

Tutorial for VR-OpenMASK, a Software Development Platform for Virtual Reality

Alain Chauffaut, Thierry Duval, Christian Le Tenier, Michaël Rouillé

IRISA SIAMES Project

Campus de Beaulieu, 35 042 Rennes cedex, France

Phone : (+33) 2 99 84 71 00 / Fax : (+33) 2 99 84 71 71

E-mail : {Alain.Chauffaut, Thierry.Duval, Christian.Le_Tenier, Michael.Rouille}@irisa.fr

1	Introduction.....	5
2	VR-OpenMASK from a project manager point of view.....	5
2.1	OpenMASK: multi-frequencies simulations.....	5
2.2	OpenMASK: event-driven simulations.....	6
2.3	OpenMASK: distribution of simulations upon a network	7
2.4	VR-OpenMASK: OpenMASK + particular features dedicated to Virtual Reality.....	7
2.5	VR-OpenMASK: collaborative VR Applications.....	8
3	VR-OpenMASK from an end-user point of view.....	8
3.1	Execution kernel, simulated objects and binary applications.....	8
3.1.1	How to install VR-OpenMASK and its associated libraries	8
3.1.2	Application overview through its main file and its associated Makefile	9
3.1.2.1	Simulation tree construction.....	9
3.1.2.2	Symbolic names for simulated object creation	9
3.1.2.3	Creation and running of the controller	9
3.1.3	Parameterization of the application with a configuration file.....	9
3.2	Interaction with the 3D Visualizer.....	10
3.2.1	Multi-views possibilities	10
3.2.2	Basic interaction capacities of the visualizer.....	11
3.2.3	Cameraman service object	12
3.2.4	Providing decors	12
3.3	Creation of a new VR-OpenMASK application.....	13
3.3.1	Using existing modules: the simulated objects types.....	13
3.3.2	Creation of the application by using a configuration file	14
3.3.3	Creation of the application by specializing the main for a unique use	15
3.3.4	Instantiated simulated objects and their data stream connections	15
3.3.5	Case of a distributed simulation: referentials and mirrors.....	16
3.3.5.1	The main file for distributed simulation.....	17
3.3.5.2	Definition of some environment variables	17
3.3.5.3	Modification of the Makefile	17
3.3.5.4	Example of a configuration file for a distributed simulation.....	17
3.3.5.5	Declaration of the directories used for a distributed simulation	18
3.3.5.6	How to launch the distributed simulation	18
4	VR-OpenMASK from a programmer point of view.....	18
4.1	Main characteristics of a simulated object.....	18
4.2	How to create new simulated objects types	19
4.2.1	The PsSimulatedObject class.....	19
4.2.2	OpenMASK PsType interpolation distribution	19
4.2.3	DSO Libraries.....	19
4.3	The Interactive 3D visualizer and its partners	19
4.3.1	Interactive 3D Visualizer principles	19
4.3.1.1	Animation principles	20
4.3.1.1.1	Animation partners	20
4.3.1.1.2	Input handlers	20
4.3.1.2	Visualization Principle	20
4.3.1.2.1	Visualization control events.....	20
4.3.1.3	Interaction principles	20
4.3.2	Predefined input handlers	21
4.3.3	Predefined partners	21
4.4	Example of a new visualizable simulated object type creation	22
4.4.1	Writing the C++ classes	22
4.4.1.1	The ShiftedForTutorial.h file	22
4.4.1.2	The ShiftedForTutorial.cxx file.....	23
4.4.2	Writing the Makefile and generating the dynamic library.....	25
4.4.3	Modifying the main.cxx file, its associated Makefile, and a configuration file.....	26
4.4.3.1	C++ code to add in the main.cxx file.....	26
4.4.3.2	C++ code to add in the Makefile file.....	26
4.4.3.3	Code to add in a configuration file.....	26
5	VR-OpenMASK from a contributor point of view: a new interaction set.....	26
5.1	The adapters: they allow to make interactive an existing simulated object	27

5.1.1	Rules which must be respected for the objects that must be made interactive	27
5.1.2	Functioning principle of an adapter.....	27
5.1.3	Examples of existing adapters	27
5.1.3.1	The ProtocolTeacher class	27
5.1.3.2	The DoubleProtocolTeacher class.....	28
5.1.3.3	The ConnectorProvider class	28
5.1.3.4	The DoubleConnectorProvider class	28
5.1.3.5	Some connectors	28
5.1.3.5.1	The SimpleConnector class.....	28
5.1.3.5.2	The DoubleConnector class	28
5.1.3.5.3	The ConstraintDefaultAdaptorOffset class	28
5.1.4	How to use an adapter.....	29
5.1.4.1	Declaration in the main file.....	29
5.1.4.2	Declaration in the configuration file	29
5.2	The interactors: they allow to learn the interaction protocol to an interaction tool	30
5.2.1	Functioning principle of an interactor.....	30
5.2.2	Rules which must be respected for an interaction tool to use with an interactor	31
5.2.2.1	Providing a doPick method.....	31
5.2.2.2	Providing an interactorConnection method.....	31
5.2.3	Examples of interactors and interaction tools	31
5.2.3.1	The NInteractor class.....	31
5.2.3.2	The DoubleInteractor class	31
5.2.3.3	Some interaction tools	31
5.2.3.3.1	The VirtualRay class.....	31
5.2.3.3.2	The VirtualHand class.....	32
5.2.3.3.3	The Virtual3DCursor class	32
5.2.4	How to use an interactor	32
5.2.4.1	Declaration in the main file.....	32
5.2.4.2	Declaration in the configuration file	32
5.3	How to create new adapters	33
5.3.1	The DoubleMultipleProtocolTeacher class	33
5.3.2	The DoubleMultipleConnectorProvider class	33
5.3.3	The DoubleMultipleConnector class	33
5.4	How to provide different behaviors to a virtual tool.....	33
5.5	How to create new interactors	34
6	Conclusion.....	34

Abstract

This tutorial will look at the VR-OpenMASK platform from different point of views: the one of a project manager who must choose a VR development software, the one of the VR-OpenMASK user, and the one of a VR-OpenMASK developer. At the end we will explain how we suggest programming the interactions of a user within our virtual universes, from a VR-OpenMASK contributor point of view.

To present VR-OpenMASK from a project manager point of view, allowing him to choose a software development platform for virtual reality, we offer a sight upon the different categories of software it is possible to realize with VR-OpenMASK. It is realized with some demonstrations that illustrate the main VR-OpenMASK concepts, showing that it is possible to visualize 3D virtual worlds, to interact within these universes, and to share universes and interactions between several users located on different workstations.

Then, from a VR-OpenMASK user point of view, we show how to install the software with the provided libraries and binaries to be able to make the demonstrations that have been presented. In this part we show also how to create new applications based upon existing simulated objects, and how we can interact with the 3D VR-OpenMASK visualizer.

We go on with the point of view of a simulated object programmer: we show here the main characteristics of the simulated objects we create for Virtual Reality applications. We insist upon the characteristics dedicated to 3D visualization and animation of the simulated objects that have to be particular partners of the 3D visualizer.

Last we talk about what we think is essential to VR-OpenMASK: the interactions with the simulated objects of our virtual worlds. We explain two essential notions: the adapters that can make interactive some existing simulated objects, and the interactors that allow to control with virtual tools the interactive objects. We show that it is important to install a communication protocol between the interactive objects and the interaction tools in order to optimize the generic parts of the interactions. Then we explain how to create new adapters and new interactors from a VR-OpenMASK contributor point of view, in order to provide generic tools to the simulated objects programmers and to the applications creators.

Key words: Virtual Reality, 3D Interactions, Software Development Platform, 3D Collaborative Work.

1 Introduction

OpenMASK [11], the successor of GASP [4], is a distributed simulation system designed by IRISA's SIAMES project. It is a distributed software platform for animation and simulation, implemented as an object oriented development environment allowing real time simulation and visualization of complex 3D systems.

An OpenMASK virtual world is constituted of a set of entities. Each entity in the system is composed of one or more simulated objects. These simulated objects are composed of a set of named outputs, inputs and control parameters that constitute the public interface of the entity and of a compute method that is in charge of their activation. Entities communicate by synchronous exchanges (data stream connections) or in an asynchronous way using events and messages. All these entities are scheduled at their own frequency by a particular object we call a controller.

OpenMASK allows also to distribute these entities over the network, either to increase the computational power, or to benefit from physical devices drivers working only onto dedicated machines. As OpenMASK was first designed for physical simulation, the distribution of the simulation relies upon a strong synchronization algorithm.

OpenMASK can benefit from a set of tools dedicated to 3D interactive graphics: a 3D visualizer and some facilities dedicated to 3D interaction, and then becomes a system dedicated to Virtual Reality we call VR-OpenMASK. As VR-OpenMASK is an extension of OpenMASK, it benefits from its distribution facilities and can be used to share 3D virtual universes between several users like DIVE [2], NPSNET [10] or MASSIVE [8]. On the contrary of what can be done in environments such like "Avatars" [1], where many users can share the same virtual world, but within which they are not strongly synchronized so that they can not share real-time simultaneous interactions upon the same object; the distribution paradigm implemented in OpenMASK enables to share Virtual Universes between only a small number of users, but these users will be able to share precise interactions upon highly technical objects, thanks to our strong synchronization algorithm.

In this tutorial we are going to show the VR-OpenMASK main features from different point of views: the one of a project manager who must choose a VR development software, the one of the VR-OpenMASK user, and the one of a VR-OpenMASK developer. At the end we will explain how we suggest programming the interactions of a user within our virtual universes, from a VR-OpenMASK contributor point of view.

2 VR-OpenMASK from a project manager point of view

The idea here is to quickly show the main functionalities of the VR-OpenMASK platform, and to illustrate them with a set of examples and demonstrations. Each example will introduce one or two new features of VR-OpenMASK, in order to offer a wide view of what it is possible to realize with this virtual reality development platform.

2.1 OpenMASK: multi-frequencies simulations

Each entity in the system is composed of one or more simulated objects. These simulated objects, which are the basic components of OpenMASK, are composed of a set of named outputs, inputs and control parameters. They constitute the public interface of the entity, which is in charge of their activation. This activation happens at the frequency associated with each object or family of objects, and it is also one of the main differences between OpenMASK and other simulation environments. For example a mechanical model of a car will need a 50-Hertz frequency to obtain a realistic mechanical simulation, while a car driver will need only 5-Hertz frequency to simulate human behavior. A particular object is in charge of the scheduling of the modules, we call it a Controller.

At each simulation step, the inputs of the object will be read and given to a calculation function, which is responsible for handling these input values, and then updating the values of the control parameters by using computed results. Then, we can use these control parameters values to calculate new values for the outputs. The inputs can be connected to the outputs of other objects at different stages of the simulation by either naming the connected objects or asking the controller for an appropriate object. Moreover, the kernel is in charge of ensuring that a value provided to an object is consistent with the value from the original output.

For example, consider the following objects, that will be used in section 3.3, where the Tracker object uses its PositionReference input, connected to the Position output of the Trajectory object, in order to calculate its new Position and Orientation control parameters, before updating its Position and Orientation outputs thanks to these new control parameters values. So these two objects are connected by a data stream (figure 1) which is regulated

by the controller.

