Rapport de stage

Metaphors of constrained interactions in 3D universe for cooperative work

Ting Yun LU
DEA Informatique

Responsible: Thierry DUVAL
February – June 2003
## Contents

### INTRODUCTION

1 INTERACTIONS IN CVEs
   1.1 User-to-user .......................................................... 4
   1.2 User-to-object ......................................................... 4
   1.3 Metaphors: ................................................................. 5
      1.3.1 Physical devices ................................................. 6
      1.3.2 Virtual Metaphors ............................................... 6
   1.4 Awareness ............................................................... 7

2 THE DESIGNS OF METAPHOR AND EXPERIMENTS .......................... 10
   2.1 Principles and tradeoffs ............................................. 10
   2.2 Ways to test the metaphors ........................................ 11

3 INTRODUCTION TO OPENMASK ............................................. 13
   3.1 OpenMASK framework overview ..................................... 13
      3.1.1 Basic distribution paradigm ................................ 13
      3.1.2 3D Visualization .................................................. 14
   3.2 Interaction in OpenMASK ............................................. 14
      3.2.1 Simulated Object .................................................. 15
      3.2.2 Interactor ............................................................ 15
      3.2.3 Adaptor ............................................................. 16

4 CONSTRAINT INDICATOR .................................................. 18
   4.1 Basic idea: ............................................................... 18
      4.1.1 User’s view: ......................................................... 18
      4.1.2 Calling view: ...................................................... 19
      4.1.3 Module view: ....................................................... 20
      4.1.4 Object’s view: ..................................................... 21
   4.2 Implementation: ......................................................... 21
      4.2.1 ConstraintIndicator .............................................. 22
      4.2.2 OmvDoubleShiftedPO ............................................ 23
      4.2.3 OmvTranslatoinIndicativeTab / OmvRotationIndicativeTab ... 24
      4.2.4 ProtocolTeacherPlus .......................................... 25
      4.2.5 Geometrical design: ............................................. 26
   4.3 Result and test: ......................................................... 27
      4.3.1 Moment of creation and life time: .............................. 27
      4.3.2 Dependency: ........................................................ 27
      4.3.3 Test Results: ...................................................... 28
4.4 Future work

4.4.1 Liberty for the user

4.4.2 Advanced capability

4.4.3 Combine with further system

5 MULTI-USER WITH CONSTRAINTS

5.1 Basic idea

5.1.1 ConstraintDefaultAdaptorOffset

5.1.2 DoubleMultipleConnector

5.2 Implementation

5.2.1 DoubleMultipleConstraintConnector

5.2.2 DoubleMultipleProtocolTeacherPlus

5.3 Result and test

5.4 Future work

CONCLUSION
INTRODUCTION

Recently, the technology of computer science allows people to immerse in the three-dimensional (3D) world, which is totally digital. That means, what a user perceives is actually generated by the computers, he can see and feel different effects that the system has supported, and he can interact with the environment or the other users who share this virtual world. For example: pick up a stone and discuss with someone about whether to throw it or not.

Among all these research, more and more attentions have been paid to the design in scenarios involving multiple users – so called Collaborative Virtual Environments (CVEs) designed to support real time work and interaction with objects and artefacts. Here we are interested in the domain of cooperative manipulation that refers to the simultaneous manipulation of a virtual object by multiple users in an immersive virtual environment.

In the virtual world, like in the real world, we use devices to interact with objects, as to manipulate them. We will distinguish physical devices like mouse, data gloves, from metaphors that are higher level virtual device that are embedded in real devices, for example, virtual hand and ray-casting.

Our work is focused on designing new metaphors of interactions that are all mostly convenient for manipulating the constrained objects (for example objects with some mechanical characteristics) during the cooperative manipulation process. In other words, to develop metaphors which represent clearly the constraints on the object to provide the necessary comprehension for users of how to interact with them.

In the first chapter I will simply describe the techniques which support interactions in CVEs, including two essential types of interaction: user-to-user and user-to-object (same with object manipulation), moreover, the metaphors and awareness tools as well, which are used to simplify the cooperative work. From these techniques, we obtain some principal ideas of the design of metaphors and the ways to experiment on them; these will be presented in chapter 2. To go on, an overview and some important aspects of OpenMASK will be introduced in chapter 3, since all projects of this training period (stage) are built using this platform as development environment. After that, we will enter the main subject of this stage: the complete process for producing the new metaphor will be represented in detail from the original ideas, the design manner, to the implementations and the test steps. Finally, a chapter used to summarize all we have gone through will stand as a conclusion.
Chapter 1

INTERACTIONS IN CVEs

The use of collaborative virtual environments (CVEs) has become more and more popular due to the cheaper, faster and more reliable facilities provided by personal computer systems and network resources. Much CVE research is devoted to the development of support tools and the minimization of network traffic. Some examples include AVOCADO [15], Bamboo [16], DIVE [14], ARéVi [17], and MASSIVE [13].

This chapter outlines some works brought forth in the domain of interactions, which consists of two main areas: user-to-user and user-to-object. Then we point to the subject of metaphors and awareness, two important parts to make cooperative work easier and more efficient.

1.1 User-to-user

Interaction with other users involves mainly computer-mediated communication and interpersonal actions. Researchers in this area try to enhance and evaluate the communication between users. The communicative aspects include speech that can be supported with various forms of non-verbal communication. The interpersonal actions are targeted at the avatar, or player character, of the other user.

The non-verbal communication aspects of CVEs have been studied, for example, in the context of user embodiment [20], communicative behaviors [21], and realistically expressing avatars [22]. Each of these approaches can be seen as a potential solution to interpersonal interaction.

Furthermore, Viullême and Thalmann [18] describe a system based on the VLNET framework in which the user can select a gesture and a facial expression from a set of options presented on a screen. After the selection, the choices are incorporated into an avatar that represents the user inside the CVE. In Spin [19] the aim is to create a kind of “conference table”. This is built on a computer screen as a set of panels placed side by side around a circular table. The panels can be rotated as if they were around the user’s head. To each panel, a user can associate another user or an application. To select one application to be executed or other user to talk with, one has simply to rotate the panels until the desired choice is in the middle of the screen. Other research has addressed the evaluation of user-to-user communication.

1.2 User-to-object (object manipulation)

Collaborative augmented reality (AR) systems [23] often include object manipulation. In such systems the users are physically located in the same space and are able to see each other using see-through glasses. Virtual objects are superimposed on real-world objects. This setting can provide the same type of collaborative information...
that people have in face-to-face interaction such as communication by object manipulation and gesture. Such a setup has been used in games like AR2 Hockey [24], in scientific visualization systems (Studierstube), in discussion support systems (Shared Space; Virtual Round Table) and in object modelers like SeamlessDesign [25]. None of these systems, however, allow cooperative manipulation.

Although some research addresses interaction in CVEs, in most of them cooperative manipulation is not possible. Usually, when one user selects an object for manipulation, the other cannot participate in the same procedure, for example, in MASSIVE [13], when user A selects an object, user B can just see what happens to the object without interacting with it. In fact, most existing research specifically forbids this simultaneity. In the work of Li et al. [26], many users can manipulate the same object at the same time, but the object must be modeled with NURBS surfaces and when one user selects the object he actually gets exclusive access to the shape, position and orientation of only one patch. The ICOME system [27], a geometric modeling framework, organizes the object in a hierarchical way allowing users to act simultaneously on different hierarchical structural levels of the same object.

However, there are some examples of actual cooperative manipulation in VEs. Noma [28] presents a study of cooperative manipulation where two users manipulate an object using force feedback device. These devices are used to constrain a user’s hand movements by simulating the forces they would feel based on the partner’s actions. The other is the platform - OpenMASK [3], which is the new generation of GASP (General Animation and Simulation Platform) [1][2] developed in IRISA by the SIAMES team. In this work the users can move an object that is controlled by a simulator. This simulator, replicated on each node, is able to receive simultaneous movement commands, combine them, and generate the resultant movement. I will explain this system in detail later in chapter 3. Also, Marcio [10] has presented a framework, which is based on the Collaborative Metaphor concept that allows the combination of multiple manipulation techniques. It combines existing “magic” interaction techniques instead of combining user movements based on physical laws, for example, user A controls the object’s position using ray-casting technique, while user B rotates it with virtual hand (see figure 1.1).

