Faculty of Mathematics and Natural Sciences

Department of
Computing Science

Interaction in a virtual environment

R.H. Pijnacker

Advisors:

Dr. J.B.T.M. Roerdink
Dr. Ir. A.J.S. Hin

January, 1997

RuG

Interaction in a virtual environment

R.H. Pijnacker

Master's Thesis
Supervised by:

Dr. J.B.T.M. Roerdink
Department of Computing Science
University of Groningen

Dr. Ir. A.J.S. Hin
TNO Human Factors Research Institute
Soesterberg

January 1997

Abstract

In virtual reality systems the computer is used to create an artificial environment. To visualise this environment as if it were real, a head-mounted display coupled to a tracking system can be used; this is called immersion. Interaction with such an artificial environment requires special input devices such as a data-glove or a 3-D mouse. The technology for displaying the environment has reached the level where it can be implemented in a software library. Speed and resolution are only limited by hardware capacities. The development of techniques for interaction in virtual environments is, however, still very immature.

At the *University* of Groningen a virtual reality system, consisting of a head-mounted display, a 3-D mouse, a tracking system and high-performance graphics hardware, is available. To use this system, a method for interacting using the 3-D mouse has been developed, based on existing literature. This method is tested using the VR system and a software library, specific for VR applications. A simple version of a chemistry program is created for this, in which it is possible to build a molecule from single atoms, move the whole or parts of the molecule and delete parts of it.

Samenvatting

In virtual reality-systemen wordt de computer gebruikt om een kunstmatige omgeving te creëren. Een head-mounted display, die is gekoppeld aan tracking-apparatuur, kan worden gebruikt om deze omgeving te visualiseren alsof het echt is; dit wordt immersie genoemd. Om een wisselwerking met zo'n kunstmatige omgeving mogelijk te maken is het gebruik van een driedimensionaal input-device, zoals een data-glove of een 3D-muis, noodzakelijk. De technologie voor het afbeelden van de omgeving heeft het niveau bereikt waarop het in een software-bibliotheek kan worden geïmplementeerd. Snelheid en resolutie worden slechts begrensd door de mogelijkheden van de hardware. De ontwikkeling van technieken voor interactie in virtuele omgevingen staat echter nog in de kinderschoenen.

Aan de Rijks*universiteit* Groningen is een virtual reality-systeem aanwezig, dat bestaat uit een head-mounted display, een 3D-muis, tracking-apparatuur en high-performance grafische hardware. Om met dit systeem te kunnen werken is een methode voor interactie ontwikkeld, die gebruik maakt van de 3D-muis en die is gebaseerd op bestaande literatuur. Deze methode is op het VR-systeem getest. Hiervoor is een eenvoudig programma op het gebied van chemie geschreven, waarmee het mogelijk is een molecuul op te bouwen uit losse atomen, het hele molecuul of delen ervan te manipuleren of delen weg te gooien.

Contents

Preface

After more than three years of studying Computing Science it was time to start thinking about graduating. Choosing between the exciting field of computer graphics and the more theoretical field of algorithms was not easy. But, since I knew the department of Computing Science had purchased a virtual reality system, and since I always had wanted to play with one, I decided to do some work in that field. So, after working for about three quarters of a year, here is the thesis is the last stretch in obtaining the Master's degree in Computing Science.

I would like to thank Jos Roerdink for helping me complete my study with this project and Andrea Hin for spending so much ink on every draft version of this thesis that I wrote, even though she left the department to work for TNO in Soesterberg.

Ronald Pijnacker

1 Introduction

In the field of *virtual reality* (VR), one uses the computer to create the illusion of being immersed inside a real world. There is a number of areas where one could apply this. One of these areas is obviously entertainment. Another is tele-presence, where a robot is remotely operated, so it can work in hazardous environments. Numerous kinds of design, such as architecture, aircraft and car design also benefit from VR. In these design fields the traditional drawing board is replaced with a computer. The whole design process from initial design through to prototyping is carried out digitally. Now also the evaluation of designs can be experienced with computer technology, using virtual worlds. Traffic and flight simulators can be used as a substitute to (potentially dangerous) training with real cars or air-plains. One can easily imagine applications in medicine, such as medical training on a virtual cadaver, ultra-sound imaging or molecular docking for drug synthesis.

Interaction with computers started with command driven interfaces, where commands were typed on a keyboard. These commands were subsequently processed by some kind of interpreter. An improvement to this situation were the menu driven interfaces; the menu selections were however still done using the keyboard. With the introduction of the desktop mouse, this also changed. This lead to the development of graphical user-interfaces, which are the standard for performing interaction in current computing systems. Some believe that with the right level of development, virtual reality and virtual environments will provide the ultimate means of interacting with computers. Future will tell if this is really true. We believe that few users will be willing to immerse themselves in a VR system for normal operation of a computer. For a limited group of applications VR will, however, provide a necessary extension to ordinary computers.

When one wants to immerse oneself in a virtual environment, one must use a number of devices. Firstly, an alternative to the computer monitor should be used that enables stereoscopic viewing. This so-called *head-mounted display* can either completely shield the real surroundings and display a completely artificial world, or it can let the surroundings be visible and overlay an image of virtual objects on it. An alternative to wearing the display on the head is the *BOOM*, where the two displays are mounted on a counterbalanced arm. A big advantage of this is that the user does not have to carry the weight of the display, so a high-resolution CRT-display can be used, instead of the LCD-displays used in HMD's. A disadvantage is the limited freedom imposed by the BOOM's arm.

To be able to interact with a virtual environment a number of different special devices is available. The best known is without doubt the *glove*. Other typical input devices in VR systems are 3-D mice, voice recognition and (the more exotic) haptic devices, that supply force feedback to further complete the experience.

In the following chapter the virtual reality system that we have used for this study is described. Chapter 3 discusses some different methods for interacting with objects in a 3-D space. A method that is applicable in our system is also developed in this chapter. This method is applied in a case study in the field of chemistry. A description of this case study is given in Chapter 4. In Chapter 5 some conclusions are presented.

2 The virtual reality system

Current virtual reality systems are build out of a number of devices. These devices include a head-mounted display (HMD), a glove, 3-D mouse or track-ball as input device and some tracking sensors to follow the users movements. To operate them, a high-performance graphics workstation is necessary.

In the first quarter of 1996 the department of Computing Science and the Centre for High Performance Computing at the *University* of Groningen have purchased a virtual reality system. This chapter gives an overview of the hardware components this system is built from. It also discusses WorldToolKit, a virtual reality software library.

2.1 Head-mounted display

One of the goals of virtual reality is to give the user the impression of being immersed inside the created virtual world. A virtual reality system can be used, e.g. to get an idea of what a building would look like (architecture) or how objects with extraordinary proportions in the real world are constructed (e.g. chemical structures or astronomical phenomena). To create this illusion, many virtual reality systems are equipped with a *head-mounted display* (HMD), a device that is placed at short distance in front of the eyes. This device consists of two screens, one for each eye. On these screens images are displayed, that correspond to the position and viewing direction of the eyes as they are looking at the scene (see Fig. 2.1).

When one displays a three-dimensional scene on a two-dimensional screen information is lost. To regain this information, which might be necessary to estimate depth, for example, a number of so-called 'cues' — occlusion, fogging, applying shadows — can be used. A more complicated technique is that of stereoscopic viewing, where two images, taken from a different angle are positioned in front of the eyes. If the parallax between the images corresponds to the distance between the eyes, the human brain is capable of reconstructing some of the 3-D information from these two 2-D images.[1] This is exactly what a HMD is used for.

The system in Groningen uses a HMD called *VR4*, manufactured by *Virtual Research Systems* (see Fig. 2.2). Specifications of the VR4 can be found in (VRS 1994).

[1] This is actually still a topic of extensive research and things are more complicated than discussed here.

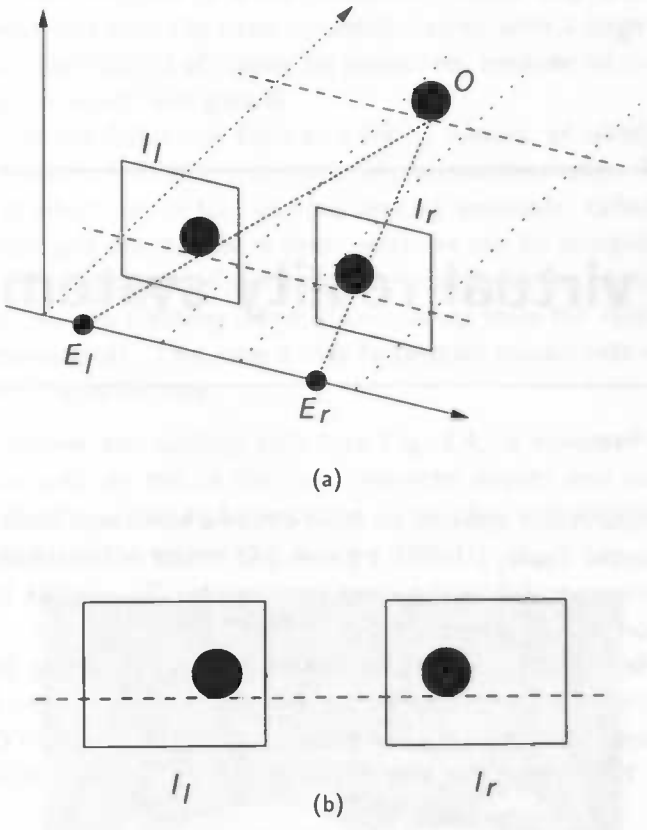


Figure 2.1 (a) Scene with two eyes (E_l and E_r) looking at object O. (b) Resulting images I_l and I_r.

Figure 2.2 The *head-mounted display* used in this project: VR4.

2.2 3-D mouse

Another important aspect of a virtual environment is the ability of user interaction. In most modern systems this is done with a glove that monitors the position of the fingers with respect to the hand and the overall position and orientation of the hand itself. This glove is commonly used to operate a virtual image of the hand. Commands are issued by making gestures, which are interpreted, e.g. by a neural network. In our system, however, the input device is a 3-D mouse (see Fig. 2.3).

Figure 2.3 The *3-D mouse*.

The 3-D mouse (also called flying mouse or flying joystick) is a stick that is held in the hand. A receiver of the tracking system (see section 2.3) is placed inside it, so the position and orientation information can be used. Just like a regular desktop mouse, the 3-D mouse has a number of control-buttons (four in our case). These can be used to issue commands. Note that this is easier than making gestures as is done with a glove.

On top of the 3-D mouse a little knob, called the *hat*, is attached, which can be moved from a default position in four directions (up, down, left and right). The position of the hat is obtained as an analogue signal. This information can be used for example to establish continuous zooming or rotation.

2.3 Position and orientation tracking

In the previous section we already mentioned the tracking system, which is an essential part of a virtual reality system. It enables direct response to a change in e.g. the orientation of the users head, and processing of this new information. It is very important that this is done with minimal time-delay, also called *lag*. Firstly, handling the application gets less

intuitive when a visible reaction to some movement occurs only after some perceptible time. Secondly, when one uses the head-mounted display, with a large lag, it is very easy to get motion sick. This should of course be prevented, because no one would be willing to operate a system in which one gets ill.

The basic part of the Polhemus Fastrak tracking system, of which more information can be found in (Polhemus 1993), is a stationary transmitter unit. This unit generates a magnetic field in which up to four remote sensing antennas, called receivers, can be placed. The position and orientation of these receivers can be computed from the signal that they receive. The strength of the magnetic field limits the spatial extent in which one can accurately use the tracking information coming from the receivers to about two metres from the transmitter. This means that large-scale movements in the virtual world must be performed in another way.

The tracking system we are working with (see Fig. 2.4) is equipped with two of these receivers. One is placed on top of the head-mounted display and can thus be used to monitor the users head position and orientation. The other is fixed inside the 3-D mouse, so that all movements of the mouse can also be used.

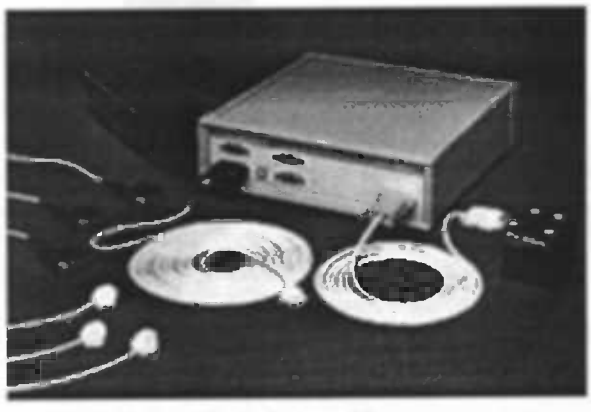


Figure 2.4 The Polhemus Fastrak *tracking system*, consisting of the transmitter unit, some receivers and the control box.

2.4 Graphics hardware

The last part is the workstation that operates all these devices. We are using a Silicon Graphics' Onyx workstation for this. This workstation is capable of rendering 600.000 polygons per second, using two 200 MHz R4400 processors and a Reality Engine^2 graphics subsystem equipped with two RM4 boards. The workstation has 128 Mb internal and 4.3 Gb external memory.

For the HMD two images must be generated. To display these images in the two screens of the HMD the system is extended with a *Multi-Channel Option* (MCO, see also SGI). This system reads the frame-buffer and splits it into two parts, both with a resolution of 640x480 pixels, which are then displayed in the HMD at a resolution of about 244x230 pixels. This is done at 30 Hz (interlaced), which is about the minimum frame-rate that is acceptable.

2.5 Software

Programming a virtual reality system is a complex task. It involves a number of unrelated fields, which must be combined into a single program, preferably in a well-organised way. These fields include:

Simulation Having objects behave or react in a certain way to input by the user is a field known as *simulation*. Objects must react to the various input commands the user can give — such as mouse or keyboard input, but also tracking information — as well as perform some tasks of their own (one could think of the bouncing of a ball).

Playing a sound effect as a reaction to an action is for example one of the things that is handled in the simulation.

3-D graphics The simulation acts on the internal structures of the underlying model of the virtual world. Creating images of such a model from a certain viewpoint, using techniques like Z-buffering, texture mapping and shading, with the right parallax when using stereoscopic viewing, requires knowledge from the field of computer graphics.

User-interface When one is programming a 'desktop virtual world' (i.e. a virtual world in a window, also called 'fish tank virtual reality', see (Ware, Arthur & Booth 1993)) the user-interface is an important aspect of the design of the program. In a way, a virtual environment can be considered as a very advanced user-interface.

Modelling Since the objects in a common virtual world are not trivial, it should be possible to build such an object inside the program. The program should at least be capable of loading object descriptions created by other 3-D modellers.

Problem domain Virtual reality is applied in a lot of fields that have no intrinsic relation with VR. These fields include architecture, chemistry, scientific visualisation, entertainment, etc. Virtual reality is merely used as an advanced interface to get a better look at the problem or just for fun.

It is very difficult to actually deal with all (and possibly more) fields at one time. Therefore, it is advisable to implement the techniques of certain fields into a software library and use them a number of times.

One of these libraries is the WorldToolKit, developed by a company called *Sense8*. WorldToolKit is a software library of over 900 C-functions in which the techniques from the fields of simulation, 3-D graphics, modelling, user-interfacing etc. are implemented. The functions in the library are grouped into classes and are object-oriented in their naming convention. Classes include for example the Universe, in which the simulation is handled, Geometries, Sensors, Lights and others. There is also a C++ wrapper library which implements a (object-oriented) C++ binding of the functions.[2] Information about the WorldToolKit can be found in (Sense8 RM 1996), (Sense8 C++ 1996) and (Sense8 HG 1996). In Appendix. B an introduction in operating the VR system is given, which includes an introduction in WTK.

[2]This library at this point is still a beta-version, but most of it works fine.

2.5.1 The simulation loop

The heart of a WTK application is the simulation loop. In this loop a number of actions are performed. These are (in sequence):

1. Read new sensory input.
2. Execute the 'action function'.
3. Objects that are linked to sensors are updated according to the new sensory input.
4. Objects perform a task.
5. The virtual world is rendered.

It is also possible to record the actions that are performed in the simulation and play these back at a later time. As one can see, a lot of aspects of the simulation and almost all of the graphics have been taken out of the hands of the programmer. The functionality offered by this library makes programming a virtual reality application much easier.

2.5.2 The scene graph

Creating a virtual world in WTK is done by assembling the various parts that have to be rendered in a structure that is called the *scene graph*. This graph is a hierarchal arrangement of nodes which describe the scene that is to be rendered. This graph is rendered in depth-first order. A number of different types of nodes are available:

Geometry Nodes of this type are the visible objects on the display.

Transformation These nodes affect the position/orientation of the nodes that are processed after this node.

Separator Separator nodes separate the position/orientation information of their children from the rest of the scene graph.

Group When one wants to treat a number of geometry nodes as one geometry, one can group them by adding them as children of a group node.

Switch This node type makes it possible to render only one of a number of children at one time. Which one of the children is rendered can be controlled at run-time.

Light Light nodes control the light intensity of the scene in the part of the scene graph where this node is found.

There are also some other types of nodes available, but these are not important for this project.

By structuring the model of the virtual world in this way it is possible to have an intuitive understanding of the structure of the model and still being able to use the graphics hardware is the most efficient way.

3 Interaction in a virtual environment

In the previous chapter it has already been stated that the ability to interact with objects in a virtual world is an important aspect of virtual reality systems. Due to the design of the virtual world, this interaction is by nature three-dimensional. (Actually it is six-dimensional, since it involves positioning with three degrees of freedom, but also orientation of the objects with three degrees of freedom.) Classical input devices like a mouse, a light pen or a joystick are restricted, however, to a two-dimensional plane.

