Master of Science Thesis
Department of Computer Science
Lund Institute of Technology

Haptic hardware support in a 3D game engine

Henrik Aamisepp [d98ha@efd.lth.se]
Daniel Nilsson [d96dn@efd.lth.se]

May 2003

Supervisors: Lennart Ohlsson, Lund Institute of Technology
Charlotte Magnusson, Lund Institute of Technology

Abstract

The aim of this master thesis is to find out if it is possible to integrate haptic hardware support in the open source 3D game engine Crystal Space. Integrating haptic support would make it possible to get a haptic representation of 3D geometry in Crystal Space and therefore take advantage of all the benefits a 3D game engine provides, when building haptic applications. An implementation of the support should be as low-cost as possible by taking advantage of available open source haptic API alternatives.

The thesis report presents an evaluation of available haptic APIs and comes up with a design and an implementation. The solution has been implemented as a Crystal Space plugin by using modified parts of the e-Touch open module API. The plugin makes it possible to utilize the Phantom haptic device to touch and feel the 3D environments in a Crystal Space application. Two demo applications have also been constructed to show the capabilities of the plugin.

Contents

1 Introduction

1.1 Background

At the Department of Design Sciences at Lund Institute of Technology there is a division for rehabilitation engineering called Certec. Among other things they do research on how to make computers accessible to blind and visually impaired people through the use of haptic technology. This means that they try to make it possible to use computers by using the sense of touch to navigate through a 3D interface. The force feedback hardware that is available in stores today cannot provide the accuracy that is needed for such a task. However there is available hardware that can be used, such as the Phantom haptic device from Sensable Technologies, Inc. The problem is that this hardware is very expensive and is therefore not available to a great extent. This in turn means that there are only a few APIs available for creating haptic environments and they do not always provide the same possibilities to easily create good 3D applications as for example a game engine would. The APIs are not designed to be integrated with other systems that can provide this functionality either.

The wish is that it should be possible to build large 3D environments that the user can navigate in and use the haptic device to touch the surroundings. These kinds of environments are very common in modern 3D computer games and thus it would be desirable to have support for haptic devices in a 3D game engine. This would make it possible to create 3D environments using existing 3D map editors, or perhaps even to use existing 3D game maps that can be downloaded from the Internet.

This master thesis is a collaboration between Certec and the Department of Computer Science at Lund Institute of Technology. The idea is to add support for haptics in the open source game engine Crystal Space, by using parts of available haptic APIs, such as e-Touch from Novint Technologies, Inc. or GHOST from Sensable Technologies Inc. It is required that the solution should support the Phantom haptic device, but it would be an advantage if there also was support for other force feedback devices.

1.2 Goals

The main goal of this master thesis is to investigate if it is possible to integrate haptics in the Crystal Space game engine. The investigation should be based upon an evaluation of different haptic APIs and how well they are suited to be integrated in Crystal Space. A software implementation will be constructed to show the possibilities of a system that integrates haptics with a game engine. The implementation should be able to give a haptic representation of 3D geometry in Crystal Space. The development process should be used to gather knowledge about the structure of the existing APIs and identify the difficulties that are associated with the integration of haptic and graphic APIs.

1.3 Document structure

The first part of this document handles the concepts of haptics and game engines. There are introductions to these topics as well as an in-dept study of different tools that are of interest to this master thesis. The second part of the document deals with the work that has been made in this master thesis project along with a presentation of results and conclusions. To explain how the developed software should be used there are also three appendices at the end of the document.

An introduction to haptics is given in chapter 2. This chapter also contains a description of the Phantom haptic device and an overview of haptic rendering. A study of different haptic APIs is also contained within this chapter.

Evaluations of the different APIs are made in chapter 3. The advantages and disadvantages of the APIs are weighed against each other and a conclusion is made about what parts are suitable for an implementation.

The concept of game engines is explained in chapter 4. This chapter also contains an overview of the Crystal Space game engine. The end of the chapter gives a short description of map creation and the Valve Hammer Editor.

Chapter 5 covers the design and implementation of the haptic plugin. The problems that occurred during the development process are presented along with our solutions. There is also a part on restrictions that had to be made.

The demo applications made to show the functionality of the plugin are described in chapter 6. The chapter gives a description on what the demos display and there is also a short overview of their design.

Chapter 7 contains the summary of the master thesis project. The results that have been accomplished are presented and evaluated. There is a part about suggestions of future improvements. Finally conclusions are drawn about the complete project and ideas on how the haptic and graphic APIs could have been designed differently to be easier to integrate.

Appendix A contains a description on requirements, methods and how to load the plugin in a Crystal Space application.

Appendix B gives a brief description on how to use the demo applications and their features.

Appendix C is a guide on how to construct worlds and objects that can be used together with the haptic plugin.

Appendix D contains a small dictionary.

2 Haptics

2.1 Introduction to haptics

Haptics originates from the Greek word haptesthai ("to touch") and is a science that studies the sense of touch. In the computer world, haptics deals with using the sense of touch to control and interact with a computer application. To be able to do this, special input/output devices, such as joysticks, wheels, data gloves or more advanced devices are connected to the computer.

The feedback from these devices is delivered as felt sensations to the user's hand or other parts of his body. The user can then interact with this feedback and control the application according to the sensations he experiences. By using a haptic device, three dimensional virtual objects do not only have to exist as graphics, but they can also be represented in haptics as physical objects. This opens up a lot of possibilities for many different kinds of applications. For example, haptics have been utilized to train people in surgery and operation of machines in hostile environments. Additional examples of the use of haptics include the entertainment business, where haptics has been used to give the player force feedback of the events occurring in a computer game. Haptics can also help to guide blind and visually impaired people in computer applications.

Three different concepts are often mentioned in connection with haptics: force feedback, tactile feedback and proprioception. Force feedback devices are often mechanical devices that can deliver forces to the user through electric motors. With force feedback one can get the feeling of weight or resistance in a virtual world. The device that is used should be able to produce an equivalent or scaled force to that of what a real object would deliver if it was touched. Force feedback can usually stop a user's motion, while tactile feedback cannot.

Tactile feedback is more about making the user sense different types of surfaces, textures and vibrations through the skin. Often one or more fingers are used to feel the tactile sensations. To achieve the effect of tactile feedback, pneumatic or electronic solutions are used. For example, when a data glove is used as a feedback device, small airbags in the glove's fingers are filled and deflated to get the feeling of a vibration or a texture. Another solution to achieve tactile feedback is to use a surface with small pin arrays that can be raised and lowered to give the user feelings of different textures.

The last concept is proprioception. Proprioception is an automatic sensitivity mechanism that is utilized by the body to send messages through the central nervous system. The messages are sent from the central nervous system to different parts of the body to inform how it should react to stimuli and with what strength. One could say that proprioception is the sense of position of the body in relation to gravity as well as our movement through space [19]. With computers it is more difficult to get a good representation of proprioception than it is with tactile feedback or force feedback.

This master thesis will make use of a haptic device that only utilizes force feedback, the Phantom haptic device. A place to find recent research articles and gain more information about haptics is the Haptics-e homepage [21].

2.2 History of haptics

One may think that the development of haptic devices and the applications adherent to them originates from the development of virtual reality, but this is actually incorrect. The first devices that one may state as haptic devices come from the teleoperation systems of the 1950s and 60s. In a teleoperation system, the user controls and manipulates objects from a remote location. These systems often use bilateral Master-Slave Manipulators (MSM), in which a master device follows and interacts with a remote slave device. A classic example of the field of application for MSM is within dangerous or unhealthy environments, like the handling of nuclear waste or underwater operations. In these cases some type of a mechanical arm is used as the master unit and a smaller similar reproduction of this is used as the slave device. These systems used to be all mechanical, but they were later made electrical which enabled the operator to work further away from the site. It also allowed the introduction of so called servomanipulators, which could give force feedback to the slave device.

A project started in 1967, which used MSM to join interactive computer graphics with force feedback, was the GROPE (I, II, III) project. This project, whose goal was to create a three dimensional real-time system to simulate forces when different substrate and molecules interacted and docked with each other, turned out to last for 20 years.

Another research area that made progress in haptics during the 1960s and 70s, but today is not as popular as it used to be, is exoskeleton research. An exoskeleton is a device that looks similar to a robot, but the user is situated inside it and is able to control the legs and arms of the exoskeleton. These machines made it possible to lift and move heavy loads and simultaneously get a reduced force feedback. However problems with unstable control, danger of leaking hydraulic and limitation in functional anatomy of the human arm have made the research area less popular.

During the 1960 there where also research in the area of sensing gloves, which use pneumatic bladders to produce feedback for independent fingers. They were made more sophisticated during the 1980s. One of the gloves developed in the late 1980s was Teletact. This glove first supported 20 inflatable air pockets, but it was later refined, with the new name Teletact II, in the early 1990s to support 30 air pockets.

The 1990s saw the introduction of desktop force feedback joysticks and force feedback wheels for games and entertainment. These devices serve their purpose quite well, but they are limited in their degree of freedom and sensitivity, making them hard to use for more advanced applications. If one only wants to simulate the vibration of a steering wheel in a racing game or the recoil of a gun in a shooting game though, they work just fine.

The Virtual reality field appeared first in the late 1970s. This started a development of special purpose tactile and force feedback devices. However it was not until the beginning and middle of the 1990s that the first commercial haptic VR devices where

introduced. Some of the companies responsible for these devices were Immersion Co. (Impulse Engine) [20] and Sensable Technologies, Inc. (Phantom) [15]. These two companies are still today two of the leading companies in haptic technology. For a more complete history of haptics, see G. C. Burdea (1996) [7] or R. J. Stone (2000) [10].

2.3 The Phantom

The Phantom was originally designed and built at MIT in 1993 by Thomas Massie and Dr. Kenneth Salisbury. They were looking for a way to reach and feel three dimensional data. It began as a thesis and ended as the company Sensable Technologies, Inc. that today has customers worldwide.

The Phantom is a desktop unit that looks like a mechanical robot arm that is holding a pen. When holding the pen (stylus) in the hand it is possible to touch, feel and manipulate computer models with six degrees of freedom. The Phantom utilizes point interaction, which means that the user will sense the virtual environment through a single point. This makes the sensation somewhat like touching objects with the end of a stick. It is possible to connect several Phantom devices to a computer and thereby get multiple interaction points.

There is an entire family of Phantom devices with different sized workspaces and different values for maximum exertable force. Three different DC motors inside the Phantom deliver the force. In this master thesis we have used both the Phantom desktop device, which is the smallest of the Phantom family devices, and an older version of the Phantom family, the Phantom Premium 1.0. The desktop model easily connects to the computer via the parallel port and in addition to that it has a very portable design. The Phantom Premium 1.0 model has a somewhat larger workspace and a bit larger maximum exertable force. The Premium model also requires a certain interface card to connect the Phantom to the computer. Both models provide force feedback in three degrees of freedom and position sensing in six degrees of freedom. The pitch-, yaw- and roll angles of the Phantom stylus are only measurable and cannot be controlled with force feedback. Besides that the stylus features a switch that can be used in the same way as a mouse button. Further information about the Phantom device can be found at Sensable's homepage [15].

Figure 2.1 *The Phantom haptic device. The left picture shows the Phantom desktop model and the right picture shows the Phantom Premium 1.0 model.*

2.4 Haptic rendering

One of the difficulties when using haptics is that the update frequency needs to be very high. A usual acceptable frame rate for a graphics loop is 60-80 Hz, whereas a haptics loop needs a cycle rate of about 800-1000 Hz. This update rate is required because it allows the motors that deliver the force to change fast enough, so that the user can get a consistent object representation and a stable haptic device. Otherwise surfaces can be represented too soft or even worse, the haptic device can start to vibrate.

Haptic programming interfaces often separate the work of the haptics loop from the work of the application and the graphic presentation. By putting them in two different processes, the haptics loop can be given a higher priority and the high cycle rate can be met. This means that when creating a haptic API one needs to think about synchronizing the two processes. For synchronization from haptics to graphics this is usually not a big problem because the haptics loop runs faster than the graphics loop. However if something is moved in graphics, for example the camera, the haptics needs this change in position more often than the graphics.

A haptic rendering algorithm consists of two parts: collision detection and a collision response. Both of these calculations need to be very fast to maintain the high update rate that is desired. A simple example to describe the concept of haptic rendering is the rendering of a haptic sphere (see Figure 2.2). Let us say that the sphere is situated at the origin of the virtual world and that the device used can only interact with it through a single interaction point, the end point of the haptic device. When the user moves the haptic device around without touching the virtual sphere no forces are delivered, but when the user penetrates the sphere the device will produce opposing forces to resist further penetration.

