Design of a model of human interaction in virtual environments

Javier Carlos Jerónimo[1], Angélica de Antonio[1], Gonzalo Méndez[2], Jaime Ramírez[1]

[1]Universidad Politécnica de Madrid, Campus de Montegancedo s/n, 28660 Boadilla del Monte, Spain. jcjeronimo@alumnos.upm.es, angelica@fi.upm.es, jramirez@fi.upm.es

[2]Universidad Complutense de Madrid. gonzalo@gordini.ls.fi.upm.es

Abstract

In this paper we present a generic model for human interaction in virtual environments and the design of a software component that has been built according to this model for its use in the Maevif platform for Intelligent Virtual Environments for Education and Training. We present the motivations and the main objectives and we detail the key design decisions and the way in which they have been realized in an object-oriented approach based on design patterns. Improved adaptability and extensibility are the main properties of the resulting interaction module.

Introduction

The real world is complex and the approaches to its 3D graphical representation are more and more detailed and accurate, but visual representation is only one part of the problem. Additionally, we need to provide the user with an interaction model which is, just like the graphical representation, as realistic as possible. This interaction model should go beyond basic actions like *move* or *touch*, dealing with the complex interactions that could take place in the real world; those in which a person moves and manipulates objects in a concrete order to achieve a certain goal.
The main objective of our work is the design of a software model of complex human interaction in virtual environments. Some requirements were considered essential for our design:

- **Scalability**: the model must be applicable and easy to use for simple as well as for complex interactions.
- **Extensibility**: the model must allow easily expanding it and making changes in some parts of the design not affecting the others. This is one of the most important things we had in mind when creating the model because it will be probably

expanded several times along its life time (to include new kinds of interactions or devices).

- **Ease of integration**: although the design is focused in a concrete platform that has fostered its design and implementation, it must be abstract enough to be easily integrated into other platforms. The overall goal is to design a generic model for human interaction in virtual environments, and the particular one, to implement and integrate it as a part of the MAEVIF platform.

MAEVIF is a platform for the development of education and training systems based on virtual environments [De Antonio, 2005], [Méndez, 2005]. It is divided into two parts:

- **Agent-based intelligent tutoring subsystem (ABITS):** this is the central part of the platform. It is composed of a group of software agents that collaborate for teaching and training. Each agent is focused in a particular topic of the teaching process: tutoring actions, student tracking, learning objectives...
- **Graphics and interaction subsystem (GIS):** this is the program that runs in the users' terminals. Its purpose is to bring the user a three-dimensional representation of the virtual world and to provide them with the possibility to interact with the environment using the devices available to them.

The agents in ABITS must share and analyze information about the actions that the students perform in the virtual environment. These actions must be assessed and evaluated to determine if the learning objectives are being achieved or not. This knowledge about the actions performed by the students is abstract in the sense that generally it does not matter how the user has effectively executed them (for instance, it may be important to know that the student opened the door, but not if he pressed the handle with his hand or if he pushed the door with his shoulder). Consequently, there is a need for an abstraction process that transforms the interactions that the user performs in the virtual world into the final actions that are important to the teaching process.
There could also be restrictions associated to the way actions can be performed. For instance, there could be actions that require a group of simple interactions to be conducted in a concrete order. Following the previous example, opening the door using the handle is composed of two elemental interactions: *put the hand on the handle* and *press down the handle*. Consequently, actions will be made up of groups of simple interactions with ordering constraints.
Additionally, nowadays there are a great variety of interaction devices that can be used in VR environments, ranging from simple ones such as keyboards, joysticks and mouses, to immersive VR devices, such as data-gloves, haptic devices, tracking systems and caves. The interaction model must not depend on the devices available in a particular execution environment, that is, it should support any kind of peripheral connected to the client's computer to allow the user to perform actions in the virtual world. Using the previous example of *"user A opens door X"*, the movement of his hand can be detected by a tracking system, and then he can press the handle by clicking a button or by doing a gesture with his real hand

(which can be detected and interpreted by a data-glove or by a system of digital video cameras). Therefore, the interaction model should offer another abstraction from the concrete devices being used to the interaction events that are meaningful for the system.

