Centrum voor Wiskunde en Informatica
Centre for Mathematics and Computer Science

P.J.W. ten Hagen, H.J. Schouten

Parallel graphical output from dialogue cells

Computer Science/Department of Interactive Systems Report CS-R8719 April

69K34, 69D13

Parallel Graphical Output from Dialogue Cells

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

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 parallel process can do output independently, but pictures can also be moved from one process to another. Each process has its own graphical output environment. The description and implementation of the system is based on GKS.

1980 Mathematics Subject Classification: 1.3.4, D.1.3.
Key Words and Phrases: computer graphics, parallelism, user interface management systems.

1. Introduction

In many modern applications, the user wants to interact with a program in a complex way. Interaction used to consist of a request and answer game via keyboard and screen respectively. Nowadays modern workstations are equipped with a large variety of input devices, that can be operated simultaneously. Input from one of these devices usually will result in some output. As the program cannot know which input, and consequently which output, will come first, it must be ready to produce any of them simultaneously.

Advanced feedback mechanisms will illustrate to the user how a given input parameter will influence the state of an application object *before* the actual choice is made. Several input parameters may be coupled to several input devices in this way simultaneously. The corresponding processes (one per parameter) will attempt to visualize the relevant object aspects independent from each other, hence they need independently to update the picture.

Contemporary graphics packages cannot deal with parallelism well. Usually the system is in some *state* at any time. The result of output calls depends completely on this state. However, if parallel processes want to do output they cannot make any assumption on the state, because a concurrent process may change it.

This report will be dealing with how pictures are structured, what kind of operations are needed and how these operations are achieved, in a system that can manipulate several constituents of one picture simultaneously. The description uses GKS [5, 9] as a starting point and explains the facilities in terms of GKS concepts. This implicitly suggests that parallel graphical output can be realized

Report CS-R8719
Centre for Mathematical and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

through a layer on top of GKS. However, more efficient realizations are possible.

The system is called the **radical system.** It was developed as a part of a project on a user interface management system, called **Dialogue Cells** [10]. To set the context of the 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 special 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 this 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 one hand be seen as 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 [2]. What actions are to be taken when a subcell returns a result 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. When a cell is activated, it gets the same screen space as its parent by default, or a window that it requests itself. Basic cells also have an input device assigned to them. Since availability of both windows and input devices is limited, these requests are handled by a global *resource manager* [4]. A more comprehensive introduction to the dialogue cell system can be found in [1].

In fig. 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 components 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.
- **Echo part,** which generates the picture changes that visualize the new state associated with the successful input parse. 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.

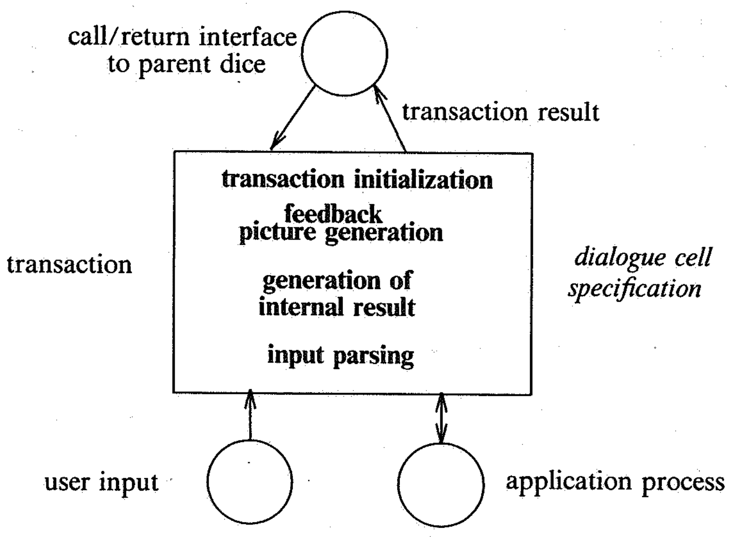

FIGURE 1. TRANSACTION SCHEME

1.2. The echo part of dialogue cells

When a subcell of a dialogue cell returns a result to its parent, several rules may be triggered by the parent. These actions are grouped in two distinct parts of a dialogue cell: the *value part* and the *echo part.*

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. [3]

The echo rules describe the visual effects on the screen that have to take place as a result of the return of a subcell. The subcell may, apart from the value result, also have an echo result, consisting of one or more pictures. These are handed over to the parent together with the internal result value. This *echo result* can subsequently be used in the echo rules of the parent.

