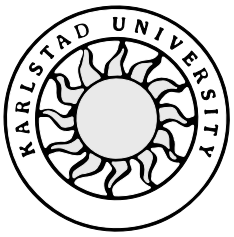


Computer Science

Henrik Hedlund

Virtual Reality Applications

Evaluation of Development Environments

Bachelor's Project

2000:06

Virtual Reality Applications

Evaluation of Development Environments

Henrik Hedlund

This report is submitted in partial fulfillment of the requirements for the Bachelor's degree in Computer Science. All material in this report which is not my own work has been identified and no material is included for which a degree has previously been conferred.

Henrik Hedlund

Approved, 00-05-31

Advisor: Hua Shu

Examiner: Stefan Lindskog

Abstract

This thesis is the documentation of a Bachelor's project at the Department of Computer Science, Karlstad University, for SAAB Bofors Dynamics. It contains an overview and evaluation of six different Virtual Reality development environments that could be of use to future Virtual Reality developments at SAAB Bofors Dynamics. The evaluation discusses the pros and cons of each development environment and in what way it could be valuable to SAAB Bofors Dynamics. A conclusion is then reached and recommendations are presented.

This thesis also documents the implementation of a testbed that has been created using one of the evaluated development environments. This testbed is an example of how a three-dimensional computerized demonstration, or manual, can be designed and constructed.

Contents

List of Figures

List of tables

1 Introduction

This is both a general introduction to the areas covered in this project and a brief introduction to the disposition of the thesis.

1.1 Simulation/visualization

Simulation can be described in two ways, either as an accurate, numeric calculation of how something behaves under certain conditions, or as a real-time, interactive visualization of a scenario based on some realistic approximations. Basically this means that it is impossible to use accurate calculations and behavior models in real-time visualizations. These accurate simulations are used for exact predictions of a scenario, and are calculated over a large time span. On the other hand it is possible to pre-calculate data and then feed it to a real-time visualization. This project does not span all the ingredients of accurate simulation; it is mainly focused on visualization.

Visualization in turn is a multi-faceted task, ranging from the illustration of simple graphs and tables to three-dimensional (3D) presentation of a sequence of actions. The particular area of interest in this project is computer-aided 3-dimensional visualizations of real-time calculated data. To achieve the visualization in real-time, simplified models of behavior are required. This principle applies in many of today's popular computer-games, where speed is crucial for viewing pleasure. Other possible areas of application include architecture, design and prototyping, education, conferencing, military training simulators, scientific visualization and surgical practice. SAAB Bofors Dynamics' connection to the defense industry leads to the particular area of military training simulators and demonstrators. These include, but are not limited to, driving, flight, ship and tank simulators.

1.2 Three-dimensional graphics

In this section, a short introduction to 3D graphics provides the reader with a basic understanding of 3D graphics programming.

The *coordinate system* shown in *Figure 1.1* is fundamental to understand 3D orientation. Placement and direction of the axes can vary dependent on different systems and environments.

The smallest building block is called a *vertex*, and represents one point in the coordinate system. Lines consist of two vertices, start-point and end-point.

By connecting multiple lines basic two-dimensional primitives such as *triangles* (3 vertices), *quadrants* (4 vertices) and *polygons* (N vertices) can be formed.

The most common 3D *primitives* are boxes, cones, cylinders and spheres. In addition, surfaces and materials are defined and solid objects can be formed. Material consists of different aspects such as light dependent attributes (shininess and emissiveness) and colors.

To add further realism to the surface a *texture* can be applied. The texture is a picture that adds the appearance of a real surface.

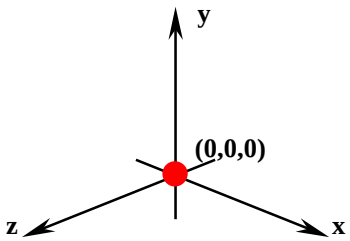


Figure 1.1 – An example 3D coordinate system

Transformation is used to translate and rotate an object. To translate an object is to place, or move, it in the coordinate system. Shading and lightning are other features that add realism and atmosphere to the scene.

A *scene* is a virtual environment that contains objects and attributes. A graph is often used to represent the scene, by ordering all attributes and objects in a hierarchic manner (*Figure 1.2*). This means that a child inherits the attributes of its parent. A very complex engine performs the actual rendering, but this is out of the scope of this thesis.

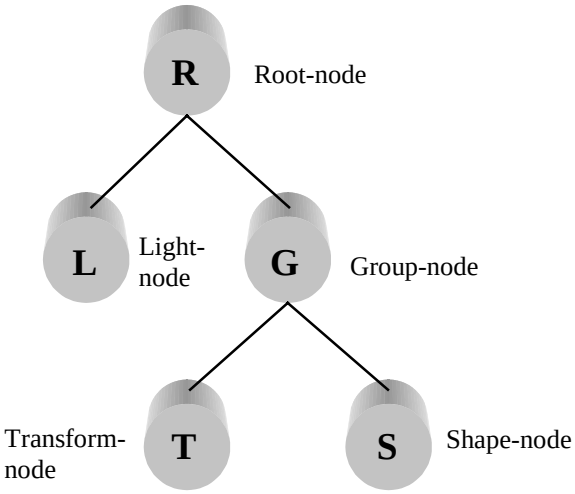


Figure 1.2 – A simple scene graph

A scene graph is a *directed acyclic graph*. The graph in *Figure 1.2* is an example that contains five different nodes. These are the most common nodes, but the different types of nodes can vary from system to system. A group-node is a node, which does not have any attributes in itself, but can have children. The root-node is basically a group-node without any parents, thus being the group-node of the whole scene. Light-nodes contain light-attributes for the scene. A transformation-node determines the placement and orientation of the geometry, which is contained in the shape-node.

1.3 Virtual Reality

Virtual Reality (VR) was a fashion-word a couple of years ago, when the Internet began to grow. People thought that within the next decade everybody was going to live on the Internet, in a *virtual reality*. Since then, not much has happened that has reached the public audience. Surely the Internet is bigger than ever, but its citizens still use an ordinary computer to interact with each other. The technology to *immerse* oneself within the virtual world exists, but it is still too expensive for ordinary consumers. 'Immerse' is one of the keywords that surround VR, since it can be thought of as a technology to increase the level of immersion in visualizations. Immersion, in the context of VR, is basically the cognitive conviction of being 'inside' a 3D scene.

The whole idea is that by placing the user in the simulation loop (i.e. increasing the level of immersion), he can apply his inherent spatial skills to work with the complex data that the visualization presents. Thus the aim is to create a sense of presence by letting the user's sensory inputs receive data generated by a computer, rather than from the physical world [1]. Today this is accomplished by replacing hearing and vision, but tomorrow's technology might offer replacements for other than these two senses.

1.3.1 Head Mounted Displays

The replacement of hearing is nothing new, and is simply accomplished by using headphones connected to a computer. Replacing vision is a bit trickier, since depth perception comes from the difference between the two images that the eyes capture. In order to trick the mind that it is really in a virtual environment, stereographic viewing is used. Stereographic viewing means that the computer generates two images of the viewed scene, where one appears to be seen from a view slightly to the right of the other. The first image is then

presented to the left eye and the second to the right. This emulates depth perception, and the user think that he sees in 3D. The device that is used to accomplish this is called a *Head Mounted Display* (HMD), and generally looks like a set of glasses, or even a helmet, that are placed on the user's head. A simpler version of the HMD is *stereo glasses* (or *shutter glasses*), which uses an ordinary monitor as the display.

1.3.2 Virtual Reality Gloves

In order to interact with the virtual world, different kinds of input devices have been developed. The simpler ones include wands and space pucks, which can be thought of as 3-dimensional mice. A more advanced device is a glove equipped with a number of sensors that feed the computer of the user's hand-motions. By using this kind of device a user can grab and hold objects in the visualized world, and the interactive experience is greatly enhanced.

1.3.3 Trackers

A third device that has been used to deepen the immersion in the virtual environment is called a *tracker*. A tracker is a device that keeps track of the placement and orientation of the user's physical body. The user's physical motions are translated to motions in the virtual world, and the user gets a feeling of completely being in the generated environment.

1.4 About the thesis

The following section is a brief introduction to each of the chapters.

Background and goals

Describes the background and the goals of the project, SAAB Bofors Dynamics' current situation and their need for this type of project.

Method and requirements

Describes the requirements for the project and the method used in the project.

Results

An introduction and an evaluation of the different development environments that were considered to be candidates for SAAB Bofors Dynamics' Virtual Reality development.

Testbed created with WorldToolKit

A part of this project was the creation of a testbed. This chapter describes the design and implementation of the program that was written using WorldToolKit.

Recommendations

A discussion concerning the results of the evaluations, as well as the author's opinions and recommendations regarding the different development environments.

Please note that chapters 1.1 and 1.2 are written together with Anders Åslund, in conjunction with his Bachelor's project "Special Effects in OpenGL."

2 Background and goals

This chapter describes the background of this project, the problems that are supposed to be solved, SAAB Bofors Dynamics' current situation and their need for this type of project.

2.1 Background

During 1999 SAAB Bofors Dynamics began to look at the area of real-time simulation and environment-animation. The idea of using VR to help SAAB Bofors Dynamics develop, educate, market and maintain their products stemmed from that project.

In today's product development, there are a plentitude of advanced software that makes the work easier for engineers and assemblers. *Computer Aided Engineering* (CAE) has come a long way, and in many areas of engineering today there is software that supports different stages of the product development. SAAB Bofors Dynamics uses a CAD-system called I-DEAS, which supports a diversity of usable functions related to 3D-product design. However, a valuable function is lacking, namely the ability to control the assembly of products. Questions such as "Are there room for the assembler's hands?" and "Is it possible to assemble in the correct order?" need to be answered. I-DEAS only offers the functionality to check if individual parts fit together. Hopefully VR can help answer the other questions.

Today's documentation of a product consists of quite a number of folders full of printed papers. A computerized equivalence to this documentation is of use for both marketing and documentation purposes.

2.2 Goals

The purpose of this project is threefold:

1. To examine in which areas SAAB Bofors Dynamics need VR systems.
2. To examine and evaluate different software, Software Development Kits (SDKs) and Application Programming Interfaces (APIs), and draw a conclusion of what software environment that will be of most use in the development of these VR systems.
3. To examine the VR-hardware market and formulate a requirements specification for the hardware that will be incorporated in these VR systems.

As a part of the Bachelor's project, a small testbed shall be created. This will act as an example of how a computerized manual can be structured. This manual will include a simple interactive 3-dimensional model of some product, and it should be possible to link the model together with a manual in a hypertext form.

3 Method and requirements

These conditions and requirements are a combination of the requirements and needs of SAAB Bofors Dynamics, and the author's opinions and ideas.

3.1 Method

This project was carried out in four different phases. The boundaries of these phases were not completely distinct, because the development of software is an iterative process and the boundaries of different stages may overlap each other.

The first step in the project was an *information gathering* process. This is one of the most important phases in the project, since SAAB Bofors Dynamics is interested in *information* about available products and their use in SAAB Bofors Dynamics' business.

After the information has been gathered, it was time to *evaluate* available products. The requirements for this are specified in chapter 3.2.

The next phase was the *implementation* of a simple testbed. This phase was actually an extension of the evaluation, since the development of the testbed further evaluates the functionality of a chosen product.

The final phase was the *documentation* of the whole process.

3.2 Requirements

The evaluation of software and SDKs are necessary to see if they fulfill certain requirements. The following paragraphs contain the requirements set up for a 3D software development environment. The first five requirements are considered to be decisive for the decision whether or not to use the software. The sixth requirement describes the general guidelines set up for the testbed (the computerized manual). Note that these requirements were not included in any specification, but rather developed during discussion about what could be demanded of a 3D application development software.

From now on the evaluated software is referred to as a SDK, for simplicity.

3.2.1 Level of abstraction

This is a general requirement. It should be fairly easy to develop software with the SDK; otherwise its use would be worthless. A system-developer should not have to be concerned

with the lower aspects of the program. By letting the SDK handle the lower, render-specific details, a developer can concentrate on creating usable applications.

There is a choice to be made whether one should use a highly abstract SDK, which provides the desired results fast, or use a less abstract one, which perhaps is more flexible. It is necessary to weight these two aspects against each other, and see what is best suited for the given project.

3.2.2 Connection to I-DEAS

The second requirement is also a very important one. All the geometries that will be used by the programs developed with the SDK are made by the CAD-system I-DEAS. Therefore it must be possible to import geometries created in I-DEAS to the chosen SDK. Preferably this should be done directly, i.e., the SDK can read the I-DEAS file-data, but this is not necessary. It is enough that some connection exists, for example through converting the I-DEAS data to some format recognized by the SDK. Yet, in case of a conversion it is necessary that the procedure is as simple and accurate as possible, since it might be necessary to replace the geometry several times during the development of an application and some conversions deteriorates the quality of the geometry.

3.2.3 Hierarchical transformation

If the SDK uses hierarchical representation of the scene data (see chapter 1.2), then it is possible to transform the geometry as a hierarchical structure. This is useful if one wants to transform a whole group of geometries, for example a missile turret consisting of several sub-geometries; all that is necessary is to transform a group node containing all of the associated geometry, and the whole group is transformed.

If, on the other hand, it is not possible to perform a hierarchical transformation, the only solution is to do calculations on each individual vertex, which is a quite time-consuming task.

3.2.4 Rotation in any chosen point

In order to implement realistic motions and animations, it is necessary to be able to rotate a geometry, or a whole group of geometries (according to paragraph 3.2.3), at an arbitrary point in the coordinate system. This is necessary since some systems only allow rotation around some global axis, which is somewhat limited. This requirement in combination with paragraph 3.2.2 is discussed further in 3.3.

3.2.5 Different levels of immersion

In order to be able to extend developed systems in the future, it is necessary to have the ability to develop applications with higher levels of immersion, as well as simple 3D applications that run on an ordinary PC. Thus, it is important to have some sort of support for the implementation of VR-hardware, and also the construction of necessary drivers (if these are not included with the hardware).

3.3 Concerning the testbed

As mentioned earlier, part of this project is to develop a simple testbed using the SDK that I have found most suitable for use at SAAB Bofors Dynamics. The testbed is a prototype of a computerized manual, consisting of a simple 3D model of BAMSE (one of SAAB Bofors Dynamics' products). This model shall be interactive in the sense that the user can trigger some reactions in the model, such as the animation of movable parts. It shall be possible to extend the program so that a window containing a hypertext documentation could be linked with the model. Using this structure a user can click on one part of the model and get information about that part in the hypertext-window.

This prototype is also part of the evaluation and further tests of the functionality of the used SDK. There are two subjects that require special attention:

1. The connection to I-DEAS must be thoroughly evaluated. It may be possible that the SDK claims to have full support for a certain file format, but in reality some aspects are missing.
2. Given a certain point in the global coordinate system of I-DEAS, it is desirable to have some sort of method to find the same point in the coordinate system used by the SDK. This is most useful when a geometry is to be rotated around a specific point (se paragraph 3.2.4).

4 Results

After an initial search on the software market, primarily using the Internet, six different development environments were considered to be of interest. These were chosen because they are widely used when developing VR applications. Four of these can be seen as SDKs with varying levels of abstraction (see paragraph 3.2.1), as illustrated in *Figure 4.1*. The fifth environment is actually a markup language for creating interactive 3D worlds (see section 4.5 below). In addition to these five environments, a somewhat different API that has no support of its own for 3D graphics also was considered to be of interest (see section 4.6).

Each of the requirements in section 3.2 will be graded according to the scale: poor, adequate or excellent. The results of each SDK's evaluation will be presented in a table at the end of the evaluation.

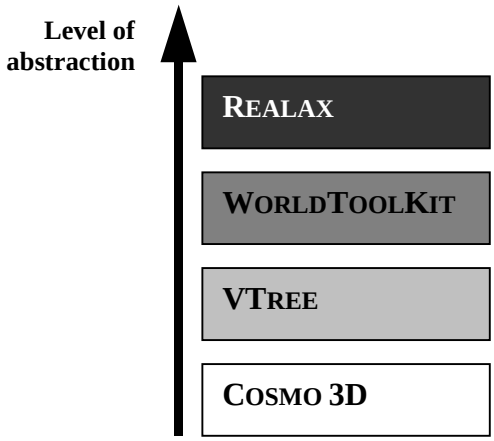


Figure 4.1 – Different levels of abstraction

4.1 WorldToolKit

WORLDTOOLKIT (WTK) is, according to the developer Sense8 Corporation, "a cross-platform development environment for high-performance, real-time 3D graphics applications" [2]. WTK is a SDK that provides a developer with the tools to create virtual worlds through C or C++ code that calls WTK functions. WTK is a library consisting of over 1,000 functions written in C that handle low-level graphics and I/O operations [3]. To provide the possibility of an object-oriented development approach, the WTK package also includes C++ wrapper classes. These are C++ classes that encapsulate the ordinary WTK functions that are written

in C, in a manner that allows a C++ programmer to use WTK. Even though the C code that WTK consists of is based on object-oriented concepts, the C++ wrappers do not offer such object-oriented strengths such as inheritance or dynamic binding, which is obviously a serious drawback.

WTK is available on a plentitude of different platforms, including Windows NT, SGI, Sun, HP, DEC, PowerPC and Evans & Sutherland. This makes WTK a frequently used development environment that has a large user base [4]. This is one of the most valuable aspects of WTK. Since it has been extensively used and evaluated, and most of the weaknesses should have been corrected by now (WTK has reached its 9th release). Another strength of WTK is that it is a complete package supporting the development of a wide variety of VR applications. These applications can range from full immersive systems, such as Caves™, to simple web browser based applications.

There are other softwares included in the WTK development suit. WORLDUP is a viewer that displays 3D worlds in a simple way, which does not include any programming. WORLD2WORLD is a system used for developing distributed 3D systems that run in a network. Unfortunately WTK does not perform as well as some other VR libraries, for instance VTREE, in terms of the frame rate.

4.1.1 Level of abstraction

WTK is the development environment with the second highest level of abstraction, as shown in *Figure 4.1*. Almost every aspect of the development of a 3D application is encapsulated in WTK's functions. The creation of an application is a quite easy task. On the other hand this can also pose a problem, because the SDK hides much of the functionality that a developer perhaps wants to change or to implement in a different way. If a situation presents itself, where the functionality that is implemented in WTK is not satisfactory and need to be changed, this is practically impossible.

4.1.2 Connection to I-DEAS

WTK has the ability to import the following formats:

AutoDesk DXF
AutoDesk 3D Studio mesh
Wavefront OBJ
MultiGen/ModelGen Flight
VideoScape GEO
Pro/Engineer RENDER SLP
Sense8 Neutral File Format
Sense8 Binary NFF
VRML 1.0
VRML 2.0

Table 4.1 – WTK import formats

In addition to these file formats, Sense8 offers a CAD geometry loader to WTK. This loader is not included in the standard WTK package and has to be purchased additionally to the approximate price of $10,000.

The format that is currently of most interest is VRML 2.0, since that is the format supported by both WTK and I-DEAS. As it happens, WTK has problems with the import of VRML 2.0 files. An investigation leads to the following hypothesis. A VRML 2.0 scene consists of a scene graph, which in turn consists of one or more nodes of different types (as described in paragraph 1.2). There exists one special type of nodes at the meta-level, i.e. it is a node that describes nodes. This type of nodes, called *Prototypes*, or PROTO-nodes, is used for creating and describing new, specialized nodes. VRML geometry exported from I-DEAS consists almost entirely of PROTO-nodes. It seems that WTK is unable to import and interpret these nodes. Sense8 has been notified of the problem, but no answer has been received.

A solution to the problem is to convert the VRML 2.0 geometry to VRML 1.0. This is not a good solution, since it involves an extra step in the connection between I-DEAS and WTK. VRML 1.0 is also a quite clumsy file format of much lower quality than VRML 2.0. This is illustrated in *Figure 4.2*, where the same geometry is viewed in VRML 1.0 format to the left, and in VRML 2.0 format to the right.

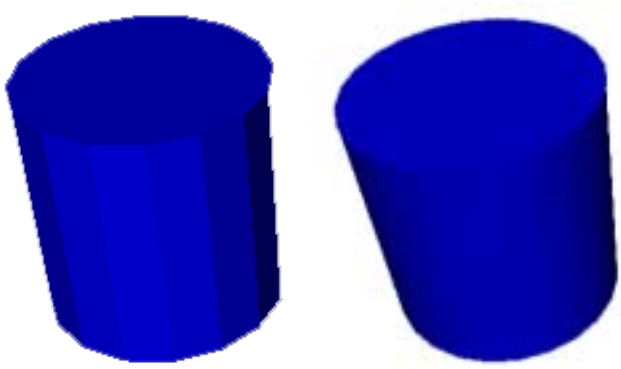


Figure 4.2 – Comparison of VRML 1.0 and 2.0

4.1.3 Hierarchical transformation

All scenes in WTK are structured in a hierarchical manner, which, as described in section 3.2.3, means that WTK allows hierarchical transformation.

4.1.4 Rotation in any chosen point

Every sub-geometry in WTK has a local coordinate system, where the origin is placed at the geometry's midpoint (see *Figure 4.3*). When a geometry is rotated, it rotates around one of its axes that run through the origin of the local coordinate system (also illustrated in *Figure 4.3*). Thus a plain rotation only rotates the geometry around its midpoint, which can be quite limited. The requirements specified that it must be possible to rotate a geometry in any chosen point. This is achieved by the following procedure:

1. The first thing that has to be done is to translate the geometry, so that the point of rotation (marked with an X in *Figure 4.3 (a)*) is located at the origin of the local coordinate system. This is illustrated in *Figure 4.3 (b)*.
2. The next step is to perform the actual rotation (*Figure 4.3 (c)*).
3. Finally, the geometry is translated back to its original position, i.e. the inverse translation of step one is performed. The final result is illustrated in *Figure 4.3(d)*.

Since all modifications to geometries in WTK are performed before a scene is rendered (in each program loop), the sequence above only displays the final result. This means that it does not show on the screen the results of the intermediate steps.

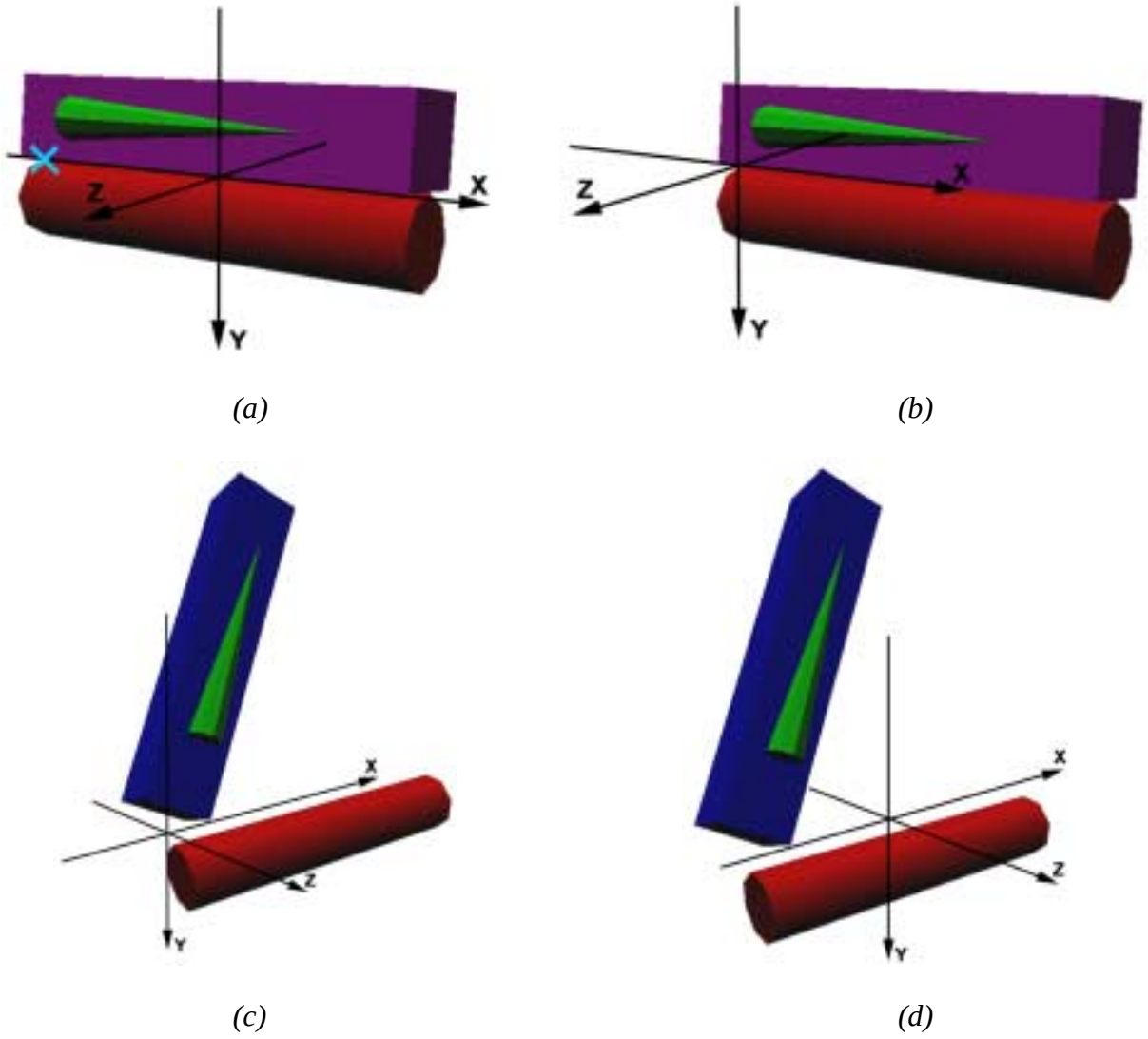


Figure 4.3 – Rotation in a specific point

Please note that the pictures shown in Figure 4.3 are screenshots from different angles. That is why the axes are located differently.

