On Scripting in Distributed Virtual Environments

Jan P. Springer Henrik Tramberend Bernd Fröhlich

IMK.VE

German National Research Center for Information Technology

Abstract

This paper presents tools and techniques to support scripting interfaces in distributed virtual environments. Our main contribution is the introduction of a shared name space, which allows scripts to access shared objects in remote execution contexts as if they were local. This greatly simplifies the development of scripted distributed applications. We successfully applied our concepts to migrate a geo-scientific application to support distributed exploration of large data sets.

Introduction

Various virtual environment systems (e.g. DIVE [1], Lightning [2], Avango [3]) provide scripting language bindings for rapid prototyping support and flexible application development. In these systems most of the application and interaction semantics are implemented using the scripting interface. Distributed virtual environment systems provide support for efficient and transparent data distribution. However, developing distributed application semantics remains often considerable work. Since scripting has become a valuable tool for stand-alone application development, it is promising to use it in a similar way for the distributed case.

In this paper we describe an approach for transparent scripting support in distributed virtual environment applications. Our work is based on Avango [3], a framework for distributed virtual environment applications. Avango achieves data distribution by transparent replication and synchronization of objects shared among the participants in a distributed application. Avango has a complete language binding to the interpreted language SCHEME [4]. Applications in Avango are typically a collection of SCHEME scripts, which create and manipulate Avango objects and define relationships between them. The key component of our distributed scripting support is a shared name space. Each participant in a distributed application can add symbols to a distributed symbol table. These global symbols are then automatically added to the local name space of all participants in a distributed application. This keeps the name spaces for all participants synchronized and allows scripts to access distributed Avango objects as if they were local.

Recently we extended a stand-alone geo-scientific application prototype [5] to work for the distributed case. The application prototype allows users to explore large seismic volumes, complex subsurface structures, and multivariate well logs of physical properties down a bore hole. Since typical data sets are in the range of hundreds of megabytes to several gigabytes, the data is usually predistributed and loaded from local disks when joining the distributed application. We control the navigation and many interaction activities through the Cubic Mouse [6], a cube-shaped physical prop representing a seismic volume. Additionally, a virtual tools-based approach for defining different interaction modes is used. Both interaction concepts make heavy use of the scripting interface and we describe their implementation for the distributed case.

The main contribution of our work is the introduction of a shared name space for scripting interfaces in distributed virtual environments. We also report on the lessons learned from migrating a geo-scientific application to support distributed exploration of large data sets. The most interesting result of our work is that our concept makes the development of distributed applications using a

scripting language basically as simple as the development of stand-alone applications.

Avango

Avango [3] is a programming framework for distributed, interactive virtual environment applications. It provides a shared scene graph, based on IRIS|Performer [7], accessible from all processes forming a distributed application. Objects in Avango are field-containers, representing object state information as a collection of fields, similar to Inventor [8]. These objects support a generic streaming interface for storing and retrieving an object and its state to and from a stream. This is the basic building block for object distribution.

Avango's internal network distribution layer provides total ordering of messages and a shared group abstraction. A shared group can be seen as a named multi-cast address where any process knowing the name of a group may join. Attaching to a non-existent group will automatically create the group and join the process.

While objects in Avango are implemented using the C++ programming language [9], Avango also features a scripting environment used for customizing the runtime environment (e.g. setup of input and output devices) and developing applications. The scripting environment has a complete binding to the Avango C++ API using the interpreted language SCHEME [4].

Stand-alone Application Development

Applications in Avango are developed by using the scripting interface. See figure 1 for a "simple" application script in Avango. In this example an `avDCS`[1] is created and passed to a chess board making function, which is omitted here for brevity. Adding the `chess-root` to `av-scene-root`, a DCS predefined by the runtime environment, actually creates a link to the scene root and makes the board visible.

Avango's scheme interpreter is available in a shell-like command line execution environment. Loading the script in figure 1 into the scheme interpreter executes the script and brings up the chess board on the output device.

[1] DCS stands for dynamic coordinate system, i.e. a transformation matrix.

```
(define chess-root (av-make-instance-by-name
                    "avDCS"))

;; loads chess pieces and chess board
;; add them into "chess-root"
(add-pieces chess-root)

;; add chess board into the scene
(av-add-1value av-scene-root 'Children
                             chess-root)
```

Figure 1: A "simple" stand-alone Avango scripting application.