![figure 1.1 - combination of two metaphors](image)

### 1.3 Metaphors

Metaphors are used as virtual input devices that are controlled by users in order to interact, mostly with the objects, in CVEs. Usually, computers catch the state of a metaphor continuously for “knowing” the user’s action so as to produce the responses relatively. For example, when moving a mouse, the computer detects this movement, so it changes the position or the form of the cursor on the screen in the same way of how the mouse moves.
1.3.1 Physical devices

Many kinds of devices have been designed, such as: keyboard or mouse (what we use ordinarily), joystick or gamepad (operating the same way with the game play), data glove (which provides a more intuitive way to manipulate objects, each finger’s action can be distinguished and detected as well as the position and orientation of the glove)...etc. Furthermore, Bernd [9] has announced a new device, Cubic Mouse (Figure 1.2), which controls the position and orientation of a virtual model and the rods move three orthogonal cutting or slicing planes through the model. Since coordinate systems are an essential building block for most computer graphics applications, the Cubic Mouse literally puts an application or object coordinate system in the user’s hand. The rods provide a strong affordance and make the device obvious to use. Turning one of the rods specifies a rotation of a virtual object around the appropriate axis of a given reference frame represented by the Cubic Mouse. In contrast to a standard six degrees of freedom (DOF) tracker, it is possible here to specify the 6 DOF independently from each other. Currently, the device is mostly used as an absolute positioning system.

figure 1.2 - Cubic Mouse

However, to make interactions in CVEs more real, we better use devices that support force-feedback. In other words, users can feel the impacts of their own actions when manipulating objects, for instance, using a data glove to push a door, which is actually locked, the user may feel a resistance on his palm to let him understand what has happened to that door. Unfortunately, such devices with force-feedback are not often used, since they do not always fit well with all kinds of situations, moreover, they are not available to all people. For this reason, we have to propose another way to let users become more aware of their own actions in CVEs.

1.3.2 Virtual Metaphors

In last paragraph, we have talked about real metaphors that work to simulate the object manipulation to be similar to the reality. Of course this conforms to our original expectation – letting users work in the same way as in the real world. However, we must know that, in fact, the virtual world does not completely resemble to the real world. Namely, there are some characteristics of VEs, which are totally different from our world, for example, the physical laws. This problem of inconsistency, even so, could be utilized as an advantage - we can handle objects in a different manner from being in the real world: manipulating them from far away, or suspending them in the air...etc. In order to deal with this environment and making the best use of it, we have to think differently.

According to the idea above, many virtual metaphors have been developed and already used frequently in manipulating objects in CVEs. Virtual hand, for example,
works as a real hand which can pick up an object, move it, then release it. The only different thing from the real hand is – we can change the form of this virtual hand arbitrary depending on various conditions, for example, extending the index finger unlimited to point to something very far away. Ray-casting technique, like virtual hand, is also often used as an ideal tool to point and select objects precisely. Nevertheless, these kinds of metaphors usually well functioned only in the case when there is just one user who interacts with the object, that is to say, if more than one user manipulate the same object simultaneously, problems occur. To give an example, imagine two users picking up an object in the same time with their virtual hands and wanting to move the object in opposite directions, then a conflict comes up. Because each of them expects the object to stay on their hand and follow their track, however, it is obviously impossible. As the result, both of them get confused for not understanding why the object didn’t move or move in another way.

Since force-feedback devices have not been used commonly, Fenals [4] has proposed a solution, which is an improved version of ray-casting technique, in order to deal with this problem when more than one user work cooperatively on one object. This new ray, unlike previous ray, is not only elastic but also deformable. It can lengthen and shorten according to the distance between the user and the object as well as change its shape in proportion to the intensity of the movement. Now we take two users who are trying to pull one object in the same time (Figure 1.3), which shows how this ray works to inform users of the constraint that keeps the object from following the expected trajectory. One end of the ray is connected with the user (displayed as a cone in this simple illustration) while the other end is always attached on the object, the more we drag or pull the object, the more the ray will become deformed. As we can see, the original rays have been conserved to give a clear idea about the deformation of the ray. This new ray-casting technique has took an important step as the purpose of designing a metaphor which shows constraints explicitly to the users in order to let them understand why the object does not behave as it should. However, there are still much work to do either to improve existing metaphors or to design new metaphors that can ease the awareness of such collaborative interactions for the users.

![Figure 1.3 - The Deformable Ray](image)

### 1.4 Awareness

Awareness of the cooperative interaction in CVEs plays an important part to provide necessary information to users about other users’ actions and the characteristics of objects. This section shows some existing problems of awareness in CVEs and methods used to try to solve them.

Many studies have shown that participants not only use objects and artefacts, such as documents, plans, and models, to accomplish their various activities, but to coordinate those activities, in real time, with the conduct of others. Indeed, it is found that many
activities within co-located working environments rely upon the participants talking with each other, and monitoring each others’ conduct, while looking, both alone or together, at some workplace artefacts. An essential part of this process is the individual’s ability to refer to particular objects, and have another sees in a particular way what they themselves are looking at. To investigate this subject, Hindmarsh and al. [6][7] have created an experiment in which participants were asked to collaboratively arrange the lay-out of furniture in a virtual room and agree upon a single design. As a result, the more important observations include:

1) **Problems due to “fragmented” views of embodiments in relation to share objects.**
   This is caused by users’ narrow field of view - it is difficult to simultaneously view the source and target of actions such as pointing, so participants are provided with an impoverished sense of action. They cannot make sense of talk and activity without seeing (and seeking) the embodiment in relation to relevant features of the environment.

2) **Difficulties in understanding others’ situations.**
   Participants faced problems assessing and monitoring the perspectives, orientations and activities of the other(s) even when the other’s embodiment was in view. This reduces opportunities for participants to design and coordinate their actions for others, since awareness of the actions and orientations of the other is significantly undermined.

Some proposals have been brought up relative to the problems above, which contain two following main methods:

1) **Increasing field of view with peripheral lenses**
   Peripheral lenses consist of two windows, which render views on a virtual environment to the left and right of the main view, with increased distortion allowing more visual information to be displayed within a smaller horizontal space (figure 1.4). Also, a “Peripheral glancing” function is provided to let users switch quickly between left or right peripheral view (figure 1.5). Hence, the use of this semi-distorted view has not only increased a user’s effective field-of view, but also supported peripheral awareness.

   ![figure 1.4 - Peripheral lenses](image)
   ![figure 1.5 - Peripheral glancing](image)

2) **More explicit representations of actions**
   It is known that the source embodiment and target object are unlikely to be simultaneously in view, so actions could be represented on the source, target and also in the surrounding environment. For example, when pointing at an object in distance, aside from raising an arm of the embodiment, we may also extend this arm up to the object pointed so that the others users can find more easily that this object has been pointed by seeing an arm attaching on it. To provide better information about others’ perspectives, the embodiment’s view was made visible as a semitransparent frustum; therefore, the other participant’s view is visible (figure 1.6). Another attempt has been made by Rouillé [5], he declaimed a
mobile camera that users can change their point of view arbitrary. Hence, users can have not only his own subjective perspective, but also other kinds of view, for example, his partner’s perspective. However, peripheral awareness need not be associated simply with visible resources. That is to say, audio signals could equally be used to make manifest certain actions.

On the other hand, the characteristics of an object have to be stated in order to enable users to understand well the way to interact with it. Namely, when manipulating an object, the more the users know the actions available on the object, the more efficient and easier the work will be done. Characteristics of an object, in fact, cover a wide range of aspects; for example, physical or chemical features. However, what is the most interesting at the moment is the mechanical relationship with other objects or with the environment.

The objective of mini-project of Champalaune and Guichard [29] was to enhance awareness when one or more than one users interact with an object on platform OpenMASK. Objects, in this project, are assumed to act in two possible ways: move and rotate, which are represented with the semi-transparent sphere and cube respectively. Equally, the colors of objects differ based on diverse interaction situations. Table 1.1 summarizes the representations, which are made to improve awareness of a selected object.

<table>
<thead>
<tr>
<th>Rule</th>
<th>Rotate</th>
<th>Move</th>
<th>Manipulated (single-user mode)</th>
<th>Manipulated by others (single-user mode)</th>
<th>Manipulated (multiple-user mode)</th>
<th>Manipulated by others (multiple-user mode)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Representation</td>
<td>sphere</td>
<td>cube</td>
<td>green</td>
<td>red</td>
<td>dark blue</td>
<td>White</td>
</tr>
</tbody>
</table>

Moreover, in multiple-user mode, the name of the user who controls the object, as the number of users manipulating the same object, can be chosen to be displayed according to one’s need (figure 1.7).

As we can see, this approach works well to inform users about the possibilities to move or rotate an object, whether there are another users interacting with the same object, and furthermore, the names and amount of the other users. Nevertheless, it’s obviously insufficient for tasks that are much more complex, for example, constrained objects, which only can be move along with x-axis or rotated 30° counterclockwise around z-axis, are impossible to be expressed clearly here. Thus, much effort has to be taken to provide more precise information to users of the ways to interact in CVEs.
Chapter 2

DESIGNS OF METAPHOR AND EXPERIMENTS

In the preceding chapter, we have seen many methods and metaphors that have been designed for supporting cooperative interaction with objects in CVEs. However, only few of them are able to provide good enough assistance to make users close to fully immersion within virtual worlds, that is to say until present time, most designs possesses more or less defects, at technological or designing aspects, as will be discussed in this chapter. In addition, we also try to analyze some existing examples and generalize principles of the work of design. In the last section, some ways to experiment these works will be outlined, since it’s the only way to determine the usability and the stability of a metaphor.