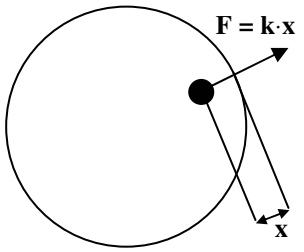

Figure 2.2 *Haptic rendering of a sphere. The reaction force is proportional to the penetration depth.*

If one assumes no friction, the magnitude of the force will be proportional to the depth of penetration and the direction of the force will be in the direction of the surface normal. The force calculation can then be modeled like the following equation (similar to Hook's law):

$$F = k \cdot x \tag{2.1}$$

where k represents the stiffness coefficient and x represents the penetration depth. A low value for the stiffness constant makes the object feel soft and a high value makes it feel hard.
By adding a second damping term, as seen below, depending on the velocity one can get a stiffer surface. But due to the nature of haptics it is not possible to have an

infinitely stiff surface. This means that it will always feel a bit soft even if one wants to represent a completely stiff material.

$$F = k \cdot x + b \cdot V \tag{2.2}$$

Here V is the velocity vector for the user point and b is a damping constant. Using too high values for the damping constant may however cause unwanted vibrations. The concept of haptic rendering seems simple when looking at the example with the sphere above, but the rendering of several and more complex 3D polygon objects requires more sophisticated algorithms that are described below.

A haptic algorithm consists, as mentioned before, of both collision detection and collision response. Collision detection algorithms have been studied in computer graphics for several years, but these algorithms are not directly suitable to haptic rendering. However, numerous concepts like space partitioning algorithms, local search approaches and hierarchical data structures can be used. Space partitioning algorithms encloses objects in smaller subspaces for fast detection of first contact. Local search approaches searches only the neighbor primitives for collision and hierarchical data structures are used for arranging links between the primitives that make up the object.

To create collision detection for haptics, one has to check if the cursor has moved through any of the polygons of the objects between loops. When using a haptic device that only has one interaction point, the check can be done by comparing the line that is created from taking the previous position of the haptic tool and the current position of the haptic tool and see if this line intersects any of the polygons in the object. This will however take a lot of time if the object has many polygons. If one for example uses a space partitioning algorithm, like an octree, for the object and only check the polygons that are close to the haptic cursor, the time can be severely reduced. An octree is a simple but powerful tree structure that divides 3D objects or 3D worlds into smaller volumes to simplify the handling of what is of interest for the collision detection. A tree is built up for each object before starting the main graphics loop by first surrounding the object by a box, which will become the root of the tree. The object is then divided into half along the x-, y- and z-axis to create eight smaller boxes. These boxes become children of the root. This iteration continues until a condition is filled, for example when a certain number of polygons remain within a box or a certain depth in the tree has been reached. Of course the iteration also stops if there are no more polygons in the current box. Only the polygons enclosed within the box that contains the haptic cursor needs to be checked. This way the octree provides a structure to handle real-time collision detection with large data sets.

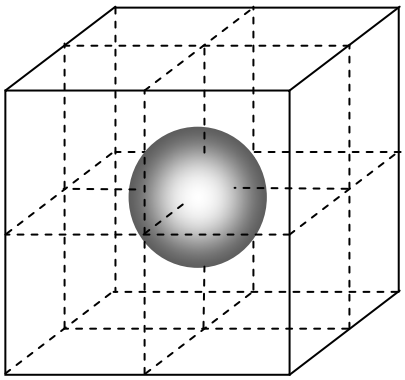

Figure 2.3 *An octree dividing a sphere one time into eight boxes. Only the part of the sphere contained within the same box as the haptic device needs to be checked for collision.*

After the collision detection has taken place the collision response or the force generation occurs. If a polygon model is used to represent the object, then as said before, the force will be in the direction of the normal and proportional to the penetration depth of the currently active polygon. There are however still some issues to take care of.
In graphics it is possible to make a polygon visible from different sides depending on the order of its vertices. This also applies to haptics, where the polygons can be made touchable on the front-face, back-face or both. This means that the list of vertices should be ordered correctly to get the feedback in the correct direction.

Another issue is, knowing which polygon is active when sliding from one polygon to another in an object. Using the projection of the polygon neighbors as borders for changing the active polygons can create a jerk in the force. This happens because the penetration depth can be different after the change has been made. A solution to this is to use an edge plane between the polygons. An edge plane is a plane created from two polygon vertices that the polygons have in common and the average of the normals of the two polygons (See figure 2.4). Using this technique will make the penetration depth consistent when moving from one polygon to another.

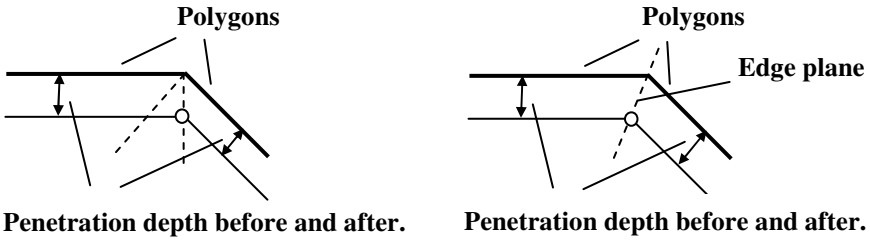

Figure 2.4 *Not using edge planes may cause the penetration depth to become different when moving from one polygon to another. This will cause a sudden change in the resulting force. Using edge planes reassures that the penetration depth is consistent when moving between polygons.*

On top of this one may want to keep the edges smooth. Interpolating forces between neighboring polygons, in a similar way to Phong shading in computer graphics can achieve this. If the distances from the cursor to the edge planes are known, the normal direction can be interpolated and normalized. Not using interpolation will result in sharp edges, while using interpolation, and modifying the size of the area where interpolation should occur, will make the objects feel rounder than they actually are.

Interpolation can be used to create smoother objects without having to use additional polygons.

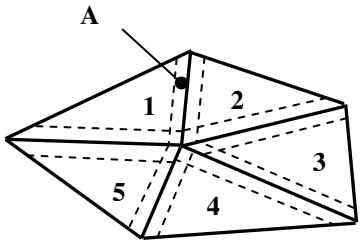

Figure 2.5 *Interpolation of normals to get smooth edges. For example, if the haptic tool is at position A, the normal is interpolated from the normals of triangles 1 and 2.*

There are quite a few ways to imitate different material properties in haptics. One way is to distribute tangential forces that relate to the surface of the geometry. Similar to the shading of bump mapping, the magnitudes of the force vectors vary depending on the current height of a simulated ridge or a valley. An additional way to create haptic textures for an object is to change the height of the surfaces with combinations of sin and square waves. Another way of creating material properties is to use friction. To create the feeling of friction, a force proportional to the normal force and in the opposite direction of the movement is added. This model is known as the Coulomb friction model. There are of course also other friction models that may be better suited for haptics. There are of course several more methods to use when creating materials and if one combines some of these methods, fairly realistic materials like sand, ice, water, plastic, wood etc can be imitated.

So far, objects have been assumed to remain static but one would also like to be able to move, rotate and scale haptic objects. Transformation in haptics can be accomplished easier than in graphics when using a haptic tool that has a single interaction point. Instead of modifying the vertices of the object, the haptic tool is transformed to the object space. This saves a lot of computation time and keeps the force calculation fast, because the octree containing the polygons of the object can still be used. If the object has been rotated or scaled the forces calculated are inconsistent with the world coordinates. Therefore when an object has been scaled or rotated the calculated force has to be re-scaled or re-rotated.

The objects have also been assumed not to be deformable. However in reality when one applies force to an object it can sometimes change its form. For example, a lump of clay deforms easily when force is applied. A way to solve this is to split polygons into six new polygons. The new vertices act as base points along with the old vertices for the deformation. When the user pushes into the polygon with the haptic tool, all points are moved a certain amount depending on where the tool is situated on the polygon. More problems arise when using deformable surfaces because vertices must be allowed to move. If there are no restrictions on how the polygons are allowed to move, a polygon can collapse on itself. Another problem is that the preprocessed octree that is used for culling can be invalid.

To get a better understanding of haptic rendering one can consult the article about haptic rendering in virtual environments [11] or the e-Touch programmer's guide [3].

2.5 Haptic API

A haptic API is an application programming interface that can be used to easily create 3D objects that can be touched. Such an API usually has some primitive shapes like boxes, spheres and cones etc, which can be created by given the appropriate parameters. They also generally have the possibility to create objects that are built up with haptic renderable polygons or triangles. This is good because it makes the transition from a graphic representation to a haptic representation easier, since the objects can be represented in a similar way. In general the APIs also have support for displaying all objects as both graphics and haptics. The graphic support can be more or less built into the API.

Three different APIs that are of interest for this project are described in the following three subchapters. In chapter 3 these APIs are evaluated to conclude which API that best suits our needs.

2.5.1 GHOST API

GHOST (General Haptics Open Software Toolkit) is a software development toolkit developed by Sensable Technologies, Inc. for creation of haptic environments and manipulation of the properties, objects or effects within them, without focusing on difficult force rendering. GHOST can be used with either one or multiple haptic devices and it supports the complete family of Phantom haptic devices. The GHOST SDK is available for Windows XP, Windows 2000, Windows NT and Red Hat Linux 7.2 platforms and supports dual processors. The toolkit is written in C++ and is object oriented and extendable through subclassing. The cost of a GHOST API license is about 2500 €.

The key concept for creating haptic environments with GHOST is the haptic scene graph. The scene graph is a hierarchical collection of nodes (see figure 2.6). The top node of the tree is always the root. After that follows the intermediate nodes (gstSeparator), which represent ways of grouping objects, scaling and orientation or adding dynamics to the subtrees. The leaves (gstSphere and gstTriPolyHaptic) of the tree represent geometry or interfaces. The haptic device is added as a leaf in the tree (gstPhantom). When the user moves the haptic device contact point into objects or effect areas, GHOST automatically calculates the correct force that the device should deliver.

Another feature of GHOST API is the possibility to use dynamic nodes to add objects that are affected by the force applied from the user. In other words, when utilizing dynamic nodes the haptic device can be used to move around haptic objects.

The possibility to set surface properties for an object in GHOST is somewhat limited, but it is possible to set damping, friction and spring constants for objects. Another way may be to expand an effect class.

GHOST does not automatically start a graphics loop and render the objects in the scene graph. If the developer wants graphics in the application he is free to use any graphics library of his choice, but GHOST does provides some classes for using GLUT and OpenGL.

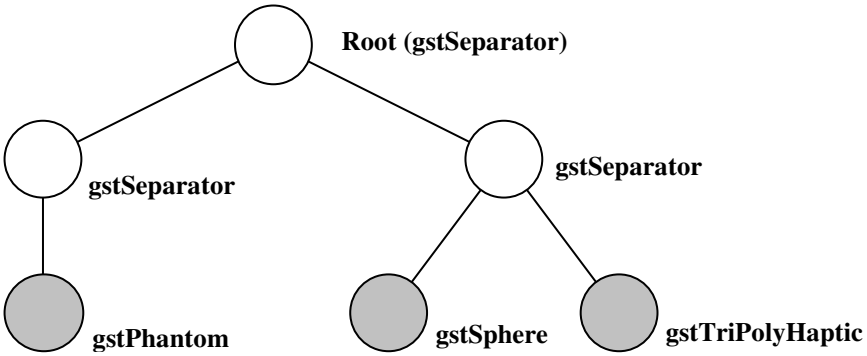

Figure 2.6 *An example of a GHOST scene graph.*

For a more in-depth guide on how to use the GHOST SDK API and a complete reference guide, see the GHOST programmer's guide [2] and the GHOST API reference [4].

2.5.2 e-Touch API

e-Touch is another API for haptic application development. It can be used to create three dimensional applications that include not only haptics, but also graphics and 3D sound. The API supports the whole family of Phantom haptic devices and the DELTA haptic device from Force Dimension. An advantage of e-Touch is that it is a free open module API. Open module is very similar to open source where the code is free, but open module provides an opportunity to make some profit for submitted expansions. Although the code is free, e-Touch utilizes GHOST to communicate with the Phantom haptic drivers and therefore GHOST needs to be installed anyway.