Distributed Application Development

In figure 2 on the following page we show a refined version of the stand-alone script from figure 1. Note that we only had to change the chess board's root node type from `avDCS` to `avNetDCS`. An `avNetDCS` is a specialization of an `avDCS` extending it to distribute all of its child nodes to the shared group it is attached to. The actual joining or creation of the shared group takes place by setting the `GroupName` field to a non-empty string value. If the shared group with that name exists the `avNetDCS` joins that group otherwise the group is created. Any participating process in the shared group will "see" the tree of child nodes below its own `avNetDCS`. The `avNetDCS` can be seen as a service responsible for sending to the participating processes as well as for listening to changes from the participating processes. The `avNetDCS` is the first scene graph element, which is implicitly shared among all participating processes in the shared group. Also, viewed from outside the shared group it represents the group's shared resources as a whole.

Starting multiple instances of the script from figure 2 on the following page does not make real sense, however. The result would be that each client loads a chess board and distributes it. In the end there would be as many chess boards as there are participating processes. To have only one chess board seen by all clients we will have to start the script from figure 2 on the next page only once and for any other client we will use the script from figure 3 on the following page. Note that the script in figure 3 on the next page does not have any code to create a chess board by itself. The joining processes import the shared

```
(define chess-root (av-make-instance-by-name
                    "avNetDCS"))

;; loads chess pieces and chess board
;; add them into "chess-root"
(add-pieces chess-root)

;; add chess board into the scene
(av-add-1value av-scene-root 'Children
                             chess-root)

;; create/join group
(av-set-value chess-root 'GroupName "dchess")
```

Figure 2: Distributed version of the "simple" Avango scripting application from figure 1 on the page before.

```
(define chess-root (av-make-instance-by-name
                    "avNetDCS"))

(av-add-1value av-scene-root 'Children
                             chess-root)

;; create/join group
(av-set-value chess-root 'GroupName "dchess")
```

Figure 3: Client version of the Avango application script from figure 2.

scene graph simply by creating an `avNetDCS` and setting the `GroupName` field.

The distribution concept behind the `avNetDCS` abstraction is not limited to client/server applications. It is possible and feasible to create resources at each client site and distribute these among the shared group. Even in a client/server application the server does not have to be the group creator. The first participating process creates the group and the following participants just join the group. It does not matter to the distribution concept which client (or server) introduces what to the group.

Today, shared scene graphs are a common concept. In scripting environments it is also necessary to share the actual scripts for the application logic in such a way that the states and context information for these scripts are shared as well. Many scripting languages represent the context and state information through global variables. To allow applications to share global variables, we have introduced a shared name space concept into Avango. The shared name space used in Avango is implemented as a map or symbol table associating a variable name with its memory object, more precisely the memory location of an Avango object, in the process local address space. Adding or removing a symbol from the map or changing the memory object of a symbol will post the new state of the map to the participants in the shared group. On reception of a symbol table a client executes a local script which looks up any symbol to be defined in its own scripting runtime environment. If the symbol is not found, it is created and set to the value of its associated (local) memory object. If the symbol already exists, it is updated. Since the memory objects in the map are Avango objects the internal evaluation mechanism keeps their states consistent.

```
(require 'gdefine)

;; create a non-shared variable
(define local-dcs (av-make-instance-by-name
                   "avDCS"))

;; create a shared variable
(gdefine 'shared-dcs (av-make-instance-by-name
                      "avDCS"))
```

Figure 4: Defining local and global variables in Avango.

Creating a shared variable in the scripting environment works basically the same way local variable creation works. We defined a wrapper around an actual `define`, called `gdefine`, that stores the name and the memory object of the definition into the (process local) symbol table, after creating the local variable itself. Figure 4 shows the creation of a local and a shared variable. Note that the syntax for creating a shared Avango object remains similar to the syntax for creating a local Avango object.[2]

Distributed Interaction

Avango supports a variety of concepts for interaction. The

[2] In the example we have to write the name of the shared variable as `'shared-dcs` to prevent the SCHEME interpreter from evaluating it. `gdefine` is implemented only as a function (scope).

most commonly used concept is the virtual tools based approach [10]. Virtual tools can be used to navigate in the environment, but here we focus on how virtual tools interact with shared virtual objects in a distributed scene graph. In addition to the virtual tools based approach an event driven interaction mechanism can be used, which is triggered by field changes in Avango objects. These objects may represent input devices or arbitrary objects in the scene graph.