This chapter discusses some methods for performing 3-D interaction with a classical 2-D input device. We then present some three- or higher-dimensional input devices, which have been developed especially for three-dimensional interaction. This makes them very suitable for application in virtual reality.

To conclude this chapter, some remarks are made about issues found in literature that are important for creating a intuitive and convincing virtual environment.

3.1 Mouse-based interaction

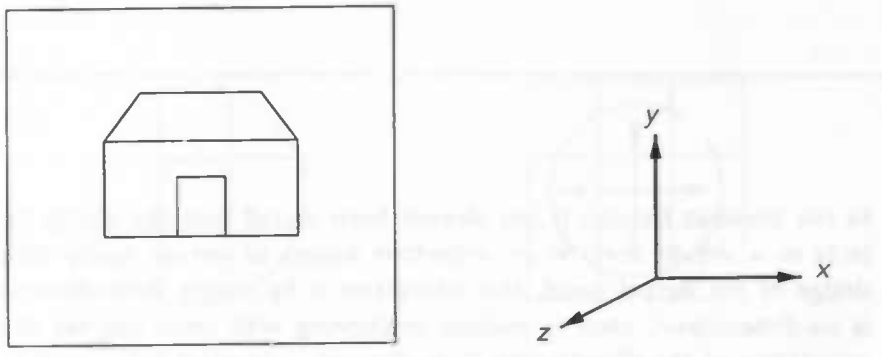
The mouse is one of the mostly used input devices for contemporary computers. It is placed on a flat surface, like a desk, and moved across it to operate a cursor that is displayed on the computer screen. This provides the user with two degrees of freedom of movement. One, two or three buttons are fixed on it that can be used to trigger actions. For most of the programs running on graphical workstations the mouse provides an accurate selecting, dragging and pointing facility.

An important point that can be made here is that users can relax their arm while operating a mouse. This point has turned out to be a crucial one. It may explain why input devices like a light pen, that require the user to keep an arm stretched out, have not received great popularity.

Using a 2-D mouse to manipulate 3-D objects in a 3-D space is not at all straightforward. A number of techniques have been developed to augment the three-dimensional rotation and translation of objects into actions that can be performed using a two-dimensional mouse.

3.1.1 Virtual controllers

Chen, Mountford & Sellen (1988) have investigated ways of using the mouse for performing translation, rotation and sizing operations on 3-D objects. In their article direct rotation using a mouse is discussed. They describe and evaluate four 'virtual controllers'. For the evaluation of these controllers they perform rotation operations on a simple model of a house, which is displayed in Fig. 3.1(a).[1] All rotations are performed with respect to the user's frame of reference, depicted in Fig. 3.1(b).



(a) (b)

Figure 3.1 (a) Object that is used to evaluate the controllers. (b) Coordinate system used in the 'virtual controller' study.

Graphical Sliders

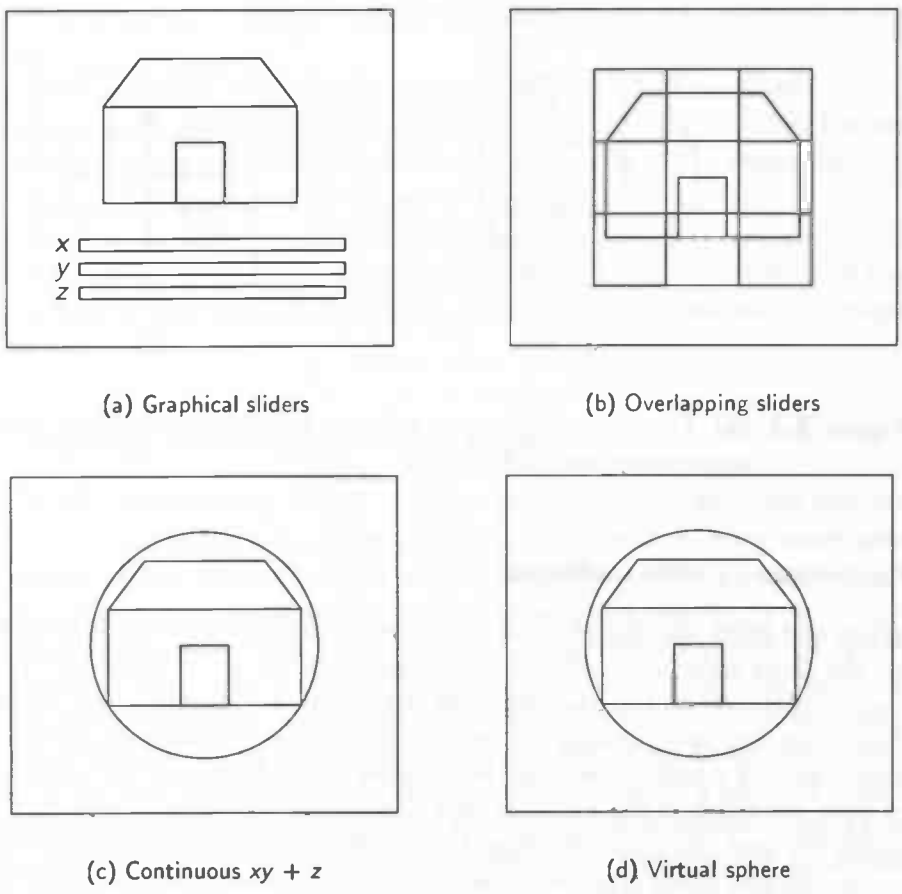
The first controller presented is the *Graphical Sliders Controller* (see Fig. 3.2(a)). This controller consists of three sliders, one for each axis. These sliders are placed horizontally below the object to be rotated. One can rotate the object by depressing the mouse button inside the slider that corresponds to the axis around which one wants to rotate, then moving the mouse horizontally and subsequently releasing the mouse button. The amount of rotation is proportional to the amount of horizontal translation of the mouse while keeping the mouse button depressed. A full sweep across one of the sliders corresponds to 180 degrees of rotation around the corresponding axis.

This controller is easy to understand, but one can only rotate the object around one axis at one time. It is included in the study by Chen et al. mainly as reference point for performance comparison.

Overlapping Sliders

The second controller is the *Overlapping Sliders Controller* (see Fig. 3.2(b)). In this controller the x-, y- and z-axes are represented by a vertical, horizontal and circular slider,

[1] The figures presented are exactly as in the article (Chen et al. 1988).



(a) Graphical sliders

(b) Overlapping sliders

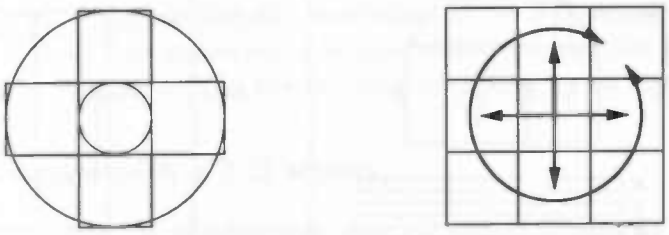
(c) Continuous $xy + z$

(d) Virtual sphere

Figure 3.2 Screen displays of the four virtual controllers with object in centre.

respectively (see Fig. 3.3(a)). These sliders are overlapped and simplified to look like a nine-square grid (see Fig. 3.3(b)), that is superimposed on the object to be rotated. One can rotate the object around its vertical axis (y-axis) by moving the mouse horizontally inside the middle row, with the mouse button depressed. Rotation around the x-axis is performed in the same way with the middle column. Rotation around the z-axis is done by making a circular movement in the outside squares. These movements are also displayed in Fig. 3.3(b). Movements other than these three are ignored.

With this controller, it is still only possible to rotate around one axis at a time. The difference with the conventional sliders, however, is that users feel they are more directly manipulating the object.



(a) (b)

Figure 3.3 (a) The three overlapped sliders. (b) Recognised user movements in the *overlapping sliders controller.*

Continuous xy with Additional z

When one takes the idea of the overlapping sliders controller one step further, one comes to the third controller. This is the *Continuous* xy *with Additional* z *Controller* (see Fig. 3.2(c)). When the user depresses the mouse button inside the circle, left-and-right movement and up-and-down movement of the mouse corresponds to rotation around the y-axis and the x-axis respectively. Moving the mouse diagonally will result in a combination of both rotations. If the mouse button is depressed while the mouse cursor is outside the circle, the user can rotate the object about the z-axis, by going around the circle.

In this way either arbitrary rotation in the xy-plane, or exact rotation about the z-axis is possible. This controller could therefore be described as a 2+1-D controller.

Virtual Sphere

The last of the four presented controllers, called the *Virtual Sphere Controller*, is depicted in Fig. 3.2(d). Although the controller has the same appearance as the previous one, the idea behind it is different. In this controller the object is thought to be fixed inside a glass sphere. Rotating the object is now a question of rolling the sphere (and therefore the object) with the mouse. Up-and-down and left-and-right movement at the centre of the circle corresponds to rotation around the x-axis and the y-axis, respectively. Movement along the edge of the circle is equivalent to rolling the sphere at the edge and produces rotation about z.

In the *Continuous* xy *with Additional* z *Controller* the mouse cursor must be outside the circle for rotation around the z-axis and inside it for rotation about the other two axes. With the Virtual Sphere Controller it is possible to rotate around all three axes without having to move the mouse outside the sphere. The report by Chen et al. states that this makes the Virtual Sphere Controller the most intuitive of the four controllers to use.

Although the presented controllers give a nice way of rotating a 3-D object in a 3-D space, the question rises of how much value these controllers have. Beside rotation, the operations translation and scaling are very important for most applications. One could implement these by using one button for the rotation operation and other buttons for translation and scaling. A similar model for translating objects in a 3-D space should then be created.

As second problem is that in most applications rotation of one object is not enough. When one wants to rotate a number of objects inside a scene, one has the problem of how to differentiate between rotating the entire scene, one of the individual objects, or a number of objects with respect to the rest of the scene.

One final drawback of this method is that superimposing the controller on the object to be rotated, is possible in some applications — provided that the controller is transparent enough to keep a good view of the object — but for some applications it might not be desirable.

3.1.2 Applying narrative handles to objects

Another study, performed by Houde (1992), considers both translation and rotation of objects. A user interacts with a three-dimensional scene of a living room using a one button mouse. When the user selects one of the objects inside the room (e.g. a chair, a lamp or a picture), a bounding box appears around this object. To indicate what the possible operations on the object are, a number of handles are placed on specific places on the bounding box, with a hand attached to it that indicates the operation that is performed when selecting it. In the lower corners of its side-planes the bounding box has four of these handles. They can be used to rotate the object about its y-axis. On the top-plane of the box another handle is placed, with an image of a grasping hand attached to it. This handle can be used to translate the object perpendicular to the xz-plane (i.e. along the y-axis). Sliding the object in the xz-plane is accomplished by depressing the mouse button somewhere in the bounding box, except on the handles. Originally there were also handles for this, but users tended to ignore them, so they were removed.

The interaction method described in this study has obvious drawbacks. Although it provides a very nice way to rotate objects that have a natural 'upright position' like chairs and lamps, other rotations (e.g. about the x- or z-axis) are not possible. One of the outcomes of the study was, however, that this actually facilitated interaction with the environment. Firstly, because the objects were not supposed to be rotated in that way — one of the test users said: "I don't mind not being able to rotate the chair [around the x- or z-axis], because it *is* a chair."; secondly, it facilitated interaction because of the reduced number of degrees of freedom. This argues for manipulating objects in a 3-D world by making repetitive operations with not too many degrees of freedom.

3.2 Interaction using a 3-D mouse

As we have seen there is a number of ways to use the mouse for manipulating objects in a three-dimensional space. They restrict interaction to a sequence of operations with fewer degrees of freedom. They also require mode shifts in order to switch between rotation and translation operations. To improve on this, tracking technology has been developed that addresses this problem more efficiently. One of the devices in which this technology is used, is what we call the *3-D mouse*.

A 3-D mouse can be seen as a direct extension of a conventional desktop mouse to three dimensions.[2] Whereas the conventional mouse is placed on a flat surface and moved across it to specify a particular point, with the 3-D mouse this is done by holding the mouse at a specific point in the air. A receiver of the tracking system is fixed inside the 3-D mouse. With it the position and orientation of the 3-D mouse can be calculated (see also Section 2.2). A 3-D mouse has a number of buttons, just like the regular mouse, which can be used to trigger actions like selecting or picking up an object.

3.2.1 A 3-D mouse with a 2-D screen

A study that investigates the effectiveness of a 3-D mouse is done by Ware & Jessome (1988). They tested a 3-D mouse, which they call a *bat*, by manipulating a hierarchical scene of objects that is displayed on a standard computer screen. The whole scene can be manipulated by selecting the top-most object of the hierarchy, and translating or rotating it. When another object is selected, all objects in the subtree starting at that object are moved with respect to the rest of the objects in the scene.

On the screen a cursor is displayed. Movement of the bat in the *xy*-plane causes this cursor to move correspondingly on the screen — one could think of the *xy*-plane as a vertical version of the surface on which a conventional mouse is moved. Selecting an object is now done by moving the cursor over it on the computer screen and pressing the (only) button.

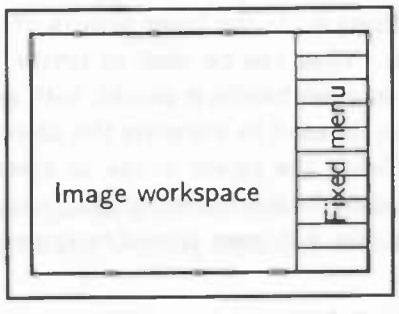


Figure 3.4 Screen layout used for evaluating the *bat*.

The user is allowed to manipulate the object in the scene in a number of interaction modes, which are selected from a fixed menu (see Fig. 3.4). These are:

- Full 6-D interaction consisting of all translations and all rotations.

[2] Again, this should actually be six dimensions.

- 3-D interaction consisting of all translations.
- 1-D interaction consisting of translation along one of the three axes.
- 3-D interaction consisting of all rotations.
- 1-D interaction consisting of rotation around one of the three axes.

The full 6-D interaction mode is reported to be the most useful for initial object placement, while some subset of the manipulations is used for precise placement.

As mentioned earlier, displaying a three-dimensional scene on a two-dimensional display causes loss of information. To regain some of this information, three special manipulation modes are suggested. These are:

Auto-rotate In this mode, the three-dimensional scene that is projected on the two-dimensional screen rotates about the vertical axis, oscillating through 90°. By doing this, the displayed scene strongly appears three-dimensional. This phenomenon is called *kinetic depth*. Although the scene is rotating, one can still perform movement operations. Ware & Jessome state that approximate object placement is possible in this mode. For precise placement it is however necessary to stop the scene from rotating. This mode is most useful for having a relaxed look at the scene.

Ninety-degree flip When an object has a correct *xy*-placement, the user can flip the scene over 90° and then perform *xz*-placement. So, in this way placing an object in a 3-D space is done by two times placing it on a 2-D plane. This mode is stated to be the most effective.

Dual mode One way to visualise the whole scene is by picking it up using the bat and rotating it freely. In *dual mode*, the rotational movement of the bat is used for this. Translational movement is at the same time used for object placement. This mode, however, is reported to be very confusing, partly because rotating the bat inevitably causes unintended translation.

The conclusions of this study are that object placement using the bat is a trivial task, which is quickly learned. Ware & Jessome adjudge this fact mostly to the kinetic correspondence between hand and object movement. Addressing the problem of arm fatigue, it is stated that this is not a problem when using the bat, because it operates on relative motion. It can therefore be held relaxed at waist level, or one can rest one's arm on the arm of a chair during interaction.

3.2.2 A 3-D mouse with a '3-D screen'

In the previously discussed study, the displaying of the scene is done on a standard computer screen. The reason for this is that at the time of writing (1988) there were no powerful enough graphics workstations for providing the necessary quality of images at the required speed for an immersive system. Also, Ware & Jessome believed "that for most applications there is little point in placing the user['s limbs] in the graphics environment." This decision requires that they devise a way for specifying different viewpoints for letting the user watch the scene from different angles. This requirement is conveniently

circumvented by letting the user rotate the entire scene — by selecting the top-most object and rotating it — instead of specifying a new viewpoint.

In the years that have passed since, the computational power of computers has increased significantly. The problem of not being able to create realistic images fast enough no longer exists. Also, it is not possible to require of 3-D applications that they always structure their objects hierarchically. We would therefore like to discuss a method of manipulating objects in a 3-D space using a 3-D mouse, displaying the scene in a head-mounted display, with stereo vision.

In the discussed study, the 3-D mouse (or bat) is actually used as a 2-D input device as long as no selection is made. A cursor is moved over the computer screen in correspondence to movements of the bat in the *xy*-plane. This means that when an object is completely occluded by (an)other object(s) — i.e. it lies behind other objects when looking along the *z*-axis — the whole scene has to be rotated before that object can be selected. Once a selection is made, manipulating the object can only be done for approximate object placement (in Auto-rotate mode), or manipulation is done by consecutive manipulations in a 2-D plane (in Ninety-degree flip mode). The method we are suggesting uses not just one tracking sensor, but two: one for the 3-D mouse, and the other to track the user's head movements.