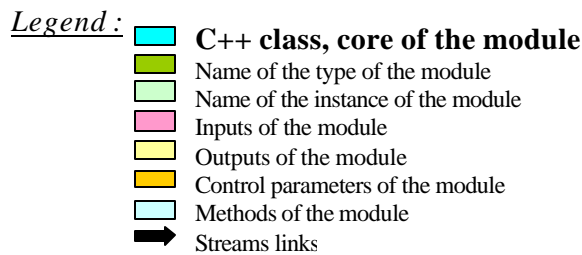
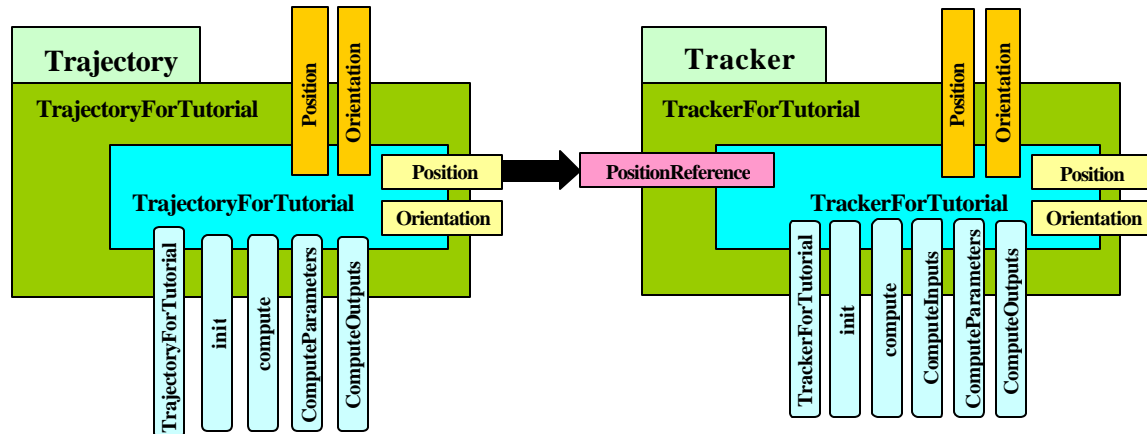


Figure 1. Data stream connection between two modules

2.2 OpenMASK: event-driven simulations

Objects are also able to communicate with each other by sending signals and events; an internal function called *processEvent()* is in charge of dealing with these dialogues and requests. During each simulation step, this method will be called as many times as there are events and messages to process.

Simulation objects can perceive broadcasted signals, if they have subscribed to them. This subscription can be made to a particular signal: *registerForSignal* (theSignal); or to a particular signal from a particular entity: *registerForSignalBy* (theSignal, theSender). These signals are sent thanks to the two methods: *fireSignal* (theSignal); and *fireValuedSignal* (theSignal, theAssociatedValue);

Simulation objects can also receive messages explicitly sent to them. These messages are sent thanks to the two methods: *sendEvent* (theReceiver, theMessage); and *sendValuedEvent* (theReceiver, theMessage, theAssociatedValue).

In both cases, a communication protocol must be set between senders and receivers in order to allow the receivers to manage properly the values possibly assigned to the signals and messages.

For example, the corresponding *processEvent* method of the *TrajectoryForTutorial* class is:

```
bool TrajectoryForTutorial::processEvent (PsEvent * event) {
    bool traite = false ;
    if (event->eventId == "addTarget") {
        traite = true ;
        addTarget () ;
    } else if (event->eventId == "removeLastTarget") {
        traite = true ;
        removeLastTarget () ;
    } else if (event->eventId == "removeTarget") {
        PsValuedEvent<PsName> * valuedEvent =
            dynamic_cast<PsValuedEvent<PsName> > * (event) ;
        if (valuedEvent != NULL) {
            traite = true;
            removeTarget (valuedEvent->value) ;
        }
    }
}
```

```
    return (traite) ;  
}
```

This method is activated by the controller each time an event (signal or message) is received by a simulation object. When you want the events to be processed as soon as possible, you can program this in the processEventASAP method.

When receiving several events between two simulation steps, the events will be processed according to their arrival: the first event to arrive will be processed first. Sometimes it can be a problem, if several objects send messages to the same object at the “same” time and if this object needs to be aware of all these messages before being able to respond correctly to all of them. In that case, the solution is to use the processEvent method only to store all the events into appropriated data structures, and to process them in the compute method in the next simulation step, rather than processing them directly in the processEvent method

2.3 OpenMASK: distribution of simulations upon a network

Each simulated object is assigned to a process and processes are assigned to workstations. Each process owns a particular simulated object: a controller, which schedules all the local simulated objects. Within each process, there will be none, one or several “real entities”, called “referentials”; and none, one or several “ghost” entities, called “mirrors”. The presence of mirrors in a process is function of the inputs needed by the referentials of this process: if there is a referential B that needs for input the output of a referential A located within another process, then there will be a mirror of A within the process where B is located. Mirrors are linked to their referentials with a data-stream connection: at each step of the simulation, a referential sends up to date values to all of its mirrors if these values changed during the simulation step. Thanks to this mechanism, each workstation can be considered to be in parallel updating the values of its mirrors and calculating a simulation step, because the simulation steps of each referential is calculated without having to wait for the network for a new value. Of course, there is a threshold to ensure synchronization: a simulation step is not calculated if the current process is faster than the others.

As OpenMASK allows entities to be distributed upon several workstations, typically to increase the power with compute upon several machines: it enables the construction and the execution of complex virtual worlds, with a lot of entities with complex and heavy weight calculation behavior.

It can also be used to use dedicated drivers upon different machines, when a computer does not own enough hardware communication ports, or when a driver is only available for a particular type of machine.

2.4 VR-OpenMASK: OpenMASK + particular features dedicated to Virtual Reality

We provide a particular simulated object allowing visualizing a virtual world: the Ps3DVisualization.

OpenMASK provides a 3D visualizer based on the OpenGL Performer library, which is a particular simulated object and is in charge of the graphical representation of all the objects associated to geometries in the simulation. In fact, during the simulation, it looks for all simulated objects that have a 3D representation, and visualizes them; moreover, it creates the inputs to connect to the visualizable outputs of these objects and therefore updates the position and orientation values according to the outputs of the objects.

The result is that we can interact with objects (pick, move, modify, and so on.) in a more intuitive way, and also make animations possible. Even more, as our visualizer is a simulated object, there can be as much visualizers as needed within the same simulation, and when distributed upon the network on several workstations, people can see the same graphical representation of the simulation, then are allowed to cooperate in a shared 3D virtual world. Also, the users can choose interactively the viewpoint of the visualizer, so one or several visualizers can be placed on the same workstation in order to allow the end-user to see the same world with different viewpoints at the same time.

We provide also data types allowing controlling animation points, associated to visualization partners.

We offer some interactors, which can encapsulate physical 3D input devices, in order to capture the inputs from the end-users, and which are able to control interactive objects.

Then we offer tools allowing making interactive the objects of our virtual universes, so that they can be manipulated by our interactors.

Last, we offer some tools to ease 3D navigation within 3D virtual worlds.

2.5 VR-OpenMASK: collaborative VR Applications

Adding visualization, navigation and interaction to the possibility of distributing a simulation enable to distribute several interactive visualizations upon different workstations upon a network, thus allowing to share 3D virtual universes between several users with VR-OpenMASK.

3 VR-OpenMASK from an end-user point of view

In this part we focus on the user point of view whose aim is to reuse and eventually parameterize existing applications.

To do that, you have to download OpenMASK and related applications from our web site (<http://www.openmask.org/>). Then you need to inspect your application binary code (mainly through the main file source) and build your configuration file. Certainly, you have to learn how to interact with our 3D interactive viewer.

We will finish this part describing the different kinds of VR -OpenMASK applications.

3.1 Execution kernel, simulated objects and binary applications

3.1.1 How to install VR-OpenMASK and its associated libraries

OpenMASK is a free and open distribution under a QPL license. To download it, you just have to register, to choose the good distribution matching your gcc and Performer versions, and to read install notes that we reproduce here. The kernel is provided with an interactive 3D visualizer simulated object.

Kernel and visualizer depend on other products:

- optional dependencies for the kernel
 - PCCTS, if the grammar of object descriptors was to be changed. We use the 1.33MR31 version.
 - Doxygen, to extract documentation. We use the 1.2.16 version.
 - PVM, to use the distributed versions of the OpenMASK kernel. We use the 3.4.2 version.
- mandatory dependencies for OpenMASK -3Dvis:
 - OpenGL Performer. Works with OpenGL Performer 2.4 or higher. This is the most difficult package to install on Linux. Note that Performer Linux is not compatible with gcc 2.96, you have to downgrade to gcc 2.95.3 (here is an explanation) or use gcc 3.x. If you use gcc 3.x you must use the corresponding version of the Performer library.
- optional dependencies for OpenMASK -3Dvis:
 - OpenInventor (required for the iv loader): currently there are no RPMs available for gcc3.x, so if you use the gcc3.x version of OpenGL Performer you must download OpenInventor sources, edit make/ivcommondefs, replace g++ by g++3 and gcc by gcc3 and then gmake install.
 - PCCTS, and Doxygen, as for the kernel.

If a complete recompilation of OpenMASK3.0 and its different examples and contributed objects is needed, here are a few tips. The Makefiles provided with the sources rely on a few environment variables to work. You can either set them in your environment or change the makefile.

- OpenMASKDIR should point to the root of the OpenMASK installation.
- PVM_ROOT should point to the root of the PVM installation, if the distributed version has to be compiled.
- PCCTSDIR and PCCTSBINDIR should point respectively to the root of the PCCTS installation and to the binaries (antlr and dlg) of PCCTS.
- LD_LIBRARY_PATH is initialized by the OpenMASK installation and you should append to it your own library directories.
- QTDIR should point to the root of the QT install if you want to compile the contribution and example using a QT control panel.
- If you compile with gcc3.x, edit the make.in in the kernel directory and replace g++ with g++3. Or you can set the environment variable COMPILER to g++3.
- If you compile under Irix, use gmake instead of make.

3.1.2 Application overview through its main file and its associated Makefile

Looking at an application's main file such as the ones presented in section 3.3.2 and 3.3.3 will give you informations about different kind of simulated objects you can use with this application.

Mainly, a program is split into three parts:

- in the first one, you built a simulation tree;
- in the second one, you create the controller and teach it how to create new simulated objects;
- in the last one, you init and run the simulation controller.

The controller is a specialized simulated object in charge of managing all the other objects, their communications and reactions to events, and the objects' distribution.

3.1.2.1 Simulation tree construction

You have to create a good simulation tree, containing the initial objects of your virtual universe. This simulation tree can be built directly in your main file or be described in a simulation file. For the first scenario, we provide a simulated object descriptor class and configuration parameters descriptor classes. For the second one, we provide a dedicated file loader. This second way is more flexible and an operator can easily build many different sessions.

3.1.2.2 Symbolic names for simulated object creation

Defining an application class is mainly declaring what kind of simulated objects could be involved into following sessions. We provide an easy way to do that through an instance creator public method allowing the matching of a string with a C++ constructor. This could also be done recursively through added declarations included into genitor objects previously declared. The string is called the class symbolic name of the simulated object class. As the method is public, you can do this declaration into the main file but also in specialized controller dedicated to your application class.

3.1.2.3 Creation and running of the controller

After that, to run a new application session, you just have to create a controller, to provide to this controller the simulation tree, and to tell it that it has to run.

Looking at a Makefile such as the one presented in section 3.3.1 tells you the list of dynamic shared libraries used for the binary code of the application detailed in section 3.

3.1.3 Parameterization of the application with a configuration file

With a configuration file, you enumerate all the simulated objects created at the beginning of the session.

File syntax is based on basic pair name-value recognizing. A value could be composed of a list of pairs {...} or an array of values [...].

A basic object is described giving:

- Its symbolic name: cameraman { }
- Its symbolic class name: Class PsCameraman
- Parameters concerning OpenMASK control: Scheduling { }
 - Compute frequency
 - Process host
- Parameters concerning the object configuration: UserParams { }
 - theses ones depend on each object classes.

For each genitor objects, you have to give its sons: Sons { }

Controller is the main example of genitor object. It is always the root the simulation tree. This controller could be the basic one or a derived one.

Through the configuration file, the operator can decide to distribute the session. In this case, the controller needs

specific scheduling parameters:

- Latency: it is the acceptable delay during with a simulated object can run without getting fresh datas from its remote partners. During this time, he controller provides it with interpolated and extrapolated values and after that time, it does a hard synchronization.
- List of matching between processes and machines: Machines { }. Then you associate a process name to an actual machine value. Different processes can be on the same machine.