As a result of such action rules new pictures may appear, existing pictures may disappear or change, either individually or in groups. The picture changes can be of many variations, e.g. shape, position, colour, size, etc. Moreover, as various dialogue cells are active concurrently such changes also may be generated concurrently or in arbitrary order.

1.3. 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 the units (segments) of a group. The parallelism implies that several groups might be altered simultaneously. Such parallelism can be implemented on top of GKS because this can be achieved through segment manipulation.

However, GKS has some drawbacks too, and dialogue cell requirements cannot directly be mapped onto GKS. This will be elaborated on in the sequel.

2. Functionality of output from dialogue cells

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. Each dialogue cell creates its own picture state, that we call *picture environment.* In this state (see fig. 2) the initial pictures (I), the own pictures (P) and the result pictures (R) are represented. Each picture can be manipulated in many ways, eventually producing the pictures that the cell will hand over to its parent. Note that during all these manipulations the pictures are visible to the user.

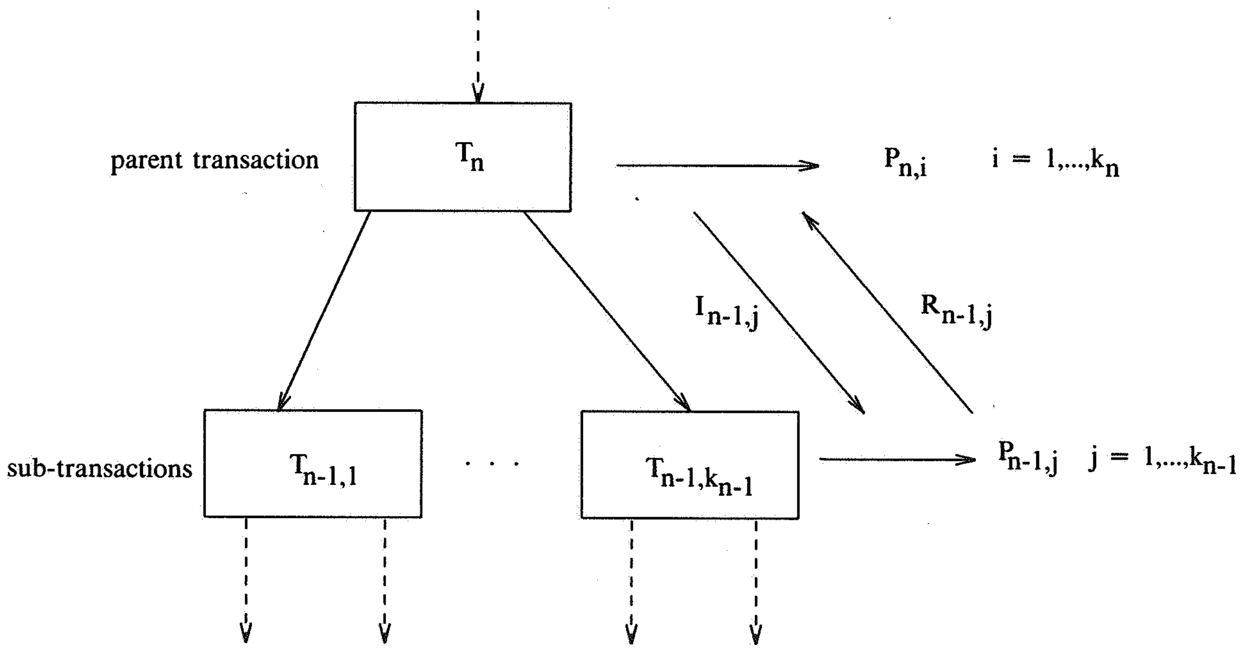

FIGURE 2. USER - SYSTEM TRANSACTION HIERARCHY

The programmer has a rich set of operators at his disposal to achieve the result he wants. On the other hand, many aspects of the picture and the picture handling do not need to be specified, as they are automatically performed by the run time system. In this way, the programmer has a great freedom in specifying what he wants to see and, on the other hand, does not need to write down every single aspect of the pictures.