Related work

Trying to separate interaction devices control from VR applications is a problem that has been addressed in quite a lot of publications, such as [Chen, 2002]. However, as in the cited work, it is usual that ad-hoc solutions are designed, so that the same problem has to be solved now and again every time a new system is built.
There are several systems that have addressed the same problem described in this paper. One of the most popular is the VR Juggler suite [Bierbaum, 2001], and more specifically one of its modules called Gadgeteer [Gadgeteer, 2007]. This module acts as a hardware device management system. It contains a dynamically extensible Input Manager that treats devices in terms of abstract concepts such as *positional*, *digital* or *gesture*. Although designed to be modifiable, it is thought to be used together with the rest of the VR Juggler suite, which makes it unsuitable in cases where other VR platforms are preferred.
Other solutions include MR Toolkit [Shaw, 1993] and VRPN [Taylor, 2001]. Both are designed to support the transmission of peripheral data via packets on a network. This way, input devices can be moved to different machines or replaced with different devices (even at runtime) without requiring any changes to the source code of the applications that use them. CIDA [Kelner, 2006] is a plug-in based input devices management platform that aims to abstract the type of the device used. According to the authors, the plug-in mechanism allows an easier addition of new devices.
Other applications, such as the ones described in [Blach, 1998], [Kessler, 2000] also provide a library or a framework to develop device independent VEs. As in the case of VR Juggler, it appears that the whole bundle is needed in order to support different kinds of devices, while in the current work, given its layered structure, the Devices Level can be used separate from the rest.
There is another kind of tools which mainly consist of libraries that are best suited to interact with specific framework, as in the case of CAVELib and CAVE or EQUIP and MASSIVE-3 [Greenhalgh, 2001]. Although they may support a wide variety of devices, the fact that they are mainly crafted to work with a specific platform makes them less suitable for extensive use.
Finally, some approaches make use of unusual interaction devices such us the ones described in [Vinayagamoorthy, 2004], where eye gaze is used as the main interaction technique. Although using an appropriate abstraction it should be easily done, it is still necessary to experiment how this kind of interaction can be integrated.

General design of the Interaction Module

Abstraction layer

Following the requirements and considerations presented in the previous section, our interaction model design provides an abstraction layer made up of three different levels:

- abstract actions that the ABITS can understand (we will call these **operators**). Some example operators are *give something to someone, take something somewhere, activate some object, establish a relationship between two objects...*
- basic interactions (we will call these **behaviors**). Example behaviors are *touch, grab, release, push...* Behaviors can be grouped to define operators.
- interaction devices that can be used to perform the basic interactions. We will call **device** to any process that can collect input from the user. It can be implemented with a physical peripheral or some other software elements like a menu, a voice recognition system, a pseudo-device like a remote administration console, or a collision detector active in the 3D graphics engine. In short, it can be anything that launches events that can be considered as user input.

The separation between the three levels provides an extraordinary flexibility to the system, allowing the developer to reuse existing behaviors and/or operators and making easier the design of new ones. The model also allows us to associate any existing or future device to any behavior. This is fundamental because of the great amount of different peripherals that could be alternatively used in the system. Furthermore, we could use various devices at the same time to generate the same behavior, which could be useful under some circumstances. This design brings us the possibility to expand any of the levels without making changes to the others.

World objects representation

Every object in the virtual world is known as an *entity*. This concept also includes the parts of an object or avatar representing the user: body, arms, head, etc.
An *entity* is any virtual object that has the following characteristics:

- a three-dimensional representation. Commonly known as a *model* in 3D terms.
- a collection of attributes: 3D coordinates and orientation, name, parent and children *entities...*
- behaviors that indicate what can be done with the object, and what the object can do with its environment (other *entities*).