The e-Touch API is written in C++ and makes use of the OpenGL graphics toolkit. At the moment the API is available for Windows XP, Windows 2000 and Windows NT 4.0. It also supports dual processor symmetric multiprocessing.

e-Touch API consists of glue and modules. Glue is the foundation components for the API and the modules are built upon these. The modules provide an additional set of objects that make it easier to add functionality for application developers.

An application written using e-Touch divides application space into two parts, world space and personal space. The world space is where the touchable objects and other data are stored. The personal workspace is used for storing various controls and indicators that are specific to the application. It provides a dashboard and window system with buttons, sliders and knobs, that are controllable by touch. To switch between the two spaces, the user only has to move the haptic cursor towards himself until a wall is felt. Once the user pushes through this wall he enters the personal space. However it is not necessary to use the personal space in an application developed with the use of the e-Touch API.

The central object in an e-Touch program is the user. The user holds references to all the other objects that are vital for force and graphic rendering, like the force rendering process, the graphic rendering process, the current touch tool and the different spaces. It is also the user object that is called to start an e-Touch application. To make the

objects able to be rendered either graphically or haptically one needs to declare a subclass for each object, one for graphic rendering and one for force rendering. When rendering occurs, the current active space examines its collection of touchable objects through a touch tool class and asks each of the objects to render themselves. The rendering will then examine both the graphic rendering subclasses and the force rendering subclasses of the objects to draw and sum up the correct force respectively.

Additional information about the e-Touch API can be found at the e-Touch homepage [16] or in the e-Touch programmer's guide [3].

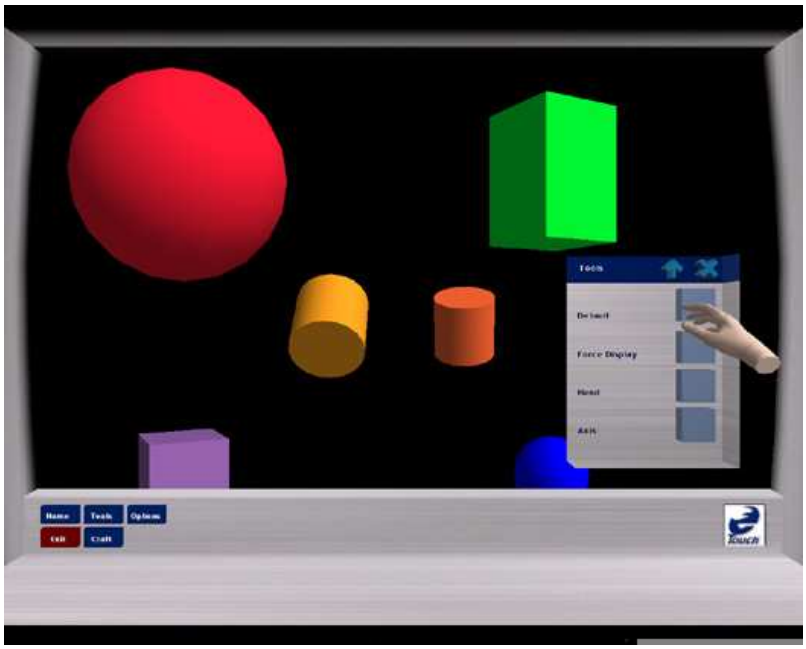

Figure 2.7 *A simple e-Touch application showing personal and world space.*

2.5.3 Reachin API

The Reachin API is yet another C++ object oriented API for developing haptic applications. It has been developed by Reachin Technologies AB. The API is available for Windows NT4 and Windows 2000 with Service Pack 5 or higher and it can take advantage of dual processors. Reachin synchronizes the graphic and haptic rendering with the use of Python, VRML and OpenGL. It supports the whole family of Phantom haptic devices, Immersion's Laprascopic Impulse engine and Magellan/SpaceMouse.

The API is quite expensive. A license that is locked to one node, which means that it can only be used on a single computer costs 11600 €. If one wants to have a floating license that can be used on several computers it gets even more expensive. A floating license costs 13980 € and a license server that is also needed costs 5790 €.

The world is built up from a VRML file with added tags for haptics. With these tags, stiffness, material, texture, dynamics, deformability etc can be added. The information read from the VRML file is translated into a scene graph with different nodes. Node is the main class that every class inherits from in Reachin. The graphic and haptic world is rendered from the scene graph.

The Reachin API is a much more complete API than both e-Touch and GHOST. It supports all the kinds of features that are available in e-Touch and GHOST, like primitive haptic objects and objects built upon polygonal models. Except for this the API has its own dynamics system and different haptic effects that can be applied. The most impressive features are however that it is possible to apply different haptic textures to objects and that the API supports deformable surfaces.

There are already some pre-defined textures that can be applied to objects but there are also possibilities to make one's own. One way to do this is to use an image as a height map over the surface. The image should be in grayscale, or else the blue color component will be used as the height information. The highest parts of the surface are represented by black color while the lowest parts are represented by white color. This means that the closer to white it gets, the lower the surface will feel like.

Black color on the surface represent a complete plane surface and the closer to white it gets the deeper into the object it will feel.

Deformable surfaces are supported in Reachin by surrounding the geometry definition in VRML with special Reachin based tag. A special dynamic stiffness can be set for this tag making it possible to get a different feedback when deforming the surface. This feature makes it easy to create applications where it is possible to carve or mold something.

The Reachin homepage [22] gives further information about the Reachin API.

3 Evaluation of haptic APIs

In this chapter an evaluation of GHOST and e-Touch will be made and a conclusion will be drawn whether e-Touch API, GHOST API or another solution will be used. Reachin API will not be evaluated because the decision has been taken that its software license costs too much to be used in this project.

3.1 GHOST

In this project we have used GHOST version 3.1. However, version 4.0 of GHOST has recently been released. The new version does not provide any speed improvements, but it does provide a new gstDeviceIO class that can be used to create non-GHOST controlled servo loops. It should also include several bug fixes and make it easier to directly access the Phantom. This version was not available at Certec so we have not been able to try if this would have been easier to use for integrating haptics in Crystal Space or give other evaluation results.

A drawback for GHOST is of course that a software license costs money. However to buy a license is not as expensive as buying a Reachin API license. A GHOST license costs 2500 € compared to a Reachin license that costs beyond 11000 €. A haptic API that has a reasonably cost or is completely free of charge would obviously be best suited for this project.

One problem with GHOST is that because it is not a free API, the source code is closed. This means that it is not possible to make changes where the haptics loop is calculated. For example, it is not possible to change the way collision detection, culling or force calculation is made. One way to introduce self-calculated forces however is to use a special force field node that is calculated in every update. This way it is possible to assign any forces at any time. Actually this is how e-Touch is originally constructed. The e-Touch driver class uses a scene with only a gstPhantom node and a gstForceField node. This is why, as mention before, e-Touch is not really a completely independent API.

There is no support for assigning different haptic materials to surfaces. However in GHOST there is support for dynamic objects. This means that one can add a dynamic object with certain properties like weight, friction etc in the scene graph and then move them around with the haptic device. GHOST makes all the necessary dynamics calculations to get a realistic feeling of moving the objects. The dynamics calculation seemed to work pretty good when we tested it. However there is a dynamics engine plugin in Crystal Space that would be nice to utilize for the dynamics calculations. It makes more sense to use this to update the haptic position than to update all dynamic objects in Crystal Space with the positions calculated from GHOST. This way also gives a more general solution with haptics and dynamics detached from each other.

A problem we noticed with GHOST happened when we moved objects that were built up with polygons, i.e. gstTriPolyHaptic nodes. Although moving standard primitive objects like spheres and boxes worked fine. The problem that occurred was that when a

gstTriPolyHaptic object was moved towards the tool position of the Phantom, no collision and force response took place. For example, we tried building a cube from twelve triangles and created a gstTriPolyHaptic node from it. A similar scene graph was built but instead of using a gstTriPolyHaptic node a gstCube node was used. If gstCube was used and the position of the cube was updated, it was possible to feel all six sides perfectly. If gstTriPolyHaptic was used instead one could not feel the side of the cube that was moving towards the Phantom (see figure 3.1). This is obviously some kind of synchronization problem in GHOST SDK that we could not change. The support at Sensable was contacted but they had no solution to the problem. Maybe there is some way to work around this, but it must be seen as a fairly big disadvantage.

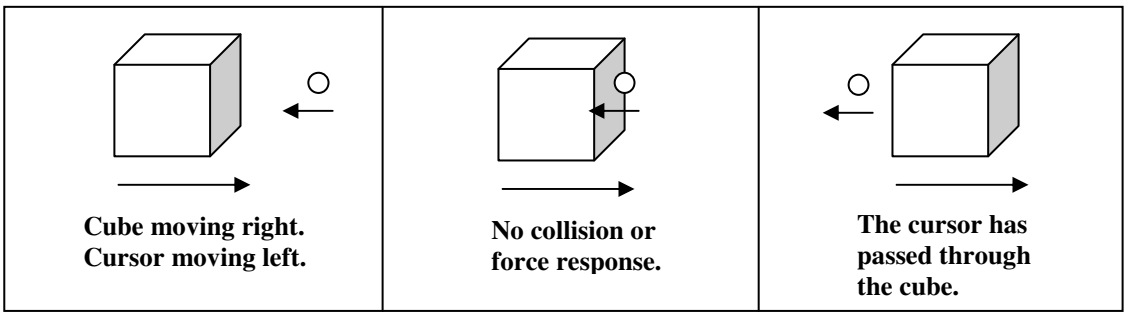

Figure 3.1 *No collision or force response when using gstTriPolyHaptic to represent the cube that moves towards the Phantom cursor.*

3.2 e-Touch

One of the main problems that we observed in e-Touch was the difficulty to disconnect the haptics part, which we wanted to take advantage of, from the graphics part. As mentioned before the central object in e-Touch is the user class. This class starts both a graphic manager part and a haptic manager part. But if one only wants to use the haptics part it is not just to remove the start of a graphic manager. Some graphics calls were buried deep down in the code so it took a lot of time to understand and remove this. Hopefully the graphics part will be better disconnected from the haptics part in future releases of e-Touch.

Another issue that can be said to be of concern is that e-Touch is still in the beta stadium. The current release used in this project is version 1.0.0 beta 3. When looking through the code one notices a few hacks and comments at certain places. This means that everything may not yet have been thoroughly tested and structured in the best way.

As it stands now e-Touch currently works on Windows NT, Windows 2000 and Windows XP platforms. GHOST on the other hand also supports the Red Hat Linux platform. This is kind of a disadvantage for e-Touch because Crystal Space also supports the Linux platform and the plugin could then perhaps have been made to support more platforms.

There is very sparse documentation about e-Touch. There is a programmer's guide that describes how to install and build e-Touch. It also describes the open module concept and how haptic rendering works. But there are no good programming examples or any good explanation how the classes are connected. To just get a perception of this one has to examine the source code, which is somewhat annoying. There is also a reference

manual that should describe the different methods and classes, but this is actually just a reference built from the class comments in the code. These comments are also very simple and sometimes during our evaluation, parameters to methods had to be assumed and tested.

However, if one gets an overview of the API there is a big gain because when one finds how everything sticks together, one can make changes to the servo loop and the force calculations. In GHOST it is impossible to change how objects or polygons are culled or how the forces for them are calculated. Using e-Touch one has the possibility to pick out and use only the interesting haptics parts. As mentioned before, these parts could have been detached from the other parts of the API to a greater extent.

One of the key advantages of e-Touch is that its algorithm for calculating forces from polygonal objects is faster than the one used in GHOST. In an article about the active-polygon polygonal algorithm used in e-Touch and presented in 2001 [6], Tom Anderson and Nick Brown show measurements of the haptic load peak and the haptic load average when using e-Touch versus GHOST. The haptic load program that comes with GHOST 3.1 was used for the measurements. This program measures how high the load is on the process that handles the haptic calculation. Four different objects were used in the test. They had different numbers of polygons and some had more complex topology. For all the objects the active-polygon polygonal algorithm had both lesser average haptic load and lesser peak haptic load. Even if this measurement is done with GHOST 3.1, the new version 4.0 has not changed anything in its haptic rendering algorithm. However, the preprocessing load times were almost always longer when e-Touch was used. This is the price one has to pay for getting a lower haptic load.