Virtual Tools

Avango's approach to virtual tools is based on three different entities: virtual tools, mediators, and interaction operators.

Virtual tools define the basic interaction mode and they are the interface from the real world into the virtual world. They allow users to perform specific tasks like rotating, scaling, or changing the color of Avango objects. Tools are typically attached to input devices like tracked wands, which contain 6DOF sensors and other inputs from the real world like buttons or potentiometers. Virtual tools are local entities, because input devices are local resources. Each active tool has an individual graphical cursor, which follows the corresponding input device. Cursors are geometric objects representing the active tool (e.g. a pointing ray emanating from the tip of the tracked wand). These cursor representations are distributed as a simple avatar showing what remote users are doing.

Mediators are attached to Avango objects in the scene graph and specify at which level in the scene hierarchy an interaction may take place. Mediators also provide the interface to the scene graph through which an interaction occurs and they carry object specific interaction information. Mediators are distributed with the Avango object they are attached to. This allows the same interactions for the local environment, where the virtual object and its mediator were created, as well as for remote environments.

Interaction operators implement the interaction functionality. They take tools and mediators as their operands. Interaction operators are temporary objects. They are instantiated for each interaction and discarded afterwards. A two-dimensional matrix defines for each pair of tool and mediator type a default interaction operator type if there is any. In our current implementation interaction operators are local objects to avoid network latency for the actual interaction.

One of the most commonly used mediators is a script mediator, which references a SCHEME script by name. For example when the user holds a pointing tool and clicks on an object with an attached script mediator, an appropriate interaction operator is instantiated, which executes the script referenced by the mediator.

Scripting Interactions

Our interaction concept allows a variety of approaches for scripting interactions. Before describing the problems and possible solutions for the distributed case we will explain how common stand-alone applications approach the issue. In the following we will distinguish between two kinds of SCHEME variables. SCHEME variables referencing SCHEME data structures and SCHEME variables referencing Avango objects.

```
(define global-variable '())

(define (callback-with-arguments data)

 ;; work with "global-variable" and "data"
)
```

Figure 5: Scripting interaction in a stand-alone application.

Figure 5 shows a callback script skeleton commonly used in stand-alone Avango applications. Such a script usually depends on the given argument, an Avango object, and global SCHEME variables. Using such types of scripts in a distributed application will not work, because SCHEME variables and SCHEME structures exist only in the run time environment of the SCHEME interpreter. Avango knows nothing about those SCHEME variables and SCHEME structures. Therefore it cannot provide a distribution mechanism for them.

A general solution for distributing SCHEME structures would require developing concepts for distributed scripting languages, which is beyond our scope. Instead, we chose the solution to use different programming paradigms which avoid accessing global SCHEME structures from within a script execution contexts.

A first alternative approach is exemplified in figure 6. Here a script receives its "execution context" with its arguments. The supplied mediator contains a Parameter field, a list of references to Avango objects. The obvious problem with this approach is that the ordering of the parameters in the argument does matter. This means the implementor of the script and the user of the script must agree on the order and the types of the parameters given to the script.

```
(define (callback-with-parameter-mediator m)

 ;; extract the parameters from the mediator
 (let (
  (param1 (car (av-get-value m 'Params)))
  (param2 (cadr (av-get-value m 'Params)))
  )

  (begin
   ;; work with extracted parameters
  )
 )
)
```

Figure 6: Scripting interaction using the callback and callback data approach in a distributed application.

The above mentioned approach may fail for problems where we want to access a shared Avango object from within the local SCHEME environment of a client application. For example, our SCHEME scripts often use hooks into the scene graph to manipulate Avango objects, but Avango distributes the scene graph anonymously. The hooks can be added to our symbol table, which introduces them to the local address space of the clients.

Figure 7 shows two scripts working on shared variables. One advantage of this approach is the support for "legacy code." The script logic as well as the argument convention of scripts in stand-alone applications and distributed applications are similar. The only "real" change is the addition of a "logical null pointer check", (**if** (**bound**? shared-variable) ...), to ensure the prerequisites for the script are fulfilled.

```
(define (callback-no-arguments)

 (if (bound? shared-variable)
  (begin
   ;; work with "shared-variable"
  )
 )
)

(define (callback-with-arguments data)

 (if (bound? shared-variable)
  (begin
   ;; work with "shared-variable" and "data"
  )
 )
)
```

Figure 7: Scripting interaction using the shared variable approach in a distributed application.