2.1 Principles and tradeoffs

An ideal metaphor should conform to these two essential requirements:

1) **Be easy to use**
   A metaphor must not be too complex to use so that users can quickly be familiar with it, and so that they will be able to concentrate on their work without caring about their tool or spending redundant effort on it.

2) **Provide sufficient awareness to users**
   The states of the object must be shown to inform users about the constraints on this object, the possible actions that can be taken, and whether other users are interacting with this object at the moment. Therefore, users are capable of organizing and designing their manipulation for the work.

To design a metaphor, detailed investigation must be made in advance to obtain necessary information, which includes: environmental properties, characteristics of the work, possible conditions during the work, and the kinds of users. However, while dealing with the virtual world, the case is much more complex, since people have become used to the constraints of physical workspaces, and work practices have involved taking advantage of them. Virtual workspaces, in contrast, do not have a fixed set of constraints, thus designers must make explicit decisions about every aspect of the environment, from the way that things will look to the ways that users will interact with them.

For getting a more precise idea, Gutwin and Greenberg [8] have brought up three areas in particular where virtual workspaces differ from physical counterpart – limited field of view, no physical constraints, and the possible various ways allowed for the representation, which also remind us about the importance of the awareness. As mentioned before, the narrow field of view causes the fragmented view which may separate one embodiment from its manipulated object; while in the physical world, people interact with objects by directly manipulating them, which provides others with a great deal of information about the nature and progress of the activity. A good method has been
used to ease this problem systematically [7], a design table (2.1) has been used in order to highlight potential problems when trying to represent others’ action more explicitly:

<table>
<thead>
<tr>
<th>Action</th>
<th>Targets</th>
<th>Represented on body?</th>
<th>Represented on targets?</th>
<th>Represented in environment?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Look</td>
<td>region</td>
<td>Orientation</td>
<td>none</td>
<td>None</td>
</tr>
<tr>
<td>Speak</td>
<td>region</td>
<td>None</td>
<td>none</td>
<td>Voice</td>
</tr>
<tr>
<td>Point</td>
<td>object or region</td>
<td>raise arm</td>
<td>none</td>
<td>None</td>
</tr>
<tr>
<td>Grasp</td>
<td>object</td>
<td>raise arm</td>
<td>may be seen to move</td>
<td>connecting line</td>
</tr>
</tbody>
</table>

*Table 2.1 – representation of actions*

As we can see, this approach involves the following steps: 1) identify all possible actions 2) identify the targets of each action 3) determine whether and how each action is represented on the source embodiment, target object(s), and in the surrounding environment. Also, make sure that these representations are consistent and distinguishable. This design table can also be extended to adapt to other design works, for example, representations of the actions available on the object - the same reason that in CVEs, people manipulate objects indirectly, therefore it’s is hard for users to understand the constraints of objects. In order to ensure the distinguishability of the representation of the actions available, it is suggested to utilize the visual characteristic of the virtual environments – we can change the colors, textures, or forms… and so on, arbitrarily of an object or a metaphor. Even more, pop-up menus, sounds, and animations are probably the suitable choices.

However, tradeoff issues are worth to be considered. Since users of CVEs act both as individuals and as members of a group, and the requirements of individuals and groups often conflict. In most cases, individuals demand powerful and flexible means for interacting with the workspace and its artifacts, while groups require information about each other to maintain awareness. In [8], some aspects of tradeoff have been addressed with techniques that try to support the needs of both individuals and groups; for example, animation is used to indicate that an action has been proceeded under symbolic manipulation technique mode (buttons, shortcuts…etc.). As a result, individual is provided a powerful and flexible ways to act while their feedback is also conserved to maintain awareness. Even so, it is found that avoiding the tradeoff at one level often just moves it to a different level, although one that is more manageable, in other words, to remove the tradeoff entirely is very hard to achieve. As long as it is now at least possible to form a compromise, which can be based on the specific needs of the task, the key point is to know what kinds of awareness information are important to the group activity, and how much detail about others’ actions is needed to maintain awareness.

### 2.2 Ways to test the metaphors

Up to now, we have discussed some principal ideas of designing a practical metaphor, but to verify whether or not this metaphor is really suitable for use, series of tests must be done to evaluate its usability and stability. The aim of the experiment is to explore and uncover the kinds of interactive phenomena, practices, and problems that may be of particular relevance to both users and designers.

Consequently, three major aspects should be taken into account carefully:

1) **Design of the experiment**

The design depends on the desired purpose, for example, whether the metaphor is capable to help users in certain aspect. Roughly, all possible conditions of the
work and all functions supported by the metaphor should be classified to cover most situations that would happen, moreover, it is suggested that to test them in the crossed and overlapped ways in order to obtain maximal potential results.

2) **Choice of the testers**
Testers are in strong relation to the objectivity of the experiment. In general, the testers better involve with a broad mixture of people, such as single and multiple participants, males and females, novices and experts, for the reason to derive various outcomes for comparison.

3) **Data collection**
A complete data collection will include not only keeping track of users’ actions and conditions when proceeding the works, for example, using video cameras to record each tester’s on-screen activities and audio from their perspective, but also an interview of the users about their impressions and experiments after the work in CVEs. Of course, the contents of interviews are needed to be considered in detail, such as the questions to ask.

As a matter of fact, for the most part, the design of experiments is beyond the scope of our design work, often, experts of this field will be responsible for it. However, it is undoubted that experiments play an important part in the evolution of the design of metaphors, that is to say, the test-modify-test cycle. A well-designed experiment will always lead us to find out potential possibilities to improve the metaphor, in fact, best metaphors mostly come into existence through the steps of evolution.
Chapter 3

INTRODUCTION TO OPENMASK

Since this project is built under OpenMASK as a platform of development, and in order to represent this work in a more clearly way, we have to obtain basic knowledge of this framework. First: an overview of this system will outline a number of important characteristics that provide relevant basis knowledge when dealing with affairs on this platform. Second: two parts which are mainly concerned with the interaction mechanism of OpenMASK – adaptor and interactor, will be explained in details.

3.1 OpenMASK framework overview

OpenMASK [3] is the new generation of GASP (General Animation and Simulation Platform) [1][2] developed in IRISA by the SIAMES project. This framework is an object oriented development environment allowing real-time simulation and visualization of autonomous or user-driven entities evolving within complex worlds.

3.1.1 Basic distribution paradigm

The basic structure of this framework is shown in figure 3.1. Each entity in the system is composed of one or more simulation objects. These simulation 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 evolution. This evolution 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.

![figure 3.1 - basic structure of OpenMASK](image-url)
At each simulation step, the inputs of the object will be read and passed 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. After that, once these result values are needed by the outputs (for example, `computeOutputs()` function), they can simply ask to the control parameters before next simulation step. 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. Objects are also able to communicate with each other by sending events and messages; an internal function (usually called `processEvent()`) is in charge of dealing with these dialogues and requests.

Each simulation object is assigned to a process and processes are assigned to workstations in a configuration file. Each process owns a particular simulation object: a controller, which schedules all the local simulation 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. 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, it enables the construction and the execution of complex virtual worlds, with a lot of entities with complex and heavy weight calculation behavior.

### 3.1.2 3D Visualization

OpenMASK provides a 3D visualization based on the SGI/Performer library, which is a particular simulation object and is in charge of the graphical representation of any objects associated to geometries in the simulation. In fact, during the simulation, it looks for all simulation objects which have a 3D representation and visualizes them; moreover, it creates the mirrors corresponding to these objects and therefore updates the position and orientation values according to the corresponding outputs of the objects.

As the result, 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 visualization is a simulation object, there can be as much visualizations 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 visualization, so one or several visualizations 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.

### 3.2 Interaction in OpenMASK

When interacting with objects in OpenMASK, we classify the objects according to their functions: 1) tools that are used to manipulate the objects, 2) objects that are manipulated by those tools. In brief, Interactions begin with a series of ‘ask-and-answer’
dialogue, with the form of messages, between a manipulator and a manipulated object. These messages are sent alternately until both sides agree to establish a dynamic connection between the outputs of the manipulator and the inputs of the manipulated object. Equally, interactions end with this messages-sending manner.

The protocols of interaction between these two types of objects, which include rules of interaction and protocols of communication, are defined. That is to say, with the help of an interactor, the tools which are the metaphors of interaction will understand how to interact with objects. In the same way, with the help of an adaptor, those simple objects will learn how to react to the interactors.