2.1. Parallelism

An important aspect of the graphical output from dialogue cells is that it is parallel in nature. Firstly, there can be several dialogue cells active simultaneously, each producing their own pictures at the same time. Secondly, there can be multiple trigger rules in the echo part of one dialogue cell. The order in which they will be triggered is in general not known in advance. The first form of the parallelism is dealt with completely by the system. The second form has to be accounted for by the programmer, i.e. he must make sure that the operations he describes in the rules are "legal", regardless of the order in which they are triggered. For this purpose a mechanism is provided in the semantics of the dialogue cell language which can collect a number of new or changed pictures and precisely determine when this change has to become visible (including removal of obsolete elements). For instance, if several subcells are active that produce points, these will not be put into a polyline before all are accepted.

2.2. Inheritance

One of the advantages of a hierarchical control structure (e.g. subroutines or, in our case, dialogue subcells) is that it simplifies specifications. A hierarchy allows for installing a common state which is shared by all descendants of the owner of that state. At the same time each descendant may locally (temporary) adjust the state. As a result the effort to specify the appropriate state is minimized. This scheme can be implemented quite efficiently, not only in hierarchical graphics systems such as ILP [8] and PHIGS [7], but also in GKS. The major semantic action associated with it is called *inheritance.*

Inheritance is an important concept in the dialogue cell system. In many parts of the system it is used in a consistent manner, that is: if a programmer does not explicitly specify something, this aspect is inherited from the parent. It applies for example to resources, but also to graphical output environments. On the top level, the default attributes as provided by the workstation are used, on a lower level the attributes have the values as they were in the parent cell, when it activated the subcell.

By this mechanism, a subtree of the dialogue can have the same set of attributes. In general a subtree produces some type of graphical objects. Certain attributes can be used to indicate this type to the user. Using the concept of inheritance, the programmer does not have to specify these attributes in every dialogue cell of the subtree anew. At the same time, another subtree can have a different set of attributes.

2.3. Moving pictures to another environment

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 subcell (by parameter). The programmer cannot handle this himself, as he generally does not know in which environment the cell is activated, because it can be activated from different contexts. Nor is it considered a task of the programmer to specify this. Therefore it is not known beforehand to which environment a picture will belong. This implies that the picture be described in an environment independent way.

Furthermore, the actual appearance of a picture will depend on the environment it is in. If a picture enters a certain environment, it will for the aspects that are marked as relative adjust to the new environment. For instance the position or colour of a picture may change.

2.4. Operations on pictures

The dialogue programmer must have complete control over the pictures that exist in the environment. He can describe the pictures to any detail in terms of the workstation characteristics. The underlying system must ensure they will be displayed precisely according to the programmer's description.

Operations known from GKS like highlighting, transformations copying etc. must be available, but also edit operations, that allow the programmer to add, delete and change picture elements, and change attribute values. The programmer may change pictures by manipulating the environment state elements or he may change pictures individually.

From the discussion of chapter 2 it may now be clear that the graphics system which supports dialogue cells must be capable of:
- maintaining a hierarchy of picture environments
- changing more than one environment at the same time
- exchanging picture elements between environments

3. The radical system

3.1. 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, polymark, ...) 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 has 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 fig. 3 a structure scheme for radicals is given, together with how the various components can be realized in GKS (in italics).

radical	= radical attributes + list of primitives
segment	*segment attributes*
primitive	= name + type + geometry + attributes
	pickid, output primitive, primitive attributes
radical attributes	= visibility, highlighting, detectability, radical transformation

FIGURE 3.

3.2. 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 a 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 fig. 4 an overview of the attribute structure of a local state is given. Each of the entries can be set and will be initialized automatically.

environment name			
priority			
window-viewport			
group attributes		visibility detectability background colour shielding on/off group transformation	
polyline attribute bundle	polymarker attribute bundle	fillarea attribute bundle	text attribute bundle

FIGURE 4.

3.3. 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 concept 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 made 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 a situation that further picture making is possible without restrictions. For instance, no workstations will be deactivated or segments left open.