A Distributed Geo-Scientific Application

We are working with a consortium of oil and gas companies and software vendors for this domain to explore VR technology for this application area. Our mission within the consortium is to develop a visualization prototype, which allows consortium members to explore and manipulate complex data sets in a virtual environment. Experts in oil and gas companies are often distributed all over the world, so there is a large potential for distributed decision making supported through VR technology. After developing a stand-alone visualization prototype [5], we have extended our system to work for the distributed case.

First, we describe the basic features of our stand-alone visualization prototype. Then we explain how we applied the concepts for the distribution described in the previous sections to this particular case. We need to emphasize that our system has been developed for only a few participating sites, typically two or three. Our ideas and concepts by no means scale for a larger number of participants, which is really not a problem for this application domain. Typically there are only two sites connected, which work together on a single data set and discuss some issues in detail.

The Stand-alone Demonstrator

The seismic cube is the central data structure for most exploration and interpretation tasks. Subsurface structures like horizons and faults are defined relative to the seismic cube and typically displayed as polygonal models. The traditional way of representing the seismic volume is through three orthogonal slices called cross-line, in-line and time slice.

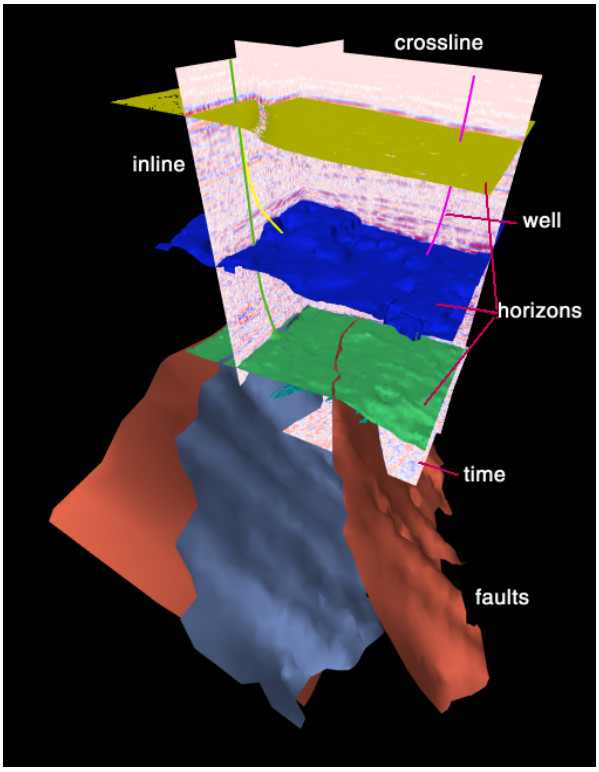

Figure 8: A typical oil exploration data set containing subsurface structures, wells, and seismic slices.

Figure 8 shows a typical oil exploration data set. The subsurface model consists of two main structures: horizons and faults. Horizons separate two earth layers, and faults are breaks in the rocks, where one side is moved relative to the other. Horizons are typically horizontal while faults are inclined. Three orthogonal slicing planes are used to visualize the seismic volume. The in-line slice is typically perpendicular to the main fault direction. The time slice is horizontal and the cross-line slice is perpendicular to both.

We developed the Cubic Mouse [6] for navigating in the seismic volume and for positioning the three slices. This cube-shaped, tracked input device, shown in figure 9, mimics the shape of the seismic cube. The Cubic Mouse is tracked with a 6DOF sensor and the orientation of the seismic cube follows in sync, effectively placing the seismic cube in the user's hand. Rotating the Cubic Mouse rotates the seismic cube. Moving the Cubic Mouse moves the seismic cube in the same direction. Since the other structures like horizons, faults, and wells are defined relative to the seismic cube, they move with it.

Figure 9: The Cubic Mouse device.

As one can see in figure 9 the Cubic Mouse has three sliding rods passing through it. Each of these rods positions one of the traditional seismic slices. The slices are

aligned with the faces of the Cubic Mouse and stay in sync as it moves, so the rods are always perpendicular to the slices.