This representation of objects comes from the design of the MAEVIF GIS, and has been adopted because of its power and ability to be easily used and expanded. An *entity* as it is defined in the GIS can be expanded with additional attributes and methods by the modules connected to the GIS (our interaction module is a module of the GIS). Only a few entities will be managed by the interaction module, namely those representing the parts of the user body that he can use to interact with the world: fingers and hands for instance. The connection of operators, via behaviors, to these avatar entities will make it possible for the user to interact with other entities of the virtual world.
We have defined three main software object classes representing the actions executed by the user, in three levels of abstraction: **Operator**, **Behavior**, and **Device**. The additional **Entity** class, besides the properties related to interaction, has information about the behaviors that the entity can launch. Two manager classes will store and keep under control the instances of the relevant classes: **OperatorsManager** and **DevicesManager**.

Adaptability and Extensibility Mechanisms

In order to enforce the adaptability of the interaction module, we need to be able to create different types of instances of these classes (the ones which are relevant for a given virtual environment and hardware setting) and to establish relationships between them with a minimum coupling. We also want to maximize the extensibility of the module, minimizing the effort required to add new behaviors or devices that were not considered in the initial design.
For instance, we want behaviors to maintain relationships with all the devices that can be used to launch it. A specific behavior such as *push* could be launched using a mouse, keyboard, data-glove... We would need to define different subclasses of the **Device** class for different types of devices, but we do not want the behavior to know any details about the creation of device instances nor even be aware of the device types.
The solution adopted is the definition of abstract factories that create the necessary objects based on external definition files provided to them. For example, if we need to create a device object, we only need to read the definition of that device and send it to the proper abstract factory, which in turn will make use of the concrete factory available for the device's subclass. In this way, the only class that will have knowledge about the relevant classes and their relationships is the *configuration manager*.
The *configuration manager* is an essential part of the interaction model. It deals with a lot of configuration settings, most of them detailing the relevant entities and devices in a given virtual environment and hardware setting. But it also allows for an easy extension of the interaction module with new *behaviors* and *device types*. The latter extensions will require, in addition to new configuration settings, the definition of new subclasses as an extension of the base classes in the framework,

and the definition of new concrete factories, but without any modification of the existing code except for the *configuration manager*.

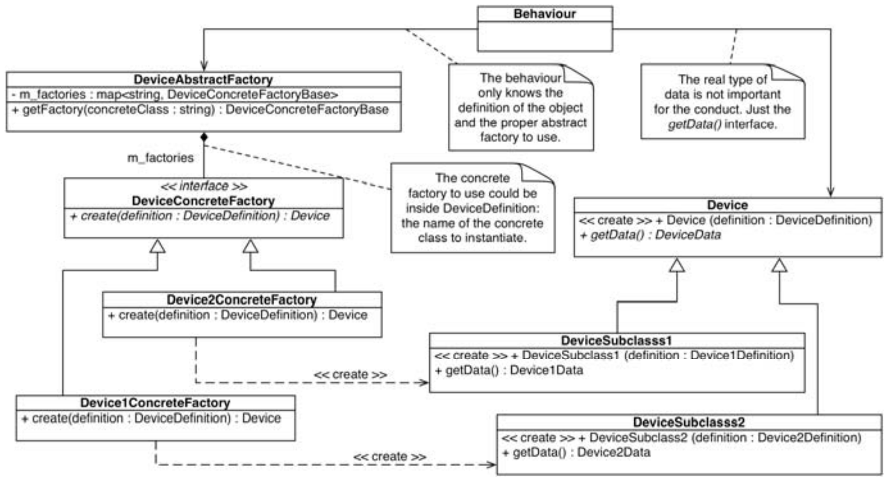
Behaviour
DeviceAbstractFactory
- m_factories : map<string, DeviceConcreteFactoryBase>
+ getFactory(concreteClass : string) : DeviceConcreteFactoryBase
The behaviour only knows the definition of the object and the proper abstract factory to use.
The real type of data is not important for the conduct. Just the getData() interface.
m_factories
<< interface >>
DeviceConcreteFactory
+ create(definition : DeviceDefinition) : Device
The concrete factory to use could be inside DeviceDefinition: the name of the concrete class to instantiate.
Device
<< create >> + Device (definition : DeviceDefinition)
+ getData() : DeviceData
Device2ConcreteFactory
+ create(definition : DeviceDefinition) : Device
DeviceSubclasss1
<< create >> + DeviceSubclass1 (definition : Device1Definition)
+ getData() : Device1Data
<< create >>
Device1ConcreteFactory
+ create(definition : DeviceDefinition) : Device
DeviceSubclasss2
<< create >> + DeviceSubclass2 (definition : Device2Definition)
+ getData() : Device2Data
<< create >>