So any of the operations on radicals and local states are atomic. This ensures that no other process controlling an other local state can mess up the state of the workstation during the execution of the operations, so that the operations will always give the expected result, regardless of other graphical processing going on in parallel.

3.4. 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 be 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 form 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.

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.

Control functions	Changing the environment	Radical manipulation	Primitive manipulation	Inquiry functions
CREATE GROUP	SET LINETYPE	CREATE RAD	REPL. DATA	INQUIRE LS
DELETE GROUP	SET MARKERSIZE	SET VISIBILITY	CHANGE TYPE	INQUIRE RAD
CREATE LS	SET AREA STYLE	SHIFT RAD	SET IND. LINETYPE	INQUIRE PRIM
DELETE LS	SET TEXT HEIGHT	CONCAT RADS	SET IND. MARKERSIZE	
UPDATE ENV	...	REMOVE PRIM	REMOVE IND. ATTR.	
SEND RAD		ADD POLYLINE	...	
ACCEPT RAD		...		

FIGURE 5.

3.5. Use of the radical system in the DICE system

The dialogue cell run time system performs the control functions of the radical system automatically, i.e. the programmer needs not specify these himself. When a dialogue cell is activated it first gets a viewport and then a local state. The resource manager guarantees that two overlapping viewports will have different priorities. The update function will automatically be called at appropriate times. When a dialogue cell is deactivated the environment is deleted from the system.

4. Implementation

4.1. 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. This 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 not the same as GKS segments for the following reasons:

- segments are stored within a fixed environment. They cannot be moved to another (see [6]).
- segments cannot be edited, only added to. Once closed, their contents are fixed. 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 deferal mode 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).

4.2. Atomicity and parallelism

Race conditions are avoided by using a mutual exclusion mechanism for the environments. Local operations on an environment therefore pose no problems towards atomicity. For the update of the environment the access is temporarily given to the workstation. Some care has to be taken when moving a radical from one environment to another. First the radical is taken out the environment, then the access to the environment is released. Next, access to the destination environment is achieved and the radical put into it. If access to he own environment is not released deadlock may occur.

4.3. 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 into 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 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 whether it has changed, so that a new segment has to be created for it.

4.4. 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 the 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 fashion. 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 lengthy 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 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 own segment administration, no longer need segments. The data 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.

5. Conclusion

5.1. 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 *viewport1* = { 0.0, 0.0, 0.6, 0.6 } and the second in *viewport2* = { 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 the 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 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.

```
grp1 = newgroup(window1, viewport1, low_prio);   /* create first viewport */
grp2 = newgroup(window2, viewport2, hi_prio);    /* create second viewport */
ls1 = alloc_ls(NIL, grp1);                       /* create local state in viewport1 */
ls2 = alloc_ls(ls1, grp2);                       /* create second local state as child of ls1 */
```

FIGURE 6.

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 new radical */
 6  id = radfill(3, points, rad, ls1);  /* add fillarea to radical */
 7  s_rad_fa_is(ls1, rad, id, HATCH);   /* set its interior style to hatched */
 8  ls_update(ls1);                     /* update the environment */