### 3.2.1 Simulated Object

All simulation objects in OpenMASK are inherited from one basic object: *PsSimulatedObject*. Figure 3.2 illustrates the structure of an object, which inherits from the *PsSimulatedObject*, and gives an example of its possible structure: as we can see, it has two inputs, two control parameters, and two outputs, which represent respectively the position and the orientation values. 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. Objects that inherit from this model will possess essential functions, such as *compute()* which does some necessary calculation in every simulation step, and *processEvent()* which handles messages sent by itself or others. Of course, those inherited objects must have their functions rewritten or declared to meets their proper needs.

![Simulated Object](figure 3.2 Simulated Object)

### 3.2.2 Interactor

An object encapsulated by an interactor will not be a simple simulation object anymore, on the contrary, it becomes an interaction tool, which is able to control and manipulate other objects. At the beginning of an interaction, an interactor will create its outputs dynamically in order to connect them to the manipulated object. After that, during the connection, it will update its outputs in order to ‘tell’ the manipulated object of the newest values, for example: the position and the orientation, expected by the user.

Figure 3.3 is an example of an interactor who integrates an object, which is generally to be an interaction metaphor such as virtual ray or mouse, into a usable interaction tool. The two outputs: *interactionPosition* and *interactionOrientation* will be connected to the inputs of the manipulated object and are responsible for offering appropriate values, according to the wish of the user, to this object.

Since there are many different types of interaction needed in the virtual world, there are quite distinct metaphors used for diverse purposes, as we have seen in chapter 1.3.2 – ray-casting, virtual hand…etc. Similarly, a single interactor will not be capable of covering and accomplishing those plenty of interaction behavior, so it has to be designed in many different ways to do a variety of manipulations.
3.2.3 Adaptor

Similarly, objects encapsulated by adapters become objects that can be manipulated. At the beginning of an interaction, adaptor will create its inputs dynamically in order to connect them to the manipulator. After that, during the interaction, adapter will take control over some status parameters of the adapted objects, for example, the positions and the orientations. Therefore, a user is able to interact with an object as her/his wish. Also, an appropriate adapter will make it possible for an object to be manipulated by more than one tool simultaneously.

1) Mono-user Adaptor
   Mono-user adaptor is used for the condition which one object is allowed to be manipulated by one interaction tool at one time. Figure 3.4 represents an adapted object and gives an idea of the function of a mono-user adaptor. An object, which is generally to be a technical object and encapsulated by an adaptor, will acquire the interaction protocol then become a controllable object. The two inputs: interactionPosition and interactionOrientation are responsible for receiving the values that are sent by the interactor at every cycle of the simulation, after that, through certain calculation steps of the compute() function, then finally apply the new result values to the control parameters of the technical object.

2) Multi-user Adaptor
   When several users want to interact with an object at the same time, we have to use a multi-user adaptor. In fact, its working principle is the same with the mono-user adaptor, except the two following points:
   - During the interaction, the multi-user adaptor is able to create or delete arbitrary sets of inputs for different interactors who ask for interaction,
while mono-user adaptor is capable of creating or deleting one set.
- Values that are read from the inputs of the adaptor will not be applied directly to the control parameters of the object. Instead, they will pass through a calculating function for combining the results of all inputs, for example their average value, and next, apply these values to their associated control parameters.

Figure 3.5 shows the basic structure of this adaptor, here we use abbreviation: iPos and iOrt for interactionPosition and interactionOrientation.

![figure 3.5 a multi-user adaptor](image)

3) **Constraint-Supported Adaptor**

As the subject of the stage is mainly concerned, in OpenMASK, it is possible to apply some constraint characteristics to objects in order to make simulations more realistic and more similar to our real world. Consequently, another type of adaptor is needed to support this new feature – constraint-supported adaptor. Actually it works exactly the same way with the mono-user adaptor; however, input data will first pass through a particular calculating function for verifying whether this value is legal or should be modified to conform to the original constraint settings. Afterwards, the reasonable results will be applied to the control parameters.

To conclude, figure 3.6 presents an example of an interaction between two objects through their own adapter and interactor. A communication link is built dynamically when both have agreed to begin an interaction, and in the same time, create their corresponding inputs / outputs for the connection. Once the connection is done, both will not have to consider about the protocol details, but simply to send or to receive data, until the request of ending the interaction is made by either side.

![figure 3.6 - adapter and interactor](image)
Chapter 4

CONSTRAINT INDICATOR

As mentioned in the introduction, the goal of this project is to design new metaphors of interactions that will be mostly convenient for manipulating the constrained objects during the cooperation process. That is to say, to develop metaphors which represent clearly the constraints on objects in order to provide necessary understanding about the interaction for users. What is called “constraints” here is mainly in connection with two aspects: constraints that are brought by the other users, and mechanical constraints that apply on the objects.

We have explored some of the metaphors which have been created for facilitating the interaction works ( chapter 1.3 ), however, few of them are designed for dealing with constraint subjects. Moreover, the main purpose of our OpenMASK platform is the development and the execution of modular applications, for example simulations for vehicle industry to build and test their automobile models. Consequently, the constraint matter appears to be especially important, since this platform will probably be used to do certain precise work and a well-designed metaphor will not only simplify the manipulation processes but also improve enormously their efficiency.

For this reason the idea of Constraint Indicator was eventually born. In fact, the original concept was from chapter 1.4 - mini-project of Champalaune and Guichard, they use different geometric representations, which envelop the manipulated object, to express different available actions. Maybe we can extend this idea to represent constraints that append to objects. This chapter will introduce thoroughly to this Constraint Indicator – from its basic idea, its development then to the final testing work.

4.1 Basic idea

The designing process was considered step by step according to different levels of aspects - from the highest user’s view to the lowest object’s view.

4.1.1 User’s view

Being a user, what will be mostly concerned with this Constraint Indicator when dealing with objects are: 1) the opportune moment it shows up to point out the constraints and assist the user; 2) the form it appears to be in order to illustrate the constraints clear and make them easy to understand.

The first issue, “moment”, is not a complicated problem – when an object is selected, the indicator will be displayed, and on the contrary, it will disappear while this object is out of manipulation. Second, the form – in fact, it is the key to decide whether this Constraint Indicator is practical or not. From my conception, this indicator will be composed of 3 main parts:
- **Coordinate positioner**
  It is made up of three bars which are perpendicular to each other as x, y, and z-axis respectively. The function of this positioner is to locate the coordinate system of the object the operations will be based on. It will put its center (the point where three bars meet) on where the selected object has designated and also orientate itself with the assigned value. In normal case, the positioner will be located at the center of the selected object with the same orientation, however there are exceptions when the relationships between objects are much more complicated; for example: several constraint objects connected to each other, we will go into details of these cases in the part of implementation. (chapter 4.2)

- **Action reflector**
  This part associates awareness to the actions that the user applies on the object, for example, when the object moves along x axis, the reflector will show a tab that shifts along the x axis of the coordinate positioner (its x-bar) simultaneously; it will be the same with the rotation around any axis.

- **Limitation symbol**
  It is for displaying the constraints of the objects relatively to our coordinate positioner. For example, if the object is limited for rotating around z-axis from -10° to +25°, we will place two limitation symbols around z-bar of the positioner on 10° and +25° respectively.

While these three parts work well in relation with each other, first we have a coordinate system which points out the object’s position and orientation; second, with the help of the action reflector, we can see clearly what we are doing and the state of the object during the interaction; then if there are constraints, we find the symbols which indicate the limits of the action available, and will understand that the action is not permitted anymore if the reflector meets the limitation symbol.

### 4.1.2 Calling view

Once we decide the moment to display the indicator, next we have to study the standard process of a common simulation; such as the time when each class is created, the order of the calling of certain functions, and the relation between the classes. After that, we are able to determine the point, or the proper place, to put (which means actually to create and to make functioning) the Constraint Indicator. The left part of figure 4.1 demonstrates the standard flow path of our most frequent used simulations; the ProtocolTeacher is in charge of providing the proper connector (same with the adaptor which is described in chapter 3.2.3) to the object which will be manipulated. With the help of the ConnectorProvider, the ProtocolTeacher is able to change the connector arbitrarily during the interaction.

Since we would like our Constraint Indicator to be applied on any manipulated object, which means on objects encapsulated by whichever connector, it is better to put the indicator in Protocol Teacher level, so that even if the connector is changed, the indicator will remain working. As the result, the right part of figure 4.1 shows that when connected to an interactor (so that the object is under control) the Constraint Indicator will call its show() function to display the constraints information, and the hide() function will be called as the interaction ends – which corresponds to what was just been discussed about the subject “moment” in chapter 4.1.1.
4.1.3 Module view

As decided in 4.1.1, our Constraint Indicator consists of three parts, now we will consider in detail what information is essential to provide to these parts for initializing and performing correctly during the simulation.