Fig. 1. Use of abstract factories to create the devices associated to one behavior

The initialization method, responsible for the creation of all the required objects is as follows:

- **registration of subclasses**: the *configuration manager* registers all the subclasses he knows in the proper abstract factories, establishing the relationship with their corresponding concrete factories. New subclasses added in the future will require the modification of this registration process.
- **definitions of objects**: the *configuration manager* creates a definition object for each subclass of the system from the configuration settings it loads.
- **creation of an object**:
  - **asking for a definition**: when one object (for example the behavior *push*) needs to create and establish a relationship with an instance of another class (for example a certain *mouse* device), it asks the *configuration manager* for the corresponding definition.
  - **calling the abstract factory**: the object sends the definition to the proper abstract factory.
  - **delegating to concrete factory**: the abstract factory locates the concrete factory corresponding to the class, as it was previously registered, and sends the definition to it.
  - **creating the object**: Finally, the concrete factory creates the object and returns it back to the object that initially requested it.

Some programming languages permit doing this by using reflection and dynamic loading of new classes, but we have decided to use a more general mechanism that allows us to use any programming language. Note that the current implementation of the module is in C++ which has not reflection support and in which dynamic loading of new symbols at runtime is platform dependent.

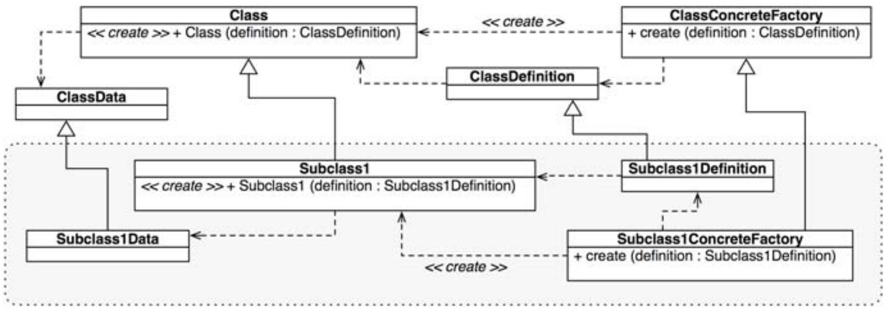
Class
<< create >> + Class (definition : ClassDefinition)
<< create >>
ClassConcreteFactory
+ create (definition : ClassDefinition)
ClassDefinition
ClassData
Subclass1
<< create >> + Subclass1 (definition : Subclass1Definition)
Subclass1Definition
Subclass1Data
Subclass1ConcreteFactory
+ create (definition : Subclass1Definition)
<< create >>

Fig. 2. Example of a class in the design with its own concrete factory, definition and data. Using this structure in the class and registering it in the proper abstract factory (in the *configuration manager* initialization) will make it's creation accessible from any other class of the design.

Devices design

We have defined devices as an abstraction of any software library that can collect user input. This includes peripheral devices, voice recognition libraries and any other software that we want to use to launch behaviors. The main purpose of devices is to return data based on user input. In order for a new device type to be considered by the interaction module, the abstract method *getData()* must be implemented in the new device subclass. The separation between behaviors and devices also requires that all devices return the same class of data. This is achieved by another abstraction: **DeviceData**.
We have defined two steps for the process of obtaining data from the device:

- **poll**: device objects are periodically polled to update its internal state with new data available in the underlying software. The *devices manager* calls the *getData()* method of each device, which is implemented in the concrete subclass.
- **notify**: devices notify associated behaviors only when new data is available.