There is a total of six buttons on the top face of the Cubic Mouse. The single button is used as a clutch allowing users to detach the Cubic Mouse from the model. Two buttons in another corner control the size of the model. The three buttons in a third corner are not used.

In summary the Cubic Mouse is used to position, orient, and scale the model as well as for moving three orthogonal cross sections through the seismic volume. In addition to the Cubic Mouse a virtual tools based approach is used. A tracked wand is used to pick up different tools from a virtual tool-bar to perform different tasks. The following virtual tools are available:

- The level-of-detail tool allows users to toggle the displayed level of detail between low and high for horizon and fault surfaces.
- The drag tool is used to drag around surfaces and drop them off in a new location.
- The snap tool is similar to the drag tool, but when a surface is released it snaps back to its original location.
- The information tool shows the name and position of an object when pointing to it.
- The slicing tool attaches a slicing plane to the tracked wand. This slicing plane can be moved through the seismic data set by moving the wand.
- The volume rendering tool attaches a volume rendering lens to the tracked wand.

Most of the geo-scientific application was developed using Avango's scripting language SCHEME. Only a few node types had to be added to implement new functionality like volume rendering for large volumes and the display of multi-attribute data along well paths.

The Distributed Demonstrator

The size of our data sets, which contain volumetric seismic data and polygonal data for horizon and fault surfaces, are typically in the range of a few hundred megabytes to several gigabytes. Our setup is laid out to work with pre-distributed data sets to avoid long startup times. The actual scene graph is loaded by one of the applications and distributed through Avango's previously described distribution mechanism.

The Cubic Mouse controls the position, orientation, and size of the model in the stand-alone version. For the distributed case we have two or more Cubic Mouse devices involved, which would potentially control the model. Since the model belongs to the shared scene graph, we could only allow one site to control the position and orientation of the model at a time. We found this to be too restrictive and decided for an approach where each site may control the model individually with the option to synchronize and stay in sync with any other site at any time. We realized this concept by transforming the Cubic Mouse controlled motion of the model into an equivalent motion of the viewer relative to the model. The perceived result for the local viewer is exactly the same.

The Cubic Mouse also controls the three orthogonal seismic sections within the model. These slices effectively belong to the model and they are shared resources. The control of the slices is an example of an event driven interaction mechanism in Avango. Moving one of the rods triggers a SCHEME script, which moves the slice. For the stand-alone implementation the resolution of a rod was generally sufficient to avoid clutching. Moving a rod from stop to stop would move the corresponding slice from one end to the other end of the data set. The position of the rods are used as absolute inputs. With the distributed version we cannot expect the rods of different Cubic Mouse devices to be in exactly the same position, which would cause the slices to jump when control is passed from one site to another. To avoid these problems altogether we decided to move to a relative control mode for the rods at the expense of occasional clutching. Moving a rod moves the corresponding slice relative to its previous position. The buttons at the ends of each rod serve as a clutch. Pressing the button and pulling or pushing the rod does not affect the corresponding slice. The relative approach has also the advantage that it does not require any synchronization mechanism like distributed locks.

Another case of a local resource is the input from each rod of the Cubic Mouse affecting the state of a shared object in the scene graph. This means that client sites which have only received a copy of the original scene graph from

the server need to get access to the node in the scene graph that is affected by the rod. These hooks into the scene graph are conveniently provided by our shared symbol table. Since the shared symbol table adds these hooks into the local name space of each client site, the SCHEME scripts for controlling the interaction on the server and the client site as well as for the non-distributed case are exactly the same.

The level-of-detail tool interacts with a SCHEME mediator allowing users to toggle the displayed level of detail between low and high resolution for horizon and fault surfaces. The SCHEME script referenced by the mediator is executed when the user points to a surface and clicks the button on the tracked wand. The script removes the low resolution version of the surface from the scene graph and adds the high resolution version and vice versa depending on the current state. The stand-alone implementation used a global SCHEME map to store references to the low and high resolution versions of the surfaces for the script to be able to access this necessary evaluation context information. Since we currently do not provide a mechanism to distribute "pure" SCHEME objects like this map, we had to resort to a different solution. We could have implemented this map as an Avango node, which would have been distributed with the scene graph. Clients would then use the shared name space to get access to the map. We decided for another option, which bundles the SCHEME script with the required data as a sort of SCHEME object in the sense of object-orientation. For this purpose we extended the script mediator to maintain references to the low and high resolution Avango nodes of the surfaces. These references are distributed with the mediators making them available for SCHEME scripts evaluated at the client sites.