- **Coordinate positioner (Bar)**
  1) Initial position & orientation value of the adapted object.
     => For locating the coordinate system.
  2) Initial position & orientation value of the center axis of the adapted object.
     => For placing the coordinate system at the beginning of the interaction by using the offset value between the location of the adapted object and the location of its axis center.
  3) Dynamic position & orientation values of the adapted object during the interaction.
     => For the coordinate system being always on the same place in relation to the adapted object (following the object).

- **Action reflector (Tab)**
  1) Initial position & orientation value of the Coordinate positioner (bar).
     => For placing the reflector at the beginning of the interaction. *
  2) Dynamic position & orientation values of the Coordinate positioner (bar).
     => For following the bar during the interaction. *
  3) Dynamic position / orientation values of the adapted object.
     => For tracing the current position or orientation value of the adapted object in order to “reflect” these values on Constraint Indicator. *

- **Limitation symbol**
  1) Initial position & orientation value of the Coordinate positioner (bar).
     => For locating the symbol.
  2) Constraint values of the adapted object.
     => For placing the symbol at the beginning of the interaction by using the offset value between the constraint value and the initial location of the bar.

*1 The reason we use the values of bar instead of the adapted object is: if we make it possible to change the location of bar dynamically, the display of this Action reflector will remain correct relatively to the whole Constraint Indicator; same with the Limitation symbol.

*2 The choice between the position or orientation values depend on the function of this tab – for a tab which reflects the translation of the object, it will take the position values; and vice versa.
3) Dynamic position & orientation values of the Coordinate positioner (bar).

=> For following the bar during the interaction.

### 4.1.4 Object’s view

After analyzing the necessary information that Constraint Indicator will use at the beginning and during the interaction, we have a rough sketch of the development steps. Next, we will use this information to fix the idea about the basic structure of this project. Figure 4.2 describes the relationship between the objects and how their inputs/outputs connect to each other; purple lines represent data which is sent from the bar; while data sent from the adapted object is represented with orange lines. As the result, the interdependence and the organization are now clear enough, what remains is to carry this idea out and to implement the internal part about the data operating and calculation.

![Figure 4.2 – relationship between objects](image)

### 4.2 Implementation

This section will present principal development issues of all units associated with Constraint Indicator. The sub-sections 4.2.1–4.2.3 are relating to the interior functions; and the way to put this indicator into use will be described in 4.2.4; finally, the external visualization effect – geometry part, will also be illustrated in 4.2.5.
4.2.1 ConstraintIndicator

**Inheritance tree**

```
PsSimulatedObject
  ↑
ConstraintIndicator
```

**Feature**

The class `ConstraintIndicator` is created by `DoubleProtocolTeacherPlus` (see 4.2.4) at the beginning of the simulation; and is deleted at the end. This object has no inputs or outputs but is responsible for creating and managing the `coordinate positioner (bar)` and the `action reflector (tab)`. Since it only needs to be capable of creating objects, I have chosen it to inherit from a primitive object – `PsSimulatedObject`, which is the fundamental object of OpenMASK and all basic low-level methods are implemented, so that objects that inherit from it need only to care about their own parts.

**Development**

Creating an object dynamically in OpenMASK requires the following steps: 1) create a new a `PsObjectDescriptor` object, 2) call current `controller` for its `createObject()` member function using the `PsObjectDescriptor` that we have created as its parameter.

An object is completely described for the other objects by its object descriptor, and when creating a new object descriptor, a `PsConfigurationParameterDescriptor` is needed. In fact, dealing with this `PsConfigurationParameterDescriptor` is one of the main issues of this project, because configuration parameters are described before the creation of a simulated object and are used by the object to parameterize its state and behavior, such as its initial position / orientation values and its constraint information. These parameters are described in a configuration file which can be modified arbitrarily by the user, and will be loaded into a `PsConfigurationParameterDescriptor` at the beginning of the simulation; it structures configuration parameters in a hierarchy, and enables access to sub-parameters.

Some member functions of `PsConfigurationParameterDescriptor`, which are mostly often used within this project, are:

- `getSubDescriptorByName("NameOfTheDesiredSubDescriptor")`: to retrieve the value of the sub-parameter by indicating the name, ex: `getSubDescriptorByName("Position")`.
- `appendSubDescriptor(SubDescriptor) / appendSubDescriptorNamed("SubDescriptorName", SubDescriptor)`: add new sub-parameter to this descriptor.
- `changeConfigurationParameter(DesiredValue)`: change the value of this sub-parameter to the desired value.

In our case, the `Constraint Indicator` is created with the same parameter descriptor as its corresponding simulated object by using copy constructor, so that it has all information about the default state and behavior of this object. Next, it will provide this information to its `coordinate positioner (bar)` and `action reflector (tab)` by creating them with the same parameter descriptor, but in the same time modifying certain values or adding some new parameters: offset values, geometry file name, and its reference name (the name of the object to establish a connection with). Therefore, the bar and the tab are able to retrieve the information they needed easily and quickly during the simulation.

Deleting an object dynamically is not complicated - just giving the name of the
object which will be deleted to the controller’s destroyObject() function and all will be done. The creating and deleting functions of the bar and the tab are declared as public methods of the Constraint Indicator, and will be called at the beginning and the end of the interaction by DoubleProtocolTeacherPlus.

4.2.2 OmvDoubleShiftedPO

**Inheritance tree**

```
PsSimulatedObject
  ↑
OmInteractiveObject
  ↑
OmvMotionLess
  ↑
OmvDoubleShiftedPO
```

**Feature**

The class OmvDoubleShiftedPO is used to “follow” a certain designate object during the simulation with a parameterized offset in position and orientation. In fact, there is already an object which does the same thing in OpenMASK – OmvShiftedPO, however, it has only one input and therefore receives both position and orientation values within one connection. Unfortunately in normal case, objects of OpenMASK have two outputs separately for the position and orientation values. Consequently, the old OmvShiftedPO is useless; and hence this OmvDoubleShiftedPO is designed to fit in with the most requirements.

As the inheriting tree shows above, OmInteractiveObject inherits from PsSimulatedObject, and to enable any interactive process, it has been decided to subdivided the compute() method in three computation methods: computeInputs(), computeParameters(), and computeOutputs(); this allows any adaptor to handle the control parameters of the templated class and objects inherited from it is able to deal with computations of inputs, outputs, control parameters separately. OmvMotionLess has two outputs but no inputs; it is given initial position and orientation, and implements computeOutputs() so that unless it is adapted, it will never move during the simulation. Back to OmvDoubleShiftedPO, as described, it has two inputs and two outputs, and during the simulation it will compute values received from connected object to ensure the distance between itself and the connected object is conformed to the default offset value.

Owing to its characteristic as able to follow others, in this project, two modules are suitable and will be created as this OmvDoubleShiftedPO - Coordinate positioner (Bar) and Limitation symbol (see 4.1.3), since their purpose is to display constant information relatively to the adapted object.

**Development**

In fact, the implementation principle of the OmvDoubleShiftedPO is very similar to the OmvShiftedPO except that two inputs and two offset parameters are created in the constructor while OmvShiftedPO creates only one.

During the initialization step, the name of the object which will be connected to and
its output names are read from the parameter descriptor (introduced in chapter 4.2.1), after that, two connections can be established. Next, at every cycle of the simulation, the incoming data (the position and orientation values of the connected object) will be read and recorded by `computeInputs()` method, then these values will be computed with the default offsets by `computeOutputs()` method and the result will be applied to update its position and orientation parameters, after that, the ancestor method of `computeOutputs()` will be called for setting the newest position the orientation. We will go into details about the computation in next section.

### 4.2.3 OmvTranslationIndicativeTab / OmvRotationIndicativeTab

#### Inheritance tree

![Inheritance diagram](diagram.png)

#### Feature

As decided in chapter 4.1.1, module *action reflector (Tab)* is responsible for displaying current status of the adapted object by using certain “tabs”, and that’s what these two classes do - three OmvTranslationIndicativeTab and three OmvRotationIndicativeTab will be created to represents each of three axis (x, y, z) respectively: during the interaction, they will not only follow the object but also in the same time, move along (OmvTranslationIndicativeTab) or around (OmvRotationIndicativeTab) each axis in proportion to the movement of the object.

The inheritance tree is similar to the previous section, so as the structure of these two classes is almost the same with OmvDoubleShiftedPO; except that they have three inputs instead of two. Their first two inputs will be connected to the *Coordinate positioner (Bar)*, and the third input of OmvTranslationIndicativeTab is of type PsTranslation and will be connected to adapted object’s position output; while the one of OmvRotationIndicativeTab is of type PsHPRRotation and will be connected to orientation output.