The first step is executed several times per second (every frame in most of the cases). Most of the device drivers use polling to obtain new data from the peripheral. When the underlying device software library provides notification mechanisms, polling can be ignored. It is possible that in most polls no updated data is available. This is why we have decided to introduce the second step: to prevent the system from overloading behaviors with messages containing redundant data. The second step ensures that only messages containing new data are sent from devices to behaviors.

To achieve this goal, a device makes two uses of the observer pattern: first, as an observer in polls; and second, if there is new data available in any poll, as a subject to notify the registered behaviors.

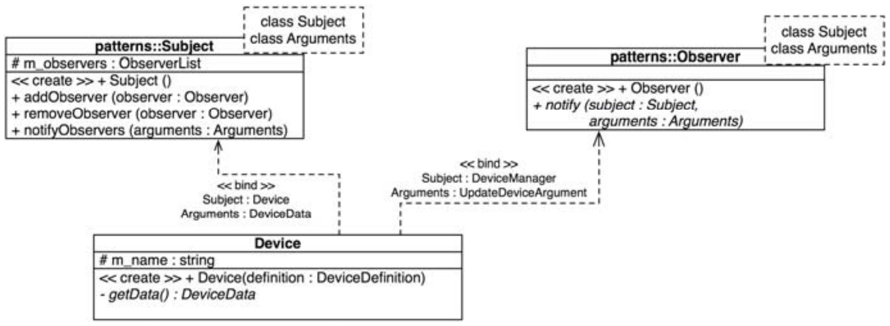
class Subject
class Arguments
patterns::Subject
# m_observers : ObserverList
<< create >> + Subject ()
+ addObserver (observer : Observer)
+ removeObserver (observer : Observer)
+ notifyObservers (arguments : Arguments)
class Subject
class Arguments
patterns::Observer
<< create >> + Observer ()
+ notify (subject : Subject, arguments : Arguments)
<< bind >>
Subject : Device
Arguments : DeviceData
<< bind >>
Subject : DeviceManager
Arguments : UpdateDeviceArgument
Device
# m_name : string
<< create >> + Device(definition : DeviceDefinition)
- getData() : DeviceData

Fig. 3. Use of the observer design pattern by the *device* class. There is also an example of a special device that notifies about collisions between entities detected in the 3D engine.

Behaviors design

A behavior is any basic interaction that the user is allowed to perform in the virtual environment. A group of behaviors, in a concrete order, can launch an operator and so let the simulation system know what the user is doing in the virtual environment.
As explained in previous sections, a behavior can have a group of devices associated to it. This provides us the possibility to launch it in various different ways. The problem here is that some devices will return a vector of real numbers, others a pair of entities (for instance a pseudo-device that detects collisions in the 3D engine), or even a sentence uttered by the user and recognized by a voice software library.
When a device notifies a behavior that it has new data and sends the data, the processing of the data to determine if the launching requirements for the behavior are met would require the behavior to have knowledge of the type of data and the way to process it, adding a dependency of the behavior on the device. Another option would be having the device itself make that decision, and not simply returning the data, but then we would have the same problem reversed: a dependency of the device on the behavior; if we created a new behavior, we should modify the existing device to introduce the knowledge related to the new behavior and its activation.
To avoid any dependency we have introduced an external *evaluator* class. There will be an evaluator specialized in each behavior-device association. For example, let's suppose we have a *touch* behavior and a *device* for joysticks that returns a float. When the user moves the joystick axis the position is represented in the in-

terval [-1.0..1.0] and when the user pushes a joystick button the data shows if it is pressed or not (-1.0 not pressed, 1.0 pressed). Let's suppose that we already have an *evaluator* that processes this kind of data and determines whether or not the behavior must be launched. Now we introduce a voice recognition library as a device, returning strings like *touch object*. It's obvious that the existing *evaluator* does not work with the new kind of data, but the only thing we must do is to create a new *evaluator* for each behavior we want to be activated by the new device.
With this design we keep devices and behaviors clearly separated, allowing new additions to the system with the minimum changes to existing classes.