At this time, distribution is built over le Parallel Virtual Machine library (PVM). For distributed session, an operator has to manage data files and binary files accesses, in a way that all the required files are at the right places.

3.2 Interaction with the 3D Visualizer

One of the main features of virtual reality applications is the visualization of a 3D scene, the interaction of the user with the objects of the scene, and finally the feeling for the user to be immersed inside the scene. We provide basic classes to do that, it is the VR-OpenMASK set. An operator of a VR-OpenMASK application can tune visualizer and interactors to his needs, through the configuration file.

3.2.1 Multi-views possibilities

Depending of the 3D display devices he uses, the operator can choose using one or more graphic pipes, displaying one or more windows, eventually splitting one window into several viewports, and so on ...

```
//-----
// OpenMASK-3DVis Configuration Parameters
// 1Pipe, 1 Window, 4 Viewports
//-----
include
"${OpenMASKDIR}/contrib/3DVis/defaultConfigurationParameters.3DVis.OpenMASK3"
Use default3DVis
pipes {
  pipe0 {
    Use defaultPipe
    windows {
      window0 {
        Use defaultWindow
        viewports {
          viewport0 {
            Use defaultViewport
            viewportSize [0.0 0.5 0.0 0.5]
          }
          viewport1 {
            Use defaultViewport
            viewportSize [0.5 1.0 0.0 0.5]
          }
          viewport2 {
            Use defaultViewport
            viewportSize [0.5 1.0 0.5 1.0]
          }
          viewport3 {
            Use defaultViewport
            viewportSize [0.0 0.5 0.5 1.0]
          }
        }
      }
    }
  }
}
```

All of the classical viewing parameters can be managed.

```
Define defaultViewport Group {
  clipping [0.1 1000]
  fov [45 -1]
  associatedObservable "cameraman"
  cameraNumber 0
  viewportSize [0 1 0 1]
```

```

viewportOffsets [0 0 0 0 0 0]
cullFace BACK
wireframeMode false
displayStatistics false
levelOfStatistics 1
saveImages false
saveImagesRate 1
earth false
sky false
}

```

Notice that in these examples, we use services like “include”, “Define” and “Use”, which minimize configuration files writing.

In the same way, the operator can define a view pyramid (or camera) and he chooses its initial position and orientation through camera support concept. A camera support is defined by a named transform node in the scene graph and this node belongs to a simulated object. An object may own several camera support. They are numbered from zero to any. In the configuration file, the operator can associate a camera support to a view by giving an object name and camera number. For that view, the cameras take place on this support at the beginning of the simulation.

```

window1 {
  windowName "OpenMASK3.0 / Truck tracking "
  origin [450 350]
  size [400 300]
  viewports {
    viewport0 {
      associatedObservable truck1
      cameraNumber 2
    }
  }#viewports
}

```

3.2.2 Basic interaction capacities of the visualizer

The first basic interaction with the visualizer is to play with the different camera supports existing in the scene. For that, the operator has to include in the configuration file, an instance of our keyboard input handler class. Then, it can switch his camera from one support to another and control the camera’s position and orientation relative to the current support. This keyboard navigator was designed to work with one instance of 3DVis, and also for manipulation of casual cameraman described in the next paragraph. This object also provides a lot of other services to the operator, like stopping the simulation, displaying graphic statistics, recording video sequences, displaying a sky or a ground, and so on ...

Default keys are equivalent to specifying the following user parameters:

```

UserParams {
  keys {
    Escape quit
    comma left
    period right
    Up forward
    KP_Up forward
    Down backward
    KP_Down backward
    Page_Up up
    Page_Down down
    Left lookLeft
    KP_Left lookLeft
    Right lookRight
    KP_Right lookRight
    Home lookUp
    End lookDown
    Delete rollClockwise
    Insert rollAntiClockwise
    s toggleStats
    S toggleStats
    z toggleSaveImages
    Z toggleSaveImages
    c toggleCullFace
    C toggleCullFace
  }
}

```

```

    e toggleEarth
    E toggleEarth
    F2 toggleCoherentViewports
    F3 toggleCoherentWindows
    F1 toggleCoherentPipes
    q incrementFieldOfView
    Q incrementFieldOfView
    w decrementFieldOfView
    W decrementFieldOfView
    cameraKeys [1 2 3 4 5 6 7 8 9 0]
    KP_Add nextCameraman
    KP_Subtract previousCameraman
    plus nextCasualCameraman
    minus previousCasualCameraman
    KP_0 resetCameramanOffset
    v viewCurrentValues
    V viewCurrentValues
  }
}

```

3.2.3 Cameraman service object

Based on these kinds of manipulations, the cameraman class is a simulated object class dedicated to a camera support management.

One way to control a cameraman is to associate it to a keyboard navigator.

Concerning the cameraman initialization, its original position can be described either as position (x y z) and orientation (h p r) coordinates in the coordinate system to which the animated node of the geometry associated to the camera is attached (generally the global coordinate system):

```

UserParams {
  yUp true
  position [0 10 70]
  orientation [0 0 0]
}

```

or in a lookAt and lookFrom point and an up vector, expressed in the same coordinated system.

```

UserParams {
  yUp true
  lookFrom [0 17 70]
  lookAt [0 17 0]
}

```

Last, it is mandatory to specify a geometry file to the camera and to specify the node of that file that will be animated to move the camera:

```

UserParams {
  geometryFileName ./ivObjects/legoman.iv
  moveableNodeName DCS_positionOrientation
  yUp true
  position [0 10 70]
  servoLookAt [tracker1 position]
}

```

3.2.4 Providing decors

To put easily static decors in a scene, we provide a simple simulated object class which permits to position and orientate a geometry file.

```

decor {
  class PsvDecor
  UserParams {
    geometryFileName ../../data/iris-tux.pfb
    position [0 0 0]
    orientation [0 0 0]
  }
}

```

Notice this object has no frequency because it does not need any compute as it is static.

3.3 Creation of a new VR-OpenMASK application

3.3.1 Using existing modules: the simulated objects types

We suppose that we want to use and visualize some OpenMASK modules that have been defined in the TutorialDir directory, which are available in the libTutorial.so shared directory:

- A TrajectoryForTutorial class: a module that calculates 3D positions and that provides them in its "position" output;
- A TrackerForTutorial class: a module that tracks a target by aiming at it and then making a step within this direction (covering only a part of the distance between itself and its target).

In order to be able to use them in a new application, we have to create a new main.cxx file, which must be able to instantiate these objects, as well as the 3D visualizer provided in the OpenMASK contribs.

This main.cxx file will look like this:

```
#include <PsException.h>
#include <PsController.h>
#include <SimpleOpenMASK3Loader.h>
#include <Psv3DVis.h>
#include <TrajectoryForTutorial.h>
#include <TrackerForTutorial.h>

int main (int argc, char * argv []) {
    Psv3DVis::printPfExitWarning = FALSE ;
    PsObjectDescriptor * simulationTree =
        new PsObjectDescriptor ("root", "trajectory", NULL, NULL) ;
    if (argc >= 2) {
        simulationTree =
            (new SimpleOpenMASK3Loader (argv [1]))->getRootObjectDescriptor () ;
    }
    PsController * controler = new PsController (*simulationTree, 0) ;
    controler->addInstanceCreator ("3DVis",
        new PsSimpleSimulatedObjectCreator <Psv3DVis> ()) ;
    controler->addInstanceCreator ("TrackerForTutorial",
        new PsSimpleSimulatedObjectCreator<TrackerForTutorial> ()) ;
    controler->addInstanceCreator ("TrajectoryForTutorial",
        new PsSimpleSimulatedObjectCreator<TrajectoryForTutorial> ()) ;
    try {
        controler->init () ;
        controler->run () ;
    } catch (PsException & e) {
        cerr << "Unresolved exception " << e << endl ;
    }
    delete controler ;
    pfExit () ;
}
```

The associated Makefile must be able to link with the main kernel library and these provided libraries:

```
ifdef OpenMASKConfigFile
    include $(OpenMASKConfigFile)
else
    include $(OpenMASKDIR)/kernel/make.in
endif

ifdef PFROOT
    PERFORMERINCLUDES = -I$(PFROOT)/usr/include/Performer \
        -I$(PFROOT)/usr/include/Performer/pfdb \
        -I$(PFROOT)/usr/include/Performer/pfui \
        -I$(PFROOT)/usr/include/Performer/pfutil \
        -I$(PFROOT)/usr/include/Performer/pf \
        -I$(PFROOT)/usr/include/Performer/pr
endif

COMPILE = $(COMPILER) $(COMPILEFLAGS) $(OPENMASKFLAGS) $(PERFORMERINCLUDES) \
    -g -O2 -ftemplate-depth-50

LINK = $(LINKER) -L/usr/lib/libpfdb -Wl,-noinhibit-exec
```

```

INCDIR =-I . \
-I$(TutorialDIR)/modules/Tutorial/inc \
-I$(OpenMASKDIR)/kernel/inc \
-I$(OpenMASKDIR)/contrib/3DVis/inc \
-I$(OpenMASKDIR)/contrib/3DVisPartners/inc \
-I$(OpenMASKDIR)/contrib/userTypes/math/inc \
-I$(OpenMASKDIR)/contrib/loaders/simpleOpenMASK3Loader/inc \
-I$(PCCTSDIR)/h

CXX = mainTutorial.cxx

%.o : %.cxx
    $(COMPILE) $(COMPILEFLAGS) $(INCDIR) -c $< -o $@

SO = $(TutorialDIR)/lib/libTutorial.so \
    $(OpenMASKDIR)/lib/lib3DVisPartners.so \
    $(OpenMASKDIR)/lib/lib3DVis.so \
    $(OpenMASKDIR)/lib/libUserTypes.so \
    $(OpenMASKDIR)/lib/libSimpleOpenMASK3Loader.so \
    $(OpenMASKDIR)/lib/libOpenMASK.so

DSO = -L$(OpenMASKDIR)/lib -L$(TutorialDIR)/lib -L/usr/lib/libpfb \
    -lTutorial -l3DVisPartners -l3DVis -l3DVisInputHandlers -lUserTypes \
    -lSimpleOpenMASK3Loader -lOpenMASK -lpf -lpfdu -lpfutil

TARGETNAME = tutorial

$(TARGETNAME): mainTutorial.o $(SO)
    $(LINK) mainTutorial.o $(DSO) -o $@

clean:
    rm -rf ii_files/ mainTutorial.o $(TARGETNAME)

```

3.3.2 Creation of the application by using a configuration file

Using the binary code defined in the previous paragraph, we can instantiate the appropriate simulated objects of our universe thanks to a configuration file.

We have to provide correct datas to the objects of the universe to initialize them properly. A trajectory will be parameterized by a list of points and the progression between the points in a simulation step, a tracker will need the names of its target and of the output of this target, and a 3D visualizer will need the name of its own configuration file. So, it is possible to describe the application to execute thanks to a configuration file, that will look that way:

```

#OpenMASK3
root {
    Class Controller
    Sons {
        Trajectory {
            Class TrajectoryForTutorial
            Scheduling {
                Frequency 75
            }
            UserParams {
                step 0.05
                trajectory [
                    [-50 100 -1]
                    [50 100 20]
                ]
            }
        }
    }
    Tracker {
        Class TrackerForTutorial
        Scheduling {
            Frequency 75
        }
        UserParams {
            geometryFileName ../Data/coneCyan.pfb
            target [Trajectory position]
        }
    }
}

```

```

    }
  }
  vis {
    Class 3DVisualizer
    Scheduling {
      Frequency 75
    }
    UserParams {
      include "./ParamsVisu/1Window_1Viewport.3DVis.OpenMASK3.A"
    }
  }
}
}

```

This simulation application will be executed with a command like:

```
./tutorial tutorial.OpenMASK3
```

3.3.3 Creation of the application by specializing the main for a unique use

We suppose here that we have copied the previous main.cxx and Makefile files into a mainAlone.cxx and Makefile.alone, in order to be able to generate a new binary file named tutorialAlone.