A difficulty we noticed with e-Touch was that when one built objects with polygons that did not share vertices, but overlapped each other like in the case of a corner of a wall with thickness, it was possible to feel through the small opening between the walls (See figure 3.2). This problem was not noticed when using GHOST. Although in some cases this could be useful, for example in a dynamic environment where a box stands on a floor and the user wants to be able to put the haptic tool under it and lift it. A solution to getting walls with thickness to work is to move the vertices so they meet at the inner and outer corners. This way it is not possible to push through the walls. On the other hand it could be time consuming to edit all objects that have the same kind of problem. The problem lies within the active-polygon polygonal algorithm and by somehow changing how the polygonal haptic objects are built or rendered one might overcome this problem. However this probably means major changes to the API.

Figure 3.2 *Wall corners seen from above. Problems when vertices are not shared.*

e-Touch has no support for dynamic objects and no support for applying different materials to surfaces. If one wants these properties they have to be implemented in the computePointForce method of the object force renderer. Some examples of this are

demonstrated in a few demos that come with e-Touch. One demo shows dynamics by using balls attached to rubber strings and another one demonstrates different textures. But one should have in mind that all physics and texture calculations have nothing to do with the API and have been implemented especially for the demo applications.

3.3 Conclusions

Due to some decisive issues it was decided that we were going to use parts of the e-Touch API for our solution. Two major factors were the main reasons. One was that e-Touch was open module and when using e-Touch one had the ability to pick out important parts, change parts and see how things were implemented. We noticed that not all parts were of interest and we saw that it would probably work to pick out classes, make some changes to them and get them to work together with our implementation.

The other main reason was the fact that the haptic polygon algorithm in e-Touch was faster. Knowing that we had a fast haptic rendering algorithm, we knew that we would be able to use objects with more polygons and possibly utilize more graphic effects in Crystal Space, without the servo loop taking to much time.

This choice of solution gave us a way to avoid having to implement the haptic rendering algorithm ourselves but we would still be able to add or modify code in it.

4 Game engines

4.1 Introduction to game engines

When creating a 3D game one needs software that handles the virtual environment and renders it to the screen. There is also need for software that handles for example physics and collision detection. These tasks are handled by the 3D game engine. One can say that the game engine is what powers the game. It is a platform upon which the game is built and it provides functionality that is common to all games, such as the things mentioned above and also the access to various hardware devices, for example keyboard, mouse, joystick, graphics accelerator, network card and so on.

The advantage of a 3D game engine is that it is not necessary to create a new game engine for every new game. Instead a developer can reuse the same game engine over and over again. This saves the game developer from a lot of work when creating new games, but also makes it easier to enhance or add features to the game engine itself. The game engine performs a lot of time consuming tasks, so every enhancement that can be made to the engine may provide for new possibilities for the game developer.

4.2 Crystal Space

The game engine that is used in this project is called Crystal Space. It is a free 3D game development kit written in C++ and it is being developed as an open source project. Crystal Space has most of the features that are needed in a 3D game engine, but it is still under development, which means that it is constantly being improved with new features. Since it is an open source project with hundreds of people working on it all around the world the documentation of the system sometimes is worse than desirable which can make Crystal Space a quite difficult system to use. However it is the open source development approach that makes Crystal Space worth looking at in the first place. Crystal Space falls under the GNU copyleft license for libraries, which means that anyone is allowed to use it in their products and even to sell their products using Crystal Space provided that Crystal Space itself remains free. This approach makes the Crystal Space development kit desirable to game developers because it is free of charge and since it is open source they can even make changes in the game engine itself to customize it to their own game.

As mentioned above Crystal Space has many features. For example, it supports six degrees of freedom, colored lighting, mip-mapping, sectors and portals, mirrors, procedural textures, particle systems, OpenGL rendering, collision detection and physics, and it also has a flexible plugin system.

The remains of this chapter will provide an explanation to how the Crystal Space game engine functions. The default main loop is explained and also how it controls the execution using the event system. How a 3D environment is built up using meshes and also how this environment can be rendered to the screen are other topics that will be given an explanation here. There is also an explanation of the collision detection system

as well as the dynamics system that are used in Crystal Space since they are important to our project. To get a more comprehensive knowledge about Crystal Space one should examine the user's manual [1]. The game engine is updated fairly often so it can also be useful to check out the homepage [13] for the latest news. There is also a Crystal Space developer homepage [14] with a forum and mailing lists.

4.2.1 The main loop

Crystal space is an event driven game engine, which means that it uses events in the communication between different parts of the engine to inform when something has happened. Each part of the game engine listens to events that are important for that part and then takes the appropriate actions to deal with the event. A Crystal Space application is built around a default main loop that controls the event handling. The loop is started when the application has been set up and is then active during the entire execution of the application. In a graphics application, an iteration of the loop would typically result in a new frame that is rendered to the screen.

During the iteration, any event that has occurred since the last iteration can be taken care of. The application can register an event handler to the game engine, allowing the application to communicate with different parts of the game engine through events that are posted to the system event queue. For example, if the user decides to shut down the application using a keyboard command the application will be notified of that through an event in the event queue. This provides an opportunity for the application to take the appropriate actions and shut itself down in a safe way, but it can also broadcast a termination event to the other parts of the game engine thus providing for a safe exit of the entire system.

The graphic rendering is also controlled by the default run loop. In every iteration, the loop broadcasts events to all parts of the game engine and to the application that initializes the rendering of the frame. When the application is notified of this it calculates any camera movements or changes in the environment that has occurred since the last frame and then tells the game engine what should be rendered. The game engine then processes the information and finally renders the picture to the screen.

4.2.2 The 3D world

In 3D games an important part is of course the 3D world in which the game takes place. In Crystal Space all the geometry in a world is built up by mesh objects. A mesh is a collection of convex polygons that are defined by their vertices. To create a mesh object it is therefore needed that all vertices of the object are defined and assigned to convex polygons to create the surface of the object. Only one side of a polygon is visible and in Crystal Space the vertices of the polygon are oriented clockwise. The other side is culled away by the 3D renderer to minimize overdraw. Except for using polygons as building blocks of meshes, Crystal Space also supports Beziér curves to represent curved surfaces. There are a few different types of mesh objects that are suitable for different purposes. Most important are the thing and the sprite mesh object types which are most suitable for static and dynamic objects respectively.

The mesh objects are only the visible content of the world. The world itself consists of larger building blocks called sectors. A sector is basically a region of space that functions as a container for mesh objects. If the designer of the world does not define

sectors manually Crystal Space will place all geometry in a single large sector. Using more than one sector can often enhance the performance of the world since each sector is handled separately from the others by the renderer. For example, if the world consists of a building with several rooms in it one probably will not be able to see parts of all rooms from where one is standing. Using different sectors for every room can cull away entire rooms from the rendering process making a noticeable performance difference. If only one sector was used all objects in the entire building would have to be considered by the renderer (See figure 4.1).

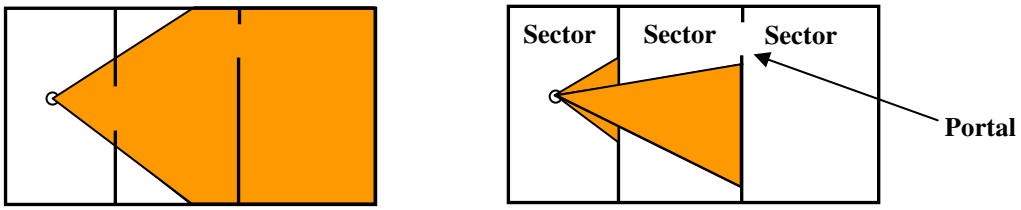

Figure 4.1 *Using sectors can cull away lots of objects from being rendered. Even entire rooms can be culled away.*

When several sectors are used they will have to be attached to each other. For this purpose Crystal Space uses portals, which functions as gateways between different sectors. A portal can be assigned to a polygon of a mesh object and directed to any sector in the world. Normally a portal connects two sectors that are located next to each other, but they can also point to sectors somewhere else in the world creating some kind of teleportation effect. They can even point back to the same sector creating a mirror. When a sector is rendered and the renderer comes across a portal the game engine makes a recursive call to render the sector seen behind the portal. Rendering portals will be explained in more detail in the next section.

4.2.3 Rendering

When the information about what should be rendered has been collected from the application, the 3D renderer has to perform some important tasks before the picture is shown on screen. As mentioned above only one sector is handled at the time, so if the current sector contains no portals these tasks are done only once.

First of all the vertices of all the objects in the current sector are transformed from world coordinates to camera coordinates. The game engine then loops over all polygons of all the objects contained within the sector and does the following for each polygon. The back-face culling algorithm is run on the polygon to find out if it has its front turned against the camera or if it can be discarded. If it faces the camera the polygon is clipped against the view plane and also against the view polygon which is a 2D polygon defining the scope of the camera. Clipping against the view plane removes polygons that are behind the camera. A texture is mapped onto a visible polygon and the resulting 2D polygon is then drawn onto the screen.

If the polygon should turn out to be a portal instead of a regular polygon it is not drawn onto the screen. Instead the sector behind it should be drawn, so a recursive call to the routine explained above is made using the portal polygon as the new view polygon. The sector behind the portal is then drawn in the same way as the first sector. By using the portal as the view polygon everything that is not visible through the portal will be culled away.

4.2.4 Collision detection and dynamics

Crystal Space provides collision detection based on the Robust and Accurate Polygon Interference Detection algorithm (RAPID). This algorithm is applicable to all types of polygonal models and thus, can handle objects of all shapes and sizes. RAPID represents an object as a tree structure, which is created by dividing the object into smaller and smaller boxes. The tree structure is used to speed up the search for colliding objects. There is also another collision detection system in Crystal Space that can be used called OPCODE. However this has not been utilized in our applications.

When collision detection is made, two objects at a time are tested against each other. This test is made by traversing the trees for both objects, searching for overlapping boxes. If there are one or more overlaps, there is a collision between the objects and we also know which parts of the objects that collide. Crystal Space saves the colliding polygons in a list that can then be used by the application to determine what should happen to the two objects after they have collided.

To do collision detection in Crystal Space a certain collider object can be created for each mesh object that can possibly be involved in a collision. The collision test is then initiated from the application by sending two colliders at a time as parameters to a method in the collide system. Testing two objects that are far apart for collision is not needed and since the RAPID algorithm does not cull away such objects before traversing their trees, it is up to the Crystal Space application to cull away them from the collision detection to minimize the calculations. For example, this can be done by first testing the bounding spheres of the objects for overlapping. If the bounding spheres do not overlap then the objects themselves cannot possibly collide and therefore collision detection is unnecessary.

Another way to get collision detection is to use the dynamics plugin. This plugin is based on ODE (Open Dynamics Engine) [17], which is a free library for simulation of rigid body dynamics. A big advantage with the dynamics system is that it takes care of both the collision detection and the collision response. The collision response is what should happen with the objects after they collide, for example if they should stop, bounce or perhaps rotate. Rigid body objects are created for the geometry and are then added to the dynamics system. The bodies can be made either static or dynamic. This makes it possible for the dynamics engine only to do force calculations on the objects that can actually move. The objects that are made dynamic are included in the calculations that are done by the dynamics system every time it is updated from the application. If an object is affected by gravity or involved in a collision with another object a force is calculated by the dynamics system and then applied to the object. A new position for the object is then calculated and the object is moved. Friction, density, elasticity and a mass should be assigned to every dynamic object for realism. As mentioned earlier the objects may be affected by gravity. The gravitational force is set in the dynamics system and is then applied to all dynamic bodies within that system, causing them to fall down.

The dynamics system is so far quite limited because one can only add objects that are shaped like boxes, spheres, cylinders and planes. This makes it hard to make it useful in a dynamics simulation where objects of other shapes are used. Hopefully this will change in a not too distant future.

4.3 Map creation and Valve Hammer Editor

To create 3D worlds for computer applications some kind of tool is needed. In our project we use the Valve Hammer Editor, formerly known as Worldcraft. Valve Hammer Editor is intended to be used for creating maps for Half-Life, but since Crystal Space comes with a program that converts maps from the Valve Hammer Editor map format to the XML format that is used by Crystal Space it is very suitable to use this editor. Creating a world without using some kind of map editor would be almost impossible. Constructing a very simple map consisting of only a few objects would of course be possible, but it requires defining each vertex and each polygon by hand. A large world containing a lot of objects is simply too complex to be created without a graphic editor.

With Valve Hammer Editor it is possible to create a lot of different objects. Of course there is support for basic geometry like cubes and cylinders, but one can also combine these to make complex objects. There are also possibilities to manipulate vertices and edges if the basic geometries are not enough. Use of predefined objects is also supported and since there are a lot of people creating maps for different games there are a lot of objects available to download from the Internet.

