user can create or edit geometric objects and parametrize properties of the objects to values of variables in the Data Manager. Hence, the researcher draws a specification of the interface. In run-mode, a two-way communication is established between the graphics tool and the simulation by binding these properties to variables. Data is retrieved and mapped onto the properties of the geometric objects. The researcher can enter text, pick and drag objects, which is translated into changes of the values of variables.

3.1 Point-Based PGOs

We considered three different approaches to define 3D parametrized geometric objects (see figure 2):

1. Based on definitions per object;
2. Based on local coordinate frames;
3. Point-based.

With the first approach, for each type of object a separate set of parameters is defined. Each object has a dialogue-box associated with it in which the actual values of those parameters can be defined. This approach has been used in Glyphmaker [10]. With the second approach, each object is assigned a local coordinate frame which is used to define the object's position, orientation, and size. With the third approach, the objects are defined by two or more *control-points* that determine the object's position, orientation and size. For instance, a sphere can be defined by the use of two points: one for the center of the sphere, defining the sphere's position, and one on the surface, defining the sphere's radius. A cylinder can be defined with three points: two for the center-axis of the cylinder, defining its position, orientation, and length, and one for its radius.

A point in 3D space has three degrees of freedom. Therefore, the total number of degrees of freedom for the set of points used to define an object will be a multiple of three. However, an object like a cylinder has only seven degrees of freedom. In such a case, not all the degrees of freedom of the points that define the object are used. For the cylinder one point is used only for a single parameter: the radius. Nevertheless, the point-based approach has a number of advantages over the other two:

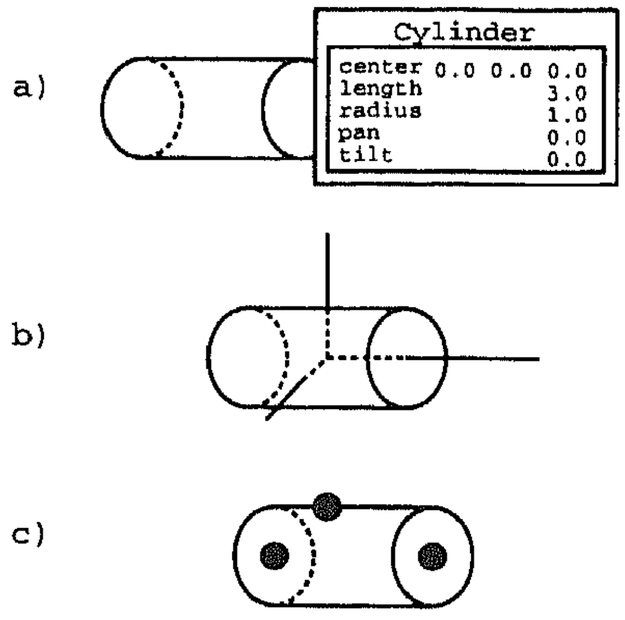


Figure 2: Different definitions for parametrized geometric objects: per object (a), frame based (b), and point based (c).

- It is uniform. All geometric objects can be defined in a similar manner;
- It is general applicable. Objects like polygons and polylines can easily be defined and parametrized;
- It is easy to use. No knowledge about (local) coordinate frames is required;
- It allows for easy interaction through direct manipulation. Object positions, orientations, and sizes can be changed by manipulation of the control-points;
- Several relations between different objects can be easily defined. By sharing control-points among objects, constraints like keeping different objects at the same position, or letting objects' surfaces touch can be enforced;
- It is flexible. Different object parametrizations can be achieved by changing the parametrizations of the control-points.

The point-based approach was also used in a previously developed 2D editor and with good results. For these reasons we have chosen the point-based approach to define the objects in the 3D PGO editor.

A set of standard geometric objects is provided by the PGO editor: polyline, fill-area, box, sphere, cylinder, cone, and text. The geometry of these objects is defined by their control-points. These points are independent of the geometric objects themselves, so that one point can be shared by various geometric objects. The objects can be displayed as solids or as wire-frames, and have four additional attributes: the hue, saturation, and value (intensity) of its color, and the linewidth used for the object or its wire-frame. These attributes can also be parametrized to values of variables in the Data

Manager and are presented to the user via a separate attribute window. In text objects references to values of variables in the Data Manager are replaced in run-mode by the value of the corresponding variable. Figure 3 shows some examples of the standard geometric objects with their control-points.

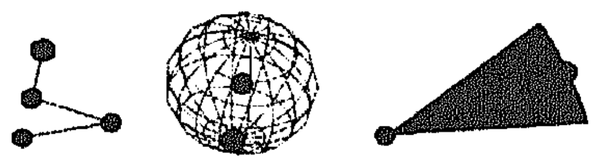


Figure 3: A polyline, sphere, and cone with their control-points.

3.2 Inter-Point Connections

The researcher can define relationships between points and thereby enforce constraints on the objects. Any point can have another point as a reference-point or parent-point. These relations are shown as grey lines with yellow arrows, and can be accomplished by simple dragging one point's 'connector' (a red cone) to another point. Cycles are not allowed, thus, the structure is a forest of trees, with points as nodes and leaves. These relations are used when points are moved. How this is done depends on the type of the point. Two types of points can be used:

- Hinge points (depicted as spheres). When a hinge point is moved, the same translation is applied to all its child-points. So, hinge points are suited for the translation of a set of points;
- Fixed points (depicted as diamonds). At a fixed point the angles between the connections that start or end at this point are constrained to be fixed. They can be used if a set of points is to be rotated around some other point (the grandparent-point).

Next these transformations are recursively applied to the children of the transformed points. Besides moving points, the user can also move objects. If an object is picked and moved, all the object's control-points are moved with it.

3.3 Point Parametrization

The position of the points can be parametrized to values of variables in the Data Manager by the use of Degrees of Freedom (DOFs). Each DOF has a minimum, a maximum, a current value, and possibly a Mapped Variable that is bound to the DOF. A DOF is presented to the user as a line, bounded by two discs. The line represents the range of allowed positions for the control-point. An arrow-head indicates the direction of the DOF. In edit-mode all aspects of the DOFs can be changed interactively via

dragging and text-editing, in run-mode only the current value can be changed. Since we are working in 3D, three DOFs per point can be used. Two types of DOFs are available for the parametrization of a point: Cartesian DOFs in *x*, *y*, *z*, or any combination of these, and spherical DOFS in *radius*, *azimuth*, *elevation*, or any combination of these, see figure 4. The reference frame for the Cartesian DOFs is the world coordinate system, i.e. the *x*, *y*, and *z* DOFs are aligned with the X, Y, and Z-axis of the world coordinate system. The reference for the sphere DOFs is provided by the parent and grandparent-point: the azimuth DOF lies in the plane perpendicular to the line defined by the parent and grandparent point, and the elevation DOF is perpendicular to this plane. If the point does not have a grandparent-point then the azimuth DOF is parallel, and the elevation DOF perpendicular to the XY-plane of the world coordinate frame. It is required that the point has a parent-point for the use of spherical DOFs.

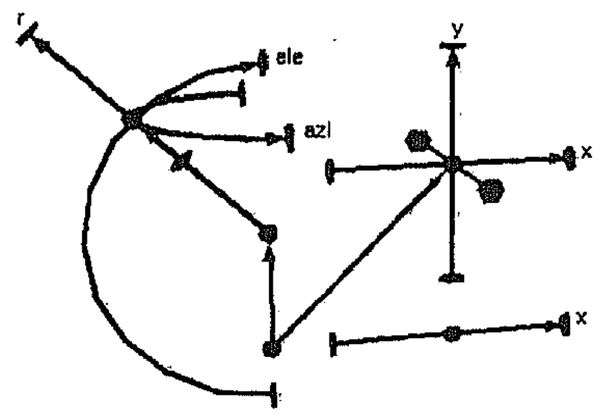


Figure 4: Visual representations of Degrees of Freedom for points.

With these options in combination with the relations between points a wide variety of coordinate frames can be defined: relative, absolute, Cartesian, spherical, cylindrical, hierarchical, although the concept of a coordinate frame itself is not provided. This is a typical example of the main principle that has guided us: provide a set of simple primitives and useful operations to combine them, so that the user can easily construct a wide range of higher order concepts.

3.4 Mapped Variables

The name that a researcher can specify for a DOF refers to a Mapped Variable (MVAR). The data of a MVAR are references to a Map and a Variable, and two on/off switches for input and output. With these switches the researcher can select if input information (from the researcher to the Data Manager) and output information (from the Data Manager to the researcher) can pass or not.

The Map contains a specification on how values must be mapped in the communication with the Data Manager. The current implementation is simple: only linear

mappings are supported, hence the specification of a minimum and maximum value suffices. Furthermore, the Map contains a specification of the format for the textual output. A Variable is a local copy of the information in the Data Manager. Here, some bookkeeping information, such as type, size, name and Data Manager id, and the current value(s) are stored. Several Variables can share the same Maps and one variable can have several associated mappings.

From all geometric objects multiple instances can be generated by mapping the DOFs of the control-points or the attributes of the object to arrays present in the Data Manager. The number of instances depends on the size of the arrays which may vary dynamically during run-mode. In addition, polylines and polygons can be expanded in-line within a single object. This means that instead of creating multiple instances of the object, the number of control-points defining this object is set according to the array size.

4 3D Interaction

The editing of the user-interface and the manipulation of the graphical objects and points requires a 3D interaction technique. We need to be able to position the control-points in 3D space, assign DOFs to these points, make connections between them, define geometric objects on them, and reposition the points and objects in both edit-mode and run-mode. We wanted a technique that

- does not require any special devices, only a traditional display, 2D mouse, and a keyboard;
- allows for direct manipulation on the points and objects in the 3D space;
- allows for precise positioning in the 3D space;
- is easy to learn and work with.

In order to achieve this, we adapted techniques from [14] and [9] (a 3D crosshair cursor), and [2] (Interactive Shadows).

4.1 3D Crosshair Cursor

In [14] a method is presented for geometric transformations of objects in 3D space through direct manipulation on a 3D crosshair. The transformations provided are translation, scaling, and rotation. Since in our system scaling and rotating of objects is achieved by repositioning the objects' control-points, we restricted the technique to only positioning and translating points and objects in the 3D space.

Whenever a positioning or translating task is to be performed, a 3D crosshair cursor appears in the 3D space. The user can then pick the crosshair with the 2D mouse cursor and drag it to a new position. The effect of the 2D mouse movement on the crosshair depends on the direction of the movement and the orientation of the projected

crosshair. For instance, if the 2D movement is along the projection of the y-axis of the crosshair, the crosshair will be moved along its y-axis. The selection of the axis along which the crosshair is to be moved is accomplished by matching the mouse cursor movement vector to the projections of all three axis of the crosshair. The axis with the best (directional) match is selected as the axis of translation. To avoid undesired movements along axes that are (almost) perpendicular to the screen, and thus have a projection that can hardly be seen, a threshold is used to allow only translations along those axes whose projections have a certain minimum length.

Additional features for (exact) positioning of the crosshair are a user defined grid-snap option, numerical feedback of the position of the crosshair as well as the possibility to numerically specify the crosshair's position, and the option to disable translations along one or more axes of the crosshair.

4.2 Shadow Editing

The virtual workspace in which the 3D visualization and interface are created is a 3D box with user-defined dimensions. The user can rotate the box, translate the viewpoint, and zoom in or out by moving the mouse over an additional box icon located in the projection window. This leaves the mouse to be used for manipulation on the objects in the workspace.

We adapted a technique from [2] to provide the user with an additional method to edit and manipulate the geometric objects and points. The user has the option to display the bounding planes of the workspace that face the user from the inside. On these planes orthogonal projections (*shadows*) of the objects in the workspace (the geometric objects, the control-points, the connections between the control-points, the 3D crosshair cursor etc.) can be displayed. Now the user cannot only interact with the 3D objects themselves, but also with their (2D) shadows.

For instance, if a point is to be repositioned, the user can pick a shadow of the point in one of the projection planes. Then the 3D crosshair appears at the position of the point in the workspace along with its projections at the positions of the shadows of the point to move. Now the movements of the mouse-cursor are mapped to the screen projections of the shadow of the crosshair and the appropriate axis of movement is selected.

These interactive shadows provide several benefits:

- They offer the user three additional representations of the 3D scene in one coherent display.
- It is possible to pick objects or points which are obstructed by other objects or points in the 3D scene. To simplify the editing of points, picking points precedes picking objects in the projection planes (all geometric objects' shadows are displayed in a grey color; point-shadows are superimposed in a darker color);
- The user can easily perform translations limited to a plane aligned with the world coordinate frame.

- It allows for the snap-on grid to be displayed. As there is no feasible technique to display a 3D grid, the projection planes can be used to display 2D projections of the grid;
- It serves as an additional depth cue.

4.3 Object Creation and Manipulation

The left of figure 5 depicts the specification of an arrow. Such an arrow could for instance be used to steer a field force in a simulation. Its length would then be parametrized to the strength of the force while its orientation would depict the direction of the force.

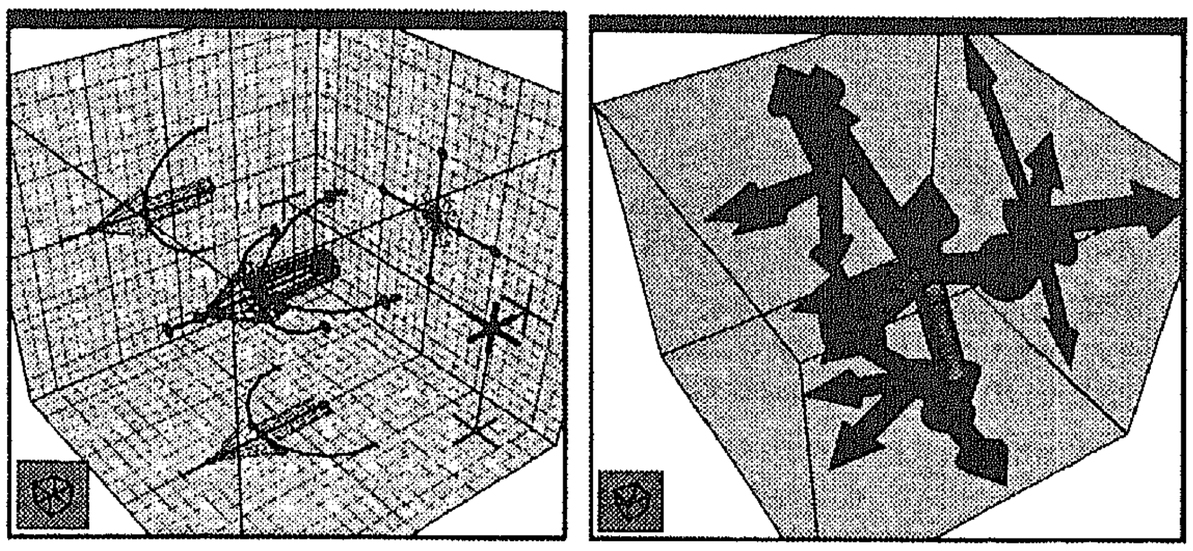


Figure 5: Arrow in edit-mode (left) and run-mode (right).

The arrow consists of two primitives: a cylinder and a cone. The cylinder is specified using three control-points: one for its base, one for its top, and one for its radius. The surface control-point is connected to the top control-point which in term is connected to the base-point. The cone is also specified by three control-points: one for the bottom, one for the tip, and one for the bottom radius. The cone's bottom-point is the same point as the cylinder's top-point, as to keep the cone on top of the cylinder. The cone's tip-point and radius-point are connected to its bottom-point (i.e. the cylinder's top-point). Now we can move the entire arrow in edit-mode by moving the cylinder or the cylinder's base-point.

The cylinder's top-point is given three DOFs: azimuth, elevation, and radius. These DOFs can be parametrized to variables in the data base, such that the cylinder points in the direction of the force and its size depicts the strength of the force. By making the cylinder's top-point a fixed point, we assure that the cylinder's diameter-point and the cone's top-point and diameter-point keep the correct orientation and distance in regard to the fixed point. This way the arrow will not change it's diameter and will point in the correct direction.

A composite object like an arrow or a slider is useful for many applications. We can save such objects as macro's, and reuse them in their original form or tailored to the particular application.

The right of figure 5 depicts six instances of the specified arrow in run-mode. The DOFs elevation, azimuth, and radius were mapped to arrays of length 6 in the Data Manager. The hue attribute of the cylinder was mapped to its length. Each arrow can be picked separately and adjusted to a new length and orientation, upon which the values in the arrays at the appropriate index are updated.

5 Applications

5.1 Path-planning

Figure 6 shows the interface to a path planning application developed by K. Trovato [13] and L. Dorst et al. [1][1]. The interface shows two representations of a car parking problem. One is the task space which visualizes the street, the car, and the two obstacles in between which the car is to be parked by the path planning program. The other is the configuration space (c-space). Here, the three parameters that describe the configuration of the car are visualized: two position parameters x and y, and the car's orientation ϕ. A hole in the representation indicates that the car can take that parameter configuration without interference with the two obstacles. The c-space is cyclic, and the user can examine the c-space by manipulating two boundary planes in its representation to select the region to be visualized. The car can be dragged to a new initial or goal po-

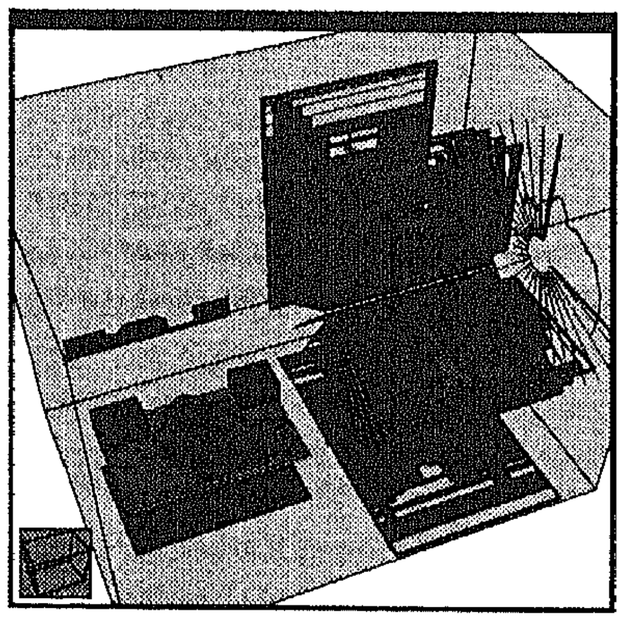


Figure 6: Interface to a path planning application.

[1] The path planning software is ©by Philips Laboratories, 1988. Philips has four patents pending relating to the vehicle planning methods and control.

sition by manipulating the car itself or its representation in the configuration space (a small sphere). Also, the two obstacles can be resized by direct manipulation. When the car is moved, by the user or the path planning program, the traveled path of the car is logged with the use of a general logging satellite. This log is visualized with wire frame projections of the car in the task space, and with a polyline in the configuration space.

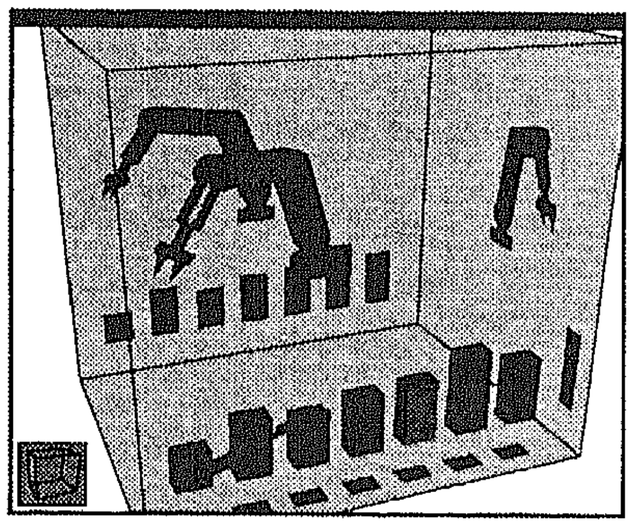


Figure 7: Robot arm.

Figure 7 depicts a robot arm. The robot arm comprises several different rotational and translational joints. Although this interface is not yet connected to a simulation, it illustrates the effectiveness of the 3D PGO editor in constructing compound objects with complex constrained relations between the different components. The user can control the robot arm by the use of the sliders, where each slider manipulates one joint, or by manipulation on the robot arm itself. We plan to develop similar robot arm interfaces in combination with path planning algorithms.

5.2 Cars Database

Figure 8 shows that the PGO editor and the Computational Steering Environment can be used for multi-dimensional visualization, and as a front-end to databases.

We used the PGO editor to visualize a table of 400 different types of cars, where for each car a number of attributes such as displacement, horsepower, and acceleration are given[2]. We used a small satellite that reads in the data and writes these to the Data Manager, and that can be used to make selections of the data. Each entry in the database is represented by a small car in the 3D space. The color of the cars depict their origin (blue: USA, green: Europe, and red: Japan) and the positions of the cars is currently parametrized to three properties: acceleration, displacement, and horsepower. The user

[2]The data set has been provided by the Committee on Statistical Graphics of the American Statistical Association for its Second (1983) Exposition of Statistical Graphics Technology.

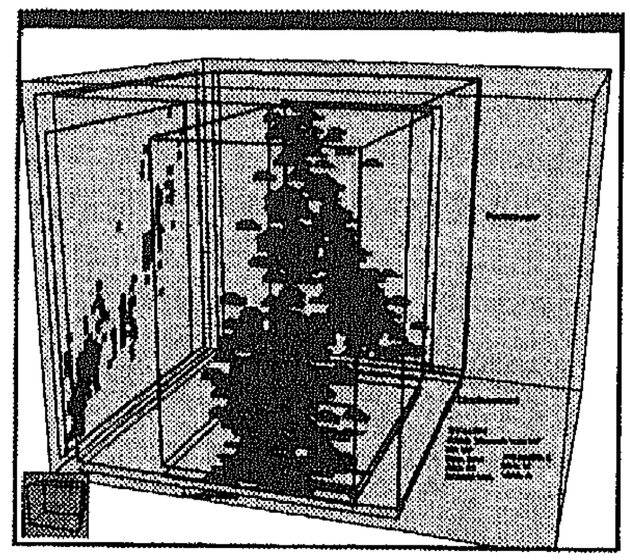


Figure 8: A front-end to a multi-dimensional database.

can select a region of interest in the 3D space by manipulating a bounding box, and he can select a car whose particular properties are then displayed.

5.3 Vector Field Visualization

We have also used our environment for vector field visualization. A small satellite was implemented that calculates the motion of a particle, advected by a vector field. A standard logging satellite was used to store the history of the calculated positions. Figure 9

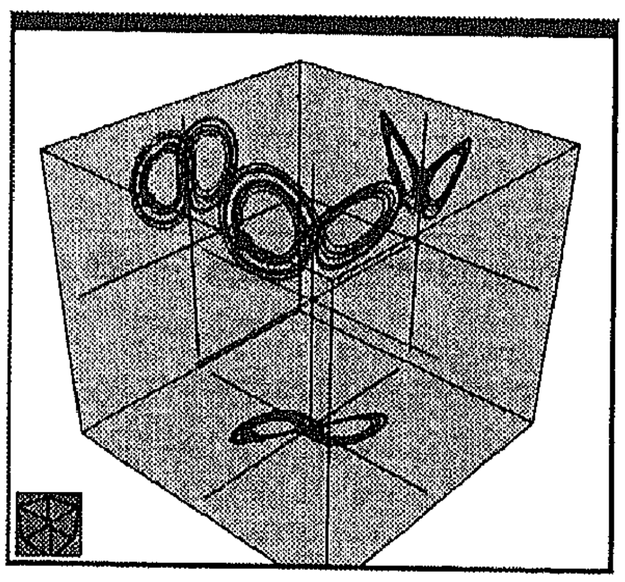


Figure 9: Particle tracing.

shows the well-known Lorenz attractor, the result when the Lorenz-equations [7] are used for the vector field. The user can drag the particle to different positions, and it is

very fascinating to observe how after a number of time-steps the particle cycles again around the attractors. The implementation of this user interface took less then half an hour.

5.4 Bouncing Balls

Figure 10 shows an interface that was constructed for the steering of a simulation of bouncing balls. Numerous steerable parameters are present in the simulation. In this

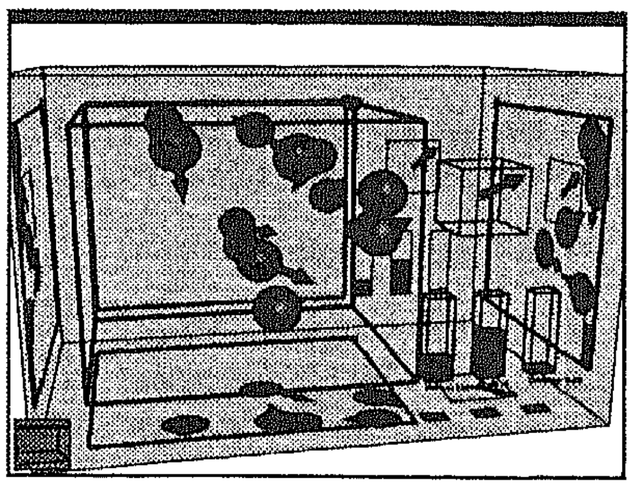


Figure 10: Simulation of bouncing balls.

particular interface, the user can steer the dampening property of the medium in which the balls move, the number of balls, and the radius of the balls by the use of sliders. The user can also steer a field force by manipulating the arrow in the small box. The direction of the arrow depicts the direction of the force, while its length represents the strength of the force. Furthermore, the user can pick and drag one of the balls to a new position. The balls are visualized with an additional arrow to depict their velocity vector.

6 Conclusion

We have shown how 3D parametrized geometric objects can be used in a tool for computational steering. The objects are used for both the visualization and the steering of the simulation. The user can easily create his own interface with the simulation by constructing glyphs and input/output widgets from the geometric primitives provided by the editor. Properties of these primitives can be bound to variables in the central Data Manager. This can be a bi-directional binding: from the Data Manager towards the editor (whenever the data in the Data Manager changes, the visual representation changes), and from the editor towards the Data Manager (if the visual representation is changed, the data in the Data Manager is changed).

The main difference between the 3D PGO editor presented in this paper and the Glyphmaker presented in [10] is that the Glyphmaker does not allow for computational steering; the bindings between the data and the glyphs are uni-directional. Other differences are the point-based object definitions in the PGO editor versus the more ad-hoc based approach in Glyphmaker, and the lack of direct manipulation techniques on the objects in Glyph Editor. Although the method of defining constraints as used in the toolkit presented in [11] allows for more complex and perhaps more powerful constraint relations between objects than the method used in the PGO editor, our method has some advantages over the other:

- It provides accurate visual feedback;
- The relations between points, the type of points, and the degrees of freedom of a point can all be changed by direct manipulation;
- It is very effective, yet remains simple and easy to use.

Superficially, PGOs have similarities with the primitives offered by IRIS Inventor [12]. However, they have a different scope: Inventor consists of tools and libraries that can be used to develop applications, whereas the system described here provides an integrated environment for end-users.

The PGO editor can easily be extended with other basic primitives. The current primitives are discrete objects. It is an interesting challenge if new primitives can be developed for continuous surface and volume data. Furthermore, the PGO editor is very suited to develop more complicated visualizations (such as glyphs) by combining simpler ones. In addition, special complex manipulators could be developed and evaluated for the control of complex data input configurations.

Acknowledgements

We would like to thank Karen Trovato of Philips Laboratories and Leo Dorst of the University of Amsterdam for providing the path planning software, Robert van Liere of CWI for his help on the implementation of the work described here, and Frans Groen of the University of Amsterdam for his valuable comments on the draft versions of this paper.

References

[1] L. Dorst, I. Mandhyan, and K. Trovato. The geometrical representation of path planning problems. *Robotics and Autonomous Systems*, 7:181–195, 1991.

[2] K.P. Herndon, R.C. Zeleznik, D.C. Robbins, D. Brookshire Conner, S.S. Snibbe, and A. van Dam. Interactive shadows. In *Proceedings UIST '92*, pages 1–6, November 1992.

[3] D.J. Jablonowski, J.D. Bruner, B. Bliss, and R.B. Haber. Vase: The visualization and application steering environment. In *Proceedings of Supercomputing '93*, pages 560–569, 1993.

[4] M. Jern and R.A. Earnshaw. Interactive real-time visualization systems using a virtual reality paradigm. In M. Göbel, H. Müller, and B. Urban, editors, *Visualization in Scientific Computing*, pages 174–189. Springer-Verlag Wien New York, 1995.

[5] G.D. Kerlick and E. Kirby. Towards interactive steering, visualization and animation of unsteady finite element simulations. In *Proceedings of the Visualization '93 Conference*, pages 374–377, 1993.

[6] U. Lang, R. Lang, and R. Rühle. Integration of visualization and scientific calculation in a software system. In *Proceedings of the Visualization 1991 Conference*, pages 268–274, October 1991.

[7] E. Lorenz. Deterministic nonperiodic flow. *Journal of the Atmospheric Sciences*, 20:130–141, 1963.

[8] R. Marshall, J. Kempf, S. Dyer, and C.-C. Yen. Visualization methods and simulation steering for a 3D turbulence model of Lake Erie. *Computer Graphics*, 24(2):89–97, 1990.

[9] G.M. Nielson and D.R. Olsen. Direct manipulation techniques for 3D objects using 2D locator devices. In *Proceedings of the 1986 Workshop on Interactive 3D Graphics*, pages 175–182, 1986.

[10] W. Ribarsky, E. Ayers, J. Eble, and S. Mukherjea. Glyphmaker: Creating customized visualizations of complex data. *IEEE Computer*, 27(4):57–64, July 1994.

[11] M.P. Stevens, R.C. Zeleznik, and J.F. Hughes. An architecture for an extensible 3D interface toolkit. In *Proceedings of the UIST '94 Conference*, pages 59–67, November 1994.

[12] P.S. Strauss and R. Carey. An object-oriented 3D graphics toolkit. *Computer Graphics*, 26(2):341–349, July 1992. Proceedings SIGGRAPH '92.

[13] K. Trovato. Autonomous vehicle maneuvering. In *Proceedings SPIE Volume 1613*, pages 68–79, November 1991.

[14] M.J.G.M. van Emmerik. A direct manipulation technique for specifying 3D object transformations with a 2D input device. *Computer Graphics Forum*, 9:355–361, 1990.

[15] J.J. van Wijk and R. van Liere. An environment for computational steering. Technical Report CS-R9448, Centre for Mathematics and Computer Science (CWI), 1994. Presented at the Dagstuhl Seminar on Scientific Visualization, 23-27 May 1994, Germany, proceedings to be published.

[16] C.-C.J. Yen, K.W. Bedford, J.L. Kempf, and R.E. Marshall. A three-dimensional/stereoscopic display and model control system for great lakes forecasts. In A. Kaufman, editor, *Proceedings of the Visualization '90 Conference*, pages 194–201, October 1990.

18. An Integrated Data Description Language for Coding Design Knowledge

B. Veth †

Centre for Mathematics and Computer Science
Kruislaan 413, 1098 SJ Amsterdam, THE NETHERLANDS

Abstract: *We present in a unifying framework the basic notions of IDDL (Integrated Data Description Language) to code design knowledge in the IIICAD system. IIICAD is an intelligent, integrated, and interactive computer-aided design environment we are currently developing at the Centre for Mathematics and Computer Science.*

Keywords: CAD, design theory, logic, theory of knowledge, theory of design objects, qualitative reasoning, object oriented programming, logic programming, knowledge engineering, software engineering, prototyping.

1. INTRODUCTION

In this paper we deal exclusively with the following issue: How to code design knowledge? We shall start in Section 2 with the IIICAD (Intelligent Integrated Interactive CAD) concepts and present our methodology to develop a useful representation language — i.e. a theoretical approach. Accordingly, we first concentrate on the theory of CAD and then derive requirements and specifications for IDDL. A full account of the implementation details will be left to an upcoming paper although we touch on this subject briefly.

The theory of CAD consists of three parts: theory of design, theory of knowledge, and theory of design objects. In Section 3.1 we introduce a design process model which is derived from a design theory, and in Section 3.2 we show its logical notation. In Section 4 we deal with the theory of knowledge. In Section 5.1 we describe the theory of machine design as an example of the theory of design objects. Designing is a process where we materialize our imagination. Any design process, therefore, cannot escape

† Group Bart Veth consists of (in alphabetical order): Varol Akman, Peter Bernus, Paul ten Hagen, Jan Rogier, Tetsuo Tomiyama, and Paul Veerkamp.

from the real world restrictions. In Section 5.2 we illustrate naive physics which treats this aspect. In Section 6 we count the general requirements for IDDL from CAD and software engineering viewpoints. Section 7 closes the paper by citing design policies for IDDL and showing our prototype implementation with an example of bridge design. A word about presentation: throughout the paper we prefix with **DM** the so-called design maxims (Yeomans, Choudry, and ten Hagen 1985) which will be collected in Section 7 and converted into specifications for IDDL.

2. OVERVIEW OF IIICAD

2.1. The Concept of IIICAD

CAD systems are vital elements of almost every facet of the technology but it is also admitted that they are plagued by inflexibility. It is not unjust to claim that the majority of the existing systems are but sophisticated workbenches for engineering drawing. As the application domain becomes serious, designing becomes unmanageable with only this type of support. Since design is essentially an intellectual activity, we need, not surprisingly, more *intelligence* in a system — hence the first I of IIICAD.

Borrowing an analogy from (Bobrow, Mittal, and Stefik 1986), until now CAD systems were built using the *low road* and *middle road* approaches. The low road approach involves *ad hoc* programming (mostly in prehistoric languages like Fortran) and is biased towards geometric information. Middle road systems are more interesting in that they are aware of the fact that they have to incorporate intelligence. They focus on a well-defined domain and collect specialized knowledge coded as say, *if-then* rules. In other words, they become expert systems (e.g. PRIDE (Bobrow, Mittal, and Stefik 1986)). An annoying problem with expert systems is that genuinely expert performance can only rest on knowledge of a model in which an underlying mechanism *understands* what is going on (Kuipers 1986).

Finally, one distinguishes the *high road* systems which IIICAD is aiming at. High road systems are deep systems (as opposed to low and middle road systems which are shallow) in that their knowledge represents the principles and theories underlying the subject "design." In the case of IIICAD, the fundamentals of General Design Theory which is based on axiomatic set theory can be found in (Tomiyama and Yoshikawa 1987).

We do not deny the fact that there are several domain-specific sides to design. For instance, VLSI design is two-dimensional (although this is changing) while mechanical design is inherently three-dimensional. IIICAD incorporates similarities in design, leaving the application-dependent issues to further consideration as side requirements and using intelligence based on a clean and robust design theory. Thus here we are not working on yet another geometric modeler or expert system.

The other two I's of IIICAD correspond to *integration* and *interactivity*. Design systems should support integration because human designers have a unified view of design objects. Interaction requires almost no validation. Good design systems cannot be obtained without using the best man-machine communication techniques.

To summarize:

- In IIICAD we pursue a top-down theoretical approach incorporating more intelligence than expert systems, more integration than geometric databases, and high-level interaction using advanced computer graphics.
- We want IIICAD to be a system based on expandable ideas and a framework where designers can exercise their faculties at large. We believe that the essential thing in a designer is that he builds us his world and IIICAD must give him the freedom to do so.

2.2. Elements of IIICAD

The Supervisor (SPV) is at the core of IIICAD and controls all the information flow. It adds intelligence to the system by comparing user actions with *scenarios* which describe standard design procedures, and by performing error handling when necessary. Since SPV is the central authority for control the following becomes relevant.

DM 1. *IDDL should be able to describe status and control information of the system with origin, destination, and time stamp of the control information.*

While SPV corrects the obvious user errors, it does not have the initiative for the design process itself because IIICAD is envisaged to be a designer's apprentice, not an automatic design environment.

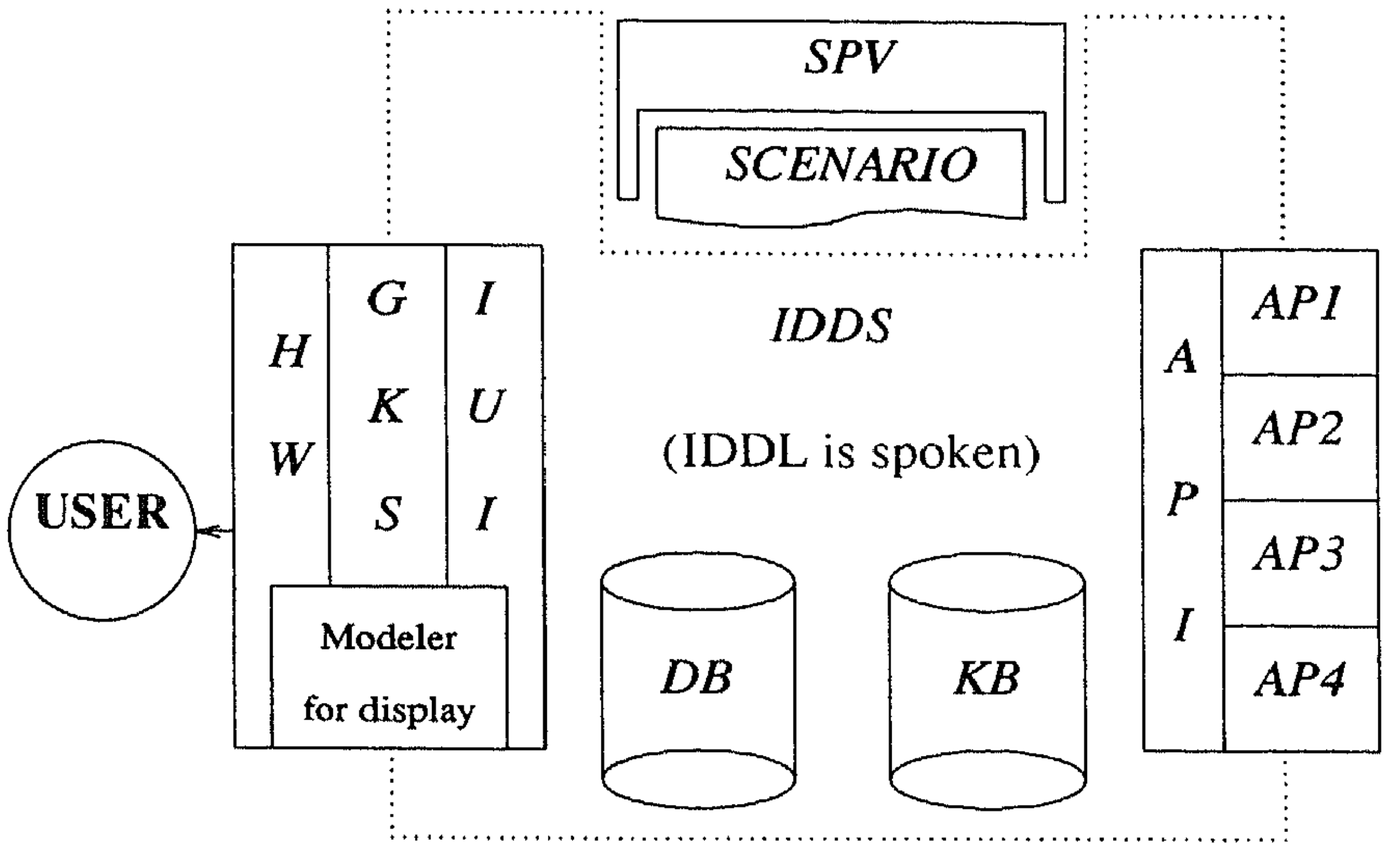


Fig. 18.1. IIICAD architecture

The Integrated Data Description Schema (IDDS) regiments the data and knowledge bases relieving the user from the burden of specifying where and how to

store/retrieve data. IDDS has a language called Integrated Data Description Language (IDDL) spoken by all system elements. IDDL is the means to code the design knowledge and the design object to guarantee integrated descriptions system-wide. Like most modern programming languages, IDDL differentiates between what is commonly known as the *external* and the *internal* contents. The former is essentially nonmathematical information such as input-output behavior and diagnostics. The latter consists of mathematical operations which do the job. More on IDDL will be said in Section 6 and Section 7 which show how IDDL codifies design knowledge. Here it should suffice to remark that internally IDDL will be based on logic and accordingly knowledge engineering is the key factor in building the IIICAD system. IDDL is an essential step in developing IIICAD.