4.1.5 Different levels of immersion

There are three different modes of stereographic viewing and WTK supports them all. The different modes are *Field Sequential Mode, Over/Under Mode* and *Interlaced Mode*.

In field sequential mode the two images (one for the left eye and one for the right eye) are rendered into a single display, and then swapped at 120 Hz to generate a field sequential view at 60 Hz. This requires the graphics hardware to support quad buffering (i.e. there are four image buffers), because each image needs a front and a back buffer. The display also must be capable of supporting 120 Hz update frequency [2].

The over/under mode splits the display in two parts along the horizontal axis. The left eye image is rendered above the right eye image. This mode also requires a display that supports 120 Hz update frequency [2].

"The interlaced mode interleaves the left and right images as alternate scan lines in a single window. All the even scan lines belong to the left eye image and all the odd scan lines belong to the right eye image (or vice-versa)" [2].

WTK also supports many sensor devices, such as trackers, joysticks and VR gloves. Among the supported devices are: Virtual Technologies' *CyberGlove*, Ascension's *Bird / Motionstar / Flock of Birds / Extended Range Bird*, Fakespace's *Pinch Glove*, Logitech's *Head Tracker* and Polhemus' *ISOTRAK / ISOTRAK II / InsideTRAK / FASTRAK / Stylus*. The VR hardware support that WTK provides should be able to satisfy most of users' needs. In the case a certain sensor device is not supported, WTK provides a set of functions that allows a developer to write specialized drivers to make the device function with WTK applications.

4.1.6 Conclusion

WTK is a complete VR development package that is very pleasant to work with. All expected functionalities are provided and everything runs smoothly. As mentioned earlier WTK applications do not get as high frame rate as applications written using VTREE. This is a serious disadvantage, since the performance is very important to the quality of an application. Another disadvantage is the lack of complete object-oriented support, which can be a nuisance when writing C++ applications.

One thing that inclines towards WTK is that many systems that are used at SAAB Bofors Dynamics are provided by the same vendor as WTK. The advantage of choosing systems from the same vendor is that the systems may work better together, and the support may become easier since the same company maintains all of the systems.

WTK is evaluated further in chapter 5, where the design of a testbed created using WTK is described.

Level of abstraction	Excellent
Connection to I-DEAS	Excellent
Hierarchical transformation	Excellent
Rotation in any chosen point	Adequate
Different levels of immersion	Excellent

Table 4.2 – WTK conclusion

4.2 VTree

"VTREE is a robust library of C++ classes and functions designed to empower you to create rich, dynamic, graphical scenes" [5]. VTREE is a complete SDK for the creation of 3D visualizations of varying use. It has great support for environmental visualizations and in particular for military real-time simulations. Just like WTK, VTREE is a complete package for development of 3D and VR applications that are runnable on ordinary Windows NT or UNIX workstations, with or without VR hardware [5].

As mentioned above, VTREE performs better than WTK, i.e. VTREE has a higher frame rate. Otherwise the two environments have about the same to offer to a developer. One major difference, though, is that VTREE is written in C++ (WTK in C) and offers a complete object-oriented structure, capable of inheritance, dynamic binding and polymorphism. This is a gap that unfortunately exists in WTK's object-oriented support. Since VTREE is partly directed towards military- and environmental visualizations, it has also a big library of special effects and other features that WTK lacks.

4.2.1 Level of abstraction

As shown in *Figure 4.1*, VTREE has a slightly lower level of abstraction than WTK. This allows a developer to better control the creation of an application, especially the lower levels of the scene graph, the different nodes in the scene graph and the geometry. The negative aspect of this is that the developer has to do more work. But a careful design may allow for extensive reuse of code in subsequent projects.

4.2.2 Connection to I-DEAS

There is very little information about the file-formats supported by VTREE. The primary format used is VTREE's own, and geometries in other formats have to be converted using some software, which is included in the VTREE software package.

4.2.3 Hierarchical transformation

VTREE uses a hierarchical scene graph to organize all geometries. As mentioned in 3.2.3, all systems that use a hierarchical representation also allow hierarchical transformation, which means that VTREE does support hierarchical transformation.

4.2.4 Rotation in any chosen point

The basic method described in the evaluation of WTK (paragraph 4.1.4) is also applicable for VTREE. The first operation is to translate the geometry so that the chosen point is located in

the geometry's origin. The second operation is to rotate the geometry. The final operation is to translate the geometry back to its original position.

4.2.5 Different levels of immersion

VTREE supports stereographic viewing, which means that applications can use HMDs and similar output devices, as a computerized binocular (widely used in military simulations). The VTREE user manual provides no other information concerning the types of stereographic viewing that are supported (see the evaluation of WTK, section 4.1.5) other than that VTREE supports HMDs.

VTREE also supports a variety of trackers, for example Intersense's *IS300* and *InterTrax*, Ascension's *Flock of Birds* and Polhemus' *FasTrak*, as well as different joysticks and other 3D input devices. To our knowledge, VTREE provides no support for VR gloves, so we have to assume that this does not exist.

4.2.6 Conclusion

VTREE is already in use at SAAB Bofors Dynamics, which makes VTREE a suitable candidate for VR development. Another positive aspect is the high performance that VTREE provides. On the negative side it seems that there is no simple connection between VTREE and I-DEAS, i.e. it is necessary to convert the I-DEAS data in several steps to finally reach the VTREE file format. Otherwise VTREE is quite pleasant to work with, and it has several features that have proven to be very useful.

Level of abstraction	Excellent
Connection to I-DEAS	Poor
Hierarchical transformation	Excellent
Rotation in any chosen point	Excellent
Different levels of immersion	Adequate

Table 4.3 – VTree conclusion

4.3 Cosmo 3D and OpenGL Optimizer

COSMO 3D is a freely distributed scene graph API developed by Silicon Graphics, Inc. (SGI). It is based on SGI's graphics API OPENGL, which is more or less industry-standard, and provides a developer with a high-level interface for complex 3D graphics applications. COSMO

3D is written in C++ and is designed in an object-oriented manner, thus providing a developer with full access to inheritance, polymorphism and other object-oriented strengths [6].

The COSMO 3D API has a slightly lower level of abstraction than WTK and VTREE (as shown in *Figure 4.1*). This means that there is a bit more coding involved in the creation of an application, but at the same time the developer has more control. There is always a choice to be made between flexibility (COSMO 3D) and abstraction (WTK). Since COSMO 3D is written in C++, careful design of developed applications may permit extensive re-use of code in later applications, which reduces the amount of development work involved.

When scene graphs have been created using COSMO 3D, a developer can use the OPENGL OPTIMIZER API to improve the performance of an application. OPENGL OPTIMIZER is also available from SGI and the two APIs are closely intertwined, making them a full development package for high quality 3D applications. Neither COSMO 3D nor OPENGL OPTIMIZER is a development environment for VR applications. Both lack support for higher levels of immersion. This is not necessarily a problem due to the fact that COSMO 3D is almost perfect for use with VR JUGGLER, an API for development of VR applications that has no support of its own for graphical operations. VR JUGGLER is discussed further in section 4.6.

4.3.1 Level of abstraction

As mentioned earlier, COSMO 3D has a slightly lower level of abstraction than WTK or VTREE, but there seems to be no significant difference. The level of the graphical programming is just slightly lower than WTK and about the same as VTREE. The scene graph representation is built on the same structure as both WTK and VTREE, but the major difference between COSMO 3D and WTK, or VTREE, is that both WTK and VTREE are more of complete packages and provide more support for VR hardware. A solution to this problem is discussed in paragraph 4.3.5 below.

4.3.2 Connection to I-DEAS

COSMO 3D and OPENGL OPTIMIZER combined are able to load the following formats:

Cosmo 3D .csb
Open Inventor .iv
IRIS Performer .pfb
VRML .wrl

Table 4.4 – Cosmo 3D and OpenGL Optimizer import formats

In addition to the formats in *Table 4.3*, OPENGL OPTIMIZER provides the developer with tools to write new loaders in order to extend the file-format support.

Again VRML is the format of most interest, and COSMO 3D has support for both VRML 1.0 and VRML 2.0. COSMO 3D has the exact same node representation and scene graph-structure as VRML 2.0, which enables COSMO 3D to have full VRML support.

4.3.3 Hierarchical transformation

As most 3D systems, COSMO 3D uses a hierarchical representation of 3D geometries, thus allowing for hierarchical transformation (see paragraph 3.2.3)

4.3.4 Rotation in any chosen point

In the evaluation of WTK a method for rotation is described (paragraph 4.1.4), which is also applicable to COSMO 3D.

4.3.5 Different levels of immersion

Neither COSMO 3D nor OPENGL OPTIMIZER has any inherent support of immersion altering devices. They are only APIs for creating high quality 3D applications, such as CAD systems, and as such there is no need for VR hardware support. There is one remedy to this situation, which is described in chapter 4.6, namely VR JUGGLER.

As described earlier, VR JUGGLER has no inherent support for 3D graphics, and has to utilize an external API for this. COSMO3D is almost perfect for this purpose, providing an advanced scene graph structure that allows features such as stereographic viewing and other necessities for VR implementation. The combination of COSMO3D and VR JUGGLER provides a developer with a development environment that is as complete as WTK or VTREE. The only difference is that COSMO3D has a slightly lower level of abstraction and offers fewer features in form of special effects, an area that currently is under development at SAAB Bofors Dynamics (see Anders Åslund's thesis *"Special Effects in OpenGL"*).

4.3.6 Conclusion

COSMO3D and OPENGL OPTIMIZER combined with VR JUGGLER is a most powerful and flexible solution for creating 3D and VR applications. That the combination is free of charge is perhaps not as important for a business of SAAB Bofors Dynamics' size, but it is worth mentioning. Both COSMO3D and VR JUGGLER exist on several platforms, both Windows NT and UNIX, which can be important if the development is moved from Windows NT to UNIX (these are the two main platforms in use at SAAB Bofors Dynamics).

The major weak point of COSMO 3D is that special effects are missing in the package. As mentioned earlier special effects is currently under development at SAAB Bofors Dynamics. These are most likely to be created using OPENGL. In that case there is nothing that prevents the usage of these special effects in COSMO 3D. On the contrary, COSMO 3D and OPENGL OPTIMIZER (as the name suggests) are built on top of OPENGL, which makes it very easy to code pure OPENGL in a COSMO 3D structure.

In conclusion, COSMO3D, and OPENGL OPTIMIZER, in combination with VR JUGGLER, should be considered as a development environment to be reckoned with because of their flexibility and availability.

Level of abstraction	Excellent
Connection to I-DEAS	Adequate
Hierarchical transformation	Excellent
Rotation in any chosen point	Excellent
Different levels of immersion	Poor

Table 4.5 – Cosmo 3D and OpenGL Optimizer conclusion

4.4 Realax

According to the REALAX manual, "REALAX is a high-performance virtual reality system, running on all Silicon Graphics and Windows NT workstations" [7]. REALAX is a modeling- as well as a real-time visualization environment. This means that programming is not needed in the development of an application (although it could be done). REALAX is divided in two different parts: RXscene and RXrealtime.

RXscene is used for modeling and scene editing. It is possible to use RXscene as a stand-alone application to create 3D models or in combination with RXrealtime (discussed below) to provide a whole development environment. The mouse, in combination with the keyboard, is used to create the models. Various tools and *gizmos* (usually called *wizards* in other Windows applications) help the user in the creation process. This part of the system is not that useful for SAAB Bofors Dynamics, since the geometry already is created in I-DEAS.

RXrealtime is a real-time environment that allows the user to view a given scene [7]. To navigate through the scene an ordinary mouse can be used, but it is also possible to use more specialized input-devices, such as a VR glove. RXrealtime is primarily made to view worlds,

which means that the user flies around in the scene. This is not so useful in creating smaller applications, for example a computerized manual, where it is only required to view a single object, not a whole world. All functionality in an ordinary programming SDK is also available in RXrealtime, through a convenient graphical user interface (GUI). It is also possible to extend the basic functionality through the use of RXapi.

RXapi is a part of RXrealtime that allows a developer to include ANSI-C programs in the RXrealtime environment. RXapi is used as an ordinary API when writing these RXrealtime-programs, i.e. consists of a library file that is statically linked with the written code.

4.4.1 Level of abstraction

REALAX is actually the SDK with the highest level of abstraction among those being evaluated, since a developer is able to use a GUI. This is not necessary an advantage; in fact it can be a disadvantage, but the existence of RXapi improves the situation a bit. Programming makes it possible to completely control a 3D scene, but when it is necessary to create some 3D application fast and easy, the RXrealtime environment could be quite useful. The RXapi only allows ANSI-C, which makes an object-oriented approach more difficult and this can also be problematic when creating larger applications.

To sum things up, REALAX is very user friendly and has a high level of abstraction. This also makes it a limiting software to use. The world-oriented approach makes it less appropriate for applications with few objects, and the absence of an object-oriented API also makes it less appropriate for creating larger application, at least if there should be programming involved.

4.4.2 Connection to I-DEAS

REALAX claims to have support for the following file-formats:

Import	Export
RXscene	DXF
3D Studio ASCII	IRIS Inventor 1.0
DXF	Multigen .flt (OpenFlt15.4)
IRIS Inventor	SGI .sgo
MultiGen .flt (OpenFlt15.4)	VRML 1.0
Nurb file	VRML 2.0
SGI .bin	Wavefront .obj
SGI .sgo	
SIG .ydl	
STL/STA	
VDAFS	
VRML 1.0	
VRML 2.0	
Wavefront .obj	

Table 4.6 – Realax import/export formats

VRML 2.0 is again the format of most interest. It seems that REALAX suffers from the same problem as WTK, i.e. it does not have full VRML 2.0 support. The main suspect is still the PROTO-node. This problem cannot be solved by converting the VRML 2.0-file to a VRML 1.0-file, since REALAX does not seem to be able to import a simple VRML 1.0 geometry with satisfying results.

An attempt has been made to import one of the simple VRML 1.0-geometries used in evaluating WTK. This geometry consists of one cylinder, one box and two cones, but after REALAX has loaded the geometry the only primitives that are displayed are the cylinder and the box. Judging from the result it seems that REALAX ignores the hierarchical data in the geometry-file, and only imports the geometry at the top of the scene graph (the two cones were children of the box). REALAX also ignores the transformations that should have been made to the box. Moreover, some of the surfaces of the geometries were missing. The imported geometry did not look anything like the one that was stored in the file (compare *Figure 4.4* to *Figure 4.3*). This is simply not acceptable, especially given that these geometries are very basic and not even remotely as complex as the geometries that will be used when SAAB Bofors Dynamics develops their applications.

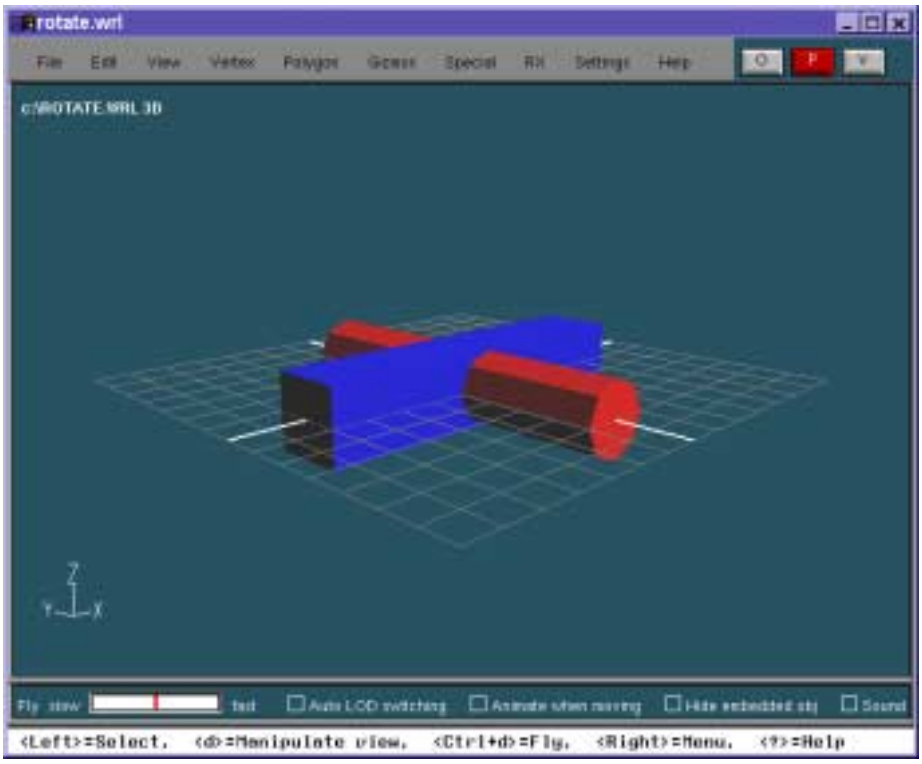


Figure 4.4 – The RXscene environment

4.4.3 Hierarchical transformation

Even though it seems like REALAX is unable to import hierarchical VRML 1.0 data, it uses a hierarchical tree structure to represent a scene, meaning that it supports hierarchical transformation (as described in paragraph 3.2.3).

4.4.4 Rotation in any chosen point

The method described for rotation described in the chapter about WTK (paragraph 4.1.4) is applicable to REALAX as well.

4.4.5 Different levels of immersion

REALAX' user manual states that "REALAX supports the output on all head-mounted displays (HMDs), stereo-systems and multi-monitor systems, as well as the input of almost any VR input- and tracking device" [7]. Other than this there are no further descriptions of exactly which products REALAX supports. Since all HMDs follow the same system (actually there are a handful systems, as described in section 4.1.5, but each HMD uses one of these) it is quite safe to assume that REALAX really supports all HMDs. The supports for input and tracking devices are harder to classify, since there are many different systems. It would have been

useful with a list of supported devices. There is no possibility to explore this further in this thesis, without access to the necessary hardware.

4.4.6 Conclusion

REALAX has proven to be disappointing. When using the program we get the feeling that it is antiquated, even though it is a modern program. Admittedly REALAX is quite easy to use, but it often fails to deliver what it claims to be able to accomplish. Because of its multi-platform support, the developers have chosen to implement a GUI of their own, which in our opinion is inferior to Windows' inherent GUI. In comparison, WTK also has multi-platform support, but uses the platforms' inherent GUIs, which is a much better solution.

Evidently REALAX does not supply anything that SAAB Bofors Dynamics cannot already achieve with other software. The conclusion of the evaluation is that REALAX is not of any value for SAAB Bofors Dynamics.

Level of abstraction	Excellent
Connection to I-DEAS	Poor
Hierarchical transformation	Adequate
Rotation in any chosen point	Adequate
Different levels of immersion	Adequate

Table 4.7 – Realax conclusion

4.5 VRML

VRML is an acronym for the Virtual Reality Modeling Language. VRML can be thought of as several different things. VRML can simply be a platform-independent 3D interchange format, which the first version (VRML 1.0) is designed to be. Another view of VRML is a modeling language that allows specification of complete 3D scenes and worlds. VRML can also be seen as a 3D analog to HyperText Markup Language (HTML), which provides online, interactive, 3D homepages [8]. VRML is designed to distribute 3D worlds over the World Wide Web (WWW) utilizing the HyperText Transfer Protocol (HTTP).

VRML is quite different from the other environments mentioned in this chapter. VRML is not a SDK, not even an API, but rather a language that describes geometry and other environmental aspects of 3D scenes and worlds. The current standard, VRML 2.0 (or VRML

97) includes user interactivity, behavior, scripting and audio support, whereas the first version (VRML 1.0) only described static 3D models [9].

To display a VRML scene, all that is needed is a VRML browser. There are many different browsers available for almost all platforms, usually as plug-ins to ordinary HTML browsers, such as Internet Explorer or Netscape Navigator on the Windows NT platform.

4.5.1 External Authoring Interface

The External Authoring Interface (EAI) is an interface specification that allows an external program to communicate with a VRML scene [9]. EAI, which is integrated in VRML 2.0, is currently only available for Java. EAI for Java functions as a set of classes whose methods can be called to control a VRML world. An applet written in Java can for example ask a VRML scene, through the EAI, for a specific geometry and then change the geometry's properties, such as color, translation or rotation. VRML in itself has support for JavaScript to add behavior to the 3D worlds, but through EAI whole applications can be built around a VRML worlds, making it much more interesting for more advanced uses.

4.5.2 Cortona SDK

The CORTONA SDK is based on the same principles as EAI, and provides a developer with an API that enables communication between an application written in either C, C++, Visual Basic® or Delphi and a VRML world. The CORTONA SDK also allows for scripting within HTML pages, in both JavaScript and Visual Basic script [10]. It has proven to be difficult to acquire any real information about the CORTONA SDK, other than ordinary advertisements. Therefore we have been unable to form any opinion regarding its functionality.

4.5.3 Cosmo Worlds

From the beginning, VRML was developed by SGI and COSMO WORLDS was the number one VRML editing tool available. COSMO WORLDS had all the features that could be expected from such an editor, and could be of most use to SAAB Bofors Dynamics if VRML is chosen to be the environment to develop VR applications. The problem is that in late 1998, SGI decided to stop financing both the VRML development and COSMO WORLDS. Now it is impossible to find a vendor that sells COSMO WORLDS. A replacement editor has to be found.

There is a plethora of different VRML software available over the Internet today, but we have found that most of the products are of low quality. For example, no VRML editor that have been tested, other than the aforementioned COSMO WORLDS, has managed to open a VRML file exported from I-DEAS, which is an obvious sign of insufficient VRML support. It

seems that there is no high-quality VRML editor that can be seen as a replacement of COSMO WORLDS. Unless this situation changes VRML will not have a future, especially not as a tool for SAAB Bofors Dynamics' activities.

4.5.4 Level of abstraction

VRML is a language in itself, and as such is quite easy to work with. The way that VRML describes geometry is easy to understand for someone who has a basic understanding of 3D graphics. There are also plenty of tools that help a developer in the creation of a VRML world. The fact that VRML is a language of its own is both a strength and a weakness. The strength is that it is structured in a way that is specially designed just for 3D use. The weakness is that a developer has to learn a new language, which is quite different from all programming languages, since it *describes* the world and its behaviors.

4.5.5 Connection to I-DEAS

I-DEAS exports VRML, which means that basic foundation is laid. All that is then needed is to specify the scene's behaviors and appearance.

4.5.6 Hierarchical transformation

VRML is built upon a hierarchical structure, which is in fact identical to the one used in COSMO 3D, and therefore supports hierarchical transformation.