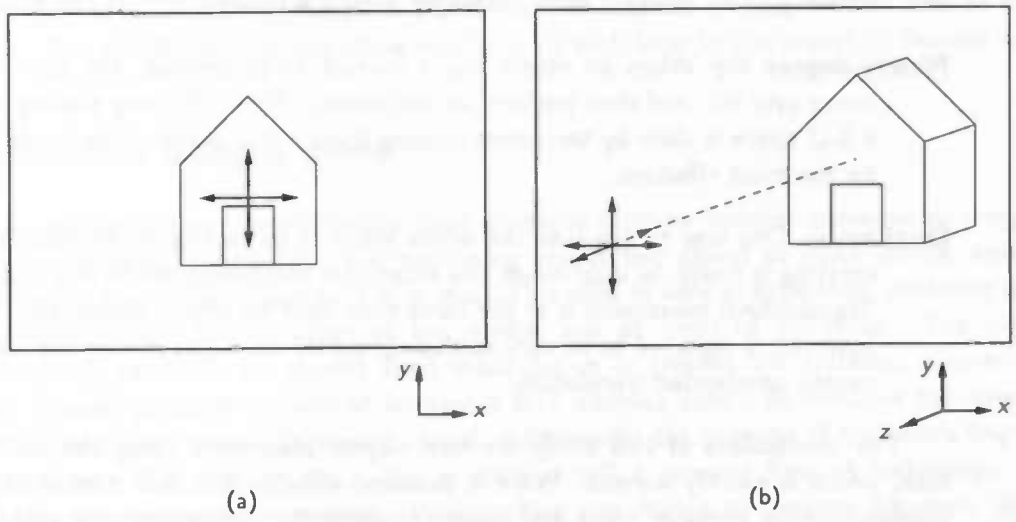


Figure 3.5 Two views of the scene with different viewing directions. (a) Viewing direction is along the *z*-axis. (b) Viewing direction not along one of the axes.

The scene is initially displayed as in Ware & Jessome (1988). The viewpoint and position of the cursor are arranged in such a way that selecting an object is done by moving the cursor over the object (see Fig. 3.5(a)). This is still done by moving the 3-D mouse in the *xy*-plane of its own frame of reference. Instead of keeping the viewpoint stationary — which inevitably means having to rotate the entire scene now and then — we let both the position of the viewpoint and the viewing direction be dependent on the tracker that is fixed on the user's head. One could now move one's head in such a way that the scene is displayed in the HMD as shown in Fig. 3.5(b).

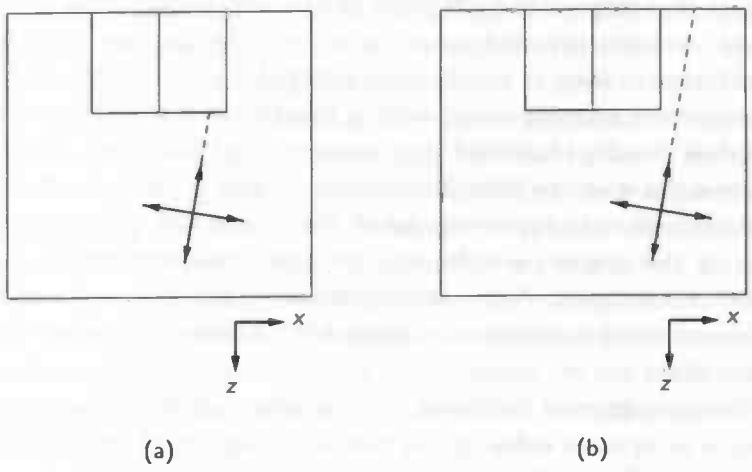


Figure 3.6 Same scene as in Fig. 3.5, seen from above. (a) The cursor lies 'over' the object, so the broken line stops at the intersection. (b) The cursor does not lie 'over' the object, so the broken line extends toward infinity.

This method seems to solve the problem of specifying the viewpoint, but a new problem arises. Provided that the various parameters involved in using stereo vision are set correctly, the user gets a fairly good impression of depth in the HMD. It can be very difficult, however, to see if the cursor lies 'over' an object or not, e.g. when the z-axis in the viewpoint's frame of reference is perpendicular to the z-axis in the frame of reference of the 3-D mouse. This is illustrated in Fig. 3.6. To solve this problem, the broken line in the figures will actually be present in the virtual world as a *pointing ray*. If the cursor intersects an object, the ray stops there. If it does not intersect any object, the ray extends to (virtual) infinity. This provides a good way of determining whether or not the cursor points to the object.

Now that we have extended the cursor to have a pointing ray, we can drop the restriction of using the 3-D mouse as a 2-D input device before having made a selection. In the study by Ware & Jessome, the z-axis of the frame of reference of the 3-D mouse and that of the viewpoint are aligned. This means that, when we move the 3-D mouse along the z-axis, the change in depth is hard to see, since in perspective projection only the scale of the cursor changes. This is no longer true when both z-axes are not aligned, which is possible in our approach by rotating one's head but not the 3-D mouse. Instead of rotating one's head one could also rotate the 3-D mouse slightly, so that the pointing ray becomes visible. Because the cursor now has this pointing ray, one can judge visually which object one is about to select. In this way, one could select an object 'from a distance' by pointing the ray at it, picking it up, moving it for some distance by rotating the 3-D mouse, and releasing it. As one can see, moving an object can thus be done using only wrist movements. During this time the user can relax his arm on the arm of a chair. Therefore the problem of arm fatigue is absent in this method.

We have seen that all movements of the 3-D mouse are directly translated into corresponding movements of the 3-D cursor in the virtual world. Because we have succeeded in applying a kinetic correspondence in all six dimensions between mouse and cursor, we

expect that this method will prove to be a very intuitive way of interacting with the virtual world. Another approach would be to have the discussed correspondence between mouse and cursor as long as no object is selected, but to transfer the correspondence from the cursor to the selected object when a selection is made. The movements of the 3-D mouse are then directly translated into movements of the object. Note that this is the way it is done in the study by Ware & Jessome. Having a correspondence between the 3-D mouse and the selected object, instead of the mouse and the cursor, basically comes down to moving the centre point for rotation from the centre point of the cursor to the centre point of the object. Approximately the same thing, however, can be done by selecting the object with the cursor near by, so that the difference between both centre points is almost negligible.

In the discussed literature, one important aspect of the presented methods is the fact that in most cases reducing the number of degrees of freedom improves precise placement of objects. One of the ways of using our method is manipulating an object from a distance. This, however, disables the possibility of very precise manipulation. The other way is that of manipulating an object, selected from near by, which requires the user to raise an arm. An inevitable consequence of this is that the arm starts trembling, thus disabling precise manipulation. We think that it is therefore necessary to introduce some modes in which the user can perform exact object placement. In these modes the possible manipulations are restricted to either translation in a 2-D plane or rotation around the two axes of this plane. The specification of the plane will be dealt with later in this report in Section 4.2.

3.3 Interaction using a glove

When people think of virtual reality, they generally seem to imagine someone wearing a head-mounted display, a body suit and using one or two gloves as input device, seeing and feeling the environment as if it is real. This view is very exaggerated, probably due to misleading information given by the media, and all kinds of TV-series. The glove is, however, probably the mostly used input device in present VR systems. Glove-like input devices generally consist of at least a 6-D tracking sensor to measure the overall position and orientation of the user's hand. Additionally the position of the user's fingers is measured. The way in which this is done is still a growing field of technology, so different gloves use different ways. When taking one degree of freedom for every joint in every finger, one comes to a total of $6 + 5 * 3 = 21$ degrees of freedom. However, since the joints are not really independent — it is very hard to move the upper two joints independently — we could reduce this number to something like 16. As one can see, this is still a very large number, and in current applications probably too large.

In Brijs (1992) the glove is used as an input device to model objects in a virtual environment. Different modes of interaction are performed by selecting one of a number of tools, e.g. a pair of scissors, a stapler, etc., that is then used to perform an operation on an object. A tool is selected by bringing an image of the user's hand in the virtual world sufficiently close to the representation of that tool. If it is close enough, a sound identifying the tool is played. By making a fist with the hand, the tool is picked up from the tool pallet, which can then be moved around. Opening the hand near the tool pallet causes the tool to be put back onto it. Although this seem natural movements when comparing them to a real world situation, it requires that the user physically moves the

hand towards the tool pallet and that he makes a fist. This is more complicated than e.g. pointing a cursor at the tool and pressing a button, both for the user, and with respect to the required software, that must recognise the action of making a fist. The user can operate the tool on an object, by first selecting the tool as described. When the hand is moved far enough from the pallet, the user can open his hand without 'losing' the tool. When he places the hand, holding the tool, close to the object and again makes a fist, the operation represented by the tool is performed on the object. We can observe two things here. The first is that the operation is started by making a fist. In contrast to picking up a tool, this is not according to how this is done in reality and it is therefore not obvious that this is an intuitive action. The second remark is, that the user is required to physically move his arm from the tool pallet to the object, in the mean time making a fist, or opening his hand. One can imagine that doing this over an extended period of time will become very fatiguing.

Making a fist is an example of something that is generally accepted as the best way to issue commands with a glove-like device, namely the issuing of commands by making *gestures*. In Bryson & Levit (1991), probably one of the best known projects in which VR is used for scientific visualisation, a VPL Data-glove is used to interact with the environment. In this project the interaction consists of moving (rakes of) seed points to new positions, placing new seed points or deleting existing ones. The way to issue these commands is also done by making gestures.

In our opinion, using gestures as the basic way of interaction with the environment has a number of drawbacks. Firstly, a way must be devised to recognise the various gestures from the input signals coming from the glove. This is mostly done using neural networks. A consequence of this is that it takes some time to compute the outcome of the network, which could increase the system lag, thus reducing intuitiveness. Secondly, the glove has to be recalibrated when it is used by different users. Lastly, and maybe most importantly, the users have to learn a number of different gestures, that might or might not be easy to reproduce. For users that are accustomed to operating a mouse, making gestures is clearly more complicated than pressing one of the buttons of a 3-D mouse.

3.4 Advanced input devices

To improve on the use of the glove with gestures, one should try to design a method of interaction that is as natural as possible, e.g. picking up an object by grabbing it at an appropriate place, such as a handle. Efforts going into this direction are reported in Figueiredo, Böhm & Teixeira (1993). For this to become really natural, however, one should be able to feel if one is touching an object or not. Devices that are capable of displaying force feedback are called *haptic displays* or *haptic devices*. One well known example is the Grope project (Brooks, Ouh-Young, Batter & Kilpatrick 1990). In this project a haptic display called *Argonne Remote Manipulator* (ARM, see Fig. 3.7) is used for examining the effectiveness of force feedback in a 3-D application. The operations in the program used for testing come from the field of molecular docking. Because these devices are only sporadically available, the best way of interacting in an virtual environment with these devices is not clear. One could imagine that techniques that are useful for the 3-D mouse are applicable for the ARM too. The question remains, however, which forces are to be displayed, and how this should be done. Because of these open issues, and because these devices are still very new, we will not discuss them further.

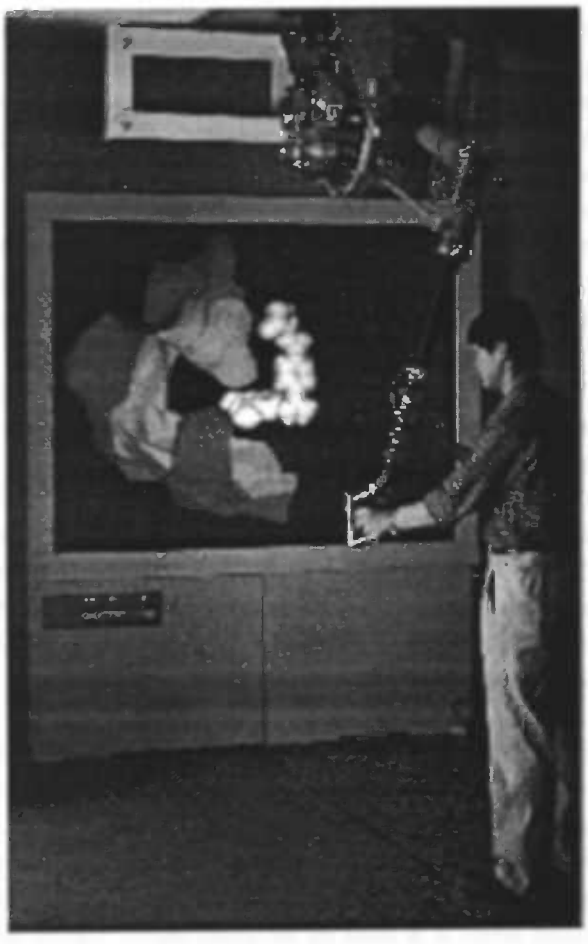


Figure 3.7 The *Argonne Remote Manipulator* (ARM) used in the Grope project at the University of North Carolina.

3.5 Design issues in Virtual Environments

As we can see from the discussed studies, creating a method of interaction for virtual worlds is not a simple task. A lot of research should and will be done to improve the existing methods. Although this chapter is about interaction methods, a few words must be said about issues that may help in designing an easy to learn and easy to use virtual environment.

3.5.1 Constraints

As remarked several times, one aspect in making interaction easier is that of applying constraints. This can be applied even stronger than we discussed before. As a first example, it is important that a virtual environment has boundaries that define the space in which the user can move around. Inside this space the user is able to move freely, but he cannot go outside it. Although this might seem trivial, in Bowman & Hodges (1995) it is reported that in many VR applications they are not defined, leading to confusion or

even frustration because the user flies through e.g. the floor, and thus outside the work environment altogether.

A second way in which interaction can be made easier, is by constrained object manipulation, as we have seen already. Bowman & Hodges report that providing multiple — and thus redundant — methods for doing the same with different levels of constraint is helpful. The user can then choose which one is the best at a certain moment, using movement in all degrees of freedom for easy approximate manipulation, and movement in only few degrees of freedom for precise manipulation.

Constraints can also be applied to the way in which various tools or interaction modes are selected. Bowman & Hodges argue that pull down menus that 'stick' to the user's field of view, are a very good way to select tools or modes with. Firstly, pull down menus are two-dimensional, which means fewer degrees of freedom. Secondly, they give an overview of all possible commands that can be issued in the program. They also recommend combining pull down menus with voice recognition. Where pull down menus are not a direct way to issue commands, giving a spoken command is. It is however not a good idea to use only voice recognition as input device. The number of degrees of freedom in speech is very large, and it requires that the user has a vocabulary of valid commands. As we saw when we were looking at glove-like devices, both can form a problem. We can do something about the first, by ordering the valid commands in a menu structure, from which the user can pick commands. After some time the user will know some commands by heart, so he can then issue them directly by speaking. This automatically reduces the second problem somewhat, since the only voice commands that are accepted are the ones that are in the menus.

3.5.2 Environment

A point made by Brijs (1992) that agrees with placing boundaries in the environment, is that it is important to actually have an environment in which one performs the interaction. This provides the user with some orientation cues, that prevent him from getting lost, something that can easily happen in a badly designed virtual world. Applying textures to large planes in this environment facilitates estimating depth, especially when the user moves around, because in this way he can experience motion parallax, which provides a very strong depth cue.

3.5.3 Feedback

Because there are no natural constraints like gravity or solid objects in a virtual world, anything is possible. It is the responsibility of the designer to create these as he thinks is necessary. The users that are going to operate the program also have to know what is allowed and what is not. It is therefore very important, that whenever the state of the program changes, the user gets feedback indicating what the change was.

4 Molecular modelling

In the previous chapter, we have studied various ways of manipulating objects in a 3-D virtual environment. Based on this, we have developed a method of interacting in such an environment using a 3-D mouse as input device. This method is applied in a case study in the area chemistry. In this chapter we will describe the implementation of this case study on the virtual reality system.

4.1 Concepts of the design

Since the molecules that are examined in the field of chemistry are three-dimensional structures, this is one of the areas where virtual reality is commonly used. We will develop an application which is directed towards Molecular Modelling. In this application the laws of chemistry will not (yet) be respected, but nevertheless we will use words like *atom*, *bond* and *molecule*. For now, this is merely to facilitate the discussion.

4.1.1 Problem domain

We will now present the requirements that are placed on the program. First we will see how the various structures in the problem domain are represented in the virtual environment, and what the possible manipulations on these representations are.

Atoms

The application we are going to develop, consists of creating and adapting a *virtual model* (which will also be called *molecule*). This model is built out of building blocks; these are spheres which represent the *atoms*. The spheres have a colour and radius which is characteristic of the type of the atom they represent (one can think of hydrogen, carbon, etc.).

The operations that the user can perform on the atoms are:

- Creating a new atom of a specified type.
- Deleting an existing atom.
- Moving an atom, i.e. translating an atom inside the virtual environment.

Bonds

In the 'real world of chemistry', atoms that are within a certain range attract each other. As a result the atoms may form a bound state, called a *bond*. Such a bond between two atoms will exist in the application if some attraction-relation is satisfied. This relation is dependent on the type of the atoms and the distance between them. It will be graphically displayed with a cylinder, drawn from the centre of one atom to the centre of the other atom. The colour and the radius of the cylinder represent the 'type' of the bond and its strength, respectively.

When one is moving an atom close to another atom, the representation of the bond is automatically created as the attraction-relation is satisfied. If the atom is moved sufficiently far away, thus violating the attraction-relation, the representation of the bond is removed.

Moving an atom away from or closer to another atom has effect on the strength of the bond. Since the radius of the cylinder represents the strength of the bond, this also has effect on the size of the radius. The radius of the cylinder will be updated in real-time to create visual feedback for the position of the atom and the distance between the atom and the other atoms.

Creating and deleting bonds is automatically done by the program. This means that there are no user-handled operations to do this.

The molecule

Aside from adding and deleting atoms to and from the model of the molecule, which is done by creating new atoms and moving them toward the model or by moving an atom away from the model, respectively, it is possible to manipulate the entire model.

The following operations are possible:

- Making a copy of the molecule.
- Deleting the molecule.
- Moving the molecule, i.e. translating as well as rotating it inside the virtual environment.
- Scaling the molecule.

When one is moving the molecule, the structure of the molecule, i.e. the distances between the atoms, remains the same.