In addition to the above principal elements, IIICAD has a high-level interface called Intelligent User Interface (IUI) which is also driven by scenarios written in IDDL, and the Application Interface (API) which secures the mappings between the central model descriptions about the design object and individual models used by application programs such as geometric modelers, finite element analyzers, etc. Figure 18.1 shows the preceding elements in block diagram level.

2.3. Software Engineering Viewpoint

Maintainable software systems should be modular both "in the small" to allow alteration of minor components in specific applications and "in the large" to allow changes in major components based on say, the advances in technology. They should be designed for evolution for long time horizons. They should be sturdy and open-ended (Wegner 1984). In this regard, software engineering will always be a leading concern in developing intelligent CAD software such as IIICAD because even the conventional CAD systems are large and complicated. In other words, knowledge engineering is more than software engineering but probably not much more (Bobrow, Mittal, and Stefik 1986).

DM 2. *IDDL, as a language to construct a knowledge base, should support easy maintenance.*

In the development of intelligent CAD systems the underlying strategy is "Plan to throw one away. You will anyhow." (Brooks 1975). Emerging trends of software engineering such as exploratory programming and rapid prototyping are thus crucial. These methods are somewhat more permissive than the more rigid method of formal specification in that they follow the idea of iterative enhancement and consequently, an evolutionary life-cycle approach (Wegner 1984). One starts with a skeletal implementation (rapid prototype) and adds new parts until the system is reasonably completed. This incremental approach is fruitful when the set of tasks and the end result are incompletely defined. Also one is more interested in seeing a glimpse of a future system built as a prototype in order to assess its strengths and weaknesses globally. Exploratory programming using powerful workstations and modern languages (e.g. Smalltalk-80[1]) makes this process very effective (Ramamoorthy, Shekhar, and Garg 1987).

DM 3. *IDDL must support incremental programming.*

[1] Smalltalk-80 is a trademark of Xerox Corporation.

3. DESIGN THEORY

3.1. Modeling of Design Processes

Design theory (Yoshikawa 1981) provides a strong basis for formalizing design processes and design knowledge. For this purpose, we use General Design Theory (Tomiyama and Yoshikawa 1987; Yoshikawa 1981) which is based on axiomatic set theory and models designing as a mapping from the function space where the specifications are described in terms of functions, onto the attribute space where the design solutions are described in terms of attributes.

There are many interesting results derived from General Design Theory; we emphasize in Section 3.2 the possibility of a logical formalization of design processes. Figure 18.2 shows a design process model derived from General Design Theory. The basic idea is as follows (Tomiyama and ten Hagen 1987a; Tomiyama and ten Hagen 1987b):

- A designer, given the specifications, may try to select a candidate and refine it in a stepwise manner, rather than trying to get the solution directly from the specifications.
- Therefore, a design process can be regarded as an evolutionary process of such intermediate descriptions of the design objects rather than just a mapping. The collection of these intermediate descriptions can be used as the central model about the design solution and we call it a *metamodel*.
- The designer will evaluate the candidate to see whether it satisfies the specifications or not. To do so, he derives various kinds of models of the design object from one central model (i.e. the metamodel).

Our discussion leads to the following design maxims:

DM 4. *IDDL should be able to describe not only design objects but also design processes.*

DM 5. *IDDL should be able to describe metamodels and models (for evaluation) derived from the metamodel.*

DM 6. *IDDL should be able to describe the stepwise nature of the design process.*

DM 7. *IDDL should be able to describe knowledge to detail the metamodel, to check its feasibility, and to control the detailing process.*

DM 8. *IDDL should be able to describe knowledge to derive models for evaluation from the metamodel and knowledge to evaluate models.*

DM 9. *IDDL should allow multiple views of a design object, which are possibly independent but still correlated.*

To illustrate a design process, we need to recognize three major components: *entities, attributes* of entities, and *relationships* among entities. A design process is thus a collection of small steps to obtain complete information about these three. We can also observe components which do not change during design. For instance, when we design a board we simply use a VLSI chip as a building block that cannot be changed. Let us call such objects *invariants* in a design process. Descriptions about the board at this level are, oppositely, dynamically changed during the design process. Let us call

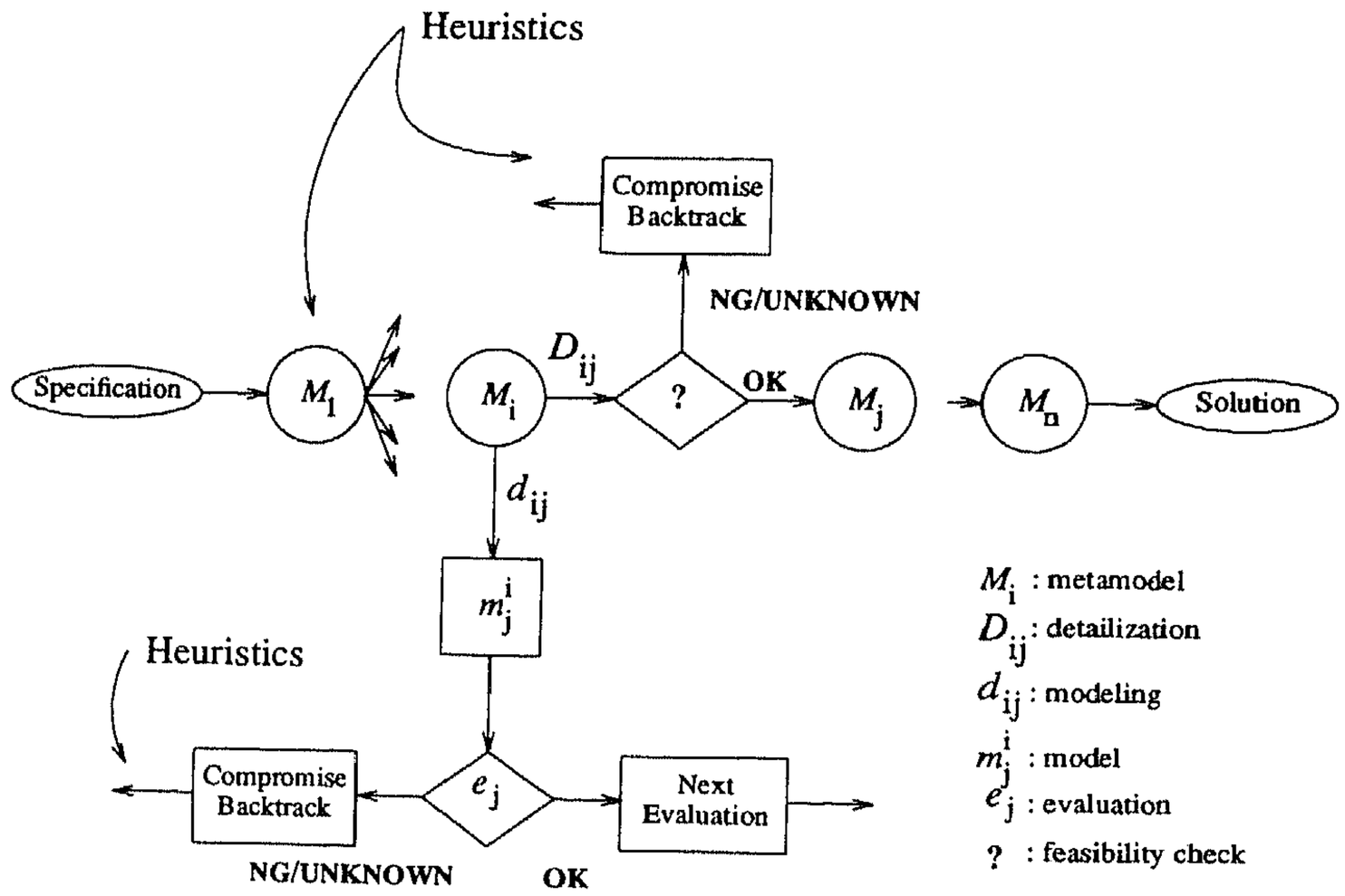


Fig. 18.2. Design process model due to General Design Theory

them *variants* in a design process. Clearly, the concept of e.g. a point should not change while designing a geometric entity; however attributes such as coordinates of a point may be frequently changed. Let us call such changeable concepts associated with variants *covariants.*

DM 10. *IDDL should be able to describe invariants, variants, and covariants in design processes.*

Another important thing about the design process model is that eventually we need to check physical constraints (i.e. feasibility check). This implies further that we need to distinguish success and failure, known and unknown, necessity and possibility, etc.

DM 11. *IDDL should be able to describe positive and negative information, known and unknown, and modalities such as necessity and possibility.*

3.2. Design Processes: Logical Formalization

Let us assume predicate logic as the basis of our discussion. To control the stepwise refinement of the design process there is a need to express unknown, uncertain (default), and temporal information about the design object. For this reason we equip our logic language with three-valued logic, modal logic, inheritance, and situational calculus.

Considering the metamodel evolution model in Fig. 18.2, the system starts from the specification S of the design object and it continues the design process until the goal G is reached:

$$S \rightarrow M^0 \rightarrow \cdots \rightarrow M^i \rightarrow M^{i+1} \rightarrow \cdots \rightarrow M^n \rightarrow G$$

We define $\{q^i\}$ as the set of propositions at the metamodel state M^i. In other words $\{q^i\}$ is the current state of knowledge about the design object. There are two possibilities: either the current state of knowledge is complete and consistent or there is some incompleteness or inconsistency. In the first case the goal is reached and we finished the design process. In the latter case we need to proceed to a next metamodel in order to solve the incompleteness or inconsistency.

DM 12. *IDDL should let the inconsistency of a certain metamodel be represented, but this inconsistency needs to be resolved when transferring to a next metamodel.*

We need language constructs to evaluate a metamodel and to derive new properties or to update uncertain or unknown properties in order to get more detailed knowledge about the design object. The decisive point is how to proceed from M^i to M^{i+1}; i.e. given $\{q^i\}$ how do we find $\{q^{i+1}\}$? We shall adopt the following strategy:

- From $\{q^i\}$ we can derive p, so the next state $M^{i+1} = \{q^i\} \cup p$. This means that the knowledge base is extended by asserting property p. In practice p might come from design procedures, default assumptions, results of engineering analyses, and so on. Before the acceptance of state M^{i+1} the consistency of the new metamodel has to be checked. For this purpose we can use the appropriate set of logical inference rules.
- We use the modal operator $\Box$ to express default values. Thus $\Box p$ means it is possible, but probably not the case that p; $\blacksquare p$ states it is necessary that p. Note that $\Box p$ is equivalent to $\neg \blacksquare \neg p$ (McDermott 1982).

DM 13. *IDDL should incorporate modal logic.*

- If we have $\Box q_j$, can derive $\neg q_j$, and it is not based upon default properties then we assume $\neg q_j$. In other words $M^{i+1} = \{q^i\} - \{\Box q_j\} \cup \{\neg q_j\}$. To see this, imagine that the designer wants to design a bridge and the system needs the length which is unknown. In this case the system knows that bridges normally have a length and it can conclude by an inheritance mechanism that the length of the bridge is L. This will be asserted to the knowledge base as $\Box$ *equal*(*length*(*bridge*), L). If in a next state of the design process the real length RL of the bridge is determined, $\Box$ *equal*(*length*(*bridge*), L) can be changed into $\blacksquare$ *equal*(*length*(*bridge*), RL).
- The Skolem constant ω is used to denote unknown values; so if we have $q_j = p(\omega)\ \Delta \cdots$ and we can derive q_j' without unknown values, then we have the next metamodel $M^{i+1} = \{q^i\} - \{q_j\} \cup \{q_j'\}$. Referring to the previous example about the length of the bridge, the system can assume an unknown value instead of a default value in *equal*(*length*(*bridge*), ω). When we want to express the fact that a certain property is unknown, we use $\perp p$.

DM 14. *IDDL should have both the Skolem constant and the unknown operator.*

- If a severe inconsistency is encountered which cannot be resolved by the system in terms of deriving more knowledge or stepping back to a previous metamodel, apparently an inconsistency in the specification provided by the designer has been found. The designer should be notified with this inconsistency together with exact transactions so that he can fix it and restart the design.

DM 15. *IDDL should have facilities for error handling, when it encounters inconsistent or incomplete states, with the help of the designer.*

To add a certain proposition an assertion operator *assert*($p(x)$) is needed. To modify propositions we need a *change*($p(x)$) operator. After these operations the knowledge base must still be consistent (cf. **DM** 12).

DM 16. *To control the behavior of the system IDDL should have metaknowledge that chooses which rule to apply at a certain time.*

In other words, IDDL must be able to describe a design process so that during designing we can "design" designing procedures using metaknowledge.

4. THEORY OF KNOWLEDGE

Theory of knowledge is necessary especially to put our knowledge into a particular framework and to utilize it in the IIICAD architecture. The following discussion is based on our result (Tomiyama and ten Hagen 1987c) and its most significant contribution is the distinction between two opposing knowledge representation methods, i.e. *extensional* vs. *intensional* descriptions.

In order to discuss design, we need to describe entities, their properties, and relationships among entities as mentioned in Section 3.1. In an extensional description method, the fact that an entity e has property p is described by $p(e)$ and the fact that entities e_1 and e_2 are in a relationship r is described by $r(e_1, e_2)$. In an intensional description method these two facts can be represented by $e(p)$ and *relation*(e_1, r, e_2), respectively. The extensional descriptions do not assume any preconceptions while the intensional descriptions assume preconceptions such as that e's property is limited to one particular p. This means that an intensional description is equivalent to an extensional description with some assumptions, such as the number of arguments, the order of arguments, the type of arguments, etc.

These two description methods are basically equivalent except for assumptions. Since an intensional description assumes something predefined, when it has to be changed this results in changing those predefined (and perhaps implicit) conditions. For instance, a mechanical part, say a shaft, might be represented by

$$shaft(diameter,\ length,\ bearing_1,\ bearing_2).$$

If we now want to add new attributes, such as transferring power, this results in a redefinition of this *shaft* predicate. On the other hand, an extensional description might be the set of the following facts:

$$shaft(s),\ equal(diameter(s),\ D),\ equal(length(s),\ L),$$
$$supported-by(s,\ b_1),\ supported-by(s,\ b_2),\ bearing(b_1),\ bearing(b_2).$$

In this example, an extensional description does not assume anything, e.g. *s* is just a name.

DM 17. *IDDL has two kinds of names: system names are internal and should be unique whereas user names are external and modifiable.*

In an extensional description we need to write numerous (and often very obvious) descriptions. However, modifying such a representation is just adding or deleting facts (thus incremental, cf. **DM** 3). On the other hand, an intensional description assumes a predefined scheme which might be difficult to change but shows high performance. For instance, it is easily predicted that the computation to determine two bearings supporting a shaft is reduced to an address calculation.

It is well-known that CAD applications request a flexible data description scheme which is easy to modify (Lorie 1982). From this point of view, the extensional description method is important in IIICAD because of the incremental nature of design processes. Independent (but still correlated) multiple views of design object demand independent small partitionings in the database. This is easily achieved by an extensional description method because we only have to pick up relevant facts. In an intensional description method this requires to create a totally new scheme.

As discussed in Section 3, there are variants which change during the design process. The extensional description can be used to describe these variants. To this end, we know that the logic programming paradigm (Kowalski 1978) is very useful to implement such description methods. On the other hand, there are invariants we use as building blocks for designing. They do not change their structural properties although values of their attributes might change. Invariants, therefore, can be represented in an intensional description method and their properties (or attributes) can be represented as covariants. Having intensional descriptions may also contribute to improving the performance.

DM 18. *IDDL should have both an extensional description method and an intensional one.*

DM 19. *Invariants in IDDL will be represented as objects, variants will be constructed on the predicate level, and covariants will be represented by functions.*

5. THEORY OF DESIGN OBJECTS

5.1. Theory of Machine Design

Design is regarded as a mapping from the function space onto the attribute space. This requires IDDL to have both attributive and functional representations. There are several issues in representing the attributive information (Tomiyama and Yoshikawa 1985; Tomiyama and ten Hagen 1987a). First, an attribute does not necessarily have a value. In the design process, it often happens that an attribute is only known to exist and its value is not yet decided. Hence IDDL should be based on three-valued logic including *unknown* (cf. **DM** 14). Second, attributive information refers to the structure of an entity. The structure of an entity might be characterized by existence of

substructures and relationships among substructures. Information on the number of substructures is also needed.

DM 20. *IDDL should make a distinction between the facts that an entity has an attribute and that an attribute has a value.*

DM 21. *IDDL should be able to represent part-assembly relationships and relationships among parts in order to represent structures.*

DM 22. *IDDL should be able to deal with cardinalities (i.e., number of elements in a set).*

On the other hand, the representation of functions is a rather difficult issue. Unfortunately, it is not yet known in which language we can describe functions of e.g. machines. There is, however, a hope that functions can be represented in terms of physical phenomena that the machine exhibits (Tomiyama and Yoshikawa 1987). From this point of view, the representation of functions can be reduced to the representation of physical phenomena and qualitative reasoning (cf. Section 5.2).

At one time in the metamodel evolution model, the designer will focus at a particular part of the design process or the design object. When this focusing is taking place, the information about the rest of the design process or the design object should not be accessible and stay unaffected. (This is the principle of abstract data type languages.) In order to make focusing more effective, we must create a small world which represents it. For example, we must be able to control applicable predicates to particular classes of entities, although this inevitably asks for higher order predicate logic.

DM 23. *IDDL should have a focusing control mechanism which is able to create a small world where it is clearly defined what kind of information is accessible.*

We must also be able to see a particular part of the design object. When we are considering a particular object, we must be able to see its inner structure represented by variants. On the other hand, we may use that object as a building block when we are working on another object. This requests transition between different abstraction levels and the same information (e.g. an object at some level) must be seen differently (e.g. as a collection of predicates) (Fig. 18.3).

DM 24. *IDDL should be able to describe hierarchical enclosing controls.*

5.2. Naive Physics, Qualitative Reasoning, and Design

Naive physics observes that people are generally very good at functioning in the physical world and tries to develop a formal framework to serve as a basis to export this human capability to computers (Hayes 1985). As such, it constitutes a major part of what is known as *commonsense reasoning* in AI. Naive physics concepts are needed in design because in many cases design objects will have a physical existence and accordingly obey natural laws. If we want to create designs corresponding to physically realizable (read manufacturable) design objects then we will have to refer to naive physics primitives such as solids, space, motion, etc. Furthermore, if we want to reason about a design object in its destined environment (think of a pressure regulator to be installed in a nuclear reactor) we will need naive physics notions such as envisioning, simulation, diagnostics, etc.

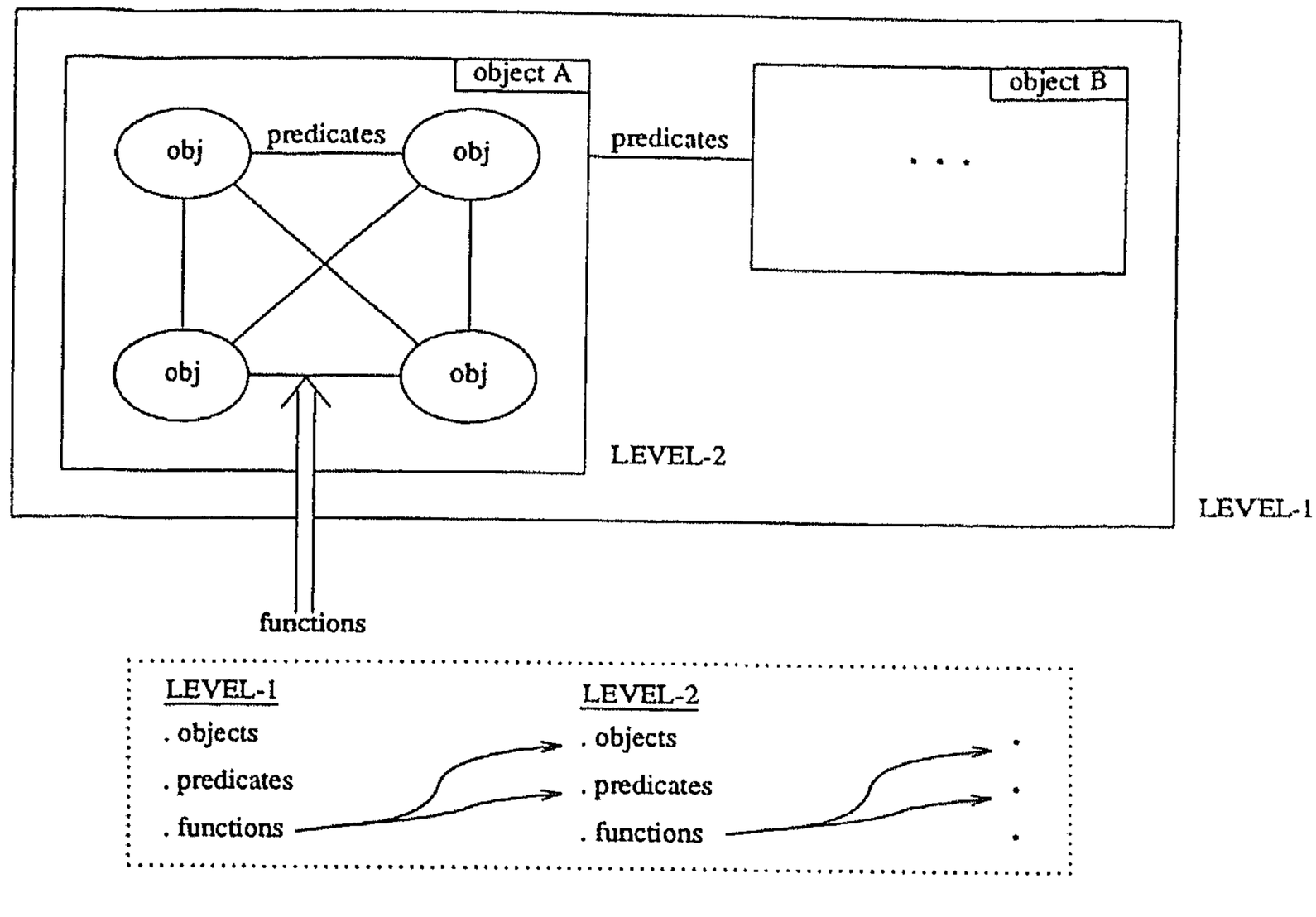


Fig. 18.3. Hierarchical enclosing control

A basic commonsense notion is *causation.* We seem to have no difficulty in grasping the relationship between two events (or states of affairs) such that the first brings about the second. Basic notions of modal (e.g. temporal) logic can be used to talk about causality. An intricacy is brought about by *teleological explanations* — i.e. certain phenomena seem to be best explained by intentions or purposes rather than by means of prior causes. This closely resembles to our way of looking at a design object from different viewpoints.

Classical physics seems not too useful in formalizing naive physics because it describes everything in exact terms. Starting with the introductory level textbooks, physics laws are based on the presupposition that the readers have a shared *prephysics* knowledge (de Kleer 1975). In fact, mathematical formalizations of physics, unless aided by verbal explanations, occasionally hide causality. What makes then a good formalization of naive physics? Both mathematical and computational criteria are important. Some guidelines may be given at this point:

- *Categorical angle:* Study objects through their universal properties which characterize them, rather than through their anatomical properties. See intriguing similarities and exploit abstractness to arrive at theories of utmost generality unattainable in other ways.
- *Mechanistic outlook and deductionism:* Regard physical situations as machines comprising individual components each of which contributing to the overall behavior of the machine. This means that the behavior of a physical structure will be completely accounted for by the way its physical constituents act. To a first order approximation, three kinds of constituents are enough (de Kleer and Brown 1984): materials (such as air, water), components (such as containers, wheels), and channels (such as electric cables, conduits).
- *No-function-in-structure:* This follows from the preceding guideline. Briefly, one has a catalog of components and these have associated laws which do not make assumptions about how they are employed in a certain context (de Kleer and Brown 1984).
- *Confluences:* These are better known as qualitative differential equations (Forbus 1984; Kuipers 1986). One first reduces continuous real-valued variables to discrete-valued variables taking only a small number of values, say +, −, and 0. This process maps differential equations to confluences. Normally, a single confluence will not be able to characterize the behavior of a component over its entire operation region.
- *Envisionment, simulation, and diagnosis:* In the former one starts with a structural description and determines all possible sequences of behavior. In simulation, one starts again with a structural description but this time he is given some initial conditions to determine a probable course of future behavior. In diagnosis, one starts with some specified behavior expected from a system and tries to see why the system is misbehaving.
- *Topological scene description:* This suggests that one may temporarily ignore the exact coordinates in some geometric situation and model it using topological notions like homotopies, isolation, etc.
- *Frame problem:* This is well-known. When some action takes place in a situational calculus-like representation, how does one tell what facts change and what facts stay unaffected? The answer is, one has to write explicit axioms that state what changes and what remains the same. One may avoid this problem at least partially by adopting *histories* (Hayes 1985) — descriptions extended through time but always spatially delimited (in contrast to situational calculus situations which are instants spatially unbounded). This reflects a choice to ban action at a distance (Forbus 1984).
- *Modal logic:* We find it handy to use logic with modes of truth, viz. modal logic with necessity and possibility. As explained earlier, here we have not only affirmations such as that proposition p is true, but also stronger ones such as that p is necessary, and weaker ones such as that p is possible. Modal logic is useful to code naive physics. Consider examples such as

$$below(obj, surface) \wedge not-glued(obj, surface) \supset \blacksquare fall(obj)$$

and

$$at-top(obj, inclined-surface) \wedge above(obj, inclined-surface) \supset \Box slide(obj)$$

where the latter possibility being dictated by our incomplete knowledge about e.g. friction.

According to these guidelines, **DMs** 9, 13, and 21 are relevant. In addition:

DM 25. *IDDL should be able to carry out simple algebraic manipulations — this is necessitated by e.g. qualitative reasoning with confluences.*

6. REQUIREMENTS TO IDDL AS A KERNEL LANGUAGE OF CAD

6.1. CAD Perspective

CAD is no longer considered as a tool for speeding up the creation of exact product definitions in the form of text and drawings. Because CAD needs a coordinated flow of information between system and user, the functional view of system design should consider the totality of design, not only those functions which will be carried out by a computer. It needs a language to represent the flow of design (cf. **DM** 1). The number of design rules which exhaust a realistic area within electrical, mechanical or civil engineering lies around tens of thousands. The amount of attributive data to be handled by the actual management of the design process could be around hundreds of megabytes, not mentioning the attributive data in the background databases of a design office.

DM 26. *IDDL should have a mechanism for structuring knowledge.*

DM 27. *As with design knowledge, design object representation calls for an encapsulation and structuring mechanism for it to be representable in reasonable form.*

Design produces intermediate results which are incomplete and even inconsistent during time spanning series of transactions. The design object's attributive representation must allow assumptions to be used for the evolution of the design object (cf. Section 3). Result of nonmonotonic reasoning must be checked for feasibility and consistency.

DM 28. *IDDL, using nonmonotonicity, should be able to retract assumed but later on unconsidered propositions.*

DM 29. *IDDL should be able to check consistency and completeness.*

From the viewpoint of interactive design, it is desirable for the human designer to be able to "mark" intermediate design stages, and later go back to them for examining or resuming from there.

DM 30. *The stages of design evolution must be representable on the level of IDDL.*

6.2. Software Perspective

From a software engineering point of view two types of requirements influence our language design. One is that the system to be built using IDDL will after all be a software system with high complexity. Therefore the language design must reflect considerations for managing complexity in software design (cf. Section 2). Especially, due to our inability of separating specification from experimentation, we would like to have an environment where the two can be done in parallel.

Excellent opportunities are found in object oriented style of programming (Stefik and Bobrow 1986). Object oriented programming delivers extensibility, flexible modifications of code, and reusability. Important issues in object oriented programming are data encapsulation and information hiding. Thus an object can be regarded as an independent program which knows everything about itself. Reuse of software is helped by the class inheritance mechanism. Other flexibilities of languages like Smalltalk-80 (Goldberg and Robson 1983) and Loops (Stefik, Bobrow, and Kahn 1986) are incremental compilation and dynamic binding. It is an additional benefit of object oriented programming systems that they offer rich system building tools and good user interfaces. Therefore object oriented programming is a good choice for creating IIICAD. On the other hand, logic programming is powerful for problem solving since it reflects the reasoning process most naturally and directly.

DM 31. *IDDL uses the logic programming paradigm to express the design process for manipulating design objects, whereas the object oriented programming paradigm is used to express design objects.*

DM 32. *In IDDL invariants, variants, and covariants will respectively be represented by objects, their internal and external relationships, and behavior of objects.*

The second aspect of language design is how representational tasks can be unified. A typical epistemological view (Brachman 1979) includes class – subclass, part – subpart, class – member-of-class, prototype-of-class – member-of-class, and functional abstraction.

DM 33. *IDDL should have mechanisms to represent inheritance.*

7. IDDL SPECIFICATIONS

7.1. From Design Policies to Specifications

In the preceding sections we have counted 33 design maxims. The derivation of IDDL specifications took place as follows. First we classified them into several categories so that we could see the relationships among them. We extracted a "keyword" from each of those categories, for example *encapsulation* was arrived at after considering **DMs** 10, 18, 19, 27, and 32; for *modalities* we considered **DMs** 13 and 28.

We then derived features considered to be essential for IDDL from those keywords. For instance, the feature *no automatic backtracking* was derived from the keyword *metaknowledge for control*. This is due to the expectation that if an automatic backtracking mechanism and metaknowledge for control are installed together then the control would become unmanageable.

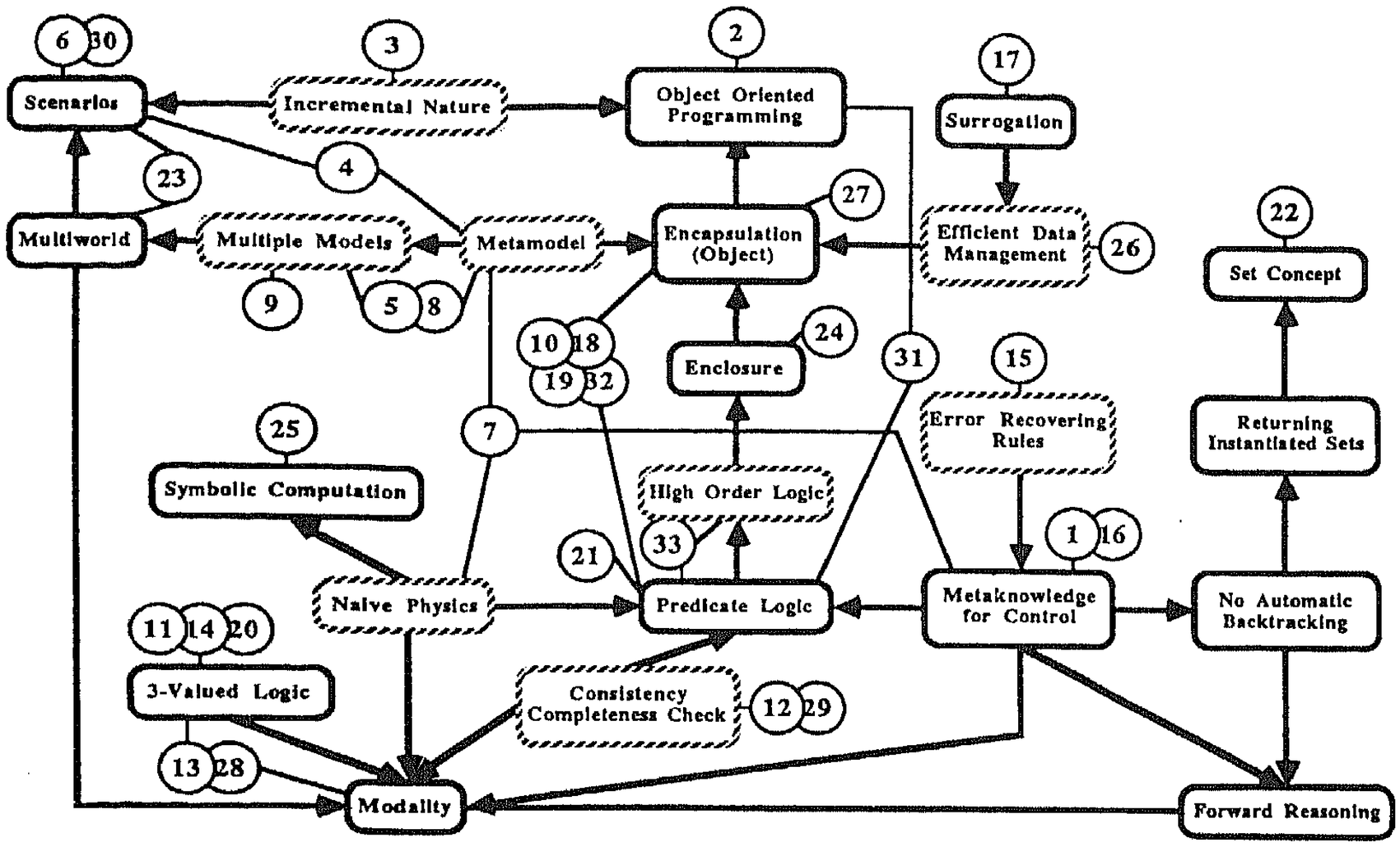


Fig. 18.4. Classification of IDDL design maxims

Figure 18.4 shows the relationships among design maxims, keywords, and features. In this figure, the small circles correspond to **DMs,** dotted boxes are keywords, and the solid boxes are features. Some nontrivial features are explained below.

- *Enclosure* is a mechanism to absorb the difference in abstraction levels (see Section 5.1).
- *Forward reasoning* is requested by the features *metaknowledge for control* and *no automatic backtracking.*
- *Metaknowledge for control* demands other features like predicate logic, no automatic backtracking, forward reasoning, and introduction of modality.
- *Modality* in the logic system is one of the main issues in IDDL, for the inference control will be dependent on it. We are thinking of incorporating necessity/possibility and known/unknown operators.
- *Multiworld* mechanism is used to describe metamodel evolution and evaluation of model. The scenario mechanism creates a completely isolated world which is independent but still capable of having relationships with other worlds.
- *Returning instantiated sets* to an inquiry is requested due to *no automatic backtracking* feature. Elements of IIICAD will return all possible instances to an

inquiry, which does not require automatic backtracking for exhaustive search and gives possibilities for both depth- and breadth-first searches. This further begs for the introduction of *set concept*.

- *Scenarios* are the kernel mechanisms to realize multiworlds which are important to describe the stepwise nature of design processes.
- *Three valued logic* will play an important role in IDDL. It will however be introduced as operators rather than truth values to avoid unnecessary complexity in the inference algorithm.

7.2. Example from IDDL Prototype

Figure 18.5 shows a typical display from our prototype implementation of IDDL. Based on the discussions in Section 7.1 we have developed this version on our Smalltalk-80 system. In Fig. 18.5, design browsers are used to manipulate design information such as scenarios, constraints, predicates. In this version of IDDL, *constraints* correspond to the covariants of Section 3.1 and are meant to be the design specifications, while *predicates* correspond to the variants and are used for object description.

The two windows on the left of Fig. 18.5 are snapshots of the constraints for the bridge and its subparts. The other two overlapping windows on the right show the part-assembly relationships among substructures. The designer can explore various possibilities by communicating with the system through these windows.

8. CONCLUDING REMARKS

In this paper, we presented a unifying framework to describe design knowledge. Our starting point, theory of CAD, allowed us to formulate design maxims which were converted into IDDL specifications. Theory of CAD enabled us to understand, clarify, model, and formalize design processes and design knowledge in an intelligent CAD environment.

The development of IDDL was given priority to the development of other subsystems of IIICAD. The current goals of the project are as follows:

- To implement more powerful prototyping tools for IDDL development.
- To construct a more complete version of IDDL by using these tools.

Near future work includes:

- To develop subsystems of IIICAD such as SPV, API, and IUI.
- To incorporate existing CAD tools and "knowledge-based systems" in the IIICAD framework.
- To justify our methodology of developing IIICAD and eventually to prove the effectiveness of our theory of CAD by case studies.