 9  s_def_pl_lt(ls2, LTDOTTED);         /* set line type in local state */
10  rad = newrad(ls2);                  /* create new radical */
11  id = radpol(9, points, rad, ls2);   /* add polyline to it */
12  ls_update(ls2);                     /* update the environment */
```

FIGURE 7.

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

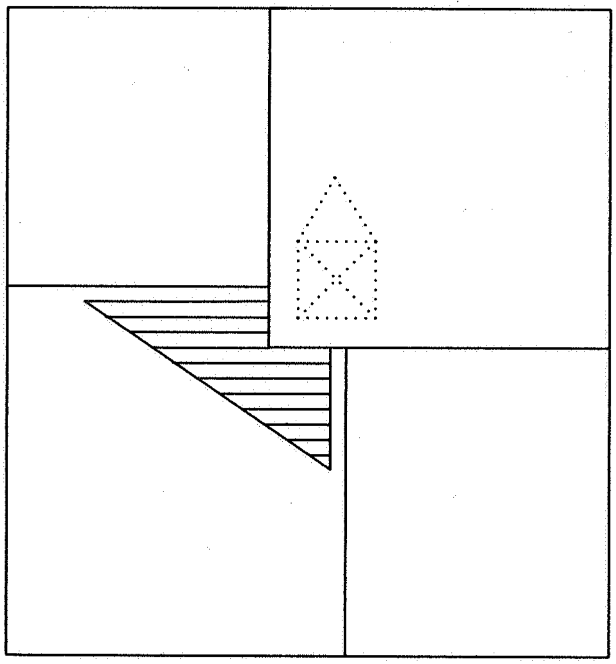

FIGURE 8.

Now ls2 may decide to pass the radical it created to its parent, by calling *rad_out_ls(rad, ls2)*, and passing a pointer 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 fig 9.

In the context of the dialogue cell system, the initial set up of the environments is done automatically; the programmer only needs to specify the viewports that are to be used. Also the programmer

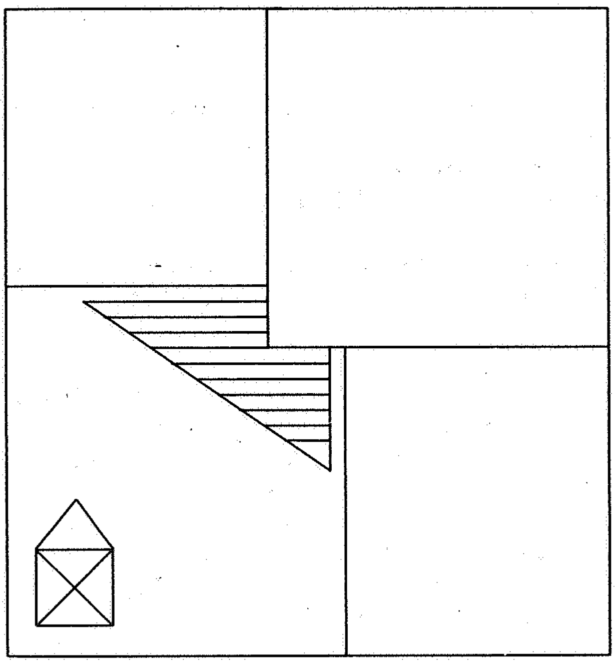

FIGURE 9.

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.

5.2. 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 we found with an 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 with 68%. The object of the 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.

5.3. 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 a dialogue cell independent from other dialogue cells. This allows for easy use of libraries of dialogue cells: no knowledge of the influence of a library dialogue cell 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 they 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.

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 differences in the semantics of the respective hierarchies as well as the parallelism greatly reduce 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. P.J.W. ten Hagen, R. van Liere: *Introduction to Dialogue Cells*
CWI report CS-R8703, 1987

2. H.J Schouten: *Basic dialogue cells*
CWI report, 1987 (to appear)

3. P.J.W. ten Hagen, W. Eshuis: *Value mapping in Dialogue Cells*
CWI report, 1987 (to appear)

4. R. van Liere: *Resource management in DICE*
CWI report, 1987 (to appear)

5. *Graphical Kernel System Functional Description*
ISO/TC97/SC5/WG2

6. P.J.W. ten Hagen, M.M. de Ruiter: *Segment grouping, an extension to the graphical kernel system*
CWI report CS-R8623, 1986

7. *Programmers Hierarchical Interactive Graphics System*
ISO/TC97/SC21, 1986

8. P.J.W. ten Hagen et al.: *ILP, Intermediate Language for Pictures*
Mathematical Centre, 1980

9. F.R.A. Hopgood, D.A. Duce, J.R. Gallop, D.C. Sutcliffe: *Introduction to the Graphical Kernel System*
Academic Press, 1983

10. H.G. Borufka, H.W. Kuhlmann, P.J.W. ten Hagen: *Dialogue Cells: a Method for Defining Interactions*
IEEE Computer Graphics and applications, july 1982

11. M. Green & R. Phelp: *An Object Oriented Language for Graphics Programming*
Proceedings of Graphics Interface 1982

12. S. Guest & E. Edmonds: *Graphical Support in a User Interface Management System*
Proceedings of Eurographics 1984