Groups

It is possible to select a number of atoms and treat them as a unit. This is called a *group*. Atoms that are to form the group are selected in such a way that there is only one bond connecting the group to the rest of the molecule. This presupposes that the molecule has a tree structure.

After a group of atoms has been selected, the following operations are possible:

- Copying the group.

- Deleting the group.
- Moving the group.

When one is moving the group, the atoms inside this group are moved with respect to the atoms outside the group. Any existing bonds between the group atoms and external atoms will be continuously updated to reflect changes in the bond strength and, if necessary, bonds are removed or added.

4.1.2 Virtual environment

In addition to requirements on the objects inside the environment, we also have requirements on the environment itself.

Environment

One of the common problems people experience when they are moving inside a virtual environment is that they tend to 'get lost'. Brijs (1992) has reported that it is therefore important to create a background environment in which one is situated. This environment should have textures applied to large planes, which facilitates estimating depth and provides cues for rotational motion. Adding some objects to the environment gives one the possibility of orienting oneself inside the environment. One does not have to be able to interact with this environment, it should just be present.

In the current application the environment consists of the following elements:

- A room in which the user is situated. The room is made out of a floor, four walls and a ceiling, which all have an appropriate texture applied to it. It is not possible to exit from this room.
- A working table. The model of the molecule is placed above this table. Since there is no gravity in the virtual world this is not necessary, but the table provides a strong orientation cue and also an extra depth cue for the structure of the molecule due to its texture.

There are also some objects in the environment which are not interactive in the sense that one can move them, but in the sense that they provide a means of interacting with the model. They are:

- A tool pallet. This is the place where the tools can be found.
- Some trash cans. One can use these to delete (parts of) models. Moving an object into a trash can and putting it down there causes the object to be deleted from the environment.

Tools

In Brijs (1992) the different operations are selected by picking a tool from a tool pallet, which is then held in the virtual hand. This tool represents the action that is going to take place. In the current program, this approach would not be completely satisfactory, because

the operations (selecting, creating/copying, moving and scaling) need to be performed on either one atom, a group of atoms or the whole molecule. Thus, one also has to indicate on which one of these three 'units' the operation must be performed. To do this, there exists the notion of *current unit of operation*, which is one of atom, group or molecule. One can change the current unit by selecting it from a pop-up menu.

A number of *modes* now exists, represented by the following tools:

- Moving mode: the tool that represents this mode is a three-dimensional cross. When a unit of operation is selected, one changes automatically into moving mode.
- Copying/moving mode: the tool that represents this mode is an augmented cross. When a unit is selected in this mode, a copy of this unit is made which is then moved as in the moving mode.
- Scaling mode: this mode is only possible for the entire molecule and is represented by a balloon.
- Unit-changing mode: this mode changes the *current unit of operation*. This is done by using a pop-up menu with three possible choices: *Atom*, *Group* and *Molecule*.
- Group defining mode: defining a number of atoms as a group is done in this mode.

The three-dimensional cursor

Interaction in the virtual world is done using a *3-D mouse* as input device. To reflect the position of the 3-D mouse and the direction in which it is oriented, a cursor is visible inside the environment. Selection is done via 'tele operation', e.g. when one issues a command to pick up an atom, a virtual ray is shot from the centre of the cursor in the direction the cursor is pointing to. The first object that this ray intersects is then selected (if it is selectable). This method is explained in more detail in section 3.2.2.

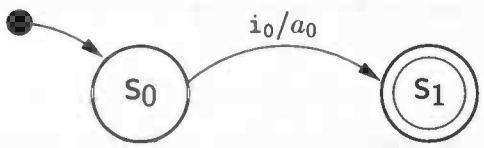
Because it is rather difficult to estimate the precise direction from the orientation of the cursor only, the ray that will be shot to select an object is constantly visible as a (three-dimensional) line. The ray emanates from the cursor, and stops at the first intersection with either an object or the environment.

4.2 Specification of the interaction

Now that we know what the various functional parts of the virtual world are, we can specify the way in which the 3-D mouse is used to perform interaction with the virtual world.

4.2.1 Selecting the tools

On the tool pallet there are five tools. When the pointing ray intersects one of the moving, copying or scaling tools and the selection button is depressed, the tool (and thus the corresponding mode) is selected. To reflect this, the cursor is changed into the three-dimensional icon representing this mode. In Fig. 4.1 a *finite state machine* (F.S.M.) illustrating this is presented. For an explanation of the notation used in the F.S.M.'s, see Appendix. A.



s_0	Ray on tool; no tool is selected
s_1	Tool is selected
i_0	Depressing and releasing selection button
a_0	Cursor is changed

Figure 4.1 F.S.M. for selecting a tool

4.2.2 Applying the tools

Once a tool is selected, the operation corresponding to that tool can be applied to either the atom, a group of atoms or the whole molecule (depending on the current mode of operation, see section 4.2.3). Applying an operation to an object is done in the same way as selecting a tool, by navigating the ray emerging from the cursor so that it intersects the object, and subsequently pressing the selection button.

The moving, copying and scaling tools are operated as follows.

Moving tool

After depressing the selection button, the movements of the input device (3-D mouse) are connected to the object; this means that when one is moving the mouse in a certain direction, the selected object moves accordingly inside the virtual world. Changes in orientation of the mouse cause the object to be rotated accordingly. Since the hand has limited freedom, it is also possible to rotate objects with a 'button' which is called the *hat* (see section 2.2). For very accurate positioning the hat is very effective.

After pressing the top button of the mouse, moving the hat left or right causes the object to rotate continuously around the y-axis (in the WTK axes system). Moving the hat up or down causes rotation around the z-axis. In this way precise rotation is possible, without having to break one's wrist. Exiting 'rotation mode' is done by pressing the top button a second time, or by pressing the bottom button. While using the hat in this way, movements of the mouse have no effect on the position of the selected unit.

To position an object accurately, the middle button is pressed. Just as in rotation mode, movements of the mouse have no effect. Moving the hat up or down causes translational movement along the y-axis. Moving the hat left or right causes the object to be moved in the xz-plane, perpendicular to the direction ray of the 3-D mouse. Precise positioning of objects, not interfered by vibrations of the hand is thus possible.

Deleting the molecule or a part of the molecule is also possible. This is done by selecting the part that has to be thrown away and moving it toward one of the trash cans that can be found within the environment. When the object comes within a certain distance from these trash cans (for example, such that their bounding boxes intersect) the object is coloured with a warning colour to indicate that one is about to throw it away. Releasing the selection button while the part is coloured in this way causes it to be deleted from the environment.

A finite state machine for the moving tool can be found in Fig. 4.2.

Copying tool

When one selects an object in copying mode, this object stays in place. A copy of this object is made (including bonds, if the selection was a group of atoms or the whole molecule). This copy can then be moved around as if it were selected by the moving tool in the first place. The finite state machine for the copying tool is therefore the same as the one for the moving tool, noting that instead of moving the selected object, a copy is made of it that is then moved.

One of the most elementary operations is adding new atoms to the model. This is done by copying an atom from a number of predefined atoms that can be found inside the virtual environment. Adding new atoms is therefore equivalent to copying one of these atoms and moving it to the model.

Scaling tool

In this mode it is only possible to scale the entire molecule, i.e. the distances between the atoms as well as the atoms themselves. This can be done in all operation modes by selecting an arbitrary atom. The centre of the selected atom will then become the centre around which the molecule is scaled.

Selecting an atom causes a bounding box to be displayed around the molecule. When moving the mouse away from the centre, the molecule is enlarged. This is indicated by a second bounding box that is representative of the size of the new molecule. Moving toward the centre has the opposite effect. When the molecule is scaled in the desired proportion, the selection button is released to actually scale the molecule.

The finite state machine for the scaling tool is displayed in Fig. 4.3.

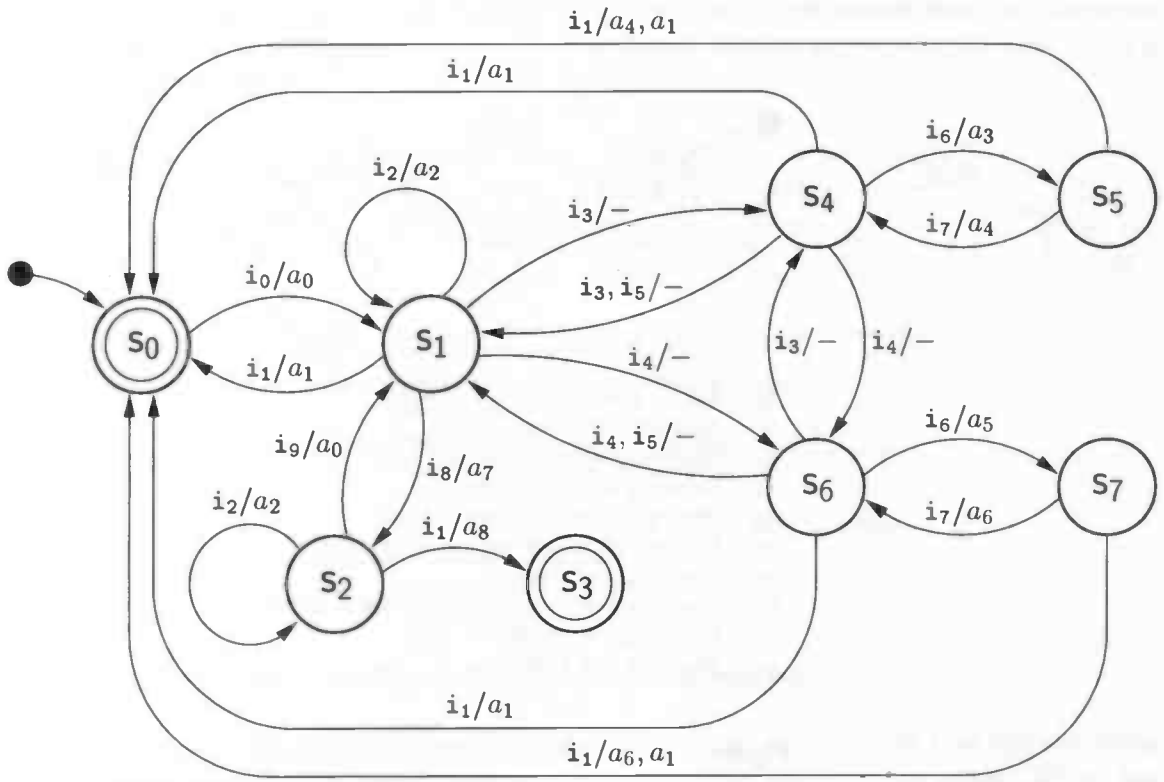
4.2.3 Changing current mode of operation

Interacting with the fourth tool (for changing the *current mode of operation*) goes as follows: selecting this tool, represented by 3-D push-button with the text *operation mode* on it, causes a pop-up menu to be displayed. This pop-up menu has three options: *Atom*, *Group* and *Molecule*. Selecting an option with the pointing ray and then releasing the selection button changes the current operation mode into the mode corresponding to the selected option. If the ray is moved off the pop-up menu, which means that no selection is (being) made, then the mode is unchanged. See Fig. 4.4.

4.2.4 Defining a group

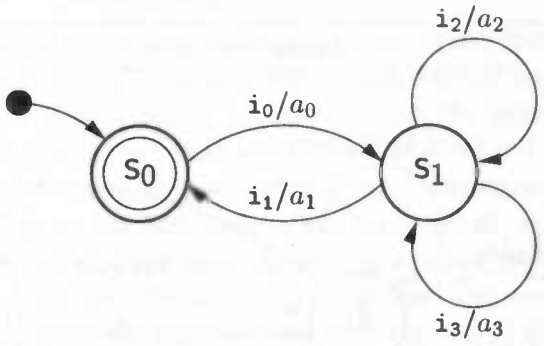
A number of atoms can be put together in a group, so that the moving, copying or deleting operations will apply to all atoms inside this group in the same way. This is where the fifth 'tool' comes in. After selecting a push-button, which is the representation of this tool, one is in *group defining mode*.

A group of atoms is defined as all atoms that are on one side of a particular bond. This can be half of the molecule, or (a part of) a side chain. Selecting a group is done by identifying that bond and first selecting the atom that *is not* to be part of the group and subsequently selecting an atom on the other side of the bond that *is* to be part of



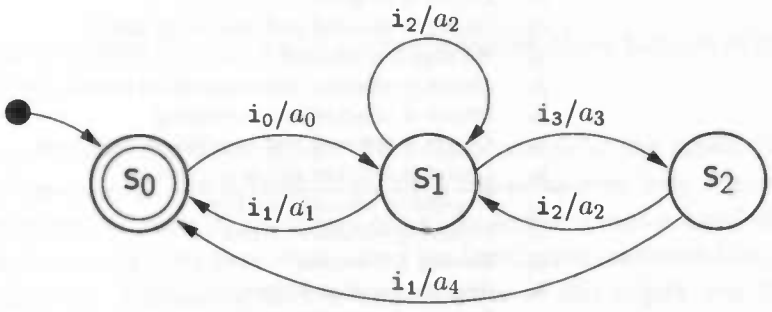
s_0	Ray is on object; no object is selected
s_1	Object is selected
s_2	Object is selected and near trash can
s_3	No object is selected
s_4	Object is selected; hat operates as rotator
s_5	Object is selected and is rotating
s_6	Object is selected; hat operates as translator
s_7	Object is selected and is translating
i_0	Depressing selection button
i_1	Releasing selection button
i_2	Moving 3-D mouse
i_3	Pressing rotation button
i_4	Pressing translation button
i_5	Pressing cancel button
i_6	Depressing hat
i_7	Releasing hat
i_8	Entering deletion range
i_9	Exiting deletion range
a_0	Object is coloured with selection colour
a_1	Object is coloured with normal colour
a_2	Object is moved; bonds are updated
a_3	Object starts rotating
a_4	Object stops rotating
a_5	Object starts translating
a_6	Object stops translating
a_7	Object is coloured with warning colour
a_8	Object is removed from scene

Figure 4.2 F.S.M. for moving an object



s_0	Ray on atom; no atom is selected
s_1	No atom is selected
i_0	Depressing selection button
i_1	Releasing selection button
i_2	Moving 3-D mouse toward centre
i_3	Moving 3-D mouse away from centre
a_0	Bounding box appears
a_1	Bounding box disappears, molecule is scaled
a_2	Bounding box shrinks
a_3	Bounding box grows

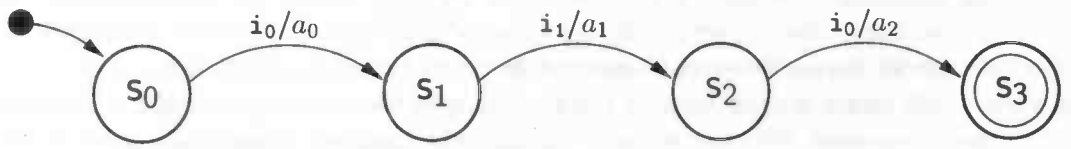
Figure 4.3 F.S.M. for scaling the molecule



s_0	Ray on push-button, no tool is selected
s_1	push-button is selected, ray on pop-up menu
s_2	push-button is selected, ray off pop-up menu
i_0	Depressing selection button
i_1	Releasing selection button
i_2	Moving 3-D mouse 'inside' pop-up menu
i_3	Moving 3-D mouse 'outside' pop-up menu
a_0	Pop-up menu appears, `mode` = current mode
a_1	Pop-up menu disappears `mode` = selection
a_2	`mode` is updated to selection
a_3	`mode` is current mode
a_4	Pop-up menu disappears, `mode` is unchanged

Figure 4.4 F.S.M. for changing the current mode of operation

the group. All atoms that are (transitively) connected to the second atom are contained in the newly formed group, stopping the transitivity relation at the selected bond. This is displayed in Fig. 4.5.



s_0	Ray on atom; no atom is selected
s_1	First atom is selected
s_2	Ray on second atom
s_3	Group defined
i_0	Depressing and releasing selection button
i_1	Moving 3-D mouse
a_0	Atom's bonds are coloured with 'indicating colour'
a_1	Pointing ray is updated
a_2	Group atoms are coloured with group colour

Figure 4.5 F.S.M. for defining a group

One can now perform operations on this group by selecting the tool corresponding to the desired operation and then selecting one of the atoms in the group. When one selects an atom that is not part of the group while in 'group mode', this individual atom is selected. Thus, for atoms not belonging to the group, there is no difference between 'atom mode' or 'group mode'.

4.3 Implementation

We pointed out in Section 2.5 that programming a virtual environment is a complex task, that involves a number of fields. We will now present an implementation of a part of the design as described above. In doing this, we will resolve the following issues:

- Representing the objects in the problem domain.
- Reacting in the specified way to user-input.
- Organising the representations for the objects both in the problem domain and in the visual context (i.e. the visual environment as discussed in section 3.5.2). This involves building and maintaining a scene graph.

Since a large part of this involves dealing with objects, we have chosen to use an object oriented language for the implementation. Because WTK only has bindings for C and C++, we will use the latter. For an introduction in C++ see Stroustrup (1991) or van Winkel & Willems (1993).

In Appendix. B an introduction to using the VR system is given, including an overview of interesting WTK-functions. We recommend that the information presented there is looked through before reading this section.

4.3.1 Initialisation

The application that we are developing makes use of the WorldToolKit. The first step in using WTK in any application is initialising the various internal structures of the library as discussed in Appendix B.3. Because we want to use the head-mounted display for stereoscopic viewing, we supply the values `WTDISPLAY_STEREO` and `WTWINDOW_NOBORDER` to the `WtUniverse::New()` method that performs the initialisation.