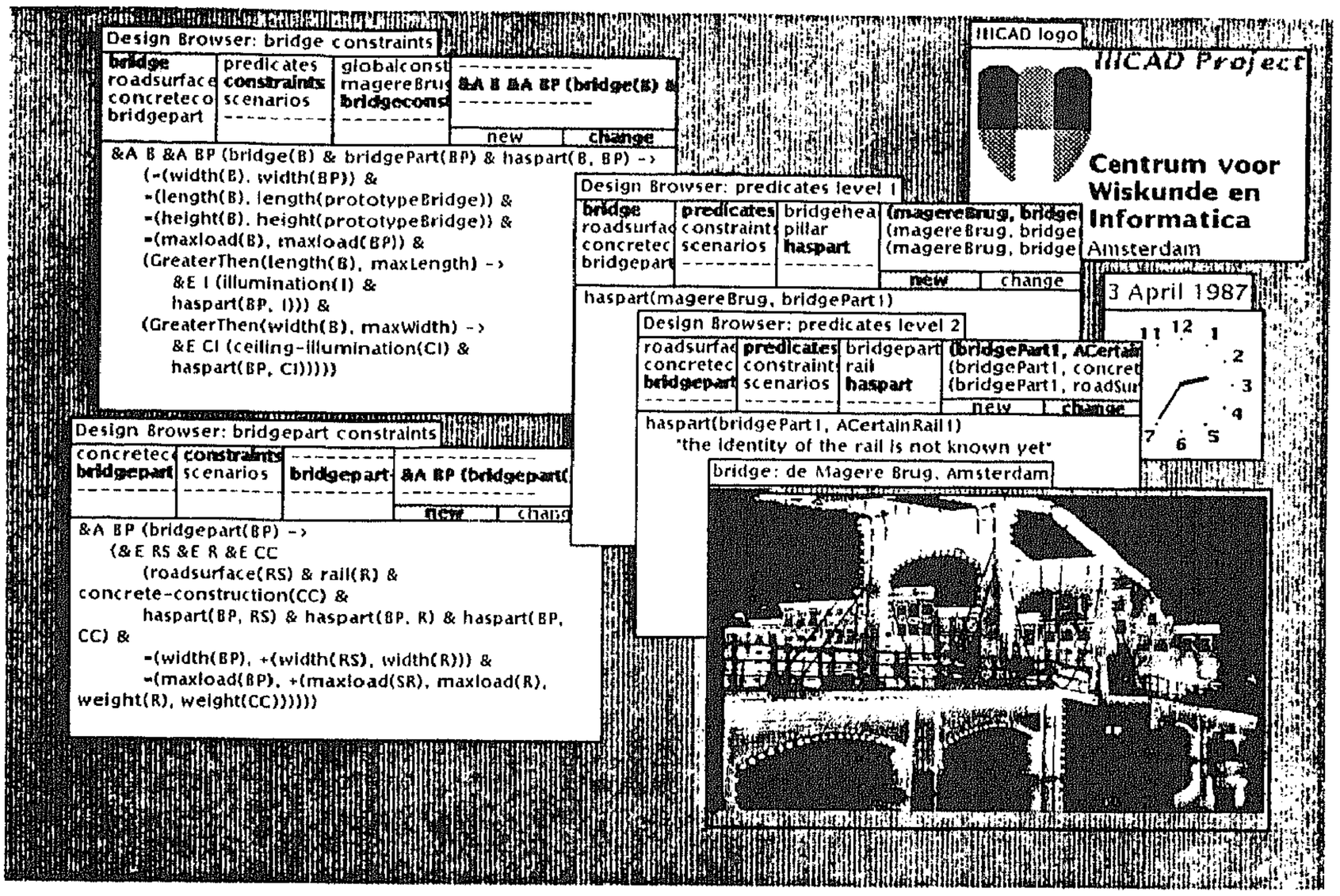


Fig. 18.5. Example bridge design with IDDL prototype

ACKNOWLEDGMENTS

We are grateful to M.M. Megens for her enthusiastic work in IDDL implementation and P.J.W. ten Hagen and Zs. Ruttkay for their critical remarks on the earlier version of this paper.

REFERENCES

Bobrow, D. G., Mittal, S., and Stefik, M. (September 1986): "Expert systems: Perils and promise," *Communications of the ACM*, **29**(9), pp. 880-894.

Brachman, R. J. (1979): "On the epistemological status of semantic networks," in *Associative Networks: Representation and Use of Knowledge by Computers*, Findler, N. V. (ed.), Academic Press, New York, pp. 3-50.

Brooks, F. (1975): *The Mythical Man-Month*, Addison-Wesley, Reading, MA, USA.

de Kleer, J. (December 1975): "Qualitative and quantitative knowledge in classical mechanics," Technical Report No. AI-TR-352, Artificial Intelligence Lab, MIT, Cambridge, MA, USA.

de Kleer, J. and Brown, J. S. (1984): "A qualitative physics based on confluences," *Artificial Intelligence*, **24** , pp. 7-83.

Forbus, K. (1984): "Qualitative process theory," *Artificial Intelligence*, **24** , pp. 85-168.

Goldberg, A. and Robson, D. (1983): *Smalltalk-80: The Language and its Implementation*, Addison-Wesley, Reading, MA, USA.

Hayes, P. J. (1985): "The second naive physics manifesto," in *Formal Theories of the Commonsense World*, Hobbs, J. R. and Moore, R. C. (eds.), Ablex, Norwood, NJ, USA, pp. 1-36.

Kowalski, R. (1978): "Logic for data description," in *Logic and Data Bases*, Gallaire, H. and Minker, J. (eds.), Plenum, London, pp. 77-103.

Kuipers, B. (1986): "Qualitative simulation," *Artificial Intelligence*, **29** , pp. 289-338.

Lorie, R. (1982): "Issues in database for design applications," in *File Structures and Data Bases for CAD*, Encarnacao, J. and Krause, F. L. (eds.), North-Holland, Amsterdam, pp. 213-222.

McDermott, D. (January 1982): "Nonmonotonic logic II: Nonmonotonic modal theories," *Journal of the ACM*, **29**(1), pp. 33-57.

Ramamoorthy, C., Shekhar, S., and Garg, V. (January 1987): "Software development support for AI programs," *IEEE Computer*, **20**(1), pp. 30-40.

Stefik, M. and Bobrow, D. G. (Winter 1986): "Object-oriented programming: Themes and variations," *AI Magazine*, **6**(4), pp. 40-62.

Stefik, M., Bobrow, D., and Kahn, K. (January 1986): "Integrating access oriented programming with a multiparadigm environment," *IEEE Software*, **3**(1), pp. 10-18.

Tomiyama, T. and Yoshikawa, H. (1985): "Requirements and principles for intelligent CAD systems," in *Knowledge Engineering in Computer-Aided Design, Proceedings of the IFIP Working Group 5.2 Working Conference 1984 (Budapest)*, Gero, J. S. (ed.), North-Holland, Amsterdam, pp. 1-23.

Tomiyama, T. and Yoshikawa, H. (1987): "Extended general design theory," in *Design Theory for CAD, Proceedings of the IFIP Working Group 5.2 Working Conference 1985 (Tokyo)*, Yoshikawa, H. and Warman, E. A. (eds.), North-Holland, Amsterdam, pp. 95-130.

Tomiyama, T. and ten Hagen, P. J. W. (April 1987): "The Concept of Intelligent Integrated Interactive CAD Systems," CWI Report No. CS-R8717, Centre for Mathematics and Computer Science, Amsterdam.

Tomiyama, T. and ten Hagen, P. J. W. (1987): "Organization of design knowledge in an intelligent CAD environment," in *Expert Systems in Computer-Aided Design, Proceedings of the IFIP W.G. 5.2 Working Conference 1987 (Sydney)*, Gero, J. S. (ed.), North-Holland, Amsterdam, pp. 119-147.

Tomiyama, T. and ten Hagen, P. J. W. (June 1987): "Representing Knowledge in Two Distinct Descriptions: Extensional vs. Intensional," CWI Report No. CS-R8728, Centre for Mathematics and Computer Science, Amsterdam.

Wegner, P. (July 1984): "Capital-intensive software technology," *IEEE Software*, **1**(3), pp. 7-45.

Yeomans, R., Choudry, A., and ten Hagen, P. J. W. (eds.) (1985): *Design Rules for a CIM System*, North-Holland, Amsterdam.

Yoshikawa, H. (1981): "General design theory and a CAD system," in *Man-Machine Communication in CAD/CAM, Proceedings of the IFIP Working Group 5.2 Working Conference 1980 (Tokyo)*, Sata, T. and Warman, E. A. (eds.), North-Holland, Amsterdam, pp. 35-58.

MADE: A Multimedia Application Development Environment

I. Herman, G.J. Reynolds
Centre for Mathematics and Computer Science
Kruislaan 413, 1098 SJ Amsterdam, The Netherlands
J. Davy
Groupe Bull
7, rue Ampère, Massy 91343, France

Abstract – MADE is the acronym for an ESPRIT III project aimed at developing a programming environment for multimedia applications. The resulting software library is based on C++ and will operate on both UNIX workstations and PC–based platforms. This paper gives a technical overview of the project and describes a number of application scenarios where the MADE environment will provide significant help for multimedia programming.

1 Introduction

The emergence of multimedia is one of the most significant developments in computing technology in recent years. Glossy multimedia applications are frequently demonstrated, often on a range of platforms. The major workstation hardware vendors feel the need to come to technical fairs with impressive displays that mix graphics, video, imaging, and sound. Technology analysts predict that multimedia related hardware development will be one of the most important boom areas of electronics in the years to come.

However, the majority of available multimedia environments aim at *hypermedia authoring*, i.e., they offer the means to interactively create hypermedia documents. We use the term "document" as a multimedia term, hence, implying more than our traditional paper–based understanding. It should be perceived as a potentially complex composition of related media information, thus it is a multimedia document, which can be "read" or viewed in a non–sequential fashion by following semantic connections (or links) between the various media components, hence it is hypermedia.

Although the concept of a hypermedia document is a powerful one, it does not cover all applications of multimedia. The ability to combine, modify, or even synthesize multimedia data is often necessary for more complex multimedia applications. For example, a user might wish to extract a frame from a video sequence, modify it with standard image processing tools, combine the image with some synthetic graphics, and then exchange the original frame with the modified image. The description of such actions does not fit easily within the model of a hypermedia document, in spite of the sophisticated interaction tools which are often provided as part of authoring environments. We conclude, therefore, that there is a clear need for a programming environment which allows for and actively supports the development of such applications.

Techniques for combining media are extremely disparate and use results from various fields of computing technology, such as, high quality synthetic graphics, image processing, speech synthesis, etc. Some of these techniques are also highly application dependent. Consequently, it is almost impossible to define a closed programming environment which encompasses all such techniques and dependencies. The "traditional" answer to this kind of challenge is to use object–oriented techniques: services are offered in the form of objects, and these can be extended by the programmer to include any necessary application–dependent tools.

The European Communities' ESPRIT III project MADE (Multimedia Application Development Environment[16]) has the ambitious goal of defining and implementing a portable object–oriented development environment for multimedia applications. The outcome of the MADE project will be a programming environment, based on C++, running on various UNIX platforms, as well as on MsWindows environments. This paper gives a technical overview of MADE : it describes its major services and a number of "application scenarios" which make significant use of these services. It is not the purpose of the paper to give a detailed description of the complete project; that would go far beyond the paper's scope. The interested reader should consult the "official" MADE documents to gain a more detailed insight (e.g., [2,8,9,10,16,30]).

The MADE project is still an ongoing activity. Consequently, some problems are still open and will be solved only later in the project. For this reason this paper sometimes raises issues without presenting complete solutions.

2 General overview

The full MADE environment contains a large number of different objects and related services. The majority of these fall into two important categories, namely: *toolkits* and *utilities*. (Note that the object–oriented nature of MADE makes it possible for an end–user to add new objects to both toolkits and utilities and to extend the functional capability of existing ones.)

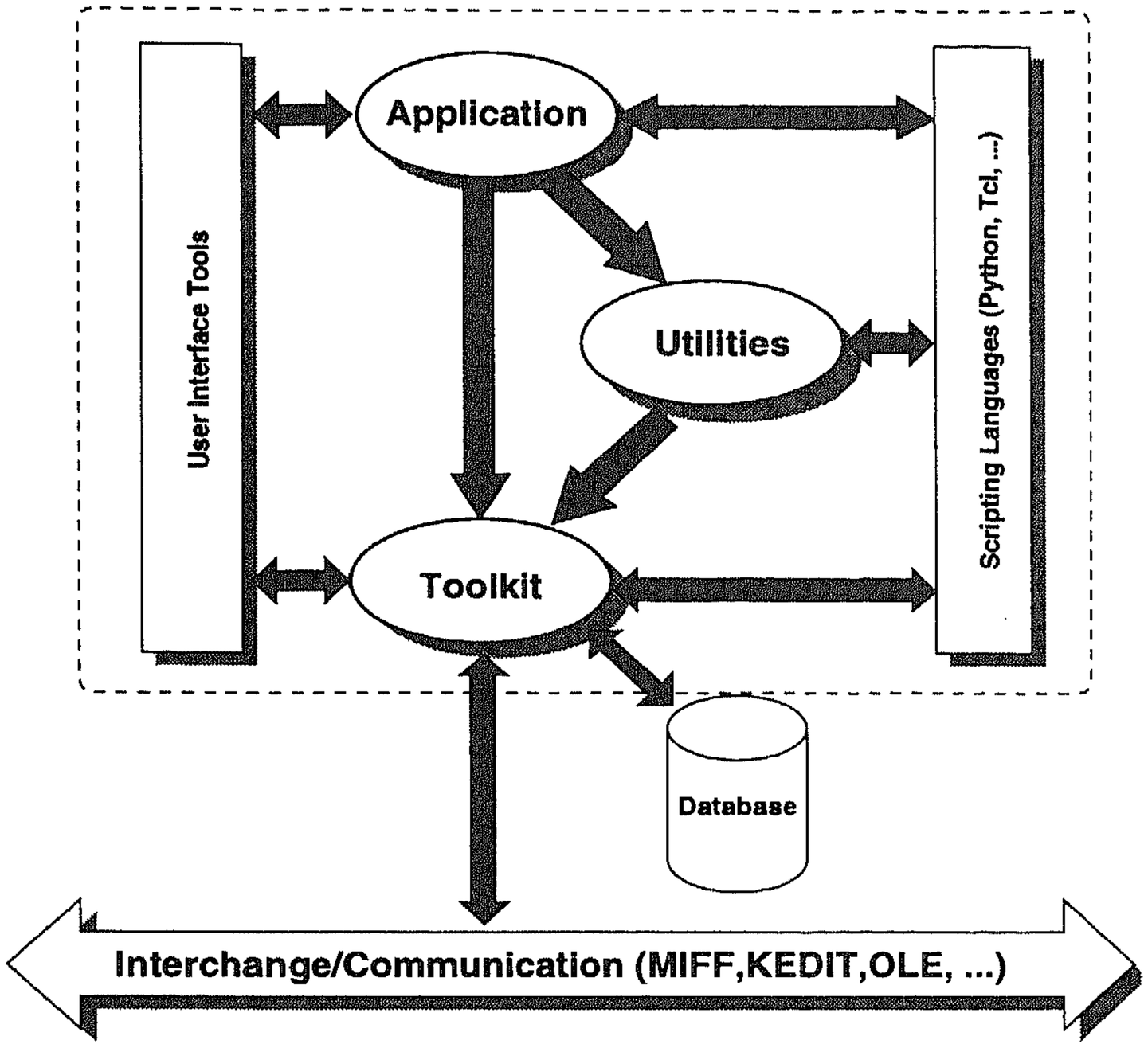


Figure 1. Tooliits and Utilities

The *toolkit* category (or level) represents a collection of objects that are considered to be fundamental to multimedia programming. It includes, objects that interface with different media. It also includes objects which, although not directly involved in handling specific media, play a fundamental role in constructing more complex multimedia applications. Some details of the toolkit level will be given below (see section 3).

Although it is possible to construct complex applications using only the MADE toolkit level, doing so may be unnecessarily tedious and error–prone. Consequently, the *utilities* level has been defined on top of the MADE toolkit. This level includes objects which implement more complex functionality and which are considered to be essential for most multimedia applications. Application programmers may choose to use some of these utility objects; however, the toolkit level is never completely obscured, and an application is free to make direct use of toolkit objects if necessary (see Figure 1; some of the terms appearing on the Figure will be described in later sections).

A common *object model/* was defined and developed at an early stage of the MADE project to ensure the smooth cooperation between objects in the MADE library and to provide a clear conceptual approach to some of the technical issues raised by multimedia programming in general. This object model defines a conceptual layer on the top of the implementation language of MADE (i.e., C++), and it describes numerous features of objects within MADE . As far as the application programmer is concerned, two characteristics of this model are of a great importance: the use of *active objects* and the presence of *delegation/*.

In MADE , objects may be *active*, that is, they can have their own thread of control (within the shared address space of the same UNIX or MsWindows process). This capability is exploited by the implementation of the MADE toolkit, and is a major tool used in defining synchronisation among media (see section 3.2.3 below). Application programmers have to be aware of this situation if they decide to use the toolkit level objects directly.

The concept of *delegation* in the MADE object model applies to an object's methods. Using delegation, an object may delegate some or all of its behaviour (i.e., the messages it serves) to any number of other objects, which then act on its behalf. The notion is not unlike inheritance, but delegation is dynamic, i.e., the target of delegation may be set and re–set at run–time. Delegation plays an important role in establishing constraints in MADE (constraint objects are part of the toolkit), and offers an advanced means to describe temporal behavioural control. A more exact semantics of delegation is described in [20]; see also [1] for a fuller description of the concept within the framework of the MADE object model.

The object model is realised in the form of an extension of C++, called mC++. The mC++ translator generates a set of C++ classes, as well as library and macro calls; this "intermediate" level can also be accessed by programmers directly if they do not wish to use yet another programming language. Details of the object model are normally hidden to most application programmers and are only of real interest to toolkit and utility developers. The full technical description of this object model is omitted here; the interested reader should refer to [1] for a general overview and to [2] for a complete description of the model and mC++.

The object model is not the only means to achieve smooth cooperation among objects. All MADE objects also include general features that allow them to be used under various circumstances in a unified way. Some examples of these features are given below.

All objects in the MADE system can be *persistent/*. This means that they may "store" themselves in a database and can restore their content at a later stage of the application's lifetime or even during the execution of some other application. This feature is present for all MADE objects by default; the only step the application program has to do is to invoke certain implicitly defined member functions. Furthermore, the MADE toolkit level includes a special object which acts as an interface to various database systems. Although this interface does not cover all known database systems, it does provide an interface to some object–oriented and relational databases. Here again, the general features required by the database access are included in all MADE objects in a database–independent way, and the specific method of database

access is hidden by the general database management object (see [29]). Interfacing to a new database system is achieved by specialisation of the general database class.

MADE objects, primarily utility objects, are also prepared for distributed access. This not only means that the MADE library includes specific objects for inter–process communication, but also that MADE objects are prepared to "convert themselves" into a format suitable for communication and, conversely, can "reconstruct" their internal state based on data coming from a communication channel. A sophisticated object–oriented communication protocol (called KEDIT [18]) is currently under development for \unix platforms, which will allow MADE applications to offer object–based services, and will provide means for the transfer of MADE objects from one MADE application to another. The features offered by the combination of MADE objects and KEDIT are similar to the kind of object services defined by the Object Management Group[1]. On MsWindows platforms the OLE protocol will be used to provide similar facilities; this is already an integral part of these environments.

All MADE objects include a general mechanism known as a "dynamic call interface". This interface makes it possible to call an object's member functions, knowing the object's handle and a *string* description of a member function's signature. This string can be constructed at run–time, hence the "dynamic" nature of the call. This feature permits MADE objects to be accessed easily from scripting languages, and provides a simple way of constructing interfaces to other programming languages.

3 Toolkit objects

The primary goal of the MADE toolkit is the provision of a basic set of features and facilities for multimedia programming. This includes control over different media, as well as other types of objects that have been identified as fulfilling a fundamental role.

3.1 Media objects

The MADE toolkit includes *media objects*, i.e., objects whose function is to directly control different media in a unified and hardware/firmware independent way.

The toolkit includes four main categories of media objects: graphics objects (for two and three dimensional graphics), animation, audio and video objects. All of these objects "hide" their respective device–dependencies behind specific low–level abstract interface objects, thereby cleanly separating their MADE specific behaviour from particular device dependent features. Adapta-

---

[1)] OMG is an industrial consortium aiming at the definition of object services in general. In their CORBA specification[24], OMG gives a specification for object services in a distributed environment. However, CORBA is still not final, nor is there a reliable implementation available yet. If, by the end of the MADE project, OMG produces a final version of their specification, replacing KEDIT with this specification will be considered

tion of a media object type to a new environment simply requires the definition of a new device–dependent subclass of the appropriate general interface object.

Some of the categories listed above contain relatively simple objects. Their task is to provide a mapping from the MADE library structure onto their respective interface object. This is the case, for example, with video and audio objects. The most critical aspect of the definition of these objects is synchronisation. The objects and their device specific interfaces must be matched with the synchronisation model of MADE (see section 3.2.3 below) and with the requirements and facilities provided be the specific hardware that is used.

Audio and video objects rely on Microsoft's Multimedia Environment for \mswin , which is a de–facto standard in this area. On UNIX , portable video and audio services are used: the Video Extension of the X Windows system for video ([5]) and the AudioFile server for audio ([19]).

2D and 3D graphics requires a greater degree of complexity. Indeed, the collections of both the 2D and the 3D graphics objects represent two full–blown subsystems per se, which are also usable in a stand alone fashion for graphics purposes.

For 2D graphics, the MADE toolkit reuses an existing object–oriented 2D graphics system, called GoPATH[7], by adapting it to the requirements of MADE . These 2D objects include various shapes, associated clipping areas, composition rules, attributes, etc. The programmer has the possibility, via sub–classing, to define new shapes and include these into the full 2D world. GoPATH is currently based on X11R5 for UNIX platforms, and on MsWindows

.

The 3D subsystem provided by MADE supports a mapping between general 3D objects (shapes, surfaces, lighting and view control, etc.) and existing 3D packages. A mapping to SGI's GL library is currently being developed. The use of PEX[6] or Open-GL, as a replacement for GL, will be considered in the future. It has to be stressed that it is *not* the goal of the project to define yet another 3D graphics package; the emphasis is more to provide an object–oriented layer on top of existing packages which is fully integrated into the MADE environment. On the other hand, due to the object–oriented nature of the MADE toolkit, it is possible to extend, by sub–classing, the basic 3D functionality (e.g., to add a proprietary ray–tracing module) if necessary.

Graphics objects (both in 2D and 3D) do not have a temporal dimension; essentially, they describe static scenes. This is in contrast with the inherently temporal nature of audio and video objects. To alleviate this contradiction, MADE includes separate *animation objects*, which describe, and even auto-

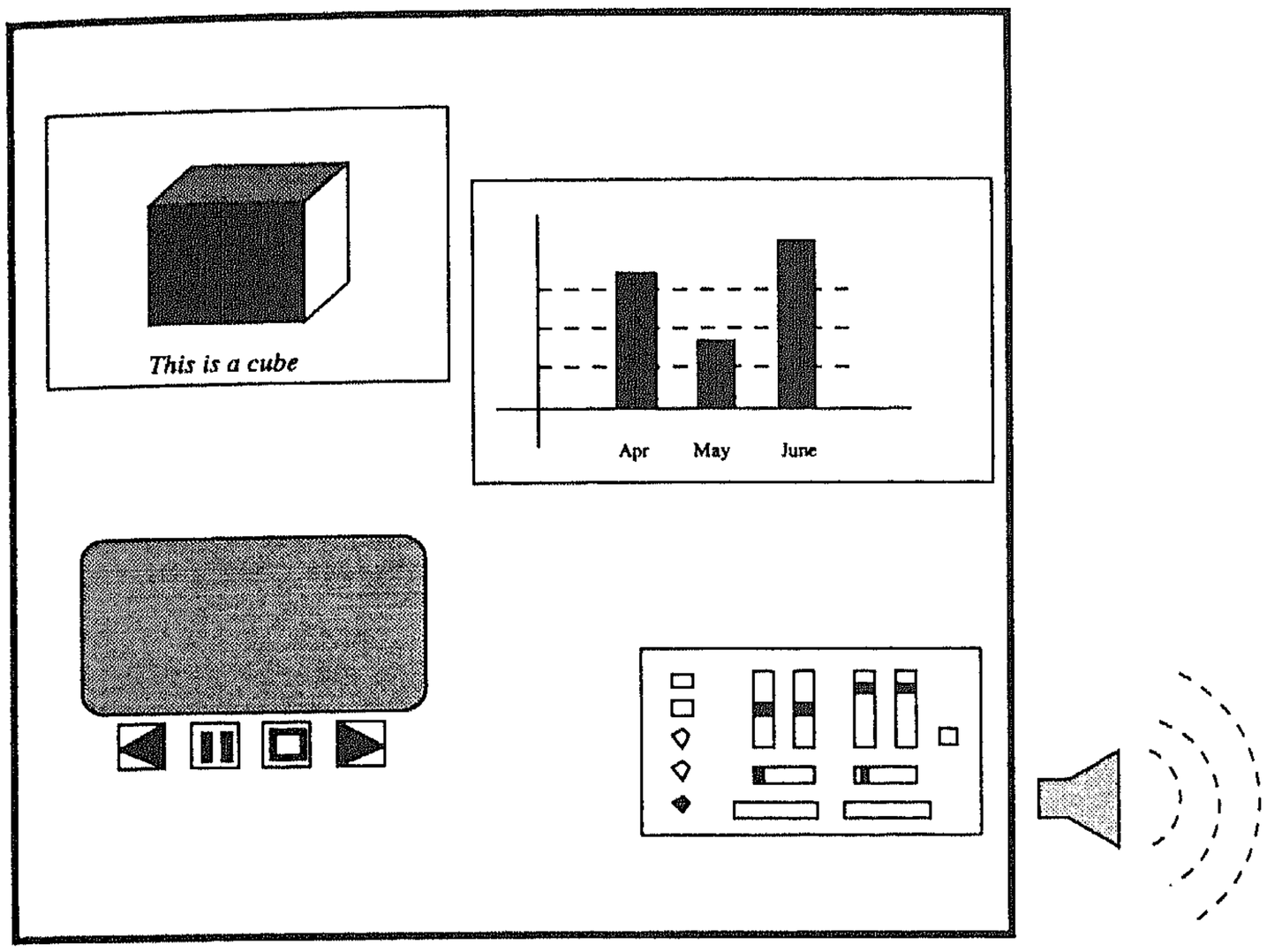


Figure 2. A rudimentary example for multiple media in an application

matically generate, sequences of scenes. The methods and algorithms used in animation may be extremely complex and, more importantly, are dependent on specific application areas. Although simpler, built–in, animation techniques based on animation curves are available, MADE animation objects also support animations defined as scripts, using a scripting language, which are then interpreted by the animation objects. Animation objects are active objects, and are thereby subject to the same synchronisation behaviour and control as audio and video objects (see section 3.2.3).

3.2 Combination objects

While it is clearly possible to build impressive applications that rely solely on MADE media objects, using complex static and animated graphics, running a video on the screen and playing audio, etc., the shortcomings of such an approach very soon become visible when more complicated application programs have to be devised and implemented on this basis. The very rudimentary example on Figure 2 already demonstrates that: interactive behaviour assigned to graphics objects has to be combined to control video output; visual representations for audio control have to be defined and implemented; 2D and 3D objects have to be combined in one picture, etc.

Basic media objects become really usable if they can be *combined/* easily in a variety of ways. The combination of media objects (and MADE objects in general) within an application has received particular emphasis in the specification of the project in order to enhance the usability of the MADE tools. Five

major areas of combination have been identified: *imaging*, *structuring*, *synchronisation*, *interaction*, and *constraint management*. Each of these will be looked at in the following sections.

3.2.1 Imaging

Video objects, 2D and 3D graphics objects, and imported image files, may all be visualised on the display screen. Very often, an application may require such "images" to be combined in some way. For example, a complex picture might be created by combining a snapshot of a video sequence, some annotated 2D text, and an imported image used to selectively filter the result.

MADE supports this kind of combination via an *image object*. All MADE media objects that produce displayable output can be directed to produce image objects. These can of course be presented, but they can also be converted into video frames or stored in a particular file format.

3.2.2 Structuring

The importance of *structuring/*, i.e., of creating aggregates of different objects in interactive programs, has long been recognised in computer graphics. The majority of graphics packages provide some form of aggregation, such as structures in PHIGS[12], the scene database of IRIS Inventor[27], or the Go trees in GoPATH[7]. Although the structures used in these examples are relatively simple (directed acyclic graphs or trees), the appearance of hypertext and, lately, of hypermedia systems makes it clear that more general aggregation facilities are necessary.

The MADE toolkit answers this requirement by including a general graph management facility. *Graph objects* are provided to support the specification, management, and traversal of general graphs, with no restrictions imposed on their types. (nodes of these graphs may refer to any MADE object).

Graph objects provide a sound basis for the type of structuring required by graphics as well as for complex hypermedia navigation systems. They are fully integrated into the MADE environment, which has a number of advantages. For example, graphs provide an automatic protection against uncontrolled concurrent access of structures by active objects, they can be exported and imported using the same persistency mechanism as is defined for all other MADE objects (i.e., complete graph structures can be stored in databases), etc.

3.2.3 Synchronisation

Synchronisation has always been one of the central problems of multimedia applications and the MADE toolkit offers a consistent solution to this issue.

The fundamental synchronisation scheme used in MADE is called *reference point* synchronisation. For each, so called, *synchronisable* object, a series of media specific reference points can be defined (for example, video frames, audio samples, etc.). Each reference point contains internal "instructions" for synchronisation and references to other synchronisable objects that are to be synchronised with it Synchronisable objects are active objects; when they reach a reference point, synchronisation is performed by exchanging messages with other active objects, waiting for their replies, etc. The reference

point model has been inspired by [3]; its details in the MADE environment are specified in [8]. Audio, video, and animation objects are obvious examples of synchronisable objects[2]. A MADE programmer may also create new, application–specific, synchronisable objects.

The MADE toolkit also includes a higher–level mechanism for time–based synchronisation, that is based ultimately on the reference point model. This mechanism defines different types of *schedulers* which an application may use as building blocks for more complex time–based synchronisation scenarios (see [8] for further details). These schedulers all assume the existence of a special synchronisable object[8] for further details). These schedulers all assume the existence of a special synchronisable object within MADE , namely a *timer*. The approach of building time–based synchronisation on the top of a more general mechanism, instead of considering it as a basic feature, allows the MADE library to be used in environments which do not offer the necessary real–time facilities.

3.2.4 Interaction objects

Multimedia applications are very often highly interactive; it is therefore essential to have very good tools to support the construction of complex interaction scenarios.

---

[2] To be very precise, certain animation objects, which describe random animation, cannot be properly synchronised, but these objects represent a small minority vis-à-vis animation objects in general.

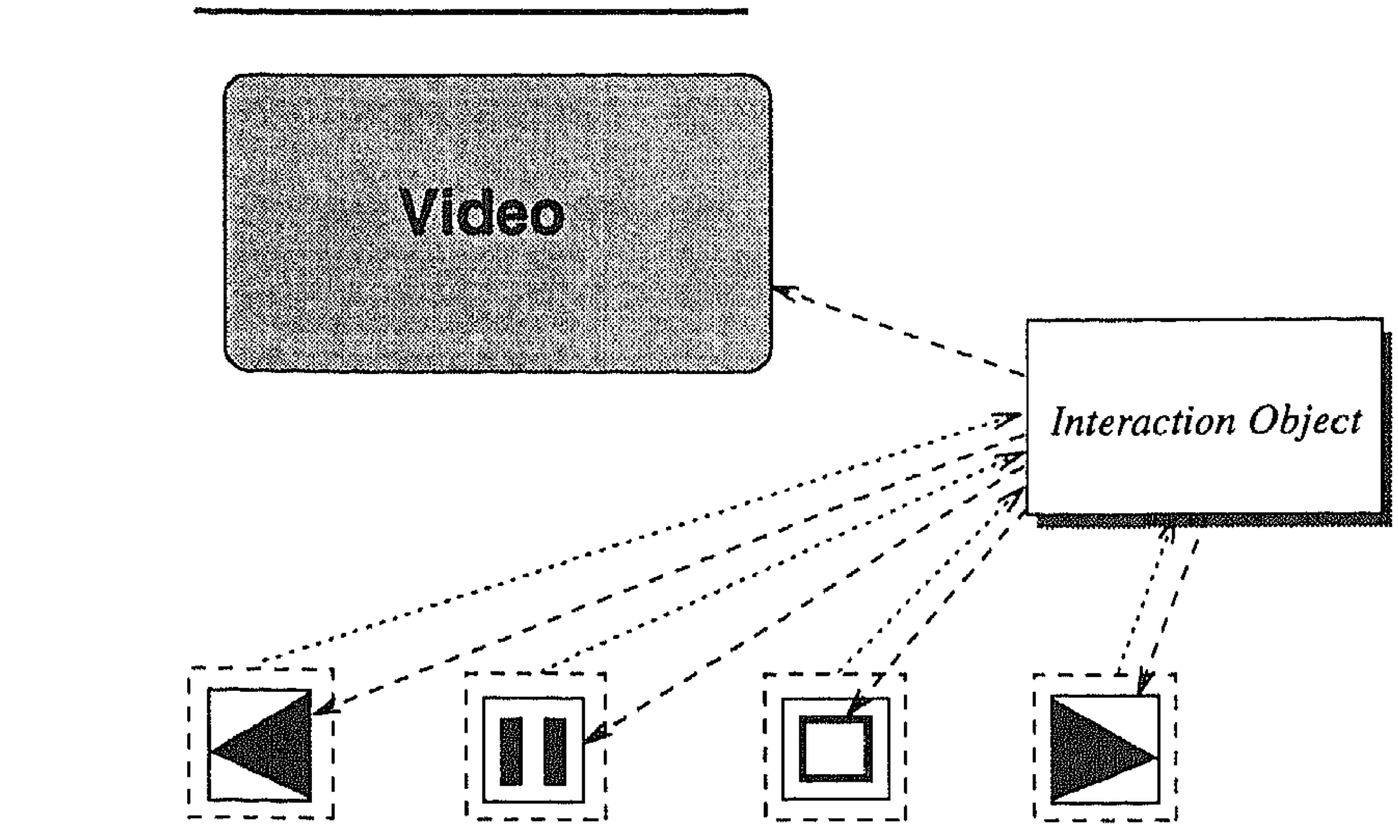


Figure 3. Use of Interaction Objects

The MADE project does *not* aim at developing a completely new user interface management system. Instead, MADE objects may be embedded into an existing user interface environment, like the Athena Widget set of X Window System, the Motif toolkit, MsWindows. Nevertheless, not all user interaction can be adequately managed by these tools; many complex interaction scenarios will involve MADE objects directly (e.g., for direct manipulation). The scheme developed in MADE for achieving these complex interaction scenarios is based on the notion of *sensors* and associated *interaction objects*.

Sensors are best understood in the context of graphics: in this context they define sensitive areas on the screen, which can be "activated" by external interaction, typically mouse events. Sensors are associated with MADE objects via interaction objects. In effect, they provide a sensitive region which acts as a focal point for interaction with these objects. For some objects, sensors cannot be attached to the object itself, but, must instead be attached to a visual representation of the object, in the form of graphics object. This might be the case for sensors attached to audio objects. The notion of sensor is general enough to accommodate regions involving higher dimensions, including time. It can also be applied in association with interaction input devices that provide non–geometric input measures, such as audio input devices, pressure sensitive devices, etc.

Sensors forward events to interaction objects; it is part of the sensor's initialisation procedure to decide which interaction object it is connected to. The interaction objects react to these events by following a pattern of behaviour that is defined as part of the interaction object. Several sensors may be connected to the same interaction object.

In very simple cases, interaction objects perform straightforward and predefined tasks (like, for example, reshaping a graphics object). In other cases, much greater complexity may be required, perhaps providing control over several MADE objects and receiving events from several sensors (e.g., the video control board depicted in Figure2 reacts on the sensors of the graphical objects describing the four push–buttons, may control the visual appearance of these buttons and, of course, controls the video object proper or perhaps a combination of objects; see also Figure 3). To describe such complex interaction behaviour, MADE introduces a type of interaction object that implements a general finite state machine (see [11]). These objects have a default finite state machine for a specific interaction scenario; however, the user can also assign a script to an interaction object, which, conceptually, includes a complete scripting interpreter (see also section 4.1.2). The assignment of a script automatically overrides the default behaviour of the interaction object. This high degree of openness, with respect to the end–user, is a very valuable feature of the MADE interaction management.

3.2.5 Constraint management

Provision of a general purpose constraint system within MADE for all of the potential uses of constraints in a multimedia development environment would justify a development project in its own right. Fortunately, there are various restricted types of constraint satisfier that, while not being as capable in some aspects, still provide useful functionality for dealing with certain categories of constraint.

The approach followed in the specification of constraints within MADE (see also [30]), is to consider those applications of constraint systems that are of direct relevance to the multimedia part of MADE. In effect, this restricts the scope of the constraint satisfier to the topics of geometric layout, user interface control, animation, and media synchronisation. For example, the MADE presentation facilities include a composition editor/player which may make use of constraints when defining the hypermedia document structure and presentation characteristics.

For the time being at least, only one–way constraints are proposed for MADE . While multi–way constraints provide greater expressive power to the constraint user, they also require more complex constraint satisfaction algorithms and may involve more effort on the part of the programmer to set up specific constraint objects.

4 Utilities

Utilities offer developers a higher level of functionality in that simplifies the implementation of both basic and more complex multimedia applications. In fact, the functionality of some of the utilities is such that, by "wrapping" them into a simple program, they can be used as a separate applications in their own right.

There are four main utility categories:

1) *application program interface utilities*: user interface metaphors, scripting, user interface builders, user monitoring;
2) *monomedia editors*: 2D and 3D graphics editors, animation, video, and audio editors;
3) *composition utilities*: framework for hyperdocument management, synchronisation editors, interaction and graph object editors;
4) *miscellaneous*: class browsers, generic on–line help facilities, object monitoring.

The different MADE utilities may rely on one another. For example, the user interface metaphors, to be presented below (seesection 4.1.1), are reused by monomedia editors (see section 4.2) and the composition utilities (see section 4.3).

Utilities, together with MADE toolkit objects, offer a set of building blocks which can be used in various ways to create different types of MADE application program architectures. Some of these architectures will be described in section 5 below.

4.1 Application program interface utilities

Application program interface utilities consist of a set of tools that help an application programmer to prototype and develop a final MADE application. Although the facilities provided by some of these utilities are fairly standard these days, it is nevertheless necessary to provide them in the context of the MADE environment. Note that only some of the more important tools are presented in this paper.