In addition to the previous C++ code, you have to include the following files, in order to be able to add new simulated objects with their parameters.

```

#include <PsKernelObjectAbstractFactory.h>
#include <PsMultipleConfigurationParameter.h>
#include <PsUniqueConfigurationParameter.h>

```

Then, the creation of the simulation tree has to be done without configuration file, we have to add manually the simulated objects in the tree, with their initial state, which can be tedious. So, here we will use defaults values for the trajectory.

```

PsObjectDescriptor * simulationTree =
  new PsObjectDescriptor ("trajectoryRoot", "rootClass", NULL, NULL) ;
simulationTree->addSon (
  new PsObjectDescriptor ("Trajectory", "TrajectoryForTutorial", 75, NULL)) ;
PsConfigurationParameterDescriptor * trackerDesc =
  new PsMultipleConfigurationParameter () ;
trackerDesc->appendSubDescriptorNamed ("geometryFileName",
  new PsUniqueConfigurationParameter ("../Data/coneCyan.pfb")) ;
PsConfigurationParameterDescriptor * targetDesc =
  new PsMultipleConfigurationParameter () ;
targetDesc->appendSubDescriptor (
  new PsUniqueConfigurationParameter ("Trajectory")) ;
targetDesc->appendSubDescriptor (
  new PsUniqueConfigurationParameter ("position")) ;
trackerDesc->appendSubDescriptorNamed ("target", targetDesc) ;
simulationTree->addSon (
  new PsObjectDescriptor ("Tracker", "TrackerForTutorial", 75, trackerDesc)) ;
simulationTree->addSon (new PsObjectDescriptor ("Vis", "3DVis", 75, NULL)) ;

```

This simulation application will be executed with a command like:

```
./tutorialAlone
```

3.3.4 Instantiated simulated objects and their data stream connections

The execution of the two programs presented in the previous sections will result in the instantiation of three simulated objects that will be connected in the way presented in figure 2.

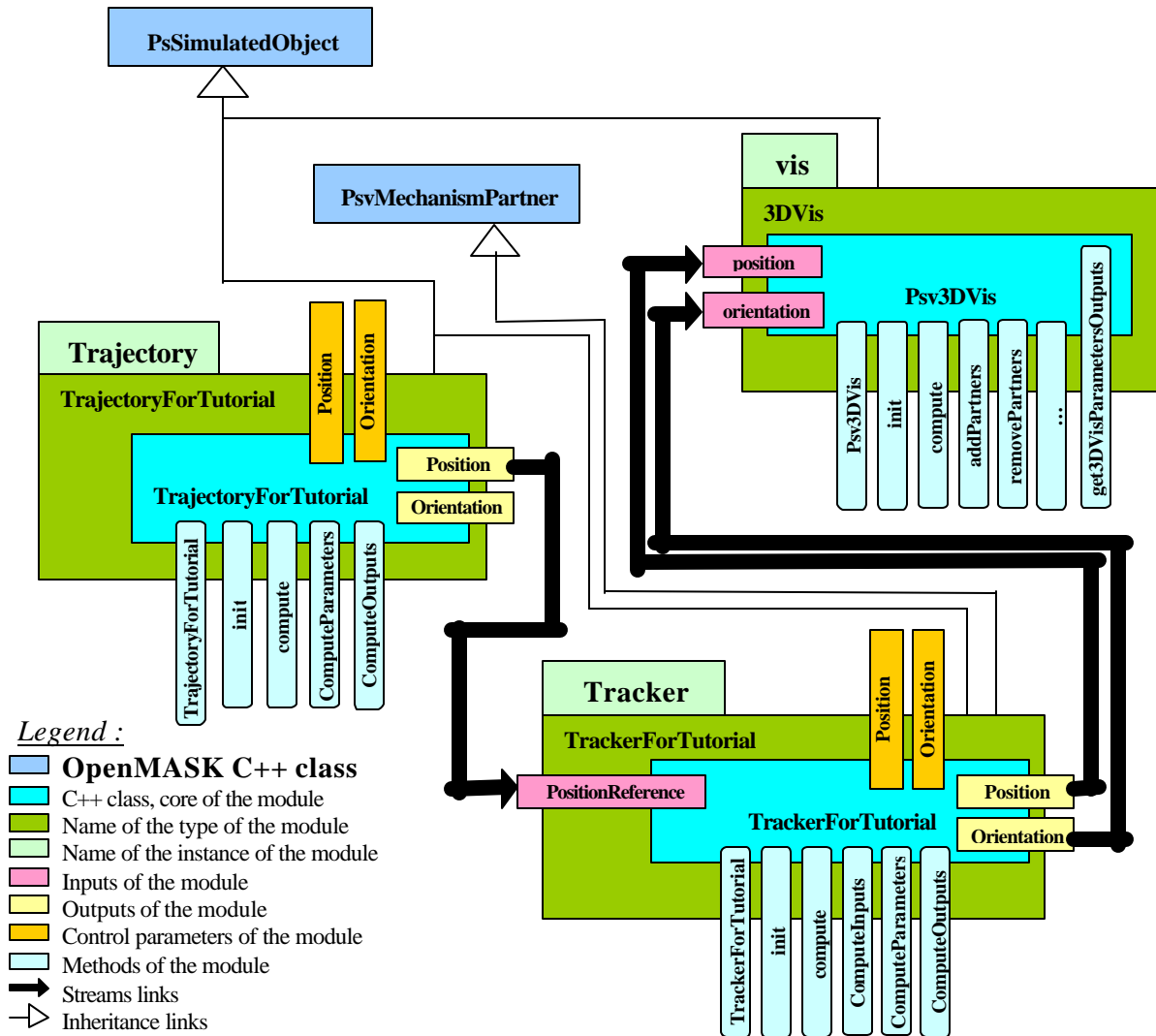


Figure 2. The simulated objects instantiated during the execution of the programs described in 3.3.2 and 3.3.3.

3.3.5 Case of a distributed simulation: referentials and mirrors

As we describe previously in the paragraph 2.3, OpenMASK's entities can be dispatched at run time across the network in order to distribute the computational weight among several workstations.

The operator must define which workstations are concerned with the distribution. Then he must declare where each simulated object will execute its calculations. Instantiations of referentials and mirrors are done automatically by the controller.

Each workstation will own one or several processes making the calculations or watching some of the results produced by the global simulation.

To obtain this possibility to dispatch the entities upon several process and workstations, we have to generate a special binary file associated to configuration files dedicated to distributed simulations.

The actual version of OpenMASK relies upon PVM for distributed simulations.

Some run-time aspects of the distributed simulations are strongly linked to PVM, so some changes must be made to allow a VR-OpenMASK application to be distributed upon the network.

3.3.5.1 The main file for distributed simulation

In the main file for a distributed simulation (usually we copy the initial main file and we add to it what is needed for PVM distribution), we have to add the following include:

```
#include <PsPvmController.h>
```

and to instantiate the corresponding type of controller:

```
...
PsPvmController * controler = new PsPvmController ( *simTree, 0, argc, argv) ;
...
```

3.3.5.2 Definition of some environment variables

Assuming that the PVM_ROOT environment variable is set, you also have to define the following ones in you .cshrc file (or any other kind of login file, so that these variables can be set when PVM spawns processes upon other workstations):

```
if (-d $PVM_ROOT) then
  setenv PVM_ARCH ` $PVM_ROOT/lib/pvmgetarch `
  setenv PVM_DPATH $PVM_ROOT/lib/pvmd
  set path=( $path $PVM_ROOT/bin/$PVM_ARCH $PVM_ROOT/lib $PVM_ROOT/lib/$PVM_ARCH)
  setenv PVM_EXPORT DISPLAY:PATH
  setenv PVM_SHMEM ON
endif
```

3.3.5.3 Modification of the Makefile

You have to add these lines in your makefile:

```
PVMLIBS = -L$(PVM_ROOT)/lib/$(PVM_ARCH) -lOpenMASKPvm -lgpvm3 -lpvm3
INCDIR += -I$(PVM_ROOT)/include
```

You have also to modify your target name so that you can have as many versions of binary files as you have different types of machines:

```
TARGETNAME = ./$(PVM_ARCH)/cooperationpvm
```

Last, you have to link with the PVM lib:

```
$(TARGETNAME): mainpvm.o $(SO)
  $(LINK) mainpvm.o $(DSO) $(PVMLIBS) -o $@
```

So you will have to build your binary file on each type of machine you want to use for your distributed simulations.

3.3.5.4 Example of a configuration file for a distributed simulation

In a simulation file dedicated to a distributed simulation, we have to add a Scheduling field in the Controller declaration, inside which we have to declare the authorized maximum latency (maximum de-synchronization between the different processes) and the names of the different processes of the simulation and the machine upon which these processes are executed:

```
#OpenMASK3
root {
  Class Controller
  Scheduling {
    Latency 20
    Machines {
      visualProcess malux.irisa.fr
      computeProcess bonux.irisa.fr
    }
  }
}
```

Then for each object declared in the simulation file, we have to say within which process it has to be executed:

```
Tracker {
  Class TrackerForTutorial
  Scheduling {
    Frequency 75
    Process computeProcess
  }
  UserParams {
```

```

        geometryFileName ../Data/coneCyan.pfb
        target [Trajectory position]
    }
}

```

3.3.5.5 Declaration of the directories used for a distributed simulation

This has to be done in a pvm description file, within which we have to declare, for each machine used in the distributed simulation, what is the working directory (where we can find the datas needed for the simulation), what is the execution path (where can we find the directory associated to the architecture of the machine containing the binary file to execute on this machine), and which PVM daemon has to be launched upon this machine:

```

&barbux.irisa.fr wd=$TutorialDir/Application
ep=$TutorialDir/Application/LINUX:$PVM_ROOT/bin/LINUX dx=$PVM_ROOT/lib/pvmd
&royaldelux.irisa.fr wd=$TutorialDir/Application
ep=$TutorialDir/Application/LINUX:$PVM_ROOT/bin/LINUX dx=$PVM_ROOT/lib/pvmd

```

3.3.5.6 How to launch the distributed simulation

Assuming that the previous pvm description file is named hosts.tutorial, we first have to execute the following command, usually in an independent shell:

```
pvm hosts.tutorial
```

Then we can start the distributed simulation in the working directory of one of the machines used by the simulation:

```
./tutorialPVM deux.multi
```

On this machine, tutorialPVM must be a symbolic link to the binary file to execute according to the architecture of the machine (LINUX/tutorialPVM for example for a Linux machine). The name of each binary file in each directory corresponding to a possible architecture must be the same than the name of this link, because PVM will spawn a process on each machine used in the distributed simulation, and it will try to launch the same binary file name in each directory specified thanks to the execution path associated to this machine.

4 VR-OpenMASK from a programmer point of view

4.1 Main characteristics of a simulated object

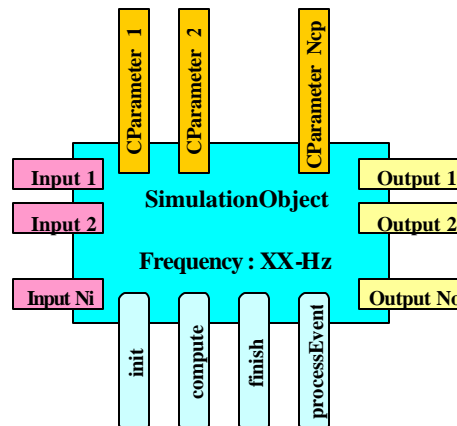


Figure 3. Main characteristics of a simulated object

The simulated object is the basic element we can use to construct virtual universes. It can have any number of inputs, control parameters and outputs. All these attributes are the communication interface of the simulated object. Control parameters play a role of an intermediary between the inputs and the outputs; for example, we can store some data in for memorizing certain values that would be used afterwards.