4.5.7 Rotation in any chosen point

The principles described in section 4.1.4 also apply to rotation in VRML.

4.5.8 Different levels of immersion

VRML has no inherent support for different levels of immersion, but does not prevent it either. We have heard of solutions where HMDs are used in conjunction with VRML, but have not seen any specifications of the practical details.

In order to get input devices such as trackers and VR gloves to work with VRML, some sort of communications software between them has to be used. One such software is VR JUGGLER. It is not clear whether this works or not, since VR JUGGLER is a C++ API and VRML only has inherent support for Java. Another possible solution to this is to use the CORTONA SDK to handle the communication between VR JUGGLER and the VRML world. This solution still needs to be verified.

4.5.9 Conclusion

VRML has one major strength, i.e. it is designed to be used over a network. Thus VRML is an excellent environment for 3D manuals that can be viewed over the Internet, or over a company's Intranet. Other than this VRML does not seem to have much to offer to SAAB Bofors Dynamics, especially not since VRML's future is quite uncertain. VRML has not changed much since SGI decided to stop financing it in 1998 and the format is in dire need of a enhancement. VRML files are usually rather large in size and therefore not suitable to download, especially when using a slow connection. Other competing formats are starting to show up, for example Cult3D, and perhaps one of these will become a new standard for Internet 3D graphics. Some day, perhaps one of these new formats will have something to offer SAAB Bofors Dynamics. VRML is truly not a suitable development environment for VR applications, unless it undergoes quite a few radical changes in the near future.

Level of abstraction	Adequate
Connection to I-DEAS	Adequate
Hierarchical transformation	Adequate
Rotation in any chosen point	Adequate
Different levels of immersion	Poor

Table 4.8 – VRML conclusion

4.6 VR Juggler

> "VR Juggler is an application framework and set of C++ classes for writing virtual reality applications. It is designed to allow the developer direct access to various graphics APIs for maximum control over applications, while still providing a generalized, easy to understand view of displays as well as input and output devices."

The above is cited from "VR Juggler: A Framework for Virtual Reality Development" [11], and narrates what VR JUGGLER is. VR JUGGLER has been developed at the Iowa Center for Emerging Manufacturing Technology (ICEMT), Iowa State University, since the beginning of 1997. The goal of VR JUGGLER is to aid a developer in the creation of VR applications, so that the developer does not have to worry about the low-level details of VR [11]. One important

thing to mention at this point is that VR JUGGLER is still under development and is not a finished product.

VR JUGGLER functions as a middleware between an application and "the complex interworkings of distributed computing, shared memory, multiprocessing, and device I/O" [11]. An example of this is the device handling, where different devices with similar functionality can be programmed using the same function calls. Data received from the devices will also have the same format. In comparison, WTK, which also supports a plentitude of I/O devices, requires that a developer write different code for each type of device. VR JUGGLER also offers run-time flexibility. This means that it is possible to reconfigure the whole system without stopping a running application. An application created using VR JUGGLER is never aware of the lower levels of the system, and thus never notices any changes, such as a restart of a tracker, or a switch from a HMD to a CAVE™ [11].

VR JUGGLER supports the following input and output devices: Ascension Technologies' *Flock of Birds*, Logitech's *3D Mouse*, Fakespace's *BOOM*, Fakespace's *PINCH* gloves, Virtual Technologies' *CyberGloves*, Immersion boxes, CAVE™, C2 (a CAVE™-like device), HMDs and ordinary screens. VR JUGGLER is currently available on the SGI-, HP- and Windows NT platforms, but since it is written using standard C++ it is an easy task to port VR JUGGLER to new platforms [11].

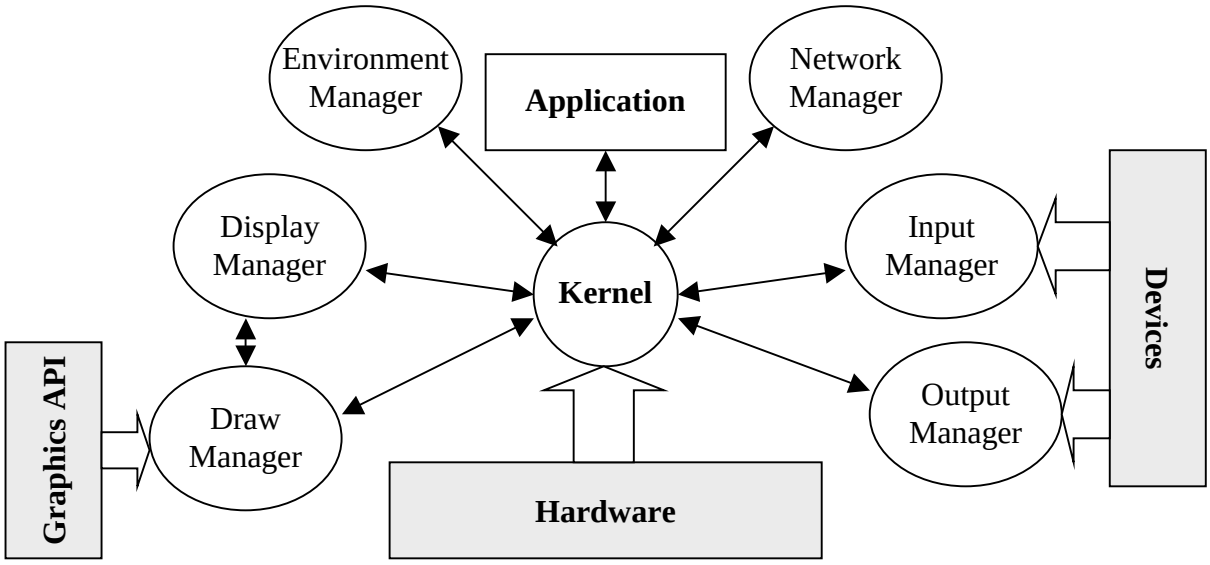


Figure 4.5 – VR Juggler's structure

Figure 4.5 illustrates the structure of VR JUGGLER. As shown in the figure, VR JUGGLER consists of objects called *managers*. Each manager handles a different aspect, or part, of the system and encapsulates all functionality and system specific details of that part. An application has to communicate with a given part through that part's manager. The kernel is

the glue that binds all the managers and the application together, and also handles all the communication within the system [4].

All of these managers will not be described in more detail; their names tell in what area they operate. This section will concentrate on the major difference between VR JUGGLER and all of the other SDKs that is mentioned in this report, namely the *draw manager*. As illustrated in *Figure 4.5* above, the draw manager uses an external graphics- and windowing API; i.e. VR JUGGLER has no support of its own for graphics. Instead, this part of the system has to be provided by some other API, for example SGI's OPENGL (multi-platform) or Microsoft's DIRECT3D (Win32 only). This also means that VR JUGGLER can be combined with any of the aforementioned SDKs to enhance their support for VR hardware, which can prove to be of most value. The fact that VR JUGGLER is a free and open source API is also a very positive aspect and provides a developer with the possibility to change the functionality of VR JUGGLER to a certain project's needs.

Given that VR JUGGLER only fulfills the requirement described in 3.2.5, *different levels of immersion*, there is no need to further evaluate this API. From a developer's viewpoint, VR JUGGLER should be considered to be a possible, and valuable, extension to another API or SDK that handles the graphical operations.

Level of abstraction	Excellent
Connection to I-DEAS	-
Hierarchical transformation	-
Rotation in any chosen point	-
Different levels of immersion	Excellent

Table 4.9 – VR Juggler conclusion

4.7 Other development environments

The development environments being evaluated are not the only ones available on the market. Obviously there are more than those covered in this project. Examples of other VR development environments include DVISE, ALICE, AVOCADO, the CAVE LIBRARY and LIGHTNING. The following information is fetched from *Software Tools for Virtual Reality Application Development* by Allen Bierbaum and Chrisopher Just [4].

DVISE is a high-level VR development environment available on several platforms, including Windows NT and Sun Microsystems. DVISE is primarily used to create virtual representations of products. As such it emphasizes the import of CAD data for the creation of such applications. Due to its specialization on virtual prototyping, DVISE is not well suited for other tasks. While it can be excellent in the development of a virtual manual, it can complicate the development of other applications. Even though it is not in the scope of this project to examine and evaluate DVISE, it should still be considered as a possible development environment for SAAB Bofors Dynamics.

ALICE is a freely available prototyping system running on Windows platforms. ALICE is designed to be easily used by non-technical users. Applications can be written using Python, which is an interpreted, object-oriented scripting language. Unfortunately ALICE is not a good candidate for SAAB Bofors Dynamics' VR developments due to some limitations regarding large and very complex geometries.

AVOCADO is a VR software system running only on SGI platforms. Thus making it impossible for SAAB Bofors Dynamics to use it as a development environment. AVOCADO is designed for rapid prototyping of applications and utilizes Scheme as a scripting language.

The CAVE LIBRARY provides a fairly low-level API for creating VR applications for projection-based system (e.g. the CAVE™). This is not an environment that SAAB Bofors Dynamics is currently looking for, and since the CAVE LIBRARY is only available on SGI platforms it is of little interest to this project.

LIGHTNING is an object-oriented VR development environment. LIGHTNING differs from the other environments discussed above, since it supports multiple programming languages; different parts of a system can be written with different languages, for example C++ or Scheme. LIGHTNING is unfortunately only available for SGI computers.

5 Testbed created with WorldToolKit

A part of the Bachelor's project was to create a testbed of a simple computerized manual using one of the evaluated SDKs. This testbed serves both as an example of how such an application could be designed, and as a further evaluation of the chosen SDK. WTK has been chosen because it is a likely candidate and it has not been used earlier at SAAB Bofors Dynamics (unlike VTREE). The product that is to be used in the manual is BAMSE, which is an anti-aircraft defence system that SAAB Bofors Dynamics develops. For more information about BAMSE, please visit SAAB Bofors Dynamics' homepage at *http://www.saab.se/missiles.*

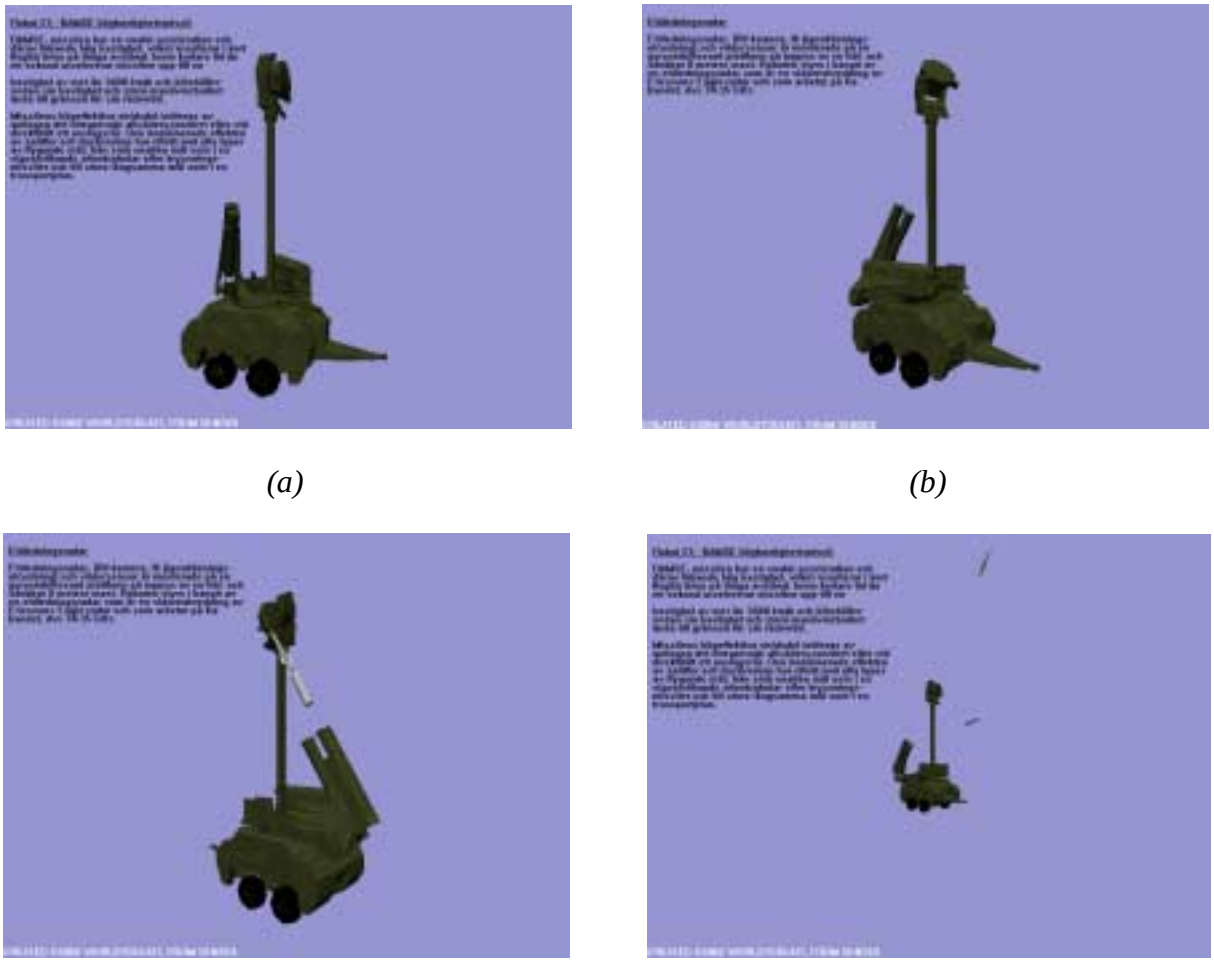
5.1 Description

The testbed that has been created using WTK is more of a 3D demonstration of a BAMSE unit than a computerized manual (see *Figure 5.1* and *Figure 5.2*). The application presents an interactive 3D model of a BAMSE unit. A user can left click on different parts of the model with the mouse to get information about that specific part. The information is presented in the same window, on top of the 3D model. By double clicking on either of the missile turrets the user can elevate or de-elevate them, depending on if they are elevated or not. The same thing can be done to the radar pylon. To illustrate these events, *Figure 5.1 (a)* shows the BAMSE unit with de-elevated missile turrets and elevated radar pylon. *Figure 5.1 (b)* shows the same BAMSE unit with left turret elevated and the radar pylon de-elevated.

(a) *(b)*

Figure 5.1 – A Bamse unit

If the user holds down the left mouse button and moves the mouse, the viewport zooms in or out depending of the direction of the movement. If, on the other hand, the user holds down the right mouse button and moves the mouse, the model is rotated. Holding down the middle mouse button and moving the mouse results in the rotation of the cannon turrets on the BAMSE-station. The radar can be controlled using the arrow keys. The up and down arrows angle the radar, ranging from 0° to 90°. The left and right arrows rotate the radar, ranging from –540° to 540°. By pressing the numeric 1 – 4 keys a user can shoot missiles from one of the turrets. The chosen turret must be elevated in order to shoot missiles. All these different features are illustrated in the following figures. *Figure 5.2 (a)* show the BAMSE unit with rotated turrets. In *Figure 5.2 (b)* the BAMSE unit has a slightly rotated radar. Finally, *Figures 5.2 (c) and 5.2 (d)* shows the BAME unit firing missiles.



(a) *(b)*

(c) *(d)*

Figure 5.2 – A Bamse unit shown in different situations

When pressing the space key, the application enters a demo mode. In this mode, the program basically runs in a demonstration loop, where all the effects are displayed. The

viewport zooms in and out, the model rotates, the radar angles and rotates, the turrets elevate, de-elevate and shoot missiles and all the information about the different BAMSE parts are showed each in turn.

5.2 Design and structure

The following diagram illustrates the object-oriented design used in the implementation of the testbed:

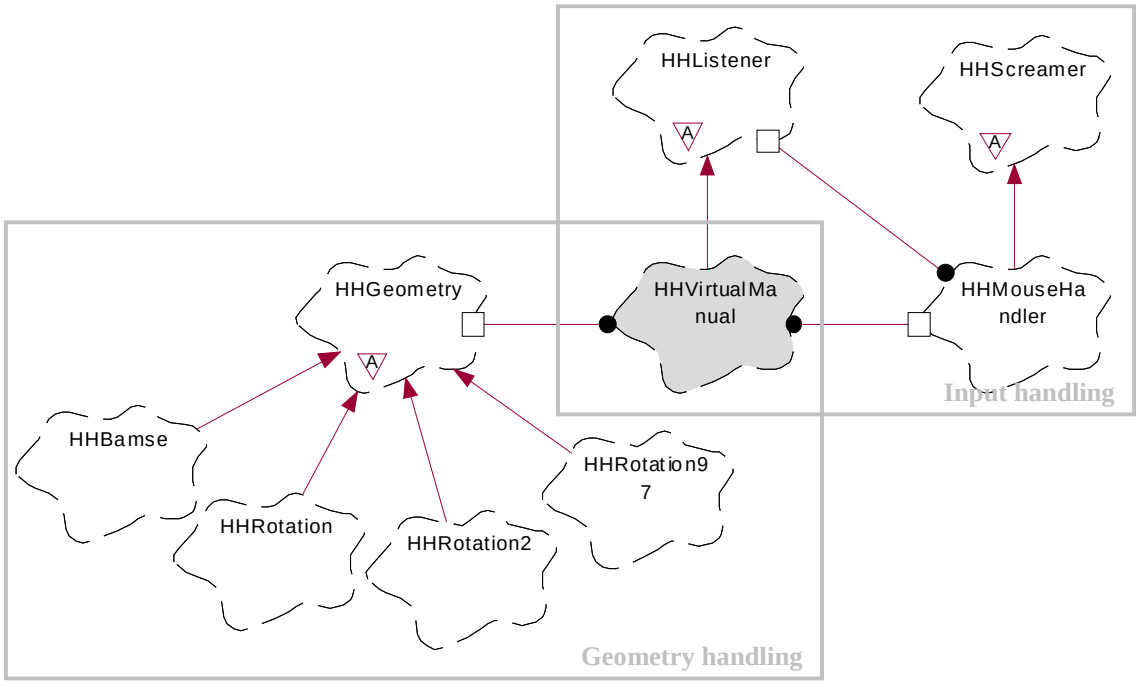


Figure 5.3 – The design of the testbed

There are two major areas in the structure: *input handling* and *geometry handling*. They are described in turn.

The input handling is only designed to handle an ordinary mouse. It is possible to extend the design to enable the handling of other types of input devices, but this was not necessary for this testbed. WTK has built in functions to handle the mouse, and it is possible to map the mouse movements directly to an object, e.g. a viewport or a geometry, in a WTK application. This inherent mouse handling can be configured in two different ways, where the mouse movements are translated to different actions in the application. None of these configurations work in a way that is suitable for this type of application. Therefore it was necessary to write a new mouse handling system, i.e. the class *HHMouseHandler*. The detailed design of the input handling is illustrated in *Figure 5.4*.

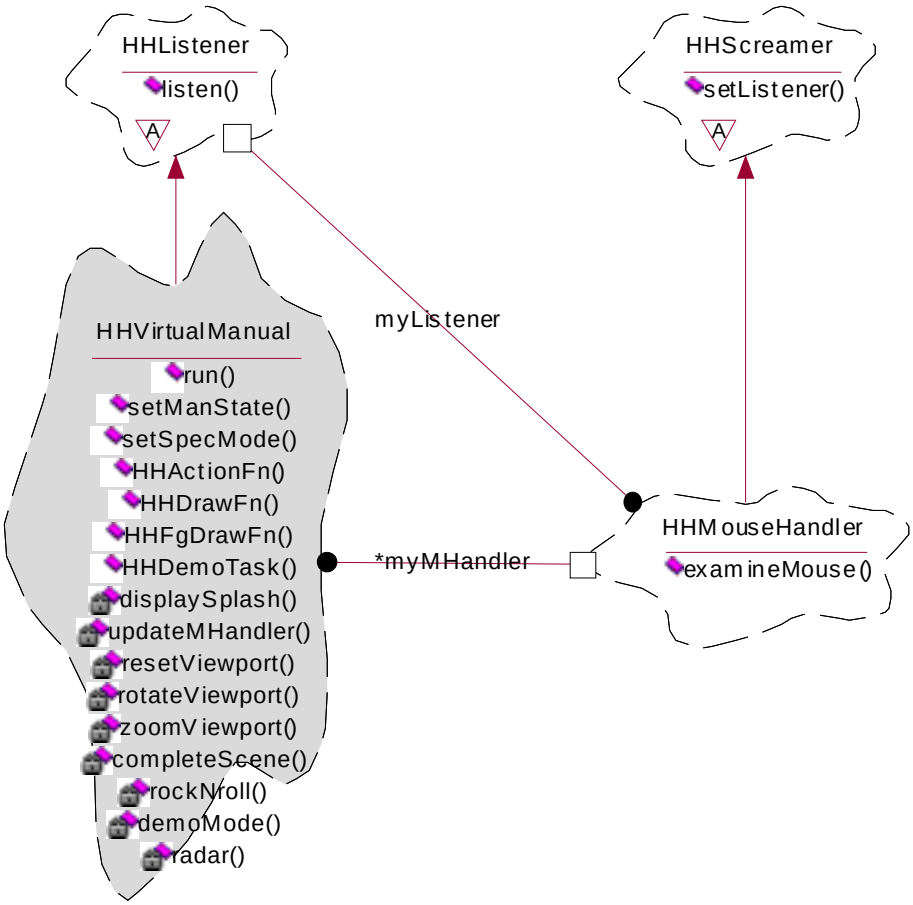


Figure 5.4 – The detailed design of the input handling

The class *HHVirtualManual* is the central class in the structure (best illustrated in *Figure 5.3*). This class contains the actual application, as well as instances of the other classes used in the design. One of the instances is a HHMouseHandler, which, as mentioned earlier, is the class that handles all input from the mouse. The handling of the keyboard events is encapsulated in HHVirtualManual. Together these two classes implement all input handling.

As shown in *Figure 5.4*, there are two abstract classes included in the input handling: *HHListener* and *HHScreamer*. These two 'interfaces' represents a relationship widely used in Java, namely that of events and event listeners. HHVirtualManual inherits from HHListener, thus making it an event listener. HHMouseHandler inherits from HHScreamer, which enables it to generate events, which HHVirtualManual can 'listen' to. This simple structure makes it possible for HHMouseHandler to send information ('scream') to HHVirtualManual. This would otherwise be impossible, since HHMouseHandler is instanced in HHVirtualManual. Thus it is possible for HHVirtualManual to invoke one of HHMouseHandler's (public) methods, but not vice versa.

This design is used to minimize the information flow between HHVirtualManual and HHMouseHandler. HHMouseHandler only send information to HHVirtualManual when it is necessary, i.e. when something interesting has happened to the mouse, for example a button click.

The geometry handling consists of several different classes, as shown in *Figure 5.3*. *HHGeometry* is an abstract class that defines how a geometry class (i.e. a class that encapsulates some sort of geometry) should be structured. It has four children, of which three can be separated from the fourth.