We insert a light node as the first node in the scene graph, so the whole scene will be illuminated. We then initialise the scene by loading a room, with inside it the working table, and putting the viewpoint in a convenient position. The next step is to open the sensors. We have incorporated this action, along with the rest of the functionality into a separate class, called `Actor`. We will discuss this class later. We now simply initialise this class by calling the method `Actor::Initialise()`.

Now that all initialisations are completed, we start the simulation by calling the methods `WtUniverse::Ready()` and `WtUniverse::Go()`. Note that we have not set an *action function*. This is because we will implement all activity of the program with *tasks*.

4.3.2 The `Actor` class

We already mentioned that all functionality of the application will be controlled by the `Actor` class. All attributes and methods of this class will be static, just as in the `WtUniverse` class. The reason for this is that pointers to these methods are stored in a table. This can only be done with static methods. Initialising the class with `Actor::Initialise()` causes a number of actions. These are:

- Opening the sensor through which the configuration of the buttons of the 3-D mouse is obtained.
- Creating an `EventHandler` object. This object will drive the application by polling the configuration of the buttons every time the simulation loop is executed. When the configuration changes, the object generates an event that is handled by passing it to an instance of a class that implements the finite state machines that are described in Section 4.2. Depending on the current state of this object and the new input symbol (i.e. the generated event) a transition is made, and the corresponding output symbol is generated. This output symbol is actually a call to a method of the `Actor` class that causes the desired change in internal and visual state. Pointers to these methods are stored in a table (just as the transitions). This is why the methods of the `Actor` class are defined as *static* methods.[1] The set of methods in the `Actor` class thus defines the functionality of the program.
- The tracking sensor on the HMD is opened and the viewpoint is connected to this sensor using a motion link (see Appendix B.3.9).
- The tracking sensor inside the 3-D mouse is opened. A 3-D cursor is loaded from file and is connected to the sensor with a motion link. This cursor has a pointing ray, as discussed in section 3.2.2. Later we will discuss this extensively.

[1] Note that to access attributes in static methods, these attributes must also be defined static.

- A graphical object that is used to toggle between molecule-, atom- and group-mode is placed in the scene .
- Another graphical object which is used to initiate the definition of a group is placed in the scene.

In the design we planned to make it possible to create molecules from scratch by copying atoms and arranging them into a molecule. For now we will create a molecule at initialisation and operate on this.

To be certain the tracking sensors are opened in the correct order, we have created a class called `Tracker`, that opens both sensors once and in the correct order. To obtain a pointer to either the HMD sensor or the 3-D mouse sensor, one can call `InitialiseTracker()`, supplying an argument that indicates in which of the two pointers one is interested.

4.3.3 Objects in the problem domain

The objects that are coming from the problem domain are the molecule, atoms, bonds and a possible group of atoms. The set of atoms together form the molecule. We can therefore implement the molecule simply by an attribute that is a set of atoms.

The atoms have a type, a flag that indicates whether or not the atom is part of the group and a set of bonds. They also have a pointer to the molecule they are part of.

A bond represents the fact that two atoms are close enough to satisfy the attraction-relation. We implement the bond by having references to the two atoms.

Graphical representation

In WTK the atoms will be represented with `WtMovGeometry` sphere objects. A pointer to the WTK-object will therefore also be included in every atom object. Creating this object is done in the classes *constructor*-function.

The attributes of the atom class are therefore:

```
class Atom
  type: atom type
  molecule: Molecule
  bonds: set of Bonds
  inGroup: boolean
  atomNode: WtMovGeometry
endclass
```

A bond is represented with a `WtMovGeometry` cylinder object that starts at the centre point of one of the atoms and ends in the other atom.

```
class Bond
  atom: set of two Atoms
  bondNode: WtMovGeometry
endclass
```

The molecule has no graphical representation. Its functionality is that of a container and this is implemented in WTK as a `WtGroup` object. Because the molecule can have

its own position and orientation independent of the rest of the scene, we are using the WtMovSep subclass.

The atoms that are in the group have a position and orientation with respect to the molecule as a group. We therefore insert an instance of the WtMoveSep class as a child of the molecule's group object to represent atoms that are inside the group. Note that if two atoms inside the group are close enough to have a bond, this bond will also be moved under the group's group object. The scene graph that includes the representation of these objects is depicted in Fig. 4.6.

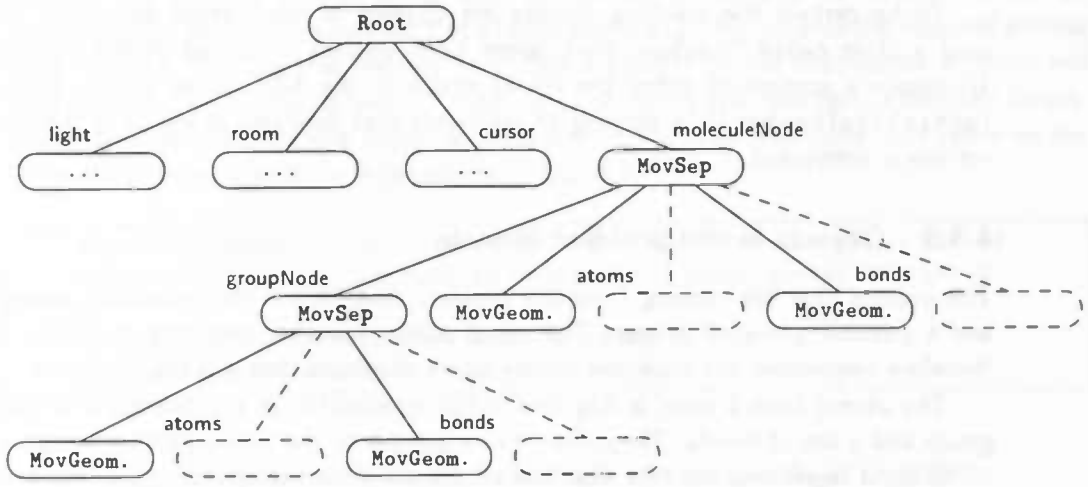


Figure 4.6 Scene graph in which representation of the objects from the problem domain is included.

When we move one of the molecule's atoms, bonds should be automatically inserted, removed or just updated. This is done using a *task*. The molecule class therefore also has a WtTask attribute that performs this action. This task is created in the molecule's constructor-function, but initially it is disabled. When an atom or the group of atoms is being moved, the task is temporarily enabled, until the atom is released again. The attributes of the molecule class are

```
class Molecule
  atoms: set of Atoms
  moleculeNode: WtMovSep
  groupNode: WtMovSep
  bondTask: WtTask
endclass
```

4.3.4 The 3-D cursor

We have inserted a number of graphical objects into the scene, namely the surrounding room (including the table), the atoms and bonds and the two objects for toggling the operation mode and defining the group. These objects can be selected by the user. For this purpose another graphical object has been inserted: the 3-D cursor. This cursor is

connected to the tracking sensor in the 3-D mouse. We discussed the necessity of a pointing ray for selecting objects in section 3.2.2. We insert this ray into the scene graph near the 3-D cursor. Therefore we have inserted a `WtMovSep` object as a child of the root node, which is actually connected to the sensor. Because the position/orientation information that is constantly updated to this object affects its entire subtree, all nodes that are inserted under it will also be updated with the new position/orientation of the mouse-sensor. The cursor and the pointing ray will therefore be inserted under the separator node. This is depicted in Fig. 4.7. One can see in this figure that other objects can also be inserted

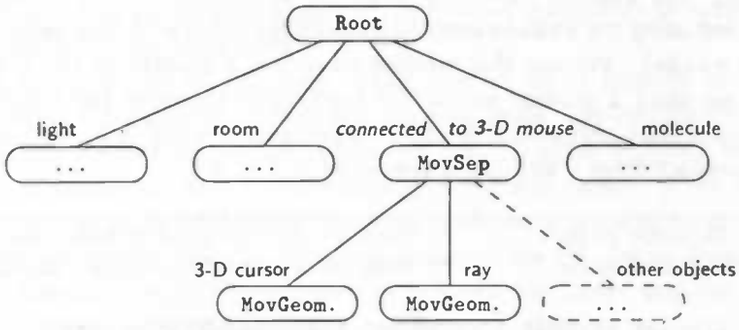


Figure 4.7 Scene graph in which the cursor and pointing ray are inserted.

under the `WtMovSep` node. In this way, one can e.g. move a selected atom, or a group of atoms around by simply relocating the corresponding node in the scene graph under this separator node. Note that this requires some repositioning to keep the node in the same position/orientation with respect to the world coordinate frame, when moving it from one local frame to another.

4.3.5 Selecting objects

The position of the 3-D cursor and the direction in which it is pointing is obtained each time the pointing ray is updated. This information is then also to determine which object one is currently pointing to. The `WtNode` method `RayIntersect()` is supplied with the direction and position of the cursor, and it stores a pointer to a node path that leads to the nearest object along this ray as well as the distance to this object.[2] Since the ray starts in the centre of the cursor, it is temporarily disabled. Otherwise, the cursor would be the nearest object,

For updating the pointing ray the distance to the nearest object is enough, so the node path will not be used. The action of updating the pointing ray is implemented as one of the cursor's tasks. In this task the existing ray is deleted, and a new one is created with a length that extends exactly to the nearest object.

When one is performing a selection, the node path is used. With this node path a pointer to the node of the selected object can be obtained. We are not interested in the node, however, but in the object of which the node is an attribute. E.g. we want to have a pointer to the atom-object, not to the sphere-object which is its visual representation.

[2]Sometimes when one selects an object, this node path is not created and the pointer that is returned is 0. This is probably a bug in WTK.

We could search through all objects and try to locate the returned pointer. The C-version of the library, however, gives an alternative. It is possible to store a pointer to an arbitrary structure (e.g. a pointer to an atom-object) in a WTK-node with the function `WTnode_setdata()`. This function takes a pointer to the node as the first parameter (note that this is a pointer to a `WTnode` of the C-binding) and a void pointer as the second parameter. The function `WTnode_getdata()` takes a pointer to a `WTnode` as the only parameter; it returns the void pointer that was stored in the node.

Since the C++-version of the WTK library only 'wraps' the C-version into a C++-binding, the `WTnode` pointer is available somewhere in the `WtNode` object. It can be obtained using the `WtBase` method `GetWTKStructure()`, that returns the required pointer (as a `void*`). We use this method to obtain a pointer to the C-structure `WTnode` and then we store a pointer to our C++-object in it, using the functions described above. The code that performs this action is presented in Fig. 4.8. The code that retrieves our C++-object from a `WtNode` is presented in Fig. 4.9.

```
Object* object;      // Pointer to the object that has to be stored
WtNode* node;        // Node in which the pointer should be stored

WTnode* wtkNode = (WTnode*) node->GetWTKStructure();
WTnode_setdata( wtkNode, (void*) object );
```

Figure 4.8 Code for storing a pointer to an object in the WTK-object.

```
WtNode* node;        // Node in which the pointer is stored

WTnode* wtkNode = (WTnode*) node->GetWTKStructure();
Object* object = (Object*) WTnode_getdata( wtkNode );
```

Figure 4.9 Code for retrieving the pointer stored in a WTK-object.

We have a number of different classes that have selectable representations: the room, the atoms, the bonds, the class for changing the operation mode and the class for defining a group. When we retrieve the pointer for objects from these classes, we do not know of which class the object is. Therefore these classes are subclasses of the class `GraphicalObject`, which has a `type` attribute that indicates the type of the object. In this way, we know that a pointer to a `GraphicalObject` is returned. By looking at its type, one can determine its correct subclass.

N.B. We mentioned above that a pointer to the selected node can be obtained using the node path that is created by `RayIntersect()`. This is done with the `WtNodePath` method `GetNode()`. When we use the procedure as we just described to store information in the C-structure, the method does not return a pointer to the selected `WtNode` from which a pointer to the desired object must be retrieved, but directly to the desired object. This is probably either a bug in the library or in the C++-compiler. However, we get what we wanted, so we will not complain about this.

4.3.6 Continuous translation and rotation

Let us have a look at the actions that have to be performed to implement the functionality of our application. A lot of the operations involve relocating nodes in the scene graph, e.g. to move an atom-node into the group, or to 'pick up' an atom. This requires some programming using the WTK-methods, but it is quite straightforward. We also have to update some of our own structures, such as the lists of bonds per atom. We have to enable or disable some of the tasks now and then such as updating the ray, which e.g. does not have to be done as long as a selection is being performed. There is one specific action that we have not yet discussed, that of continuous translation or rotation using the hat. We will briefly discuss them here.

When the hat is moved in 'fine-tune' mode, the action of rotation or translation is started. The action is finished when the hat is released again. We implement these actions using tasks. When the fine-tune mode is entered, the 2-D plane in which the fine-tuning will take place is calculated from the orientation of the 3-D cursor. If the hat is used, a moving action is started by enabling a task that performs a translation or rotation over some small amount in this plane in every execution of the simulation loop. During the time one is in fine-tune mode, the motion link between the cursor and the sensor of the 3-D mouse is disabled, so hand movements will have no effect.

4.4 Working with the application

In this section a short guide to operating the application in its current state of development is given.

Starting the application

When the application is started, the tracking sensors in the head-mounted display and the 3-D mouse are initialised. The position and orientation that the sensors have at the time of initialisation are going to correspond to the default state of the coordinate systems of the viewpoint and the cursor that will be connected to the sensors. It is therefore important that during the initialisation the HMD and the 3-D mouse are held in the position that one wants to operate in.

Selecting objects

After the initialisation is complete, a number of objects are present in the virtual environment, some of which can be selected by pointing the ray at them and pressing the *trigger button*. The objects are:

- The 3-D cursor, from which the pointing ray emerges. The pointing ray extends from the cursor to the first object that it intersects. This object is the one that is selected if the trigger button is depressed.
- The enclosing room and the working table. These objects belong to the visual context and are therefore not selectable. If one tries to select one of them, a short 'beep' is generated.

- The molecule, constructed of atoms and bonds, is situated above the table. The atoms be selected, but bonds cannot be selected. When one tries to select a bond, a short 'beep' is generated.
- On the left-hand side of the table a graphical text-object is placed for changing the operation mode. Initially the text in this object spells: "molecule mode".
- On the right-hand side of the table another graphical text-object is placed, that is used to initiate a group definition. The text therefore spells: "define group".

A screen shot of the application in which the objects are displayed is presented in Fig. 4.10.

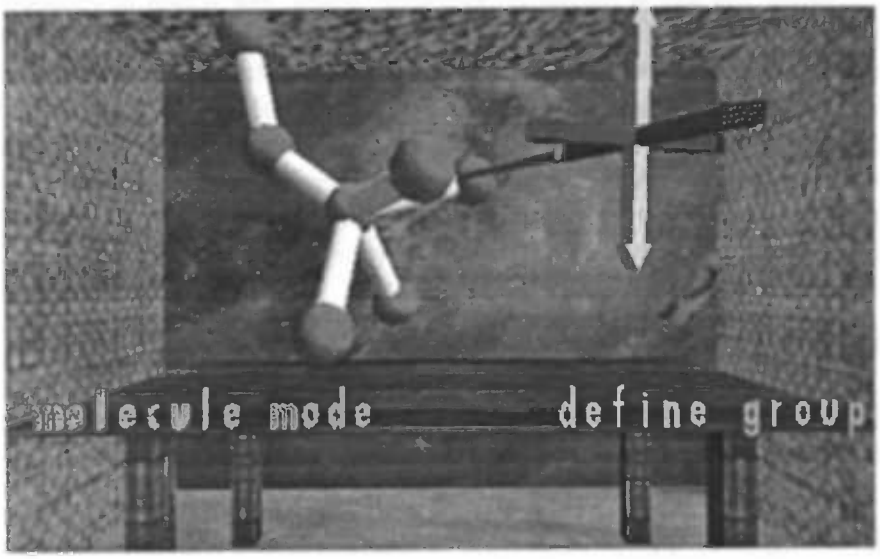


Figure 4.10 The virtual environment for a chemical application. It consists of a room with a work table, a molecule and a 3-D cursor.

Selecting an atom can either initiate a moving action of the whole molecule, a single atom or a group of atoms. This depends on the current mode of operation, that is indicated in the text of the graphical object to the left of the table. While the trigger button remains depressed, the selected part of the molecule will move according to the movements of the 3-D mouse. If the trigger button is released, the object no longer moves along with the 3-D mouse.

Continuous translation/rotation

During a selection of a part of the molecule, the object will follow the movements of the 3-D mouse. It is also possible, however, to fine-tune the placement of the object, by using the hat. To enter the fine-tune mode, one can press either the top button for continuous rotation, or the middle button for continuous translation, while keeping the trigger button depressed. This will remove the connection to the 3-D mouse; the object will therefore remain in its place in the environment. Pressing the same button for a second time will reconnect the object to the mouse. One can toggle between the two fine-tune modes

by pressing the corresponding button. Note that the connection will not be restored by toggling from translation to rotation or vice versa.

In fine-tune mode, the object can be translated or rotated in a 2-D plane with the hat. The axes of this plane are: the y-axis of the 3-D scene (i.e. the vertical axis) and the line in the xz-plane of the 3-D scene perpendicular to the pointing ray. Moving the hat left or right causes either translation along the horizontal axis, or rotation around the vertical axis. Moving the hat up or down causes translation along the vertical, or rotation around the horizontal axis.

Releasing the trigger button in one of the fine-tune modes ends the selection. It is therefore not necessary to exit the fine-tune mode before releasing the trigger button. That would probably involuntarily move the object somewhat after one has just carefully positioned it correctly.

Changing the operation mode

The current mode of operation can be one of "molecule mode", "atom mode" and "group mode". The text displayed on the left side of the table indicates the mode in which one is currently operating. Selecting the text with the pointing ray and pressing the trigger button causes a change of the current mode, cycling through the modes every time the text is selected. Note that the group mode is equivalent to the atom mode when no group has been defined, or if an atom not part of the group is selected.