Before its actual instantiation, a simulated object is first described by a data structure provided by the kernel. The format of the data that are inside this structure is a set of pairs (dataId, effectiveData). The format of effectiveData is either a single or a compound value, according to what is needed by the programmer of the simulated object. This data structure (a descriptor) must be completed by the user of the module, usually in a configuration file. The set of all the descriptors corresponding to all the initial simulated objects, constitute the

simulation tree.

After the entity is created, its *init* method is invoked in order to make the proper initialization of the object, particularly the connections of its inputs to the outputs of other simulated objects. Then its *compute* method, within which should be the main part of the behavior of the object, is invoked according to its activation frequency. Last, before the object is destroyed, its *finish* method is invoked, to allow it to end properly. The only use of classical C++ constructor and destructor could not allow to initialize and finish properly a simulation, because generally, at the beginning of a simulation, all the objects have to be created before we can enable properly the links between inputs and outputs, that is why the connections are made in the *init* method rather than in the C++ constructor.

A simulated object is able to process events and messages thanks to its *processEvent* method, which is invoked at each simulation step, as many times as needed to process all the events the object has received. These events can be either user-defined events, which can come from any simulated objects, or system events, which come usually from the simulation controller.

4.2 How to create new simulated objects types

4.2.1 The PsSimulatedObject class

All simulated objects in OpenMASK are inherited from one basic class: *PsSimulatedObject*. Objects that inherit from this model will possess essential functions, such as *compute()*, which does some necessary computation in every simulation step, and *processEvent()* that handles messages sent by itself or others. Of course, those inherited objects must have their functions rewritten to have their own behavior.

At each simulation step, the inputs of the object can be read by the calculation function: *compute()*, which is responsible for handling these input values, then for computing the internal state of the object and updating the values of the control parameters, and finally for updating the outputs thanks to the values of the new internal state (usually the values of some control parameters).

The inputs can be connected to the outputs of other objects at different stages of the simulation by either naming the connected objects or asking the controller for an appropriate object. Moreover, the kernel is in charge of ensuring that the value provided to the object is consistent with the value from the output.

4.2.2 OpenMASK PsType interpolation distribution

As different simulated objects do not work at the same frequencies, they do not produce data at the same date. So the data are dated. And when a simulated object needs a data at a given date, if it has not been produced at this date, the kernel does automatically an interpolation or an extrapolation to adapt the data.

4.2.3 DSO Libraries

As we do for the kernel and each VR-OpenMASK component, we suggest to put together all the binary codes corresponding to a set of dependant simulated objects in one dynamic shared library. Then, it is easier to reuse and to combine to build new applications.

4.3 The Interactive 3D visualizer and its partners

Programming a virtual reality application involves visualizing a 3D scene, animating its 3D objects and interacting with them. We describe in the 3.2 paragraph, from the user point of view, a VR-OpenMASK class providing such functionalities. To do that, an interactive 3D visualizer object communicates with some simulated objects of the application concerned with the animation of "their" 3D objects in the scene. These simulated objects are called 3Dvis partners.

In this part, we describe, from the programmer point of view, the programming interface of the 3DVis partner to correctly inter-operate with the visualization service.

You can have several visualizer objects during a session, particularly when you distribute your session for collaborative work. Each visualizer proceeds in the same way.

4.3.1 Interactive 3D Visualizer principles

The visualisation is based on a collaboration between the interactive 3D visualizer object and one or more 3DVis

partners.

4.3.1.1 Animation principles

The visualizer is in charge of modelling and visualisation through an adhoc 3D graphic library: presently the OpenGL Performer library, shortly also the OpenSG library.

At the init step, the visualizer:

- initialises the scene graph with a scene node, a light source
- opens the pipes, windows, viewports according to the configuration file
- list all the 3Dvis partners, for each one:
 - contact it to get the sub-grap modelling the 3D objects it manages,
 - insert it in the scene graph,
 - install all the input handler that will manage the data arriving from the partner to the visualizer

At compute step, the visualizer get data from its partners and update the scene graph, reflecting the animation.

As you see, the visualizer prepare all what is need for modelling and visualisation, but these are the partners that provide the actual datas for the modelling and the animation.

4.3.1.1.1 Animation partners

To be a good partner, a simulated object has also to inherit from a partner class which provides a method to give the geometric sub-graph. Then it uses the visualizeOutput method to declare which outputs have to be handled to manage the animation.

4.3.1.1.2 Input handlers

All the animation is managed through the different input handlers installed for each visualizable output of the partners.

An input handler connects a partner output to a visualizer input, and he matches this input with a named node of the scene sub-graph associated to the partner. This node is called an animation point of the partner.

For each partner compute step, this partner have to set a new value of its output relative to the animation it wants to get.

For each visualizer compute step, the input handler to get the new value and update the animation point relative to the type of input handler it is.

4.3.1.2 Visualization Principle

For simple VR -OpenMASK application, visualization control is done through the configuration file parameters as seen in the 3.2 paragraph.

4.3.1.2.1 Visualization control events

But if you need, several visualisation parameters can be controlled through events to the visualizer:

"light" , "interOcularDistance" , "eyeAngle" , "clipping" , "fov" , "associateObservable" , "cameraNumber" , "viewportSize" , "viewportOffset" , "displayStatistics" , "levelOfStatistics" , "cullFace" , "saveImages" , "earthSky" .

4.3.1.3 Interaction principles

With the OpenGL Performer library, visualization is done through X11 windows, so interactions based on the visualization are managed by the visualizer and forwarded to the concerned simulated objects thanks to events.

The visualizer catches X11 events, encapsulates them in OpenMASK signals ("XEvent") and dispatches them

through the simulation (it fires a signal "XEvent" encapsulating the X11 event).

For picking interaction, the visualizer offers four picking services: 2DPick, 3DPick, multiPick and OmniPick. They are called upon events.

- 2Dpick: used for screen to world-space ray intersections on a viewport's scene. Intersections will only occur with parts of the database that are within the viewing frustum, and that are enabled for picking intersections.
 - parameters: pipe number, window number, x and y coordinates
 - return: "picked"<objectName, animatedNodeName> or "noPicked"
- 3Dpick: used for world space ray intersections on the scene. Intersections will occur with parts of the databases that are enabled for picking intersections.
 - parameters: ray position and orientation, ray length
 - return: "picked"<objectName, animatedNodeName> or "noPicked"
- multiPick: used for world space rays intersections on the scene. Intersections will occur with parts of the databases that are enabled for picking intersections.
 - parameters: rays position and orientation, rays lengths
 - return: "picked"<objectName, animatedNodeName> or "noPicked"
- omniPick: used for world space rays intersections on the scene. Intersections will occur with parts of the databases that are enabled for picking intersections.
 - parameters: position of the original point of the rays, rays lengths
 - return: "picked"<objectName, animatedNodeName> or "noPicked"

4.3.2 Predefined input handlers

We provide a set of input handlers to manage moving of 3D objects or parts of 3D objects. We have focused on this kind of animation because our main application field was mechanisms.

Here are some examples:

- PsvEulerRotationInputHandler,
- PsvQuaternionInputHandler,
- PsvTranslationInputHandler,
- PsvScaleInputHandler ,
- PsvAxisRotationInputHandler,
- PsvTranslationXRotationInputHandler
- ...

All this classes inherit from an abstract template class that gives the model to build new input handlers classes.

4.3.3 Predefined partners

We provide two examples of partners:

- PsvMechanismPartner: loads a geometry file – but clones a geometry instead of loading the same file twice, so warning: the geosets are not duplicated and their eventual manipulation can become tricky. To avoid this, we can set the “clone” parameter of the visualization to “false”. Configuration parameter:
 - GeometryFileName "path/name"

- PsvDecorPartner: allows to position and orientate a geometry file in a 3D scene to act as a static decor. Inserts a pfSCS node at the top of the partner scene subgraph. Configuration parameters:
 - geometryFileName "path/name"
 - position [x y z]
 - orientation [h p r]

Here too, these classes inherit from an abstract class that gives the model to build new partners classes.

4.4 Example of a new visualizable simulated object type creation

We are going to show how to write a simple 3D object, which behavior is only to stay somewhere relatively to another object. We will call this kind of object a ShiftedForTutorial object. It will be a visualizer's partner.

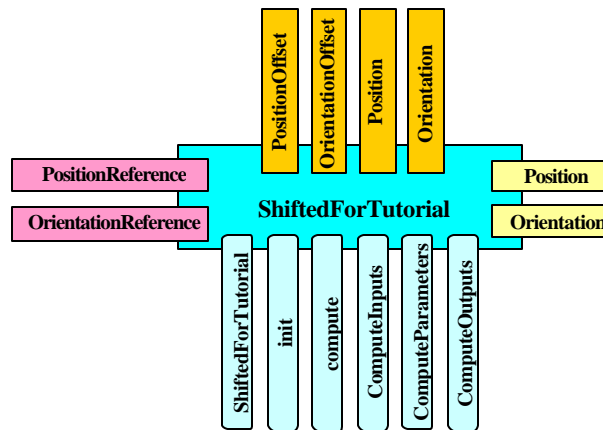


Figure 4. The ShiftedForTutorial simulated object

The attributes of this simulated object will be:

- two inputs: the position and orientation of its reference object;
- four control parameters: the offsets for position and orientation relatively to the reference object, and the effective position and orientation of the object itself;
- two outputs: its absolute position and orientation, so that it can be visualized.

Its interesting methods will be:

- the constructor initializes the references for inputs, control parameters and outputs; it gives initial values to the control parameters and the outputs; it makes the outputs able to be used by the 3D visualizer; and it reads from the configuration parameters the initial values for the offsets of position and orientation.
- the init method reads from the configuration parameters the values of the names of the reference object and of its outputs (it could have been done in the constructor), and then it connects the inputs to the outputs of this reference object.
- the compute method, which will invoke the three following methods:
 - the computeInputs method reads the values of the inputs.
 - the computeParameters method calculate new values for the position and orientation control parameters according to the values read from the inputs (the values of the reference position and orientation) and to the values of the offsets (stored in the corresponding control parameters).
 - the computeOutputs method puts the values of the control parameters for position and orientation into the corresponding outputs.

4.4.1 Writing the C++ classes

4.4.1.1 The ShiftedForTutorial.h file

```
#ifndef ShiftedForTutorialHEADER
```

```

#define ShiftedForTutorialHEADER

#include <PsSimulatedObject.h>
#include <PsvMechanismPartner.h>
#include <PsOutput.h>
#include <PsControlParameter.h>
#include <PsTranslation.h>
#include <PsHPRRotation.h>

class ShiftedForTutorial : public PsSimulatedObject,
                          public PsvMechanismPartner {

public :

    ShiftedForTutorial (PsController& ctrl,
                       const PsObjectDescriptor& objectDescriptor) ;
    virtual ~ShiftedForTutorial (void) ;

    virtual void init (void) ;
    virtual void compute (void) ;

protected :

    virtual void computeInputs (void) ;
    virtual void computeParameters (void) ;
    virtual void computeOutputs (void) ;

    PsInput<PsTranslation> & referencePositionInput ;
    PsInput<PsHPRRotation> & referenceOrientationInput ;

    PsControlParameter<PsTranslation> & positionParameter ;
    PsControlParameter<PsHPRRotation> & orientationParameter ;
    PsControlParameter<PsTranslation> & positionOffsetParameter ;
    PsControlParameter<PsHPRRotation> & orientationOffsetParameter ;

    PsOutput<PsTranslation> & positionOutput ;
    PsOutput<PsHPRRotation> & orientationOutput ;

    PsTranslation referencePosition ;
    PsHPRRotation referenceOrientation ;
    PsTranslation position ;
    PsHPRRotation orientation ;
    PsTranslation positionOffset ;
    PsHPRRotation orientationOffset ;

} ;

#endif