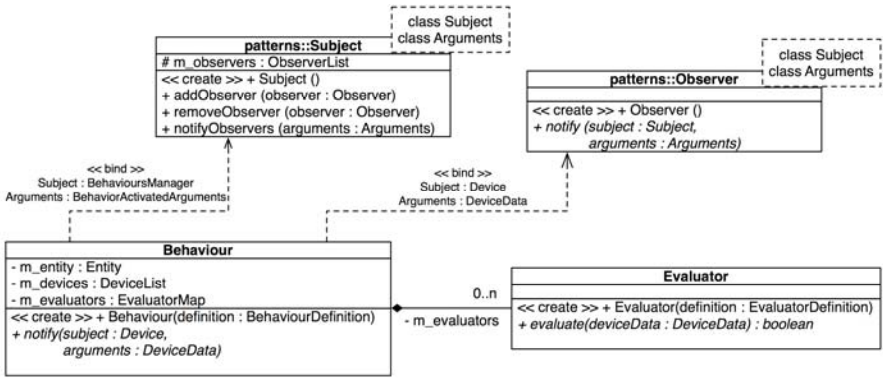
class Subject
class Arguments
patterns::Subject
# m_observers : ObserverList
<< create >> + Subject ()
+ addObserver (observer : Observer)
+ removeObserver (observer : Observer)
+ notifyObservers (arguments : Arguments)
class Subject
class Arguments
patterns::Observer
<< create >> + Observer ()
+ notify (subject : Subject, arguments : Arguments)
<< bind >>
Subject : BehavioursManager
Arguments : BehaviorActivatedArguments
<< bind >>
Subject : Device
Arguments : DeviceData
Behaviour
- m_entity : Entity
- m_devices : DeviceList
- m_evaluators : EvaluatorMap
<< create >> + Behaviour(definition : BehaviourDefinition)
+ notify(subject : Device, arguments : DeviceData)
0..n
- m_evaluators
Evaluator
<< create >> + Evaluator(definition : EvaluatorDefinition)
+ evaluate(deviceData : DeviceData) : boolean

Fig. 4. Behavior class with its two uses of the observer design pattern: it receives events from the devices associated, and it sends events to the behavior manager.

Operators design

An **operator** represents an abstract interaction of the user with the virtual environment, such as *take something* or *give something to someone*. These operators are independent from initiator and targets, as the action is always the same but with different arguments. Therefore, there is only one instance of each operator class. Furthermore, an operator can be made of several behaviors executed in a concrete order. Thus, we have introduced a concept similar to the *evaluator* that here is called **detector**. A *detector* is an external class that the operator uses to know when it has been activated. The detector will process behaviors executed by the user and it will launch the operator when the requirements are meet.
We have designed *detector* as an abstract class to avoid restricting its internal design. Different types of detectors can be implemented if the existing ones are too not enough for new operators to be created. For a *detector* to do its job, it is required to detect the creation of new behaviors. Moreover, whenever a behavior is executed by the user, all the detectors depending on it must know about this event. These two situations have been solved with two applications of the observer pat-

tern (see the figure 5}): first, when a behavior's concrete factory creates a new object it notifies the behaviors manager so that it can notify this event to all the detectors; second, after a behavior has been created and the detectors are notified, if the behavior is relevant to the operator's requirements, it will be observed so that when it is launched by the user, the detector is notified of this event.