Defining a group

On the right-hand side of the table the text "define group" is displayed. Selecting this text with the pointing ray and pressing the trigger button starts a group definition. The text changes into "select #1".

A group is defined by selecting the two atoms which are on both sides of a bond. First the atom that should not be part of the group must be selected. If that is done, the text changes into "select #2" and atoms on the other side of the bonds that the selected atom has are coloured red. One of these atoms has to be selected as the second atom, that is to be part of the group. All atoms that are (transitively) connected with bonds to this second atom are going to be part of the newly formed group. The bond between the two atoms will not be considered while the group is being created.

If anything other than an atom is selected during the definition, the 'define group' operation will be broken off. The group that possibly existed before the definition was started will still exist and the text will be restored to "define group". One can of course try a second time to define a group.

Other operations

If no selection is currently in progress, pressing the top button causes the viewpoint to be restored in its default position. This is the same position it has at start up.

Pressing the middle button under the same circumstances causes the 3-D cursor to be reset to its default position. Note that only the position, and not the orientation is restored.

Pressing the bottom button without a selection causes the program to exit.

5 Conclusion

5.1 The virtual reality system

One of the goals of this research has been to work with the system that was purchased in the beginning of 1996 and to investigate its potential for application. About the hardware part of the system it can be said that the head-mounted display works fine. The 3-D mouse has some problems — some of the buttons stopped working, and the tracking sensor inside it sometimes passes incorrect information to the computer. The rest of the tracking system works all right. The graphics hardware works well too, but the question arises if the maximal rendering capacity of 600.000 polygons per second is enough for a serious VR application. This is, however, probably one of the problems in computing science that will never be solved, because with the increasing capacity of the hardware there will always be new applications that require even more.

The software that we use to operate the hardware uses the WTK-library. The programming paradigm of the scene graph that is used in this library is a very good one. It gives the user a clear idea of how the scene is built. Using the functions that are available, one can manipulate this scene graph in an easy way. It should be noted, however, that some people are convinced that by using the scene graph, WTK does not create the most efficient way of rendering the scene.

5.2 Interaction using the 3-D mouse

The method we have developed for using the 3-D mouse to interact with the virtual environment seems to be a good one. The user can easily judge which object is about to be selected by looking at the pointing ray. Objects that are occluded by other objects can, in most cases, be selected by moving the cursor around these objects. One can select the object as soon as the pointing ray is able to hit it. Ware & Jessome (1988) claim that it is not necessary to immerse oneself in a VR system. We have seen, however, that using the HMD for stereo vision improves the understanding of the 3-D structure of the scene very much. We therefore think that our method will be applicable in a wider range of applications than that of Ware & Jessome. One should however restrict use of the HMD as much as possible, because using it over a longer period of time might lead to health problems, such as motion sickness or strained eyes.

With our method of interaction using the 3-D mouse, it is possible to interact with the environment while holding one's arm relaxed at waist level, or on the arm of a chair. This prevents the user from getting a tired arm from holding it stretched out. The fine-tune modes enable accurate placement of objects in a nice way. In future projects it will be interesting to try out variations of the way fine-tuning is implemented — other 2-D planes could be chosen, but one could also try performing fine-tuning by having the same physical movement correspond to a finer movement in the virtual world.

5.3 Future work

In the introduction it has already been pointed out that there are many areas where virtual reality could be applied. In Groningen the research started by this project is aiming at application of virtual reality in the field of molecular modelling. Other fields of application that are currently considered are visualisation of data from computerised tomography, graphical animation in the context of driving simulation, and visualisation of fluid flows.

A Finite State Machines

In this chapter, I will give a short introduction in notation I use for the *finite state machines* (F.S.M.) in this report.

The particular type of F.S.M. I use are called Mealy machines. More information about Mealy machines can be found in (Cohen 1991).

A finite state machine is built out of a finite number of *states*. These states represent a particular configuration of the program. One can step from one state into another by means of a *transition*. This is done as a reaction to an *input symbol*, which in our case is an operation on one of the buttons of the flying joystick.

The states in a F.S.M. are graphically represented by circles, with a label of the form $s_?$ inside (see Fig. A.1(a)). Since not every state represents a situation in which a complete set of actions has been made—for example, in all states s_1 through s_6 in Fig. 4.2, the selection button is still depressed, while the moving action can only stop if it is released—some states have to be identified in which this is the case. The representation of these *end states* is given in Fig. A.1(b).

Just as one has to be able to finish the action by means of end states, one also has to find a state—typically state s_0—in which to begin traversal of the F.S.M. A *start state* is denoted by a fat dot with an arrow to this state (see Fig. A.1(c)).

Transitions are represented by arrows, starting at one state and ending in another, possibly the same state. Normally a transition is labelled with one label, which represents the input symbol, but as I mentioned earlier, the F.S.M.'s I use are Mealy machines. This means that besides an input symbol (typically $\mathtt{i}_?$) there is also a second label (of the form $a_?$). This label is used to indicate that there is output on this transition. Output in our case is visual output. This output the change in graphical representation of the model etc., e.g. colouring the object with a warning colour if it is about to be thrown away. In cases where there is no necessity for visual output, a default label (-) will be used. The representation of a transition is given in Fig. A.1(d).

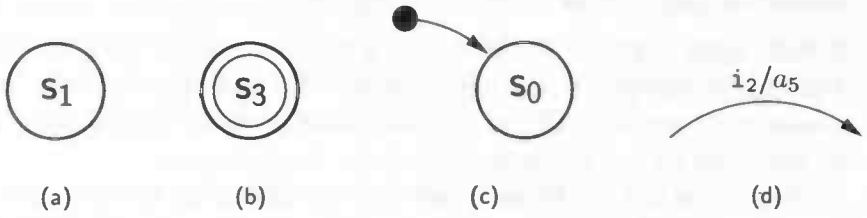


Figure A.1 Different parts of a *finite state machine*: (a) State; (b) End state; (c) Start state; (d) Transition.

B Using the VR system

In this appendix we give an overview of what actions must be performed to operate the VR system as described in Chapter 2. Also a short introduction to the WTK-library (the C++-wrapper) is given, that covers the functions mostly used in this work.

The workstation on which the applications are executed is `virtual.service.rug.nl` and will from now on be referred to as *the virtual*. Another workstation presently available in the research laboratory is `visual.service.rug.nl` and will be called *the visual*. It is possible that in the future the facilities in the laboratory will be extended with additional terminals or workstations. The procedures that are described below will probably apply to those machines as well.

N.B. Currently we are running WTK release 6. It is possible that the procedure for later versions is different from what is discussed here or that paths will change.

B.1 Preparing for the first session

Before the first WTK-application can be run on the virtual, a few environment variables must be set, so the WTK-library knows where to find its resources. The first thing that must be dealt with is finding the licence code. This code can be found in the file `/usr/wtk/WTKCODES`. One can copy this file to the working directory so WTK can find it, but a simpler approach is to set the environment variable `WTKCODES` to the directory in which the code file is found. The variable can be set by typing

```
% WTKCODES=/usr/wtk
% export WTKCODES
```

under the Bourne-Again shell (or by placing the lines in `.bashrc`), or by typing

```
% setenv WTKCODES /usr/wtk
```

under the c-shell (or by placing the lines in `.csh`). If the `WTKCODES` file cannot be found, an error-message is issued.

When the application requires models or texture images it is advisable to set directory paths so that WTK is able to retrieve them from the standard distribution or via additional user-defined paths. The path for directories containing images is set by defining the `WTIMAGES` variable, e.g. to `/usr/wtk/images` in the same way as above. For the model path, the `WTMODELS` variable is used. This can be set to e.g. `/usr/wtk/models`. When multiple multiple directories should be searched, they are separated by a colon, e.g. `/usr/wtk/models:/usr/wtk/buttons`. For both images and models the current working directory is first searched and subsequently the directories in the path specified in the corresponding environment variable.

More information about these and additional environment variables can be found in the hardware guide (Sense8 HG 1996).

B.2 Using the head-mounted display

After defining the environment variables we are ready to start our first WTK-application. If we want to make use of the head-mounted display (HMD) in the application, we must configure the system to use the Multi-Channel Option (MCO) to send the created image as a stereo pair to the HMD. Since this process kills any running sessions on the virtual, and makes using the console impossible, we must operate from another terminal or workstation.

When we are logged in on another workstation, e.g. the visual, we can operate the virtual via remote login. This is done with the **rlogin** command.

```
% rlogin virtual.service.rug.nl [-l <username>]
```

The first step is now to disable console output and redirect the output to the MCO. This is done by typing

```
% mcoGxf
```

This kills any login processes directly running on the console. Since the output is now directed through the MCO, the console display is no longer usable as output device. The keyboard and mouse, however, can still be used to input commands in a running WTK-application.

The process of switching from console to MCO or back (with **consoleGfx**) can take some time, occasionally even several minutes. The program **gfxinfo** can be used to see whether the program is ready or not.

```
% /usr/gfx/gfxinfo
Graphics board 0 is "REV" graphics.
        Managed (":0.0") 1280x1024
        Display 1280x1024 @ 60Hz
        12 GE (GE10 rev. 0x7)
        2 RM4 boards
        Medium pixel depth
        10-bit RGBA pixels
        Not using Multi-Channel Option
```

This output of `gfxinfo` states that the console is functional, and that the MCO is not being used. If the output states that the display is *un*managed, in either resolution, one should wait longer. When the output is as follows

```
% /usr/gfx/gfxinfo
Graphics board 0 is "REV" graphics.
        Managed (":0.0") 640x972
        MCO Display 0 640x486 @ 30Hz interlaced, origin (0, 0)
        MCO Display 1 640x486 @ 30Hz interlaced, origin (0, 486)
        12 GE (GE10 rev. 0x7)
        2 RM4 boards
        Large pixel depth
        10-bit RGB pixels
        Driving Multi-Channel Option
```

the system is ready to be used. If the VR4 control box (for the HMD), the immersion interface box (for the mouse buttons) and the control box of the tracking system[1] are all switched on, we can start the WTK-application.

When one is finished with the system, or one does not need to use the HMD any more, one should restore the console output by typing

```
% consoleGfx
```

Further information, concerning error-messages etc. can be found in `/usr/VE/info`.

B.3 Programming with WorldToolKit

Now that we are ready to run a WTK-application, we should know a little more about how to make programs that use WTK-functions. In this section a number of functions will be presented that give rudimentary knowledge of the WTK-library. With this it should be possible to create some simple applications. For more information the WTK manuals can be consulted (Sense8 RM 1996, Sense8 C++ 1996). We will present the examples using the bindings from the C++-wrapper library. The bindings from the C library are not very different, e.g.

```
WTnode* node;
WTnode_gettranslation(node, ...);
```

in the C library is equivalent to

```
WtNode* node;
node->GetTranslation(...);
```

in the C++library. Where this is not the case, we will make a comment.

[1] Note that after switching on the control box of the tracking system, the power-indicator will turn on and off a few times, indicating that it is performing an initialisation. One should wait until the LED stays on, then the box is ready to be used.

B.3.1 Joining the WTK User's Group

When one is planning to do some serious programming in WTK, one should consider joining the WTK mailing list. Common problems are discussed in this group. When one has a problem, one can post it there and reactions (possibly solutions) can be expected within a few days. More information about how to join is described in the WTK-manual.

B.3.2 Compiling programs

To create an executable, one must tell the compiler where to find the header-files, and the linker where to find the libraries. *Compiling* a program that only uses the C-functions can be done in the following way

```
ncc -c wtk.c -I/usr/VE/include -I/usr/wtk/include
```

When the C++-bindings are used, one should use the following line

```
NCC -c wtk.cc \
    -I/usr/VE/include -I/usr/wtk/cppwrap/include -I/usr/wtk/include
```

Linking all object-files into an executable requires specification of a lot of libraries. For C-programs this goes as follows

```
ncc -o a.out *.o \
    -L/usr/VE/lib -L/usr/wtk/lib \
    -ljoystick -lwtk -lvsiaudiostub -lsgiaudiostub \
    -lGLw -lGL -lXm -lXt -lX11 -lm
```

and for C++-programs as follows

```
NCC -o a.out *.o \
    -L/usr/VE/lib -L/usr/wtk/cppwrap/lib -L/usr/wtk/lib \
    -ljoystick -lwtkcpp -lwtk -lvsiaudiostub -lsgiaudiostub \
    -lGLw -lGL -lXm -lXt -lX11 -lm
```

It is clear that using `make` to recompile programs is strongly recommended. In the directory `/usr/wtk` a sample program, as well as a sample makefile can be found.

B.3.3 Header files

Now that we know how to compile and link programs, we can look at how WTK is used in program code. In every code file that uses WTK-functions, a header file must be included so the compiler can do type-checking.

```
#include <wt.h>
```

The C++-functions come with their own header file. The header file for the C-version should also be included, though.

```
#include <wt.h>
#include <wtcpp.h>
```

B.3.4 Initialising the application

`static void WtUniverse::New(int display_config, int window_config)`

Before we can use functions from WTK-library, we have to initialise its run-time data structures using the method `WtUniverse::New()`. This method should be the first WTK-function in the program, and it should only be used once. It initialises the internal state of the universe and opens and initialises the graphics device. A viewpoint that is attached to this device is also created. This is by default the viewpoint through which the scene is displayed.

The `display_config` argument is used to specify which type of display is needed. Possible values are either `WTDISPLAY_MONO` (or equivalently `WTDISPLAY_DEFAULT`) for use on a standard display or `WTDISPLAY_STEREO` for use with the HMD.

The `window_config` argument is used to specify whether or not 'decorations' should be applied to windows. One of the values `WTWINDOW_DEFAULT` for normal use or `WTWINDOW_NOBORDER` for use with the HMD should be passed as parameter.

`static void WtUniverse::SetActions(void (*actionfn)(void))`

The action function can be used to control the activity in the simulation. It is a user-defined function that is called every time the simulation loop is executed. A possible use is programming termination by having a button-press trigger the termination of the simulation loop or to handle events for user interfacing. The action function can be set using `WtUniverse::SetActions()`.

`static void WtUniverse::Ready(void)`

Every time that the execution of the simulation loop is started (i.e. before the method `WtUniverse::Go()` is called) `WtUniverse::Ready()` must be called to prepare the application for entering the loop. This should be done after all graphical entities have been created.

`static void WtUniverse::Go(void)`
`static void WtUniverse::Go1(void)`

Starting the simulation is done by making a call to `WtUniverse::Go()`. This passes the control of the application to the simulation handler. One can influence the control by using the action function or tasks.

When one wants to execute the simulation loop only once (e.g. to draw something on the display before doing a time consuming initialisation) one can use the method `WtUniverse::Go1()`.

`static void WtUniverse::Stop(void)`

One is able to exit the simulation loop by making a call to `WtUniverse::Stop()`. After this method is called, the current execution of the loop is continued to the end and then the control is returned to where `WtUniverse::Go()` was called. One can for example make a call to `WtUniverse::Stop()` from the action function when the user presses a certain button.

`static void WtUniverse::Delete(void)`

When the simulation is ended, one should call `WtUniverse::Delete()`. It deletes all objects in the universe, and frees all used memory. It also closes the graphics device. This must be the last WTK call in the application.

The framework of a standard WTK-application is displayed in Fig. B.1.

```
#include <wt.h>
#include <wtcpp.h>

void actionfn(void);
void main(int, char*)
{
    WtUniverse::New(..., ...);
    // Set up lights; open sensors; initialise scene graph; etc.

    WtUniverse::SetActions(actionfn);
    WtUniverse::Ready();
    WtUniverse::Go();

    // WtUniverse::Stop() has been called!!!
    WtUniverse::Delete();
}

void actionfn(void)
{
    ... // Do your own stuff here!
    if(...) WtUniverse::Stop();
}
```

Figure B.1 Framework of a WTK-program.

N.B. Normally, methods are dependent on an instance. They should be called as `<Instance>.<Method>()` or `<InstancePtr>-><Method>()`. For example

```
WtP3  position;    position.Norm();
WtP3* positionPtr; positionPtr->Norm();
```

Methods that are defined as *static* methods, however, are not dependent on an instance. They can be invoked at all times by calling them as `<Class>::<Method>()`. Examples of this are

```
WtUniverse::Ready();
WtUniverse::Go();
```

Note that all methods of the `WtUniverse` class are defined as static methods.

B.3.5 Creating a scene graph

The scene graph is a hierarchical structure in which the scene is defined that has to be displayed at the end of every simulation loop. The scene graph consists of a number of different types of objects, which are called *nodes*. Examples of these are geometry nodes, light nodes, transform nodes, group nodes, switch nodes, etc. The scene is displayed in a depth first order, starting at the root of the scene graph.

`static WtRoot* WtUniverse::GetFirstRootNode()`

When the universe is created with `WtUniverse::New()`, a number of lists are initialised. One of these lists is the list of scene graphs. A pointer to the first scene

graph[2] can be obtained using the `WtUniverse::GetFirstRootNode()` method. In the C-version this function is called `WTuniverse_getrootnodes()`.

Handling nodes

All nodes in the scene graph are instances of derived classes of the `WtNode` class.

`FLAG WtNode::IsEnabled(void)`
Instances of some derived classes of `WtNode` can be disabled to (temporarily) remove them from the scene. One can check with the method `IsEnabled()` whether or not a node is enabled. The returned value can be either `TRUE` if the node is currently enabled or `FALSE` if it is not enabled.