```

4.4.1.2 The ShiftedForTutorial.cxx file

```

#include <ShiftedForTutorial.h>
#include <PsConfigurationParameterDescriptor.h>
#include <PsvTranslationInputHandler.h>
#include <PsvHPRRotationInputHandler.h>
#include <PsSystemEventIdentifier.h>
#include <sstream>

ShiftedForTutorial::ShiftedForTutorial
(PsController & ctrl, const PsObjectDescriptor & objectDescriptor)
: PsSimulatedObject (ctrl, objectDescriptor), PsvMechanismPartner (),
  positionOutput (addOutput<PsTranslation>
                 ("position", new PsPolator<PsTranslation> ())),
  orientationOutput (addOutput<PsHPRRotation>
                    ("orientation", new PsPolator<PsHPRRotation> ())),
  positionParameter (addControlParameter<PsTranslation> ("position")),
  orientationParameter (addControlParameter<PsHPRRotation>
                       ("orientation")),
  positionOffsetParameter (addControlParameter<PsTranslation>
                           ("positionOffset")),

```

```

orientationOffsetParameter (addControlParameter<PsHPRRotation>
                                ("orientationOffset")),
referencePositionInput (addInput<PsTranslation> ("referencePosition",
                                                true)),
referenceOrientationInput (addInput<PsHPRRotation>
                            ("referenceOrientation", true)),
position (0, 0, 0),
orientation (0, 0, 0) {
visualiseOutput<PsvTranslationInputHandler> (positionOutput,
                                                "DCS_position");
visualiseOutput<PsvHPRRotationInputHandler> (orientationOutput,
                                                "DCS_orientation");
if (getConfigurationParameters () != NULL) {
    if (getConfigurationParameters ()
        ->getSubDescriptorByName ("positionOffset") != NULL) {
        for (int i = 0 ; i < 3 ; i ++) {
            PsFloat temp;
            istringstream is (getConfigurationParameters ()
                ->getSubDescriptorByName ("positionOffset")
                ->getSubDescriptorByPosition (i)
                ->getAssociatedString ().c_str ());
            is >> temp ;
            positionOffset [i] = temp ;
        }
    }
    if (getConfigurationParameters ()
        ->getSubDescriptorByName ("orientationOffset") != NULL) {
        for (int i = 0 ; i < 3 ; i ++) {
            PsFloat temp ;
            istringstream is (getConfigurationParameters ()
                ->getSubDescriptorByName ("orientationOffset")
                ->getSubDescriptorByPosition (i)
                ->getAssociatedString ().c_str ());
            is >> temp;
            orientationOffset [i] = temp ;
        }
    }
} else {
    cerr << getName () << " : no configuration file" << endl ;
    sendEvent (getController (), PsSystemEventIdentifier::MaskStop) ;
}
positionOffsetParameter.set (positionOffset) ;
orientationOffsetParameter.set (orientationOffset) ;
positionParameter.set (positionOffset) ;
orientationParameter.set (orientationOffset) ;
computeOutputs () ;
}

ShiftedForTutorial::~ShiftedForTutorial (void) {

}

void ShiftedForTutorial::init (void) {
    PsString referenceName ;
    if (getConfigurationParameters ()
        ->getSubDescriptorByName ("referenceName") != NULL) {
        referenceName = getConfigurationParameters ()
            ->getSubDescriptorByName ("referenceName")
            ->getAssociatedString () ;
    }
    PsString positionName ;
    if (getConfigurationParameters ()
        ->getSubDescriptorByName ("referencePositionName") != NULL) {
        positionName = getConfigurationParameters ()
            ->getSubDescriptorByName ("referencePositionName")
            ->getAssociatedString () ;
    }
    referencePositionInput.connect (referenceName, positionName) ;
    PsString orientationName ;
    if (getConfigurationParameters ()

```

```

        ->getSubDescriptorByName ("referenceOrientationName") != NULL) {
            orientationName = getConfigurationParameters ()
            ->getSubDescriptorByName ("referenceOrientationName")
            ->getAssociatedString () ;
        }
        referenceOrientationInput.connect (referenceName, orientationName) ;
    }

void ShiftedForTutorial::compute () {
    computeInputs () ;
    computeParameters () ;
    computeOutputs () ;
}

void ShiftedForTutorial::computeInputs (void) {
    referencePosition = referencePositionInput.get () ;
    referenceOrientation = referenceOrientationInput.get () ;
}

void ShiftedForTutorial::computeParameters (void) {
    positionOffset = positionOffsetParameter.get () ;
    orientationOffset = orientationOffsetParameter.get () ;
    pfCoord coordOffset, coordReference, coordResultat ;
    pfMatrix matOffset, matReference, matResultat ;
    for (int i = 0 ; i < 3 ; i ++ ) {
        coordOffset.xyz [i] = positionOffset [i] ;
        coordOffset.hpr [i] = orientationOffset [i] ;
        coordReference.xyz [i] = referencePosition [i] ;
        coordReference.hpr [i] = referenceOrientation [i] ;
    }
    matOffset.makeCoord (&coordOffset) ;
    matReference.makeCoord (&coordReference) ;
    // shifting relative to the reference
    matResultat.mult (matOffset, matReference) ;
    matResultat.getOrthoCoord (&coordResultat) ;
    for (int i = 0 ; i < 3 ; i ++ ) {
        position [i] = coordResultat.xyz [i] ;
        orientation [i] = coordResultat.hpr [i] ;
    }
    positionParameter.set (position) ;
    orientationParameter.set (orientation) ;
}

void ShiftedForTutorial::computeOutputs (void) {
    positionOutput.set (positionParameter.get ()) ;
    orientationOutput.set (orientationParameter.get ()) ;
}

```

4.4.2 Writing the Makefile and generating the dynamic library

```

ifdef OpenMASKConfigFile
    include $(OpenMASKConfigFile)
else
    include $(OpenMASKDIR)/kernel/make.in
endif

ifdef PFROOT
    PERFORMERINCLUDES = -I$(PFROOT)/usr/include/Performer \
        -I$(PFROOT)/usr/include/Performer/pfdb \
        -I$(PFROOT)/usr/include/Performer/pfui \
        -I$(PFROOT)/usr/include/Performer/pfutil \
        -I$(PFROOT)/usr/include/Performer/pf \
        -I$(PFROOT)/usr/include/Performer/pr
endif

COMPILE = $(COMPILER) $(COMPILEFLAGS) \
    $(OPENMASKFLAGS) $(PERFORMERINCLUDES) -g -O2

LINK = $(LINKER) $(LINKFLAGS) -L/usr/lib/libpfdb -Wl,-nohibit-exec

INCDIR = -I../inc\

```

```

-I$(OpenMASKDIR)/kernel/inc \
-I$(OpenMASKDIR)/contrib/3DVis/inc \
-I$(OpenMASKDIR)/contrib/3DVisPartners/inc \
-I$(OpenMASKDIR)/contrib/userTypes/math/inc \
-I$(OpenMASKDIR)/contrib/userTypes/events/inc \
-I$(OpenMASKDIR)/contrib/3DVisInputHandlers/math/inc \
-I$(OpenMASKDIR)/contrib/3DVisInputHandlers/switch/inc \
-I$(PCCTSDIR)/h

OBJ = ShiftedForTutorial.o

TARGETLIBNAME = $(MyApplicationDIR)/lib/libTutorial.so

%.o : %.cxx
    $(COMPILE) $(INCDIR) -c $< -o $@

$(TARGETLIBNAME): $(OBJ)
    $(LINK) $(OBJ) -o $@

cleanall:
    rm -rf ii_files/ *.o $(TARGETLIBNAME)

```

4.4.3 Modifying the main.cxx file, its associated Makefile, and a configuration file

4.4.3.1 C++ code to add in the main.cxx file

```

...
#include <ShiftedForTutorial.h>

...
    controller->addInstanceCreator ("ShiftedForTutorial ",
        new PsSimpleSimulatedObjectCreator <ShiftedForTutorial> ) ;
...

```

4.4.3.2 C++ code to add in the Makefile file

```

INCDIR = ...
    -I$(TutorialModulesDIR)/inc \
    ...

SO = ...
    $(MyApplicationDIR)/lib/libTutorial.so \
    ...

```

4.4.3.3 Code to add in a configuration file

```

WhiteCone {
    Class ShiftedForTutorial
    Scheduling {
        Frequency 75
    }
    UserParams {
        geometryFileName ../Data/WhiteCone.pfb
        referenceName Tracker
        referencePositionName position
        referenceOrientationName orientation
        positionOffset [-5 0 0]
        orientationOffset [0 0 0]
    }
}

```

5 VR-OpenMASK from a contributor point of view: a new interaction set

We present here the communication principles between interactive objects and interaction tools: we have to teach to these objects a common communication protocol in order to allow them to establish dynamic connections between them, so enabling the interaction tools to manipulate the interactive objects.

5.1 The adapters: they allow to make interactive an existing simulated object

This can be realized *a posteriori*, if the simulated object follows some defined rules. In that case, we can use some classes that will teach a simulated object the communication protocol allowing this object to be manipulated by another one.

5.1.1 Rules which must be respected for the objects that must be made interactive

First, in order to allow an object to be interactive using our adapters, the designer of a simulated object class has to store the state of its simulated object into control parameters. The adapters will be able to override the values of some of these control parameters during the interactions.

Then, the designer has to split the compute method of this class into 3 methods: computeInputs, computeParameters and computeOutputs. Typically, an adapter will override the computeInputs and computeParameters methods to take the interactions into account thanks to new inputs, calculating new values for some control parameters, and letting the initial compute outputs provide the new outputs values thanks to the control parameters values.

5.1.2 Functioning principle of an adapter

The adapters are a set of C++ template classes that know how to respond to a request for interaction and what to do then in order to make the interaction possible. The principal classes are called protocol teachers, they are in charge of the communication protocol, so they know how to initiate and how to finish an interaction, but, in order to be the most generic possible, they do not possess the knowledge of how the interaction behaves, it is delegated to a connector.

To realize that, a protocol teacher is associated to an instance of a class that is able to provide a connector that knows what to do when accepting or finishing an interaction. The connector provider is able to change dynamically this connector, thus allowing the interactive object to behave differently during a simulation session, according to its needs.

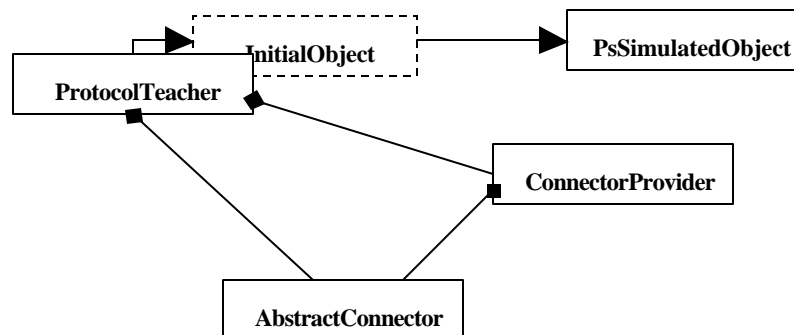


Figure 5. Class diagram of the main classes involved in the process to make interactive a simulated object

5.1.3 Examples of existing adapters

5.1.3.1 The ProtocolTeacher class

This class implements the interactive object side of a basic communication protocol for interaction, without knowledge about the exact nature of the interaction (neither the types of the parameters to control nor the way the values will be calculated), it only knows the numbers of parameters to control (indicated by a template parameter). It knows how to answer to messages like “Control takeover”, “Control release”, “changeConnector”. Its policy is to always agree to a request for control takeover by an interactor, sending it a “Control takeover acceptance” message, so it sometimes needs to end the current interaction by sending a “Control takeover end” to the current interactor before accepting a new one. It also always accepts the release of the control by sending a “Control release acceptance” to the current interactor.

When agreeing an interaction, it asks its associated ConnectorProvider to obtain a connector that will be able to create new inputs to this ProtocolTeacher and to connect them to the corresponding outputs of the interactor asking for interaction.