Apart from ordinary objects Valve Hammer Editor also supports entities. Entities can be things that are placed within the map, but have no geometry or mass. This could for example be a light, a sound or a starting point for the player. An entity can also be assigned to an existing object making it possible to attach certain information to that object. The information is saved in the map file and can then be retrieved by either the game engine to correctly render the object or by the application to determine what should happen when the user encounters the object.

As mentioned above, Crystal Space does not use the same map format as is used by Valve Hammer Editor. Therefore a small program named map2xml is needed to make the conversion of the map from the Valve Hammer Editor format to the Crystal Space XML format. When the map is used by an application for the first time, or when the map has been changed, the "-relight" option must be used on the command line to recalculate the light maps.

When using Valve Hammer Editor we found that there was a problem when using the vertex manipulation tool. Sometimes the objects created with the use of vertex manipulation simply disappeared when the map was saved. It seems that the problem lies within Valve Hammer Editor and not in the conversion between the different map formats. In any case one should be aware of the problem when using the vertex manipulation tool.

When creating a world that should be used together with the haptic system there are a few things one has to think about when creating the map. Appendix C contains more information about these issues and also a guide on how to use Valve Hammer Editor to create maps that work well together with the haptics.

5 Design

When we started to look at how to integrate haptics with the game engine, we decided quite fast that a plugin to Crystal Space was the best way to do it. Implementing the haptics part as a plugin divides up the functionality in a very nice way so that the haptics does not affect other functionalities in the game engine.

This chapter is a detailed description of the design of the plugin, but it also presents the problems that we encountered in the development stage and how we chose to solve them.

5.1 Task

The intention with the plugin is that it should provide a haptic representation of the geometry contained within the Crystal Space engine making it possible to feel the environment by using the Phantom haptic device. It should also be able to handle movement of the camera and changes in the environment such as movements of dynamic objects. When the camera is moved in the application the Phantom should give the sensation of moving with the camera. The Phantom workspace can be thought of as being attached to the camera and oriented in the same direction. When an object is moved in the application, for example due to gravity, and collides with the Phantom pointer, the movement should be felt through the Phantom. This should also work in the other direction, that is, if the user presses the Phantom pointer against an object that is a dynamic object, the object should start to move if the force is sufficient. Furthermore there should be a possibility to set different haptic properties for objects. By modifying these properties the user will be able to feel a variety of sensations.

5.2 Problems and solutions

5.2.1 Representing the touchable world

One of the first big problems that had to be solved during the development of the plugin was how to represent and store the haptic objects that build up the touchable world. Since Crystal Space has no support for haptic devices the mesh objects used in the game engine are not designed to be used to represent touchable objects. The e-Touch API however, uses mesh objects that contain their own algorithm for calculating forces, which is exactly what we need for our plugin. The e-Touch classes contained a lot of code that we had no use for in our plugin. Among other things, it contained code for showing the objects graphically in an e-Touch environment, but since we are using the game engine for graphics, we decided to remove all such unnecessary code. The concept of using a special force renderer class for each object was also removed and the force algorithm was moved into the object class itself.

Our solution to the problem of how to represent the touchable world was to create a world consisting of touchable objects that we kept inside the plugin and then synchronize this with the world used for the graphic representation by the game engine. Using this approach, all mesh objects loaded into the game engine have to be copied.

This means that a new haptic object has to be created for each object in Crystal Space. Since Crystal Space supports mesh objects constructed from polygons with more than three sides and the mesh objects we wished to use in our plugin only uses triangles, it was not as easy as we thought. To solve this problem we had to create the haptic objects from scratch, which means that all vertices had to be copied and then each polygon had to be converted into one or more triangles and added to the new haptic object.

So far we have only solved the problem of creating the objects. The other problem still remains, that is, how to store these object inside the plugin. The haptic calculations need to be very fast because the Phantom device has to be updated once every millisecond. Since the forces are calculated inside the haptic objects, a way to cull away objects that are far from being touched by the Phantom, to reduce the amount of calculations, would be a good thing.

To store the objects we decided to use an ordinary vector which is very easy to use and which allows for random access by indexing. To get an efficient culling of objects we decided to use an octree to store indices into this vector. An octree is a space partitioning structure used for efficient culling of objects. The general structure of an octree is described in chapter 2.4. However the content of the octree described in chapter 2.4 differs from the content of the octree used for culling away haptic objects from the force calculations. As mentioned earlier this octree contains indices into the vector holding the haptic objects. It therefore has to span the entire haptic world. The leaf nodes of the tree are either empty or they contain a list with one or more indices. The reason that we use indices to represent a certain haptic object in the octree is that each object can span several different tree nodes. Having the tree containing the actual mesh objects could mean that an object would have to be copied several times, which in turn would result in a huge waste of memory space. By traversing the tree, checking the position of the Phantom against the span of the nodes, the objects that are close to the Phantom cursor can easily and efficiently be found.

A problem that we encountered when we had first implemented our octree was that a very large object had to be represented in a lot of tree nodes that actually did not contain the object. This problem occurs because we check the bounding boxes of the objects against the bounds of the tree node to find out which objects that overlap with the node. The problem is easiest to understand when considering the outer walls of a building containing the entire world. If the walls are created as a single object the bounding box of this object will overlap with every node in the octree, even though the walls are actually only contained within the outermost octree nodes. The reason that this is a problem is because when this happen, no tree nodes will become empty and therefore the tree will grow to its maximum size. This means that the tree becomes less efficient and that is against the whole idea of using an octree.

To solve this problem we decided to give the designer of the world the opportunity to mark large object by assigning a certain name to them. The objects that are marked are then left out of the octree. Using this approach they are never culled away from the force calculations, but since they would have been represented in every node of the tree, they would not have been culled away anyway.

The optimal depth of the tree of course depends on the amount of objects within the world. A world containing a lot of objects needs a larger tree than a world containing only a few objects. However a maximum depth of five or six will probably be appropriate for most worlds. The biggest problem with creating a tree that is very deep is that the time to create the tree grows rapidly with each new level. This means that the pre-processing time will grow very fast and that the amount of memory acquired becomes much larger.

5.2.2 Static and dynamic objects

There are two different kinds of objects that build up a haptic 3D environment. Those are static and dynamic objects. The static objects are those that always stay in the same position. Static objects could be walls of buildings, stairs or furniture for example. Since the static objects never move they are relatively easy to take care of. Dynamic objects however are objects that can be moved in some way. A dynamic object could be a ball or a car for example. When using dynamic objects, an application will not be as fast as if only static objects were used. Moving an object involves a lot of calculations and it is not as easy to cull away dynamic objects from the force calculations.

In the beginning of the project we thought that adding support for dynamic objects to the plugin would be a too difficult and time consuming task for us to do, so we were prepared to create a plugin that would only be able to handle static objects. Some time into the implementation of the plugin though, we found that this probably was not such a difficult task after all, so we decided to add support for dynamic objects. Being able to use dynamic objects allows for the creation of much more realistic worlds and provides the developer with lots of possibilities when creating applications. Another idea is that the plugin should be able to be used with the dynamics system plugin that comes with Crystal Space.

When an object is moved in a Crystal Space application its transform is automatically updated. The transform of an object holds information about how much the object has been moved, rotated and scaled from its original state. This information is the same for the corresponding haptic object so it does not have to keep track of its own transform. When computing the forces from an object the Phantom position is transformed to the local coordinate system of the object. This makes it possible to move objects in the application and to feel the movements through the Phantom device. For example, this can be used by a dynamics system simulating gravity. If an object is affected by gravity it will fall to the ground if nothing holds it up. When touching the object with the Phantom one can actually feel the object falling.

But there is also another possibility of moving objects, and that is to push on them using the Phantom. This works in the opposite direction, that is, if one pushes an object one wants to see it move on the screen. To solve this task we used the fact that the dynamics system in Crystal Space uses forces to move an object. In other words, when a force is applied to an object and the force is large enough, the object moves in the direction of the force. This fits very well with the Phantom device since the force with which the user pushes on an object can be fetched from the device. By fetching this force and applying it to the corresponding object in the dynamics system, the user can see the object move as he pushes on it. But it is not enough to see the object move. One would also like to feel it move as one pushes on it. This however, takes care of

itself since an object that is moved in Crystal Space can be felt moving with the Phantom, as described in the previous paragraph.

We do not use any algorithm to cull away dynamic objects from the force calculations. Storing them in an octree as with the static objects would result in the octree having to be updated every time an object is moved, which would be quite inefficient. During the construction of the haptic world the plugin checks if the objects should be considered as movable and if so it does not add them to the octree. Instead these objects are kept in a vector and are considered for collision every time the force calculations are made. Objects that are considered to be movable by the plugin are movable things and sprites.

5.2.3 Synchronizing graphics and haptics

Because of the method we used to represent the haptic world separately from the graphic world, we faced the problem of having two different sets of meshes that needed to be synchronized. The plugin would not be very useful if what the Phantom device is touching is not what is shown on the screen.

Chapter 5.2.2 described how the objects can be moved by the application and by the Phantom respectively and how these movements were communicated between the two parts. But there are other things except for the objects that can move and the most important is the camera. We wanted to be able to create a world were we could walk around, touching things that were located in different parts of the world. This means that the position of the Phantom workspace had to follow the camera position in some way.

The plugin uses a small GHOST scene graph, where the Phantom node is located under a gstSeparator node (see figure 5.5). This separator node can be used to translate and to rotate the workspace of the Phantom. The Phantom position is relative to its workspace, so when moving the location of the workspace, the position of the Phantom follows. This behavior is what we took advantage of when creating the plugin. By setting the workspace of the Phantom to follow the camera movements the developer can build the application so that when the user steers the camera, for example by using the keyboard, he also steers the Phantom in the same way.

5.2.4 Calculating forces

The force calculations are performed once every servo loop and are initiated by a call from the servo loop to a callback function that is implemented in the plugin. The first thing the plugin does when this call comes is to find all static objects that should be considered for exact collision detection. The position of the Phantom stylus is passed as an argument to the callback function and it is this point that is used when traversing the octree. If the leaf that contains the point is empty there are no nearby static objects, except if the developer has tagged certain objects like outer walls as large objects, to do collision detection on. If the leaf on the other hand contains a list of indices, this list is used to get the haptic objects and to invoke their force calculation algorithm. This algorithm performs the exact collision detection, and if there is a collision, calculates the normal force, which is proportional to the penetration depth of the cursor. It also depends on the stiffness coefficient that can be set for the haptic objects. The force is then applied to the Phantom, creating the illusion that the user is touching the surface

of the object. If the point should collide with more than one object, the forces from all these objects are added.

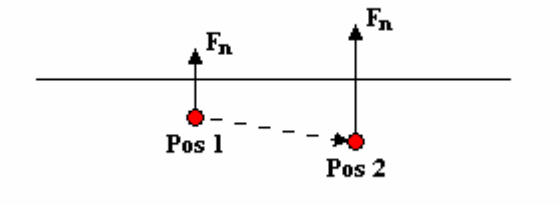

Figure 5.1 *The normal force is proportional to the penetration depth.*

The procedure for calculating forces for dynamic objects is a little different from the procedure mentioned above. The main difference is, as mentioned before, that the dynamic objects are not contained within an octree. Instead all dynamic objects are considered for the exact collision detection. Since the dynamic objects move around, the Phantom position has to be transformed into the object's local coordinate system before the force calculation algorithm can be applied. This is done by using the transformation matrix contained within the Crystal Space representation of the object.

Finally the force from the dynamic objects is added to the force from the static objects and the resulting force is then sent to the Phantom device. The force sent to the Phantom can only be represented as a single vector so it is important to have summed up the forces acting on the stylus to a single force before delivering it to the device.

5.2.5 Friction

If the forces are calculated without taking friction forces into account the objects become very slippery, making it difficult to feel the shape of the object. Friction is also very important when trying to create a realistic feel to an object. Because of this we decided to make an attempt to add friction to our force calculations.

Friction is a force acting between two bodies opposing their relative movement. Usually, we distinguish between static friction and kinetic friction. Static friction is the frictional force working on a body at rest, opposing setting it into motion, whereas kinetic friction is the frictional force working on a body that is already in motion. In general, static friction is larger than kinetic friction [18]. As seen in figure 5.2, as the applied force increases, the friction grows until it reaches its maximum value. In this area the object remains stuck in its original position. As the applied force grows larger than the maximum value the object slips and starts to move. The friction now decreases until it reaches the kinetic friction value and the object continues to slide.