#### Development

Because the computation here is more complicated than others, a function named `mapping()` is declared which is specially in charge of computing output by using current input values, and it will be called in every cycle of the simulation. Moreover, the result values will be applied in `computeOutputs()`.

In three-dimension world, we often use matrix to deal with operations about the points, for example the transformation, the rotation, and the scale, and so on. With the help of the mathematical library - `pfLinMath.h` which provides necessary functions for matrix manipulation, computations in our project become much easier. Take OmvDoubleShiftedPO for example: to maintain a fixed distance to certain object, it only
needs to make two matrixes, one is filled with the predefined offset values and another is filled with the current position & orientation values of the followed object, after that, we will obtain the result position & orientation values by multiplying these two matrixes.

Similarly, to calculate the outputs of these two kinds of tab, except for the offset matrix and matrix of the position & orientation of the reference, the components of the position or orientation value of the adapted object on each axis are required, and then the result matrix will be formed by the following equation:

\[
\text{matResult} = \text{matOffset} \times \text{matComponent} \times \text{matReference}
\]

Note that when calculating the component value, the orientation of the adapted object must being taken into account, that is to say at first, the total movement of the adapted object will be transformed to map to its local coordinate, after that the corresponding component values of each axis are then retrieved.

4.2.4 DoubleProtocolTeacherPlus

**Inheritance tree**

```
PsSimulatedObject
   ↑
InteractiveObject
   ↑
ProtocolTeacher
   ↑
DoubleProtocolTeacher
   ↑
DoubleProtocolTeacherPlus
```

**Feature**

As said before, a protocol teacher, through a connector provider, can create the proper connector which will handle the mechanical part of the interaction. Moreover, during the simulation, the protocol teacher can change the connector used to carry the interaction. From the inheritance tree, DoubleProtocolTeacher possesses all characteristics of ProtocolTeacher, but furthermore, is specialized in dealing with objects which has two “type to control” – usually are of type PsTranslation and PsHPRRotation, as the design of the OmvDoubleShiftedPO is for the most normal use case.

DoubleProtocolTeacherPlus here works exactly like DoubleProtocolTeacher, however, it is able to create a ConstraintIndicator to show constraints information during the interaction.

**Development**

The ConstraintIndicator will be created at the initializing moment, the importance is: except for the original PsConfigurationParameterDescriptor, the name of the adapted object and its output names are retrieved, then added to the configuration parameters descriptor (see 4.2.1) so that the ConstraintIndicator is able to make its own sub-module (coordinate positioner (bar), action reflector (tab), and limitation symbols) connect to the
adapted object afterwards.

While the interaction begins, *DoubleProtocolTeacherPlus* gets the pointer of its own created *ConstraintIndicator*, and call indicator’s `show()` function in order to show up the sub-modules. In the contrary, indicator’s `hide()` function will be called every time when the interaction ends.

### 4.2.5 Geometrical design

#### Open Inventor file format

Creating a geometric figure that can be displayed in OpenMASK, we have first to write certain scripts in the *Open Inventor* file format (.iv), then transform the script into .pfb file format which is now the selected standard format of OpenMASK and thus can be recognize by our platform. A simple structure of an .iv file is as following:

```inventor
#Inventor V2.0 ascii
Separator {
  Scale {
    scaleFactor 1 1 1
  }
  Material {
    diffuseColor 0.0 0.0 1.0
  }
  Cube {
    width 2
    height 2
    depth 2
  }
}
```

The first line of an Open Inventor file will indicate that this is, in fact, meant to be an Open Inventor file. Then the node named “Separator”, which encloses the description of the scene, forms a group of nodes and separates their effects from the rest of the scene. After that, the node “Scale” is a kind of transformation nodes which change the geometric properties of objects in a scene, like location and size. And the easiest way to change the appearance of objects is with a “Material” node. Most of the fields for this class are devoted to controlling color in various ways; here it indicates the color blue of this figure with `diffuseColor` field. Finally the shape node called “Cube”; shape nodes are objects that can be made visible in a scene. Particular classes of shape nodes generate particular shapes of objects in the scene. Each class has its own collection of fields whose values modify the appearance of a shape. In fact, Open Inventor file has more complex and powerful functions, however up to present, it is already quite enough to create some simple geometric figures for our project.

### Development

Recall that as presented in the design issues discussed in chapter 2, the two essential points for a good metaphor are: easy to use and provide sufficient awareness to users. Therefore, the geometric representation of this *Constraint Indicator* must be as simple as possible, but at the same time, has to be capable to express the information clearly. So the appearance of the indicator is composed of primitive geometric figures: three lines which
are perpendicular to each other form a coordinate, furthermore, three cylinders with red green and blue color attach separately on the top of each lines to emphasize the three axis (x, y and z). Afterwards, I have chosen to use cubes for representing the translation phase while the spheres stand for the rotation phase, since the impression of the shapes can imply closely to the corresponding actions: cubes for moving straight; spheres for turning around. Concerning to the color aspects, colors with warning properties such as red or orange are used for pointing out the constraints, and colors with indicative meanings such as green or blue present current actions – the same idea with the design of the traffic lights. Finally, to make the track of the rotation more evident (the translation already has an original trajectory – the coordinate), I have put three white rings around each of the cylinders so that the actions or the constraint symbols of each axis can take places or be put on their own circular trajectory. The images of this geometric design can be found in chapter 4.3.3 – test result.

4.3 Results and test

4.3.1 Moment of creation and life time

To summarize the development, figure 4.3 shows the creation flow chart of all modules we have just talked about. Remember that the creation of the Constraint-Indicator takes place at the beginning of the simulation, and it will remain existent for being referred by DoubleProtocolTeacherPlus and managing its sub-modules until the end of the simulation. While different from that, the creation of the OmvDoubleShiftedPO and two IndicativeTab are at every time the interaction begins (so that the show() method is called) and they will be destroyed at every time the interaction terminates (as that the hide() method is called).

4.3.2 Dependency

In last version of constrained objects, chain reaction of the movement between objects is now implemented, that is to say, the motion of one object will propagate to others which connect or attach to it – just like in our real world. For example, a car and a car door can be two separate objects, but once they are combined to each other, when we move the car, naturally, the door will also move and stay at the same place on the car. On the contrary, when the door is opened, the car will not be influenced, however if the user continues pulling the door to exceed its maximum possible angle, in an environment without friction, the car will start rotating with the direction of the force. Consequently, objects now are classified into two types:

1) **Supporter** : usually being the principal part of a group of objects, ex. the car of the preceding example.

2) **Supported object** : objects that are supported by supporter as its sub-parts, ex.
the car door of the preceding example.

Figure 4.4 and 4.5 show two kinds of dependency between objects in our project: in the former case, the supporter is obviously the adapted object; while in the later case, the supported object is the one who is currently under control. Notice that the action reflector (tab) always relates with the adapted object in order to detect its movement, and the coordinate positioner (bar) always associates with the supporter to locate the object. Besides, since the constraint information will be displayed on the bar, the tab and the limitation symbol (symbol) both depend on it.

4.3.3 Test results

Here we will take a look at the outcome of this project through three typical conditions – when one of the following objects is the target of interaction:

1) **Supporter**

The images show an empty pocket of the car and its main part is under controlled, we can find that with the help of the coordinate positioner (bar), the whole Constraint Indicator is located at the principal axis of the object, in the same time; the orientation of this object is also clearly presented due to the x, y, and z axis of the bar. The orange cubes inform us the possible amount to shift the object along each axis, while the orange cones indicate the limited angels to do the rotation. When we move the object, those action reflectors (tab), as the green cubes and blue balls, will move as well according to the amount of movement on each axis. Furthermore, if the object reaches the extremity of the constraint (+90° of the z-axis), the corresponding tab (blue ball of the z-axis) will meet the limitation symbol (orange cone of the z-axis) to tell the user that the object is not able to go further out of the boundary.

Notice that the color of the symbol represents different types of constraint: in
ordinary situation the color is orange, but if the constraint happens to be strict – that is to say the object is fixed on a certain point of an axis (ex. +42 unit of x-axis, or -0° of y-axis) and can not be moved along or around this axis at all, we use a red symbol to indicate this.

2) Supported object

![Figure 4.8 – supported object(1)](image)

![Figure 4.9 – supported object(2)](image)

When working on a supported object (the door of the empty pocket on figure 4.8 and 4.9), there is only one point in particular compared to the previous case: the coordinate positioner (bar) will still locate the Constraint Indicator at the principal axis of the object. Unless the supporter moves (usually owing to the chain action of the movement of the object), the bar will stay motionless. As the result, actions which apply on the object will be treated as base on the local coordinate of the supporter, and the effects of the representation of the action reflectors (tab) and the limitation symbol are more obvious and clear in this way.