4.1.1 User interface metaphors

The visual representation and control of media objects is not always obvious. Indeed, to control certain attributes of media objects, relatively complex visual tools with associated interaction behaviour have to be developed. These tools can be used on different levels: in program development, in authoring, or in the final playback of authored documents. These user interface metaphors play an essential role in defining complex interactions operating on the objects; and it may sometimes be much easier to attach a sensor to these metaphor objects, rather than to try to define a sensor on the object proper (see section 3.2.4).

There are numerous examples for such user interface metaphors. Some examples are:

- Video control board for stopping, playing, rewinding, providing fast forward and backward motion, etc.
- Audio panel containing volume control, channel control, etc.

- Control boards for the manipulation of graphics object attributes (colour, lighting, shading attributes, etc.)

All these objects, collectively called *user interface metaphor* objects, are part of the MADE utility library. Other utilities (primarily the editors, see section 4.2), reuse these objects, thereby providing a common look–and–feel among MADE utilities. MADE applications may of course choose to ignore these objects and to implement similar user interface facilities by themselves.

4.1.2 Connection to scripting Languages

Several MADE objects make use of scripting languages: animation and interaction objects have been mentioned in the preceding sections, and there are others, too. It is also perfectly feasible to create full–blown applications, either in a prototype or in final form, where the "user–level" program is a script.

MADE does not introduce its own scripting language. Instead, all objects that make potential use of scripting access the interpreter functionality via an abstract general scripting interface. This general scripting interface is then specialised to access specific languages and their interpreters. This lets the final choice over which scripting language is used be made by the MADE application developer or even the end–user. Furthermore, several scripting languages can coexist within the same MADE application (see [10]).

In order to be usable for MADE , a scripting language should have an embeddable interpreter. That is, it should be possible to link the interpreter to C/C++ and C/C++ functions should be accessible from the language somehow. Conversely, functions of the scripting language should be accessible from C/C++. Note that the availability of the dynamic call interface of MADE objects plays an essential role in interfacing such interpreters: it is not necessary to create a special "stub" for each MADE object in the scripting language; indeed, MADE objects can be created, and their methods invoked, based only on their signature.

There are several general embeddable interpreters available. Currently, the MADE toolkit includes an interface to Python, a language developed at CWI ([31]), and to Tcl ([25]).

4.1.3 User interface builder

The MADE utilities workpackage includes a prototype authoring toolset that incorporates a user interface builder for use on UNIX platforms. This based on an existing tool that combines Tcl and Motif, extended to included specific user interface entities for multimedia applications. A similar development (being carried out independently of the MADE project) for Python may be used in later stages of the project.

On MsWindows environments, Visual C++ will be used as a user interface builder. For the integration of MADE objects and utilities, subclasses of the "Microsoft Foundation Classes" will be developed and accessed directly from Visual C++. This has already been validated for the 2D editor of GoPATH[7].

4.2 Monomedia editors

The role of monomedia editors is relatively straightforward: they offer the means to create, modify, and display media objects. There is nothing particularly unusual or new in these utilities, except that they all abide to the architectural demands for MADE editors, as described above and they incorporate the notion of configurability. Each monomedia editor is able to be configured at start up in one of a few modes of operation. For example, its possible to configure an editor for use as a player-only tool. This mechanism is used extensively by the composition utilities (seesection 4.3). Note that MADE editors make use of the visual metaphors described in section 4.1.1 to give a unified outlook.

MADE editor objects may be used in various application settings. This includes being activated alongside with other MADE objects, e.g., other editors. In this case, editor objects may be active objects, and the mechanism provided by the MADE object model will ensure that data managed by several editors will not be corrupted by concurrent access. Editors may also be wrapped up into separate application programs to run as stand–alone processes. In this case, editors may operate on MADE objects residing in a database or they can manage objects received via a communication channel using, e.g., the KEDIT protocol (see section 2).

The *2D graphics editor* is based on an existing program, called `godraw` (related to GoPATH, mentioned earlier). The facilities supported by this editor are relatively straightforward, and are in line with other 2D graphics editors, available for different platforms.

The *3D graphics editor* emphasises two aspects of 3D editing: editing of scenes by composing 3D objects in space, and simple 3D solid modelling to create 3D bodies. It includes dialogues to control attributes like texture, colour, reflectance, opacity, etc.

The *audio editor* offers facilities to "cut" and "paste" audio tracks, apply (possibly user–specified) filters on the sound tracks, and modify their characteristics. A MIDI editor will also be available.

The *video editor* offers similar facilities to that of the audio editor: "cut" and "paste" of video sequences, modification of its characteristics (if the underlying hardware permits it), retrieve and replace frames as images, etc.

A separate *animation editor* is also provided, which allows for the interactive creation and editing of animation curves, and animation scripts.

Note that, under MsWindows , Microsoft's Multimedia Environment already contains some multimedia editors; to avoid duplication, these editors will be reused as much as possible.

4.3 Composition utilities

Composition editing and playback is the mechanism within MADE for developing and viewing multimedia/hypermedia documents, both from the point of view of an author of such documents and also from the point of view of the final user(s) of a MADE application based on the document concept. The composition editing and playback utility is one of the main integrating components of the MADE application environment. It is through the definition of an abstract document structure that a hypermedia document is created and it is the pres-

entation of this hyperdocument which the end user may interact with. During both the authoring and playback modes of operation the composition utility makes direct use of the other MADE utilities for viewing or editing particular media objects, for presenting help information, for navigating the hyperdocument structure, and perhaps also for monitoring the user's actions. The composition utility drives the operation of these other utilities based on a composition graph (i.e., the internal representation of the hyperdocument).

An essential aspect of the composition facilities is the ability to define and manipulate an abstract document structure[3]. The abstract document structure is a representation of logical components which describes not only the specific types of media involved in the presentation, but also the semantic connections between media, the synchronisation constraints associated with the presentation of the logical components, geometric and other presentation attributes for each component, and specific interaction entities to be used in reading and interacting with the multimedia document.

The authoring and presentation of a hyperdocument is not only determined by the media and the composition utilities. There may be a number of alternative styles (or metaphors) for presenting a particular hyperdocument that are dependent not on the specific document itself but on the application domain in which the MADE application exists.

A specific goal of the composition utilities of MADE as a whole is to separate the presentation metaphor used for authoring and viewing a MADE hyperdocument from the underlying composition graph. The aim is to accommodate different styles of authoring and different forms of visually structuring the hypermedia information. Within the MADE project, a prototype authoring tool is being developed with a specific presentation metaphor. It will, however, be possible for another authoring tool to choose a radically different presentation scheme and implement it on the "top" of the MADE composition utilities.

The composition utilities also make provision for using an interchange format to represent the abstract document structure in a more persistent form. An interchange format enables the reuse of existing compositions, either fully or in part, and enables the exchange of documents among MADE applications. There are a number of contenders at the moment, HyTime[13] and MHEG[14] are standardised formats, and there are a other proprietary ones. A third choice would be to develop a MADE specific format (temporarily denoted as MIFF), perhaps based partly on either of the above or some other industry format. At the time of writing, Apple's Bento format has provisionally been chosen for use as the MADE interchange format (MIFF).

The composition utilities include some sub–modules with well specified tasks. These include:

An *interaction editor*, used to create or modify interaction objects (see section 3.2.4). This involves defining sensors associated with MADE objects (or with their associated visual metaphor), specifying the objects the interaction object has to control, and editing the corresponding script. The definition and/or the modification of sensors may involve, e.g., graphics editing, which

---

3) This abstract document structure is also referred to in this specification as a composition graph

means that the interaction editor may also start up a 2D graphics editor internally. In this setting, interaction objects provide a possible internal representation for hyperlinks.

The role of the *synchronisation editor* is to interactively define the synchronisation patterns among several synchronisable MADE objects. This may involve the specification of reference points, setting references of other object the synchronisable object has to synchronise with, defining the details of this synchronisation, etc. Time objects are also managed by this editor; the user may indeed prefer to use the notions of time, scheduler, and time–constraints for the purpose of synchronisation, rather than the concept of reference points. (As described in section 3.2.3, both mechanisms are available within the MADE toolkit.)

The choice of the interchange format will greatly influence whether, in the synchronisation editor, the emphasis will be placed on reference point or time–based synchronisation. HyTime, for example, expresses all synchronisations using an abstract notion of time; quite naturally, if the HyTime format, or a subset of it, is chosen, this will determine the final shape of the synchronisation editor, too.

The *graph* or *layout editor* gives a visual interface for the direct manipulation and visualisation of the composition graph (i.e., the hyperdocument structure).

Finally, the *composition editor* is the most complex composition utility, which combines and controls all other composition utilities as well as the monomedia editors, and MADE toolkit objects. It is this module which lies at the heart of all composition utilities, and which is responsible for providing all the general functionalities described above.

5 Application architectures

The notion of *multimedia application* is a very broad concept and application programmers may make use of a package like MADE in different ways. Also, the concept of a *user* of MADE (or of similar packages) has become a somewhat fuzzy notion; there are, in fact, different types of users (toolkit or utility developers, C++, script programmers, hypermedia document authors, etc) which are all, in some way or other, "users" of the MADE environment. Without claiming to be exhaustive, this section will give some, typical examples of application program architectures.

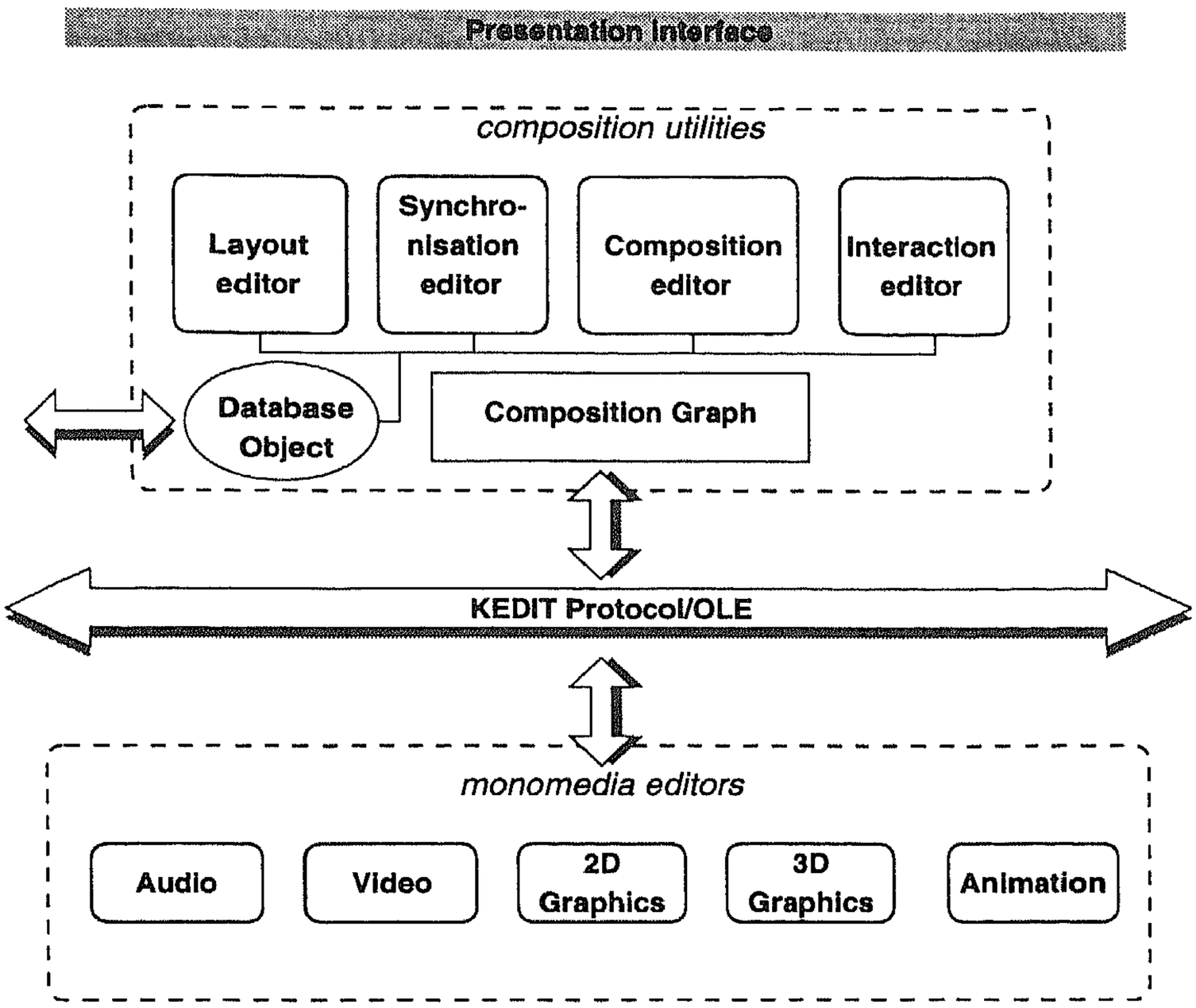


Figure 4. Composition Utilities in a MADE Application

Note that the full MADE ESPRIT project includes the development of some pilot applications. It is not the purpose of this paper to give a thorough description of the whole \esprit project, hence these applications are not described here. Suffice it to say, however, that the application program architectures, as presented below, are all represented in these various pilot applications.

5.1 "Traditional" programming

The MADE toolkit objects, plus some of the utility objects, form a powerful, albeit "traditional" programming environment for C++ programmers. This means that applications may be developed in C++ or C, and then linked to a set of run–time MADE libraries.

Figure1 gives a faithful picture of a traditional program using MADE . The application program (which is usually a single UNIX, or MsWindows task) uses different toolkit objects, either directly or indirectly, via some utility objects. A more elaborate application would also make use of an external database, accessed via the MADE database object facilities.

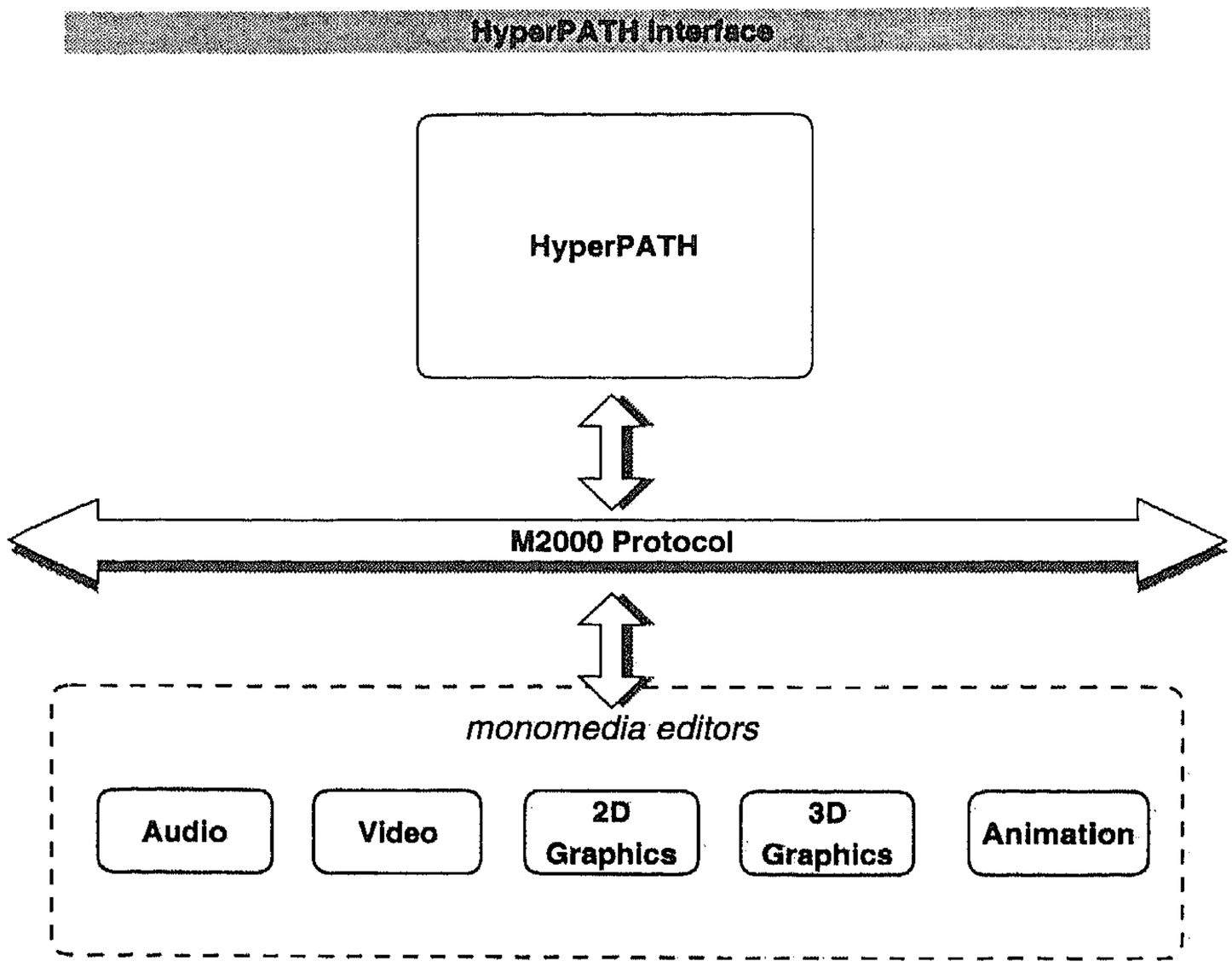


Figure 5. Use of an External Composition Tool: HyperPATH

The application program may interchange data with other applications via, e.g., the MIFF exchange format. Alternatively, the application program may offer *services*, in the form of a sophisticated multimedia server, using either the KEDIT protocol or OLE. Other applications may then either directly manipulate MADE objects via this protocol or full MADE objects may be transferred back and forth and be manipulated by different modules.

Various objects, such as the interaction and animation objects, can use scripting languages, which may be revisable by the end–user. In fact, the skeleton of the application program may also be written in a scripting language instead of C or C++; the script would then manipulate MADE objects (written in mC++) via the appropriate MADE –script interpreter interface.

Another possibility is to use C++ and, e.g., Motif to create the user–interface; this is when a graphics user interface application builder may play an important role.

5.2 Hyperdocument editing and playback

Figure 4 illustrates the possibility of hypermedia document manipulation using the full–blown composition utilities described in section 4.3 . The programming environment offered by MADE in this setting is hypermedia document authoring; quite naturally, the user community for such an environment differs radically from the community of "traditional" programmers. (Very often, to make the distinction, members of this community are referred to as "authors", as opposed to "users".)

In this authoring environment, the composition utilities are conceptually separate from the media editors. The composition utilities act as the coordinating central hub of the complete architecture. Effectively, there is an inter–editor message facility that is used to both control the operation of the media editors and to provide information to the composition utilities representing actions performed by the user through dialogues with the media editors. In this setting the media editors may be considered as separate applications or, in other terms, as separate service providers. These applications may be realised following the scheme described in the previous section.

This organisation implies that media objects or references to objects are passed between the composition utility and the media editors in order to "render" them. Similarly, edited media objects may need to be passed back to the composition editor and placed into the multimedia database.

Note that a simpler version of the architecture, including a simpler version for each of the media editors, may be configured to be used for "playback" only.

5.3 Other application schemes

The application architectures presented in the preceding two sections represent, in a way, the two extremes of a large palette. Intermediate architectures, making use of only part of the full MADE functionality are also possible and feasible. It is possible to create, for example, a HyTime–like engine based on the MADE toolkit and some of the utilities only (although these utilities may be distributed services rather then linked to the HyTime engine)[4]; interactive modelling applications, or scientific visualisation applications, are also possible, which may use the services of media editors, just as a full hypermedia authoring tool does, but with a fundamentally different user–interface.

The application architecture shown on Figure 5 illustrates another possibility for an authoring environment. As said earlier, media editors, realised as MADE applications, may be used as independent servers, provided that the external communication protocol is understood by the "wrapper" around the MADE editor objects. In such a case, an "external" (i.e., not closely MADE dependent) hyperdocument authoring tool may be used instead of the MADE composition utilities. The example used in Figure 5 is HyperPATH, formerly known as Multicard ([26]), a hypermedia editing tool developed by Bull. (The M2000 protocol referred to in the figure is the internal communication protocol defined for HyperPATH.)

---

[4] In fact, creation of an engine for a specialised set of HyTime documents is one of the pilot applications that is part of the full ESPRIT project.

6 Standardisation

In a somewhat unexpected way, activities in the MADE project have become very much relevant recently for an ongoing standardisation process within ISO. Indeed, after several years of preparations, the ISO committee ISO/IEC JTC 1/SC 24 (the committee which developed graphics standards in the past) has engaged into a project for the standardisation of a presentation environment for multimedia programming. The scope and purposes of this new project, called PREMO[15] are indeed very close to the project specifications of MADE : an object–oriented presentation environment for multimedia objects, including graphics, video, audio, etc., which incorporates specific means for the synchronisation, interaction, and combination of such media.

Fortunately for the MADE project (and, hopefully, for the PREMO project, too), contacts between MADE project members and the relevant ISO committee could be set up very quickly, due to some earlier ISO activities of several participants of the MADE project. Concepts developed within the MADE project have been included into the PREMO activities, and, conversely, some of the issues that have arisen at the PREMO meetings have provided valuable input in the design work of MADE. It can be expected that this fruitful interaction will help to shape the outcome of the MADE project in the future, too.

Acknowledgements

Obviously, MADE is a large–scale teamwork project, involving experts from a number of industrial and academic institutions[5]. Although only some of the partners are involved in the specification details of the MADE framework (others being responsible for the pilot applications), the team of experts is still rather voluminous. Instead of trying to list everybody and thereby incurring the danger of forgetting and therefore offending somebody, we prefer to omit such a long list. We would just like to express our gratitude to the full MADE team altogether.

References

1. F. Arbab, I. Herman, and G.J. Reynolds, "An Object Model for Multimedia Programming", *Computer Graphics Forum (Eurographics'93 Conference Issue)*, 12(3):C101–C114, September 1993.

2. F. Arbab, P.J.W. ten Hagen, M. Haindl, F.C. Heeman, I. Herman, G.J. Reynolds, and A. Siebes. *Specification of the MADE object model*, Technical Report, T/OM S1, Version 0.5, Esprit Project 6307 (MADE), 1993.

---

5) Namely: Groupe Bull (France), CWI (The Netherlands), INESC (Portugal), INRIA (France), FhG-IAO (Germany), BaE (UK), NR (Norway), ESI (France), Iselqui (Italy).

3. G. Blakowski, J. Hübel, and U. Langrehr, "Tools for Specifying and Executing Synchronized Multimedia Presentation", in: *Proceedings of the Second International Workshop on Network and Operating System Support for Digital Audio and Video*, 1992.

4. V. Bouthors. *Egeria Reference Manual.* Bull SA, Paris, version 2.1 edition, August 1992.

5. D. Carver. *X video extenstion protocol*, version 2. Technical report, DEC Technical Report, MIT X11 Contributions, 1991.

6. W. Clifford, J.I. McConnell, and J. Saltz, "The development of PEX", in *Eurographics'88 Conference Proceeings*, D.A. Duce and P. Jancèbe (eds), North Holland (1988).

7. J. Davy, "Go: a graphical and interactive C++ toolkit for application data presentation and editing", in Proceedings of the 5th Annual Technical X Conference on the X Windows System, (1991).

8. N. Guimarães and N. Correia. *Specification of the MADE time objects*, Technical Report, T/TO S0, Esprit Project 6307 (MADE), 1993.

9. M. Haindl, I. Herman and G.J. Reynolds. *Presentation scheme — preliminary specification*, Technical Report, T/PRS S0, Esprit Project 6307 (MADE), 1993.

10. I. Herman, F.C. Heeman and F. Leygues. *Interfacing scripting languages*, Technical Report, Esprit Project 6307 (MADE), 1993.

11. I. Herman, F.C. Heeman and G.J. Reynolds. *Interaction objects — functional specification*, Technical Report, T/IAO S1, Esprit Project 6307 (MADE), 1993.

12. T.L.J. Howard, W.T. Hewitt, R.J. Hubbold and K.M. Wyrwas, *A Practical introduction to PHIGS and PHIGS PLUS*. Addison–Wesley, Workingham – Reading, 1991.

13. International Organization for Standardization, *Information Technology — Hypermedia/Time–based Structuring Language (HyTime), (ISO/IEC 10744:1992(E))*, 1992.

14. International Organization for Standardization, *Information Technology — Coding of Multimedia and Hypermedia Information (MHEG) (ISO/IEC SC29 N354)*, 1993.

15. International Organisation for Standardization, *Information processing systems — Computer graphics — Presentation environment for multimedia objects (PREMO); ISO/IEC JTC1 SC24 WG6 OME35*, 1993.

16. J. Davy (ed), *MADE 1, ESPRIT III Project 6307, Technical Annex*, 1992.

17. O. Jojic and J. Davy, *C++ API implementation*, Technical Report, Esprit Project 6307 (MADE), 1993.

18. P. Kaplan and A. Baird–Smith, *The KEDIT Protocol,* Technical Report, Esprit Project 6307 (MADE), 1993.

19. T.M. Levergood, A.C. Payne, J. Gettys, W. Treese and L.C.S. Steward, *AudioFile: A network–transparent system for distributed audio applications*, Technical Report CLR 93/8, Digital Equipment Corporation, Cambridge Research Laboratory, Cambridge, MA, 1993.

20. H. Lieberman, "Using prototypical objects to implement shared behavior in object oriented systems" in *Proceedings of the First ACM Conference on Object–Oriented Programming Systems, Languages, and Applications, Portland*, ACM Press (1986).

21. Microsoft Inc. *AVI — Microsoft Technical Note*, 1992.

22. Microsoft Inc. *Users' Guides, Microsoft Visual C++, Development System for Windows,* 1993.

23. C. Nahaboo, *Koala project, Wool2 Reference Manual*, Bull SA, 1992.

24. OMG, *Common Object Services Specification (COSS)*, Version 1.0, Technical Report, Object Management Group, March 1993.

25. J.K. Ousterhout, *An Introduction to Tcl and Tk.* University of California, Berkeley, 1992.

26. A. Rizk and L. Sauter, "Multicard: an open hypermedia", in *European Conference on Hypertext ECHT'92*, Cambridge University Press, 1992.

27. P.S. Strauss and R. Carey, "An object–oriented 3D graphics toolkit", Computer Graphics (SIGGRAPH'92), **26**, 1992.

28. B. Stroustroup, *The C++ Programming Language*, Addison–Wesley, Reading, MA, 1991.

29. F. van Dijk and A. Siebes, *Specification of the database objects*, Technical Report, T/DBO S1, Esprit Project 6307 (MADE), 1993.

30. J.E.A. van Hintum and G.J. Reynolds, *Constraint objects*, Technical Report, T/COO S0, Esprit Project 6307 (MADE), 1993.

31. G. van Rossum, *Python Reference Manual*, Centrum voor Wiskunde en Informatica, Amsterdam, 1993.

32. P. Wayner, *Inside QuickTime*, BYTE, 1991.

Reusable Coordinator Modules for Massively Concurrent Applications

F. Arbab, C.L. Blom, F.J. Burger and C.T.H. Everaars
CWI, P.O. Box 94079, 1090 GB Amsterdam, The Netherlands

Abstract

Isolating computation and communication concerns into separate pure computation and pure coordination modules enhances modularity, understandability, and reusability of parallel and/or distributed software. MANIFOLD is a pure coordination language that encourages this separation. We use real, concrete, running MANIFOLD programs to demonstrate the concept of pure coordination modules and the advantage of their reuse in applications of different nature. Performance results for the examples presented in this paper show that the overhead of using MANIFOLD to achieve this enhanced modularity and reusability is in practice small, compared to the more conventional paradigms for the design and programming of parallel and distributed software.

1 Introduction

Some of the shortcomings of the common approaches to the design and development of parallel and distributed applications stem from the fundamental properties of the various models of communication used to construct this software[4, 5]. Without a proper programming paradigm for expressing the coordination of the cooperation of various active components that comprise a single concurrent application, programmers are forced to use low-level communication constructs, such as message passing, directly in their code. Because these primitives are generally scattered throughout the source code of the application and are typically intermixed with non-communication application code, the protocols of coordination generally never manifest themselves in a tangible form as easily identifiable pieces of source code. Thus, in spite of the fact that the coordination protocols are often the most complex and expensive-to-develop part of a non-trivial parallel or distributed application, they are not treated as a separate commodity that can be designed, developed, debugged, maintained, and reused, in isolation from the rest of the application code.

Intermixing communication concerns with computation decreases the comprehensibility, maintainability, and reusability of software modules. Moreover, the targeted send primitives used in message passing models of communication strengthen the dependence of individual processes on their environment. This too diminishes the

reusability and maintainability of processes[4]. It also complicates debugging and proving correctness of programs because a process that depends on the existence and certain expected "valid" behavior of some other processes for its own correctness, by itself is not a well encapsulated concept.

Coordination languages[22] ameliorate these problems to some extent. A variety of interesting models, languages, systems, and language extensions have been proposed for coordination. Among them are Interaction Abstract Machines[1] and Linear Objects[3]; shared data/tuple space models such as Linda[14], Gamma[10], and Bauhaus[15]; ToolBus[11]; extensions to enhance (various flavors of) the low-level message passing models by additional concepts such as synchronizers, contracts, constraints, rules, and events[20, 9, 23, 24, 2]; adaptations of Logic Programming languages such as PMS-Prolog[26], Multi-Prolog[16], Prolog-D-Linda[25], Shared Prolog[12], and most notably, Strand[19]. A review of these and other related work is out of the scope of this paper and appears in [4], [7], and [5]. Most existing coordination languages still do not go all the way to support reusable, pure coordination program modules.

Using a specific coordination model or language generally influences the architectural design and the program structure of a parallel/distributed application in significant ways[7]. The most well-known, well-established, and practically used coordination languages, e.g., Linda and its variants, induce a data-oriented approach to software design, which may or may not be a natural fit for a given application. Comparatively, less work has been reported on models and languages for coordination with specific focus on control-oriented applications. Most of such relevant work takes the message passing paradigm as its base and modifies it by introducing such additional notions as synchronizers, contracts, constraints, and events. A major drawback of these approaches to control-oriented coordination models and languages is that they cannot overcome the deficiencies that are inherent in their underlying message passing paradigm[7].

The goal of this paper is to demonstrate the concept of pure coordination modules and their reusability in different applications through real, concrete, running examples using the coordination language MANIFOLD, which is based on the IWIM model of communication. The IWIM model offers a paradigm for control-oriented coordination. Only a brief summary of the IWIM model is presented here in §2. An overview introduction to the MANIFOLD language appears in §3 in this paper. More detailed description of IWIM and MANIFOLD appear in [6] and [5]. An overview of an earlier version of the MANIFOLD language and its implementation was published in [8], which also contains a series of other examples. For our purpose in this paper, the syntax and semantics of some of the relevant constructs of MANIFOLD are introduced in §4 through a trivial example program. In §5, we discuss a quite non-trivial example of sorting and show how MANIFOLD encourages isolating communication and computation concerns into separate modules. The reusability of the pure coordination module developed for sorting is then demonstrated in §6, where the same MANIFOLD program is applied to coordinate a parallel/distributed numerical optimization application using a domain decomposition algorithm. All examples presented in this paper run, without any change to any source code, on a variety of parallel and/or distributed heterogeneous computing plat-

forms. Some performance results for the optimization example of §6 are summarized in §7 and compared to the results obtained for alternative non-MANIFOLD solutions to the same problem. We close this paper with a short conclusion in §8.

2 The IWIM Model of Communication

IWIM stands for *Idealized Worker Idealized Manager* and is a generic, abstract model of communication that supports the separation of responsibilities and encourages a weak dependence of workers (processes) on their environment[4, 5]. The IWIM model is described only in terms of its most significant characteristics. As such, it indeed defines not a concrete model of communication, but a family of such models. Various members in this family can have different significant characteristics, e.g., with regards to synchronous vs. asynchronous communication.

The basic concepts in the IWIM model are *processes*, *events*, *ports*, and *channels*. A process (instance) is a *black box* with well defined ports of connection through which it exchanges *units* of information with the other processes in its environment. A port is a named opening in the bounding walls of a process through which units of information are exchanged using standard I/O type primitives analogous to read and write. Without loss of generality, we assume that each port is used for the exchange of information in only one direction: either into (input port) or out of (output port) a process. We use the notation $p.i$ to refer to the port i of the process instance p.

The interconnections between the ports of processes are made through channels. A channel connects a (port of a) producer (process) to a (port of a) consumer (process). We write $p.o \rightarrow q.i$ to denote a channel connecting the port o of the producer process p to the port i of the consumer process q. The specific details of inter-process communication through channels depends on the types of the channels used in such communication. IWIM defines a synchronous channel type and four basic protocols for asynchronous channels (§2.1). A specific concrete model in the IWIM family can support any number of different channel types, each of which will be a subtype of one of the five basic channel types in IWIM.

Independent of the channels, there is an event mechanism for information exchange in IWIM. Events are broadcast by processes in their environment, each broadcast yielding an *event occurrence* for which the broadcasting process becomes its *event source*. In principle, any process in the environment of an application can pick up a broadcast event occurrence. In practice, usually only a few processes pick up occurrences of each event, because only they are *tuned in* to their sources. The concrete details of the broadcast mechanism, how observers tune in to event sources, and other details such as whether or not communication through events automatically synchronizes the source of an event occurrence with its observers, are all (intentionally) left unspecified in IWIM.

The IWIM model supports *anonymous communication*: in general, a process does not, and need not, know the identity of the processes with which it exchanges information[1].

[1] Of course, nothing prevents a process from divulging its own identity as (part of) the contents of a unit

This concept reduces the dependence of a process on its environment and makes processes more reusable.

A process in IWIM can be regarded as a worker process or a manager (or coordinator) process. The responsibility of a worker process is to perform a (computational) task. A worker process is not responsible for the communication that is necessary for it to obtain the proper input it requires to perform its task, nor is it responsible for the communication that is necessary to deliver the results it produces to their proper recipients. In general, *no process in* IWIM *is responsible for its own communication with other processes.* It is always the responsibility of a *manager process* to arrange for and to coordinate the necessary communications among a set of worker processes.

There is always a bottom layer of worker processes, called *atomic workers*, in an application. In the IWIM model, an application is built as a (dynamic) hierarchy of (worker and manager) processes on top of this layer. Note that a manager process may itself be considered as a worker process by another manager process.

2.1 Communication Channels

A channel is a communication link that carries a sequence of bits, grouped into (variable length) *units*. A channel represents a reliable, directed, and perhaps buffered, flow of information in time. Reliable means that the bits placed into a channel are guaranteed to flow through without loss, error, or duplication, with their order preserved. Directed means that there are always two identifiable ends in a channel: a *source* and a *sink*. Once a channel is established between a producer process and a consumer process, it operates autonomously and transfers the units from its source to its sink.

With no assumptions about the internal operation of the producer and the consumer of a channel C, it is possible for C to contain some pending units. Therefore, it may or may not be desirable for a channel to immediately disconnect itself from its source or its sink as soon as its connection at its opposite end is broken. The *pending units* of a channel C are those that have already been delivered to C by its producer, but not yet delivered by C to its consumer. The possibility of the existence of pending units in a channel gives it an identity of its own, independent of its producer and consumer; e.g., it makes it meaningful for a channel to remain connected at one of its ends, after it is disconnected from the other. Thus, two types of connection can be identified between a port and a channel: *keep*-type and *break*-type. A *break*-type connection between a port and a channel breaks automatically when the connection at the other end of the channel breaks. A *keep*-type connection, on the other hand, persists even after the connection at the other end of the channel breaks.

There are five different types of channels in the IWIM model, a synchronous channel type (S) and four asynchronous channel types (BB, BK, KB, and KK[2]):

1. S channel: There is never a pending unit in a channel of this type. It is meaning-

it produces through one of its ports to other processes.

[2]The letters "B" and "K" in the name of a channel type, respectively designate *break*- and *keep*-type connections to the ports at their corresponding ends of the asynchronous channels of that type.

less to talk about a channel of this type without a complete producer-consumer pair.

2. BB channel: A channel of this type is disconnected from either of its producer or consumer automatically, as soon as it is disconnected from the other.

3. BK channel: A channel of this type is disconnected from its producer automatically, as soon as it is disconnected from its consumer, but disconnection from its producer does not disconnect the channel from its consumer.

4. KB channel: A channel of this type is disconnected from its consumer automatically, as soon as it is disconnected from its producer, but disconnection from its consumer does not disconnect the channel from its producer.

5. KK channel: A channel of this type is not disconnected from either of its processes automatically, if it is disconnected from the other.

2.2 Primitives for Workers and Managers

There are two means of communication available to a process in IWIM: via its ports, and via events. The communication primitives that allow a process to exchange units through its ports are analogous to the traditional read and write I/O primitives. A process p can broadcast an event e to all other processes in its environment by *raising* that event. The identity of the event e together with the identity of the process p comprise the broadcast *event occurrence*. A process can also pick up event occurrences broadcast by other processes and react on them.

Any process in IWIM can raise an event, react on event occurrences it has detected, and read from and write to its own ports. Only manager processes can (dynamically) create process instances and channels, and (re)connect channels to the ports of (their own or other) processes; and that is all they can do: managers have no primitives to enable them to carry out any computation themselves.

Each manager process typically controls the communications among a (dynamic) number of process instances in a data-flow like network. The processes themselves are generally unaware of their patterns of communication, which may change in time, by the decisions of their respective manager processes.

3 The Manifold Coordination Language

In this section, we briefly introduce MANIFOLD: a coordination language for managing complex, dynamically changing interconnections among sets of independent, concurrent, cooperating processes[6], which is based on a concrete model in the IWIM family. Specifically, the MANIFOLD model is a concrete version of the IWIM model where:

1. Each of the basic concepts of process, event, port, and channel in IWIM corresponds to an explicit language construct.

2. All communication is asynchronous. Thus, there is no synchronous communication channel (type S in §2.1), and raising (broadcasting) and reacting to events do not synchronize the processes involved.

3. The separation of computation and communication concerns, i.e., the distinction between workers and managers, is more strongly enforced.

MANIFOLD is a strongly-typed, block-structured, event-driven language. The primary entities created and manipulated in a MANIFOLD program are processes, ports, events, and streams. The MANIFOLD system supports separate compilation. A MANIFOLD source file constitutes a program *module*, encapsulating all that is declared locally within its scope. A module can access entities defined in other modules by `importing` a definition that is `exported` by them, or share the ones declared as `extern`. Scope rules and the syntactic structure of programs are used in MANIFOLD to influence the broadcast of events and to determine which processes are tuned in to observe the event occurrences of which other processes.