The friction force is proportional to the normal force, which is the force that presses the two objects together. The equation is written:

$$F_f = \mu \cdot N \tag{5.1}$$

where μ is the friction coefficient and N is the normal force [8]. Keeping the normal force constant and increasing the applied force will cause the friction to change in the way shown in figure 5.2.

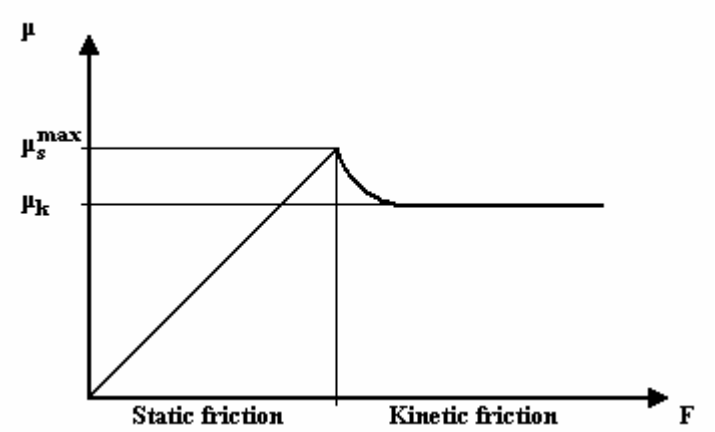

Figure 5.2 *The friction (μ) varies with the applied force. When the static friction reaches its largest value (μ_s^{max}) the object starts to move. At this point the friction will normally decrease to a constant value (μ_k).*

The friction model that we chose for the plugin can be explained with two points connected by a spring (see figure 5.3). The first point is the stick-point, which is the point on the surface of the object where the Phantom cursor first touched down. The other point is the position of the Phantom cursor. Depending on the distance between the two points the model has three different states [9].

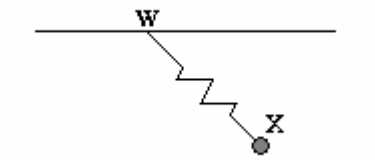

Figure 5.3 *The virtual spring that connects the stick point (W) and the position of the Phantom (X).*

The first state is when the cursor moves within a very small radius of the stick-point. In this case the friction force will be zero. This state is needed because there are small vibrations in the position of the Phantom cursor, due to small involuntary hand movements. When the distance between the two points exceeds a certain value the force is set proportional to the distance. In this state the model acts like a spring, pulling the Phantom towards the stick-point. The sensation is that the Phantom is stuck in one position. Stretching the spring beyond its maximum will cause the transition into the sliding state. In this state the stick-point is moved along with the cursor and a constant friction force is applied.

Since the cursor is not restricted to the surface of the object it touches, but actually moves around just under the surface, we use the surface contact point when calculating the friction and not the actual position of the cursor. The surface contact point is the point, on the surface, that is closest to the cursor. By doing this we assure that the friction force is always a tangential force.

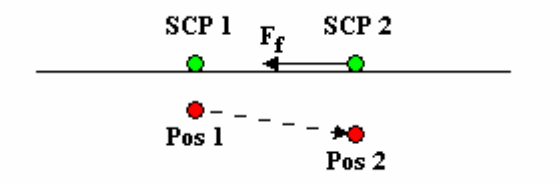

Figure 5.4 *The friction force is calculated by the use of the surface contact point (SCP) and not the actual cursor position. This is to make sure that the friction force is parallel to the surface of the object.*

As mentioned above the friction force should be proportional to the normal force and thus the resulting friction force is scaled using the normal force calculated from the penetration depth.

5.2.6 Viscous objects

We wanted to try to implement some other type of object than the solid objects that were part of the e-Touch API. We chose to implement support for liquid objects in the plugin. When moving around in water for example, the movement is damped due to the viscosity of the fluid. The damping force is proportional to the velocity of the object, so implementing this effect seemed to be a problem that was not to difficult to solve.

We decided to implement the force calculations in a way similar to the sliding part of the friction calculation. That is, by using a point sliding along behind the cursor. The difference is that with friction the distance between the two points is kept constant to ensure a constant friction force. In this case however, the distance continues to grow as the velocity of the cursor increases, causing the damping force to grow with it.

A solid object and a liquid have so different characteristics that we decided to create a new kind of object to represent liquids. The object is created as a triangle mesh but instead of pushing the Phantom out of the object when it penetrates the surface, a smaller force is applied opposite the direction of movement, making it possible to move around inside the entire liquid object.

We decided to reserve the Crystal Space name "liquid" for this type of objects. This means that an object named "liquid" is recognized by the plugin and a liquid object is created. To be able to create liquids with different characteristics it is also possible to decide the viscosity of the object. Read more about this in appendix C.

5.2.7 Basic geometry objects

Optimizing performance is a key issue when working with haptics, so the use of mesh objects is not really the best choice for objects with basic geometry such as boxes, spheres and cylinders that can be represented by their sizes, instead of by a mesh of triangles. A box can be created from the length of its sides, a sphere by its radius and a cylinder by height and radius. Such an object is easier to work with than a triangle mesh and the force calculations are easier and faster than for a mesh object.

To take advantage of this fact, we decided to take the box, sphere and cylinder objects from e-Touch and rebuild them so that they would fit into our plugin, as we did with the triangle meshes in the beginning of the project. When this was done we found that there were some difficulties in creating the objects in the correct position in the world. The problem occurred because the objects loaded from a map-file have their vertices in world coordinates, meaning that they all have their origin in the origin of the world. We decided that we would keep having the world coordinates for static objects and use the object's transform from Crystal Space for dynamic objects. Because of this the basic geometry objects can only be used for dynamic objects.

5.2.8 Different coordinate systems

A problem that we struggled with during a big part of the development process was that Crystal Space, e-Touch and GHOST does not use the same coordinate system. The problem is that Crystal Space and GHOST uses a right-handed coordinate system while e-Touch uses a left-handed coordinate system. Because of this the haptic world had to be built as a mirror of the graphic world. To solve this we simply decided to change the sign of the z-value on all coordinates and force vectors that were communicated between parts that used different coordinate systems. This does not affect the way the world has to be constructed.

The problem however, became greater when we decided to add support for moving objects. Moving them in the right direction was not very hard but getting them to rotate correctly took quite some time to figure out. After a lot of testing, we eventually found the correct series of transformations that had to be made to get the Phantom position correctly transformed into the local coordinate system of the object and the calculated forces correctly transformed back to the world coordinate system.

5.3 Program structure

5.3.1 GHOST and e-Touch parts used

After the evaluation of GHOST and e-Touch we decided only to use a small part of their functionality and to implement the rest ourselves. For example, the part of the plugin that keeps track of all the haptic objects and the part that manages the force collection from the objects are not taken from the API's.

As in e-Touch, we use a small GHOST scene graph to be able to communicate with the Phantom device. The scene graph contains a gstPhantom node, which is needed to be able to retrieve the status of the Phantom and also to be able to change its status. In our case it is the position of the stylus that is retrieved from the Phantom device and it is the position of the Phantom workspace that needs to be changed when the camera moves around in the 3D environment. There is also need for a gstForceField in the scene graph. The force field is needed to be able to send forces directly to the Phantom because GHOST is not used for the haptic calculations.

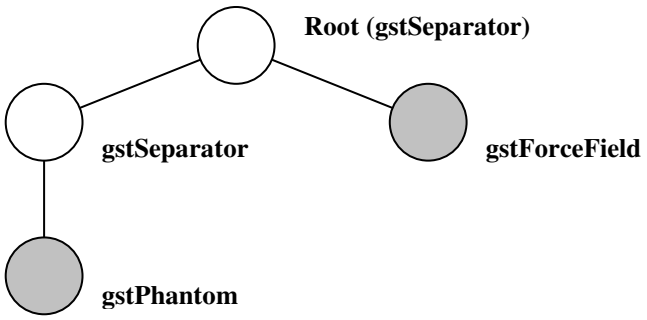

Figure 5.5 *The small GHOST scene graph used in the plugin to communicate with the Phantom.*

Other things that we also use from e-Touch are modified versions of the haptic objects classes. These classes are used to represent the objects and they contain the force calculation algorithms. To make these classes fit into the rest of our plugin we

examined them thoroughly. We then removed as much code as possible that was not needed for the force algorithm to work and made a few changes to create a more suitable interface. There are also other parts of the e-Touch API that are used but they have not been modified in any way.

5.3.2 Classes and structs

With this section we wish to show how we have divided the functionality of the plugin into different classes. Here is a small description of each class and a UML class-diagram to show how the different classes are connected to each other. The e-Touch classes used in our implementation have all been modified in some way. However to get an overview of their original functionality one can also have a look in the e-Touch reference manual [5].

iPhantom

The iPhantom struct serves as the interface of the plugin to the Crystal Space application. To fit into the Crystal Space Shared Class Facility (SCF), this interface is derived from the iBase class. Since it is an interface, iPhantom contains no member function implementations and no constructor and thus, cannot be instanciated. The functions declared by this interface are implemented in the csPhantom class.

csPhantom

This class holds the implementations of the functions declared in the iPhantom interface. It is through these functions that the application communicates with the plugin. For example, the application can set the list of mesh objects that should be represented as haptic objects by the plugin. It can also start and stop the servo loop. When the servo loop is up and running there are functions that can be used to fetch the position of the Phantom stylus and to change the position and orientation of the Phantom workspace according to the camera movements in the application.

World

The World class can be seen as a container of all the haptic objects. It is responsible for creating and keeping a haptic representation of the 3D environment used by the application. To keep track of the static objects contained in the world, an instance of the Octree class is used. World also collects the forces from the haptic objects.

Octree

The octree is used for culling away objects that are not in the proximity of the Phantom, thus providing a performance increase and making it possible for the plugin to handle a larger amount of objects. The Octree object is responsible for storing the root of the tree and providing access to the tree. The octree itself actually consists of Node objects.

Node

The node objects are what build up the octree. Every node that is not a leaf of the tree has eight child-nodes. Each node represents a certain volume of space inside the world and the children of the node each represent an eighth of this volume. A leaf of the tree is either empty or it contains a list with one or more indices representing haptic objects. An object is added to the list if its bounding box overlaps with the node.

etObject
The etObject is the base class for all haptic objects in e-Touch. To create a new kind of haptic object this object should be used as the super class. The class has been slightly modified in our plugin compared to the e-Touch implementation. Every etObject has a certain stiffness. This has been utilized in the plugin along with an added property for friction. All graphics properties have been removed from the original e-Touch version.

etHapticTrisurface
The etHapticTrisurface class is what represents a general haptic object. The etHapticTrisurface class is derived from the etObject class. It contains a triangle mesh defining the shape of the object and also an algorithm to efficiently calculate forces that are caused by the Phantom position being inside the object.

etOctTree
The etOctTree class is used by the etHapticTrisurface for the collision detection part of the force calculation. By turning the triangle mesh into an octree structure, the collision detection can be made very efficient. It basically works in the same way as the octree used for culling away static objects. However this octree is used to cull away triangles on a single object.

etBox
The etBox class is a sub class of etObject and describes a simple box only by its size and not by different polygons. Because of its simpler structure the culling and force calculations are easier than if using an etHapticTrisurface. The etBox is currently only used for dynamic objects.

etSphere
The etSphere class is another primitive shape that is a sub class of etObject. It can be used to represent a haptic sphere and is only represented by its radius. The etSphere is currently only used for dynamic objects.

etCylinder
The etCylinder class is also a primitive shape that is a sub class of etObject. It can be used to represent a haptic cylinder. It is represented by its radius and its height. The etCylinder is currently only used for dynamic objects.

gstForceField
A possibility to determine the way that the forces are calculated within the servo loop is given through the gstForceField class. By subclassing this class and overloading a member function that is called in the servo loop, developers can implement their own algorithms for how to calculate the force. For example, this allows the developer to design the haptic objects, the data structures to hold these objects and so on.

ForceField
As mentioned above the gstForceField class can be subclassed to give developers the possibility to calculate the forces to the Phantom in their own way. For this we use the ForceField class, which overloads the member function also mentioned above. This function starts our own force calculations by calling a method in the World class, which

keeps track of all the haptic objects. It then returns the calculated force which is then sent to the Phantom.