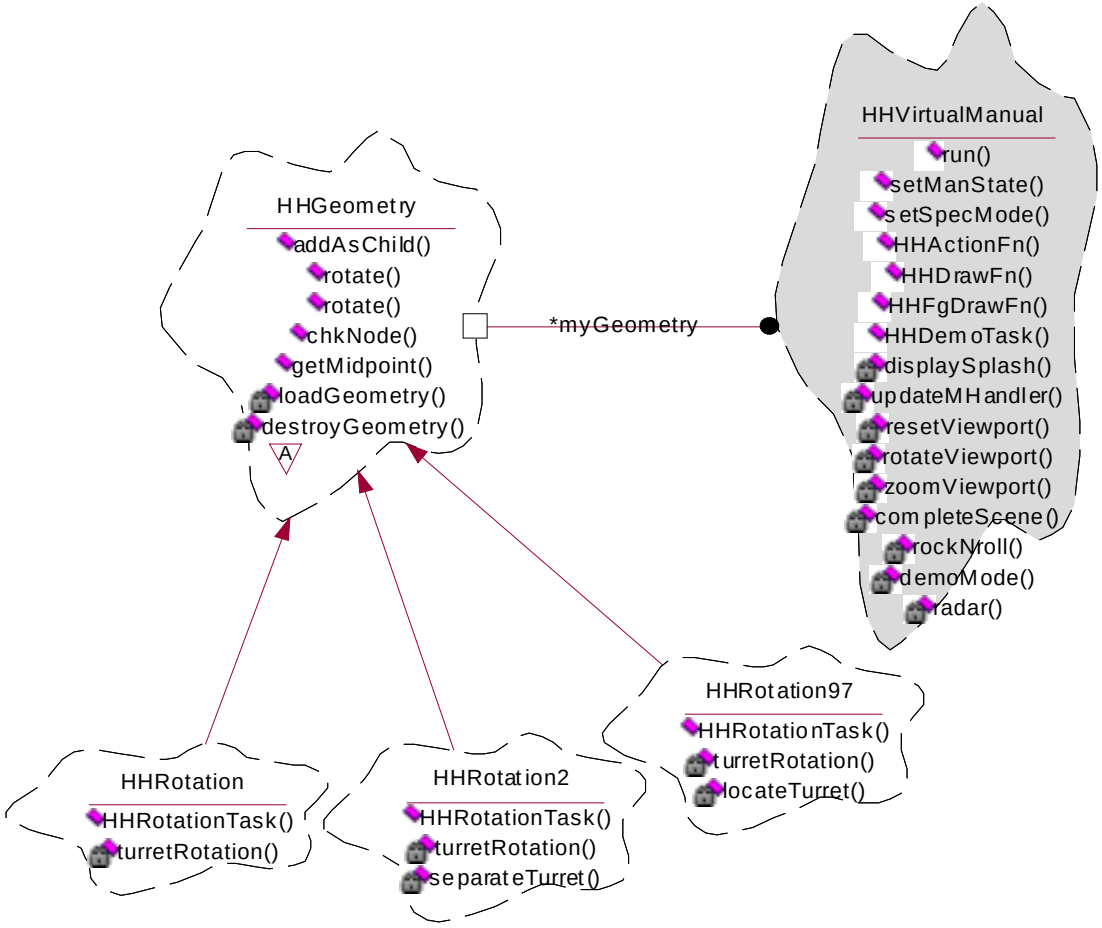


Figure 5.5 – The detailed design of the geometry handling

The three children are geometry classes that have been used for evaluation purposes only. They contain the simple geometry that is shown in *Figure 4.3*. This geometry consists of one cylinder and a box that has two cones as children, i.e. it is a hierarchical structure. The basic idea is that the box and its children should be able to rotate while the cylinder is unaffected.

To achieve this, *HHRotation* simply loads the geometry from two files, where the movable part is separated from the rest of the geometry. This is the usual method used to create movable parts of a geometry. It is not satisfactory for use at SAAB Bofors Dynamics though, since it complicates the connection to I-DEAS (the geometry has to be divided into different files).

To solve this problem, *HHRotation2* loads the whole geometry as a single file. After the file is loaded, the scene graph is traversed until the specific sub-geometry (the box) is found. Then this geometry is removed from the scene, converted into a movable geometry (WTK separates movable geometry from ordinary, non-movable geometry), and finally put back in the scene. This way all geometry can be stored in one file and then be divided *after* the file has been loaded.

HHRotation97 is basically the same class as HHRotation2, but uses a VRML 2.0 geometry file instead of a VRML 1.0. This class has been used to evaluate WTK's VRML 2.0 support. It has been proven that even if there are no PROTO-nodes in a geometry, it still is impossible to traverse the scene graph after loading a VRML 2.0 file. Thus VRML 2.0 is useless together with WTK, since it is impossible to localize specific parts of a geometry.

The three HHRotation classes were only used for evaluation and a fourth geometry class was created for the actual testbed: *HHBamse*. This class is more complex than the previous three classes. It also contains more behaviors, so it was necessary to change the 'interface' HHGeometry.

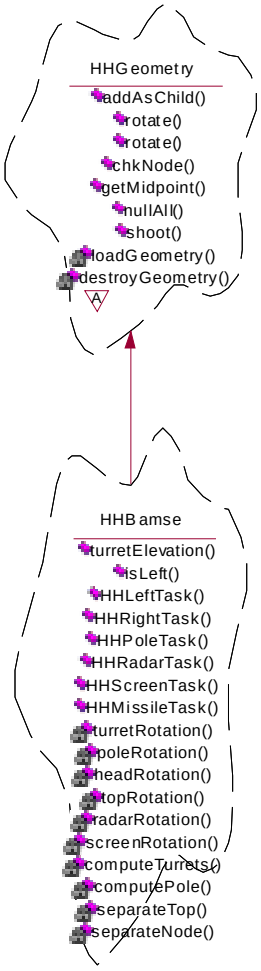


Figure 5.6 – The detailed design of the Bamse geometry class

HHBamse works in the same way as HHRotation2, i.e. it loads the whole geometry (a BAMSE unit) from a single file and then separates the movable parts. The HHBamse geometry contains several movable parts: two missiles turrets, a missile, the turret base, the radar pylon and the radar disc. All of these are localized in the scene graph and converted into movable geometry. The HHVirtualManual can then invoke methods, through the HHGeometry 'interface', to rotate the different parts and shoot missiles.

The design of the HHBamse class is not recommended for a real application. It is only used here since the whole structure is designed to be used as an evaluation of WTK's functions. There are too many different aspects that are encapsulated in a single class. It is recommended to move these aspects into several separate classes, to better utilize the object-oriented paradigm. An example of one such structure is shown in *Figure 5.7*.

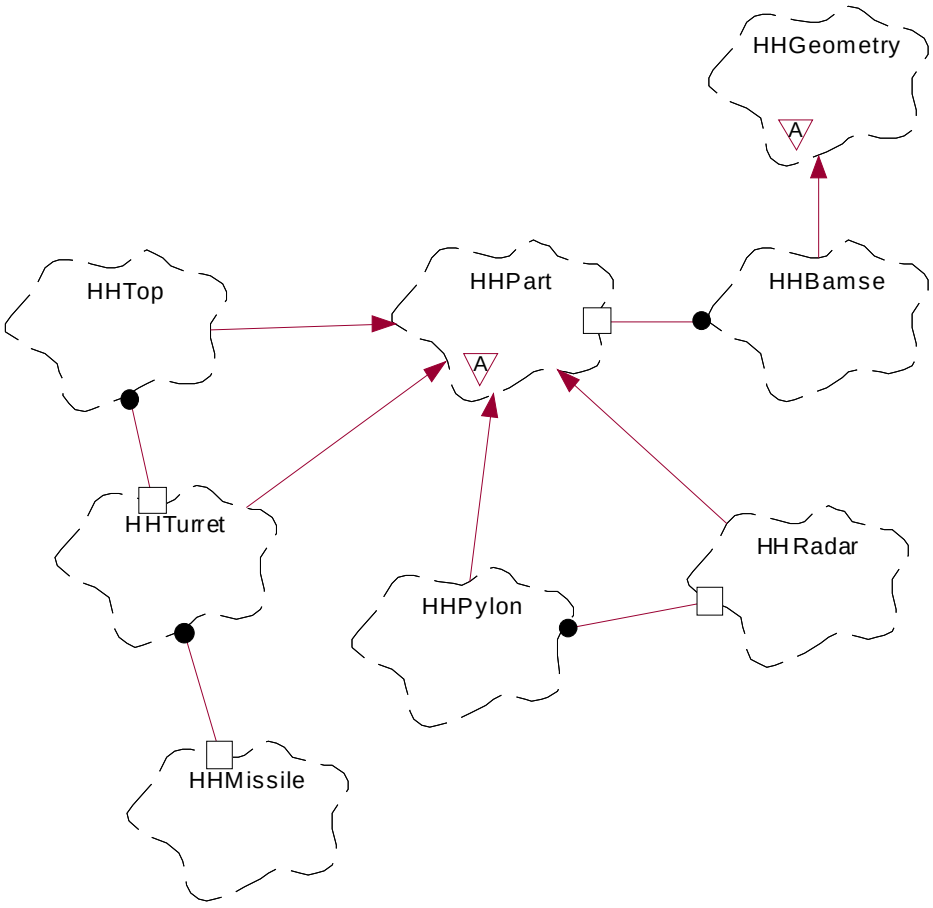


Figure 5.7 – Recommended design of the geometry classes

Each of the movable parts is separated into different classes. In this manner each class handles it's own geometry, instead of one large class handling all of the different geometries. HHBamse still loads the geometry from a single file, separates the geometry into the different movable parts and instances classes that encapsulates the parts. The HHVirtualManual still only has access to HHBamse and thus HHBamse has to coordinate the behaviors of the different parts.

5.3 Execution

When the application is executed, HHVirtualManual initiates all WTK attributes and instantiates an HHBamse and an HHMouseHandler. The HHBamse instance then loads the BAMSE-geometry and initiates the movable parts of the geometry (as described above). WTK has an internal program loop that every application must follow. This is illustrated in *Figure 5.8*.

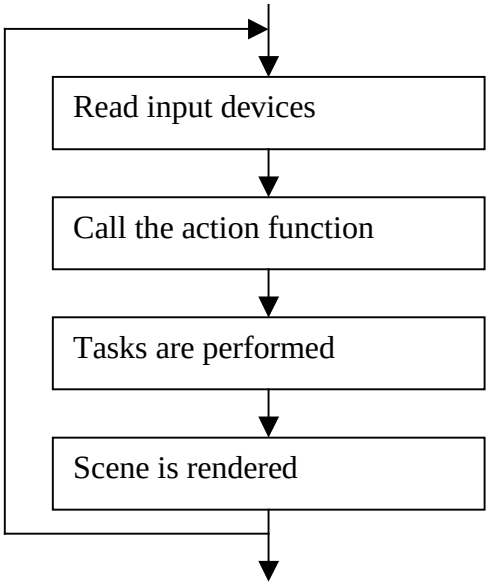


Figure 5.8 – The WTK program loop

This program loop, designed for C programs, poses a problem when making an object-oriented design. The universe's action function, located in box number two in *Figure 5.8*, is the major function that is called in each program loop. In this function all logical operations have to be performed. The problem is that C++ does not have functions, it has methods that are part of classes, and WTK expects a function. The solution is to either write a special function for the program loop, and disregard object-oriented rules, or include the program loop as a static method in some class. I chose the latter approach in which the action function is implemented as a static method in HHVirtualManual.

In the action function the keyboard is read and corresponding actions are taken (as described in section 5.1). After that HHVirtualManual tells the HHMouseHandler to check the mouse. If anything interesting happens to the mouse, the HHMouseHandler will inform the HHVirtualManual about this via the HHListener/HHScreamer relationship.

If, for example, a missile turret is to be elevated, or de-elevated, the HHBamse assigns a task to that movable part. A task is a *function* (in functional programming terms), thus the C++ programmer has to use a static method as described above regarding the action function. This function executes outside the normal program loop (or action function). All tasks are executed after the action function, but before the scene is rendered. This means that changes to objects can be made automatically through the use of a task. I have used this in such a way that each task is assigned to a specific geometry, for example a missile turret, and then rotates it a little bit in each program loop, until a given value has been reached, e.g. the turret is fully elevated. This results in the effect of the missile turret elevating itself after a user has double

clicked on it. When the application enters the demo mode, a special task is assigned to handle all the functionality described in 5.1 (i.e. the zooming, the rotations, etc.).

The source code of the testbed, where each method and class is commented, is included in Appendix B.

5.4 Connection between WorldToolKit's and I-DEAS' coordinate systems

One task in the evaluation of WTK was to find a method to locate a specific point in the coordinate system for a specific geometry, given the global coordinates of this point in I-DEAS (described in requirement 3.2.6). The following procedure accomplishes that task. Note that the whole scene has a *global* coordinate system, and each geometry node has a *local* coordinate system of its own.

1. Calculate the full transformation for the given geometry node (there are functions in WTK to do this). The result is the coordinate of the local coordinate system's origin in the global coordinate system.
2. Inverse the result of step 1 (-x, -y, -z) to get the coordinate of the global coordinate system's origin in the local coordinate system.
3. Subtract the rotation-point's coordinate (the one taken from the global coordinate system of I-DEAS) from the result of step 2.
4. Translate the result of step 3 so that it is located in the origin of the local coordinate system.
5. Perform the actual rotation that is the goal of this procedure. This usually is around one or more of the three axes X, Y and Z.
6. Perform an inverse translation of the one in step 4, i.e. translate the geometry back to its original position.

Steps 4 to 6 are actually the same procedure as the rotation in any chosen point described in section 4.1.4 (the evaluation of WTK). The only difference is that a coordinate from I-DEAS is used, after a conversion.

5.5 Implementation issues

There are some issues that are important to keep in mind when developing applications using WTK. I ran into four of them and they are discussed in the following sections.

5.5.1 Coordinate systems

WTK uses the coordinate system shown in *Figure 5.9*. When compared to the coordinate system shown in the introduction (*Figure 1.1*), one can see that the y-axis is inverted. This can cause some problems when importing models from formats that use another coordinate system. An example of this is VRML 1.0, which uses the previously mentioned coordinate system (with an y-axis pointing up). The result of this is that when one imports a geometry from a VRML 1.0-file, this geometry is displayed up-side-down in WTK. An easy solution to this problem is to rotate the geometry 180° around the z-axis (or the x-axis) directly after loading it. However, this problem does not exist when loading the geometry from a VRML 2.0-file!

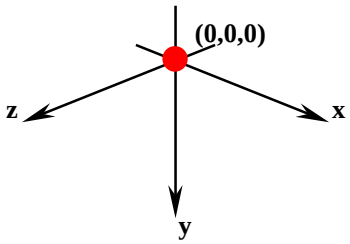


Figure 5.9 – WTK's coordinate system

There is another occasion when it is important to keep the coordinate system in mind. In WTK it is possible to draw 2D objects (images, points, lines, text, etc.) on top of the 3D scene. This is very useful when one wants to textually display information along with the 3D scene, or when one is creating a *heads up display* (HUD). The important matter is that the 2D coordinate system is somewhat strange; the lower left corner has the coordinate (0.0, 0.0) and the upper right corner has the coordinate (1.0, 1.0). Usually the origin of such coordinate systems is located in the top left corner of the screen. If one just keeps this in mind, this should not be a problem. This kind of relative coordinates is otherwise very useful because there is no need to keep track of what size the current window has, since the upper right corner always has the coordinates (1.0, 1.0).

5.5.2 Viewpoints

It is important to keep in mind that some file formats contain more information, or objects, than what is displayed. An example of this is that VRML-files can contain viewpoints, called *Cameras* in VRML. When a VRML-file that contains a camera is imported into WTK, this camera is added to the universe's list of viewpoints. However, it cannot be modified as a regular viewpoint. It is for example not possible to get any information about the viewpoint's (camera's) location or to move it, which can cause quite annoying errors.

5.5.3 Project settings

It is important to have correct project settings. There are different settings for different types of projects. These are all described in chapter 3 of the WorldToolKit Hardware Guide.

5.5.4 Flaws and errors

WTK is not 100% correct and still contains flaws and errors, especially the C++ wrapper-classes. Sense8's homepage (*http://www.sense8.com*) provides the updates. If one encounters an error in WTK it is important to notify Sense8, so that they can fix the error.

6 Recommendations

I have reached the conclusion that three development environments are suitable for SAAB Bofors Dynamics' VR development; WTK, VTREE and COSMO 3D/OPENGL OPTIMIZER combined with VR JUGGLER. Each of the three has its positive and negative aspects and these are mentioned in the evaluations in chapter 4.

6.1 WorldToolKit

WTK is a complete VR development package that has all the functionality that are needed. SAAB Bofors Dynamics already uses software systems from the same vendor, and it is preferable that the whole software suit originates from the same company. WTK's developer also offers a CAD conversion add-on, which allows WTK to import CAD data directly from I-DEAS, which is of great value for the development of VR applications. This is the strong point of WTK. It is a great tool for VR prototyping of products. On the other hand, WTK has some flaws in its object-oriented structure that can be a nuisance when developing applications. WTK also has a performance that is worse than for instance VTREE. When developing large applications that run with VR hardware such as HMDs, this performance loss can lead to problems such as simulator sickness.

6.2 VTree

VTREE is slightly more flexible than WTK. However, it does not have as much VR support. While WTK is excellent for product prototyping, VTREE is more directed towards large-scale simulations, such as environmental visualizations. VTREE also lacks the import capabilities that WTK offers. Since SAAB Bofors Dynamics already uses VTREE in simulations and visualizations, it could be practical to also use VTREE in the VR development as well.

6.3 Cosmo 3D, OpenGL Optimizer and VR Juggler

The combination of COSMO 3D/OPENGL OPTIMIZER and VR JUGGLER provides the same flexibility as VTREE. It also provides the same, or probably better, VR hardware support as WTK. What the combination lacks are the special effects included in the VTREE SDK, and the import capabilities of WTK. Other than that COSMO 3D/OPENGL OPTIMIZER and VR JUGGLER

offer a solid object-oriented foundation for developing dynamic VR applications. The fact that the three APIs are also completely free is also a positive aspect, although the purchase costs are usually not an issue for a company of SAAB Bofors Dynamics' size.

6.4 Summary

I cannot really point out one development environment that I recommend SAAB Bofors Dynamics' to use. It all depends on the priority. WTK is great for prototyping and importing CAD data, VTREE provides better performance, but are directed towards a slightly different use, and COSMO3D/OPENGL OPTIMIZER/VR JUGGLER lies somewhere in between the two. WTK is developed by the same vendor as other systems already in use, VTREE is already used in other developments, while SAAB Bofors Dynamics has no previous experience with COSMO3D/OPENGL OPTIMIZER/VR JUGGLER.

References

[1] Michael Louka. *An Introduction to Virtual Reality*. Østfold College, 3rd revision, 1998. *http://w1.2691.telia.com/~u269100246/vr/vrhiof98/*

[2] Engineering Animation, Inc. *WorldToolKit® Reference Manual*. Release 9, 1999.

[3] Sense8 Corporation. *WorldToolKit™ Release 8 Technical Overview*. Release 8, 1998.

[4] Allen Bierbaum and Christopher Just. *Software Tools for Virtual Reality Application Development*. Iowa Center for Emerging Manufacturing Technology, Iowa State University, 1998.

[5] CG2, Inc. *VTree User's Manual*. 5th edition, 1999.

[6] George Eckel. *Cosmo 3D™ Programmer's Guide*. Silicon Graphics, Inc., 1998.

[7] Realax Software. *Realax Reference Manual*. 1999.

[8] Rikk Carey and Gavin Bell. *The Annotated VRML 2.0 Reference Manual*. Addison-Wesley Developers Press, 1997.

[9] Daniel K. Schneider and Sylvere Martin-Michiellot. *VRML Primer and Tutorial*. TECFA, Faculte de Psychologie et des sciences de l'education, University of Geneva, Draft version 1.1a, 1998. *http://tecfa.unige.ch/guides/vrml/vrmlman/vrmlman.html*

[10] ParallelGraphics. *Cortona Software Developers Kit Documentation*.1999.

[11] Christopher Just, Allen Bierbaum, Albert Baker and Carolina Cruz-Neira. *VR Juggler: A Framework for Virtual Reality Development*. Iowa Center for Emerging Manufacturing Technology, Iowa State University, 1998.

A Future features, implementing Virtual Reality hardware

The computer market has always been turbulent and the VR market is no exception. Vendors and products that are popular today might be gone tomorrow. Because of this we have chosen only to include vendors with a good reputation in the following product overview. We have also chosen only to include prices that we are certain are up to date, thus we only include prices that have been given to us directly from the vendor or an distributor.

There are three types of hardware included, that are of interest to SAAB Bofors Dynamics. These are HMDs, gloves and trackers (all of these are introduced in chapter 1.3). There are a plentitude of other devices available on the VR market, but judging from the preliminar guidelines we have been given from SAAB Bofors Dynamics, these three types are all that is needed.

A.1 Head Mounted Displays

Vendor	Products	Price-class
Virtual Research	V6	$6,900
	V8	$11,900
Interactive Imaging	VFX3D	$1,795
i-O Display Systems	I-glasses LC	$399
	I-glasses	$499
	I-glasses X2	$799
	I-glasses ProTec	$4,000
O I P	HOPROS	
Ericsson SAAB Avionics AB	AddVisor 100	160,000 SEK
Virtual Vision Inc.	V-Cap 1000	$1,500
	eGlass	$4,000
Kaiser Electro Optics	HiDef 60º	
	HiDef 90º	
	ProView 30	$7,995
	ProView 40ST	
	ProView 50ST	
	ProView 60	$11,995
	ProView 80	
	ProView XL35	
	ProView XL50	
StereoGraphics	CrystalEyes	$795
	CrystalEyes Wired	$299

Table A.1 – Head Mounted Displays

A.2 Gloves

Vendor	Products	Price-class
iReality, Inc.	5th Glove 5th Glove – 14 Sensor	$500 - $2,000 ~$4,000
Virtual Technologies	CyberGlove	
Fakespace	Pinch Gloves	

Table A.2 - Gloves

A.3 Trackers

Vendor	Products	Price-class
Ascension Technology	Flock of Birds MotionStar MotionStar Wireless pcBIRD SpacePad	
Polhemus	FastTrak IsoTrak II Star*Trak	
InterSense	IS-300 IS-600 Mark 2 Plus IS-900 InterTrax	

Table A.3 - Trackers

B Glossary

3D, 3-dimensional

API, Application Programming Interface

Bamse, an anti-aircraft defence system that SAAB Bofors Dynamics develops

CAD, Computer Aided Design

CAE, Computer Aided Engineering

Cave™, a VR projection system, where a cube surrounds the user with projections on several, or all, of the walls

Direct3D, a 3D graphics API developed by Microsoft

Directed acyclic graph, a graph, e.g. a scene graph, that is directed, i.e. it's nodes direct to order of the graph, and acyclic, i.e. there cannot be cycles in the node structure

EAI, External Authoring Interface

External Authoring Interface, an interface that allows external programs to communicate with VRML scenes

Gizmo, a wizard, i.e. a program guide, in Realax

Glove, a VR input device consisting of a glove equipped with sensors

GUI, Graphical User Interface

Head Mounted Display, a pair of glasses, or a helmet, that utilizes stereographic viewing to produce a visual 3D effect

HMD, Head Mounted Display

HTTP, HyperText Transfer Protocol

OpenGL, *Open Graphics Library*, low-level graphics API developed by SGI. Industry standard used in several CAD and 3D applications

Open source, the source code is freely available for anyone who wants it

I-DEAS, a CAD-system that SAAB Bofors Dynamics uses. Homepage: *http://www.sdrc.com*

Immersion, the cognitive conviction of being 'inside' a 3D scene

PROTO-node, a prototype node in VRML that is used to describe other specialized nodes

Render, to generate an image, from a 3D scene, that will be displayed on a 2D screen

Scene graph, a hierarchical representation of a 3D graphics scene

SDK, Software Development Kit

SGI, Silicon Graphics, Inc.

Shutter glasses, see *stereo glasses*

Simulator sickness, disturbances, such as headaches, nausea and vomiting, produced by simulators

Stereo glasses, LCD screens that are linked to the frame rate of a monitor to produce a stereographic effect

Stereographic viewing, the use of two images of the same scene, where one is presented to the left eye and the other to the right eye

Tracker, a VR input device that tracks the user's motions

Translation, the position of a geometry in a coordinate system; *to translate*, to position a geometry in the coordinate system

Vertex, The smallest component of a 3D scene, consisting of just one point

VR, Virtual Reality

VRML, Virtual Reality Modeling Language

WTK, WorldToolKit

WWW, World Wide Web

C Source code

This appendix contains all the source code that have been written during this project. The source code is divided into different sections, based on the design described in chapter 6.2. Each section contains the header file as well as the source code file, if one exists (abstract classes only have headers files). Each file begins with a comment describing the content, as shown at the top of section C.1 below.

All the classes and methods are commented directly in the source code. Thus there are no need for more descriptions.