A MANIFOLD application consists of a (potentially very large) number of (light- and/or heavy-weight) processes running on a network of heterogeneous hosts, some of which may be parallel systems. Processes in the same application may be written in different programming languages. Some of them may not know anything about MANIFOLD, nor the fact that they are cooperating with other processes through MANIFOLD in a concurrent application.

The MANIFOLD system consists of a compiler, a run-time system library, a number of utility programs, libraries of builtin and predefined processes[6], a link file generator called MLINK and a run-time configurator called CONFIG. Our current implementation of the MANIFOLD system uses PVM[21] and a number of different thread packages on various platforms. This implementation was developed with emphasis on portability and support for heterogeneity of the execution environment. It can be ported to any platform that supports a thread facility functionally equivalent to a small subset of the Posix threads, plus an inter-process communication facility roughly equivalent to a small subset of PVM. In our environment, it currently runs on SGI IRIX 5.3 and 6.2, SUN Solaris 5.2, and IBM SP/2.

MLINK uses the object files produced by the (MANIFOLD and other language) compilers to produce link files needed to compose the application executable files for each required platform. At the run time of an application, another utility program, CONFIG, consults a user-supplied host-configuration file to determine the actual host(s) on which the processes that are dynamically created in the MANIFOLD application will run. The library routines that comprise the interface between MANIFOLD and processes written in other languages (e.g. C), automatically perform the necessary data format conversions when data is routed between various different machines.

3.1 Processes

In MANIFOLD, the atomic workers of the IWIM model are called atomic processes. Any operating-system-level process can be used as an atomic process in MANIFOLD; such

processes are called *non-compliant* atomic processes. However, MANIFOLD also provides a library of functions that can be called from a regular C function running as an atomic process, to support a more appropriate interface between the atomic processes and the MANIFOLD world. Atomic processes that use this interface are called *compliant* atomic processes. A compliant atomic process can run as a separate operating-system-level process (heavy-weight process), or as a thread (light-weight process) within an operating-system-level process (called a *task*). Atomic processes can only produce and consume units through their ports, generate and receive events, and compute. In this way, the desired separation of computation and coordination is achieved.

Coordination processes are written in the MANIFOLD language and are called manifolds. A manifold definition consists of a header and a body. The header of a manifold gives its name, the number and types of its parameters, and the names of its input and output ports and their attributes (see §3.4). The body of a manifold definition is a block. A block consists of a finite number of states. Each state has a label and a body. The label of a state defines the condition under which a transition to that state is possible. It is an expression that can match observed event occurrences in the event memory of an instance of the manifold. The body of a simple state defines the set of actions that are to be performed upon transition to that state. The body of a compound state is either a (nested) block, or a call to a parameterized subprogram known as a *manner* in MANIFOLD. A manner consists of a header and a body. As for the subprograms in other languages, the header of a manner essentially defines its name and the types and the number of its parameters. A manner is either atomic or regular. The body of a regular manner is a block. The body of an atomic manner is a C function that can interface with the MANIFOLD world through the same interface library as for the compliant atomic processes.

3.2 Streams

All communication in MANIFOLD is asynchronous. In MANIFOLD, the asynchronous IWIM channels are called streams. A stream is a communication link that transports a sequence of bits, grouped into (variable length) units.

A stream represents a reliable and directed flow of information from its *source* to its *sink*. As in the IWIM model, the constructor of a stream between two processes is, in general, a third process. Once a stream is established between a producer process and a consumer process, it operates autonomously and transfers the units from its source to its sink. The sink of a stream requiring a unit is suspended only if no units are available in the stream. The suspended sink is resumed as soon as the next unit becomes available for its consumption. An attempt by the source of a stream to place a unit into the stream is never suspended because the infinite buffer capacity of the stream is never filled[3].

[3]The "infinite" capacity of a stream is not really as impractical as it seems. Once the volume of the units contained in a stream exceeds a threshold, its surplus can be diverted into a file. An application that generates so much buffered information to fill up the capacity of the file system either needs a larger file system or a bit of restructuring to introduce some synchronization over the buffered information.

There are four basic stream types designated as BB, BK, KB, and KK (corresponding to the four basic asynchronous channel types in IWIM), each behaving according to a slightly different protocol with regards to its automatic disconnection from its source or sink. Furthermore, in MANIFOLD, the BK and KB type streams can be declared to be *reconnectable*. A reconnectable BK type stream remains connected to its sink after it is disconnected from its source and allows its dangling end to be reconnected to another source. This way the output from several sources can be sent to a single sink with their order preserved. Likewise, a reconnectable KB type stream remains connected to its source after it is disconnected from its sink and allows its dangling end to be reconnected to another sink. This way the output from a single source can be distributed to several sinks in a desired order. (An example of the use of reconnectable KB streams appears in §5 in this paper.)

The connection between a stream and the port (of a process) it is connected to is severed when either the stream or the process to which the port belongs dies. Furthermore, a *break*-type connection between a stream and a port (which corresponds to the occurrence of the letter "B" in the name of its stream type) can be severed (1) by the stream, when it realizes that the connection at its other end is severed, and/or (2) by the manifold process that set up the connection. In the latter case, we say the stream is *preempted* by its constructor. Thus, preemption of a stream simply severs its *break*-type connections. See [6] or [5] for details.

3.3 Events and State Transitions

In MANIFOLD, once an event is *raised* by a process, it continues with its processing, while the resulting event occurrence propagates through the environment independently. By definition, a process x is interested in the events broadcast by another process y, if x is in a (syntactic) scope where it *knows* y. For x to know y, it must have a name for, reference to, or handle for y or one of the ports of y; e.g., y must be declared or created somewhere such that it is known in the current scope of x, y must be defined in a global scope, y or one of its ports must be passed as an actual parameter to x, or a reference to y or one of its ports must be received (as a data unit) and dereferenced by x, etc. The process x is then called an *observer* of (the events of) the process y.

Any observer process that is interested in an event occurrence will automatically receive it in its *event memory*. The observed event occurrences in the event memory of a process can be examined and reacted on by this process at its own leisure. In reaction to such an event occurrence, the observer process can make a transition from one labeled state to another.

The only control structure in the MANIFOLD language is an event-driven state transition mechanism. More familiar control structures, such as the sequential flow of control represented by the connective ";" (as in Pascal and C), conditional (i.e., "if") constructs, and loop constructs can be built out of this event mechanism, and are also available in the MANIFOLD language as convenience features[6].

Upon transition to a state, the primitive actions specified in its body are performed

atomically in some non-deterministic order. Then, the state becomes *preemptable*: if the conditions for transition to another state are satisfied, the current state is preempted, meaning that all streams that have been constructed are preempted and a transition to a new state takes place. The most important primitive actions in a simple state body are (1) creating and activating processes, (2) generating event occurrences, and (3) connecting streams to the ports of various processes.

3.4 Ports

A *port* is a regulated opening at the boundary of a process, through which the information produced and/or consumed by the process is exchanged with other processes. Regulated means that the information can flow in only one direction through a port: it either flows into or out of the process. The information exchanged between a process and other processes through its ports is quantized in discrete bundles called units. A *unit* is a packet containing an arbitrary number of bits that are produced, transferred, and consumed in an integral fashion; i.e., there are no partial units.

Ports are structural properties of processes and are defined and owned by them. Ports that belong to a particular process are called its *local* ports. Ports through which units flow into a process are called the *input* ports of the process. Similarly, ports through which units flow out of a process are called the *output* ports of the process. The standard ports `input`, `output`, and `error` are always defined for all process instances in MANIFOLD.

Each port has two distinct sides: an *arrival* side and a *departure* side. Normally, a process has exclusive access to the departure side of its own input ports and to the arrival side of its own output ports, while other processes have access only to the opposite sides of its ports. For this reason, the arrival sides of input ports and the departure sides of output ports are called the *public sides* of the ports, while the departure sides of input ports and the arrival sides of output ports are called their *private sides*. A process may allow other processes access to the private sides of its ports through parameter passing or sending out port reference units.

3.4.1 Port Connectivity

The type of the stream connections that can be made to either side of a port can be controlled by a certain attribute of the port, called its *connectivity*. The connectivity values defined in a manifold header for its local ports become attributes of the real ports of its process instances. This guarantees that any connection made to a port of an instance of this manifold will always have the desired connection type.

3.4.2 Port Conditions and Guards

The ports of a process are its only interface to the topology of the communication network of an application. Generally, this topology is changed dynamically by some manifold instances, without the direct involvement of the affected processes. The knowledge

Condition name	Type	Meaning
a_connect	transitory	Connection the arrival side.
a_connected	non-transitory	Connection on the arrival side.
a_disconnect	transitory	Disconnection on the arrival side.
a_disconnected	non-transitory	No connection on the arrival side.
a_everconnected	non-transitory	a_connected has been true.
a_everdisconnected	non-transitory	a_disconnected was/is true.
d_connect	transitory	Connection to the departure side.
d_connected	non-transitory	Connection on the departure side.
d_disconnect	transitory	Disconnection on the departure side.
d_disconnected	non-transitory	No connection on the departure side.
d_everconnected	non-transitory	d_connected has been true.
d_everdisconnected	non-transitory	d_disconnected was/is true.
full	non-transitory	Unit is available on the arrival side.
empty	non-transitory	Stream connect to the departure side.
transport	transitory	Unit passed through the port.

Table 1: Simple port conditions

of certain facts about this topology in the local neighborhood of a port is often crucial for the proper operation of the process to which this port belongs, and more generally, can also be of interest to some other processes that know this port. For example, a process may want to know if there is a stream connection to one of its ports, or be notified when such a connection is broken. Also, it is often "natural" for a process to interpret the disconnected status of one of its input ports after an initial connection, as its cue to wrap up its affairs and terminate, because its services are no longer required. For instance, some of the processes in the example described in §5 rely on this useful protocol for their termination.

Such useful facts about the status of a port and its stream connections are available in MANIFOLD through port conditions. A *port condition* is a predicate that is either true or false about a port. A port condition is either simple or composite (see below). Any process that has access to either side of a port can use the `guard` primitive action of MANIFOLD to receive a notification when a specified port condition becomes true for that port. The primitive action `guard`(*PortName*, *Condition*, *Event*) installs (i.e., creates and activates) a special virtual process on the specified port. This virtual process waits, if necessary, for the specified condition to become true. Once the condition is true, the guard virtual process posts the specified event in the event memory of its installer and terminates. An installed guard remains on its port either until it fires its event, or until it is removed by a subsequent guard installation by the same installer on the same port for the same condition.

A *composite port condition* is made out of simple port conditions using the binary sequencing operator `!`. A composite port condition of the form $c_1!c_2!...!c_n$ applied to a port p means wait, if necessary, until c_1 is true for p, then wait, if necessary, until c_2

is true for p, etc., and then wait, if necessary, until c_n is true for p; it is only then that the whole composite port condition becomes true for p.

The simple port conditions available in the MANIFOLD language are listed in Table 1. The three simple port conditions `full`, `empty`, and `transport` relate to the availability and flow of units through a port. The remaining simple port conditions relate to the incidents (the four `connect` and `disconnect` conditions), the status (the four `connected` and `disconnected` conditions), and the history (the four `everconnected` and `everdisconnected` conditions) of the connections and disconnections made to and from the two sides of a port. Simple port conditions are either transitory or non-transitory.

A *transitory* condition is one that can be detected only if it is expected, i.e., if there is already a `guard` installed that is waiting for the transitory condition to become true. In other words, the values of transitory conditions are not "remembered" by ports. Thus, if a `guard` is installed expecting a transitory condition after the condition has become true, the `guard` will not be able to detect that and will wait until the next time the condition becomes true.

A *non-transitory* condition is one whose value is remembered by the port. This means that if a non-transitory condition is already true at the time that a `guard` expecting it is installed, the `guard` will recognize the condition to be true.

4 Hello World!

For our first example, consider a simple program to print a message such as "Hello World!" on the standard output. The MANIFOLD source file for this program contains the following:

```
1 manifold PrintUnits import.
2
3 auto process print is PrintUnits.
4
5 manifold Main
6 {
7   begin: "Hello World!" -> print.
8 }
```

The first line of this code defines a manifold named `PrintUnits` that takes no arguments, and states (through the keyword `import`) that the real definition of its body is contained in another source file. This defines the "interface" to a process type definition, whose actual "implementation" is given elsewhere. Whether the actual implementation of this process is an atomic process (e.g., a C function) or it is itself another manifold is indeed irrelevant in this source file. We assume that `PrintUnits` waits to receive units through its standard input port and prints them. When `PrintUnits` detects that there are no incoming streams left connected to its input port and it is done printing the units it has received, it terminates.

The second line of code defines a new instance of the manifold `PrintUnits`, calls it `print`, and states (through the keyword `auto`) that this process instance is

to be automatically activated upon creation, and deactivated upon departure from the scope wherein it is defined; in this case, this is the end of the application. Because the declaration of the process instance `print` appears outside of any blocks in this source file, it is a global process, known by every instance of every manifold whose body is defined in this source file.

The last lines of this code define a manifold named `Main` that takes no parameters. Every manifold definition (and therefore every process instance) always has at least three default ports: `input`, `output`, and `error`. The definition of these ports are not shown in this example, but the ports are defined for `Main` by default.

The body of this manifold is a block (enclosed in a pair of braces) and contains only a single state. The name `Main` is indeed special in MANIFOLD: there must be a manifold with that name in every MANIFOLD application and an automatically created instance of this manifold, called `main`, is the first process that is started up in an application. Activation of a manifold instance automatically posts an occurrence of the special event `begin` in the event memory of that process instance; in this case, `main`. This makes the initial state transition possible: `main` enters its only state – the `begin` state.

The `begin` state contains only a single primitive action, represented by the stream construction symbol, "→". Entering this state, `main` creates a stream instance (with the default BK-type) and connects the `output` port of the process instance on the left-hand side of the → to the `input` port of the process instance on its right-hand side. The process instance on the right-hand side of the → is, of course, `print`. What appears to be a character string constant on the left-hand side of the → is also a process instance: conceptually, a constant in MANIFOLD is a special process instance that produces its value as a unit on its `output` port and then dies.

Having made the stream connection between the two processes, `main` now waits for all stream connection made in this state to break up (on at least one of their ends). The stream breaks up, in this case, on its source end as soon as the string constant delivers its unit to the stream and dies. Since there are no other event occurrences in the event memory of `main`, the default transition for a state reaching its end (i.e., falling over its terminator period) now terminates the process `main`.

Meanwhile, `print` reads the unit and prints it. The stream type BK ensures that the connection between the stream and its sink is preserved even after a preemption, or its disconnection from its source. Once the stream is empty and it is disconnected from its source, it automatically disconnects from its sink. Now, `print` senses that it has no more incoming streams and dies. At this point, there are no other process instances left and the application terminates.

Figure 1: The "Hello World" example in Manifold

Note that our simple example, here, consists of three process instances: two worker

processes, a character string constant and `print`, and a coordinator process, `main`. Figure 1 shows the relationship between the constant and `print`, as established by `main`. Note also that the coordinator process `main` only establishes the connection between the two worker processes. It does *not* transfer the units through the stream(s) it creates, nor does it interfere with the activities of the worker processes in other ways.

5 Bucket Sort

The example in the previous section was simple enough to require only a static pattern of communication. In this section, we illustrate the dynamic capabilities of MANIFOLD through a program for sorting an unspecified number of input units. The particular algorithm used in this example is not necessarily the most effective one. However, it is simple to describe, and serves our purpose of demonstrating the dynamic aspects of the MANIFOLD language well. The sort algorithm is as follows.

There is a sufficiently large (theoretically, infinite) number of *atomic sorters* available, where each atomic sorter a_i is able to sort a bucket of $k_i > 0$ units very efficiently. (The number k_i may vary from one atomic sorter to the next in an "unpredictable" way; e.g., each atomic sorter can internally decide for itself how many units it is willing to sort, taking into consideration such "unpredictable" and (ir)relevant factors as the load of the system, the phase of the moon, etc.) Each atomic sorter receives its input through its `input` port; raises a specific event it receives as a parameter to inform other processes that it has filled up its input bucket; sorts its units; produces the sorted sequence of the units through its `output` port; and terminates.

The parallel bucket sort program is supposed to feed as many of its own input units to an atomic sorter as the latter can take; feed the rest of its own input as the input to another copy of itself; merge the two output sequences (of the atomic sorter and its new copy); and produce the resulting sequence through its own `output` port. Merging of the two sorted sequences can be done by a separate merger process, or by a subprogram (i.e., a manner) called by the sorter.

We assume our application consists of several source files. The first source file contains our `Main` manifold, as shown below. We assume that the merger is a separate process. The merger and the atomic sorter can be written in the MANIFOLD language, but they will be more efficient if they are written in a computation language, such as C. We do not concern ourselves here with the details of the merger and the atomic sorter, and assume that each is defined in a separate source file.

```
1 manifold printunits import.
2 manifold ReadFile(process filename) atomic {internal.}.
3 manifold Sorter import.
4 manifold AtomicSorter(event) atomic {internal.}.
5 manifold AtomicIntMerger port in a, b. atomic {internal.}.
6
7 /*********************************************************************/
8 manifold Main
9.{
10   auto process read is ReadFile("unsorted").
11   auto process sort is Sorter.
```

```
12    auto process print is printunits.
13
14    begin:    read -> sort -> print.
15 )
```

The main manifold (instance) in this application creates read, sort, and print as instances of manifold definitions ReadFile, Sorter, and PrintUnits, respectively. It then connects the output port of read to the input port of sort, and the output port of sort to the input port of print. The process main terminates when both of these connections are broken.

The process read is expected to read the contents of the file named unsorted and produce a unit for every sort item in this file through its output port. When it is through with producing its units, read simply terminates. The process sort is an instance of the manifold definition Sorter, which is expected to sort the units it receives through its input port. This process terminates when its input is disconnected and all of its output units are delivered through its output port.

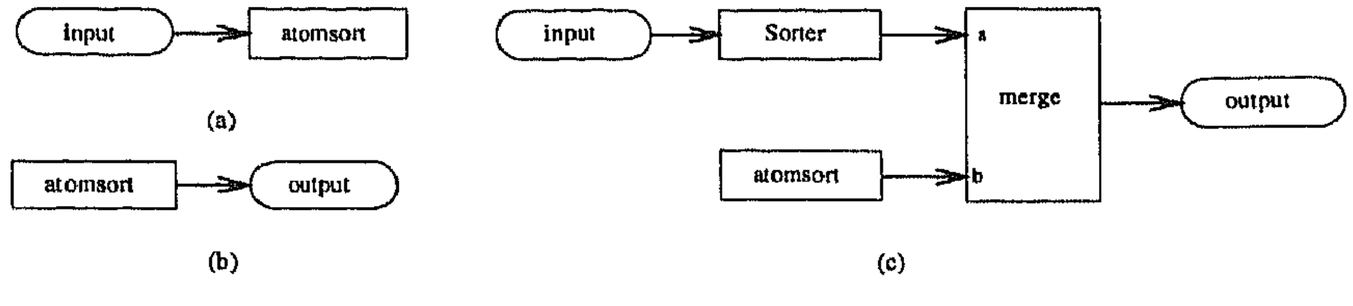


Figure 2: Bucket sort

The manifold definition Sorter, shown below, is our main interest. The keyword export on line 1 allows other separately compiled MANIFOLD source files to import and use this coordinator manifold. In its begin state, an instance of Sorter connects its own input to an instance of the AtomicSorter, it calls atomsort. It also installs a guard on of its input port using the guard primitive action. This guard posts the event finished if it has an empty stream connected to its departure side, after the arrival side of this port has no more stream connections, following a first connection. This means that the event finished is posted in an instance of Sorter after a first connection to the arrival side of its input is made, then all connections to the arrival side of its input are severed, and all units passed through this port are consumed. The connections in this state are shown in Figure 2.a.

```
1 export manifold Sorter()
2 (
3   event filled, flushed, finished.
4   process atomsort is AtomicSorter(filled).
5   stream reconnect KB input -> *.
6   priority filled < finished.
7
8   begin: (
9     activate(atomsort), input -> atomsort,
10    guard(input, a_everdisconnected!empty, finished) // no more input
11  ).
12
```

```
13    finished: {
14      ignore filled. //possible event form atomsort
15
16      begin: atomsort -> output  //your output is only that of atomsort
17    ).
18
19    filled: {
20      process merge<a, b | output> is AtomicIntMerger.
21      stream KK * -> (merge.a, merge.b).
22      stream KK merge -> output.
23
24      begin: (
25        activate(merge),
26        input -> Sorter -> merge.a,
27        atomsort -> merge.b,
28        merge -> output
29      ).
30
31      end | finished:.
32    ).
33
34    end: {
35      begin: (
36        guard(output, a_disconnected, flushed), // ensure flushing
37        terminated(void) //wait for units to flush through output
38      ).
39
40      flushed: halt.
41    ).
42  }
```

Two events can preempt the `begin` state of an instance of `Sorter`: (1) if the incoming stream connected to `input` is disconnected (no more incoming units) and `atomsort` reads all units available in its incoming stream, the guard on `input` posts the event `finished`; and (2) the process `atomsort` can read its fill and raise the event `filled`. Normally, only one of these events occurs; however, when the number of input units is exactly equal to the bucket size, k, of `atomsort`, both `finished` and `filled` can occur simultaneously. In this case, the `priority` statement on line 6 makes sure that the handling of `finished` takes precedence over `filled`.

Assume that the number of units in the input supplied to an instance of `Sorter` is indeed less than or equal to the bucket size k of an atomic sorter. In this case, the event `finished` will preempt the `begin` state and cause a transition to its corresponding state in `Sorter`. In this state, we ignore the occurrence of `filled` that may have been raised by `atomsort` (if the number of input units is equal to the bucket size k); and deliver the output of `atomsort` as the output of the `Sorter`. The connections in this state are shown in Figure 2.b.

Now suppose the number of units in the input supplied to an instance of `Sorter` is greater than the bucket size k of an atomic sorter. In this case, the event `filled` will preempt the `begin` state and cause a transition to its corresponding state in `Sorter`. In this state we create an instance of the merger process, called `merge`. A new instance of the `Sorter` is created in the `begin` state of the nested block. The rest of the input is passed on as the input to this new `Sorter`, and its output is merged with the output of the atomic sorter and the result is passed as the output of the `Sorter` itself. The connections in this state are shown in Figure 2.c. An occurrence of `finished` in this

state preempts the connected streams and causes a transition to the local `finish` state in this block. This preemption is necessary to inform the new instance of `Sort` (by breaking the stream that connects `input` to it) that it has no more input to recei so that it can terminate. The empty body of the `finished` state means that it caus an exit from its containing block.

In the `end` state, a `Sorter` instance installs a guard on its `output` port, to p the event `flushed` after there is no stream connected to the arrival side of this port f lowing its first connection. This means that the event `flushed` is posted in an instan of `Sorter` after a connection is made to its arrival side, and all units arriving at t port have passed through. The `Sorter` instance then waits for the termination of t special predefined process `void`, which will never happen (the special process `vo` never terminates). This effectively causes the `Sorter` instance to hang indefinite The only event that can terminate this indefinite wait is an occurrence of `flush` which indicates there are no more units pending to go through the `output` port of t `Sorter` instance.

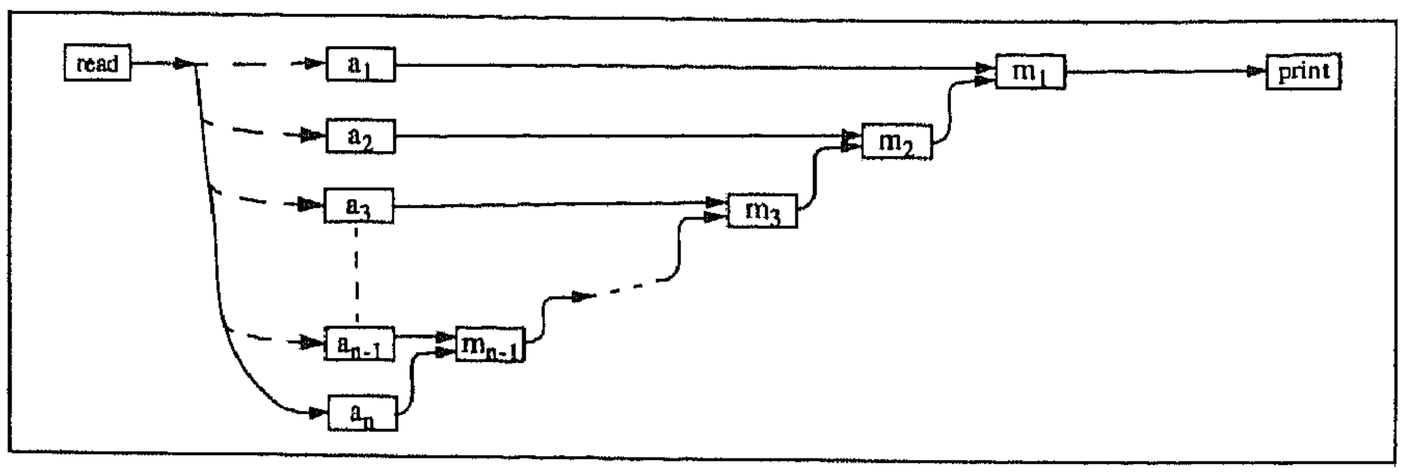


Figure 3: The atomic processes at work

Figure 3 shows the connections among the various instances of `AtomicSort` and `AtomicIntMerger` during an execution of our sort program. Each a_i and m denote the i^{th} instance of the `AtomicSorter` and `AtomicIntMerger`, respe tively. Note the dynamic way in which the reconnectable `KB` stream emanating fro `read` cascades its output to the various atomic sorters, allowing each of them to f up its bucket with as many input units as it decides for itself. This reconnection of t same stream to successive processes is depicted by the dashed arrows in Figure 3. Wh is not shown in this figure is the n instances of the `Sorter` manifold. For $0 < i <$ the i^{th} instance of `Sorter` coordinates the i^{th} instance of `AtomicSorter`, a_i, t i^{th} instance of `AtomicIntMerger`, m_i, and the $i + 1^{st}$ instance of `Sorter`. The nal (i.e., n^{th}) instance of `Sorter` is a degenerate case and has nothing to "coordinat but the n^{th} instance of `AtomicSorter`, a_n.

An interesting aspect of the `Sorter` manifold is the dynamic way in which switches connections among the process instances it creates. Perhaps more interestin is the fact that, in spite of its name, `Sorter` knows nothing about sorting! If change its name to X, and systematically change the names of the identifiers it uses

Y_1 through Y_k, we realize that all it knows is to divert its own input to an instance of some process it creates; when this instance raises a certain event, it is to divert the rest of its input to a new instance of itself; and to divert the output of these two processes to a third process, whose output is to be passed out as its own output.

What `Sorter` embodies is a protocol that describes how instances of two process definitions (e.g., `AtomicSorter` and `AtomicIntMerger` in our case) should communicate with each other. Our `Sorter` manifold can just as happily orchestrate the cooperation of any pair of processes that have the same input/output and event behavior as `AtomicSorter` and `AtomicIntMerger` do, regardless of what computation they perform. The cooperation protocol defined by `Sorter` simply doles out chunks of its input stream to instances of what it knows as `AtomicSorter` and diverts their output streams to instances of what it knows as `AtomicIntMerger`. What is called `AtomicSorter` needs not really sort its input units, the process called `AtomicIntMerger` needs not really merge them, and neither has to produce as many units through its `output` as it receives through its `input` port. They can do any computation they want.

By parameterizing the names of the manifolds used in `Sorter` and changing its name to `ProtocolX`, we obtain a more general program:

```
1 export manifold ProtocolX(manifold M1(event), manifold M2<a, b | output>)
2 {
3   event filled, flushed, finished.
4   process m1 is M1(filled).
5   stream reconnect KB input -> *.
6   priority filled < finished.
7
8   begin: (
9     activate(m1), input -> m1,
10    guard(input, a_everdisconnected!empty, finished) // no more input
11  ).
12
13  finished: {
14    ignore filled. //possible event form m1
15
16    begin: m1 -> output  //your output is only that of m1
17  }.
18
19  filled: {
20    process m2<a, b | output> is M2.
21    stream KK * -> (m2.a, m2.b).
22    stream KK m2 -> output.
23
24    begin: (
25      activate(m2),
26      input -> ProtocolX(M1, M2) -> m2.a,
27      m1 -> m2.b,
28      m2 -> output
29    ).
30
31    end | finished:.
32  ).
33
34  end: {
35    begin: (
36      guard(output, a_disconnected, flushed), // ensure flushing
37      terminated(void) //wait for units to flush through output
38    ).
```

```
39
40      flushed: halt.
41    }.
42 }
```

The new version of our bucket sort main program using `ProtocolX` is:

```
 1 manifold printunits import.
 2 manifold ProtocolX(manifold M1(event), manifold M2) import.
 3 manifold ReadFile(process filename) atomic {internal.}.
 4 manifold AtomicSorter(event) atomic {internal.}.
 5 manifold AtomicIntMerger port in a, b. atomic {internal.}.
 6
 7 /*****************************************************************/
 8 manifold Main
 9 {
10   auto process read is ReadFile("unsorted").
11   auto process sort is ProtocolX(AtomicSorter, AtomicIntMerger).
12   auto process print is printunits.
13
14   begin:   read -> sort -> print.
15 }
```

As a concrete demonstration of the reusability of coordinator modules, in the next section, we present an example that uses the coordinator `ProtocolX` in a numerical optimization problem.

6 Domain Decomposition

Consider the following optimization problem:

$$\max z = x^2 + y^2 - 0.5 * \cos(18 * x) - 0.5 * \cos(18 * y) \text{ with (x, y) } \in \text{ [-1.0, 1.0]} \quad (1)$$

Figure 4 shows the landscape formed by this function on its domain.

Analytical solutions to such problems are, in general, non-existent and domain decomposition is a common search technique used to solve them through numerical methods. Domain decomposition imposes a grid on the domain of the function, splitting it into a number of sub-domains, as determined by the size of the grid. Next, we obtain a (number of) good rough estimate(s) for the highest value of z in each sub-domain. Then, we select the sub-domains with the most promising z values and decompose them into smaller sub-domains. New estimates for the highest value of z in each of these sub-domains, recursively narrow this search process further and further into smaller and smaller regions that (hopefully) tend towards the area with the real maximum z, while the estimates for the obtained maximum z values become more and more accurate. In single grid domain decomposition, the same grid is imposed on all successive sub-domains. Multiple-grid domain decomposition techniques allow a different grid for each sub-domain, whose granularity and other properties may depend on the attributes of the sub-domain and those of the function within that region.

For our example, we consider a single grid method. Extension of this example for multiple-grid domain decomposition is straight-forward and involves only a small

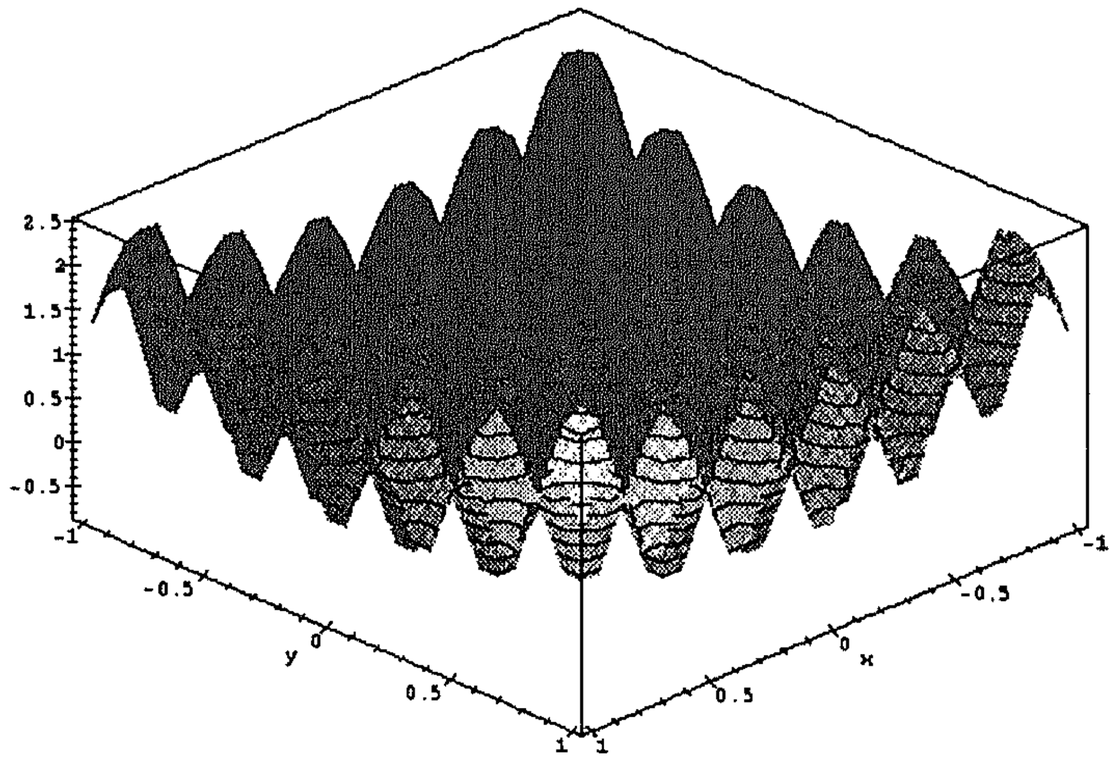


Figure 4: The function $z = x^2 + y^2 - 0.5 * \cos(18 * x) - 0.5 * \cos(18 * y)$

change to the main program of this application[17]. We need four simple computation modules for our current example: `ap_printobjects`, `Split`, `AtomicEval`, and `AtomicObjMerger`. An instance of `ap_printobjects` simply prints the units it reads from its `input`, each of which describes a (sub-)domain and the x, y, and z values for the estimated maximum z value in that (sub-)domain. An instance of `Split` receives as its parameters the specification of a grid (in our case, 6×6). Next, it reads from its `input` port a unit that describes a (sub-)domain, produces units on its `output` port that describe the sub-domains obtained by imposing the grid on this input domain, and terminates. The i^{th} instance of `AtomicEval` reads a bucket of $k_i > 0$ sub-domains (for simplicity, let $k_i = 1$ for all i) from its `input` port and raises a specific event, which it receives as a parameter, to inform other processes that it has filled up its input bucket with some sub-domains descriptions. It then finds the best estimate for the optimum z value in each of its sub-domains, producing an ordered sequence of units describing the best solutions it has found through its `output` port, and terminates. In our example, we use sampling: we simply evaluate z for a number of (say 1000) sample points in each sub-domain and consider the sample point with the maximum z as the best estimate for that sub-domain. An instance of `AtomicObjMerger` reads from its ports `a` and `b` two ordered sequences of units describing sub-domains and their best estimates, and produces a sequence of one or more of its best sub-domains on its `output` port.