Then, at each simulation step, during its compute method, the ProtocolTeacher will invoke:

- its computeInputs method, which first invokes the computeInputs method inherited from the initial encapsulated object, and then invokes a method of its associated connector able to read its new inputs;
- its computeParameters method, which in its turn invokes the computeParameters method inherited from the initial encapsulated object, and then invokes a method of its associated connector able to override the values of some control parameters according to the values of the new inputs provided by the interactor currently in interaction;
- its inherited computeOutputs method, which will provide new output values according to the new values of the control parameters.

This class cannot be used as is, it has to be derived in order to be able to choose properly the number and types of the parameters of the initial object to control.

5.1.3.2 The DoubleProtocolTeacher class

This class inherits from the ProtocolTeacher class, and allows interacting with an object in order to take control of at most two of its control parameters. It is often used to control the position and orientation of objects.

5.1.3.3 The ConnectorProvider class

This abstract class defines what is needed to be able to provide a connector to a ProtocolTeacher. Initially, the choice of this connector is made according to a declaration in the parameters of the ProtocolTeacher, or according to a default value if there is no such declaration in the configuration parameters of the ProtocolTeacher. Then this choice of connector can evolve dynamically according to the decision of the end-user, who is able to change interactively the current connector associated to an interactive object, in order to make it behave differently, by sending a “changeConnector” message to the ProtocolTeacher.

In this class, the changeConnector method is pure virtual, so it has to be defined in a derived class.

5.1.3.4 The DoubleConnectorProvider class

This class inherits from the ConnectorProvider class, and defines its changeConnector method, thus proposing several connectors: SimpleConnector, DoubleConnector and ConstraintDefaultAdaptorOffset.

5.1.3.5 Some connectors

We propose several connectors, which can be used with our default ProtocolTeacher, but which could also be used by other kinds of protocol teacher.

5.1.3.5.1 The SimpleConnector class

This class allows an interactor to override the value of one control parameter of one interactive object during one interaction. The type of the control parameter is a template parameter, so it can control any kind of parameter. It is often used to control the position or the orientation of an interactive object, but can also be used for example to control its scale or its color.

5.1.3.5.2 The DoubleConnector class

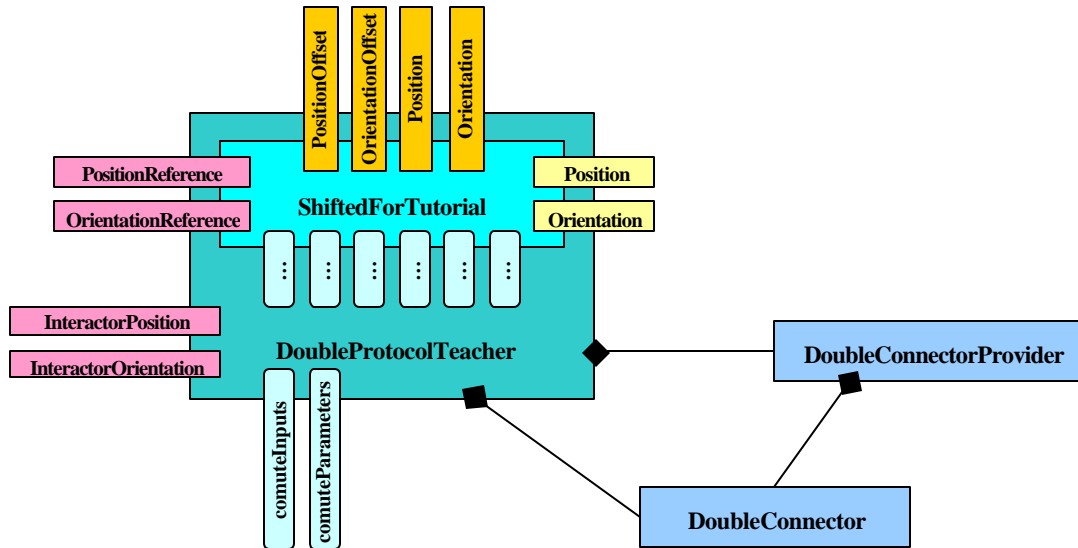
This class allows an interactor to override the values of two control parameters of one interactive object during one interaction. The types of the control parameters are template parameters, so it can control any kind of parameters. It is often used to control the position and the orientation of an interactive object.

5.1.3.5.3 The ConstraintDefaultAdaptorOffset class

This class allows a user to override the value of two control parameters of one interactive object during one interaction. The types of the control parameters are PsTranslation and PsHPRRotation, and they can be constrained between two values for each of the six degrees of freedom.

5.1.4 How to use an adapter

Figure 6. How to use an adapter to make interactive a simulated object



5.1.4.1 Declaration in the main file

To allow the instantiation of interactive object, we have to associate to a symbolic name the instantiation of an object encapsulated within a protocol teacher. For example, in order to make interactive an instance of the MotionLess class, to allow taking control of its position and orientation parameters, we have to write the following C++ instruction in the main.cxx file :

```
controller->addInstanceCreator ("InteractiveShiftedForTutorial",
    new PsSimpleSimulatedObjectCreator
    <DoubleProtocolTeacher
    <ShiftedForTutorial, PsTranslation, PsHPRRotation> > ());
```

5.1.4.2 Declaration in the configuration file

To allow the correct instantiation and initialization of an interactive object, we have to provide the same values than those needed by the initial object, and also to provide new values dedicated to the adapter. We can choose either to take control of the offsets for position and orientation of the object relatively to its reference (the result of the interaction will be persistent), or to take control of its position and orientation (but the effects of the interaction will cease at the end of the interaction).

```
WhiteCone {
  Class InteractiveShiftedForTutorial
  Scheduling {
    Frequency 75
  }
  UserParams {
    geometryFileName ../Data/whiteCone.pfb
    referenceName BlueCone
    referencePositionName position
    referenceOrientationName orientation
    positionOffset [-5 0 0]
    orientationOffset [0 0 0]
    // adaptor's parameters
    connectorClassName DoubleConnector
    controlParameterNames [position orientation]
    interactorOutputNames [interactorPosition interactorOrientation]
    adaptorAwarenessKind [boundingBox]
    adaptorInteractiveAreaAwareness no
    adaptorAwarenessNode DCS_position
  }
}

YellowCone {
  Class ShiftedForTutorial
```

```

Scheduling {
  Frequency 75
}
UserParams {
  geometryFileName ../Data/yellowCone.pfb
  referenceName BlueCone
  referencePositionName position
  referenceOrientationName orientation
  positionOffset [5 0 0]
  orientationOffset [0 0 0]
  // adaptor's parameters
  connectorClassName DoubleConnector
  controlParameterNames [positionOffset orientationOffset]
  interactorOutputNames [interactorPosition interactorOrientation]
  adaptorAwarenessKind [boundingBox]
  adaptorInteractiveAreaAwareness no
  adaptorAwarenessNode DCS_position
}
}

```

5.2 The interactors: they allow to learn the interaction protocol to an interaction tool

The interactors are symmetrical to the adapters: they are a set of C++ template classes which know how to ask for interaction to an interactive object, and which kind of outputs are needed by these interactive objects in order to propose new values for some of their control parameters.

So, they are in charge of the communication protocol: they know how to initiate and how to finish an interaction, but, in order to be the most generic possible, they do not possess the knowledge of how it is possible to select an interactive object in order to interact with it. Thus they are in charge of teaching the interaction communication protocol to a virtual pointing or selecting device.

note: faire une nouvelle démo, avec les outils virtuels en plus par rapport à la démo précédente, sans partage d'interaction sur un même objet

5.2.1 Functioning principle of an interactor

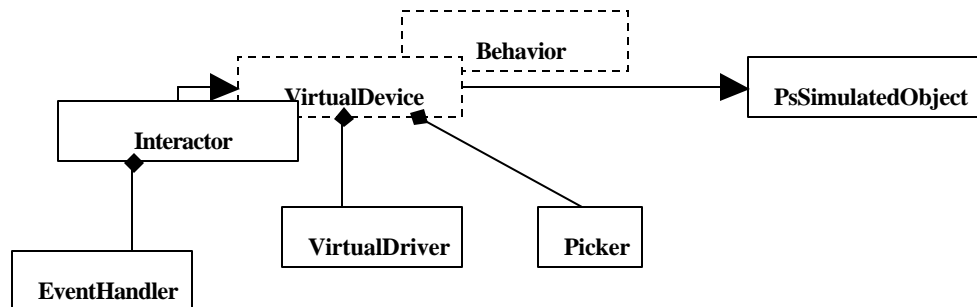


Figure 7. Class diagram of the main classes involved in the process to create an interaction tool

We suppose that a virtual interaction tool is only a virtual device able to select an object in a virtual universe and to propose new values for some control parameters of this object. An interaction tool must be strongly separated from the physical devices that can be used to trigger and control it, in order to be able to change these physical devices easily, or even to emulate them. It must also be strongly separated from the communication protocol, in order to be able to control different kinds of interactive objects that implement different communication protocols for interaction.

So, an interactor will have to teach a communication protocol to such a virtual interaction tool: it will be a C++ template class encapsulating the virtual device, able to process some messages from the interactive objects. It must also be able to process messages from an associated object in charge of the triggering of the selection on the objects, and to process messages from an associated picker object.

5.2.2 Rules which must be respected for an interaction tool to use with an interactor

5.2.2.1 Providing a doPick method

In order to be encapsulated within an interactor, an interaction tool must implement a doPick method, which aim is to try to obtain the name of the simulated object to be controlled.

5.2.2.2 Providing an interactorConnection method

The interaction tool has also to provide an interactorConnection method, which is in charge of the correct initialization of the values that will be provided to the simulated object to be controlled.

This method has several parameters: the values of the parameters to control at the time of the control takeover, in order to be able to compute the good values to override the control parameters of the object in interaction.

5.2.3 Examples of interactors and interaction tools

5.2.3.1 The NInteractor class

This class implements the interactor side of the basic communication protocol that the ProtocolTeacher can understand.

It is able to process some messages from the interactive objects like: “Control takeover acceptance”, “Control release acceptance”, “Control takeover end” , “Control release”, “Control takeover denial”. It is also be able to process messages from an associated object in charge of the triggering of the selection on the objects: “Trigger”, and to process messages from an associated picker object: “picked” and “noPicked”.

When the interactor receives the “Trigger” message, it triggers the doPicking method of the virtual interaction tool. Then, according to the result of the picking, it will receive either a “noPicked” or a “picked” message. If it receives a “noPicked” message, it will end the current interaction if there is one, by sending a “Control release” message to the object currently in interaction. If it receives a “picked” message, it will try the same way to end the current interaction if there is one, and then, it will send a “Control takeover” message to the picked object, unless this object was already in interaction before the picking result.

As the ProtocolTeacher class, this class cannot be used as is, it has to be derived in order to be able to choose properly the number and the types of the parameters to control.

5.2.3.2 The DoubleInteractor class

This class inherits from the NInteractor class, and enables the control takeover of at most two control parameters (usually the position and orientation of a simulated object).

It defines the outputs that an interactive object will need for a control takeover, and the associated computeOutputs method that will provide values for these outputs according to the values of control parameters that the encapsulated interaction tool will compute, according to its own associated behavior.

5.2.3.3 Some interaction tools

We present here some virtual interaction tools that can be used with the DoubleInteractor class.

5.2.3.3.1 The VirtualRay class

It needs a support: it can be a driver for a physical 3D device (3D magnetic captor) or any simulated object able to provide a position and an orientation, so it is really independent from any physical device.

It needs also an associated picker, able to provide the name of a simulated object in response to a picking request. The most commonly used picker is the 3D interactive visualizer.

So it implements the doPick method, by sending to its associated picker a “3DPick” message parameterized by the position, the orientation and the length of the virtual line used to do the 3D picking.

It needs also an associated behavior: it is the way the virtual ray calculates the new values that should override the values of the control parameters of the object in interaction with. We provide a RayBehavior, which propose

new values of position and orientation so that the object in interaction stays always at the same place relatively to the virtual ray, i.e. the object stays stuck upon the virtual ray.