`FLAG WtNode::IsMovable(void)`
One of the derived classes of `WtNode` is `WtMovable`. The position and orientation in the scene of instances of this derived class can be changed during the simulation. One can check with the method `IsMovable()` whether or not a node can be moved.

```
WTpoly* WtNode::RayIntersect(WtP3 ray, WtP3 origin,
                             float* distance, WtNodePath** npath)
```
In the program it will be necessary once in a while to obtain the nearest object along a certain ray. This can be done with the `RayIntersect()` method. It returns a pointer to a `WTpoly` structure.

The first parameter is the direction of the ray, given in the node's own reference frame. The second parameter is the point where the ray starts, also in the node's reference frame. As the third parameter the address of a floating point variable can be passed. If the value passed is not 0, then the distance to the front-most object will be stored at that address. If the fourth parameter is unequal to 0, a node path is created that starts at the node on which the method was called and ends in the node that was selected as being the front-most node. A pointer to this node path is stored at the address specified by the fourth parameter. Note that it is the responsibility of the programmer to delete the created node path when it is no longer needed.

`FLAG WtNode::Remove(void)`
When one wants to remove a certain node from all of its parents in the entire scene graph, one can use the method `Remove()`. If this node has any associated tasks (see section B.3.8), they will no longer be performed. When one also wants to remove the object from memory, `delete` should be used.

Group nodes

The `WtRoot` class is a derived class of the `WtGroup` class. It offers little extra functionality with respect to its base class, so we will discuss `WtGroup` here. Note that `WtGroup` is a derived class of the `WtNode` class, so the methods discussed in the previous section also apply to instances of the `WtGroup` class.

[2]Although it is possible to have multiple scene graphs, most applications will only use one. Typical applications that use more than one scene graph have multiple rooms, with one graph per room, or have multiple windows, with one scene graph per window.

When making an instance of the WtGroup class, it will probably be an instance of one of its derived classes, because they do provide extra functionality that is essential.

`FLAG WtGroup::AddChild(WtNode* child)`

Assembly of the scene graph is done by creating new nodes and adding them as children to a group node, somewhere in the scene graph. This is done using `AddChild()`. The only parameter is the node (that can itself be a group node) that one wants to add.

In the C-version this method is called `WTnode_addchild()`.

It is of course also possible to remove nodes from the scene graph. This can be done in two ways. The first is using one of the methods `DeleteChild()` or `RemoveChild()`. They require the programmer to keep track of the child's number, when it was inserted. The second way is easier, but can only be applied when the node that has to be deleted only occurs once in the scene graph. If that is true, `WtNode::Remove()` will do the same as the other two methods mentioned above.

```
FLAG WtGroup::GetExtents(WtP3& extents)
FLAG WtGroup::GetMidPoint(WtP3& midpoint)
float WtGroup::GetRadius(void)
```

One can obtain information about the geometrical properties of single nodes or subtrees of the scene graph using three methods. `GetExtents()` supplies the user with a vector from the midpoint to any corner of the extents box of the node. `GetMidPoint()` returns the midpoint of the extents box. `GetRadius()` gives the distance between the midpoint and any corner of the extents box, i.e. the length of the vector returned by `GetExtents()`. Note that using `GetRadius()` on the root node of the scene graph gives the radius of the entire virtual world. .

`int WtGroup::NumChildren(void)`

One can obtain the number of children a group node has at a certain moment with the method `NumChildren()`.

Separator nodes

The `WtSep` class is a derived class of `WtGroup`. Instances of this class prevent state information from propagating from their descendent nodes to their sibling nodes. This can be used to have transformation or light nodes apply to one part of the scene graph, but not to the rest. This makes using separator nodes the natural way to group objects that have a common orientation which should not apply to the rest of the scene.

`WtSep::WtSep(WtGroup* parent)`

New separator nodes can be created with `new WtSep()`. The only parameter indicates the parent in the scene graph.

If 0 is passed as parent, the separator node will not be inserted into the scene graph. This can then be done at a later time using `WtGroup::AddChild()`.

`FLAG WtSep::Enable(FLAG enable)`

Instances of the `WtSep` class can be enabled or disabled using the `Enable()` method. It is passed a parameter indicating whether it should be enabled (`TRUE`) or should not be enabled (`FALSE`).

Switch nodes

The `WtSwitch` class is also a subclass of `WtGroup`. It allows the programmer to determine which of its children should be processed.

`WtSwitch::WtSwitch(WtGroup* parent)`
: New switch nodes can be created with `new WtSwitch()`. The only parameter indicates its parent in the scene graph.

  If 0 is passed as parent, the switch node will not be inserted into the scene graph. This can then be done at a later time using `WtGroup::AddChild()`.

`int WtSwitch::GetWhichChild(void)`
`FLAG WtSwitch::SetWhichChild(int which)`
: Switch nodes allow the programmer to determine which of its children should be processed at a certain time. The number of the child that is currently being displayed can be obtained using the `GetWhichChild()` method.

  Switching between children is done with the `SetWhichChild()` method. This method is passed a parameter that indicates the child that should be displayed. Valid values are `0`, `1`, ..., `NumChildren()-1` to select one of the children; additional values are `WTSWITCH_ALL` to select all children and `WTSWITCH_NONE` to select none of the children. By default no children are processed, i.e. `WTSWITCH_NONE`.

Geometry nodes

In WTK graphical objects are also a part of the scene graph. They are called *geometry nodes*. When the scene graph is traversed and a geometry node is encountered, it is rendered in the frame buffer.

There are three ways of creating geometries. The first is by constructing them oneself; we will not discuss this. The second way is by using one of the predefined geometries available. The third way is by loading them from file.

Predefined geometries There are a number of types of predefined geometries available in WTK. These include cylinder, blocks, cones, etc. A constructor function is available for all types that takes some parameters that specify the properties of the geometry. The first parameter reflects the type of geometry. This can be one of `WTBLOCK`, `WTCONE`, `WTCYLINDER`, `WTEXTRUSION`, `WTHEMISPHERE`, `WTRECTANGLE`, `WTSPHERE` or `WTTRUNCONE`.

Depending on the type, the `WtGeometry` method accepts different parameters. The `bothsides` parameter is accepted in all methods, however. This parameter specifies whether both sides of each polygon in the geometry should be visible. If `FALSE` is passed, then back-facing polygons (i.e. their inside surfaces) are not rendered.

Most of the geometry node constructor functions also accept the `gouraud` parameter. This parameter influences the rendering of the geometry. When it is passed `TRUE` Gouraud-shading is applied, which looks better, but takes longer to render.

`WtGeometry::WtGeometry(int type, float height, float radius, int tess, FLAG bothsides, FLAG gouraud)`
: When creating a cylinder geometry, this method is used. The height of the cylinder and its radius are set with the `height` and `radius` parameters, respectively. The

tess parameter defines the number of polygons that should be used to create the cylinder; e.g. passing a value of 4 creates a block.

`WtGeometry::WtGeometry(int type, float radius, int nlat, int nlong, FLAG bothsides, FLAG gouraud)`

When creating a sphere geometry, this method is used. The size of the sphere is set using the radius parameter. The number of latitude and longitude subdivisions used in creating the sphere is set with nlat and nlong, respectively.

For other geometry types, consult Sense8 C++ (1996, page 13).

In the C-version creating geometries is done somewhat differently. First a geometry is created using e.g. `WTgeometry_newcylinder()`. This function accepts the same parameters as the C++-method used to create a cylinder, except for the type parameter, which is not necessary. It creates a WTgeometry, from which a WTnode can be constructed using the `WTgeometrynode_new()` function. For more information one should consult Sense8 RM (1996).

Loading geometries from file

`WtGeometry* GeometryNodeLoad(char* filename, float scale=1.f)`
`WtGeometry* WtGroup::GeometryNodeLoad(char* filename, float scale=1.f)`

WTK can load geometries from files in the following formats: Autodesk DXF, Wavefront OBJ, VRML, WorldToolKit NFF and others. These files are loaded using the `GeometryNodeLoad()` method. The first parameter is the filename, the second the scale at which it should be loaded.

This method is defined as a method of the WtGroup class, but also as a 'classless' method. The loaded geometry node is inserted as a child of the group instance if it is used in the former way, otherwise the loaded geometry node will have no parent. It can be inserted in the scene graph at a later time using `WtGroup::AddChild()`.

Other geometry node methods

`FLAG WtGeometry::Enable(FLAG enable)`

Instances of the WtGeometry class can be enabled or disabled using the Enable() method. It is passed a parameter indicating whether it should be enabled (TRUE) or should not be enabled (FALSE).

`FLAG WtGeometry::GetExtents(WtP3& extents)`
`FLAG WtGeometry::GetMidPoint(WtP3& midpoint)`
`float WtGeometry::GetRadius(void)`

See **Group Nodes**, p. 52.

`void WtGeometry::SetRGB(unsigned char r, unsigned char g, unsigned char b)`

The colour of an instance of the WtGeometry class can be changed using the SetRGB() method. It takes three parameters, that have values from 0 to 255. They define the colour in which the geometry should be rendered.

Transform nodes

The position and orientation of the geometries is dependent on the state information at the time they are rendered. This state information can be altered using *transform nodes*. A transform node consists of a 4x4 matrix in which position and orientation information can be stored. If a transform node is encountered when the scene graph is traversed, the current transform matrix is multiplied with the matrix in the encountered node. Any geometry nodes that are encountered will be rendered using the result of this multiplication.

`WtXform::WtXform(WtGroup* parent)`

A new transform node is created with the `WtXform()` method. The only parameter indicates its parent in the scene graph.

If 0 is passed as parent, the transform node will not be inserted into the scene graph. This can then be done at a later time using `WtGroup::AddChild()`.

`WtMotionLink* WtXform::AddSensor(WtSensor* sensor)`
`void WtXform::RemoveSensor(WtSensor* sensor)`

It is possible to make the position and orientation of the transform node dependent on a sensor. This is achieved with the `AddSensor()` method. The sensor, to which the transform node should be linked is passed as the single parameter. Deleting the connection can be done using `RemoveSensor()`, to which the sensor from which the transform node should be disconnected is passed.

More information about creating motion links is provided in section B.3.9.

```
FLAG WtXform::GetOrientation(WtQ& ori)
FLAG WtXform::GetRotation(WtM3& mat)
FLAG WtXform::GetTransform(WtM4& mat)
FLAG WtXform::GetTranslation(WtP3& pos)
FLAG WtXform::SetOrientation(WtQ ori)
FLAG WtXform::SetRotation(WtM3 mat)
FLAG WtXform::SetTransform(WtM4 mat)
FLAG WtXform::SetTranslation(WtP3 pos)
```

The position and orientation information can be stored in WTK in a number of different formats. These are a vector of three floats (`WtP3`), for positions; a *quaternion* (`WtQ`), which is a compact way of storing orientations; a 3x3 matrix (`WtM3`), for storing rotations; and a 4x4 matrix (`WtM4`), for storing both positions as well as orientations. For more information about these classes, see section B.3.10.

WTK offers functions both for retrieving and setting the position and orientation of a transform node in all available formats.

```
FLAG WtXform::Rotate(float ang_y, float ang_x, float ang_z, int frame)
FLAG WtXform::RotateQ(WtQ rot, int frame)
FLAG WtXform::Translate(WtP3 pos, int frame)
```

To change the orientation and position relative to the current value, one can use the `Rotate()`, the `RotateQ()` and `Translate()` methods.

The `Rotate()` parameter takes four parameters, the first three being the amount of rotation (in degrees) that is applied to the transform matrix. The rotations are applied around the *y*-, *x*- and the *z*-axis in that order. Note that this is also the order in which the parameters are passed to the method. The `RotateQ()` and

`Translate()` take an instance of the `WtQ` and a `WtP3` class, respectively, as the first parameter.

The last parameter in all three methods specifies the frame of reference in which the transformation is applied. This can be either `WTFRAME_LOCAL`, which means that the transformation is applied with respect to the transform node's own frame of reference; or it can be `WTFRAME_PARENT`, if the transformation is to be applied with respect to the reference frame of the transform node's parent.

Movable nodes

When one wants to create a geometry that can have its own orientation and position in the scene, one has to create a transform node that sets the correct transformation, and a geometry node that is inserted in the scene graph after this transform node. All nodes that are processed subsequently will, however, also have the transformation applied to it. To prevent this from happening, one will generally also create a separator node and insert the transform and geometry nodes as children of this separator node. In this way a *movable geometry* is created.

This idea can also be applied to other node types. A derived class of `WtXform` has been created, that is called `WtMovable`. The `WtMovable` class has some extra functionality with respect to the `WtXform` class, but that functionality will only be interesting in a limited range of applications. Some derived classes of `WtMovable` have been created, however, that are interesting. For the example given above, an instance of the `WtMovGeometry` class can be made. It has exactly the same functionality as the three separate instances, but combined into one object. The `WtMovGeometry` is created as a derived class of both `WtMovable` and `WtGeometry`. Note that multiple inheritance is used here. The derived class has no additional functionality with respect to both its base classes. Other classes to which a similar procedure has been applied are `WtMovSep` and `WtMovSwitch`.

N.B. Using multiple inheritance in this way is very elegant, but it also creates a problem. Suppose one wants to use a method of the `WtNode` class, e.g. `Remove()`, for an instance of `WtMovGeometry`. Since `WtMovGeometry` does not define that method itself, the compiler will try to find this method in its base class. In this case these are `WtMovable` and `WtGeometry`. Both these classes are a derived class of `WtNode`, which defines the `Remove()` method. The compiler must, however, use a unique way of going from the derived class to the base class. We must therefore first pretend that the instance of `WtMovGeometry` is actually an instance of e.g. `WtMovable` before we can use the `Remove()` method. This is done by *casting* the instance to one of its base classes, in this case `WtMovable`. Invoking `Remove()` is e.g. done in the following way

```
WtMovGeometry* ball;
((WtMovable*) ball)->Remove();
```

Node paths

In WTK it is allowed to insert a node several times into the scene graph. This can be necessary, e.g. when one wants to display a memory consuming geometry more than once, in different places. One can instantiate multiple separator and transform nodes and

insert the geometry node as a child in all separator nodes. When one is interested in the properties of a specific instance of such a geometry, e.g. its orientation with respect to the world's coordinate system, one could traverse its ancestors and keep track of the desired information. The same can be done using a *node path*, which is a way of distinguishing between multiple occurrences of the same node.

`WtNodePath::WtNodePath(WtNode* leaf, WtNode* root, int instnum)`
: Creating a node path is done with the method `new WtNodePath()`. A unique node path is defined by specifying three arguments: the node in which one is interested (`leaf`), the ancestor (`root`), and the occurrence number `instnum`, which must have a value between 0 and the total number of ways, minus 1, that one can go from `root` to `leaf`.

  Deleting a node path is done using the `delete` function.

`WtNode* WtNodePath::GetNode(int num)`
: One can obtain a pointer to a node in the node path using the `GetNode()` method. It is passed one argument that indicates the required node, 0 being the first node and `NumNodes()-1` the last node along the path.

`FLAG WtNodePath::GetOrientation(WtQ& ori)`
`FLAG WtNodePath::GetTransform(WtM4& mat)`
`FLAG WtNodePath::GetTranslation(WtP3& pos)`
: One can obtain information about the transform a node has with respect to one of its ancestors by creating a node path between these nodes and using one of the functions `GetOrientation()`, `GetTranslation()` and `GetTransform()`. These supply the orientation, translation or transformation, respectively, of the node with respect to the specified ancestor. Note that when the root node of the scene graph is given as ancestor, the transform with respect to the world coordinate system is obtained.

`int WtNodePath::NumNodes(void)`
: `NumNodes()` returns the number of nodes in the node path.

B.3.6 Controlling the viewpoint

`static WtViewPoint* WtUniverse::GetFirstViewPoint(void)`
: When the universe is created with `WtUniverse::New()`, a graphics device is opened. This provides the user with a view at the scene from a certain point. This is called the *viewpoint*. In WTK a pointer to the first viewpoint in a list of viewpoints[3] can be obtained using the `WtUniverse::GetFirstViewPoint()` method. In the C-version this function is called `WTuniverse_getviewpoints()`.

`void WtViewPoint::SetParallax(float p)`
: When one wants to use stereoscopic viewing in the HMD, the images for the left and the right eye should be drawn from a different position. The distance between the points from which the two images are generated is called *parallax*. The function `SetParallax()` is used to set the parallax. Its default value is `0.0`; this means that in order to benefit from using the HMD it should be set to an appropriate value. We have found that a value of 0.05 times the scenes radius gives a satisfactory result.

[3] Although it is possible to have multiple viewpoints, most applications will only use one.

`void WtViewPoint::SetPosition(WtP3 pos)`
`void WtViewPoint::SetOrientation(WtQ ori)`

It is possible to directly set the position of the viewpoint and the orientation of the viewing coordinate system using the `SetPosition()` or `SetOrientation()` methods, respectively. The `SetPosition()` method is passed a `WtP3` object, in which positional information has been stored. The `SetOrientation()` method is passed a `WtQ` object, in which orientational information has been stored.

`void WtViewPoint::GetPosition(WtP3& pos)`
`void WtViewPoint::GetOrientation(WtQ& ori)`

One can also obtain information regarding the viewpoint's current position and the orientation of the viewing coordinate system. This is done using the `GetPosition()` and `GetOrientation()` methods, respectively. The `GetPosition()` method is passed (a reference to) a `WtP3` object, in which the position of the viewpoint will be stored. The `SetOrientation()` method is passed a `WtQ` object, in which the orientation of the viewing coordinate system will be stored.

`WtMotionLink* WtViewPoint::AddSensor(WtSensor* sensor)`

It is possible to make the position and orientation of the viewpoint dependent on a sensor. This is achieved with the `AddSensor()` method. The sensor, to which the viewpoint should be linked is passed as the single parameter. Since updating the sensors is done relative to the value in the last execution of the simulation loop, one can place the viewpoint somewhere in the virtual world, and have it linked to the sensor with relative movements.