We need a MANIFOLD program, say *Eval*, to coordinate the cooperation of the instances of `AtomicEval` and `AtomicObjMerger` to solve our optimization problem

in a parallel/distributed fashion. *Eval* receives through its `input` port units describing (sub-)domains. It is supposed to feed as many of its own input units to an atomic evaluator as the latter can take; feed the rest of its own input as the input to another copy of itself; merge the two output sequences (of the atomic evaluator and its new copy); and produce the resulting sequence through its own `output` port. The similarity between the description of *Eval* and that of `Sort` in §5 suggests that we can use the same coordination module for our optimization problem. Indeed, *Eval* is merely a version of `ProtocolX` with `AtomicEval` and `AtomicObjMerger` as its parameters. The following MANIFOLD program shows a single iteration of our domain decomposition application using the separately compiled `ProtocolX` of §5.

```
 1 manifold ap_printobjects atomic {internal.}.
 2 manifold ProtocolX(manifold M1(event), manifold M2) import.
 3 manifold Split(port in, port in) atomic {internal.}.
 4 manifold AtomicEval(event) atomic {internal.}.
 5 manifold AtomicObjMerger port in a, b. atomic {internal.}.
 6
 7 /*****************************************************************/
 8 manifold Main
 9 {
10   auto process split is Split(6, 6).
11   auto process eval is ProtocolX(AtomicEval, AtomicObjMerger).
12   auto process print is ap_printobjects.
13
14   begin:  <<1, -1.0, -1.0, 1.0, 1.0>> -> split -> eval -> print.
15 }
```

The output of this program, below, shows the result produced by 36 instances of `AtomicEval`, each taking in the description of a single sub-domain. The top four lines show the best estimates to be in the neighborhoods of the four corners of the domain for our symmetric function in Figure 4.

```
domain = (-1.000, -1.000) (-0.667, -0.667) point = (-0.883, -0.880), z =  2.541
domain = ( 0.667,  0.667) ( 1.000,  1.000) point = ( 0.889,  0.884), z =  2.540
domain = ( 0.667, -1.000) ( 1.000, -0.667) point = ( 0.881, -0.889), z =  2.539
domain = (-1.000,  0.667) (-0.667,  1.000) point = (-0.878,  0.881), z =  2.539
domain = (-0.667,  0.667) (-0.333,  1.000) point = (-0.528,  0.884), z =  2.048
domain = ( 0.333, -1.000) ( 0.667, -0.667) point = ( 0.533, -0.882), z =  2.048
domain = ( 0.667, -0.667) ( 1.000, -0.333) point = ( 0.885, -0.527), z =  2.048
domain = (-0.667, -1.000) (-0.333, -0.667) point = (-0.534, -0.885), z =  2.047
domain = (-1.000, -0.667) (-0.667, -0.333) point = (-0.881, -0.535), z =  2.047
domain = (-1.000,  0.333) (-0.667,  0.667) point = (-0.883,  0.536), z =  2.046
domain = ( 0.667,  0.333) ( 1.000,  0.667) point = ( 0.877,  0.535), z =  2.043
domain = ( 0.333,  0.667) ( 0.667,  1.000) point = ( 0.537,  0.878), z =  2.043
domain = ( 0.000, -1.000) ( 0.333, -0.667) point = ( 0.177, -0.885), z =  1.802
domain = (-1.000,  0.000) (-0.667,  0.333) point = (-0.885,  0.173), z =  1.801
domain = ( 0.667,  0.000) ( 1.000,  0.333) point = ( 0.881,  0.181), z =  1.800
domain = ( 0.000,  0.667) ( 0.333,  1.000) point = ( 0.171,  0.883), z =  1.799
domain = (-0.333,  0.667) ( 0.000,  1.000) point = (-0.183,  0.884), z =  1.798
domain = (-1.000, -0.333) (-0.667,  0.000) point = (-0.876, -0.175), z =  1.798
domain = (-0.333, -1.000) ( 0.000, -0.667) point = (-0.169, -0.885), z =  1.797
domain = ( 0.667, -0.333) ( 1.000,  0.000) point = ( 0.875, -0.174), z =  1.796
domain = ( 0.333,  0.333) ( 0.667,  0.667) point = ( 0.530,  0.531), z =  1.555
domain = ( 0.333, -0.667) ( 0.667, -0.333) point = ( 0.528, -0.529), z =  1.555
domain = (-0.667,  0.333) (-0.333,  0.667) point = (-0.532,  0.531), z =  1.555
domain = (-0.667, -0.667) (-0.333, -0.333) point = (-0.521, -0.534), z =  1.548
domain = (-0.667, -0.333) (-0.333,  0.000) point = (-0.533, -0.179), z =  1.307
domain = ( 0.333, -0.333) ( 0.667,  0.000) point = ( 0.527, -0.178), z =  1.307
```

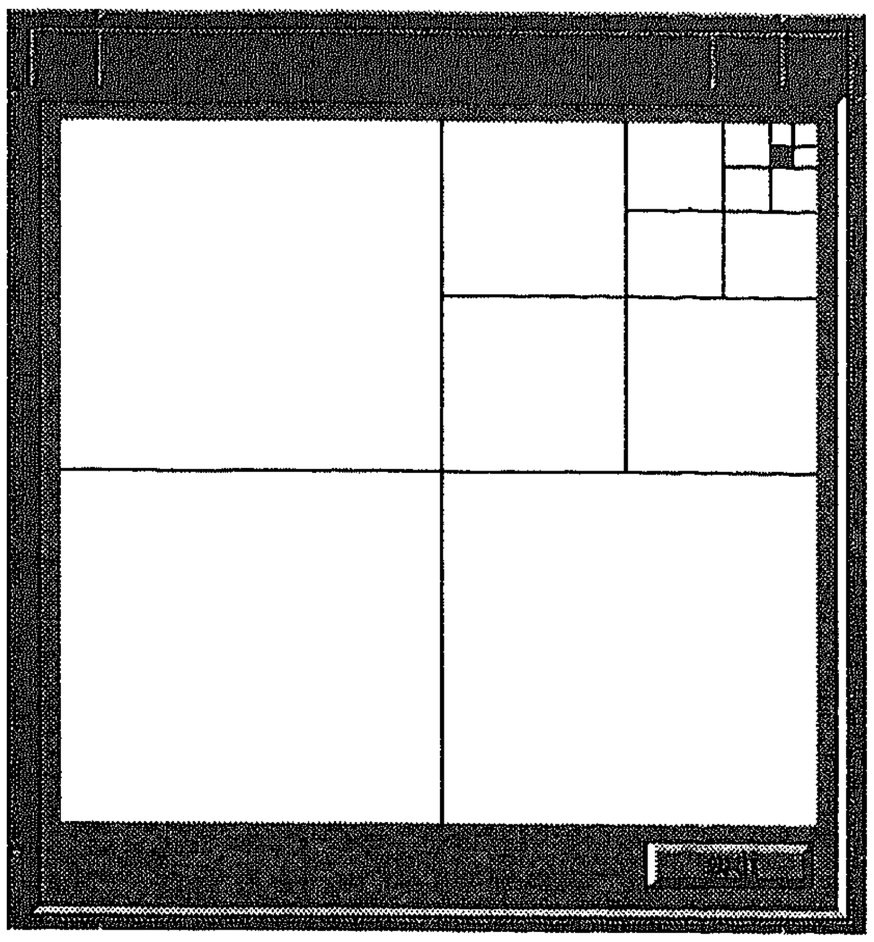


Figure 5: Visualizer snap-shot of 2×2 distributed domain decomposition

```
domain = (-0.333, -0.667) ( 0.000, -0.333) point = (-0.172, -0.531), z =  1.307
domain = (-0.667,  0.000) (-0.333,  0.333) point = (-0.532,  0.181), z =  1.307
domain = (-0.333,  0.333) ( 0.000,  0.667) point = (-0.180,  0.534), z =  1.306
domain = ( 0.000, -0.667) ( 0.333, -0.333) point = ( 0.177, -0.524), z =  1.305
domain = ( 0.333,  0.000) ( 0.667,  0.333) point = ( 0.537,  0.176), z =  1.304
domain = ( 0.000,  0.333) ( 0.333,  0.667) point = ( 0.164,  0.528), z =  1.295
domain = ( 0.000, -0.333) ( 0.333,  0.000) point = ( 0.175, -0.174), z =  1.061
domain = (-0.333, -0.333) ( 0.000,  0.000) point = (-0.179, -0.172), z =  1.059
domain = (-0.333,  0.000) ( 0.000,  0.333) point = (-0.177,  0.183), z =  1.058
domain = ( 0.000,  0.000) ( 0.333,  0.333) point = ( 0.171,  0.182), z =  1.057
```

A straight-forward generalization of this program repeats this single step until a termination criterion (such as a maximum number of iterations, or the diminishing of improvements below a threshold) is reached. Each iteration selects a (few of the) best sub-domain(s) found so far as input to another instance of `Split` and *Eval*. This would be yet another MANIFOLD program that coordinates the cooperation of different instances of *Eval* and `Split`. The following output is produced by such a program using a 2×2 grid. The first line in this output is our initial input unit representing the whole domain. Each succeeding group of four lines then represents one iteration. The best sub-domain found in each iteration is fed as input to the next iteration. The first line of the last group (representing the fifth iteration) shows the best solution found ($z = 2.542$) which is slightly better than the best solution we found using our single step 6×6 grid ($z = 2.541$).

```
domain = (-1.000, -1.000) ( 1.000,  1.000)
```

```
domain = ( 0.000,  0.000) ( 1.000,  1.000) point = ( 0.885,  0.879), z =   2.540
domain = (-1.000, -1.000) ( 0.000,  0.000) point = (-0.884, -0.890), z =   2.539
domain = ( 0.000, -1.000) ( 1.000,  0.000) point = ( 0.890, -0.893), z =   2.532
domain = (-1.000,  0.000) ( 0.000,  1.000) point = (-0.880,  0.911), z =   2.484

domain = ( 0.500,  0.500) ( 1.000,  1.000) point = ( 0.879,  0.892), z =   2.536
domain = ( 0.000,  0.500) ( 0.500,  1.000) point = ( 0.498,  0.866), z =   1.941
domain = ( 0.500,  0.000) ( 1.000,  0.500) point = ( 0.880,  0.490), z =   1.920
domain = ( 0.000,  0.000) ( 0.500,  0.500) point = ( 0.498,  0.499), z =   1.400

                                    :

domain = ( 0.875,  0.875) ( 0.938,  0.938) point = ( 0.884,  0.885), z =   2.542
domain = ( 0.875,  0.938) ( 0.938,  1.000) point = ( 0.884,  0.938), z =   2.346
domain = ( 0.938,  0.875) ( 1.000,  0.938) point = ( 0.938,  0.883), z =   2.344
domain = ( 0.938,  0.938) ( 1.000,  1.000) point = ( 0.940,  0.938), z =   2.134
```

The highly modular structure of this application is remarkable. Its computation modules (C functions) are simple and have no idea of how they relate to or cooperate with one another. The coordination module *Eval* knows nothing about what these computation modules actually do; it is just as happy coordinating the sorter workers in §5 as it is managing these numerical optimization workers. The various processes comprising this application can run on parallel or distributed platforms without any change to their source code.

The plumbing paradigm of MANIFOLD makes it easy to divert the flows of units, change the coordination structures, and dynamically modify the topology of communication links among (computation as well as coordination) modules to adapt an application to new requirements. We can plug in graphics modules to display an on-going computation. Indeed, we have a small computational steering environment built around this example, using MANIFOLD coordinators and a few generic graphics interaction modules[17]. The user interface for this program graphically shows the on-going activity of various atomic evaluators (that may be running on different hosts) and allows the user to interactively direct the focus of the attention of the program onto one or more areas of interest, simply by drawing a box to designate a sub-domain.

Figure 5 shows a snap-shot of a simple visualizer interface as our 2×2 distributed domain decomposition optimization application moved on (in its fifth iteration) to the top-right corner sub-domain in the run that produced the above output. Our computational steering interface includes the capabilities of this simple visualizer and also allows a user to select a domain by mouse (by drawing a rectangle) and start (by pressing mouse buttons) the recursive decomposition of that domain. Figure 6 shows such an interaction whereby a user steers a multiple-grid domain decomposition computation using the same MANIFOLD program, working on the function shown in Figure 4.

7 Performance Evaluation

It is difficult to produce meaningful numerical measures to fairly compare the performance of a MANIFOLD application with more conventional parallel/distributed versions of the same application. This is so because one of the primary reasons for using

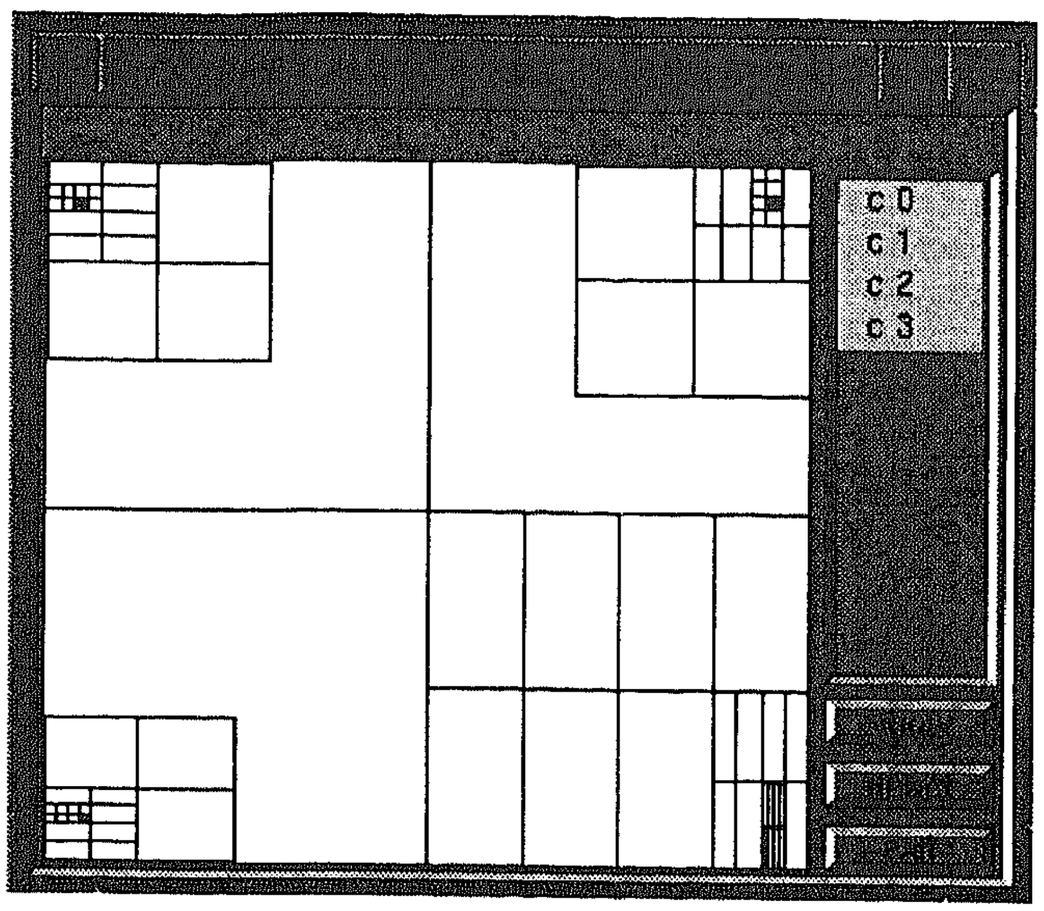


Figure 6: Snap-shot of multiple-grid decomposition of interactively defined domains

MANIFOLD is to increase the modularity and reusability of the software components in and across applications. In fact, the (positive or negative) *overhead* in a MANIFOLD application, compared with any other incarnation of the "same" application, can be classified into two categories:

1. The overhead incurred by designing and organizing the software in a style that separates computation and coordination components into smaller, more reusable modules; and

2. The overhead involved in the realization of this design using the MANIFOLD language, and its execution on the existing MANIFOLD run-time environment.

Accepting, for the moment, the premise that isolating computation and coordination concerns into separate software modules is desirable, it is not difficult to program a given application in this "style" using various tools, including MANIFOLD. In this case, the first component of the overhead will be fixed and one can then fairly compare the performance of MANIFOLD with that of other tools. Such measures are very useful, e.g., for the implementors of MANIFOLD by indicating how additional optimization can improve the performance of their implementation. On the other hand, such performance measures are likely to be rather meaningless for end users.

Comparing the performance of a MANIFOLD application with that of the "same" application done in a different, more conventional "style" using another parallel/distributed

programming tool may be unfair (generally against MANIFOLD), but we feel such measures are more meaningful to end users. Given that the goal of this paper is to advocate a certain style of software design that results in reusable pure coordination modules (as well as reusable pure computation modules), it is our responsibility to show that the price of the enhanced modularity gained through this style is not a heavy loss of run-time efficiency.

Thus, we set to carry out a number of experiments to compare the performance of the domain decomposition program described in §6 against two other versions of the "same" application: a sequential version and one done in PVM[21]. All three versions of this application use the exact same computation code, the same domain, the same 6×6 grid, and the same function. Each version, though, is written in a different style – one that reflects a straight-forward implementation of the application by a reasonable programmer, in a way that fits "naturally" in its respective underlying paradigm.

The sequential version contains a simple loop that calls an evaluator function to evaluate the maximum of the function in each of the 36 sub-domains and then chooses the best result. This is *not* the most effective sequential algorithm for evaluating a function in a domain. Nevertheless, it serves the purpose of providing a meaningful reference base for the other two versions.

The PVM version consists of a main task, a splitter task, and 36 evaluator tasks. The main task spawns off the splitter, and then waits to receive the results coming from the 36 evaluators, before it selects the best result and reports it. The splitter task imposes the 6×6 grid and spawns off the 36 evaluators, informs them that they must report their results to the main task, and terminates. Each evaluator works on a sub-domain and reports the local maximum of the function in that domain to the main task. All coordination and communication necessary for this version of the application are directly coded as PVM calls in program modules, because this is how a reasonable PVM programmer would do it.

Compared to the MANIFOLD version of the application, described in §6, there are roughly three times fewer (user) processes in the PVM version than in the MANIFOLD version.[4] The bulk of this extra number of processes is actually the "overhead of style" rather than the "overhead of implementation" of the application in MANIFOLD, or the overhead of the implementation of the MANIFOLD system itself. Fortunately, our implementation of MANIFOLD uses threads and thus considerably reduces the overhead of having so many processes in an application. Nevertheless, by ignoring the distinction between the two categories of overhead, above, we know that we are being unfair to the MANIFOLD version and effectively end up comparing apples and oranges.

All experiments were run on an SGI Challenge L with four 200 MHZ IP19 processors, each with a MIPS R4400 processor chip as CPU and a MIPS R4010 floating point chip for FPU. This 32-bit machine has 256 megabytes of main memory, 16 kilobytes of instruction cache, 16 kilobytes of data cache, and 4 megabytes of secondary unified instruction/data cache. This machine runs under IRIX 5.3, is on a network, and is used

[4] In fact the MANIFOLD version has more than three times as many processes, considering the ones that are needed by the MANIFOLD run-time system itself.

as a server for computing and interactive jobs. Other SGI machines on this network function as file servers.

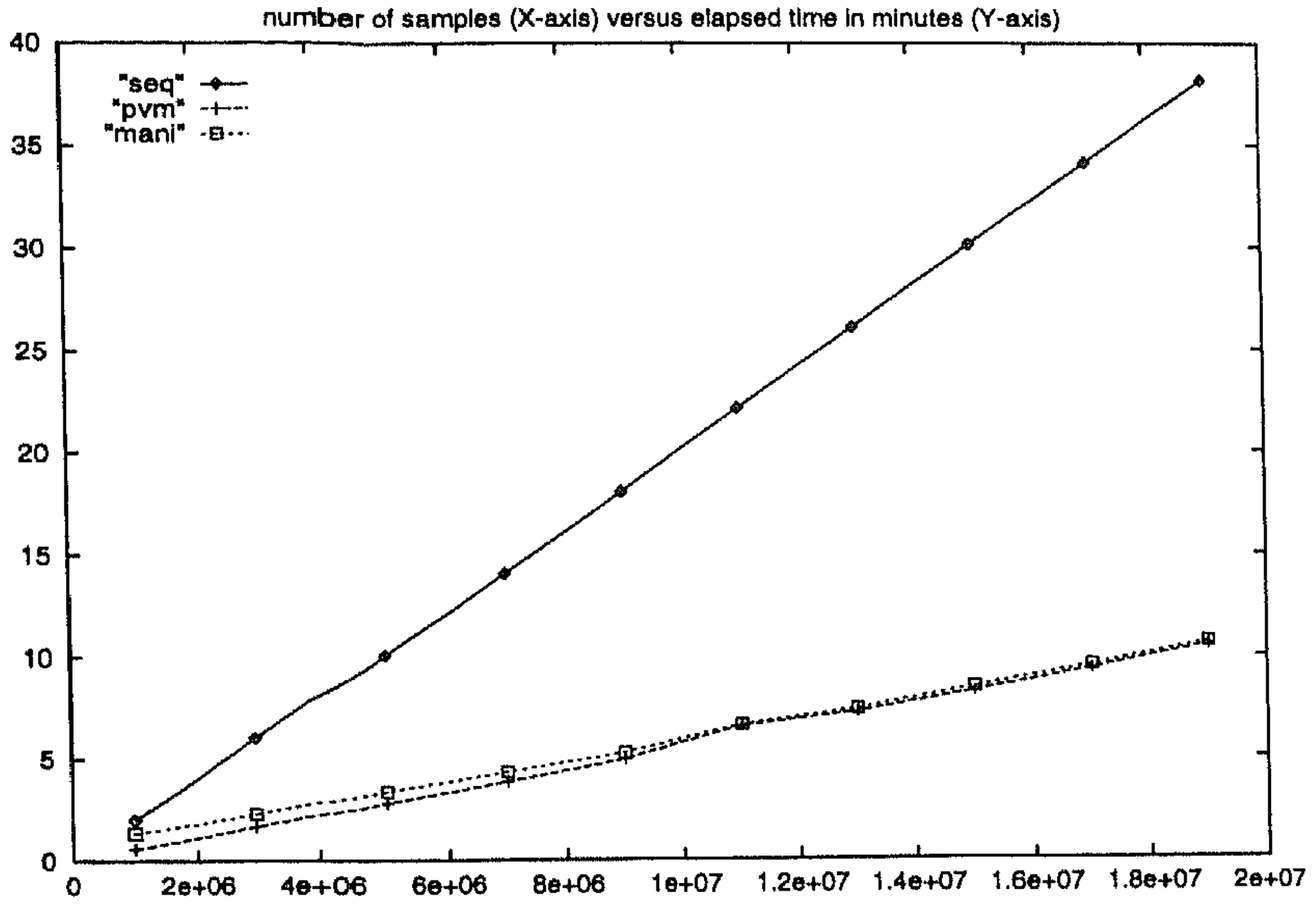


Figure 7: Performance of the three versions of the domain decomposition application

The results of our performance measurements are summarized in Figures 7 and 8, which show the elapsed time vs. the amount of computation. To obtain these results, we ran each of the three versions of our application on a number of case-points in the range of 10^6 to 1.9×10^7, representing gradually more intensive computation. The case-point 10^6, for example, means that each evaluator uses 10^6 sample points in its sub-domain to approximate the maximum of the function, and the case-point 1.9×10^7 means that the function is evaluated at 1.9×10^7 random sample points in each sub-domain to find its local maximum, etc. Such high sampling rates may seem unreasonably excessive for this domain decomposition problem; however, note that we are not really interested here in the particular function depicted in Figure 4, nor in domain decomposition in general. We use these high sampling rates only as a means to increase the computation intensity of this application in order to obtain our performance measures.

All experiments were run during quiet periods of the system, but, as in any real contemporary computing environment, it cannot be guaranteed that we were its only users. Furthermore, such unpredictable effects as network traffic and file server delays, etc., cannot be eliminated and are reflected in our results. To even out such "random" perturbations, we ran the three versions of the application on each case-point during the same period (so that they ran close to each other in real time) and ran each version of the application five times on each case-point. The raw numbers obtained from these experiments are shown in Tables 2, 3, and 4.

In these tables, the first column shows the intensity of the computation performed by each evaluator in terms of the number of sample points. The five results obtained for each case are shown in its corresponding row, in columns labeled "result 1" through "result 5" in these tables, representing the real (elapsed) time for each run of the application. (The time data in Tables 2, 3, and 4 appear in the *mm:ss.ss* or the *ss.ss* format, where *mm* is minutes and *ss.ss* shows seconds in the decimal fraction notation.) The five results of each experiment are sorted separately in each row of the Tables 2, 3, and 4 to always have the best and the worst performance measures appear, respectively, in their "result 1" and "result 5" columns. The best and the worst performance measures in each row were then discarded, and the average of the remaining three results appear in the "average" column in each table. The performance curves in Figures 7 and 8 were obtained using this average data.

The three curves in Figure 7 represent the real (elapsed) time for the sequential (marked "seq"), PVM (marked "pvm"), and MANIFOLD (marked "mani") versions of our experiments. The first observation about this graph is the significant difference between the performance of the sequential and the other two versions. Note that the sequential version, naturally, consists of a single process, which, by definition, can run on only one processor. Both the PVM and the MANIFOLD versions consist of over 36 processes[5], which is more than enough to take full advantage of the parallelism offered by the four processors of the machine.

Observe that the real time data for our concurrent versions are roughly four times smaller than those for our sequential version, i.e., at least for this application and in this computation range, we get a nearly linear speed-up by increasing the number of processors from one to four. However, this is not as glorious an achievement as it may seem at first. A folklore wisdom in the parallel programming community says that whenever you achieve linear speed-up, it is *not* an indication of the smartness of your parallel algorithm, but, rather, an indication of the stupidity of the sequential algorithm you compare it with! Without any sound theoretical basis, this maxim turns out to be true in practice, with annoying enough frequency, and certainly holds in our case. The only reason for the inclusion of this trivially-comparable sequential version in our experiment is to confirm our expectation of this linear speed-up; it would have been a disappointing surprise if it did not occur in this case. Nevertheless, even this not-so-smart sequential version should not be dismissed so prematurely: at lower computation intensities not shown in Figure 7 (e.g., below 700000 sample points) this sequential version out-performs the MANIFOLD version and at still lower intensities (e.g., below 40000 sample points) it outperforms the PVM version as well. At such low computation intensities, the overhead of having all these processes and their required underlying machinery is simply too high compared to the simple sequential version of the application.

[5]The underlying thread facility in our implementation of MANIFOLD on the SGI IRIX operating system allows each thread to run on any available processor. Thus, both PVM and MANIFOLD versions have, at least, the same number (36) of evaluator processes that can keep all of the four available processors of this machine busy all the time.

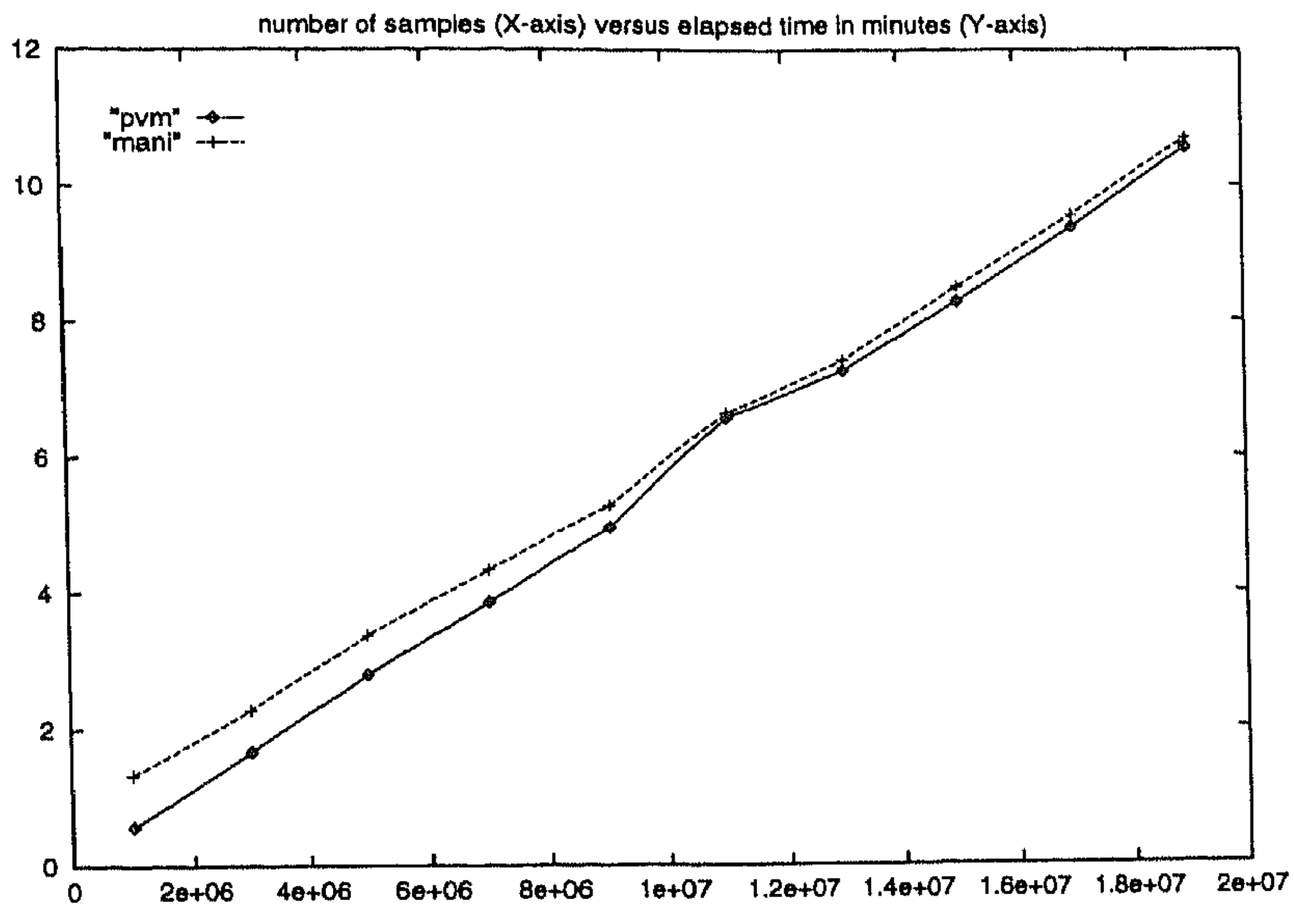


Figure 8: Performance of the parallel versions of the domain decomposition application

The closeness of the performance curves for the PVM and the MANIFOLD versions is rather remarkable, considering that our implementation of the MANIFOLD system uses PVM. Figure 8 shows these two performance curves at a larger scale. The distance between these two curves represents the total overhead of the MANIFOLD version with respect to its PVM version. A major contributor to this additional overhead is the difference in the two styles used in these two versions, which translates into over three times as many processes in the MANIFOLD version as compared to the PVM version. Observe that this overhead diminishes and the two curves overlap as the intensity of computation increases and the overhead of having the extra processes spreads over the longer execution time of the application. The other contributor to the difference between the two curves is our currently less-than-optimal implementation of the MANIFOLD system itself. Several rather straight-forward optimizations are in our agenda to, e.g., reduce the amount of dynamic memory allocation at execution time, which more importantly, saves on the number of (rather expensive) calls to the (thread-safe versions of) the `malloc/free` functions. Nevertheless, the largest single source of this extra overhead is due to the fact that the current version of PVM used in our present implementation of MANIFOLD is not thread-friendly (i.e., it was not designed to be used in conjunction with threads). This leads to a situation where a thread in each MANIFOLD task ends up wasting a good deal of time making (relatively expensive `select`) system calls through `pvm_receive`, in a polling loop.

We project that without this polling loop, the performance of the MANIFOLD version of this application will be slightly better than its PVM version for higher computation

intensities. This may seem paradoxical, but the fact that MANIFOLD uses threads means that it has the opportunity to use the common address space of the threads in the same task for communication among the processes corresponding to those threads, and avoid the more expensive inter-task communication of PVM. Generally, judicious grouping of processes within tasks by programmers when linking a MANIFOLD application can significantly improve the performance of the application by allowing the MANIFOLD run-time system to by-pass PVM when possible.

8 Conclusion

IWIM is a model of communication that supports anonymous communication and separation of computation responsibilities from communication and coordination concerns. MANIFOLD is a coordination language that takes full advantage of these two key concepts of IWIM. Unlike other coordination languages, MANIFOLD encourages decomposition of a parallel and/or distributed application into a hierarchy of pure computation and pure coordination modules, none of which contain hard-coded dependencies on their environment. This leads to highly reusable computation modules, and more interestingly, also to highly reusable coordination modules.

The examples in this paper show a single coordination module used in two very different applications. We have also used MANIFOLD to reorganize existing Fortran 77 sequential code into a parallel and distributed application[18]. The usefulness of the IWIM model and, in particular, the MANIFOLD language in these and other applications has been very encouraging. Performance results presented in this paper show that for applications with realistic computation intensities, the overhead of using MANIFOLD to achieve this enhanced modularity and reusability is in practice small, compared to the more conventional paradigms for the design and programming of parallel and distributed software. The resulting programs are easily scalable and thus can make effective use of any number of available processors and computers, even in heterogeneous environments.

The plumbing paradigm inherent in IWIM makes it easy to compose and recompose a MANIFOLD application and adapt it to new requirements. To enhance the effectiveness of this coordination language, we are presently developing a visual programming environment around MANIFOLD which takes advantage of its underlying plumbing paradigm[13].

References

[1] ANDREOLI, J., CIANCARINI, P., AND PARESCHI, R. Interaction Abstract Machines. In *Trends in Object-Based Concurrent Computing*. MIT Press, 1993, pp. 257–280.

[2] ANDREOLI, J., GALLAIRE, H., AND PARESCHI, R. Rule-Based Object Coordination. In *Object-Based Models and Languages for Concurrent Systems* (1995),

P. Ciancarini, O. Nierstrasz, and A. Yonezawa, Eds., vol. 924 of *Lecture Notes in Computer Science*, Springer-Verlag, Berlin, pp. 1–13.

[3] ANDREOLI, J., AND PARESCHI, R. Linear Objects: Logical processes with built-in inheritance. *New Generation Computing 9*, 3-4 (1991), 445–473.

[4] ARBAB, F. Coordination of massively concurrent activities. Tech. Rep. CS–R9565, Centrum voor Wiskunde en Informatica, Kruislaan 413, 1098 SJ Amsterdam, The Netherlands, November 1995. Available on-line http://www.cwi.nl/ftp/CWIreports/IS/CS-R9565.ps.Z.

[5] ARBAB, F. The IWIM model for coordination of concurrent activities. In *Coordination Languages and Model* (April 1996), P. Ciancarini and C. Hankin, Eds., vol. 1061 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 34–56.

[6] ARBAB, F. Manifold version 2: Language reference manual. Tech. rep., Centrum voor Wiskunde en Informatica, Kruislaan 413, 1098 SJ Amsterdam, The Netherlands, 1996.

[7] ARBAB, F. The influence of coordination on program structure. In *Proceedings of the* 30^{th} *Hawaii International Conference on System Sciences* (January 1997), IEEE.

[8] ARBAB, F., HERMAN, I., AND SPILLING, P. An overview of Manifold and its implementation. *Concurrency: Practice and Experience 5*, 1 (February 1993), 23–70.

[9] ATKINSON, C., GOLDSACK, S., MAIO, A. D., AND BAYAN, R. Object-oriented concurrency and distribution in DRAGOON. *Journal of Object-Oriented Programming* (March/April 1991).

[10] BANATRE, J., AND METAYER, D. L. Programming by multiset transformations. *Communications of the ACM 36*, 1 (January 1993), 98–111.

[11] BERGSTRA, J., AND KLINT, P. The ToolBus Coordination Architecture. In *Proc. 1st Int. Conf. on Coordination Models and Languages* (Cesena, Italy, April 1996), P. Ciancarini and C. Hankin, Eds., vol. 1061 of *Lecture Notes in Computer Science*, Springer-Verlag, Berlin, pp. 75–88.

[12] BORGI, A., AND CIANCARINI, P. The concurrent language Shared Prolog. *ACM Transactions on Programming Languages and Systems 13*, 1 (1991), 99–123.

[13] BOUVRY, P., AND ARBAB, F. Visifold: A visual environment for a coordination language. In *Coordination Languages and Model* (April 1996), P. Ciancarini and C. Hankin, Eds., vol. 1061 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 403–406.

[14] CARRIERO, N., AND GELERNTER, D. LINDA in context. *Communications of the ACM 32* (1989), 444–458.

[15] CARRIERO, N., GELERNTER, D., AND ZUCK, L. Bauhaus Linda. In *Object-Based Models and Languages for Concurrent Systems* (1995), P. Ciancarini, O. Nierstrasz, and A. Yonezawa, Eds., vol. 924 of *Lecture Notes in Computer Science*, Springer-Verlag, Berlin, pp. 66–76.

[16] DEBOSSCHERE, K. Blackboard communication in Prolog. In *Parallel Execution of Logic Programs*, vol. 569 of *Lecture Notes in Computer Science*. Springer-Verlag, 1991, pp. 159–172.

[17] EVERAARS, C. T. H., AND ARBAB, F. Coordination of distributed/parallel multiple-grid domain decomposition. In *Proceedings of Irregular '96* (August 1996), A. Ferreira, J. Rolim, Y. Saad, and T. Yang, Eds., vol. 1117 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 131–144.

[18] EVERAARS, C. T. H., ARBAB, F., AND BURGER, F. J. Restructuring sequential Fortran code into a parallel/distributed application. In *Proceedings of the International Conference on Software Maintenance '96* (November 1996), IEEE, pp. 13–22.

[19] FOSTER, I., AND TAYLOR, S. *Strand: New Concepts in Parallel Programming*. Prentice-Hall, 1990.

[20] FROLUND, S., AND AGHA, G. A language framework for multi-object coordination. In *Proc. ECOOP '93* (1993), vol. 707 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 346–360.

[21] GEIST, A., BEGUELIN, A., DONGARRA, J., JIANG, W., MANCHEK, R., AND SUNDERAM, V. PVM 3 user's guide and reference manual. Tech. Rep. ORNL/TM-12187, Oak Ridge National Laboratory, September 1994.

[22] GELERNTER, D., AND CARRIERO, N. Coordination languages and their significance. *Communication of the ACM 35*, 2 (February 1992), 97–107.

[23] HELM, R., HOLLAND, I., AND GANGOPADHYAY, D. Contracts: Specifying behavioral compositions in object-oriented systems. *SIGPLAN Notices 25*, 10 (October 1990), 169–180.

[24] HOLLAND, I. Specifying reusable components using contracts. In *Proc. ECOOP '92* (July 1992), O. L. Madsen, Ed., vol. 615 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 287–308.

[25] SUTCLIFFE, G. Prolog-D-Linda v2: A new embedding of Linda in SICStus Prolog. In *Proc. Workshop on Blackboard-based Logic Programming* (June 1993), pp. 105–117.

[26] WISE, M., JONES, D., AND HINTZ, T. PMS-Prolog: A distributed Prolog with processes, modules and streams. In *Implementations of Distributed Prolog*, Series in Parallel Computing. Wiley, 1992, pp. 379–404.

noc	result 1	result 2	result 3	result 4	result 5	average
1000000	2:00.62	2:00.64	2:00.66	2:00.87	2:01.20	2:00.72
3000000	6:01.62	6:01.69	6:01.71	6:02.20	6:02.33	6:01.87
5000000	10:02.53	10:02.62	10:03.14	10:09.32	10:13.38	10:05.03
7000000	14:03.37	14:03.45	14:04.28	14:06.09	14:06.30	14:04.61
9000000	18:04.71	18:04.84	18:05.11	18:05.78	18:07.48	18:05.24
11000000	22:06.91	22:08.00	22:09.18	22:10.05	22:11.25	22:09.08
13000000	26:06.28	26:06.46	26:06.63	26:06.89	26:07.49	26:06.66
15000000	30:07.20	30:07.90	30:09.40	30:10.68	30:22.88	30:09.33
17000000	34:08.03	34:08.59	34:08.90	34:09.88	34:10.03	34:09.12
19000000	38:09.36	38:09.55	38:12.12	38:12.52	38:14.53	38:11.40

Table 2: The elapsed time in the sequential version of domain decomposition

noc	result 1	result 2	result 3	result 4	result 5	average
1000000	34.36	34.50	34.76	34.79	34.81	34.68
3000000	1:40.15	1:40.34	1:40.41	1:40.73	1:40.94	1:40.49
5000000	2:46.12	2:47.06	2:48.37	2:49.45	3:10.39	2:48.29
7000000	3:50.67	3:50.97	3:51.32	3:52.19	3:52.28	3:51.49
9000000	4:56.47	4:56.72	4:57.54	4:57.87	4:57.90	4:57.38
11000000	6:03.86	6:09.14	6:30.00	7:04.40	7:10.82	6:34.51
13000000	7:10.27	7:10.57	7:16.14	7:19.59	7:26.08	7:15.43
15000000	8:15.91	8:16.61	8:16.77	8:17.08	8:29.04	8:16.82
17000000	9:22.09	9:22.21	9:23.60	9:37.94	12:08.92	9:27.92
19000000	10:26.89	10:30.56	10:33.14	10:37.16	13:38.87	10:33.62

Table 3: The elapsed time in the PVM version of domain decomposition

noc	result 1	result 2	result 3	result 4	result 5	average
1000000	1:15.17	1:19.79	1:19.91	1:19.97	1:21.25	1:19.89
3000000	2:15.52	2:16.55	2:17.58	2:17.60	2:19.22	2:17.24
5000000	3:19.18	3:20.82	3:21.53	3:25.38	3:44.29	3:22.58
7000000	4:14.17	4:19.80	4:20.17	4:20.38	4:22.50	4:20.12
9000000	5:11.98	5:15.38	5:16.21	5:17.86	5:18.59	5:16.48
11000000	6:24.84	6:28.33	6:35.86	6:51.47	6:54.89	6:38.55
13000000	7:18.54	7:26.35	7:29.21	7:50.92	10:29.03	7:35.49
15000000	8:24.03	8:26.42	8:29.19	8:33.86	9:00.74	8:29.82
17000000	9:24.78	9:33.10	9:33.46	9:34.24	9:52.26	9:33.60
19000000	10:35.32	10:39.81	10:42.20	10:43.57	10:45.33	10:41.86

Table 4: The elapsed time in the Manifold version of domain decomposition

A Dataflow Graphics Workstation

P.J.W. ten Hagen, I. Herman
Center for Mathematics and Computer Science (CWI), Dept. of Interactive Systems
Kruislaan 413, 1098 SJ Amsterdam, The Netherlands

J.R.G. de Vries
Dataflow Technology Nederland bv.
Maanweg 156, Gebouw BN 086, 2516 AB Den Haag, The Netherlands

Abstract — A new, revolutionary architecture for a superworkstation for graphical purposes is presented. The architecture is based on the use of advanced graphics components and, mainly, on the heavy use of dataflow processing technology, a still unexplored field of parallel computing as far as graphics is concerned. The resulting initial configuration is able to produce 200.000 to 250.000 Gouraud shaded and Z-buffered 3D triangles in a second with a colour palette of 24 bits per pixels.