C.1 The main function

```
////////////////////////////////////////////////////////////////////////////////
// main.cpp - The main-function                                               //
////////////////////////////////////////////////////////////////////////////////

#include <iostream.h>
#include "HHVirtualManual.h"


/*** Global variables ***/
HHVirtualManual *application;          // The application (must be global so that the
                                       // global (static) action-functions can access it.


/**  The main-function, which instantiates HHVirtualManual, i.e. the whole application,
      and then runs it. At the end it shows whether the execution was a success or not.
  */
int main(void)
{
     application = new HHVirtualManual;

     if (0 == application->run())
     {
            cout << "Application successfully executed." << endl;
            return 0;
     }
     else
     {
            cout << "Failed to properly execute application." << endl;
            return -1;
     }
}
```

C.2 The application

```
////////////////////////////////////////////////////////////////////////////////
// HHVirtualManual.h - The application                                        //
////////////////////////////////////////////////////////////////////////////////

#ifndef __HH_VIRTUAL_MANUAL_H_
#define __HH_VIRTUAL_MANUAL_H_


#include <math.h>
```

```
#include "wt.h"
#include "wtcpp.h"
#include "HHBamse.h"                    // Include one of these depending on what geometry
//#include "HHRotation.h"               // that should be used.
//#include "HHRotation2.h"
//#include "HHRotation97.h"
#include "HHListener.h"
#include "HHMouseHandler.h"


/*** Constants **********/
#define LIGHTS_FILENAME "lights"            // Filename for the light-data

#define BG_RED 150                          // Background-color for the window
#define BG_GREEN 150
#define BG_BLUE 210

#define WIN_XPOS 0                          // Position and size for the window.
#define WIN_YPOS 0
#define WIN_WIDTH 800
#define WIN_HEIGHT 600


/*** Enumerations *******/
enum HHManualState {BAMSE_IDLE, BAMSE_WHOLE, BAMSE_TURRET, BAMSE_POLE, BAMSE_SPEC};

/**  The actual application which includes all necessary functions (WTK unfortunately
       needs some thing done in C-style) and methods for the program-loop and the
       interactivity. The class implements the HHListener-interface, which is necessary
       for effective communication with the mouse-handler (HHMouseHandler).
  */
class HHVirtualManual : public HHListener
{
      public:
            HHVirtualManual();
            ~HHVirtualManual();

            int run(bool displayTree = false);
            void listen(void* data);
            void setManState(HHManualState newState);
            void setSpecMode();

            static void HHActionFn();
            static void HHDrawFn(WtWindow* win, FLAG eye);
            static void HHFgDrawFn(WtWindow* win, FLAG eye);

            static void HHDemoTask(void *myApp);

      private:
            void displaySplash();
            void updateMHandler();
            void resetViewport();
            void rotateViewport(float radians);
            void zoomViewport(float transZ);
            void completeScene();

            void rockNroll(int thatOne);
            void demoMode();
            void radar(int part, int positive);

            HHGeometry *myGeometry;         // A geometry-class of some sort
            WtRoot *myRoot;                 // The root-node to the primary scene-graph
            WtViewPoint *myView;            // The viewpoint to the primary scene-graph
            WtWindow *myWindow;             // The window to the primary scene-graph
            HHMouseHandler *myMHandler;     // The mouse-handler (which is a HHScreamer)
            WtP3 origPos;                   // Original position of the viewpoint
            WtQ origOrient;                 // Original orientation of the viewpoint

            HHManualState currManState;     // Current state of the 'manual'

            BOOL demoOn;                    // Is the application in demo-mode?
            WtTask *demoTask;               // The task that handles the demo-animations
            int viewPortCount, currentTurret,
                  radarCount, headCount;    // Counters for the demo-loop
            BOOL countUp, radarUp, headUp;  // Direction-flags (up/down) for the counters
};
```

```
#endif


////////////////////////////////////////////////////////////////////////////////
// HHVirtualManual.cpp - The application                                      //
////////////////////////////////////////////////////////////////////////////////

#include "HHVirtualManual.h"


/*****************************************************************************/
/*** PUBLIC METHODS...                                                     ***/
/*****************************************************************************/

/**  The default-constructor. Creates a new universe, a new geometry, and a mouse-handler
     (to which it registers itself as a listener).
  */
HHVirtualManual::HHVirtualManual()
{
    WtUniverse::New(WTDISPLAY_DEFAULT, WTWINDOW_DEFAULT | WTWINDOW_NOBORDER);

    myGeometry = new HHBamse;
    myMHandler = new HHMouseHandler(this);

    currManState = BAMSE_IDLE;   // HHBamse-specific!!!
    demoOn = FALSE;
    demoTask = NULL;
}

/**  The destructor. Releases the memory used by the instance, it's geometry and mouse-
     handler.
  */
HHVirtualManual::~HHVirtualManual()
{
    delete myMHandler;
    delete myGeometry;
}


/**  Activates the application. Initiates the variables that is used by the application.
     HHVirtualManual's static function HHActionFn is registered as the universe's
     action function.
     The parameter specifies whether or not a scene-tree should be printed to the screen.
     The return value is the result of the execution (0 = fine, -1 = bad).
  */
int HHVirtualManual::run(bool displayTree)
{
    myRoot = WtUniverse::GetFirstRootNode();
    myView = WtUniverse::GetFirstViewPoint();
    myWindow = WtUniverse::GetFirstWindow();

    myWindow->SetBgRGB(BG_RED, BG_GREEN, BG_BLUE);
    myWindow->SetPosition(WIN_XPOS, WIN_YPOS, WIN_WIDTH, WIN_HEIGHT);

    myRoot->LightNodeLoad(LIGHTS_FILENAME);

    if (myGeometry->addAsChild(myRoot))
    {
          if (displayTree) myRoot->Print();

          myWindow->ZoomViewPoint();
          myView->GetPosition(origPos);
          myView->GetOrientation(origOrient);

          WtUniverse::SetActions(HHVirtualManual::HHActionFn);
          myWindow->SetFgActions(HHVirtualManual::HHFgDrawFn);
          //myWindow->SetDrawFn(HHVirtualManual::HHDrawFn); // Uncomment this line if a

      // picture should be used as the

      // background.
          WtKeyboard::Open();
          WtUniverse::Ready();
          displaySplash();

          WtUniverse::Go();
```

```
            WtKeyboard::Close();
            WtUniverse::Delete();

            return 0;
      }
      else
            return -1;
}

/**  Since HHVirtualManual implements the HHListener-interface this method is needed.
      It enables the transfer of data between the mouse-handler (a HHScreamer) and the
      application. The information that is received from the mouse-handler is a
      HHMouseData-struct, which is sent as an void*. The information is extracted and
      relevant actions are taken, for example rotation of the geometry.
  */
void HHVirtualManual::listen(void* data)
{
      float transX, transY;
      int x0, y0, width, height;
      WTpoly *pickedPoly;
      WtNodePath *nodePath;
      WtP2 point;
      WtP3 point3D;

      if (!demoOn)
      {
            transX = (float)((HHMouseData*)data)->deltaX;
            transY = (float)((HHMouseData*)data)->deltaY;

            if (NONE == ((HHMouseData*)data)->doubleClick)
            {
                  if (((HHMouseData*)data)->leftButton)
                        myGeometry->rotate(transY, transX, 0);
                  if ((((HHMouseData*)data)->rightButton)&&(0 != transY))
                        zoomViewport(transY);
                  if ((((HHMouseData*)data)->middleButton)&&(0 != transX))
                        myGeometry->rotate(transX, transY, 5);
            }
            else
            {
                  switch (((HHMouseData*)data)->doubleClick)
                  {
                        case LEFT:       myWindow->GetPosition(&x0, &y0, &width, &height);
                                         point[0] = transX - x0;
                                         point[1] = transY - y0;
                                         pickedPoly = myWindow->PickPoly(point, &nodePath,
point3D);
                                         if (NULL != pickedPoly)
                                               myGeometry->chkNode(nodePath);
                                         delete nodePath;
                                         break;

                        case MIDDLE:     resetViewport();
                                         break;
                  }
            }
      }
}

/**  Sets the state of the manual, i.e. the textual content that are displayed on the
      screen.
  */
void HHVirtualManual::setManState(HHManualState newState)
{
      currManState = newState;
}

/**  Initiates the specification-mode, which shows the specifications on the screen, as
      well as resets the geometry.
  */
void HHVirtualManual::setSpecMode()
{
      myGeometry->nullAll(FALSE);
      resetViewport();
      currManState = BAMSE_SPEC;
}
```

```
/*****************************************************************************/
/*** STATIC METHODS...                                                     ***/
/*****************************************************************************/

/**  Is used as the WTK universe's action-function, which is executed every program-loop.
      It reads the keyboard and activates events considering this, and tells the
      application to update the mouse-handler.
  */
void HHVirtualManual::HHActionFn()
{
    extern HHVirtualManual *application;
    short key = WtKeyboard::GetLastKey();

    switch (key)
    {
          case 27:
          case 'Q':
          case 'q':  WtUniverse::Stop();
                     break;

          case 'C':
          case 'c':  if (!application->demoOn) application->completeScene();
                     break;

          case 'Z':
          case 'z':  if (!application->demoOn)
                     {
                          application->myGeometry->nullAll(TRUE);
                          application->resetViewport();
                     }
                     break;

          case 'H':
          case 'h':  application->setManState(BAMSE_IDLE);
                     break;

          case 'S':
          case 's':  if (!application->demoOn) application->setSpecMode();
                     break;

          case 'F':
          case 'f':  WTmessage("Current framerate: %f\n", WtUniverse::FrameRate());
                     break;

          case 1003: if (!application->demoOn) application->radar(6, 1);
                     break;

          case 1002: if (!application->demoOn) application->radar(6, -1);
                     break;

          case 1000: if (!application->demoOn) application->radar(7, 1);
                     break;

          case 1001: if (!application->demoOn) application->radar(7, -1);
                     break;

          case '1':
          case '2':
          case '3':
          case '4':  if (!application->demoOn) application->rockNroll((int)key - 48);
                     break;

          case ' ':  application->demoMode();
                     break;

          //default: WTmessage("%d\n", key);
    }

    application->updateMHandler();
    application->myWindow->SetFgActions(HHVirtualManual::HHFgDrawFn);
}

/**  A function that displays an image behind the 3D scene.
  */
void HHVirtualManual::HHDrawFn(WtWindow* win, FLAG eye)
{
    win->LoadImage("sky3.tga", -0.999f, FALSE, TRUE);
```

```
}

/**  The function that draws the 2D-content (text information) on top of the 3D scene.
  */
void HHVirtualManual::HHFgDrawFn(WtWindow* win, FLAG eye)
{
    extern HHVirtualManual *application;

    win->Set2DColor(255, 255, 255);
    win->Draw2DText(0.f, 0.005f, "CREATED USING WORLDTOOLKIT, FROM SENSE8");

    if (application->demoOn)
    {
           win->Set2DColor(255, 0, 0);
           win->Draw2DText(0.89f, 0.005f, "DEMO MODE");
    }
    win->Set2DColor(0, 0, 0);
    switch(application->currManState)
    {
           case (BAMSE_IDLE):          win->Draw2DText(0.01f, 0.95f, "Välkommen till BAMSE
VirtualManual");
                                              win->Draw2DText(0.01f, 0.94f, "------------------
-------------------------------------------");

                                              win->Draw2DText(0.01f, 0.91f, "Mus:");
                                              win->Draw2DText(0.01f, 0.89f, "     Vänster
nedtryckt - Rotera");
                                              win->Draw2DText(0.01f, 0.87f, "     Vänster dbl
klick - Aktivera");
                                              win->Draw2DText(0.01f, 0.85f, "     Mitten
nedtryckt - Rotera lavett");
                                              win->Draw2DText(0.01f, 0.83f, "     Höger
nedtryckt - Zooma");
                                              win->Draw2DText(0.01f, 0.79f, "Tangentbord:");
                                              win->Draw2DText(0.01f, 0.77f, "     1-4 - Avfyra
missil");

                           win->Draw2DText(0.01f, 0.75f, "     pilar - Rotera radar");

                                              win->Draw2DText(0.01f, 0.73f, "     space - Demo
mode av/på");
                                              win->Draw2DText(0.01f, 0.71f, "     z - Nollställ
vyn");
                                              win->Draw2DText(0.01f, 0.69f, "     c - Anpassa
till fönster");
                                              win->Draw2DText(0.01f, 0.67f, "     h - Visa
denna hjälp");
                                              win->Draw2DText(0.01f, 0.65f, "     s - Visa
specificationer");
                                              win->Draw2DText(0.01f, 0.63f, "     q / Esc -
Avsluta");
                                              break;

           case (BAMSE_WHOLE):         win->Draw2DText(0.01f, 0.95f, "Bofors RBS 23 BAMSE -
Luftvärnsrobotsystem");
                                              win->Draw2DText(0.01f, 0.94f, "------------------
-------------------------------------------");

                                              win->Draw2DText(0.01f, 0.91f, "År 1993 beställde
den svenska regeringen");
                                              win->Draw2DText(0.01f, 0.89f, "fullskalig
utveckling av luftvärnsrobotsystemet RBS");
                                              win->Draw2DText(0.01f, 0.87f, "23 BAMSE. BAMSE-
systemet är ett");
                                              win->Draw2DText(0.01f, 0.85f, "samarbetsprojekt
mellan Bofors Missiles och");
                                              win->Draw2DText(0.01f, 0.81f, "Ericsson Microwave
Systems i vilket Bofors har");
                                              win->Draw2DText(0.01f, 0.79f, "det övergripande
systemansvaret. Serieproduktion");
                                              win->Draw2DText(0.01f, 0.77f, "kommer att
påbörjas vid sekelskiftet.");

                                              win->Draw2DText(0.01f, 0.73f, "BAMSE har
allväderskapacitet och en räckvidd");
                                              win->Draw2DText(0.01f, 0.71f, "som överstiger det
maximala standoff-avståndet för");
```

```
                                        win->Draw2DText(0.01f, 0.69f, "elektro-optiskt
styrda vapen. Med en effektiv");
                                        win->Draw2DText(0.01f, 0.67f, "höjdtäckning på
upp till 15 km och en räckvidd på");
                                        win->Draw2DText(0.01f, 0.65f, "mer än15 km är
BAMSE-systemet lämpligt inte");
                                        win->Draw2DText(0.01f, 0.63f, "bara för skydd av
vitala militära objekt och rörliga");
                                        win->Draw2DText(0.01f, 0.61f, "marktrupper, utan
också för skydd av infrastruktur");
                                        win->Draw2DText(0.01f, 0.59f, "av vital betydelse
för hela nationen. I sådana fall då");
                                        win->Draw2DText(0.01f, 0.57f, "en standoff-missil
avfyras från ett flygplan utanför");
                                        win->Draw2DText(0.01f, 0.55f,
"luftförsvarssystemets räckvidd har BAMSE");
                                        win->Draw2DText(0.01f, 0.53f, "fortfarande
förmåga att bekämpa den attackerande");
                                        win->Draw2DText(0.01f, 0.51f, "missilen.");
                                        break;

          case (BAMSE_TURRET):   win->Draw2DText(0.01f, 0.95f, "Robot 23 - BAMSE
höghastighetsmissil");
                                        win->Draw2DText(0.01f, 0.94f, "----------------
-----------------------------------------------");

                                        win->Draw2DText(0.01f, 0.91f, "BAMSE- missilen
har en snabb acceleration och");
                                        win->Draw2DText(0.01f, 0.89f, "därav följande hög
hastighet, vilket resulterar i kort");
                                        win->Draw2DText(0.01f, 0.87f, "flygtid även på
långa avstånd. Inom kortare tid än");
                                        win->Draw2DText(0.01f, 0.85f, "en sekund
accelererar missilen upp till en");
                                        win->Draw2DText(0.01f, 0.81f, "hastighet av mer
än 3000 km/h och bibehåller");
                                        win->Draw2DText(0.01f, 0.79f, "sedan sin
hastighet och stora manövrerbarhet");
                                        win->Draw2DText(0.01f, 0.77f, "ända till gränsen
för sin räckvidd.");

                                        win->Draw2DText(0.01f, 0.73f, "Missilens
högeffektiva stridsdel initieras av");
                                        win->Draw2DText(0.01f, 0.71f, "antingen det
integrerade allväderszonröret eller vid");
                                        win->Draw2DText(0.01f, 0.69f, "direktträff ett
anslagsrör. Den kombinerade effekten");
                                        win->Draw2DText(0.01f, 0.67f, "av splitter och
tryckverkan har effekt mot alla typer");
                                        win->Draw2DText(0.01f, 0.65f, "av flygande mål,
från små snabba mål som t ex");
                                        win->Draw2DText(0.01f, 0.63f, "signalsökande
attackrobotar eller kryssnings-");
                                        win->Draw2DText(0.01f, 0.61f, "missiler och till
stora långsamma mål som t ex");
                                        win->Draw2DText(0.01f, 0.59f, "transportplan.");
                                        break;

          case (BAMSE_POLE):          win->Draw2DText(0.01f, 0.95f, "Eldledningsradar");
                                        win->Draw2DText(0.01f, 0.94f, "----------------
----------");

                                        win->Draw2DText(0.01f, 0.91f, "Eldledningsradar,
IRV-kamera, IK (igenkännings-");
                                        win->Draw2DText(0.01f, 0.89f, "utrustning) och
vädersensor är monterade på en");
                                        win->Draw2DText(0.01f, 0.87f, "gyrostabiliserad
plattform på toppen av en höj- och");
                                        win->Draw2DText(0.01f, 0.85f, "sänkbar 8 meters
mast. Roboten styrs i banan av");
                                        win->Draw2DText(0.01f, 0.83f, "en
eldledningsradar som är en vidareutveckling av");
                                        win->Draw2DText(0.01f, 0.81f, "Ericssons Eagle-
radar och som arbetar på Ka-");
                                        win->Draw2DText(0.01f, 0.79f, "bandet, dvs 34-35
GHz.");
                                        break;
```

```
           case (BAMSE_SPEC):          win->Set2DColor(39, 255, 25);
                                       win->Draw2DText(0.38f, 0.95f, "BOFORS BAMSE -
SPECIFIKATION");
                                       win->Draw2DText(0.38f, 0.94f, "----------------
-------------------------------------");

                                       win->Draw2DText(0.001f, 0.50f, "BAMSE Missile");
                                       win->Draw2DText(0.001f, 0.49f, "----------------
--------");


                                       win->Draw2DText(0.001f, 0.46f, "Hastighet:");
                                       win->Draw2DText(0.001f, 0.42f, "Manöverbarhet:");
                                       win->Draw2DText(0.001f, 0.38f, "Effektiv räckvidd
(robot):");
                                       win->Draw2DText(0.001f, 0.36f, "Effektiv
höjdtäckning (robot):");
                                       win->Draw2DText(0.001f, 0.34f, "Styrning:");
                                       win->Draw2DText(0.001f, 0.32f, "Verkansdel:");
                                       win->Draw2DText(0.001f, 0.28f, "Tändrör:");
                                       win->Draw2DText(0.001f, 0.26f, "Måltyper:");


                                       win->Draw2DText(0.25f, 0.46f, "Hög hastighet hela
den effektiva");
                                       win->Draw2DText(0.25f, 0.44f, "räckvidden");
                                       win->Draw2DText(0.25f, 0.42f, "Mycket stor inom
den effektiva");
                                       win->Draw2DText(0.25f, 0.40f, "räckvidden");
                                       win->Draw2DText(0.25f, 0.38f, "15 km (+)");
                                       win->Draw2DText(0.25f, 0.36f, "Upp till 15.000
meter");
                                       win->Draw2DText(0.25f, 0.34f,
"Siktlinjestyrning");
                                       win->Draw2DText(0.25f, 0.32f, "Kombinerad
funktion splitter- och");
                                       win->Draw2DText(0.25f, 0.30f, "RSV-laddning");
                                       win->Draw2DText(0.25f, 0.28f, "Zonrör och
anslagsrör");
                                       win->Draw2DText(0.25f, 0.26f, "Attackflyg, bomb-
och transportplan,");
                                       win->Draw2DText(0.25f, 0.24f, "attack- och
transporthelikoptrar,");
                                       win->Draw2DText(0.25f, 0.22f, "standoff missiler
(Kryssnrb, Ssarb,");
                                       win->Draw2DText(0.25f, 0.20f, "etc.) och
laserstyrda bomber");


                                       win->Draw2DText(0.52f, 0.88f, "BAMSE MCC");
                                       win->Draw2DText(0.52f, 0.87f, "----------------
---");


                                       win->Draw2DText(0.52f, 0.84f, "Funktion:");
                                       win->Draw2DText(0.52f, 0.82f, "Hydda:");
                                       win->Draw2DText(0.52f, 0.78f, "Skydd:");
                                       win->Draw2DText(0.52f, 0.74f, "Besättning:");
                                       win->Draw2DText(0.52f, 0.72f, "Målföljningsradar
(BEER):");
                                       win->Draw2DText(0.52f, 0.68f, "Frekvens:");
                                       win->Draw2DText(0.52f, 0.66f, "Räckvidd:");
                                       win->Draw2DText(0.52f, 0.64f, "Andra sensorer:");
                                       win->Draw2DText(0.52f, 0.62f, "IK:");


                                       win->Draw2DText(0.75f, 0.84f, "Stridsledning och
eldledning");
                                       win->Draw2DText(0.75f, 0.82f, "Dragen, luft-,
sjö- och land-");
                                       win->Draw2DText(0.75f, 0.80f, "transportabel");
                                       win->Draw2DText(0.75f, 0.78f, "Mot splitter och
NBC-");
                                       win->Draw2DText(0.75f, 0.76f, "stridsmedel");
                                       win->Draw2DText(0.75f, 0.74f, "1 - 2
operatörer");
                                       win->Draw2DText(0.75f, 0.72f, "Baserad på
Ericsson Eagle");
                                       win->Draw2DText(0.75f, 0.70f,
"målföljningsradar");
```

```
                                                win->Draw2DText(0.75f, 0.68f, "Ka (K) -band, 34-
35 GHz");
                                                win->Draw2DText(0.75f, 0.66f, "30 km");
                                                win->Draw2DText(0.75f, 0.64f, "IRV-kamera,
vädersensor");
                                                win->Draw2DText(0.75f, 0.62f, "Inbyggd IK-
antenn");
                                                break;

    }
}

/**  This task is run during the demo-mode. It rotates the geometry, zooms the viewpoint in
     and out, shoots missiles, operates the radar (on the geometry) and changes the
     manual-state!
  */
void HHVirtualManual::HHDemoTask(void *myApp)
{
    HHGeometry *temp;

    if (!((HHVirtualManual*)myApp)->demoOn)
    {
          delete ((HHVirtualManual*)myApp)->demoTask;
          ((HHVirtualManual*)myApp)->demoTask = NULL;
    }
    else
    {
          temp = ((HHVirtualManual*)myApp)->myGeometry;
          temp->rotate(0.f, 1.f, 0);
          if (((HHVirtualManual*)myApp)->countUp)
          {
               ((HHVirtualManual*)myApp)->viewPortCount++;
               if (((HHVirtualManual*)myApp)->viewPortCount > 100)
                    ((HHVirtualManual*)myApp)->countUp = FALSE;
               else
                    ((HHVirtualManual*)myApp)->zoomViewport(2.f);
          }
          else
          {
               ((HHVirtualManual*)myApp)->viewPortCount--;
               if (((HHVirtualManual*)myApp)->viewPortCount < -400)
                    ((HHVirtualManual*)myApp)->countUp = TRUE;
               else
                    ((HHVirtualManual*)myApp)->zoomViewport(-2.f);
          }

          if (((HHVirtualManual*)myApp)->radarUp)
          {
               ((HHVirtualManual*)myApp)->radarCount++;
               if (((HHVirtualManual*)myApp)->radarCount > 30)
                    ((HHVirtualManual*)myApp)->radarUp = FALSE;
               else
                    temp->rotate(PI/180, 0.f, 7);
          }
          else
          {
               ((HHVirtualManual*)myApp)->radarCount--;
               if (((HHVirtualManual*)myApp)->radarCount < 0)
                    ((HHVirtualManual*)myApp)->radarUp = TRUE;
               else
                    temp->rotate(-PI/180, 0.f, 7);
          }

          if (((HHVirtualManual*)myApp)->headUp)
          {
               ((HHVirtualManual*)myApp)->headCount++;
               if (((HHVirtualManual*)myApp)->headCount > 90)
                    ((HHVirtualManual*)myApp)->headUp = FALSE;
               else
               {
                    temp->rotate(PI/180, 0.f, 6);
                    //temp->rotate(1.f, 0.f, 5);
               }
          }
          else
          {
               ((HHVirtualManual*)myApp)->headCount--;
```

```
                if (((HHVirtualManual*)myApp)->headCount < -90)
                      ((HHVirtualManual*)myApp)->headUp = TRUE;
                else
                {
                      temp->rotate(-PI/180, 0.f, 6);
                      //temp->rotate(-1.f, 0.f, 5);
                }
          }

          if ((-1 == ((HHVirtualManual*)myApp)->currentTurret)&&((HHBamse*)temp)-
>turretElevation(TRUE))
                ((HHVirtualManual*)myApp)->currentTurret = 1;
          else if ((0 == ((HHVirtualManual*)myApp)->currentTurret)&&((HHBamse*)temp)-
>turretElevation(FALSE))
                ((HHVirtualManual*)myApp)->currentTurret = 3;
          else if (temp->shoot(((HHVirtualManual*)myApp)->currentTurret))
          {
                ((HHVirtualManual*)myApp)->currentTurret++;
                if (3 == ((HHVirtualManual*)myApp)->currentTurret)
                      ((HHVirtualManual*)myApp)->currentTurret = 0;
                else if (5 == ((HHVirtualManual*)myApp)->currentTurret)
                      ((HHVirtualManual*)myApp)->currentTurret = -1;
          }

          if ((-400 > ((HHVirtualManual*)myApp)->viewPortCount)||(100 <
((HHVirtualManual*)myApp)->viewPortCount))
          {
                switch (((HHVirtualManual*)myApp)->currManState)
                {
                      case (BAMSE_WHOLE):         ((HHVirtualManual*)myApp)-
>setManState(BAMSE_TURRET);
                                                        break;

                      case (BAMSE_TURRET):   ((HHVirtualManual*)myApp)-
>setManState(BAMSE_POLE);
                                                        break;

                      case (BAMSE_POLE):          ((HHVirtualManual*)myApp)-
>setManState(BAMSE_SPEC);
                                                        break;

                      case (BAMSE_SPEC):          ((HHVirtualManual*)myApp)-
>setManState(BAMSE_WHOLE);
                                                        break;
                }
          }
    }
}

/*****************************************************************************************/
/*** PRIVATE METHODS...                                                                ***/
/*****************************************************************************************/

/**  Displays a simple welcome-message in the console window and loads the splash screen
      to the render-window.
  */
void HHVirtualManual::displaySplash()
{
    WTmessage("\n\nWelcome to the BAMSE test-session of World ToolKit.\n");
    WTmessage("-------------------------------------------------\n");
    WTmessage("                          Written by Henrik Hedlund.\n\n\n");
    myWindow->LoadImage("bamse.tga", 1.0, TRUE, FALSE);

    for (unsigned int i=0; i<200000000; i++) ;    // No wait-method worked with WTK, and all
                                                        // keyboard-events must be
read from the
                                                        // console window, since
the Universe
                                                        // isn't activated yet...
}

/**  Tells the mouse-handler that it should check the mouse. This method is used
      by the HHActionFn at every program-loop.
  */
void HHVirtualManual::updateMHandler()
{
    myMHandler->examineMouse();
```

```
}

/**  Resets the viewpoint to the starting-point...
  */
void HHVirtualManual::resetViewport()
{
    myView->SetPosition(origPos);
    myView->SetOrientation(origOrient);
}

/**  Rotates the viewpoint according to the transformations along both the X- and the
     Y-axes that the mouse-handler reports!
  */
void HHVirtualManual::rotateViewport(float radians)
{
    WtP3 geoMid, oldPos, newPos;
    float radius, angle, refAngle;

    geoMid = myGeometry->getMidpoint();
    myView->GetPosition(oldPos);

    oldPos -= geoMid;
    radius = sqrtf(oldPos[0]*oldPos[0] + oldPos[2]*oldPos[2]);

    angle = acosf(oldPos[0] / radius);
    refAngle = asinf(oldPos[2] / radius);

    if (0 > refAngle) angle *= -1;
    angle += radians;

    newPos[0] = radius * cosf(angle);
    newPos[1] = oldPos[1];
    newPos[2] = radius * sinf(angle);

    newPos += geoMid;

    myView->SetPosition(newPos);
    myView->AlignAxis(Z, (geoMid-newPos));
}

/**  Zooms the viewpoint in and out of the 3D scene (i.e. zooms along the z-axis). The
     degree of zooming is specified with transZ.
  */
void HHVirtualManual::zoomViewport(float transZ)
{
    WtP3 newPos;

    newPos[0] = 0.0;
    newPos[1] = 0.0;
    newPos[2] = (float)(1 * transZ);
    myView->Translate(newPos, WTFRAME_WORLD);
}

/**  Fits the complete scene in the window.
  */
void HHVirtualManual::completeScene()
{
    myWindow->ZoomViewPoint();
}

/**  Tells the HHBamse-geometry to rotate the radar. "part" specifies which part of the
     radar that shall be rotated, and "positive" specifies if the rotation is positive
     or negative.
  */
void HHVirtualManual::radar(int part, int positive)
{
    float rot;

    if (6 == part)
           rot = (float)(positive * 2.5 * PI/180);
    else
           rot = (float)(positive * 1.5 * PI/180);
    myGeometry->rotate(rot, 0.f, part);
}

/**  Tells the HHBamse-geometry to shoot a missile from the "thatOne"-turret.
  */
```

```
void HHVirtualManual::rockNroll(int thatOne)
{
    myGeometry->shoot(thatOne);
}

/**  Activates and deactivates the demo-mode. Initiates the variable that are needed
     (mostly counters) for the demo-task.
  */
void HHVirtualManual::demoMode()
{
    if (!demoOn)
    {
           demoOn = TRUE;
           viewPortCount =
           radarCount =
           headCount = 0;
           countUp =
           radarUp =
           headUp = TRUE;
           myGeometry->nullAll(FALSE);
           resetViewport();
           setManState(BAMSE_WHOLE);
           if (!((HHBamse*)myGeometry)->isLeft()) ((HHBamse*)myGeometry)-
>turretElevation(TRUE);
           currentTurret = 1;

           demoTask = new WtTask(this, HHVirtualManual::HHDemoTask, 1.f);
    }
    else
           demoOn = FALSE;
}
```