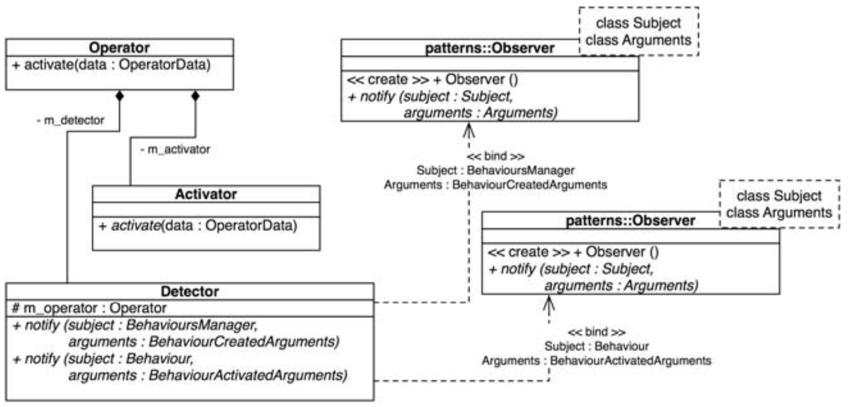
Operator
+ activate(data : OperatorData)
- m_detector
- m_activator
Activator
+ activate(data : OperatorData)
Detector
# m_operator : Operator
+ notify (subject : BehavioursManager, arguments : BehaviourCreatedArguments)
+ notify (subject : Behaviour, arguments : BehaviourActivatedArguments)
patterns::Observer
class Subject
class Arguments
<< create >> + Observer ()
+ notify (subject : Subject, arguments : Arguments)
<< bind >>
Subject : BehavioursManager
Arguments : BehaviourCreatedArguments
patterns::Observer
class Subject
class Arguments
<< create >> + Observer ()
+ notify (subject : Subject, arguments : Arguments)
<< bind >>
Subject : Behaviour
Arguments : BehaviourActivatedArguments

Fig. 5. Operator design with its detector and activator

Our detector class implements a finite state automaton that tracks the behaviors launched by the user and determines when the requirements of the operator are met. Figure 6 shows an example: the *take something* operator's detector has been implemented as a finite state automaton with three states. The requirement to pass from the initial state to the intermediate one is the *touch object* behavior. Then, the transition from this intermediate state to the final one is allowed by the *take* behavior (with the meaning of *close hand*). In the final state we can be sure that the user wants to launch the operator *take something* as he has touched an object and closed his hand (in this order).

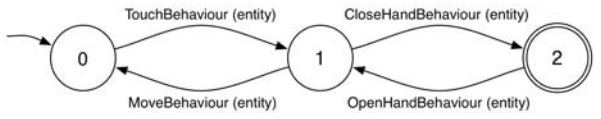

Fig. 6. Operator's detector designed as a finite state automaton.

This design is very simple and allows us to completely define the specific automata in the configuration file. For more complex operators, if automata are not powerful enough, it will be possible and easy to define a new kind of detector (for instance based on a Petri net) directly usable by the system.
We have created an automata template library in such a way that it is easy for the user to completely define a new automaton just by using configuration settings (the system will create dynamically the automata at runtime). The assignment of a detector object to a particular operator is done at runtime.

The activation of the operator by the detector when the activation requirements are met is realized by a message from the detector to the operator containing all the arguments related to the abstract action that the operator represents. For example, in the *take something* operator this message must contain the actor and the object. In the MAEVIF platform, the process of activating an operator implies the creation of a JSON message which is sent to the ABITS, but we anticipate that in other uses of this interaction module the consequence might be different. This led us to design the operator in such a way that it uses an external class to do the real activation of the operator. In our case, this activation object will be a network client that will marshal the operator's activation arguments in a JSON string that will be sent across the network to the agents platform. Figure 1.7 summarizes the data flow involved in the process of collecting user input data and analyzing it.

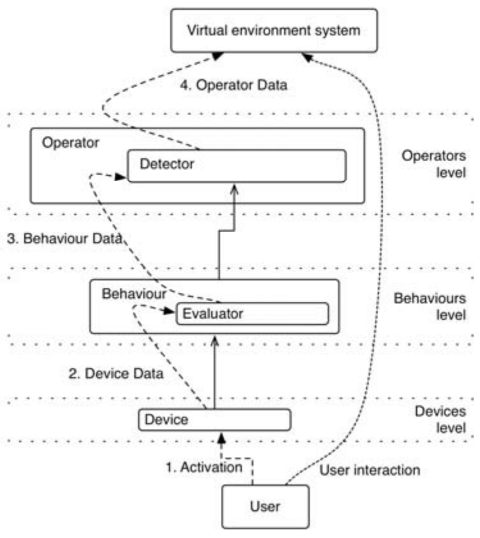

Fig. 7. When a device has new data based on user input it activates the behaviors associated and they process the data using the corresponding evaluator object. If the device data meets the requirements for activating the user behavior, the data flow will reach the top level of operators in which there will be more processing to determine whether or not an operator must be activated.