Class diagram

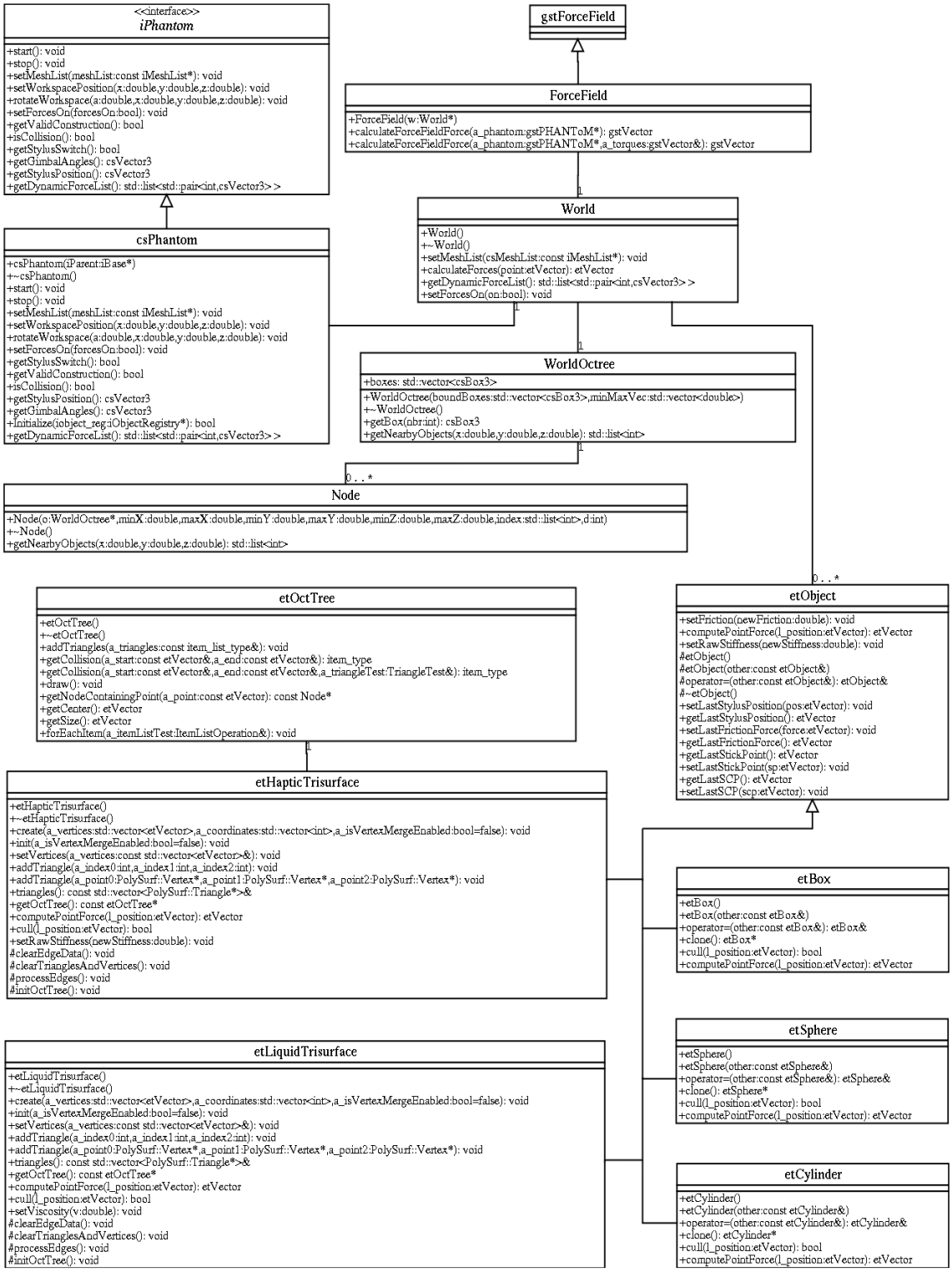

Figure 5.6 *The diagram shows the most important classes used in the plugin. Some e-Touch classes of lesser importance that have not been modified are left out.*

5.4 Threads and processes

When the plugin is used together with Crystal Space there are two different processes running parallel to each other. There is the graphics loop that is run by the game engine and the servo loop, which is needed to update the haptic device with forces. As mentioned before the Phantom needs a very high update rate whereas the demands on the frame rate are not that high. The servo loop is therefore run at a fairly constant speed while the graphics are run without a minimum frame rate, which in practice means as fast as possible.

To communicate with each other the two loops use shared variables that are protected through the use of semaphores for mutual exclusion. The semaphore ensures that only one thread at a time is able to read or write a shared variable. If one thread is inside a critical region and the other one should try to take the semaphore at the same time, it will have to wait until the semaphore is released again before it can access the shared variable.

5.4.1 Crystal Space graphics loop

The graphics loop is controlled by the game engine and is started after the initialization of the application. Every loop of this thread results in a new frame that is shown on the screen. The frame rate is not controlled by the game engine. Instead every frame is drawn as fast as possible. This results in a frame rate that can vary relatively much depending on the amount of computations that has to be made to update the graphics. A typical frame rate when the haptic plugin is running lies somewhere between 50 and 70 frames per second. Every frame the loop calls the SetupFrame callback function, where the application programmer can control what happens in the application. For example, he can update the position of movable objects or the camera.

5.4.2 GHOST servo loop

To keep the haptic device updated at a sufficient speed, the plugin uses a process running at a speed of about 600-1000 Hz. This rate is controlled by the process and if the computations should happen to slow the loop down below 600 Hz, the loop is stopped for safety reasons and no forces are sent to the Phantom. The loop is also stopped if the forces become larger than what the Phantom can handle or if its motors overheat. Every loop, the position of the stylus is checked against the haptic objects and all forces are collected and sent to the haptic device.

5.5 Restrictions

Since this project has a limited time span it has not been possible to implement all the functionality that we would have liked to have in the plugin. A few things simply had to be left out. This chapter contains descriptions of functionalities that we have considered for the plugin, but that we have thought to be of lesser importance or to difficult to do and therefore have decided not to implement.

One thing that has been left out is the ability to dynamically add and remove objects during program execution. As mentioned earlier the world is converted into a haptic world before the start of the Crystal Space graphics loop. If a mesh object is added or

removed during the setup of a frame it will become visible in graphics but cannot be touched. This is a typical feature that could be implemented in a future version of the plugin.

The plugin features, as described in chapter 5.2, the ability to add stiffness and friction to objects. However it does not support more advanced haptic texturing techniques or effects. For example, there is no support for haptic textures built upon techniques like distributed tangential forces similar to bump mapping in graphics or height changes with the help of sine- and square waves. Another feature that could be of interest is the addition of an effect class that would implement different haptic effects like magnetism, flypaper or inertia. This feature would help to guide a visually impaired person in a 3D world or a 3D GUI.

As mentioned earlier there is support for some primitive haptic shapes like boxes, spheres and cylinders. These shapes are up to now only supported for dynamic objects. To arrange so these primitive haptic shapes also are available for static objects requires a transformation of all objects. As it is implemented now only the dynamic objects uses an extra transform because these objects needs to be able to move. For the static objects the world coordinates are used directly for creating the object's coordinates. This makes it faster and simpler during creation and during force calculation, but it makes the structure a bit messier.

Another matter of interest that could easily be added to the plugin is that there are some functions in the core class gstPhantom that have not been transferred to the plugin. These functions give information like calibration status, what state the Phantom is in, what Phantom is used and which driver is used, workspace dimensions etc. Nevertheless we made the decision that we were just interested in the most essential information in the plugin and therefore so far only that has been implemented.

The most difficult thing to add in the haptic plugin is support for deformable surfaces. As stated before, this would require the vertices of the haptic surfaces to be able to move. Since this is not supported in the e-Touch API, the API would most certainly need to be rewritten to support this or a completely new API with another haptic rendering algorithm would have to be used or implemented. However this project does not have the time-span to develop such an API.

6 Demo applications

The next two subchapters describe the two demo applications that were made to demonstrate the capabilities of the haptic plugin. They give an overall description of how the demos were designed and an explanation of what haptic capabilities that are shown with the demos. It is assumed that the reader has some knowledge of how a normal Crystal Space application works.

6.1 Demo application 1

The first demo application shows how to use the haptic plugin in a first person game or a first person like application. This means that the user is able to walk around in a large 3D world and touch and feel the objects within the world with the Phantom. The keyboard is used to move an avatar around in the world. The position of the stylus is shown as a green sphere that turns red if the user touches an object. Of course it also turns red if the user walks into an object using the keyboard with the stylus position in front of him. This way the user can feel the collision with the object.

Figure 6.1 *Screenshots of demo application 1 showing two different maps.*

The demo application was designed in a similar way to a demo that comes with the Crystal Space API called walktest. However it does not have all the features of the walktest demo. The demo application is divided into two classes.

The Demo1 class is used for all the initialization of plugins and the updating of the frames as in a usual Crystal Space application. It handles the communication with the haptic plugin by calling different methods of the haptic plugin interface. First it converts the entire Crystal Space world to a haptic representation. Then it starts the plugin's servo loop. When the haptic world needs to be updated due to a movement, this class calls the suitable method with the appropriate parameters. In the SetupFrame method, the updates of the haptic objects are made additional times during a rendering of a frame to make the haptics feeling somewhat smoother when moving the avatar. Otherwise the haptics will only be updated at the same rate as the graphics.

The CollisionDetection class initializes colliders for all Crystal Space objects and is used to check for collisions every frame when moving the avatar. Actually it also tests for gravity every frame by trying to move the avatar down in an accelerating motion and if it collides with the ground it is moved back before updating the frame. The method that handles collision also makes it possible to step up upon low objects. For example, this makes it possible to climb stairs in a 3D world. The CollisionDetection class uses the RAPID plugin to verify collisions.

The demo shows that it is easy to load different maps in the Crystal Space format and use them with the haptic plugin. To get the proper haptic result one must however think about some factors that are presented in appendix C when designing the maps.

The demo also shows how tags for the objects have been used to set different stiffness and friction for the objects to get a more realistic feeling of materials. It also demonstrates the use of a special liquid object with a tag for viscosity. This object was designed to see that other objects could be inherited and implemented from the etObject class.

Because the plugin does not rely upon other plugins in Crystal Space it should work with all the available plugins and features that can be loaded or used in Crystal Space. However as mentioned before it only represents the meshes that are of type iPolygonMesh as haptic objects. For example, as shown on one of the levels in the first application, a particle system is used. This system can of course only be seen and not touched.

6.2 Demo application 2

The second demo application demonstrates how the haptic plugin can be used in a basic game that takes advantage of the dynamics system plugin in Crystal Space. The game is a 3D labyrinth game where the player pushes a ball around the labyrinth and tries to find the exit in as good time as possible. To stop or help the user along the way there are some dynamic objects that the player can move or use to get passed difficult areas. The Phantom is used to push the ball and the dynamic objects around the labyrinth. The labyrinth is seen from above and the user is able to steer the camera in the x- and z-direction. The player cannot go outside the boundaries of the map with the camera. In the same way as in the first demo application, the position of the stylus is shown as a green sphere that turns red when collision occurs. If the ball gets stuck or has landed in an impossible situation there is a possibility to restart the game at the start position. The demo supports that different labyrinths are loaded.

The demo has a quite simple design because all the dynamics calculations are made by the Crystal Space dynamics plugin. Only one class is used, Demo2. The Demo2 class converts the world to a haptic world in the same way as in the first demo application. There are some differences though. When the world is converted to a haptic world the objects are either added to the dynamics system as static objects or as dynamic objects. This is handled by setting the Crystal Space mesh object name as dynamic_box, dynamic_sphere or dynamic_cylinder (See appendix C). If a mesh object is tagged with this it is added as a dynamic rigid body to the dynamics system. All other mesh objects are considered static and added as static rigid bodies. After the haptic world and the

corresponding dynamics system have been created the servo loop and Crystal Space default graphics loop is started. In each calculation of a frame in the Demo2 method SetupFrame method, several steps in the dynamics system are taken. The dynamics system needs to be updated more often than the graphics to get a proper simulation. When one step is taken in the dynamics system the haptic plugin workspace is also updated. This prevents some vibrations because the haptics is then updated more often than the graphics.