C.3 Listener and screamer

```
////////////////////////////////////////////////////////////////////////////////
// HHListener.h - Listener-interface                                          //
////////////////////////////////////////////////////////////////////////////////

#ifndef __HH_LISTENER_
#define __HH_LISTENER_


/**  The interface that a class must implement to make it possible to use the
     HHListener / HHScreamer-system. A message from a HHScreamer-class is sent to the
     HHListener-class (which has registered itself as a listener at the screamer-class)
     as a void*, which one have to typecast to the information that should be read from
     it.
  */
class HHListener
{
    public:
           virtual void listen(void* data) = 0;
};

#endif



////////////////////////////////////////////////////////////////////////////////
// HHScreamer.h - Screamer-interface                                          //
////////////////////////////////////////////////////////////////////////////////

#ifndef __HH_SCREAMER_H_
#define __HH_SCREAMER_H_

#include "HHListener.h"


/**  The interface that must be implemented to make a class use the HHListener / HHScreamer
     relationship. See HHListener for more information on this.
  */
class HHScreamer
{
```

```
    public:
           virtual void setListener(HHListener* newListener) = 0;
};


#endif
```

C.4 The mouse handler

```
////////////////////////////////////////////////////////////////////////////////
// HHMouseHandler.h - The mouse handler                                       //
////////////////////////////////////////////////////////////////////////////////

#ifndef __HH_MOUSE_HANDLER_H_
#define __HH_MOUSE_HANDLER_H_

#include "wt.h"
#include "wtcpp.h"
#include "HHScreamer.h"


enum HHMsDblClk {NONE, LEFT, MIDDLE, RIGHT};       // Is used to show if there has been
                                                   // a doubleclick

/**  Contains all the information the mouse-handler needs to send to the application.
  */
typedef struct HH_MOUSE_DATA
{
    int          deltaX;
    int          deltaY;
    FLAG         leftButton;
    FLAG         middleButton;
    FLAG         rightButton;
    HHMsDblClk   doubleClick;
} HHMouseData;


/**  A class that is used to handle the mouse instead of WTK's own mouse-routines (which
     are working in a very strange way). The mouse-handler implements the HHScreamer-
     interface, which makes communcation with the application possible.
     The handler reads raw data from the mouse and interprets these in a way that is
     proper for HHVirtualManual!
  */
class HHMouseHandler : public HHScreamer
{
    public:
           HHMouseHandler();
           HHMouseHandler(HHListener *newListener);
           ~HHMouseHandler();

           void examineMouse();
           void setListener(HHListener *newListener);

    private:

           WtMouse *myMouse;
           HHListener *myListener;
           int lastX, lastY;
           FLAG leftClicked, middleClicked, rightClicked;
};


#endif


////////////////////////////////////////////////////////////////////////////////
// HHMouseHandler.cpp - The mouse handler                                     //
////////////////////////////////////////////////////////////////////////////////

#include "HHMouseHandler.h"

/**  The default constructor. Creates a new mouse device and resets all necessary
```

```
     variables.
  */
HHMouseHandler::HHMouseHandler()
{
     myMouse = new WtMouse;
     myListener = NULL;

     leftClicked = FALSE;
     middleClicked = FALSE;
     rightClicked = FALSE;
}

/**  A constructor where the listener (HHListener) is registered directly at the
      instantiation.
  */
HHMouseHandler::HHMouseHandler(HHListener *newListener)
{
     myMouse = new WtMouse;
     setListener(newListener);

     leftClicked = FALSE;
     middleClicked = FALSE;
     rightClicked = FALSE;
}

/**  The destructor. Deallocates the memory used by the instance!
  */
HHMouseHandler::~HHMouseHandler()
{
     delete myMouse;
}

/**  Registers the class (HHListener) that shall listen to the mouse handler (HHScreamer).
  */
void HHMouseHandler::setListener(HHListener *newListener)
{
     myListener = newListener;
}

/**  Examines the mouse and collects all the raw data and analyzes it. The information is
      stored in a HHMouseData struct. If there is any interesting information collected,
      this struct is sent to the listener as a void* (only zero values are not sent).
  */
void HHMouseHandler::examineMouse()
{
     int buttonData = myMouse->GetMiscData(),
            currentX, currentY;
     WTmouse_rawdata *mouseData;
     HHMouseData *screamerData = NULL;

     myMouse->RawUpdate();
     mouseData = (WTmouse_rawdata*)myMouse->GetRawData();
     currentX = (int)mouseData->pos[0];
     currentY = (int)mouseData->pos[1];

     /**** Check for buttons just pressed ****/
     if (buttonData & WTMOUSE_LEFTBUTTON)
     {
            lastX = currentX;
            lastY = currentY;
            leftClicked = TRUE;
     }
     if (buttonData & WTMOUSE_RIGHTBUTTON)
     {
            lastX = currentX;
            lastY = currentY;
            rightClicked = TRUE;
     }
     if (buttonData & WTMOUSE_MIDDLEBUTTON)
     {
            lastX = currentX;
            lastY = currentY;
            middleClicked = TRUE;
     }
     /***************************************/

     /**** Check for buttons released ********/
```

```
    if (buttonData & WTMOUSE_LEFTUP)
    {
          leftClicked = FALSE;
    }
    if (buttonData & WTMOUSE_MIDDLEUP)
    {
          middleClicked = FALSE;
    }
    if (buttonData & WTMOUSE_RIGHTUP)
    {
          rightClicked = FALSE;
    }
    /*************************************/

    /*** Check for other button-functions ***/
    if ((buttonData & WTMOUSE_LEFTDOWN)||(buttonData & WTMOUSE_MIDDLEDOWN)||(buttonData &
WTMOUSE_RIGHTDOWN))
    {
          screamerData = new HHMouseData;
          screamerData->deltaX = currentX - lastX;
          screamerData->deltaY = currentY - lastY;
          screamerData->leftButton = leftClicked;
          screamerData->middleButton = middleClicked;
          screamerData->rightButton = rightClicked;
          screamerData->doubleClick = NONE;

          lastX = currentX;
          lastY = currentY;
    }
    else if ((buttonData & WTMOUSE_LEFTDBLCLK)||(buttonData &
WTMOUSE_MIDDLEDBLCLK)||(buttonData & WTMOUSE_RIGHTDBLCLK))
    {
          screamerData = new HHMouseData;
          screamerData->deltaX = currentX;
          screamerData->deltaY = currentY;
          screamerData->leftButton =
               screamerData->middleButton =
               screamerData->rightButton = FALSE;
          if (buttonData & WTMOUSE_LEFTDBLCLK) screamerData->doubleClick = LEFT;
               else if (buttonData & WTMOUSE_RIGHTDBLCLK) screamerData->doubleClick = RIGHT;
               else if (buttonData & WTMOUSE_MIDDLEDBLCLK) screamerData->doubleClick =
MIDDLE;
    }
    if ( (NULL != screamerData) && ((NONE != screamerData->doubleClick)||(0 != screamerData-
>deltaX)||(0 != screamerData->deltaY)) )
    {
          myListener->listen(screamerData);
    }
    /*************************************/

    delete screamerData;
}
```

C.5 The geometry interface

```
////////////////////////////////////////////////////////////////////////////////
// HHGeometry.h - Geometry interface                                          //
////////////////////////////////////////////////////////////////////////////////

#ifndef __HH_GEOMETRY_H_
#define __HH_GEOMETRY_H_

#include "wt.h"
#include "wtcpp.h"

/**  The interface that specifies how all geometry classes should be structured. Contains
     all methods that are of interest to a geometry class. This HHGeometry is modified to
     suit the HHBamse geometry class!
  */
class HHGeometry
{
    public:
          virtual FLAG addAsChild(WtGroup *parent) = 0;
```

```
            virtual void rotate(int axis, float radians = PI/180) = 0;
            virtual void rotate(const float &transX, const float &transY, int part) = 0;
            virtual void chkNode(WtNodePath *nodePath) = 0;
            virtual WtP3 getMidpoint() = 0;
            virtual void nullAll(BOOL justOrient) = 0;
            virtual BOOL shoot(int whichOne) = 0;

    private:
            virtual void loadGeometry() = 0;
            virtual void destroyGeometry() = 0;
};


#endif
```