1 Introduction

The architecture for a graphics workstation presented in this paper is a revolutionary one because it is the first in a new and promising line of architectures. The revolution stems from the fact that with the dataflow processors used, arbitrary combinations of highly dedicated processing can be dynamically created. For the case we will describe here all processing is dedicated to graphics but the application is by no means restricted to this field. In fact, such processors have been, until now, primarily used in image processing applications. Once the case is made for the graphics area, it is obvious that the next step will be to provide a system which combines graphics and image processing. There are many applications in the field waiting for a system which integrates both possibilities. The purpose of this paper is however, to illustrate how well dataflow architectures can support three dimensional interactive graphics.

The system described here is not an experimental system. It is much more. It uses existing hardware components only, albeit they are all the most advanced hardware components currently available. In addition, the entire architecture has been designed and is being implemented. Testing of the integrated system will start in June 1989. Production for the market might start before the end of 1989. The performance figures mentioned in the paper are based on testing the working individual components and/or simulation outputs.

The discussion about the results and possible improvements is based on a particular initial configuration chosen around 32 dataflow processors in a hypercube arrangement. The flexibility of the dataflow hardware and the overall design guarantee that simple extensions to larger configurations containing 64 or more processors are possible. However, in this paper all figures and possible improvements are only concerned with this initial configuration. In this way

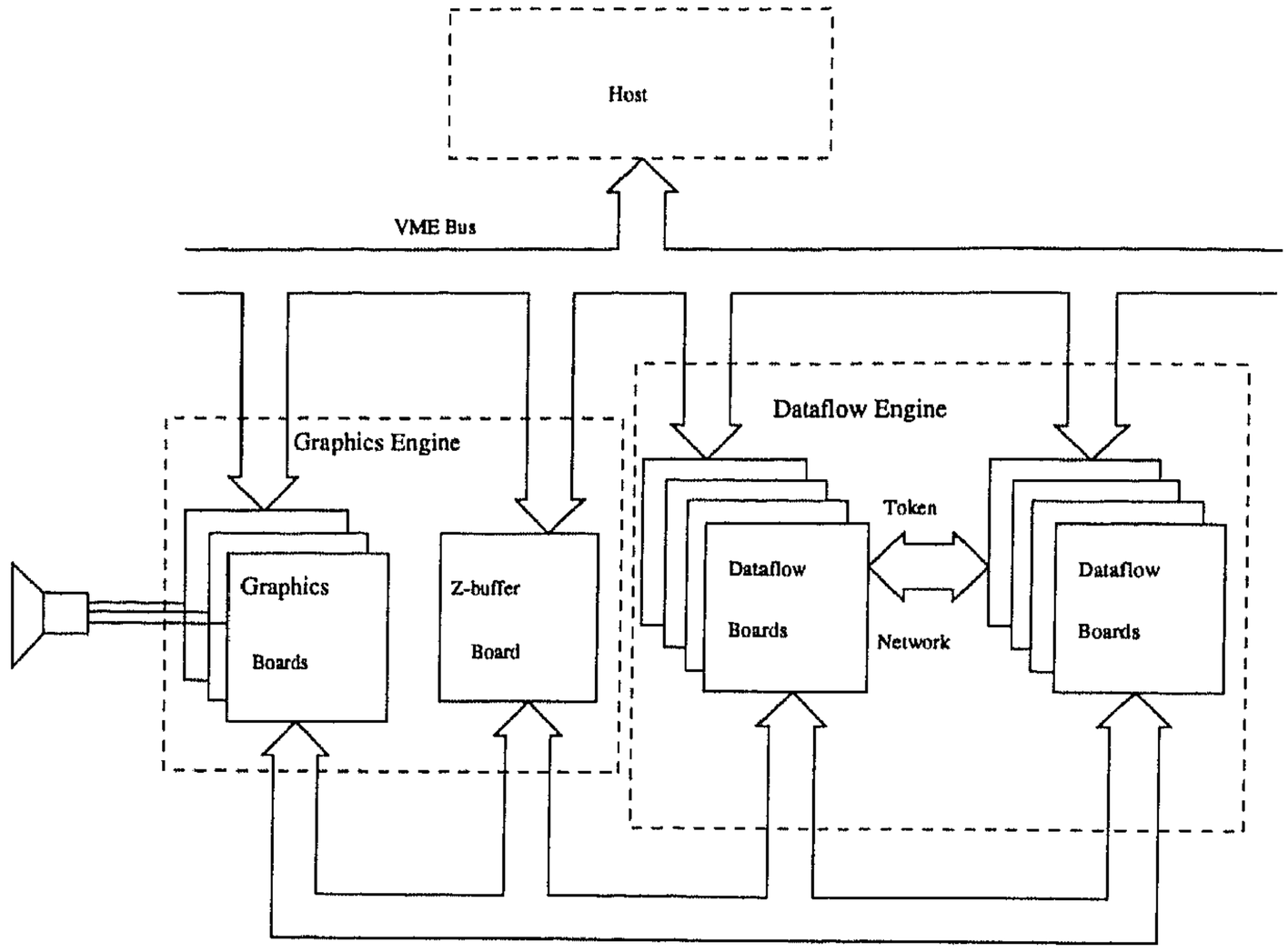


Figure 1. Overall view of the DFC

the reader can make a proper judgement about the possibilities opened up by this line of systems without getting confused by the great variety of extensions and further performance boosts possible..

2 The Hardware

2.1 General Overview

The hardware of DFC (Data Flow Computer) is centered around three basic building blocks: the *Host*, the *Graphics Engine* and the *Dataflow Engine* (see Figure 1). The Host may be any UNIX BSD machine provided that a VME access is available (we use currently a SUN 3 Workstation). Both the Graphics Engine and the Dataflow Engine are used as co–processors of this Host[1]. The building blocks are connected via two busses: a standard VME bus and a so called *Distributed Memory Bus*. The former is primarily used to connect the

---

[1]) It is theoretically possible to put for example two independent Dataflow Engines on the same VME bus, getting
therefore two independent co-processors of this type. However, we will not deal with this configuration in the present
paper.

Host with the two other blocks. The Host may control the processing parts of the two other blocks via register I/O and DMA; all these possibilities are quite standard on such configurations

The Distributed Memory Bus is used to provide data transfer between the Graphics Engine and the Dataflow Engine as well as among the different internal components of these. This bus is much faster than the VME bus: the current transfer speed is 36Mbytes/sec (16 million data transfers per seconds). All memories on all boards are connected via this bus, including the video memories and the 3D Z–buffer used by the graphics processors.

A third means of data transfer is provided by the dataflow *Token Network* with a capacity of 320Mbytes/sec, which connects the components of the Dataflow Engine. The role of this network will become clear in the following paragraphs.

2.2 The Graphics Engine

The Graphics Engine contains three identical components, namely the so called *Advanced Graphics Boards*, and, additionally to them a *Z-Buffer Board*. The conception of the Advanced Graphics Boards is such, that they are also usable as stand-alone graphics boards in a VME environment; however, details of this version are not covered in the present paper.

Each *Advanced Graphics Board* contains a 4Mbytes VRAM for pixel memory with RGB input and output. Using the three boards together we get therefore a frame buffer of 24 bit/pixel assuming a resolution of max. 2048×2048. Each board controls 8 bits out of the 24; the first board is responsible for the generation of the red pixels, the second one for the green pixels and, finally, the third one for the blue ones. Additional cursor hardware is available to mix a cursor on video output without overwriting the frame buffer. The board has also extensive facilities to generate, synchronise and input video signals in virtually all video formats; this feature might be very important for future image processing applications

Each board contains a Hitachi HD63484 graphics processor, which may be used for high-speed two-dimensional graphics. However, as we will see in the following, the main use of the whole DFC is to provide three dimensional graphics; in other words, the HD63484 processor is used first of all for video timing generation and to help video input.

The main "processing unit" on the Graphics Board is a Systolic Array which contains four 16 MIPS bit slice processors (16 bits wide). Each one is accompanied by a small multiplier and a function table. This Systolic Array is microprogrammable, and with an adequate set of microcoded instructions it turns the Advanced Graphics Board into a powerful graphics co-processor. Because of the uniform nature of the three Graphics Boards, the very same microcode runs in parallel on the boards controlling, as we have already mentioned, the red, green and blue pixels respectively. The available instruction set contains the usual BitBlt operations (Block Copy, Block Fill, etc.), vector drawing and some image processing instructions. The speed of pixel generation is considerable: in case of Block Copy, for example, 20 million pixels/sec. may be copied; the peak vector generation speed (for horizontal lines) is of 30 million pixels/sec.

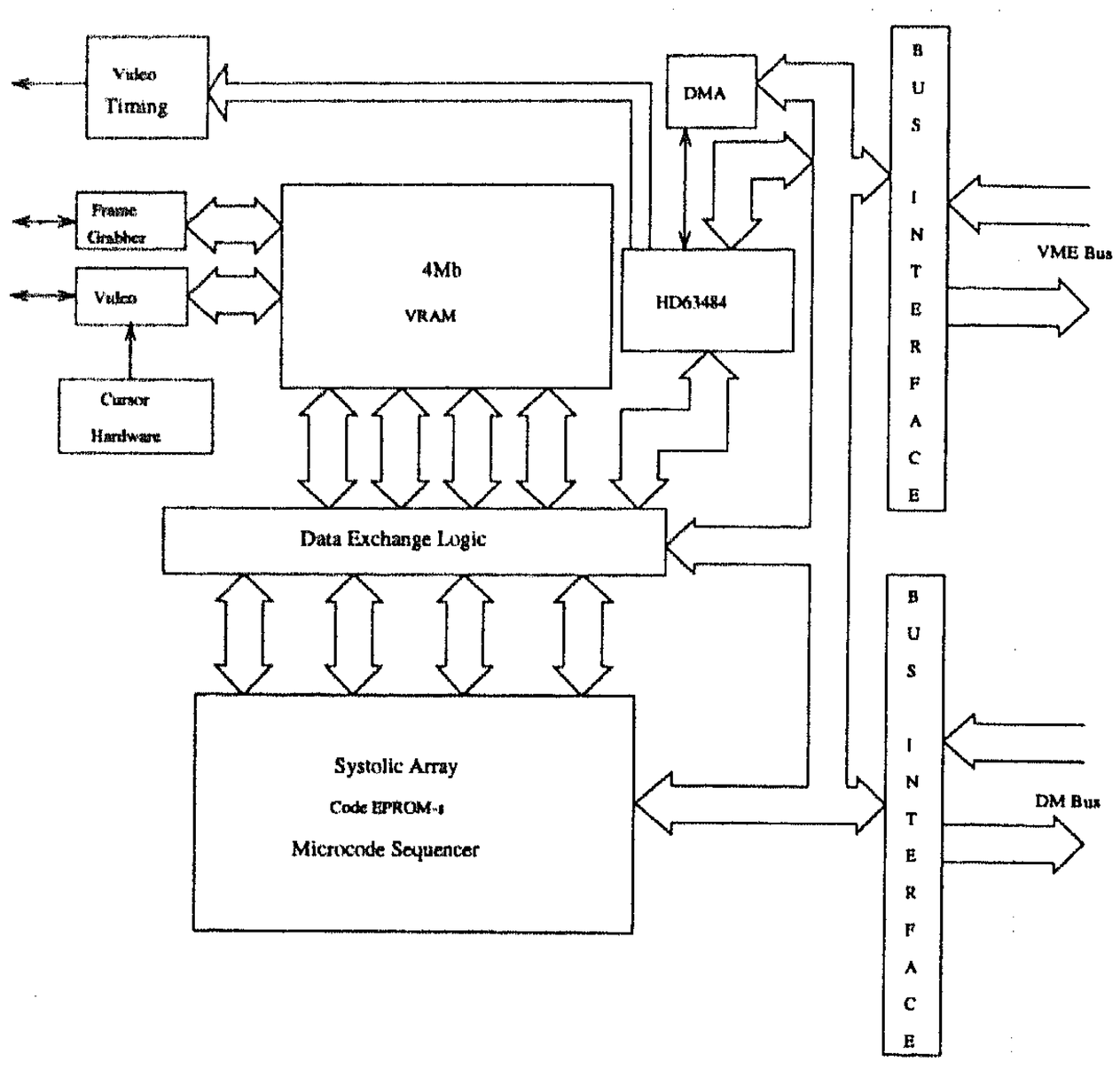


Figure 2. The Advanced Graphics Board

.For the purpose of a 3D Workstation, however, the most important instruction is the possibility of generating 3D Gouraud shaded triangles. The Systolic Array gets the colour values (red, green and blue values respectively) for each vertex; during scan-conversion these values are linearly interpolated to set the right colour value for each pixel. This instruction, however, has to cooperate with the fourth component of the Graphics Engine: the Z-buffer Board.

The *Z-Buffer Board* contains a similar Systolic Array to the Graphics Boards but with a different microcode. Additionally, the board contains a 16Mbytes Z-buffer memory. This memory is able to store the Z value of the incoming primitives, with a depth of 16 bits.

The microcode of the Z-Buffer Board determines which portion of a 3D triangle is effectively visible by performing a scan conversion in the third dimension and comparing the value of the generated depths with the content of the Z-buffer. As a result, the Z-Buffer Board dispatches commands to the Graphics Boards so that these latter ones will generate the visible *x* and *y* pixels by interpolating the colour values. In other words, via four cooperating boards

and microcodes the Graphics Engine has the possibility to generate 3D Gouraud shaded triangles with Hidden Surface effects automatically calculated via the Z-buffer. This instruction will serve as a basis for the 3D graphics capabilities of the DFC; in a way, the whole software realised in the machine aims at the fast creation of appropriate triangles. At present, the generation speed is on average 180.000 to 200.000 triangles/sec (the real speed depends on the size of the triangles, of course).

Although the possibility of triangle generation is by far the most important feature of the Graphics Engine, it has also some additional instructions. An interesting example is the possibility of *shielding*.

A *shield* is a polygon which is set into the Z-buffer to any "location", that is, conceptually, onto a plane which is parallel to the *x-y* plane. The effect is that all pixels whose coordinates belong the *x-y* area covered by the shield and which are "farther" as the depth position of the shield will be clipped automatically. Such shields may be "or"-ed in the Z buffer, combining therefore simple shields to produce more complicated ones.

This facility offers a natural tool for traditional clipping. Furthermore, if some of the shields in use are the set-theoretical negation of a rectangle in the frame buffer with respect to the full 2048×2048 frame buffer, by the combination of these and "classical" clipping rectangles the clipping operations required by a window manager environment are authomatically covered. Indeed, if such a "negated" shield is set first for a given window and, subsequently, all rectangles belonging to overlapping windows are set, the resulting shield in the Z buffer will correspond exactly to the clipping area required by the windowing environment. Clearly, this facility has a great practical importance.

2.3 The Dataflow Engine

The Dataflow Engine is without doubt the most original and most interesting part of the whole Workstation, which gives it a very individual flavour. The Engine is a very fast computing co-processor of the Host; its primary task is to provide transformation of three dimensional points as well as offering a higher-level software interface to the Host than the plain 3D triangles. To be able to use the speed of the Graphics Engine (which is quite high), this co-processor should also be very fast. To achieve the required speed (that is ≈200.000 triangles/sec) the technique of *dataflow computing* is used. As this technique is not yet widely known, we have to make a small detour to introduce the basics of it; for a more detailed introduction the reader is referred to Veen[1] or Herath et al. [2].

2.3.1 Dataflow Computing

Dataflow computing is an approach to a hardware and software organisation which aims at offering a highly parallelised structure as an alternative to the traditional von Neumann computing architecture. Its basic notion is *dataflow nets* (also called *dataflow graphs*), which may be defined as follows.

A *dataflow net* is a directed graph; the vertices of the graph are called the *nodes* in the dataflow terminology, while the edges are the *links* of the net. Each node represents a tiny processing element; the actual processing per-

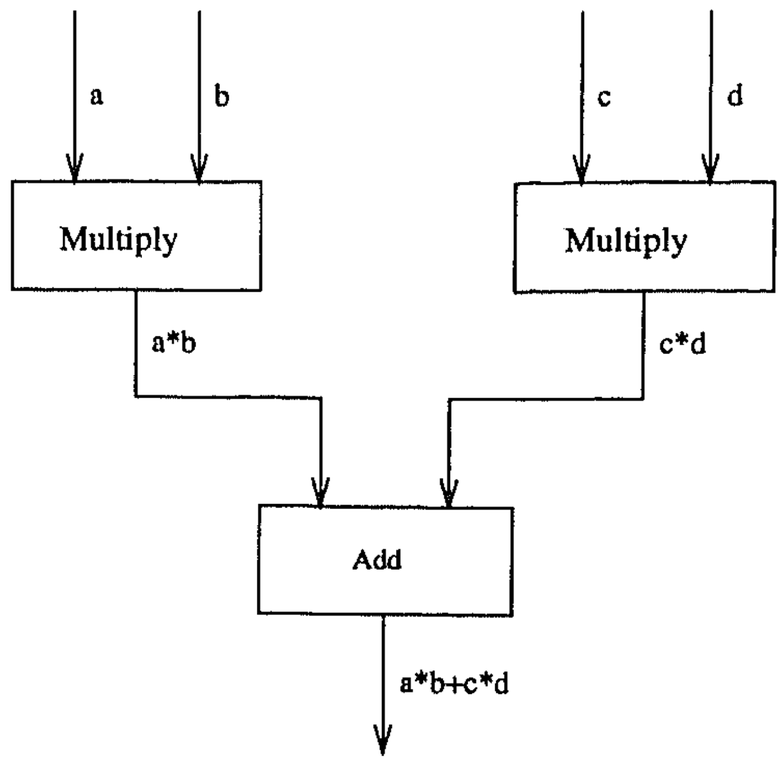


Figure 3. A simple dataflow net

formed by the node is defined when defining the graph itself. A node may perform very elementary operations only; its possible instruction set may be compared to the assembly instructions of a traditional computer.

Data are encapsulated in *tokens*. These tokens are transferred from one node to another by the links, following the direction of the link. The nodes themselves operate on the data carried by the input token(s) of the node and the operation itself may either delete the token (if no output link is defined) or may produce one or more token(s) again, each of them being output on one of the output links defined in the net.

A node will perform its defined operation when all input links contain a token. This rule may remind the reader of the theory of Petri nets, although a dataflow net is *not* exactly a Petri net (in the latter case, all processing nodes are of analogous type, which is by far not the case in a dataflow net). No assumption may be used as far as the relative timing of the nodes: if all tokens are present, a node may process *at any time, regardless of the remaining nodes* (the exact timing depends on the hardware realisation of the dataflow net). This means that a dataflow net represents a very high level of parallelism; that is why it is also referred to as *data-driven fine-grain* parallelism.

Figure 3 shows a very simple example of a dataflow net; this net performs the calculations for $a \times b + c \times d$. The data for a, b, c and d are sent to the net on the corresponding links (see Figure 3); they may arrive in a random order. Once a *and* b are present (respectively c *and* d), the node which performs the multiplication may process and will produce $a \times b$ (resp. $c \times d$). The multiplier nodes will operate in parallel; no assumption may be made as far as their relative timing is concerned. On the other hand, once *both* $a \times b$ and $c \times d$ are produced on their respective links, the node responsible for the addition may proceed and will produce the final result. Figure 9 at the end of the paper gives an example for a much more complicated dataflow net.

A *dataflow computer* is a computer which is *programmable* using the dataflow model. In other words, such a computer should be fed somehow with the description of the net, including the topology of the net as well as the instruction(s) assigned to each node. Such a network is then *executed* by feeding all input links with the necessary tokens so that the nodes of the network could start proceeding following the rule cited above.

2.3.2 The Dataflow Boards

The Dataflow Engine consists of eight identical and interconnected *Dataflow Boards*. Each Dataflow Board is based, in turn, on NEC's μPD72181 chip, the so called Image Pipelined Processor, [3], which we will call the DFP chip hereafter.

A DFP is a tiny dataflow computer. The appropriate description of a dataflow net may be loaded onto the chip and by getting the necessary tokens the chip is able to execute the net. It is tiny, because the dataflow net which may be loaded is relatively simple: it may contain at most 64 nodes and 128 links. In contrast to a von Neumann microprocessor, the coded instruction cannot be stored on an external memory; all instructions should be stored within the chip (which makes it, of course, much faster!). Some part of the instruction set, which may be used when defining the nodes, corresponds to an early day microprocessor: 16 bits integer addition, multiplication, comparison (no division!), bit setting/resetting instructions as well as shift operations. Additionally to these ones there are a number of operations to "manage" the net (token deletion, token duplication, dispatching over links etc.) as well as instructions specially dedicated for image processing purposes (e.g. shift and bit count instructions).

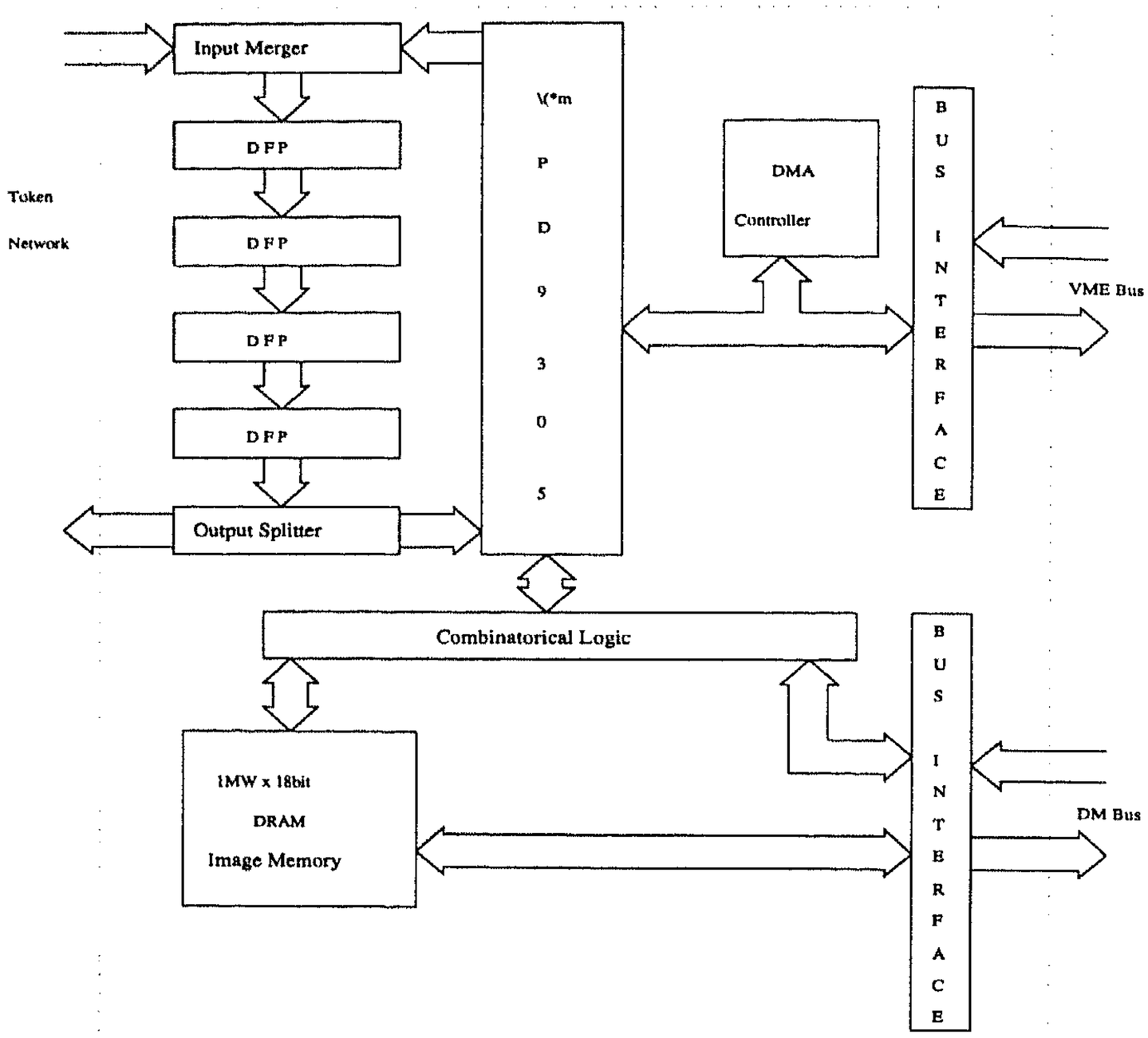


Figure 4. The Dataflow Board

However, the DFP *is* a revolutionary machine: it is the first commercially available single-chip dataflow computer, which makes it extremely interesting to explore its capabilities for example for graphics. It may operate at 20MHz and, for example, a 16 bits by 16 bits multiplication is performed at 100 ns. The extensive use of internal parallelism makes it very fast indeed.

The DFP needs additional instructions to communicate with its environment. There are basically two ways of communications: either a token is sent to another DFP or some external medium is to be accessed for reading and/or writing. To manage this latter task, NEC offers also an additional chip, the µPD9305. Via the µPD9305 chip, a DFP has the possibility to access a 24 bits address range for external I/O.

Each DFC Dataflow Board contains 4 DFP-s, one µPD9305, a so called Image Memory, which is a 1 Mwords DRAM (one word equals 18 bits) and, finally, a DMA controller for fast memory I/O. As we have already mentioned, the board's registers may be accessed from the VME as well as from the Distributed Memory Bus. The DFP-s may access the Image Memory by making use of the facilities offered by the µPD9305 (see Figure 4).

'he whole Dataflow Engine consists of 8 Dataflow Boards; in other words, ntains 32 DFP-s and 8 Mwords of Image Memory, which represents alto- ıer quite an impressive processing power. Each DFP may uniformly ac- ; *the whole* Image Memory (not only the part of it which resides on the e board). As we have already mentioned, this memory is also accessible ctly from the Host. Token communication among the DFP chips is routed ıe Token Network; that is, token communication of the DFP-s has a private ı path for itself.

łow is this Engine programmed? Each DFP has to be loaded with its own ıflow net. The whole program of the Dataflow Engine is *not* one huge net; ead, it consists of 32 independent dataflow nets which may communicate one another via special nodes and links. The Host may also send tokens ny of the DFP-s and the DFP-s are also able to send data to the Host in form of tokens. By using these facilities the synchronisation of the dataflow as well as the synchronization of the whole Dataflow Engine with the Host /or the Graphics Engine may be achieved. How this program effectively ks will be the topics of the forthcoming sections.

The Software

Some Generalities

ıre are different forms of parallelisms which may be used in a graphics sys- . The choice where and in which form a parallel architecture is introduced, y lead to very different hardware/firmware/software configurations.

The image generation on a raster device may be cut into two distinct steps. the one hand, high level geometric structures should be manipulated (clip- gs, transformations etc.) and, on the other hand, the objects should be ren- ed on scan converting level. The different attempts of parallelisation in phics systems follow roughly the same division; there are approaches ch try to introduce parallelism on the structure level while others try to ex- t the possibilities of scan converting level parallelism. We do not want to ə an exhaustive survey of all different approaches here. As examples for ıcture level parallelism we might cite all different pipeline architectures, ch are used in a number of commercially available graphics workstations g. Silicon Graphics), the PHIGS machine of Abi-Ezzi and Milicia [4], or the GIC II machine of Finch et al.[5]. The different "smart" pixel memories like Pixel Machine[6], the DisArray machine[7,8], Scan Line Access Memo- ı[9], or others are good examples for scan converting level parallelism. We ›uld also remark that this classification is, as all such classifications, not re- ' precise; there are "hybrid" approaches as well (see e.g. ten Hagen et 10], where the distinction created by the presence of a frame buffer disap- ars).

In case of the DFC, both forms of parallelism are present, although to a dif- ənt extent. As we have seen when presenting the hardware, the Graphics gine does include a certain level of parallelism; indeed, the pixel generation red, green and blue pixels as well as Z-buffer handling are done in parallel different pieces of hardware. Once "inside" the boards the fact that there ı four processors within the systolic arrays is a source of parallelism again.

Finally, the microporgramming technique itself is also a non-negligible source of parallelism. In other words, the Graphics Engine does contain elements of scan converting level parallelism; however, to exploit the possibilities of the Dataflow Engine, we had to concentrate first of all on the possibilities of structure level parallelism.

As far as structure level parallelism is concerned, there are again different possible approaches. Some of them are as follows.

1) A scene to be visualised consists of a number of more or less independent objects (as far as geometry is concerned). In case of a full regeneration of a picture, the individual objects may be transformed and generated independently, provided that appropriate Z-buffer hardware takes care of hidden surface removal (such a hardware is available in the DFC). In other words, a set of independent output pipelines, including transformations and some basic clips may run in parallel.
2) In case of interactive use, the whole scene is not necessarily to be regenerated starting on the top level of representation. If e.g. only one object has been moved, it is superfluous to retransform all objects, except the one whose position or orientation have been changed. On the other hand, to achieve correct output, more (sometimes all) objects must be re-drawn. This is another source of parallelism: while transforming one object, the others may be subject to a redrawing process, provided that the intermediate (i.e. transformed but not yet scan-converted) storage of the object is also available.
3) The output pipeline itself may also be highly parallel. The same kind of operations (matrix-vector multiplication, shading calculations, projective division etc.) have to be performed on a set of points belonging to one objects.
4) Finally, the mathematical formulas in use (e.g. vector-vector multiplication) include possible sources of parallelism.

In a "conventional" architecture it is fairly complicated to take into account all these possibilities of parallelism (and there may be even more!). Usually, one has to concentrate on one or two such aspects, to have a managable task at hand. This is why the data flow approach seems to be fruitful: by its very nature it is fairly straightforward to model all kinds of parallelism, by providing the appropriate software; in fact, all classical approaches to parallelism (SIMD, pipelines, MIMD etc.) may be "simulated" easily in a data flow environment. In the software we have managed to provide for the DFC, all aspects of parallelisms cited above could be realised in a natural way.

Figure 5 gives an overall view of the software running on the DFC Workstation. All of these software blocks will not be explained in detail; this would go far beyond the scope of the present paper. In the following, we will concentrate first of all on the most original part of this software, namely on the programs running on the Dataflow Engine.

4 The Dataflow Software

4.1 Overall Structure

The dataflow programs realise a number of logical functions operating on well defined data. These functions are resident in the DFP processors and are activated by sending some specified tokens to them and are invoked by the Host application program(s). One may view these as being co-routines of some concurrent programming languages, where sending one or more starting tokens corresponds to the activation of the co-routine. The hardware and the software organisation makes it certain that there is no hidden sequentiality; e.g. logically different functions do not share DFP-s. Each function sends back a token to the Host once its task has been finished and, therefore, the corresponding DFP is ready to process again.

There is, however, one significant difference: there might be more *instances* of the same function. This is, as we will see later, the case of e.g. the transformation function: the Host has *two* transformation functions at its disposal, which are identical as far as their functional specification is concerned but may run absolutely in parallel.

The functions operate on two types of entities: *point lists* and *colour values*.

Point lists are arrays of four dimensional (homogeneous) points stored in Image Memory in form of integers The points may be logically located in two coordinate systems. The first one is (to use the usual terminology) a *World Coordinate System* whose limits are not specified. The second coordinate system is the *Device Coordinate System* where the *x, y* coordinates are expected to be in the range of the display resolution and *z* within the limits of the Z-buffer. In the latter case the *w* coordinates have no real geometrical meaning but they are used for internal purposes.

Point lists are transformed by a dataflow transformation function which performs a matrix-vector multiplication and the projective division: in other words, it turns point lists stored in World Coordinates into point lists stored in Device Coordinates. The starting points (that is the ones described in World Coordinates) are put into the Image Memory by the Host. As we have already mentioned, there are two instances of this function; in other words, the Host has the possibility to start a transformation function two times *in parallel*; this is a significant source of speeding up the overall speed of the Workstation. The choice of having two such transformation function instances (and not more) is

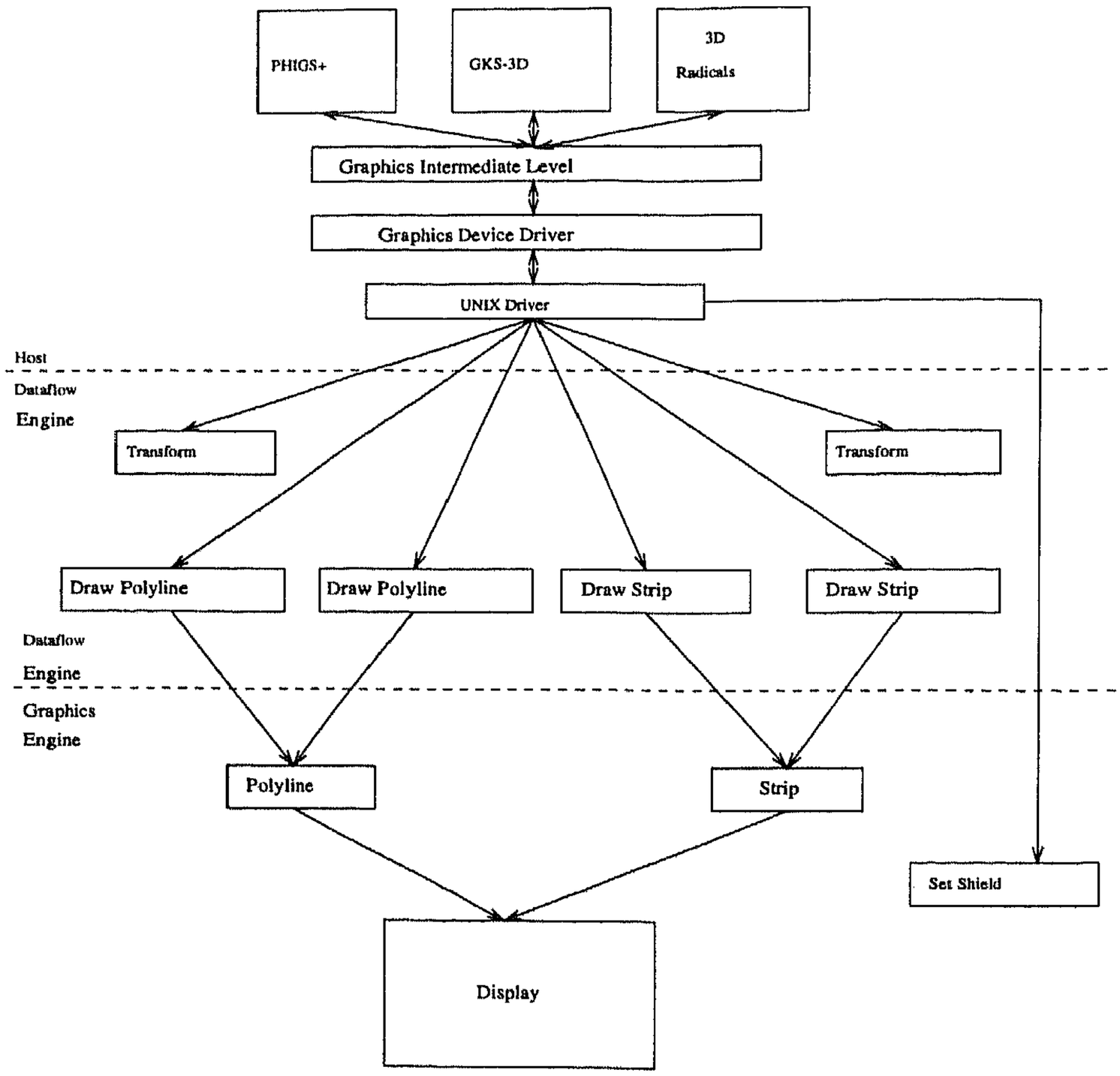


Figure 5. Overall view of the DFC Software

the outcome of a balance between DFP chip numbers and required overall speed; clearly, if the hardware were different e.g. by having newer versions of the dataflow processors, this number could change as well.
How the point lists are interpreted depends on various drawing functions. In the actual version there are basically two of them: the *"draw polyline*" and the *"draw strip"*. Both of them read the points from the point list and read the associated colour (R,G,B) from the colour list. Here again, there might be several instances of the same function; at present we plan to have two instances of both the "draw polyline" and "draw strip" functions.

While the role of the "polyline" function is clear, the "draw strip" function needs some explanations. This function uses a list of points and a list of colour values to generate a series of adjacent triangles (see Figure 6). Such strips may be generated by the Host by decomposing polygons and they are also the natural outcome of a number of surface approximation methods. Using the two available entities the "draw strip" dataflow function can create a set of 3D

triangles to be sent to the Graphics Engine which, on its turn, will render the Gouraud shaded triangles on the screen. (We have to remark that this primitive appears directly in the functional specification of PHIGS+[11]).

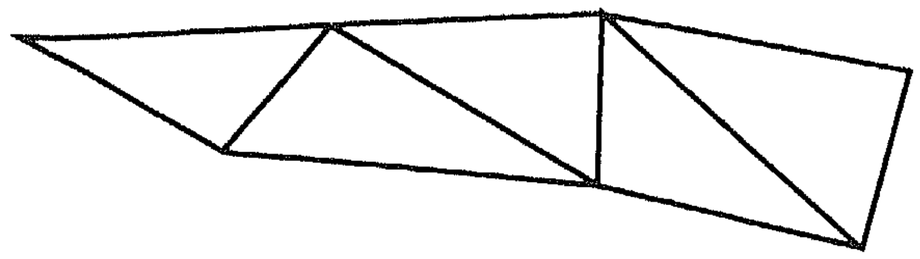


Figure 6. A strip

Figure 7 gives an overall picture of these basic dataflow functions. Not all functions are shown in the figure only the ones cited above. There are additional functions as well like "reset DFC", "set transformation matrix" and there is also a function aimed at speeding up pick input.