Conclusions

Our main goal when designing our interaction model was to create a powerful but at the same time simple design, and an easily extensible module. This paper presents the resulting design detailing the rationale for the most important design decisions. The flexibility of the designed interaction module allows for an easy generation of alternative user interfaces, based on different combinations of inte-

raction devices, and perform tests to select the most appropriate one(s) for the task(s). Together with adaptability and extensibility, performance is a critical factor that should not be forgotten. Our design of the interaction module is meant to perform real-time processing of interaction data in virtual environments. The most intensive processing will be required by the user movement behavior evaluators, because the user will probably update those peripherals every frame. Other behaviors will be associated to asynchronous events too, but those events (*take*, *push*, *touch*) will be less frequent than the previous ones. In order to improve performance we have minimized the number of messages between objects.
The designed module has already been implemented and some preliminary tests are being conducted, with satisfactory results regarding real-time performance and memory use, but extensive testing is still required. Moreover, this interaction module has been integrated in the MAEVIF platform, and we are currently working in the development of several test virtual environments to explore the possibilities and constraints of the module. There is also the need for additional specializations of the design targeted to different types of interaction devices. The customization of the model for its use in a specific system involves just declaring the set of available devices, and defining the desired behaviors and operators, in a first level of abstraction, and then configuring the settings of the devices for each specific user terminal.

References

de Antonio, A., Ramírez, J., Méndez, G.: Developing Future Interactive Systems, chap. VIII,pp. 212–233. Idea Group Publishing (2005)

Méndez, G., de Antonio, A.: Training agents: An architecture for reusability. LNAI, vol. 3661, pp. 1–14. Springer-Verlag (2005)

Bierbaum, A., Just, C., Hartling, P., Meinert, K., Baker, A., Cruz-Neira, C.: Vr juggler: A virtual platform for virtual reality application development. IEEE Virtual Reality. (2001)

Blach, R., Landauer, J., Rsch, A., Simon, A.: A highly flexible virtual reality system. Future Generation Computer Systems Special Issue on Virtual Environments (1998)

Chen, J.: A virtual environment system for the comparative study of dome and hmd. Master's thesis, Department of Computer Science, University of Houston (2002)

Greenhalgh, C., Izadi, S., Rodden, T., Benford, S.: The equip platform: bringing together physical and virtual worlds. Tech. rep., University of Nottingham - UK (2001)

Gadgeteer: Device Driver Authoring Guide. Iowa State University: (2007)

Kelner, J., Teichrieb, V., Farias, T., Rodrigues, C., Pessoa, S., Teixeira, J., Costa, N.: Cida: an interaction devices management platform. In: Proceedings SVR 2006, pp. 271–284 (2006)

Kessler, G.D., Bowman, D.A., Hodges, L.F.: The simple virtual environment library: An extensible framework for building ve applications. Presence 9(2), 187–208 (2000)

Shaw, C., Green, M., Liang, J., Sun, Y.: Decoupled simulation in virtual reality with the mr toolkit. In: ACM TOIS, pp. 287–317. ACM Press (1993)

Taylor, R., Hudson, T., Seeger, A., Weber, H., Juliano, J., Helser, A.: Vrpn: a device independent, network-transparent vr peripheral system. In: ACM VRST, pp. 55–61 (2001)

Vinayagamoorthy, V., Garau, M., Steed, A., Slater, M.: An eye gaze model for dyadic interaction in an immersive virtual environment: Practice and experience. Computer Graphics 23(1),1–11 (2004)