C.6 The HHRotation classes

```
////////////////////////////////////////////////////////////////////////////////
// HHRotation.h – The test geometry #1                                        //
////////////////////////////////////////////////////////////////////////////////

#ifndef __HH_ROTATION_H_
#define __HH_ROTATION_H_


#include <math.h>
#include "HHGeometry.h"


/**  The first of the HHRotation classes, which loads a simple geometry from two different
      different files. This class is used to test the functionality of WTK.
  */
class HHRotation : public HHGeometry
{
    public:
            HHRotation();
            ~HHRotation();

            FLAG addAsChild(WtGroup *parent);
            void rotate(int axis, float radians = PI/180);
            void rotate(const float &transX, const float &transY, int part);
            void chkNode(WtNodePath *nodePath);
            WtP3 getMidpoint();

            static void HHRotationTask(void *myRotation);

    private:
            void loadGeometry();
            void destroyGeometry();
            void turretRotation(float angle);

            /*** Geometry variables ***/
            WtNode *whole, *turret;               // The geometries
            WtP3 extents;                         // The extents of the turret geometry (the
box)

            /*** Task variables *******/
            FLAG taskRunning;                     // Is a task active right now?
            int currentAngle;                     // The actual angle (if a task is running)
            WtTask *rotationTask;                 // The task
};


#endif


////////////////////////////////////////////////////////////////////////////////
// HHRotation.cpp - The test geometry #1                                      //
////////////////////////////////////////////////////////////////////////////////

#include "HHRotation.h"
```

```
/*****************************************************************************/
/*** PUBLIC METHODS...                                                     ***/
/*****************************************************************************/

/**  The default constructor. Creates a new HHRotation instance and loads it's geometry.
  */
HHRotation::HHRotation()
{
    loadGeometry();
    taskRunning = FALSE;
}

/**  The destructor. Deallocates the memory that the instance has used.
  */
HHRotation::~HHRotation()
{
    destroyGeometry();
}

/**  Adds the HHRotation geometry as last child to the specified group-node parent.
     The return value is the result of the operation (TRUE / FALSE),
  */
FLAG HHRotation::addAsChild(WtGroup *parent)
{
    return parent->AddChild(whole);
}

/**  Rotates the specified geometry (part) around the specified axis (x, Y, Z). The angle
     radians should of course be in radians.
  */
void HHRotation::rotate(int axis, float radians)
{
}

/**  Rotates the specified geometry (part) according to the transformation values along
     the X and Y axes that have been read by the mouse.
  */
void HHRotation::rotate(const float &transX, const float &transY, int part)
{
    switch (part)
    {
          case 0:    ((WtMovable*)whole)->MovAxisRotation(X, transX*PI/180);
                     ((WtMovable*)whole)->MovAxisRotation(Y, transY*PI/180);
                     break;

          case 1:    turretRotation(transY*PI/180);
                     break;
    }
}

/**  Controls the given node path if any of the movable parts of the geometry is along it.
     If so, a task is activated (HHRotationTask), which rotates this geometry.
  */
void HHRotation::chkNode(WtNodePath *nodePath)
{
    int currentNodeNum;
    WtNode *currentNode;

    currentNodeNum = nodePath->NumNodes() - 1;

    while (0 <= currentNodeNum)
    {
          currentNode = nodePath->GetNode(currentNodeNum);
          if (turret == currentNode)
          {
               if (!taskRunning)
               {
                    currentAngle = 0;
                    taskRunning = TRUE;
                    rotationTask = new WtTask(this, HHRotation::HHRotationTask, 1.f);
               }
               break;
          }
          currentNodeNum--;
    }
}
```

```
/**  Returns the midpoint of the whole geometry.
  */
WtP3 HHRotation::getMidpoint()
{
    WtP3 midpoint;

    ((WtGroup*)whole)->GetMidpoint(midpoint);

    return midpoint;
}

/*****************************************************************************************/
/*** STATIC METHODS...                                                                 ***/
/*****************************************************************************************/

/**  The rotations task function for the turret geometry. This function rotates the
     specified HHRotation instance's turret geometry 360 degrees (one degree each turn),
     and then terminates the task.
  */
void HHRotation::HHRotationTask(void *myRotation)
{
    ((HHRotation*)myRotation)->rotate(1, 1, 1);
    ((HHRotation*)myRotation)->currentAngle += 1;
    if (((HHRotation*)myRotation)->currentAngle >= 360)
    {
          ((HHRotation*)myRotation)->taskRunning = FALSE;
          delete ((HHRotation*)myRotation)->rotationTask;
    }
}

/*****************************************************************************************/
/*** PRIVATE METHODS...                                                                ***/
/*****************************************************************************************/

/**  Loads all geometry used in the instance and makes sure that it's location and
     rotation is correct (a VRML 1.0 geometry needs to be rotated 180 degrees, due to
     different coordinate systems).
  */
void HHRotation::loadGeometry()
{
    float trans[3] = {-2, -2, 0};
    WtP3 turretTrans(trans);
    whole = MovNodeLoad("rotate1.wrl", 1.f);
    whole->SetName("Whole");
    turret = MovNodeLoad("rotate2.wrl", 1.f);
    turret->SetName("Turret");

    ((WtMovable*)turret)->SetTranslation(turretTrans);
    ((WtGroup*)turret)->GetExtents(extents); // Får en FLAGGA som jag inte fångar
    ((WtGroup*)whole)->AddChild(turret);
}

/**  Destroys the geometry that previously have been loaded.
     PRECON: Geometry must have been loaded with loadGeometry().
  */
void HHRotation::destroyGeometry()
{
    delete whole;
    delete turret;
}

/**  Rotates the HHRotation instance's turret geometry the given angle (in radians).
  */
void HHRotation::turretRotation(float angle)
{
    WtP3 translation;

    ((WtMovable*)turret)->MovAxisRotation(Z, angle);
}


////////////////////////////////////////////////////////////////////////////////////////
// HHRotation2.h - The test geometry #2                                               //
////////////////////////////////////////////////////////////////////////////////////////
```

```
#ifndef __HH_ROTATION_2_H_
#define __HH_ROTATION_2_H_


#include <math.h>
#include "HHGeometry.h"


/**  HHRotation2 is basically the same as HHRotation, but it has a significant
     difference; all geometries are loaded from ONE file and then separated into different
     parts.
  */
class HHRotation2 : public HHGeometry
{
    public:
           HHRotation2();
           ~HHRotation2();

           FLAG addAsChild(WtGroup *parent);
           void rotate(int axis, float radians = PI/180);
           void rotate(const float &transX, const float &transY, int part);
           void chkNode(WtNodePath *nodePath);
           WtP3 getMidpoint();

           static void HHRotationTask(void *myRotation);

    private:
           void loadGeometry();
           void destroyGeometry();
           void turretRotation(float angle);
           void separateTurret(WtNode *currentNode);

           /*** Geometry variables ***/
           WtNode *whole, *turret;                // The geometries
           WtP3 extents;                          // Extents of the turret geometry

           FLAG rockOn;                           // Is used to recursively search for a sub-
                                                  // geometry

           /*** Task variables *******/
           FLAG taskRunning;                      // Is a task running right now?
           int currentAngle;                      // The current angle (if a task is running)
           WtTask *rotationTask;                  // The actual task

           FLAG taskElevate;                      // Shall the turret be elevated or de-
elevated?
};


#endif


////////////////////////////////////////////////////////////////////////////////
// HHRotation2.cpp - The test geometry #2                                     //
////////////////////////////////////////////////////////////////////////////////

#include "HHRotation2.h"


/*****************************************************************************/
/*** PUBLIC METHODS...                                                     ***/
/*****************************************************************************/

/**  The default constructor. Creates a new HHRotation2 instance and loads it's geometries.
  */
HHRotation2::HHRotation2()
{
    loadGeometry();
    taskRunning = FALSE;
    taskElevate = FALSE;
}

/**  The destructor. Deallocates the memory used by the instance.
  */
HHRotation2::~HHRotation2()
{
    destroyGeometry();
```

```
}

/**  Adds the HHRotation2 geometry as last child to the specified griup-node parent.
      The return value is the result of the operation (TRUE / FALSE).
  */
FLAG HHRotation2::addAsChild(WtGroup *parent)
{
    return parent->AddChild(whole);
}

/**  Rotates the specified part of the geometry around the specified axis (X, Y, Z). The
      angle radians should of course be given in radians!
  */
void HHRotation2::rotate(int axis, float radians)
{
}

/**  Rotates the specified part of the geometry according to the transformation values
      read from the mouse.
  */
void HHRotation2::rotate(const float &transX, const float &transY, int part)
{
    switch (part)
    {
          case 0:    ((WtMovable*)whole)->MovAxisRotation(X, transX*PI/180);
                     ((WtMovable*)whole)->MovAxisRotation(Y, transY*PI/180);
                     break;

          case 1:    turretRotation(transY*PI/180);
                     break;
    }
}

/**  Controls the specified node path if any of the movable parts of the geometry is
      along it. If so, a task (HHRotationTask) is activated, which makes sure that this
      sub-geometry is rotated in a specific pattern (different for different parts).
  */
void HHRotation2::chkNode(WtNodePath *nodePath)
{
    int currentNodeNum;
    WtNode *currentNode;

    currentNodeNum = nodePath->NumNodes() - 1;

    char *name;
    while (0 <= currentNodeNum)
    {
          currentNode = nodePath->GetNode(currentNodeNum);

          if (NULL != currentNode)
          {
               name = currentNode->GetName();
               if ((turret == currentNode)||((NULL != name)&&(0 == strcmp(name, turret-
>GetName()))))
               {
                    if (!taskRunning)
                    {
                         if (taskElevate) // Thus the turret is elevated; time to de-
elevate
                         {
                              currentAngle = 60;
                              taskElevate = FALSE;
                         }
                         else                  // Time to elevate
                         {
                              currentAngle = 0;
                              taskElevate = TRUE;
                         }
                         taskRunning = TRUE;
                         rotationTask = new WtTask(this, HHRotation2::HHRotationTask, 1.f);
                    }
                    break;
               }
          }
          currentNodeNum--;
    }
}
```

```
/**  Returns the midpoint of the geometry.
  */
WtP3 HHRotation2::getMidpoint()
{
    WtP3 midpoint;

    ((WtGroup*)whole)->GetMidpoint(midpoint);

    return midpoint;
}


/*****************************************************************************/
/*** STATIC METHODS...                                                     ***/
/*****************************************************************************/

/**  the rotation task function that rotates HHRotation2's turret geometry. If the turret
     geometry is already elevated, it de-elevates it (0 degrees), otherwise it elevates the
turret
     geometry (60 degrees).
  */
void HHRotation2::HHRotationTask(void *myRotation)
{
    if (((HHRotation2*)myRotation)->taskElevate)
    {
          ((HHRotation2*)myRotation)->rotate(1, -1, 1);
          ((HHRotation2*)myRotation)->currentAngle += 1;
          if (((HHRotation2*)myRotation)->currentAngle >= 60)
          {
               ((HHRotation2*)myRotation)->taskRunning = FALSE;
               delete ((HHRotation2*)myRotation)->rotationTask;
          }
    }
    else
    {
          ((HHRotation2*)myRotation)->rotate(1, 1, 1);
          ((HHRotation2*)myRotation)->currentAngle -= 1;
          if (((HHRotation2*)myRotation)->currentAngle <= 0)
          {
               ((HHRotation2*)myRotation)->taskRunning = FALSE;
               delete ((HHRotation2*)myRotation)->rotationTask;
          }
    }
}

/*****************************************************************************/
/*** PRIVATE METHODS...                                                    ***/
/*****************************************************************************/

/**  Loads all the geometries used in the instance. Also makes sure that the location and
     rotation of the geometries are correct (a VRML 1.0 geometry must be rotated 180
     degrees).
  */
void HHRotation2::loadGeometry()
{
    float trans[3] = {-2, -2, 0};
    WtP3 turretTrans(trans);
    whole = MovNodeLoad("rotateT.wrl", 1.f);      // "rotate.wrl" for an untextured geometry
    whole->SetName("Whole");

    rockOn = TRUE;
    separateTurret(whole);
    ((WtGroup*)turret)->GetExtents(extents);
    ((WtMovable*)whole)->MovAxisRotation(Z, PI);
}

/**  Destroys the geometry that previously have been loaded.
     PRECON: Geometry must have been loaded with loadGeometry().
  */
void HHRotation2::destroyGeometry()
{
    delete whole;
    delete turret;
}

/**  Rotates the HHRotation2 instance's turret geometry the given angle (in radians),
```

```
     around a point that is located in the geometry's left end.
  */
void HHRotation2::turretRotation(float angle)
{
    WtP3 translation;

    translation[0] = extents[0];
    translation[1] = -extents[1];
    translation[2] = 0;
    ((WtMovable*)turret)->Translate(translation, WTFRAME_LOCAL);

    ((WtMovable*)turret)->MovAxisRotation(Z, angle);

    translation[0] *= -1;
    translation[1] *= -1;
    ((WtMovable*)turret)->Translate(translation, WTFRAME_LOCAL);
}

/**  Separates the turret sub-geometry from the other, makes it to a movable
     node and finally puts it back at the same place in the scene graph.
  */
void HHRotation2::separateTurret(WtNode *currentNode)
{
    if (rockOn)
    {
          char *name = currentNode->GetName();
          if ((NULL != name)&&(0 == strcmp(name, "Turret")))
          {
               WtNode *parent = currentNode->GetParent(0);

               if (NULL == parent)
                    WTmessage("Unable to trace turret...\n");
               else
               {
                    int childNum = ((WtGroup*)parent)->NumChildren();
                    for (int i=0; i<childNum; i++)
                         if (((WtGroup*)parent)->GetChild(i) == currentNode) break;

                    WtGroup *newTurret = new WtMovSep(NULL);
                    newTurret->AddChild(currentNode);
                    ((WtGroup*)parent)->InsertChild(newTurret, i);
                    ((WtGroup*)parent)->DeleteChild(i+1);

                    currentNode->SetName("TurretGeo");
                    newTurret->SetName("Turret");
                    turret = newTurret;
               }

               rockOn = FALSE;
          }
          else
          {
               int children = ((WtGroup*)currentNode)->NumChildren();
               if (0 != children)
               {
                    for (int i=0; i<children; i++)
                         separateTurret(((WtGroup*)currentNode)->GetChild(i));
               }
          }
    }
}


////////////////////////////////////////////////////////////////////////////////
// HHRotation97.h - The test geometry adapted for VRML 2.0 / VRML 97          //
////////////////////////////////////////////////////////////////////////////////


#ifndef __HH_ROTATION_97_H_
#define __HH_ROTATION_97_H_


#include <math.h>
#include "HHGeometry.h"


/**  HHRotation97 is basically th same as HHRotation2, but it uses VRML 2.0-geometry instead
```

```
     of VRML 1.0. Another difference is that it does not work, since WTK is unable to handle
     VRML 2.0!
  */
class HHRotation97 : public HHGeometry
{
    public:
           HHRotation97();
           ~HHRotation97();

           FLAG addAsChild(WtGroup *parent);
           void rotate(int axis, float radians = PI/180);
           void rotate(const float &transX, const float &transY, int part);
           void chkNode(WtNodePath *nodePath);
           WtP3 getMidpoint();

           static void HHRotationTask(void *myRotation);

    private:
           void loadGeometry();
           void destroyGeometry();
           void turretRotation(float angle);
           void locateTurret(WtNode *currentNode);

           /*** Geometry variables ***/
           WtNode *whole, *turret;               // The geometries
           WtP3 extents;                         // The extent of the turret geometry

           FLAG rockOn;                          // Is used to recursively search after a
                                                       // geometry

           /*** Task variables *******/
           FLAG taskRunning;                     // Is a task currently active?
           int currentAngle;                     // The current angle (is a task is running)
           WtTask *rotationTask;                 // The actual task

           FLAG taskElevate;                     // Should the turret be elevated, or de-
elevated?
};


#endif


////////////////////////////////////////////////////////////////////////////////
// HHRotation97.cpp - The test geometry adapted for VRML 2.0 / VRML 97                //
////////////////////////////////////////////////////////////////////////////////

#include "HHRotation97.h"


/*****************************************************************************/
/*** PUBLIC METHODS...                                                     ***/
/*****************************************************************************/

/**  The default constructor. Creates a new HHRotation97 instance and loads it's geometry.
  */
HHRotation97::HHRotation97()
{
    loadGeometry();
    taskRunning = FALSE;
    taskElevate = FALSE;
}

/**  The destructor. De-allocates the memory used by the instance.
  */
HHRotation97::~HHRotation97()
{
    destroyGeometry();
}

/**  Adds the HHRotation97 geometry as the last child of the specified group-node parent.
     The return value is the result of the operation (TRUE / FALSE).
  */
FLAG HHRotation97::addAsChild(WtGroup *parent)
{
    return parent->AddChild(whole);
}
```

```
/**  Rotates the specified part of the geometry around the given axis (X, Y, Z). The angle,
      radians, should of course be in radians! NOT IMPLEMENTED!!!
  */
void HHRotation97::rotate(int axis, float radians)
{
}

/**  Rotates the specified part of the geometry according to the transformation values that
      have been read from the mouse.
  */
void HHRotation97::rotate(const float &transX, const float &transY, int part)
{
     switch (part)
     {
            case 0:    ((WtMovable*)whole)->MovAxisRotation(X, transX*PI/180);
                       ((WtMovable*)whole)->MovAxisRotation(Y, transY*PI/180);
                       break;

            case 1:    turretRotation(transY*PI/180);
                       break;
     }
}

/**  Controls the incoming node path if any of the intsance's movable parts lies in it.
      If so, is activates a task (HHRotationTask), which rotates the geometry accordingly.
      THIS METHOD DOES NOT WORK BECAUSE WTK DO NOT SUPPORT VRML 2.0 PROPERLY!!!
  */
void HHRotation97::chkNode(WtNodePath *nodePath)
{
     int currentNodeNum;
     WtNode *currentNode;

     currentNodeNum = nodePath->NumNodes() - 1;

     char *name;
     while (0 <= currentNodeNum)
     {
            currentNode = nodePath->GetNode(currentNodeNum);

            if (NULL != currentNode)
            {
                  name = currentNode->GetName();

                  if ((turret == currentNode)||((NULL != name)&&(0 == strcmp(name, "Turret"))))
                  {
                        if (!taskRunning)
                        {
                              if (taskElevate)
                              {
                                    currentAngle = 60;
                                    taskElevate = FALSE;
                              }
                              else
                              {
                                    currentAngle = 0;
                                    taskElevate = TRUE;
                              }
                              taskRunning = TRUE;
                              rotationTask = new WtTask(this, HHRotation97::HHRotationTask,
1.f);
                        }
                        break;
                  }
            }
            currentNodeNum--;
     }
}

/**  Returns the midpoint of the geometry.
  */
WtP3 HHRotation97::getMidpoint()
{
     WtP3 midpoint;

     ((WtGroup*)whole)->GetMidpoint(midpoint);
```

```
    return midpoint;
}


/*****************************************************************************/
/*** STATIC METHODS...                                                     ***/
/*****************************************************************************/

/**  The rotation task function for HHRotation97's turret geometry. Elevates (60 degrees
     angle) or de-elevates (0 degrees angle) the turret according to it's initial position.
  */
void HHRotation97::HHRotationTask(void *myRotation)
{
    if (((HHRotation97*)myRotation)->taskElevate)
    {
          ((HHRotation97*)myRotation)->rotate(1, -1, 1);
          ((HHRotation97*)myRotation)->currentAngle += 1;
          if (((HHRotation97*)myRotation)->currentAngle >= 60)
          {
               ((HHRotation97*)myRotation)->taskRunning = FALSE;
               delete ((HHRotation97*)myRotation)->rotationTask;
          }
    }
    else
    {
          ((HHRotation97*)myRotation)->rotate(1, 1, 1);
          ((HHRotation97*)myRotation)->currentAngle -= 1;
          if (((HHRotation97*)myRotation)->currentAngle <= 0)
          {
               ((HHRotation97*)myRotation)->taskRunning = FALSE;
               delete ((HHRotation97*)myRotation)->rotationTask;
          }
    }
}

/*****************************************************************************/
/*** PRIVATE METHODS...                                                    ***/
/*****************************************************************************/

/**  Loads all geometry that is used in the instance. One difference between using VRML 2
     geometry and VRML 1, is that there is no need to rotate the geometry after loading a
     VRML 2 geometry.
  */
void HHRotation97::loadGeometry()
{
    float trans[3] = {-2, -2, 0};
    WtP3 turretTrans(trans);
    whole = MovNodeLoad("bamse666.wrl", 1.f); //"assy*.wrl" "rotate2_0T.wrl"
    whole->SetName("Whole");

    rockOn = FALSE;
    locateTurret(whole);

    if (NULL == turret) WTmessage("Unable to trace turret...\n");
}

/**  Destroys the geometry that previously has been loaded.
     PRECON: The geometry has been loaded using loadGeometry().
  */
void HHRotation97::destroyGeometry()
{
    delete whole;
    delete turret;
}

/**  Rotates the HHRotation97 instance's turret geometry the given angle (in radians)
     around a point that is located on the geometry's left end.
  */
void HHRotation97::turretRotation(float angle)
{
    WtP3 translation;

    translation[0] = extents[0];
    translation[1] = -extents[1];
    translation[2] = 0;
    ((WtMovable*)turret)->Translate(translation, WTFRAME_LOCAL);
```

```
    ((WtMovable*)turret)->MovAxisRotation(Z, angle);

    translation[0] *= -1;
    translation[1] *= -1;
    ((WtMovable*)turret)->Translate(translation, WTFRAME_LOCAL);
}

/**  Separates the turret part of the geometry from the others.
     DOES NOT WORK WITH VRML 2.0!!!!
  */
void HHRotation97::locateTurret(WtNode *currentNode)
{
    if (rockOn)
    {
           char *name = currentNode->GetName();

           if ((NULL != name)&&(0 == strcmp(name, "Turret")))
           {
                 turret = currentNode;
                 rockOn = FALSE;
           }
           else
           {
                 int children = ((WtGroup*)currentNode)->NumChildren();
                 if (0 != children)
                 {
                       for (int i=0; i<children; i++)
                             locateTurret(((WtGroup*)currentNode)->GetChild(i));
                 }
                 children = ((WtMovable*)currentNode)->NumAttachments();
                 if (0 != children)
                 {
                       for (int i=0; i<children; i++)
                             locateTurret(((WtMovable*)currentNode)->GetAttachment(i));
                 }
           }
    }
}
```