3) Object with its axis tilted

![Figure 4.10 – tilted object (1)](image)

![Figure 4.11 – tilted object (2)](image)

Often, the object is not oriented as “upright” as we imagined, for example the steering wheel of figure 4.10 and 4.11, its z-axis tilted to +65° and if the Constraint Indicator still locates like before, several serious errors will be produced, hence this is an important matter to be concerned with. Fortunately, thanks to the matrix manipulation we have mentioned at chapter 4.2.3, the problem is now resolved successfully. From the images, both tabs and symbols work well with the new titled coordinate.

4.4 Future work

4.4.1 Liberty for the user
Rather than writing fixedly the geometry file names and the offset values of the sub-modules of the Constraint Indicator in the source code, it would be better to put them in a configuration file, as those setting of the position, orientation or constraint values of other objects which are created at the beginning of the simulation. So that the user can modify these values easily and change the appearance of its own Constraint Indicator arbitrarily according to their preference.

Moreover, it is possible to let the user move the Constraint Indicator and place it in wherever she/he wishes during the interaction, maybe for them to see clearer the information or more comprehension about the current situation of the action. In theory, the more liberty we provide to the user, the more ways they will likely find to use for enhancing their works.

### 4.4.2 Advanced capability

Although the Constraint Indicator is now capable of providing evident constraint data, there are still many functions that can be added or improved. If the indicator is able to scale itself automatically to fit the size of the adapted object, then that we won’t be worry anymore about either the indicator is going to “cover” the whole object or too small to be seen. This is going to be realized by beginning with re-designing a new set of geometric figures which can be resized in OpenMASK.

In addition, since now the indicator is only able to explain two types of constraints – translation and rotation, other more complex constraints could be taken into consideration in the future. It will be very interesting if we are capable of displaying information about those curved trajectories, sliding tracks, or even the chain reaction which has been mentioned in chapter 4.3.2.

### 4.4.3 Combine with the further system

As the Constraint Indicator provides a good visualization of the constraint data, we may benefit from this still more in other way. For example if at a later time, the constraint values can be changed dynamically during the interaction, the indicator will be an ideal interface to help the user doing the modification – the user simply has to move the limitation symbol for indicating the desired new constraint value, then the indicator will transfer this new value to the system. However, notice one thing very important: in current implementation version of Constraint Indicator, we obtain the constraint values by reading them directly from a certain pre-written configuration file, which means in a static way. So if these constraint settings become dynamic in the future, we have to design another manner to acquire and memorize new constraint values dynamically; for example, by asking the object for its newest values every time when the constraint settings have been modified.

Finally, the main advantage of this Constraint Indicator is that its well-designed appearance is very practical for reuse or adjusting, and the sub-modules were being developed separately as different classes, so that it is very easy to modify or extend its functions for further needs.
Chapter 5
MULTI-USER WITH CONSTRAINTS

Now we have successfully developed a good metaphor for displaying the constraints, however up to present, it is limited to put into practice only with the interaction of single user. In fact, OpenMASK has already supported the interaction of multi-user, and as we know, supported objects with constraints; unfortunately, these two capabilities have been developed separately and cannot be applied on one simulation at the same time.

So that for improving the realism of our virtual world when manipulating objects and taking the most advantage of the Constraint Indicator, our work next is to study the mechanism of the functions which support the interaction with multi-user and with constraint objects respectively, then try to combine them together; after that, apply the Constraint Indicator on this newly created mechanism. Before entering the key subject of the development and the test results, we will first take a look at two fundamental modules that deal with the protocol affairs of the interaction with constraint objects and multi-user separately.

5.1 Basic modules

5.1.1 ConstraintDefaultAdaptorOffset

Inheritance tree

```
NConnector
  ↑
DoubleConnector
  ↑
ConstraintAdaptorOffset
  ↑
ConstraintDefaultAdaptorOffset
```

Feature

The class ConstraintDefaultAdaptorOffset is a specialized adaptor, which allows making any InteractiveObject (when the type of the parameters to control are PsTranslation and PsHPRRotation) to behave like a constraint object, depending on the constraint parameters values which are set in advance. Also this class allows the adapted object to communicate with any Constraint controller (like controllers of minimum or
maximum limits), or even to be controlled by another object.

From the inheritance tree, we can see very clearly that the ancestor of this class is \textit{NConnector}, which is in fact the prototype of an adaptor and is inherited from a primitive class called \textit{AbstractConnector}. Next, the \textit{DoubleConnector} possesses the necessary functions of an adaptor which is suitable for working with objects having two type of the parameters to control (Ps\textit{Translation} and Ps\textit{HPRRotation}), and so as is the most used adaptor in ordinary cases when no constraints or multi-user features are required. After that, the \textit{ConstraintAdaptorOffset} does indeed the principal part of work for carrying out the possibility of making objects to hold its constraints, and then this \textit{ConstraintDefaultAdaptorOffset} just redefines some functions in order to use the correct methods to set any data member of type of the parameters to control.

\textbf{Principle}

The greater part of the member functions in \textit{ConstraintAdaptorOffset} are inherited directly from \textit{NConnector} and without making any changes, for example: the method of the connection with the interactor (\textit{interactorConnection()}), disconnection with the interactor (\textit{interactorDeconnection()}) or the output computation methods. Even so, there are two methods that are different from its ancestors’ and are the key to accomplish those constraint characteristics—\textit{PreParameterCalculation()} and \textit{PostParameterCalculation()}.

The former method is responsible for gathering and summing up all input values, including the translation offset and the rotation offset values of all supported object (see 4.3.2). After that, method \textit{PostParameterCalculation()} will check if the movement request which is given by the interactor can pass the test of the constraint span for each axis (in position and orientation). So the constraint object will try to achieve the movement value in the limit of its constraints, and the part of this movement which is beyond its constraints limits will be propagated to its support (if it has one), which will pass this request of movement through the check of its own constraints. Afterwards, the correct value for the movement will be calculated and assigned to current control parameter.

5.1.2 \textbf{DoubleMultipleConnector}

\textbf{Inheritance tree}

\begin{center}
\begin{tikzpicture}
  \node[draw, fill=orange!30] (root) {DoubleMultipleConnector};
  \node[draw, fill=orange!30, above of=root] (child1) {DoubleConnector};
  \node[draw, fill=orange!30, above of=child1] (root2) {NConnector};
  \draw[-] (root) -- (child1);
  \draw[-] (child1) -- (root2);
\end{tikzpicture}
\end{center}

\textbf{Feature}

The class \textit{DoubleMultipleConnector} is a specialized adaptor, which allows making any \textit{InteractiveObject} (when the type of the parameters to control are Ps\textit{Translation} and Ps\textit{HPRRotation}) to be connected by several interactors, that is to say, manipulated by several users at the same time.

\textbf{Principle}
In *DoubleMultipleConnector*, the connection or disconnection methods are very
different from those adaptors before. In fact since there may be many interactors which
interact with the object simultaneously, the original type for carrying data of single
interactor is now insufficient, hence, a STL container – *map* is used at the moment to
store all information needed for each interactor. When an interactor connects to the object,
a new entry of the input *map* for this interactor will be created, and at each cycle of
simulation, each input value of interactors will be read and calculated for an appropriate
result (usually the average of all input values), then the control parameter will be updated
to this result value. And when an interactor wants to end the interaction, the
*DoubleMultipleConnector* will not really call the ancestor’s disconnection method, on the
contrary, it does a disconnect-like trick: deleting the corresponding input entry of this
interactor. In another word, the interactor is not actually disconnected from the object but
just not being read by the object, so that the request of this interactor through the object
will be ignored until next connection.

5.2 Implementation

This section will present principal development issues of supporting multi-user
interaction with constraints. The sub-sections 5.2.1 is relating to the new adaptor that is
responsible for the interior mechanism; and the way to put our *Constraint Indicator* into
use in this case will be described in 5.2.2 and 5.2.3.

5.2.1 DoubleMultipleConstraintConnector

**Inheritance tree**

```
+-----------------------------
<p>| |
|                           |
| ConstraintDefaultAdaptorOffset |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td>ConstraintDefaultAdaptorOffset</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>-----------------------------</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>DoubleMultipleConstraintConnector</td>
</tr>
</tbody>
</table>
```

**Feature**

This new adaptor combines the important functions of both *ConstraintDefault-
AdaptorOffset* and *DoubleMultipleConnector*, so that it enables several users to interact
with one object, and in the same time, preserves the constraint characteristic of the object.
Hence, this *DoubleMultipleConstraintConnector* inherited directly from *Constraint-
DefaultAdaptorOffset*, therefore it possesses all capability that its ancestor has and what
we have to care about is how to modify or add methods on this adaptor to support
interaction with more than two.