These are two representative examples which show how we extended script based interaction to the distributed case. Similar approaches were used to make the other tools available for the distributed case.

Conclusion

We have presented tools and techniques to support scripting interfaces in distributed virtual environments. A shared name space allows convenient access for scripts to shared objects. These concepts have been successfully applied to migrate a geo-scientific application to support distributed exploration of large data sets. We found that our scripts from the stand-alone case generally require minimal changes to work for the distributed case.

For our development we used an extension to SCHEME, which provides a framework for quasi object-oriented scripting. This extension greatly helped us to structure our distributed application. However, inherently object-oriented scripting languages, like PYTHON, might be a better alternative to the SCHEME extension, which only mimics object-orientation.

We have tested our geo-scientific application only in environments with 10 MBit to 100 MBit Ethernet connections and low latency networks. As a next step we plan to install the distributed prototype at two offices of an oil company, which are located in Europe and the US. This will give us further insight into how usability is affected by networks with lower bandwidth and higher latency.

From our experience it would have often been convenient to be able to distribute "pure" Scheme objects directly without explicitly going through the Avango shared object distribution mechanism. Concepts from parallel programming languages like LINDA could be a solution for this problem. These concepts need to be evaluated with respect to their compatibility with the interactive real-time requirements of virtual reality applications.

Acknowledgments

This work was partially supported by the VRGeo consortium. We would like to thank the members of the consortium for their valuable feedback during consortium meetings. We also thank the VE group at GMD for their support.

References

[1] C. Carlsson and O. Hagsand, "DIVE: A Multi User Virtual Reality System," in *Proceedings of IEEE Virtual Reality Annual International Symposium (VRAIS '93)*, pp. 394–400, IEEE, Sept. 18–22 1993.

[2] R. Blach, J. Landauer, A. Rösch, and A. Simon, "A Highly Flexible Virtual Reality System," *Future*

Generation Computer Systems, Special Issue on Virtual Environments, vol. 14, no. 3–4, pp. 167–178, 1998.

[3] H. Tramberend, "Avocado: A Distributed Virtual Reality Framework," in *Proceedings IEEE Virtual Reality '99 Conference* (L. Rosenblum, P. Astheimer, and D. Teichmann, eds.), pp. 14–21, IEEE Computer Society, Mar. 13–17 1999.

[4] W. Clinger and J. Rees, "Revised^4 Report on the Algorithmic Language Scheme," Technical Memo AIM-848b, Massachusetts Institute of Technology, Artificial Intelligence Laboratory, Nov. 2 1991. originally published in *SIGPLAN Notices* 21 (12) December 1986.

[5] B. Fröhlich, S. Barass, B. Zehner, and M. Göbel, "Exploring Geo-Scientific Data in Virtual Environments," in *IEEE Visualization '99*, pp. 169–174, Oct. 1999.

[6] B. Fröhlich and J. Plate, "The Cubic Mouse: A New Device for Three-Dimensional Input," in *Proceedings of CHI '2000*, ACM, Apr. 1–6 2000.

[7] J. Rohlf and J. Helman, "IRIS Performer: A High Performance Multiprocessing Toolkit for 3D Graphics," in *Proceedings of SIGGRAPH '94* (A. Glassner, ed.), pp. 381–395, ACM, July 24–29 1994.

[8] P. S. Strauss, "IRIS Inventor: A 3D Graphics Toolkit," in *Proceedings of the 8th Annual Conference on Object-Oriented Programming Systems, Languages and Applications* (A. Paepcke, ed.), pp. 192–200, ACM, Sept. 26–Oct. 1 1993.

[9] B. Stroustrup, *The C++ Programming Language*. Reading, MA, USA: Addison-Wesley, 3rd ed., 1998.

[10] H. Tramberend, F. Hasenbrink, and B. Fröhlich, "Tools, Mediators, and Interaction Operators: A Concept for Interaction in Virtual Environments," in *Proceedings of 3. International Immersive Projection Technology Workshop*, pp. 77–79, Center of the Fraunhofer Society Stuttgart IZS, May 10–11 1999.