C.7 The Bamse geometry class

```
////////////////////////////////////////////////////////////////////////////////
// HHBamse.h - The Bamse geometry class...                                    //
////////////////////////////////////////////////////////////////////////////////



#ifndef __HH_BAMSE_H_
#define __HH_BAMSE_H_


#include <math.h>
#include "HHGeometry.h"
#include "HHVirtualManual.h"

enum HHParts {LTURRET, RTURRET, POLE, HEAD, TOP, RADAR, SCREEN};

/**  A geometry class that contains the Bamse geometry, which is the geometry actually
      used in the testbed. Is based on HHRotation2 and thus provides the same functionality,
      but slightly enhanced, as it.
  */
class HHBamse : public HHGeometry
{
    public:
           HHBamse();
           ~HHBamse();

           FLAG addAsChild(WtGroup *parent);
           void rotate(int axis, float radians = PI/180);
           void rotate(const float &transX, const float &transY, int part);
           void chkNode(WtNodePath *nodePath);
           WtP3 getMidpoint();
```

```
            void nullAll(BOOL justOrient);

            BOOL shoot(int whichOne);
            BOOL turretElevation(BOOL left);
            BOOL isLeft();

            static void HHLeftTask(void *myRotation);
            static void HHRightTask(void *myRotation);
            static void HHPoleTask(void *myRotation);
            static void HHRadarTask(void *myRotation);
            static void HHScreenTask(void *myRotation);
            static void HHMissileTask(void *myRotation);

    private:
            void loadGeometry();
            void destroyGeometry();
            void turretRotation(float angle, int part);
            void poleRotation(float angle);
            void headRotation(float angle);
            void topRotation(float angle);
            void radarRotation(float angle);
            void screenRotation(float angle);
            void computeTurrets();
            void computePole();
            void separateTop();
            void separateNode(WtNode *currentNode, char* partName, HHParts whichPart);


            void loadMissile();

            /*** Geometry variables ***/
            WtNode *whole;
            WtGroup *left, *right, *top;
            WtP3 leftExt, rightExt;                // The extent of the turret geometries

            WtGroup *pole, *head, *radar, *screen;
            WtP3 poleExt, headExt, screenExt;
            WtM4 poleTrans, headTrans, wholeTrans, topTrans, leftTrans, rightTrans,
                   radarTrans, screenTrans;
            float radarAngle, screenAngle;

            /*** Seek variables *******/
            FLAG rockOn;                           // Is used to seek recursively for a sub-
                                                         // geometry
            WtNode *theParent;                     // The parent of the sought node
            int theLocation;                 // The location at the parent that the sought
                                                         // node was located


            /*** Task variables *******/
            BOOL leftRun, rightRun;                // Is tasks running?
            int leftAngle, rightAngle;        // The actual angles (if tasks is running)
            WtTask *leftTask, *rightTask;     // The tasks
            BOOL leftElevate, rightElevate;   // Should we elevate or de-elevate?

            BOOL poleRun, poleElevate, elevateOther;
            int poleAngle, headAngle;
            WtTask *poleTask, *radarTask, *screenTask, *missileTask;


            /*** Missile related ******/
            WtNode *missile, *motor;
            WtP3 mTranslation, moTranslation;
            WtM4 initMissileTrans, initMotorTrans, missileTrans, motorTrans;
            int missileCount;
            float motorAngle;
            BOOL missileRun, missileSeparated;
};


#endif


////////////////////////////////////////////////////////////////////////////////
// HHBamse.cpp - The Bamse geometry class...                                  //
////////////////////////////////////////////////////////////////////////////////

#include "HHBamse.h"
```

```
/*****************************************************************************/
/*** PUBLIC METHODS...                                                     ***/
/*****************************************************************************/

/**  The default constructor. Creates a new HHBamse instance and loads it's geometry.
  */
HHBamse::HHBamse()
{
    loadGeometry();

    // Initiates task variables
    leftRun = FALSE;
    rightRun = FALSE;
    poleRun = FALSE;
    leftElevate = FALSE;
    rightElevate = FALSE;
    poleElevate = TRUE;
    leftTask = NULL;
    rightTask = NULL;
    poleTask = NULL;
    radarTask = NULL;
    screenTask = NULL;

    radarAngle =
    screenAngle = 0.f;

    missileTask = NULL;
    missileRun = FALSE;

    elevateOther = FALSE;
}

/**  The destructor. De-allocates the memory used by the instance.
  */
HHBamse::~HHBamse()
{
    destroyGeometry();
}

/**  Adds the HHBamse geometry as last child of the specified group-node parent.
     The return value is the result of the operation (TRUE / FALSE).
  */
FLAG HHBamse::addAsChild(WtGroup *parent)
{
    return parent->AddChild(whole);
}

/**  Rotates the specified part of the geometry around the given axis (X, Y, Z).
     NOT USED, THUS NOT IMPLEMENTED!!!
  */
void HHBamse::rotate(int axis, float radians)
{
}

/**  Rotates the specified part of the geometry according to the transformation values
     read from the mouse.
  */
void HHBamse::rotate(const float &transX, const float &transY, int part)
{
    switch (part)
    {
          case 0:    if (missileRun)
                     {
                          ((WtMovable*)missile)->SetTransform(missileTrans);
                          ((WtMovable*)motor)->SetTransform(motorTrans);
                     }
                     ((WtMovable*)whole)->MovAxisRotation(Y, transY*PI/180);
                     break;

          case 1:
          case 2:    turretRotation(transY*PI/180, part);
                     break;

          case 3: poleRotation(transY*PI/180);
                     break;
```

```
            case 4: headRotation(transY*PI/180);
                        break;

            case 5:     topRotation(transX*PI/180);
                        break;

            case 6:     if (poleElevate) radarRotation(transX);
                        break;

            case 7:     if (poleElevate) screenRotation(transX);
                        break;
    }
}

/**  Controls if if the specified node path contains any of the instance's movable
     geometries. If so, a task is activated that rotates the geometry accordingly!
  */
void HHBamse::chkNode(WtNodePath *nodePath)
{
    extern HHVirtualManual *application;

    int currentNodeNum;
    WtNode *currentNode;

    currentNodeNum = nodePath->NumNodes() - 1;

    char *name;
    while (0 <= currentNodeNum)
    {
           currentNode = nodePath->GetNode(currentNodeNum);

           if (NULL != currentNode)
           {
                 name = currentNode->GetName();

                 if ((left == currentNode)||((NULL != name)&&(0 == strcmp(name, left-
>GetName()))))
                 {
                       application->setManState(BAMSE_TURRET);
                       turretElevation(TRUE);
                       break;
                 }
                 else if ((right == currentNode)||((NULL != name)&&(0 == strcmp(name, right-
>GetName()))))
                 {
                       application->setManState(BAMSE_TURRET);
                       turretElevation(FALSE);
                       break;
                 }
                 else if ((pole == currentNode)||((NULL != name)&&(0 == strcmp(name, pole-
>GetName()))))
                 {
                       application->setManState(BAMSE_POLE);

                       if (!poleRun)
                       {
                             if (poleElevate)
                             {
                                   headAngle = 180;
                                   poleAngle = 90;
                                   poleElevate = FALSE;
                             }
                             else
                             {
                                   headAngle =
                                   poleAngle = 0;
                                   poleElevate = TRUE;
                             }
                             if (0 != radarAngle)
                             {
                                   radarTask = new WtTask(this, HHBamse::HHRadarTask, 1.f);
                             }
                             if (0 != screenAngle)
                             {
                                   screenTask = new WtTask(this, HHBamse::HHScreenTask, 1.f);
                             }
```

```
                        poleRun = TRUE;
                        poleTask = new WtTask(this, HHBamse::HHPoleTask, 1.f);
                    }
                    break;
                }
                else if ((whole == currentNode)||((NULL != name)&&(0 == strcmp(name, whole-
>GetName()))))
                    application->setManState(BAMSE_WHOLE);
            }
            currentNodeNum--;
    }
}

/**  Returns the midpoint of the whole geometry.
  */
WtP3 HHBamse::getMidpoint()
{
    WtP3 midpoint;

    ((WtGroup*)whole)->GetMidpoint(midpoint);

    return midpoint;
}

void HHBamse::nullAll(BOOL justOrient)
{
    ((WtMovable*)whole)->SetTransform(wholeTrans);
    ((WtMovable*)top)->SetTransform(topTrans);
    ((WtMovable*)radar)->SetTransform(radarTrans);
    ((WtMovable*)screen)->SetTransform(screenTrans);
    radarAngle = 0.f;
    screenAngle = 0.f;

    if (!justOrient)
    {
        if ((!leftRun)&&(!leftElevate))
        {
            turretElevation(TRUE);
        }

        if ((!poleRun)&&(!poleElevate))
        {
            headAngle =
            poleAngle = 0;
            poleElevate = TRUE;
            poleRun = TRUE;
            poleTask = new WtTask(this, HHBamse::HHPoleTask, 1.f);
        }
    }
}

BOOL HHBamse::turretElevation(BOOL left)
{
    if (!missileRun)
    {
        if ((!leftRun)&&(!rightRun))
        {
            if (left)
            {
                if (leftElevate)
                {
                    leftAngle = 60;
                    leftElevate = FALSE;
                    leftRun = TRUE;
                    leftTask = new WtTask(this, HHBamse::HHLeftTask, 1.f);
                }
                else
                {
                    if (rightElevate)
                    {
                        elevateOther = TRUE;
                        turretElevation(FALSE);
                    }
                    else
                    {
                        leftAngle = 0;
                        leftElevate = TRUE;
```

```
                              leftRun = TRUE;
                              leftTask = new WtTask(this, HHBamse::HHLeftTask, 1.f);
                        }
                  }
                  return TRUE;
            }
            else
            {
                  if (rightElevate)
                  {
                        rightAngle = 60;
                        rightElevate = FALSE;
                        rightRun = TRUE;
                        rightTask = new WtTask(this, HHBamse::HHRightTask, 1.f);
                  }
                  else
                  {
                        if (leftElevate)
                        {
                              elevateOther = TRUE;
                              turretElevation(TRUE);
                        }
                        else
                        {
                              rightAngle = 0;
                              rightElevate = TRUE;
                              rightRun = TRUE;
                              rightTask = new WtTask(this, HHBamse::HHRightTask, 1.f);
                        }
                  }
                  return TRUE;
            }
      }
    }

    return FALSE;
}

/**  Is the left turret elevated? Return TRUE or FALSE.
  */
BOOL HHBamse::isLeft()
{
    return leftElevate;
}

/*****************************************************************************/
/*** STATIC METHODS...                                                     ***/
/*****************************************************************************/

/**  The rotation task function for the HHBamse instance's left turret geometry.
  */
void HHBamse::HHLeftTask(void *myRotation)
{

    if (((HHBamse*)myRotation)->leftElevate)
    {
          ((HHBamse*)myRotation)->rotate(1, -1, 1);
          ((HHBamse*)myRotation)->leftAngle += 1;
          if (((HHBamse*)myRotation)->leftAngle >= 60)
          {
                ((HHBamse*)myRotation)->leftRun = FALSE;
                delete ((HHBamse*)myRotation)->leftTask;
                ((HHBamse*)myRotation)->leftTask = NULL;
          }
    }
    else
    {
          ((HHBamse*)myRotation)->rotate(1, 1, 1);
          ((HHBamse*)myRotation)->leftAngle -= 1;
          if (((HHBamse*)myRotation)->leftAngle <= 0)
          {
                ((HHBamse*)myRotation)->leftRun = FALSE;

                if (((HHBamse*)myRotation)->elevateOther)
                {
                      ((HHBamse*)myRotation)->elevateOther = FALSE;
                      ((HHBamse*)myRotation)->turretElevation(FALSE);
```

```
                }

                delete ((HHBamse*)myRotation)->leftTask;
                ((HHBamse*)myRotation)->leftTask = NULL;
          }
     }
}

/**  The rotation task function for the HHBamse instance's right turret geometry.
  */
void HHBamse::HHRightTask(void *myRotation)
{
     if (((HHBamse*)myRotation)->rightElevate)
     {
          ((HHBamse*)myRotation)->rotate(1, -1, 2);
          ((HHBamse*)myRotation)->rightAngle += 1;
          if (((HHBamse*)myRotation)->rightAngle >= 60)
          {
                ((HHBamse*)myRotation)->rightRun = FALSE;
                delete ((HHBamse*)myRotation)->rightTask;
                ((HHBamse*)myRotation)->rightTask = NULL;
          }
     }
     else
     {
          ((HHBamse*)myRotation)->rotate(1, 1, 2);
          ((HHBamse*)myRotation)->rightAngle -= 1;
          if (((HHBamse*)myRotation)->rightAngle <= 0)
          {
                ((HHBamse*)myRotation)->rightRun = FALSE;

                if (((HHBamse*)myRotation)->elevateOther)
                {
                     ((HHBamse*)myRotation)->elevateOther = FALSE;
                     ((HHBamse*)myRotation)->turretElevation(TRUE);
                }

                delete ((HHBamse*)myRotation)->rightTask;
                ((HHBamse*)myRotation)->rightTask = NULL;
          }
     }
}

/**  The rotation task function for the HHBamse instance's radar pylon geometry.
  */
void HHBamse::HHPoleTask(void *myRotation)
{
     if (((HHBamse*)myRotation)->poleElevate)
     {
          ((HHBamse*)myRotation)->poleAngle += 1;
          if (((HHBamse*)myRotation)->poleAngle <= 90)
                ((HHBamse*)myRotation)->rotate(1, (float)(90 - ((HHBamse*)myRotation)-
>poleAngle), 3);

          ((HHBamse*)myRotation)->headAngle += 2;
          if (((HHBamse*)myRotation)->headAngle <= 180)
                ((HHBamse*)myRotation)->rotate(1, (float)(180 - ((HHBamse*)myRotation)-
>headAngle), 4);
          else
          {
                ((HHBamse*)myRotation)->poleRun = FALSE;
                delete ((HHBamse*)myRotation)->poleTask;
                ((HHBamse*)myRotation)->poleTask = NULL;
          }
     }
     else
     {
          ((HHBamse*)myRotation)->poleAngle -= 1;
          if (((HHBamse*)myRotation)->poleAngle >= 0)
                ((HHBamse*)myRotation)->rotate(1, (float)(90 - ((HHBamse*)myRotation)-
>poleAngle), 3);

          ((HHBamse*)myRotation)->headAngle -= 2;
          if (((HHBamse*)myRotation)->headAngle >= 0)
                ((HHBamse*)myRotation)->rotate(1, (float)(180 - ((HHBamse*)myRotation)-
>headAngle), 4);
          else
```

```
            {
                  ((HHBamse*)myRotation)->poleRun = FALSE;
                  delete ((HHBamse*)myRotation)->poleTask;
                  ((HHBamse*)myRotation)->poleTask = NULL;
            }
      }
}

/**  The rotation task function for the HHBamse instance's radar 'head' geometry.
  */
void HHBamse::HHRadarTask(void *myRotation)
{
      if (((HHBamse*)myRotation)->radarAngle < 0)
      {
            ((HHBamse*)myRotation)->radarAngle += 7.f * PI/180;
            if (((HHBamse*)myRotation)->radarAngle > 0) ((HHBamse*)myRotation)->radarAngle = 0;
      }
      else if (((HHBamse*)myRotation)->radarAngle > 0)
      {
            ((HHBamse*)myRotation)->radarAngle -= 7.f * PI/180;
            if (((HHBamse*)myRotation)->radarAngle < 0) ((HHBamse*)myRotation)->radarAngle = 0;
      }

      ((HHBamse*)myRotation)->radarRotation(0.f);   // Just to update the screen!

      if (0 == ((HHBamse*)myRotation)->radarAngle)
      {
            delete ((HHBamse*)myRotation)->radarTask;
            ((HHBamse*)myRotation)->radarTask = NULL;
      }
}

/**  The rotation task function for the HHBamse instance's radar disc geometry.
  */
void HHBamse::HHScreenTask(void *myRotation)
{
      ((HHBamse*)myRotation)->screenAngle -= PI/180;
      if (((HHBamse*)myRotation)->screenAngle < 0) ((HHBamse*)myRotation)->screenAngle = 0;

      ((HHBamse*)myRotation)->screenRotation(0.f);  // Just to update the screen!

      if (0 == ((HHBamse*)myRotation)->screenAngle)
      {
            delete ((HHBamse*)myRotation)->screenTask;
            ((HHBamse*)myRotation)->screenTask = NULL;
      }
}

/*****************************************************************************/
/*** PRIVATE METHODS...                                                    ***/
/*****************************************************************************/

/**  Loads all the geometry used by the instance and rotates it 180 degrees, to compensate
      for the fact that VRML 1.0 has an inverse coordinate system compared to WTK.
  */
void HHBamse::loadGeometry()
{
      float trans[3] = {-2, -2, 0};
      WtP3 turretTrans(trans);
      whole = MovNodeLoad("xxxxBamse.wrl", 1.f);
      whole->SetName("Whole");

      computeTurrets();
      computePole();

      separateTop();

      theParent = NULL;
      ((WtMovable*)whole)->MovAxisRotation(Z, PI);
      ((WtMovable*)whole)->GetTransform(wholeTrans);

      loadMissile();
}

/**  Destroys the geometry that previously have been loaded.
      PRECON: The geometry must have been loaded using loadGeometry().
  */
```

```
void HHBamse::destroyGeometry()
{
    delete whole;
    //delete turret;
}

/**  Rotates the HHBamse instance's turret geometry the given angle (in radians)
     around a point at the base of the missile turret.
  */
void HHBamse::turretRotation(float angle, int part)
{
    WtP3 translation;

    if (1 == part)
    {
           translation[0] = leftExt[0];
           translation[1] = -leftExt[1];
           translation[2] = 0;
           ((WtMovable*)left)->Translate(translation, WTFRAME_LOCAL);

           ((WtMovable*)left)->MovAxisRotation(Z, angle);

           translation[0] *= -1;
           translation[1] *= -1;
           ((WtMovable*)left)->Translate(translation, WTFRAME_LOCAL);
    }
    else if (2 == part)
    {
           translation[0] = rightExt[0];
           translation[1] = -rightExt[1];
           translation[2] = 0;
           ((WtMovable*)right)->Translate(translation, WTFRAME_LOCAL);

           ((WtMovable*)right)->MovAxisRotation(Z, angle);

           translation[0] *= -1;
           translation[1] *= -1;
           ((WtMovable*)right)->Translate(translation, WTFRAME_LOCAL);
    }
}

/**  Rotates the radar pylon geometry the specified angle.
  */
void HHBamse::poleRotation(float angle)
{
    WtM4 r, s;
    WtP3 translation;

    ((WtMovable*)radar)->GetTransform(r);
    ((WtMovable*)screen)->GetTransform(s);

    ((WtMovable*)pole)->SetTransform(poleTrans);
    ((WtMovable*)head)->SetTransform(headTrans);
    ((WtMovable*)radar)->SetTransform(radarTrans);
    ((WtMovable*)screen)->SetTransform(screenTrans);

    translation[0] = 8.f;
    translation[1] = -poleExt[1];
    translation[2] = 0;
    ((WtMovable*)pole)->Translate(translation, WTFRAME_LOCAL);

    ((WtMovable*)pole)->MovAxisRotation(Z, -angle);

    translation[0] *= -1;
    translation[1] *= -1;
    ((WtMovable*)pole)->Translate(translation, WTFRAME_LOCAL);

    ((WtMovable*)radar)->SetTransform(r);
    ((WtMovable*)screen)->SetTransform(s);
}

/**  Rotates the radar the given angle.
  */
void HHBamse::headRotation(float angle)
{
    WtM4 r, s;
    WtP3 translation;
```

```
    ((WtMovable*)radar)->GetTransform(r);
    ((WtMovable*)screen)->GetTransform(s);


    ((WtMovable*)head)->SetTransform(headTrans);
    ((WtMovable*)radar)->SetTransform(radarTrans);
    ((WtMovable*)screen)->SetTransform(screenTrans);


    translation[0] = 0;
    translation[1] = -headExt[1];
    translation[2] = 0;
    ((WtMovable*)head)->Translate(translation, WTFRAME_LOCAL);

    ((WtMovable*)head)->MovAxisRotation(Z, angle);

    translation[1] *= -1;
    ((WtMovable*)head)->Translate(translation, WTFRAME_LOCAL);

    ((WtMovable*)radar)->SetTransform(r);
    ((WtMovable*)screen)->SetTransform(s);
}

/**  Rotates the base of the missile turrets the given angle.
  */
void HHBamse::topRotation(float angle)
{
    WtM4 h, p, l, r, rd, s;
    WtP3 translation;

    ((WtMovable*)head)->GetTransform(h);
    ((WtMovable*)pole)->GetTransform(p);
    ((WtMovable*)left)->GetTransform(l);
    ((WtMovable*)right)->GetTransform(r);
    ((WtMovable*)radar)->GetTransform(rd);
    ((WtMovable*)screen)->GetTransform(s);

    ((WtMovable*)head)->SetTransform(headTrans);
    ((WtMovable*)pole)->SetTransform(poleTrans);
    ((WtMovable*)left)->SetTransform(leftTrans);
    ((WtMovable*)right)->SetTransform(rightTrans);
    ((WtMovable*)radar)->SetTransform(radarTrans);
    ((WtMovable*)screen)->SetTransform(screenTrans);
    if (missileRun)
    {
          ((WtMovable*)missile)->SetTransform(missileTrans);
          ((WtMovable*)motor)->SetTransform(motorTrans);
    }

    translation[0] = -31.f;
    translation[1] = 0;
    translation[2] = 0;
    ((WtMovable*)top)->Translate(translation, WTFRAME_LOCAL);

    ((WtMovable*)top)->MovAxisRotation(Y, angle);

    translation[0] *= -1;
    ((WtMovable*)top)->Translate(translation, WTFRAME_LOCAL);

    ((WtMovable*)head)->SetTransform(h);
    ((WtMovable*)pole)->SetTransform(p);
    ((WtMovable*)left)->SetTransform(l);
    ((WtMovable*)right)->SetTransform(r);
    ((WtMovable*)radar)->SetTransform(rd);
    ((WtMovable*)screen)->SetTransform(s);
}

/**  Rotates the radar head the given angle.
  */
void HHBamse::radarRotation(float angle)
{
    WtM4 s;
    WtP3 translation;

    ((WtMovable*)screen)->GetTransform(s);
    ((WtMovable*)radar)->SetTransform(radarTrans);
```

```
    ((WtMovable*)screen)->SetTransform(screenTrans);

    if ((radarAngle + angle) < (-3*PI))
           angle = (-3*PI) - radarAngle;
    else if ((radarAngle + angle) > (3*PI))
           angle = (3*PI) - radarAngle;
    radarAngle += angle;

    translation[0] = 5.f;
    translation[1] = 0.f;
    translation[2] = 0.f;
    ((WtMovable*)radar)->Translate(translation, WTFRAME_LOCAL);

    ((WtMovable*)radar)->MovAxisRotation(Y, radarAngle);

    translation[0] *= -1;
    ((WtMovable*)radar)->Translate(translation, WTFRAME_LOCAL);

    ((WtMovable*)screen)->SetTransform(s);
}

/**  Rotates the radar disc the given angle.
  */
void HHBamse::screenRotation(float angle)
{
    WtP3 translation;

    if ((screenAngle + angle) < 0)
           angle = -screenAngle;
    else if ((screenAngle + angle) > (PI/2))
           angle = (PI/2) - screenAngle;
    screenAngle += angle;

    ((WtMovable*)screen)->SetTransform(screenTrans);

    translation[0] = 4.5f;
    translation[1] = 5.f;
    translation[2] = 0.f;
    ((WtMovable*)screen)->Translate(translation, WTFRAME_LOCAL);

    ((WtMovable*)screen)->MovAxisRotation(Z, -screenAngle);

    translation[0] *= -1;
    translation[1] *= -1;
    ((WtMovable*)screen)->Translate(translation, WTFRAME_LOCAL);
}

/**  Separates the movable parts of the geometry from the other geometry and
     converts them to movable nodes.
  */
void HHBamse::separateNode(WtNode *currentNode, char* nodeName, HHParts whichPart)
{
    if (rockOn)
    {
           char *name = currentNode->GetName();
           if ((NULL != name)&&(0 == strcmp(name, nodeName)))
           {
                 theParent = currentNode->GetParent(0);
                 name = theParent->GetName();

                 if (NULL == theParent)
                       WTmessage("Unable to trace node...\n");
                 else
                 {
                       int childNum = ((WtGroup*)theParent)->NumChildren();
                       for (theLocation=0; theLocation<childNum; theLocation++)
                             if (((WtGroup*)theParent)->GetChild(theLocation) == currentNode)
break;

                       switch (whichPart)
                       {
                             case LTURRET:    left->AddChild(currentNode);
                                              break;

                             case RTURRET:    right->AddChild(currentNode);
                                              break;
```

```
                              case POLE:       pole->AddChild(currentNode);
                                               break;

                              case HEAD:       head->AddChild(currentNode);
                                               break;

                              case TOP:        top->AddChild(currentNode);
                                               break;

                              case RADAR:      radar->AddChild(currentNode);
                                               break;

                              case SCREEN:     screen->AddChild(currentNode);
                                               break;
                        }
                  }

                  rockOn = FALSE;
            }
            else
            {
                  int children = ((WtGroup*)currentNode)->NumChildren();
                  if (0 != children)
                  {
                        for (int i=0; i<children; i++)
                              separateNode(((WtGroup*)currentNode)->GetChild(i), nodeName,
whichPart);
                  }
            }
      }
}

/**  Performs all the calculations needed to convert the missile turrets to movable nodes.
  */
void HHBamse::computeTurrets()
{
      left = new WtMovSep(NULL);
      left->SetName("Left-turret");
      rockOn = TRUE;
      separateNode(whole, "DOF_H_RampV12", LTURRET);
      if (NULL != theParent)
      {
            ((WtGroup*)theParent)->InsertChild(left, theLocation);
            ((WtGroup*)theParent)->DeleteChild(theLocation+1);
      }
      rockOn = TRUE;
      separateNode(whole, "H_RorVU13", LTURRET);
      if (NULL != theParent) ((WtGroup*)theParent)->DeleteChild(theLocation);
      rockOn = TRUE;
      separateNode(whole, "H_RorVN14", LTURRET);
      if (NULL != theParent) ((WtGroup*)theParent)->DeleteChild(theLocation);
      left->GetExtents(leftExt);
      ((WtMovable*)left)->GetTransform(leftTrans);


      // Search for right turret
      right = new WtMovSep(NULL);
      right->SetName("Right-turret");
      rockOn = TRUE;
      separateNode(whole, "DOF_H_RampH15", RTURRET);
      if (NULL != theParent)
      {
            ((WtGroup*)theParent)->InsertChild(right, theLocation);
            ((WtGroup*)theParent)->DeleteChild(theLocation+1);
      }
      rockOn = TRUE;
      separateNode(whole, "H_RorHU16", RTURRET);
      if (NULL != theParent) ((WtGroup*)theParent)->DeleteChild(theLocation);
      rockOn = TRUE;
      separateNode(whole, "H_RorHN17", RTURRET);
      if (NULL != theParent) ((WtGroup*)theParent)->DeleteChild(theLocation);
      right->GetExtents(rightExt);
      ((WtMovable*)right)->GetTransform(rightTrans);
}

/**  Performs all calculations needed to convert the whole radar pylon to movable
      nodes.
```

```
  */
void HHBamse::computePole()
{
    // Search for pole
    pole = new WtMovSep(NULL);
    pole->SetName("Pole");
    rockOn = TRUE;
    separateNode(whole, "o1719", POLE);
    if (NULL != theParent)
    {
          ((WtGroup*)theParent)->InsertChild(pole, theLocation);
          ((WtGroup*)theParent)->DeleteChild(theLocation+1);
    }
    rockOn = TRUE;
    separateNode(whole, "o4136", POLE);
    if (NULL != theParent) ((WtGroup*)theParent)->DeleteChild(theLocation);

    // Search for head
    head = new WtMovSep(NULL);
    head->SetName("Head");
    rockOn = TRUE;
    separateNode(whole, "o1618", HEAD);
    if (NULL != theParent)
    {
          pole->AddChild(head);
          ((WtGroup*)theParent)->DeleteChild(theLocation);
    }
    rockOn = TRUE;
    separateNode(whole, "o4035", HEAD);
    if (NULL != theParent) ((WtGroup*)theParent)->DeleteChild(theLocation);
    rockOn = TRUE;
    separateNode(whole, "H_Vagga20", HEAD);
    if (NULL != theParent) ((WtGroup*)theParent)->DeleteChild(theLocation);
    rockOn = TRUE;
    separateNode(whole, "M_Vagga37", HEAD);
    if (NULL != theParent) ((WtGroup*)theParent)->DeleteChild(theLocation);

    radar = new WtMovSep(head);
    radar->SetName("Radar");
    rockOn = TRUE;
    separateNode(whole, "o2621", RADAR);
    if (NULL != theParent) ((WtGroup*)theParent)->DeleteChild(theLocation);
    rockOn = TRUE;
    separateNode(whole, "o4238", RADAR);
    if (NULL != theParent) ((WtGroup*)theParent)->DeleteChild(theLocation);

    screen = new WtMovSep(radar);
    screen->SetName("RadarScreen");
    rockOn = TRUE;
    separateNode(whole, "d522", SCREEN);
    if (NULL != theParent) ((WtGroup*)theParent)->DeleteChild(theLocation);
    rockOn = TRUE;
    separateNode(whole, "d1539", SCREEN);
    if (NULL != theParent) ((WtGroup*)theParent)->DeleteChild(theLocation);


    head->GetExtents(headExt);
    pole->GetExtents(poleExt);
    screen->GetExtents(screenExt);
    ((WtMovable*)head)->GetTransform(headTrans);
    ((WtMovable*)pole)->GetTransform(poleTrans);
    ((WtMovable*)radar)->GetTransform(radarTrans);
    ((WtMovable*)screen)->GetTransform(screenTrans);
}

/**  Separates and converts the base of the missile turrets.
  */
void HHBamse::separateTop()
{
    top = new WtMovSep((WtGroup*)whole);
    top->SetName("Top");
    rockOn = TRUE;
    separateNode(whole, "o2511", TOP);
    if (NULL != theParent) ((WtGroup*)theParent)->DeleteChild(theLocation);
    rockOn = TRUE;
    separateNode(whole, "o3928", TOP);
    if (NULL != theParent) ((WtGroup*)theParent)->DeleteChild(theLocation);
```

```
    rockOn = TRUE;
    separateNode(whole, "Left-turret", TOP);
    if (NULL != theParent) ((WtGroup*)theParent)->DeleteChild(theLocation);
    rockOn = TRUE;
    separateNode(whole, "Right-turret", TOP);
    if (NULL != theParent) ((WtGroup*)theParent)->DeleteChild(theLocation);

    ((WtMovable*)top)->GetTransform(topTrans);
}


/***************** MISSILE RELATED CODE ************************/


/**  Loads the geometry needed for the missile.
  */
void HHBamse::loadMissile()
{
    missile = MovNodeLoad("xMissile.wrl", 1.f);
    missile->SetName("Missile");

    motor = MovNodeLoad("xMotor.wrl", 1.f);
    motor->SetName("Motor");
    ((WtGroup*)missile)->AddChild(motor);
    ((WtMovable*)motor)->GetTransform(initMotorTrans);
    ((WtMovable*)missile)->GetTransform(initMissileTrans);
}

/**  Shoots a missile from one of the turrets, specified by whichOne.
  */
BOOL HHBamse::shoot(int whichOne)
{
    if (!missileRun)
    {
          if ((((whichOne < 3)&&(leftElevate)) || ((whichOne >=
3)&&(rightElevate)))&&(!leftRun)&&(!rightRun))
          {
               ((WtMovable*)motor)->SetTransform(initMotorTrans);
               ((WtMovable*)missile)->SetTransform(initMissileTrans);

               if (whichOne < 3)
                    ((WtGroup*)left)->AddChild(missile);
               else
                    ((WtGroup*)right)->AddChild(missile);

               switch(whichOne)
               {
                    case 1:    mTranslation[0] = -40.f;
                               mTranslation[1] = 62.f;
                               mTranslation[2] = 0.f;
                               break;

                    case 2:    mTranslation[0] = -40.f;
                               mTranslation[1] = 42.f;
                               mTranslation[2] = 0.f;
                               break;

                    case 3:    mTranslation[0] = -40.f;
                               mTranslation[1] = 62.f;
                               mTranslation[2] = -60.5f;
                               break;

                    case 4:    mTranslation[0] = -40.f;
                               mTranslation[1] = 42.f;
                               mTranslation[2] = -60.5f;
                               break;
               }

               ((WtMovable*)missile)->Translate(mTranslation, WTFRAME_LOCAL);
               ((WtMovable*)missile)->MovAxisRotation(Z, PI/6.f);

               ((WtMovable*)missile)->GetTransform(missileTrans);
               ((WtMovable*)motor)->GetTransform(motorTrans);
               ((WtMovable*)missile)->GetTranslation(mTranslation);
               ((WtMovable*)motor)->GetTranslation(moTranslation);
               missileRun = TRUE;
```

```
                missileSeparated = FALSE;
                missileCount = 0;
                motorAngle = 0.f;
                missileTask = new WtTask(this, HHBamse::HHMissileTask, 1.f);
                return TRUE;
            }
    }

    return FALSE;
}

/**  The task function that handles the movement of the missile!
  */
void HHBamse::HHMissileTask(void *myRotation)
{
    WtNode *temp;

    temp = ((HHBamse*)myRotation)->motor;
    ((WtMovable*)temp)->SetTransform(((HHBamse*)myRotation)->motorTrans);
    temp = ((HHBamse*)myRotation)->missile;
    ((WtMovable*)temp)->SetTransform(((HHBamse*)myRotation)->missileTrans);

    if (!((HHBamse*)myRotation)->missileSeparated)
    {
           ((HHBamse*)myRotation)->mTranslation[0] += -15.f;
           ((WtMovable*)temp)->SetTranslation(((HHBamse*)myRotation)->mTranslation);
    }
    else
    {
           ((HHBamse*)myRotation)->mTranslation[0] += -45.f;
           ((WtMovable*)temp)->SetTranslation(((HHBamse*)myRotation)->mTranslation);

           temp = ((HHBamse*)myRotation)->motor;
           ((HHBamse*)myRotation)->missileCount++;

           ((HHBamse*)myRotation)->motorAngle += PI / 30.f;
           ((WtMovable*)temp)->MovAxisRotation(Z, ((HHBamse*)myRotation)->motorAngle);

           ((HHBamse*)myRotation)->moTranslation[0] += 45.f * 0.9f - 25 /
((HHBamse*)myRotation)->missileCount;
           ((HHBamse*)myRotation)->moTranslation[1] += -27.0f * 1.5f - ((HHBamse*)myRotation)-
>missileCount;
           ((HHBamse*)myRotation)->moTranslation[2] += 0.f;
           ((WtMovable*)temp)->SetTranslation(((HHBamse*)myRotation)->moTranslation);
    }

    if (((HHBamse*)myRotation)->mTranslation[0] < -200) ((HHBamse*)myRotation)-
>missileSeparated = TRUE;
    if (((HHBamse*)myRotation)->mTranslation[0] < -2650)
    {
           ((HHBamse*)myRotation)->missile->Remove();
           ((HHBamse*)myRotation)->missileRun = FALSE;
           delete ((HHBamse*)myRotation)->missileTask;
           ((HHBamse*)myRotation)->missileTask = NULL;
    }
}
```