More information about creating motion links is provided in section B.3.9.

B.3.7 Opening and addressing the sensors

Fastrak sensors

WTK supports a number of different sensors. Among these a class for the Polhemus Fastrak tracking system is included. This class is called `WtFastrak`. It initialises the tracking system to operate on the default baud rate. Opening the tracking sensors is done using

```
WtFastrak* hmd_trak = new WtFastrak(SERIAL2, 1);  // For the HMD
WtFastrak* tdm_trak = new WtFastrak(SERIAL2, 2);  // For the 3-D mouse
```

where `SERIAL2` is the port on which the tracking system is connected to the virtual and `1` and `2` are the sensor numbers. Note that they should always both be opened, and always in this order. For the C-version the macro definition `WTfastrak_new()` with the same parameters can be used.

We have changed the default setting to increase the update rate. In the C-version this is no problem, since we can redefine the `WTfastrak_new()` macro. For the C++-version, however, things are more difficult, since we cannot redefine a class-definition. We can use a different way, though, because the `WtSensor` class allows a valid `WTsensor*` — e.g. the one created by `WTfastrak_new()` — to be passed as parameter. Since we do not need the methods that the `WtFastrak` class offers extra, we can simply use `WTfastrak_new()` to open the sensors at the higher baud rate and pass the thus obtained `WTsensor*` to the `WtSensor` class.

```
WtSensor* hmd_trak = new WtSensor(WTfastrak_new(SERIAL2, 1));
WtSensor* tdm_trak = new WtSensor(WTfastrak_new(SERIAL2, 2));
```

Note that the sensors still have to be opened in this order.

`void WtSensor::SetSensitivity(float s)`
: It is necessary to scale the amount of translation of the sensors (in the case of the tracking sensors) to match the dimensions of the virtual world. This can be done using the method `SetSensitivity()`. The value that is passed as parameter can be made dependent on the scene, using the `WtRoot::GetRadius()` method.

3-D mouse buttons

The buttons of the 3-D mouse are not included as a standard class in WTK. It is, however, possible to create an interface so that WTK can pretend that it is a standard class. This has been done for the buttons. The functions that define this interface are placed in `/usr/VE/lib/libjoystick.a`.[4] A header file for these functions called `joystick.h` is also provided. This file defines a macro `IWIfloystick_new` (without parameters) that returns a `WTsensor*`. For the C++-version this pointer can again be used to create an instance of the `WtSensor` class, in a similar way to the tracking sensors.

```
#include <joystick.h>
WtSensor* buttons = new WtSensor(IWIfloystick_new);
```

The header file also defines some constants that can be used when accessing the configuration of the buttons.

`int WtSensor::GetMiscData(void)`
: The configuration of the buttons can be accessed using the `GetMiscData()` method. This method returns the current configuration as an integer value. From this value the state of the various buttons can be obtained by performing a bit-wise AND (`&`) of the desired button on the returned value.

```
int configuration = buttons->GetMiscData();
if (configuration & FLOY_TRIGGERDOWN) {
    ... // Trigger button is pressed
} else {
    ... // Trigger button is not pressed
}
```

Other sensor methods

`FLAG IsValid(WtBase* base)`
: Opening a sensor always creates an instance of the `WtSensor` class, also if the initialisation has not been correct. We must therefore check if the instance is valid. For this the method `IsValid()` is provided. Although this method is intended to be used for instances of any WTK-class, for the sensors it is very important. For

[4] This explains the `-L/usr/VE/lib` and `-ljoystick` in the link command.

the C-version such a function does not exist, but here testing if the returned pointer equals `NULL` is sufficient.

When `IsValid()` is used, it returns either `TRUE` or `FALSE`. It should be noted that in the current version of the C++-wrapper the meaning of the returned values has been switched; i.e. the function returns `TRUE` if the class is not valid and it returns `FALSE` if it is valid. In future versions this will probably be corrected.

Although it might seem necessary to obtain the orientation or position of the sensors once in a while, this is actually not the case. There are methods for doing this, but using them to have moving objects requires a lot of unnecessary programming. When one wants to couple objects in the virtual world to the tracking sensors, this can best be done using a *motion link*, see section B.3.9.

To ensure that the sensor is closed in the proper way when the application is ended, one should always use the provided method for this. In the C++-version this is handled using the `delete` function, conform to deletion of regular objects. In the C-version one can use the `WTsensor_delete()` function, to which the sensor that has to be closed is passed as the only parameter.

B.3.8 Using tasks

Defining the behaviour of the program is achieved by using the *action function*. It defines which operations need to be executed in every simulation loop. Some operations, however, are very specific for a certain object, e.g. the bouncing of a ball. Therefore, a second method to define program behaviour is created in WTK, namely *tasks*. Tasks can be assigned to any object in the simulation. In it one can program the behaviour of the associated object.

`WtTask::WtTask(void* object, WTtask_function func, float priority=1.f)`
: A task can be created using `new WtTask()`. A pointer to the object to which the task should be assigned is given as the first parameter. The second parameter is the function that should be executed. When this function is executed, the object specified in the first parameter is given as the only argument. Valid functions therefore accept one parameter, a pointer to a `void`, and should not return any value. A (optional) third parameter specifies the priority of the task; tasks with a low-numbered priority are executed before tasks with a high-numbered priority. If this parameter is not specified, it is set to `1.0`.

`FLAG WtTask::Add(void)`
`FLAG WtTask::Remove(void)`
: Tasks can be dynamically added or removed from the simulation using the `Add()` and `Remove()` methods, respectively. If a task is removed from the simulation it is no longer executed, but it still exists. Deleting it is done using the `delete` function.

B.3.9 Using motion links

A *motion link* is a connection between a source of position and orientation, such as a tracker sensor, and a target that moves correspondingly to changes in that position and

orientation. We have already seen that making such a connection can be done using `AddSensor()` methods that are defined in some classes. It can also be done by making an instance of the `WtMotionLink` class.

`WtMotionLink::WtMotionLink(void* source, void* target, int source_type, int target_type)`

A motion link is created using `new WtMotionLink()`. The first parameter is a pointer to the source from which the new positional and orientational information should be taken; possible sources are sensors and paths (which we will not discuss). The second parameter is a pointer to the object to which the new information should be applied; possible targets are viewpoints, movable nodes, transform nodes, and node paths. The third parameter specifies the type of the source, which is one of `WTSOURCE_SENSOR` and `WTSOURCE_PATH`. The fourth parameter specifies the type of the target. This can be either `WTTARGET_VIEWPOINT`, `WTTARGET_MOVABLE`, `WTTARGET_TRANSFORM` or `WTTARGET_NODEPATH`.

`FLAG WtMotionLink::AddConstraint(int dof, float min, float max)`
`FLAG WtMotionLink::RemoveConstraint(int dof)`

The motion link can be constrained so that position or orientation of the target stays within a certain range. The constraint is applied for one degree of freedom (at a time) that is specified in the first argument. Possible values are `WTCONSTRAIN_X`, `WTCONSTRAIN_Y` and `WTCONSTRAIN_Z` for translational constraints and `WTCONSTRAIN_XROT`, `WTCONSTRAIN_YROT` and `WTCONSTRAIN_ZROT` for rotational constraints. The `min` and `max` arguments specify the range within which the target is constrained. For rotational constraints these values are in radians.

A constraint can be removed using the `RemoveConstraint()` method. The degree of freedom from which the constraint should be removed is passed as the only argument.

B.3.10 Mathematical operations

The position and orientation information that is contained in transform nodes can be stored in WTK in a number of different formats. These are a vector of three floats (`WtP3`), for positions; a *quaternion* (`WtQ`), which is a compact way of storing orientations; a 3x3 matrix (`WtM3`), for storing rotations; and a 4x4 matrix (`WtM4`), for storing both positions as well as orientations.

These formats are available in the C++-wrapper as classes that define a number of methods for manipulating the transformational information. Because most of these methods speak for themselves, comments will be scarce.

WtP3

`WtP3::WtP3(void)`
`WtP3::WtP3(float x, float y, float z)`

Creating an instance of `WtP3` can be done in two ways. The first is without specifying parameters; the values will then be initialised to `0.0`. The second way is by specifying these values as parameters.

```
void Frame2Frame(WtPQ frame1, WtPQ frame2, WtP3& pout)
void Local2WorldFrame(WtPQ frame, WtP3& pout)
void World2LocalFrame(WtPQ frame, WtP3& pout)
```
Transforming position information from one frame of reference to another can be done using the specified three methods.

```
double WtP3::Mag(void)
```
Returns the size of the vector.

```
void WtP3::Norm(void)
```
Normalises the vector.

```
WtP3 WtP3::operator~(void)
```
Returns the reflection of the vector with respect to the origin.

```
WtP3 WtP3::operator+(WtP3& second)
WtP3 WtP3::operator-(WtP3& second)
WtP3 WtP3::operator*(float scalar)
WtP3 WtP3::operator/(float scalar)
```
Redefinitions of the +, -, * and / operators. With these operator one can easily add, subtract, etc. WtP3 objects. E.g. one can use

```
WtP3 a,b,c;
c = a+b;
```

to add two WtP3 objects.

WtQ

The components of a quaternion can be thought of as representing a vector in 3-D and a twist around that vector. Quaternions in WTK are defined in the WtQ class.

```
void WtQ::WtQ(void)
```

```
void WtQ::WtQ(float x, float y, float z, float w)
```
Creating an instance of WtQ can be done in two ways. The first is without specifying parameters; the first three values will then be initialised to 0.0, the last value to 1.0. The second way is by specifying these values as parameters.

```
void WtQ::Frame2Frame(WtPQ frame1, WtPQ frame2, WtQ& qout)
void WtQ::Local2WorldFrame(WtPQ frame, WtQ& qout)
void WtQ::World2LocalFrame(WtPQ frame, WtQ& qout)
```
Transforming orientation information from one frame of reference to another can be done using the specified three methods.

```
double WtQ::Mag(void)
```
Returns the size of the quaternion.

```
void WtQ::Norm(void)
```
Normalises the quaternion.

```
WtQ WtQ::operator~(void)
```
Returns the inverse of the quaternion.

WtM4

A transform can be described using a 4x4 matrix.

`void WtM4::M42PQ(WtPQ& pq)`
Transform an instance of `WtM4` into an instance of `WtPQ`.

`WtM4 WtM4::operator*(WtM4& second)`
Matrix multiplication.

WtPQ

A transform can also be described with a position (`WtP3`) and an orientation (`WtQ`). These are combined into the `WtPQ` class.

`WtPQ::WtPQ(void)`
`WtPQ::WtPQ(WtP3 p, WtQ q)`
A `WtPQ` instance can be created without or with parameters. In the former case the values are initialised as in the separate classes, in the latter case the values are initialised according to the arguments.

`void WtPQ::Frame2Frame(WtPQ frame1, WtPQ frame2, WtPQ& pqout)`
`void WtPQ::Local2WorldFrame(WtPQ frame, WtPQ& pqout)`
`void WtPQ::World2LocalFrame(WtPQ frame, WtPQ& pqout)`
Transforming position and orientation information from one frame of reference to another can be done using the specified three methods.

`void PQ2M4(WtM4& mat)`
Transform an instance of `WtPQ` into an instance of `WtM4`.

B.4 Inheritance graph

Fig. B.2 shows the inheritance relation that the classes have. In this graph it can clearly be seen that the classes `WtMovGeometry` and `WtMovSep` are created by multiple inheritance. In this way two inheritance paths to `WtNode` are possible.

The classes `WtUniverse`, `WtP3`, `WtQ`, `WtPQ` and `WtM4` are separate classes.

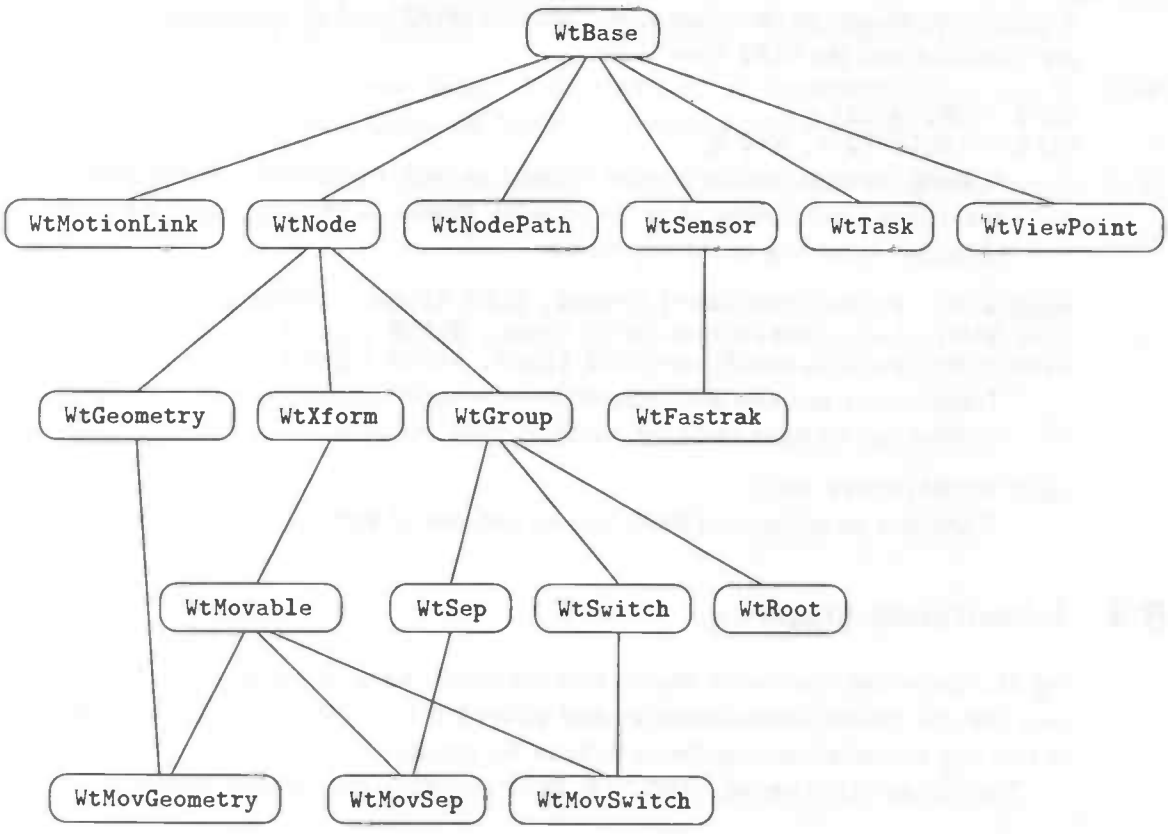


Figure B.2 Inheritance graph for a number of WTK-classes.

Bibliography

D.A. Bowman and L.F. Hodges, 1995. *User Interface Constraints for Immersive Virtual Environment Applications.* Technical Report 95-26, Graphics, Visualisation and Usability Center, Georgia Institue of Technology, USA.
URL: *ftp://ftp.gvu.gatech.edu/pub/gvu/tech-reports/95-26.ps.Z*

P. Brijs, 1992. *MOVE: Modeling Objects in a Virtual Environment.* Master's thesis, University of Technology, Delft.

F.P. Brooks, M. Ouh-Young, J.J. Batter and P. Kilpatrick, 1990. Project Grope: Haptic displays for scientific visualization. *Computer Graphics*, 24 (4): 177–185.

S. Bryson and C. Levit, 1991. The virtual windtunnel: An environment for the exploration of three-dimensional unsteady flows. In *Proceedings of Visualization '91*, pages 17–24. IEEE CS Press.

M. Chen, S.J. Mountford and A. Sellen, 1988. A study in interactive 3-D rotation using 2-D control devices. *Computer Graphics*, 22 (4): 121–129.

D.I.A. Cohen, 1991. *Introduction to Computer Theory*, chapter Finite automata with output, pages 154–176. John Wiley & Sons, Inc., New York, second edition.

M. Figueiredo, K. Böhm and J. Teixeira, 1993. Advanced interaction techniques in virtual environments. *Computers & Graphics*, 17 (6): 655–661.

S. Houde, 1992. Iterative design of an interface for easy 3-D direct manipulation. In *Human Factors In Computing Systems, CHI '92 Conference Proceedings*, pages 135–142.

Polhemus, 1993. *3Space Fastrak User's Manual.* Polhemus Inc., Colchester, Vermont. Revision F.

Sense8 C++, 1996. *WorldToolKit Release 6 C++ Programming Guide.* Sense8 Corporation, Mill Valley, California.

Sense8 HG, 1996. *WorldToolKit Release 6 Hardware Guide (for SGI).* Sense8 Corporation, Mill Valley, California.

Sense8 RM, 1996. *WorldToolKit Release 6 Reference Manual*. Sense8 Corporation, Mill Valley, California.

SGI, 1993. *Multi-Channel Option Owner's Guide*. Silicon Graphics, Inc., Mountain View, California.

B. Stroustrup, 1991. *The C++ Programming Language*. Addison-Wesley Publishing Company, Inc., second edition.

VRS, 1994. *VR4 User's Guide*. Virtual Research Systems, Inc., Santa Clara, California.

C. Ware, K. Arthur and K.S. Booth, 1993. Fish tank virtual reality. In *Human Factors In Computing Systems, INTERCHI '93 Conference Proceedings*, pages 37–42.

C. Ware and R. Jessome, 1988. Using the bat: A six-dimensional mouse for object placement. *IEEE Computer Graphics & Applications*, 8 (6): 65–70.

J. van Winkel and L. Willems, 1993. *Het C++ boek*. Academic Service B.V., Schoonhoven.