**Development**

In fact the development of the *DoubleMultipleConstraintConnector* is not too
difficult as imagined, however, the most complicated is to understand the mechanism of
the old adaptors described at chapter 5.1; for example: the work of each functions, the
usage of all storage tables (*map*) and the relationships between each others. Once these
are clear, what remains for combining two adaptors are simply to call the correct function
in the right time and to put the appropriate data in the storage tables.
When connecting to an interactor, first the ancestor’s connection method will be called, then just like `DoubleMultipleConnector`: a new entry of the input map for this interactor will be created. For updating the value of the control parameters at each cycle of simulation, each movement request of the interactors and the input offset values of the supported objects will be read and calculated as the first step combination result, after that, this result will be passed to a check function for ensuring this value to be within the constraint setting (if not, the value will be adjusted a little to become available). Consequently, the value of the control parameters will always conform to both the expectation of the interactors and the constraints that the object holds. At last, to disconnect from a certain interactor, the adaptor deletes the corresponding input entry of this interactor – as the `DoubleMultipleConnector` does, then performs the disconnection method of `ConstraintDefaultAdaptorOffset` while avoiding the command which does the real disconnection work.

Notice that to put this new adaptor into use, we have to add this adaptor to the list of connectors in the corresponding connector provider so that the provider will be able to propose this adaptor to the protocol teacher, and the name of the connector in the configuration file of the object also must be modified.

### 5.2.2 DoubleMultipleProtocolTeacherPlus

#### Inheritance tree

```
   DoubleProtocolTeacher
     ↑
DoubleMultipleProtocolTeacher
     ↑
DoubleMultipleProtocolTeacherPlus
```

#### Feature

A protocol teacher, as described before with the help of the connector provider, is responsible for providing proper connectors to its own created object, and during the simulation, the protocol teacher can change the connector according to different situations. The detailed function of the `DoubleProtocolTeacher` was explained in chapter 4.2.4, and its descendant `DoubleMultipleProtocolTeacher` is especially for dealing with objects that can be manipulated by several users.

Thus, the `DoubleMultipleProtocolTeacherPlus` here works exactly like its ancestor, however, it is able to create a `ConstraintIndicator` to show constraints information during multi-user interaction.

#### Development

The principle of the implementation of this class is, in fact, the same as the `DoubleProtocolTeacherPlus`: to create the `ConstraintIndicator` in the initializing steps and while the interaction begins, the pointer of the indicator is obtained for calling its `show()` function in order to display the geometric illustrations of the constraints.

However remember that we are now in face of the case that many users may interact with one object at the same time. Therefore, the time to delete the geometric figures is not the moment when any interactor ends the interaction; imagining two users control a certain object together and both of them acquire the constraints information with the help of the `ConstraintIndicator`, then in a while later, one of the users decides to finish
its work, so the disconnection method is called. Now if the hide() function of the indicator is also called as in DoubleProtocolTeacher, apparently the other user will suddenly lose the information of the constraints and gets confused since he is still manipulating the object. Consequently, the appropriate moment to delete the geometries is when there is no one who takes control of the object. For this reason, when the interaction ends, the DoubleMultipleProtocolTeacherPlus will first check if there is still any interaction with this object, afterwards decide whether or not to call the hide() method of the indicator to erase the constraint illustrations.

5.3 Test result

Here we will take a look at the outcome of this project through a complete process of interaction by two users with one object.

Figure 5.1 shows the first user who uses a green virtual ray to take control of a backrest of a car seat, we can see that the Constraint Indicator appears immediately as usual and with the presentation of the two orange cones, we understand that the possible movement for this backrest is limited within some degrees of rotation around z-axis. Next, a second user joins in the interaction with its tool as the blue virtual ray (figure 5.2).

During the interaction, the movement of the backrest is composed by the actions of the two rays, from figure 5.3 we can discover that the constraint setting is well respected: since the backrest has bent to its extremity, the blue ball is never able to go through the orange cone below.
Now the user with blue ray decides to end the interaction (figure 5.5), however the Constraint Indicator remains working and displays the geometric illustration for the user with green ray to continue working. At figure 5.6 we can see that the geometric figure, which shows constraint information, has disappeared because the last user has also finished the interaction with the backrest.

As the result, it is found that the new adaptor - DoubleMultipleConstraintConnector functions successfully to support both multi-user interaction and the objects with constraints; in the same way, our Constraint Indicator also well integrates with this new adaptor and works satisfactorily to our original expectation.

5.4 Future work

Concerning to the subject of multi-user interaction, we can find that although with the help of our Constraint Indicator, it is still not very obvious for the users to understand the behavior of the object; for example, one user moves a cube along +x axis while another moves it along –x axis, now if the cube doesn’t move at all, each of the user might consider that there is another force which equals to his own but in opposite direction. Hence, for the moment, users can just guess or image possible cases that form the current state of the object, and however there is no way for them to be sure what has happened exactly. The major reason is that in fact, the indicator only shows the result of the sum of each applied force, but without providing the users with the cause of such an outcome. Therefore one possibility of our next work can focus on overcoming this defect: it may be very practical if each component of force of each user is illustrated when there are several users manipulate one object simultaneously; thus the process and the result of the interaction will become clearer, easier to comprehend, moreover at the same time, users are able to have others actions in hand, then to go a step further, to master the whole situation of the task.
CONCLUSION

*Collaborative Virtual Environments* (CVEs) are designed to support real time work and interaction with objects and artefacts. We have presented a variety of techniques that support interactions in CVEs, which include two essential types: user-to-user and user-to-object. Since our goal is to design new metaphors of interactions that are mostly convenient for manipulating the constrained objects during the cooperative manipulation process, some important metaphors and awareness tools, which are used to simplify the cooperative work, have also been described. When trying to put forward new ideas of the metaphor, we better follow some major design points, which are derived from those existing designing examples represented above. Moreover, some ways to experiment the interaction tools have been outlined, since best metaphors mostly come into existence through the steps of evolution - the *test-modify-test* cycle.

Before the development of the new metaphor, we have studied the basic knowledge of our platform - OpenMASK, which includes two important mechanisms: *Adaptor* and *Interactor* that help to create interaction tools and associate these tools with the controlled objects. Adaptors enable ordinary objects to be manipulated, while interactors “teach” the metaphors how to interact with the objects.

The major work of this stage includes the development of the *Constraint Indicator*, which is responsible for displaying constraint information of objects to provide necessary comprehension for users about the interaction. This indicator works well not only with common objects but also with complicated ones, like compounded objects – which composed by a main supporter and certain supported sub-objects, and objects with its axis tilted. Even more, with the implementation of the new adaptor- *DoubleMultipleConstraintConnector*, which supports successfully both multi-user interaction and the objects with constraints, the *Constraint Indicator* is now of the applicable level on OpenMASK and capable of assisting users to improve their efficiency when facing to the constrained interaction in three-dimension universe for cooperative work.

As the development continues, more and more ideas are born about the possibility to extend or improve the ability of the *Constraint Indicator*, like to be more convenient to use, more powerful capability, or to combine with other modules in the system for offering different service. For example in multi-user interaction, it may be very practical for the users to master the whole situation more easily, if the component movement of each one is shown. In brief, it is important to go deep into these ideas and take them into account in the future development of the interaction metaphors.
# INDEX OF THE FIGURES

1.1 combination of two metaphors ................................................................. 5  
1.2 Cubic Mouse ......................................................................................... 6  
1.3 the deformable ray ................................................................................. 7  
1.4 Peripheral lenses ................................................................................... 8  
1.5 Peripheral glancing ................................................................................ 8  
1.6 a semitransparent frustum ........................................................................ 9  
1.7 the name of the user is displayed ............................................................ 9  
3.1 basic structure of OpenMASK ................................................................. 13  
3.2 Simulated Object .................................................................................... 15  
3.3 an interactor ........................................................................................... 16  
3.4 a mono-user adaptor ............................................................................... 16  
3.5 a multi-user adaptor ............................................................................... 17  
3.6 adapter and interactor ............................................................................ 17  
4.1 flow chart of the simulation .................................................................... 20  
4.2 relationship between objects ................................................................. 21  
4.3 creation flow chart .................................................................................. 27  
4.4 dependency (1) ...................................................................................... 28  
4.5 dependency (2) ...................................................................................... 28  
4.6 supporter(1) ......................................................................................... 28  
4.7 supporter(2) ......................................................................................... 28  
4.8 supported object(1) .............................................................................. 29  
4.9 supported object(2) .............................................................................. 29  
4.10 tilted object (1) .................................................................................... 29  
4.11 tilted object (2) .................................................................................... 29  
5.1 first user .................................................................................................. 35  
5.2 two users ................................................................................................ 35  
5.3 during the interaction .............................................................................. 35  
5.4 during the interaction .............................................................................. 35  
5.5 one user quits the interaction ................................................................. 36  
5.6 two users quit the interaction ................................................................. 36
REFERENCES