Clearly, the main advantage of this kind of software architecture resides in the fact that the different functions may be activated separately and they may run fully in parallel; in other words the structure level parallelism of the kind (1) and (2) listed in the previous paragraph are fully covered. Using this possibility of parallelism is of course the task of the Host; this is the unit which should have a full control over all objects on the screen, which should know which objects are to be retransformed, which ones have to be redisplayed only etc. In other words a rather complicated piece of software had to be realised to drive the whole DFC.

It is not possible to present here all details of all functions implemented in DFC. As an example, we will just give some details of the implementation of the transformation function; this will be enough, we hope, to give a flavour of the techniques used in programming the Dataflow Engine of DFC.

4.1.1 Details of a Dataflow Function: The Transformation Function

The primary task of the function is to transform four dimensional homogeneous coordinates. Mathematically, this transformation consists of two steps: the multiplication of a four dimensional vector with a 4×4 matrix and secondly, the projective division, that is the division of the x, y and z coordinates with the fourth w coordinate.

The particular difficulty in programming this function resides in the fact that the DFP processor has no division instruction. In other words, a separate dataflow program had to be written to implement this operation. Additionally, all arithmetic instructions are based on 16 bits arithmetic; unfortunately, however, 32 bits should be used for internal calculation, otherwise the resulting errors in the calculations would become excessively high.

However, all operations to be performed have a common feature, namely the fact that they can be performed on individual points of the point list absolutely independently from one another. In other words, there is a possibility of parallelism here which it is worthwileto exploit

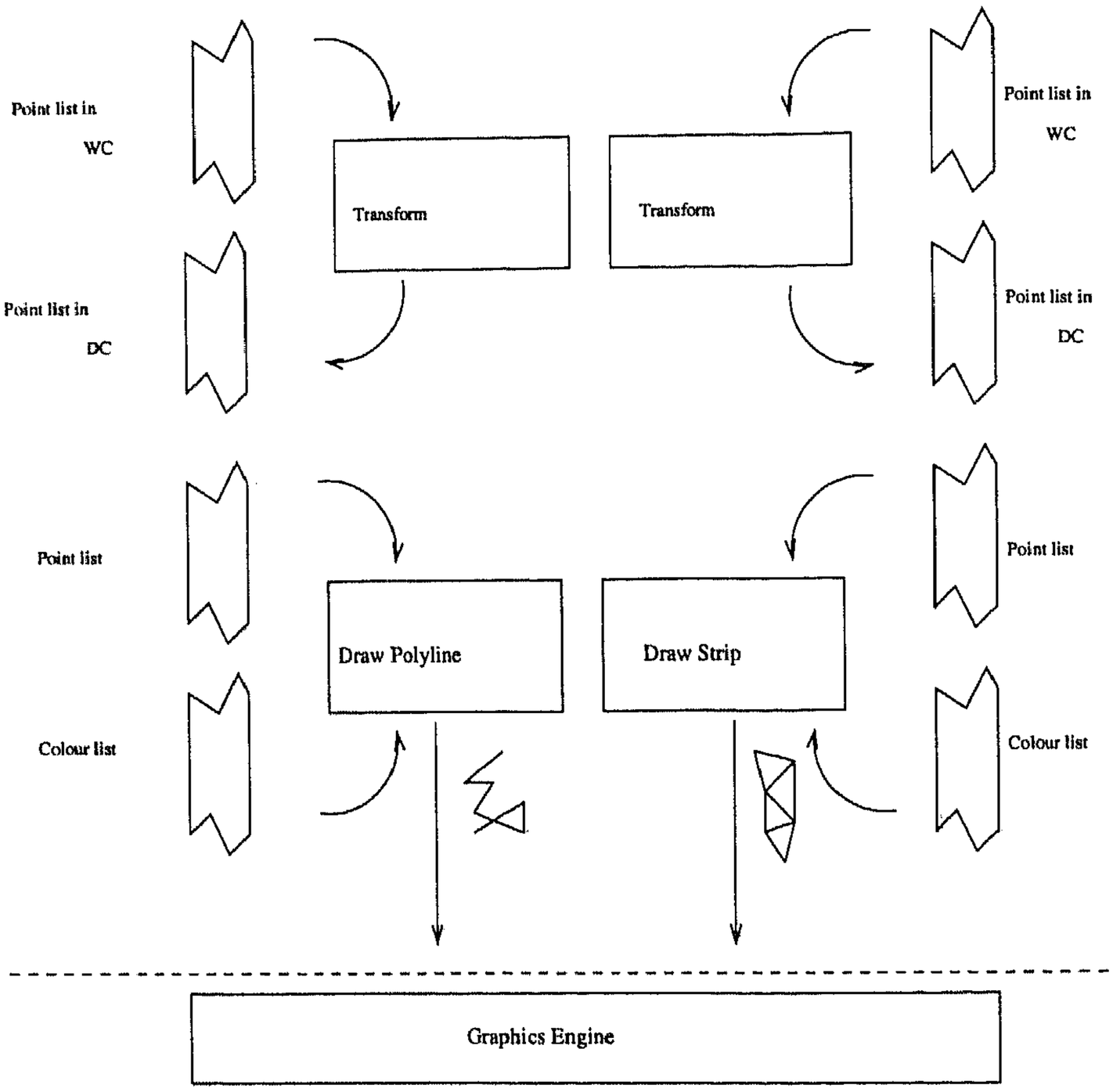


Figure 7. Overall structure of the DFC dataflow software

Figure 8 gives the overall view of how the transformation function is implemented. The function is spread over 7 DFP processors, each processor performing a specific task. All processors work in parallel and they form, as we will see, a kind of a transformation pipeline.

As it can be seen in the figure, some of the DFP processors perform functionally the same task; they realise several instances of the same function. The repartition of the tasks is as follows.

The "top" function, called the supervisor, actually reads the points from Image Memory. As we have already remarked, reading from memory is a time consuming task, it is therefore better to have a separate function for that purpose. The points are then dispatched to the transformation modules via the token network.

The transformation modules perform the matrix-vector multiplication, operating on 16 bits data but calculating, as a result, 32 bits data. The time needed by the DFP processor to perform this task is almost twice as much as reading the data from Image Memory; this is the reason why two instances of the same

function are in use. The two corresponding DFP-s get the next point alternatively from the supervisor; using this approach the two instances *together* still keep at the rate of reading produced by the supervisor.

At the lowest level we find four instances of a divider function; this latter has the task of performing the projective division on a point and to store the result in Image Memory again. The reason for using four instances altogether is the same as before: the time required by this function is roughly twice as much as for the transformer; by getting the point alternatively again, there is no loss of speed.

As a result, the speed of the whole transformation function is determined by the speed of the supervisor; actually, this latter one can produce ca. 150.000 points/sec. As we have seen before, the whole DFC contains, in the present version, *two* of these functions (that is all functions in Figure 8 are duplicated); this results in a theoretical possibility of transforming 300.000 points/sec.

Finally, each DFP of Figure 8 contains a dataflow program in a "classical" sense. A simplified portion of the matrix-vector multiplication may be seen in Figure 9; in fact, the matrix-vector multiplication may be considered as a typical example of a calculation which may be optimally structured for dataflow purposes. The reason why this dataflow net is relatively complicated is that the DFP instruction set is centered around 16 bits arithmetic whereas 32 bits is needed for our purposes. Here is a short description of the ne.

The *x, y, z* and *w* coordinates are duplicated 4 times in the first row to use them for the final *x', y', z'* and *w'* coordinate values. The multiplication instruction has the additional feature (not shown in the figure) that it may read values from internal memory "cyclically", in other words it will read first the element belonging to the first row of the matrix, then the second one, the third one and finally the fourth one. This ensures that after the multiplication the flow of tokens represent the right multiplicative factors.

The "multiply" as well as the "add" node may (on request) output two tokens: one for the high 16 bits value and one for the low 16 bits. These values must be successively accumulated to get the final high and low value of the result. This is what is done in the net of "add" nodes. Note that if only one output token is requested for an "add" or "multiply" node, this means the output of the low value only.

4.2 The Host software

As we have already mentioned, a rather complicated device driver is necessary on the host side to drive DFC. This device driver will serve as a basis for more elaborate software systems which are aimed at running on the Host. For this purpose, we plan to adapt GKS-3D[12], PHIGS+[11], and a three dimensional extension of the so-called radical system[13] on the top of this driver. As a later step, PEX[14] should also be ported, as soon as a public domain soft-

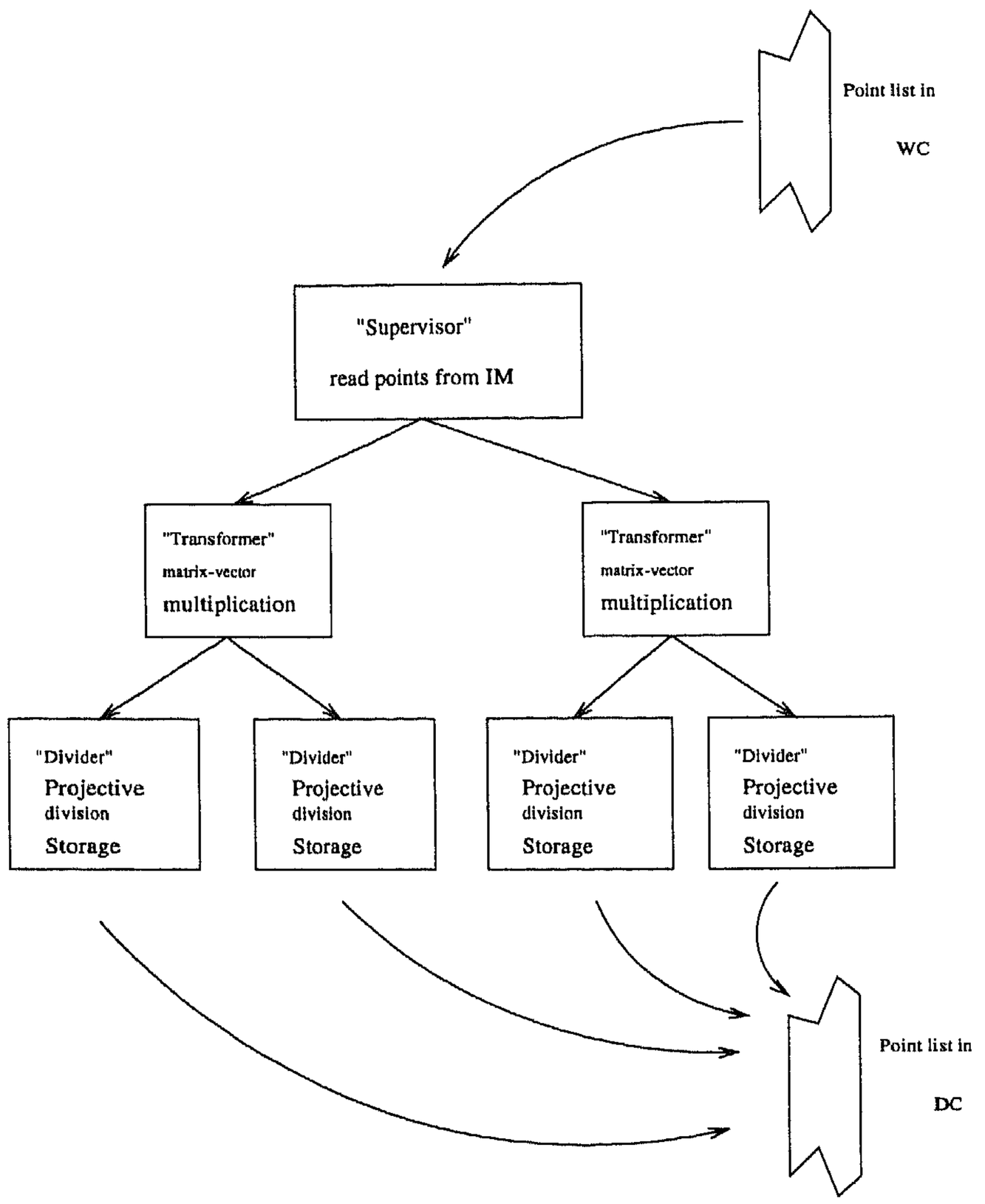


Figure 8. The transformation function

ware becomes available for that purpose. The shielding facility of the graphics engine gives a particular help for the adaptation of window-oriented environments like PEX. Details of these adaptations go far beyond the scope of the present paper (see also Figure 5)

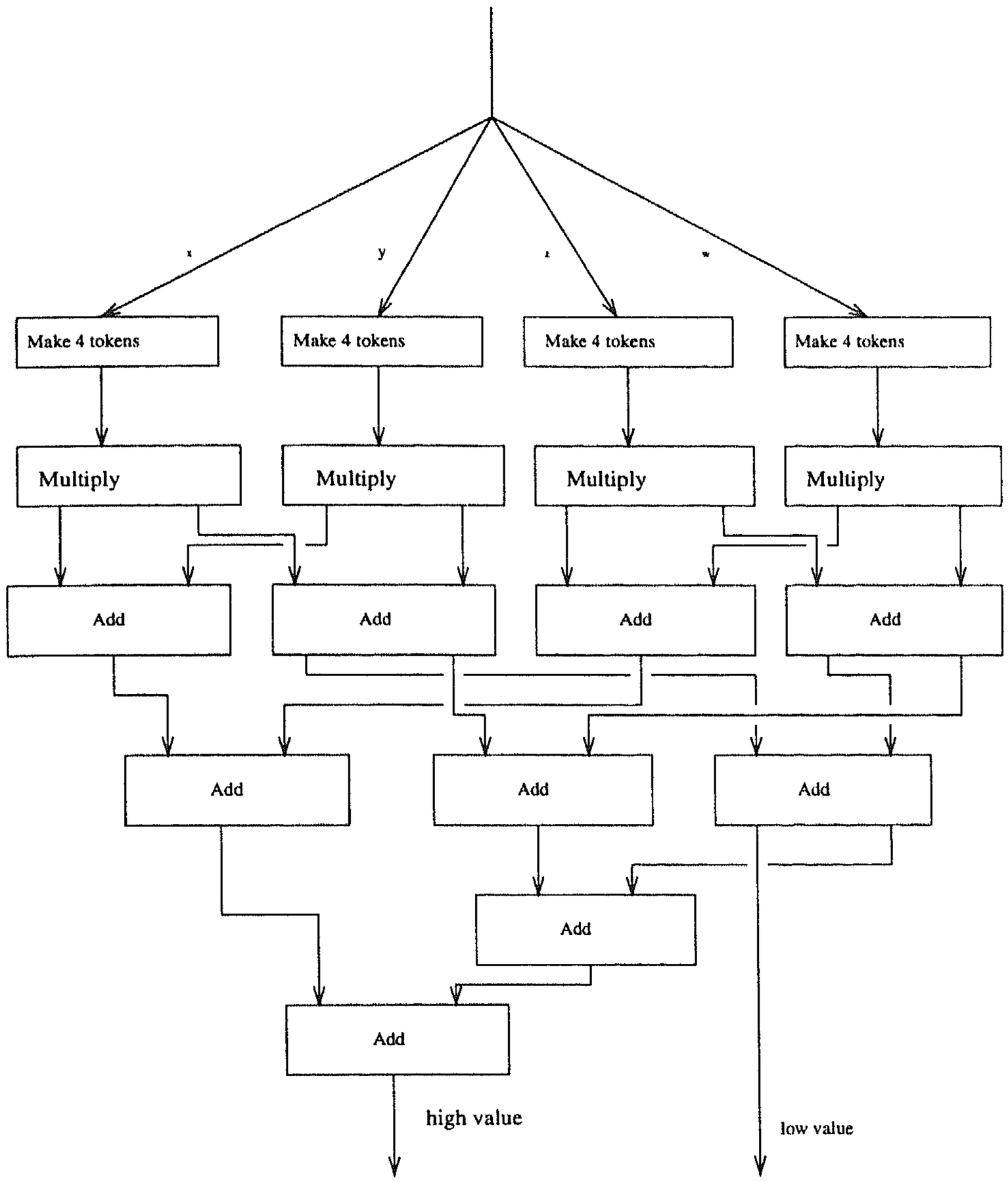


Figure 9. Matrix-vector multiplication in dataflow

5 Conclusions

.t It is fairly difficult to give an impartial evaluation on the performance of graphics workstations; there is a lack of reliable bench-mark software for this purpose. What we expect in our case is that the overall graphics performance of the DFC will be around 150.000 to 200.000 Gouraud shaded triangles/sec, with peaks raising up to 230.000 triangles/sec. This speed is comparable to

the highest end of the graphics super-workstations available on the market today.

The fact that the expected speed is not higher than the speed of some commercially available workstations should not be interpreted as a failure. All these workstations use well established technology, whereas DFC is the first exemple of a graphics workstation exploring the possibilities of dataflow technology as adapted for graphics. What we have to realise is that the basic processing element, that is the NEC μPD72181 itself is also the first commercially available dataflow processor on chip; in other words some improved versions are to be expected soon. Here are some aspects of such improvements we would welcome:

- The instruction set should be richer. In particular, division as well as floating point operations should be included.
- All internal memories should be significantly larger.
- A better internal structure should ensure higher on-chip parallelism.
- The overhead of accessing the Image Memory via the μPD9305 should be reduced (we should remark that a new and faster version of this chip is already in production, although still in prototype; the present hardware is prepared to use this chip instead of the older one).
- The way of connecting several processors should be improved. It should be possible to create large dataflow-nets where the fact whether a token is sent from one processor to the other or that it remains on-chip would be immaterial as far as software is concerned. This would allow a much easier way of programming the processor.

In addition to the improvements on the chip level itself, there might be improvements on the software level as well. Our personal experience in dataflow programming was practically non existent when starting the project and we have gained some experience only "run-time". It might well be that if we began reprogramming the whole DFC today, we could find new ways to improve the overall result just by using better programming techniques.

It is by itself an enormous advantage of this architecture that such reprogramming may be easily realized in contrast to an architecture which is based on custom design VLSI chips which is difficult and expensive to change. Furthermore, an eventual adaptation of the hardware-software environment for application-specific purposes is also easily realisable by reprogramming the dataflow processors. Merits of such flexible architectures have already been presented by a number of authors (see eg. England[15]).

By taking all these factors into consideration, an improved version of the DFP as well as a more experienced programming could raise the speed significantly; a fairly pessimistic estimation would still indicate a speed factor of 5 to 10, in comparison to the actual data. On the other hand, in addition to the improved speed of the already existing functions, new functions could also be added to the Dataflow Engine. e.g. some parts of the Host device driver, more elaborate shading calculations etc. It is almost impossible to measure exactly all impacts of such improvements.

Acknowledgements

We are grateful to all participants of the project who have contributed to the development of the ideas presented in the paper and who have also taken an active part in the actual development. We should cite the names of Frans van der Markt (DTN), Fons Kuijk, Bert Rouwhorst, Bèhr de Ruiter, Robert van Liere, Markus van Dijk (CWI) and Arthur Veen (Parallel Computing). Without their active role, this project would have no chance of success.

References

1. A.H. Veen, "Dataflow machine architecture". *ACM Comp. Surveys*, **18**, 365–396 (1986).

2. J. Hertah, Y. Yamagushi, N. Saito, and T. Yuba, "Dataflow computing models, languages, and machines for intelligence computations". *IEEE Trans. on Software Engineering*, **14**, 1805–1988 (1988).

3. NEC, μPD7281, *Image pipelined processor, product description*, NEC Electronics (1986).

4. S.S. Abi-Ezzi and M.A. Milicia, "An approach for a PHIGS machine". in *Data Structures for Raster Graphics*, L.R.A. Kessener, F.J. Peters and M.L.P. van Lierop (eds), EurographicSeminar Series, Springer Verlag (1986).

5. H.R. Finch, A M. Agate, A A.A. Garel, P.F. Lister and A R.L. Grimsdale, "A multiple application graphics integrated circuit MAGIC II", in *Advances in Computer Graphics Hardware II*, A.A.M. Kuijk and W. Straßer (eds), EurographicSeminar Series, Springer Verlag (1988).

6. J. Eyles, J. Austin, A H. Fuchs, A T. Greer and A J. Poulton, "Pixel-Planes 4: a summary", in *Advances in Computer Graphics Hardware II*, A.A.M. Kuijk and W. Straßer (eds), EurographicSeminar Series, Springer Verlag (1988).

7. I. Page, "DisArray: A 16 × 16 RasterOp processor", in *Eurographics'83 Conference Proceedings*, P.J.W. ten Hagen, North-Holland, 1983.

8. Th. Theoharis and I. Page, "Incremental polygon rendering on a SIMD processor array", *Computer Graphics Forum* **7**, 331–341 (1988).

9. S. Demetrescu, "High speed image rasterization using scan line access memories", in *Proceedings of the 1985 Chapel Hill Conference on VLSI*, H. Fuchs (ed), Computer Science Press, (1985).

10. P.J.W. ten Hagen, A.A.M. Kuijk and C.G. Trienekens, :Display architecture for VLSI-based graphics workstations", in *Advances in Computer Graphics Hardware I*, W. Straßer (ed), EurographicSeminar Series, Springer Verlag (1987).

11. ISO, *Information processing systems — Computer graphics — Programmers' Hierarchical Interactive Graphics System (PHIGS), Part 4, Plus Lumière und Surfaces (PHIGDisplay Functions, SPLUS), ISO/ IEC 9592-4, rev. 3 (1989)*

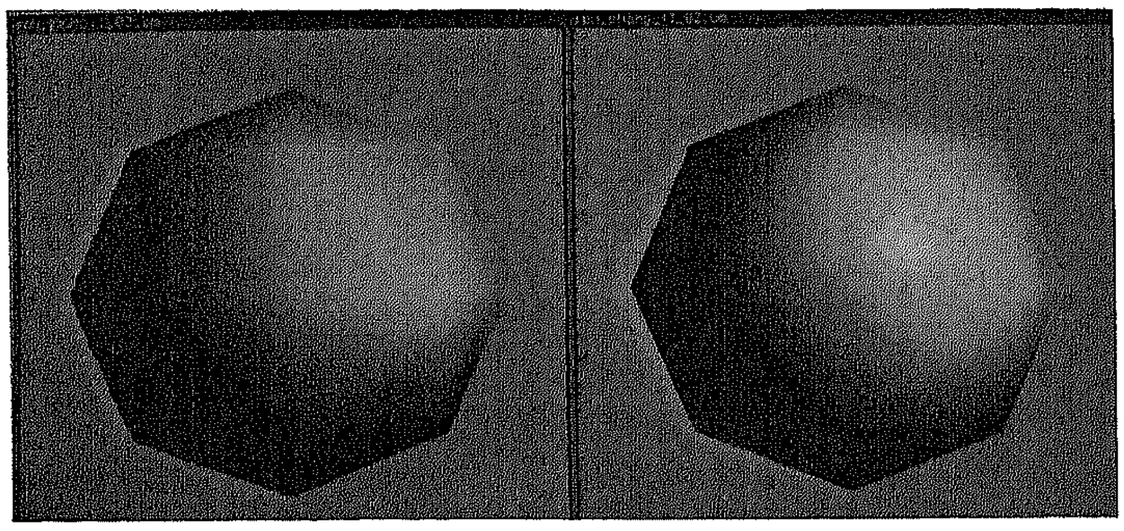
12. ISO, *Information processing systems — Computer graphics — Graphical Kernel System for Three Dimensions (GKS-3D), functional description*, ISO 8805 (1988).

13. P.J.W. ten Hagen and H.J. Schouten, "Parallel graphical output from dialogue cells", in *Eurographics'87 Conference Proceedings*, G. Maréchal (ed), North–Holland, (1987).

14. W. Clifford, J.I. McConnell and J. Saltz, "The development of PEX", in *Eurographics'88 Conference Proceedings*, D.A. Duce and P. Jancène (eds), North–Holland, (1988).

15. N. England, "A graphics system architecture for interactive application–specific display functions, *IEEE Computer Graphics and Applications*, **6**, 60–70 (1986).

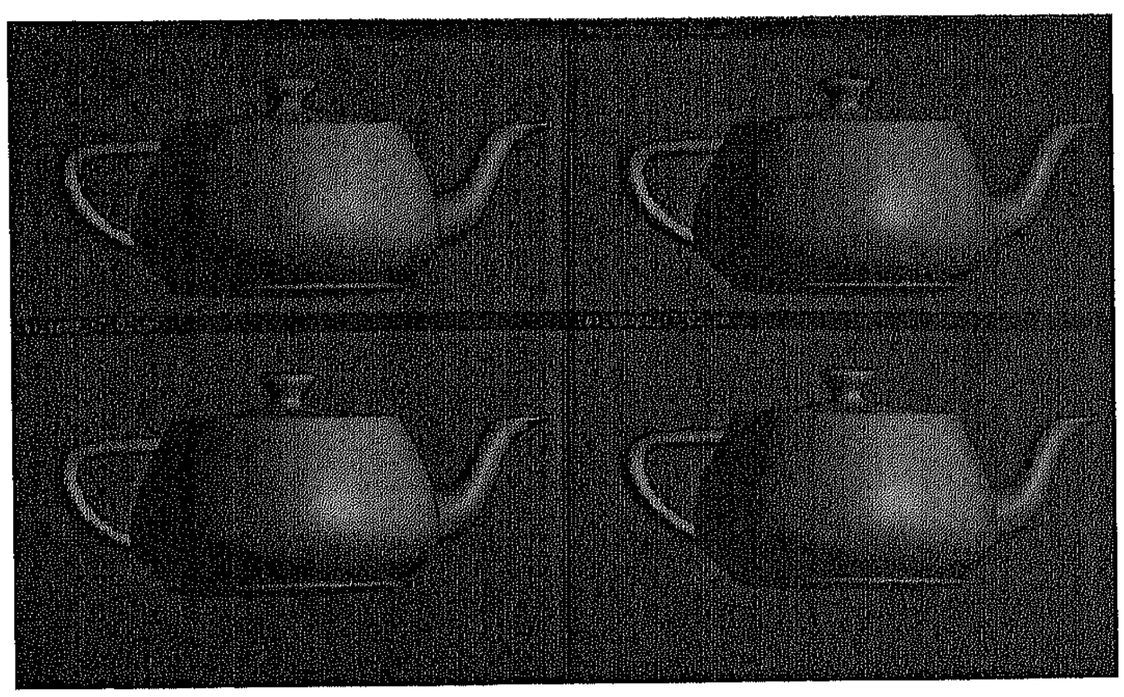
Faster Phong Shading via Angular Interpolation.



On the left a Gouraud shaded and on the right a Phong shaded image of a "sphere" approximated by 32 triangle patches. The light source is at a distance of four times the radius of the sphere. This and all following images have a parallel projection.

Obvious differences between these two images is 1) the amount of Mach banding, due to which the triangle patches are clearly visible for the Gouraud shaded image. This is not the case in the Phong shaded image. 2) the specular highlight is absent in the Gouroud shaded image.

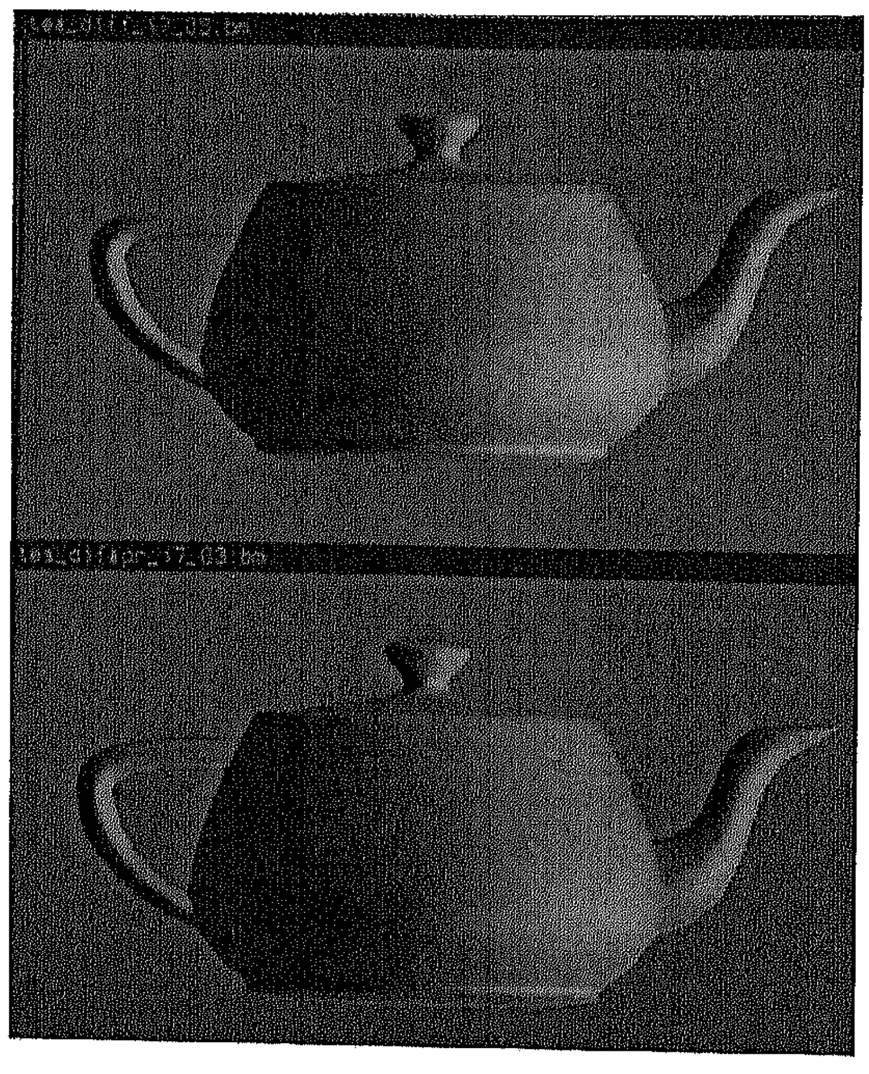
These differences become less when the patch size is reduced, but at the cost of having more patches.



Four images of the Utah teapot (3136 triangular facets). The top left is a Gouraud shaded image, the top right a Phong shaded image generated by the standard vector

interpolation method. For each pixel this method involves interpolation and normalisation of four vectors (two for the diffuse and two for the specular contribution) and evaluation of two scalar products. The diference in quality between these two images is obvious.

At the bottom, the leftmost image is generated by angular interpolation of the vectors, which involves interpolation of four angles (two for the diffuse and two for the specular term) and evaluation of a trigonometric function per pixel. Finally, the rightmost image is generated using the single vector approximation method as described in this paper. This involves interpolation of two angles (one for the diffuse and one for the specular term) and evaluation of a very simple trigonometric function. Although the computational costs of the latter two methods is less, there is no difference in image quality when compared with the image generated by the standard vector interpolation method.



Diffuse shaded images of the same teapot lit by two light sources. On top we see the result of calculation of the intensity by a cosine function. It shows Mach banding due to a sharp discontinuity in slope of intensity where the cosine reaches zero. For the bottom image the diffuse term is evaluated using the quadratic approximation method presented in this paper, resulting in a smooth transition to zero. This agrees with reality where natural effects cause a smooth transition as well.

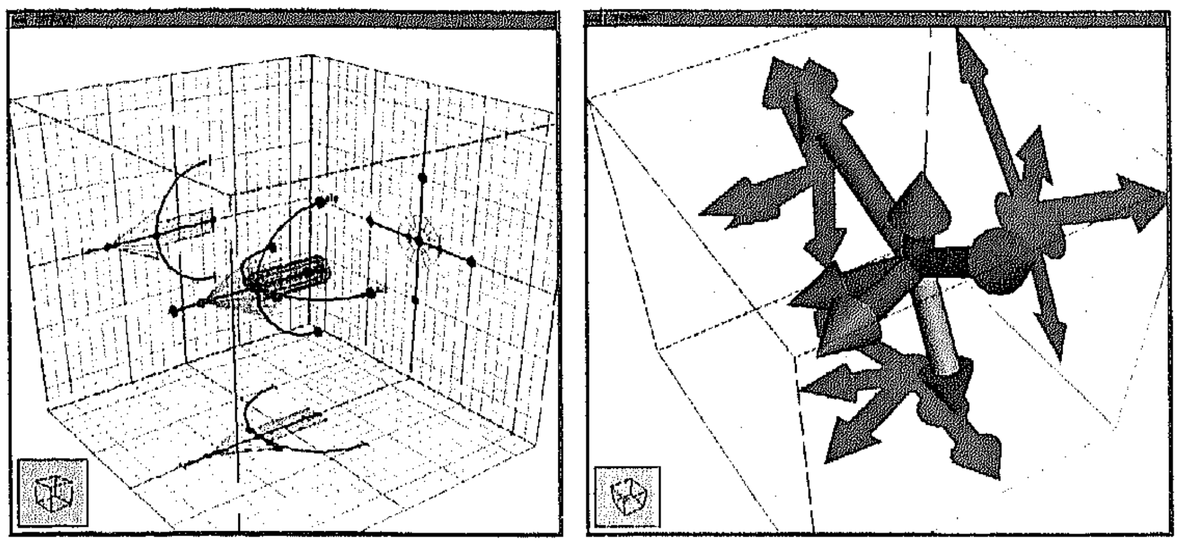


Figure 6: Arrow in edit-mode (left) and run-mode (right).

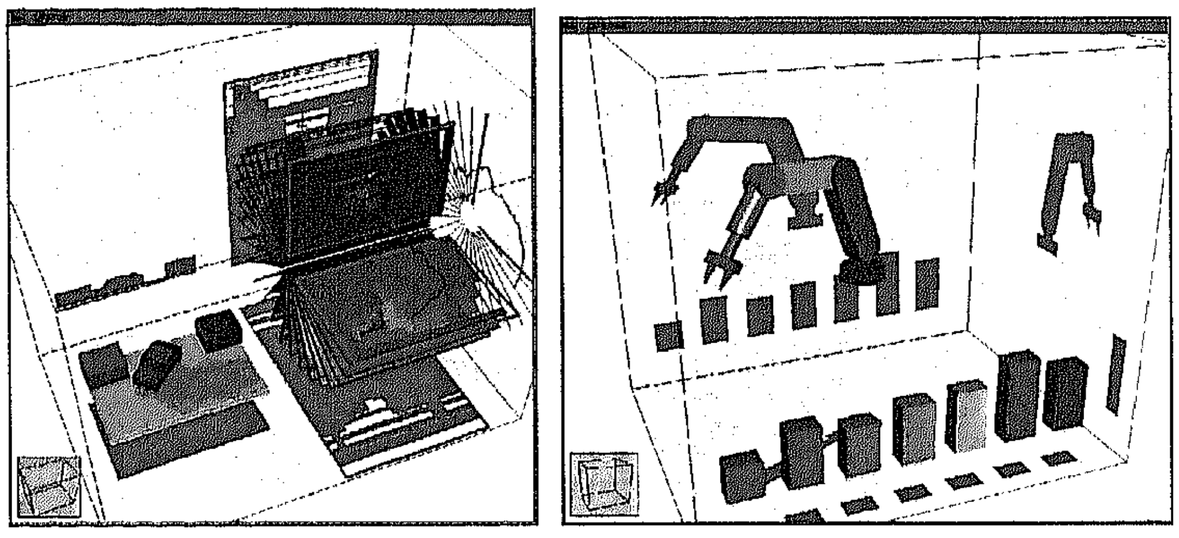


Figure 7: Path planning (left) and robot arm (right).

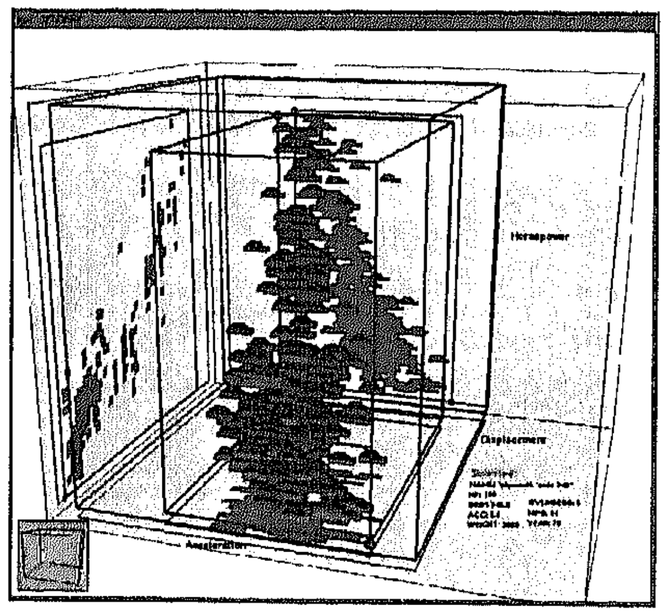


Figure 9: The front-end to a multi-dimensional database

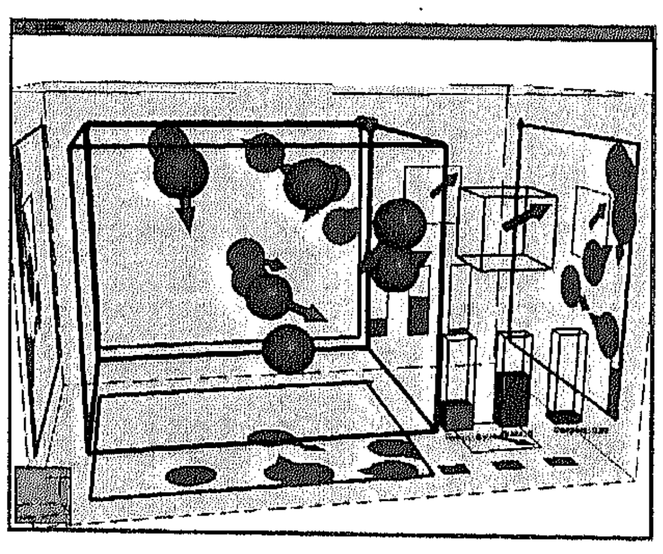


Figure 11: Simulation of bouncing balls.

G-815

ONTVANGEN 2 JUL 2003

69 k 33
00B60