5.2.3.3.2 The VirtualHand class

It inherits from the VirtualRay, and overrides only the doPick method, which sends a “multiPick” message to its associated picker, parameterized by the position and the orientation of the 3D point from which must leave the virtual lines for the multi directional picking: it defines the half-sphere within which the objects are searched, as it is also parameterized by the radius of this sphere.

5.2.3.3.3 The Virtual3DCursor class

Like the VirtualHand class, the Virtual3DCursor class inherits from the VirtualRay class and overrides the doPick method. This method sends an omniPick message, parameterized by the position and the radius of the sphere within which this omni directional picking occurs.

5.2.4 How to use an interactor

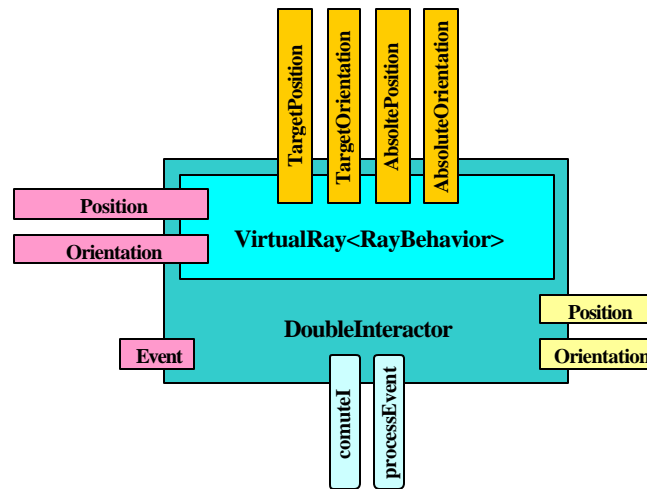


Figure 8. Example of interactor: a virtual ray

5.2.4.1 Declaration in the main file

To allow the instantiation of a virtual interaction tool, we have to associate to a symbolic name the instantiation of an object encapsulated within an interactor. For example, in order to enable a virtual ray to control the position and the orientation of an interactive object, making the controlled object attached to the virtual ray, we have to write the following C++ instruction in the main.cxx file:

```
controller->addInstanceCreator ("VirtualRay",
    new PsSimpleSimulatedObjectCreator
    <DoubleInteractor
    <VirtualRay <RayBehavior>, PsTranslation, PsHPRRotation> > ());
```

5.2.4.2 Declaration in the configuration file

To allow the correct instantiation and initialization of a virtual interaction tool, we have to provide some values needed by the virtual interaction tool, and also to provide new values dedicated to the interactor:

```
blueVirtualRay {
  Class VirtualRay
  Scheduling {
    Frequency 75
  }
  UserParams {
    // VirtualRay's parameters
    geometryFileName ./data/blueRay.pfb
    positionToFollow [raySupport position]
    orientationToFollow [raySupport orientation]
    associatedPicker vis
  }
}
```

```

// DoubleInteractor's parameters
targetPositionParameter proposedTargetPosition
targetOrientationParameter proposedTargetOrientation
interactorPositionOutput interactorPosition
interactorOrientationOutput interactorOrientation
}
}

```

To have a trigger for this virtual ray, we can use for example an `InteractorTrigger`, which is here a filter for a serial mouse driver, and which responsibility is to send a “Trigger” message to its associated interactor when the user press the left button of the mouse.

```

blueVirtualRayInteractorTrigger {
  Class InteractorTrigger
  Scheduling {
    Frequency 75
  }
  UserParams {
    associatedInteractor blueVirtualRay
    associatedMouse theMouse
  }
}

```

We can also use another object, which subscribes to X11 events, and which process the `LeftButtonPressed` event by sending a “2DPick” message to its associated picker, and then by sending a “Trigger” message to the object associated to a “picked” message from this picker.

```

xMouseTriggerA {
  Class XMouseTrigger
  Scheduling {
    Frequency 75
  }
  UserParams {
    associatedPicker visA
  }
}

```

5.3 How to create new adapters

For example, if we want to allow several people to share control upon an interactive object, we have to create a new adapter, which will be able to accept several control takeover, at the same time, from different interactors.

5.3.1 The `DoubleMultipleProtocolTeacher` class

This class inherits from the `DoubleProtocolTeacher` class and overrides the processing of the `ControlTakeover` message in order to allow several interactors to interact simultaneously with one interactive object.

It overrides also the type of the connector provider to be used: a `DoubleMultipleConnectorProvider` instead of a `DoubleConnectorProvider`.

Last, it overrides the default connector to be employed: a `DoubleMultipleConnector` instead of a `DoubleConnector`.

5.3.2 The `DoubleMultipleConnectorProvider` class

This class inherits from the `DoubleConnectorProvider` class, and overrides the `changeConnector` method in order to be able to provide a `DoubleMultipleConnector`.

5.3.3 The `DoubleMultipleConnector` class

This class inherits from the `DoubleConnector` class and allows several interactors to be in interaction at the same time and combine their proposed values to override the values of two control parameters of the interactive object during interaction. The types of the control parameters are `PsTranslation` and `PsHPRRotation`, so it is used to allow several users to combine their efforts to control the position and the orientation of an interactive object.

5.4 How to provide different behaviors to a virtual tool

In order to allow proposing different values to control an interactive object, we can associate a different behavior

to the VirtualRay. This behavior must propose values for the outputs corresponding to those of the VirtualRay (the same than for the RayBehavior), and for the outputs corresponding to what is proposed to the simulated object in interaction.

So, it has to define the interactorConnection method (to do the correct initializations) and the computeParameters method (to provide different values).

For example, the FallBehavior class proposes values to the object in interaction by diminishing its altitude.

Writing these C++ lines in the main.cxx file can then create such a new interactor:

```
controller->addInstanceCreator (VirtualRayWithFallBehavior,
    new PsSimpleSimulatedObjectCreator
    <DoubleInteractor
    <VirtualRay<FallBehavior>, PsTranslation, PsHPRRotation> > ());
```

5.5 How to create new interactors

The traditional interaction tools encapsulated by DoubleInteractor can take control of a basic interactive object adapted by a DoubleProtocolTeacher, and also of an object adapted thanks to a DoubleMultipleProtocolTeacher. But in this latter case, if several interactors share the control of an interactive object, they may have difficulties about understanding how the interaction works.

For example, if two classical virtual rays, with ray behaviors, share an interaction upon an object, controlling together its position and orientation, the object's position and orientation will be the average of the positions and orientations proposed by these two rays, so the object will not be stuck to any of these rays, and the users may wonder why their rays do not behave as usual.

It can be interesting to provide to such users a new virtual ray, with a particular 3D rendering to make the users aware of the sharing of interaction upon a shared object: for example a line adjusted to make a link between the position where the shared object is and the position where it should be if this virtual ray was the only one in interaction.

Basically, this new interactor behaves like the DoubleInteractor, so we can easily propose a DoubleMultipleInteractor, which could provide this awareness dedicated to shared interactions.

Thus, the DoubleMultipleInteractor class we propose inherits from the DoubleInteractor class. It creates two new outputs for the absolute values proposed to the object in interaction with, and it creates a new simulated object dedicated to make the user aware of the difference between the values proposed by the interactor and the effective ones of the object in interaction. Here, this object will be a cylinder able to link two 3D positions. This object will have to be created at the time of the control takeover of the interactive object, and to be deleted at the time of the control release. This cylinder will be connected to the new absolute position output of the interactor and to the position of the controlled object. Last, the behavior of the interactor will override the behavior of the DoubleInteractor, in order to compute new values for its new outputs.

Writing these C++ lines in the main.cxx file can then create such a new interactor:

```
controller->addInstanceCreator ("MultipleVirtualRay",
    new PsSimpleSimulatedObjectCreator
    <DoubleMultipleInteractor
    <VirtualRay<RayBehavior>, PsTranslation, PsHPRRotation> > ());
```

To be more generic, this DoubleMultipleInteractor could be parameterized by an object in charge of the creation of the object in charge of providing the awareness for sharing interactions, so that it would be independent from the type of the parameters to control.

6 Conclusion

In this tutorial we have tried to present the main features of VR-OpenMASK from different points of views: from the one of a project manager toward the one of a VR-OpenMASK contributor. Thus, we have seen different levels of detail of the VR-OpenMASK structure, and we hope that the people who will read this tutorial will find inside it the material to better understand, at the level of knowledge they are interested in, the capabilities of our platform.

For further information about OpenMASK, go to the official website: <http://www.openmask.org/>.

If you need an answer while developing an OpenMASK application, consult the OpenMASK's mailing list

archive at <https://www.irisa.fr/wws/info/openmask.users> or ask for information to the OpenMASK developers through the mailing list. You will have to register yourself by selecting the "Subscribe" option, you will be asked for your E-mail address, and you will receive afterwards by mail your member's password.

The following links may be useful to get information about some the OpenMASK's components.

- Official OpenGL Performer home: <http://www.sgi.com/software/performer/>.
- OpenGL Performer manuals: <http://www.sgi.com/software/performer/manuals.html>.
- Open Inventor home: <http://www.sgi.com/software/inventor/>.
- Open Inventor manuals: <http://www.sgi.com/software/inventor/manuals.html>.
- PVM home: http://www.csm.ornl.gov/pvm/pvm_home.html.
- PCCTS home: <http://www.polhode.com/pccts.html>.

Doxygen home: <http://www.doxygen.org/>.

References

- [1] B. Damer, S. Gold, J. de Bruin and D.-J. de Bruin. Conferences and Trade Shows in Inhabited Virtual Worlds: A Case Study of Avatars98 & 99. In Proceedings of the Second International Conference on Virtual Worlds (VW'2000), Springer LNCS/AI, Paris, France, 1-11, 2000.
- [2] C. Carlsson and O. Hagsand. DIVE - a Platform For Multi-user Virtual Environment. Computer and Graphics, pages 663-669, 1993.
- [3] J. De Oliveira and N. Georganas. VELVET: An Adaptive Hybrid Architecture for VEry Large Virtual EnvironmenTs. Proceedings IEEE ICC'2002, New York, May 2002.
- [4] S. Donikian, A. Chauffaut, T. Duval and R. Kulpa. GASP: from Modular Programming to Distributed Execution. Computer Animation'98, IEEE, Philadelphia, USA, pages 79-87, June 1998.
- [5] T. Duval and C. Le Tenier. Interactions 3D Coopératives en environnements virtuels avec OpenMASK pour l'exploitation d'objets techniques. Virtual Concept'2002, pages 116-121, Biarritz, France, October 2002.
- [6] T. Duval and D. Margery. Using GASP for Collaborative Interactions within 3D Virtual Worlds. Proceedings of the Second International Conference on Virtual Worlds (VW'2000), Springer LNCS/AI, pages 65-76, Paris, France, 2000.
- [7] T. Duval and D. Margery. Building Objects and Interactors for Collaborative Interactions with GASP. In Proceedings of the Third International Conference on Collaborative Virtual Environments (CVE'2000), ACM, San Francisco, USA, pages 129-138, 2000.
- [8] C. Greenhalgh and S. Benford. MASSIVE: A Distributed Virtual Reality System Incorporating Spatial Trading. International Conference on Distributed Computing Systems, Vancouver, Canada, 1995.
- [9] M. Kallmann and D. Thalmann. Direct 3D Interaction with Smart Objects. Proceedings of the ACM symposium on Virtual reality software and technology, London, UK, pages 124-130, 1999.
- [10] M. Macedonia, M. Zyda, D. Pratt, P. Barham and S. Zeswitz. NPSNET: a Network Software Architecture for Large Scale Virtual Environments. Presence, Vol3, No.4, pages 265-287, 1994.
- [11] D. Margery, B. Arnaldi, A. Chauffaut, S. Donikian and T. Duval. OpenMASK: Multi-Threaded or Modular Animation and Simulation Kernel or Kit: a General Introduction. VRIC 2002 Proceedings, pages 101-110, June 2002.