Figure 6.2 *Screenshots of demo application 2 showing two different labyrinths.*

To be able to apply the correct force to the object that the Phantom is currently pushing or poking at, the plugin method getDynamicForceList is used. This method returns a list with indices to the currently active dynamic objects. The indices and forces are used to add the correct forces to the correct dynamic rigid bodies in the dynamics system. In every step in the dynamics system a new list is obtained and forces are added.

This demo application shows that the haptic plugin can easily be integrated in a 3D game. The Crystal Space dynamics system can be used to create a dynamic environment for the Phantom where one can feel weight and friction and have the ability to move around and rotate objects in a reasonably realistic way. This way more interesting haptic games can be created. Although due to the limitation that only primitive objects are supported by the dynamics plugin the ideas to games where the dynamics plugin can be used becomes a bit limited. Obviously one can always implement a specific dynamics system for the game that is being created or simply just move the meshes around as one wants. The haptics will work fine anyway because the limitation does not lie in the haptic plugin. Probably the dynamics plugin will work with more general mesh objects in the future and thus provide more options.

7 Summary

The intention of this master thesis project was to investigate the possibilities of adding haptics to a 3D game engine, and thus be able to take advantage of the fact that a game engine is a very versatile instrument for creating 3D applications. A part of this project was also to implement software, demonstrating the possibilities of having haptics integrated in a game engine.

In this chapter we will evaluate the solution that we have implemented, discuss what could have been made differently and suggest improvements. Future additions and modifications to the plugin are also presented and discussed. At the end of the chapter we present the conclusions that we have made in this master thesis project. There will be a discussion about the problems of integrating haptics in a game engine.

Since we have worked with a game engine and a haptic API, which have not been designed to be used together, we will also discuss how these parts could have been designed differently to better take advantage of each others benefits and to be easier to integrate.

7.1 Results

The work on this master thesis project has resulted in a haptic plugin for the Crystal Space game engine. With the plugin it is possible to construct a virtual 3D environment which is touchable with the Phantom haptic device. The plugin allows the user to navigate in the virtual environment as in common first person 3D games.

The plugin constructs a haptic copy of the 3D world that is loaded by the game engine. This makes it possible to load a world in the same way as in any other Crystal Space application. The objects can either be created directly in the code or they can be loaded from a map file. However, downloading map files constructed for 3D games like Half-Life or Quake will probably not work very well with the haptic plugin. This is due to incompatibility between the way that the Phantom cursor is implemented as a single point and the way that 3D worlds are usually constructed when haptics are not involved. This was one of the things that we wanted to be able to do when we started on the project, but we realized pretty quickly that this was not possible within the scope of this project. The problems associated with constructing a world for the haptic plugin are explained in appendix C.

It is also possible to use dynamic objects as a part of the application. These objects are taken care of by the plugin, which makes it possible to touch objects as they are moving. It should be noted that the haptic objects do not have to be updated explicitly. All movements that are made in the application are handled automatically by the plugin. We also want to point out that the plugin works quite well together with the dynamics system in Crystal Space. This provides the possibility to use a realistic physics model to control the dynamic objects in the world. Another benefit with the dynamics system is that it takes care of all collision detection between the different objects. Not using the

dynamics system when using moving objects requires that collision handling is implemented in the application.

There are a few different types of haptic objects that have support in the plugin. For static objects the plugin uses triangle meshes, but for dynamic objects it is also possible to use boxes, spheres and cylinders that are not constructed from meshes. This will enhance performance somewhat for dynamic objects. When we added these primitive shapes for the dynamic objects we realized that it would be a good idea to be able to use them for static objects as well, but as we already had a working system for this and we knew that it would be a big change, we did not put a high priority on this. This is one of the improvements that we suggest could be added in the future.

The surface of a touchable object is very important for how the object is perceived. By changing the stiffness and the friction of the objects it is possible to create a world that is a bit more realistic than if all objects feel exactly the same. There is a problem with the friction model though. When pushing down on a surface quite hard and only moving around slowly, the friction force constantly changes direction due to involuntary hand movements. This causes the Phantom to start to vibrate which is not what we want. The problem may lie within the friction model itself or in how the different parameters are set. We have however, despite a lot of testing, not been able to fully remove this problem from the plugin. We still feel that the problem is not so severe that we have to remove the possibility to use friction, but it is something that could be improved in the future.

It is fairly easy to add new types of haptic objects by subclassing the etObject class. For example, we added objects that feel like liquids to the world. The viscosity for these objects can be set for each object, which allows the developer to use water or some other kind of liquid in the world. The liquid object is constructed as a triangle mesh, so it is added to the map file as any other mesh, except that it requires to be named differently.

The licensing cost of the haptic system was something that we wanted to minimize in this project. We did not however manage to completely get rid of software that requires some licensing cost. The drivers for the Phantom device and a small part of the GHOST API still have to be used to be able to send data to the device. We would have liked to avoid this and to have a completely free solution, but since this would have required implementing our own Phantom drivers we decided, at an early stage, not to deal with this problem.

7.2 Future work

A performance improvement for the plugin would be not to use mesh objects for basic geometry like boxes, spheres and cylinders that can have easier and more efficient force calculations. To do this, the static objects would have to be created from shape, size and position, just as the dynamic objects are currently, and not from the world coordinates. This means that they would have to use a transform to change the position of the cursor into the local coordinate system of the object to do the force calculations. The way that the octree is constructed, will probably also have to be modified slightly.

The mesh objects should however not be removed completely from the plugin. It must still be possible to construct more detailed objects than the basic shapes.

Haptic texturing is a nice feature that would be great to have in the plugin. An idea on how to do this is to use a grayscale picture that describes the height differences through the different shades. The point, on the surface of the object, that is touched by the cursor could be mapped onto the texture with a function depending on the shape of the object. If the cursor is mapped onto the highest possible part of the texture, the force should be the same as if a texture was not used. Mapping the cursor onto a lower value would mean that the cursor is transformed, out along the surface normal, by the amount collected from the texture and the forces are then calculated as usual. This would make it possible to move in under the actual surface of the object without getting a force. Hence it would feel like a hole. Instead of mapping the cursor position onto a picture, it could be mapped onto a mathematical function to get some other kind of surface structure.

It would also be nice to have some kind of effect class that can be used for applying different effects to haptic objects. This could for example be magnetism that attract or repel the cursor when it comes close to the object. It might also be any other form of force field or even vibrating effects, whatever the imagination can come up with. These effects could be created as objects without visible geometry. This should make it possible to place them anywhere in the world, whether it is in an empty space, as may be the case with a constant force field, or if it is aligned with some geometry, to create a magnetic object for example.

Since there are some problems with the friction model, as described in chapter 7.1, some work could also be done in this area. We have tried to tune the parameters of the model but we have not fully succeeded, so it might be the case that a new friction model has to be implemented to get rid of the problem entirely.

As we have mentioned before, it is not possible to run the plugin with just any map downloaded from the Internet and get a good result when touching the objects. As we also have mentioned, the problem occurs because the Phantom cursor is a single point that is possible to move in between objects where there are no visible gaps. We have a couple of ideas on how to solve this problem, or how to make it better anyway. The first one and probably the easiest, is to implement the cursor as a small ball or some other kind of geometry that is not just a single point. This would make the cursor unable to fit into very small gaps. Using this approach would probably slow down the force calculations to some extent due to more difficult collision detection, but that would probably not be a big problem. The other way would be to recompute the haptic object so that the gaps between different parts are removed. This would involve reconstructing the triangle mesh by dividing up surfaces and merging vertices so that each object consists of only an outer hull and that there are no polygons, or parts of polygons, extending into the hull. If this is even possible would have to be investigated. This approach would probably only slow down the startup stage of the application and not the actual force calculation, making it quite interesting from a real-time perspective.

Adding and removing objects during the execution of an application would be a good feature. This could probably be added rather effortlessly by adding the object to the

dynamic list if it is a moving object or inserting the object in the octree at the correct position if it is a static object. Some new functions in the octree would have to be added and the way that the dynamic objects are stored will have to be modified to some extent.

7.3 Conclusions

During this master thesis project we have found that it is possible to integrate support for an advanced haptic device in a game engine. Our solution of implementing a haptic plugin for the Crystal Space engine works satisfactory and shows that it is possible to walk around in a virtual 3D environment and touch and feel the surroundings with the Phantom device.

Since the documentation of the APIs that we have worked with was very poor, we had to use the implementation process to gather knowledge about the APIs. This means that a lot of trial and error work had to be made to find solutions for problems that occurred. Because of this we were not able to make a complete design in an early stage of the implementation process and therefore our implementation is perhaps not the optimal solution for the problem. Our solution does however work, so this only strengthens our beliefs that an integration of haptics in a game engine can provide a good platform for creating haptic applications.

As we implemented the plugin we came across a few problems that were difficult or even not possible to solve. This was because the functionality that we needed was not present in the APIs or the APIs were not compatible with each other in some areas. We would therefore like to take up how the APIs could have been designed differently to make the integration easier.

One problem that was possible to solve but took quite some time to get right was that Crystal Space and e-Touch use differently oriented coordinate systems. Perhaps this does not seem like a big problem but it is very annoying and the code gets a lot more difficult to understand. This is a good example of what can happen when the different parts are not designed to be used together and there are no standards established.

A feature that would have been nice to have in Crystal Space is a more advanced thread class. With the existing thread class it is not possible to determine the period and it does not support the use of semaphores for mutual exclusion. Had these features been available, it perhaps would have been possible to solve the problem of the graphics loop and the haptics loop running at different speeds. By adding a loop between these it might be possible to smooth out the differences by dividing up the updates from Crystal Space into smaller pieces and instead sending them to the servo loop at a higher rate. In this way the Phantom would be updated more often and it would feel smoother when walking into a wall for example.

There are also a few things that could be different in the e-Touch API. For example, it would be good if the force calculations used a cursor in the form of a sphere or some other geometry and not just a single point. This would make it much easier to use maps that can be downloaded from the Internet. It would also be good if the force algorithm

worked on polygons other than triangles. Then it would be easier to copy or subclass the mesh objects in Crystal Space.

These suggestions for better design of the different APIs would provide a better starting point for a project of this kind, since it would give more alternatives on how to design the system. We have however found that it is possible to get a working system even without these features implemented so they are not essential in any way. The processing power of modern PCs is already sufficient for running this kind of system with a satisfactory result, but with even more powerful computers the result would be even better. It is first and foremost the difference in the rates of the graphics and the haptics loop respectively that would be reduced and the result would be an overall smoother feeling in the applications.

The possibilities of creating good quality graphics with a game engine exceed the possibilities provided by the haptic APIs. A game engine also provides a good way of creating and maintaining static as well as dynamic 3D worlds. The camera can easily be moved around the world and by using plugins for collision detection or physics it is possible to introduce a realistic feeling in the application. Hence there are definitely some good aspects with using a game engine as the base for creating haptic applications.

Appendix A: Haptic plugin

A.1 Requirements

A.1.1 Hardware requirements

To be able to use the Crystal Space haptic plugin one needs a reasonable fast computer and a 3D accelerated graphics card. A minimal setup would be a Pentium or Athlon processor at 500 Mhz and a TNT2 or Geforce 256 card. However to get a standard or better performance it is recommended to use a 800 Mhz+ processor and a Geforce 2 card or better. Ideal is obviously to have a computer with dual processor so the haptic and graphic rendering can be divided between them. Of course one also needs to have one of the haptic devices from the Phantom family.

A.1.2 Software requirements

The plugin should work fine under Windows NT, Windows 2000 and Windows XP. To be able to run it one must have installed the Phantom drivers. In this project we have used version 3.1.6 of the Phantom drivers. If one wants to edit and recompile the code, the GHOST 3.1.6 SDK and Crystal Space version 0.96r003 should also be installed.

Sometimes one might what to be able to have a different workspace for the Phantom than the default that is set in the windows register. For example, when running demo application 1 it would probably be better if the Phantom could reach further in front of the camera than behind the camera. There is unfortunately no other way of changing this than to edit the register data.

A.2 Loading the plugin

To load the plugin in a Crystal Space application one first have to register it with a small program that comes with Crystal Space called scfreg that registers the plugin in the SCF (Shared Class Facility). To register the plugin one first needs to copy the phantom.dll to the Crystal Space directory. After that the plugin can be registered by running:

```
scfreg phantom.dll
```

This will make a new entry in the scf.cfg file. To be able to load it in a Crystal Space application one needs to include the phantom.h file. It can then be loaded into the application in two different ways. One way is to load it through the csInitializer at initialization time like this:

```
if (!csInitializer::RequestPlugins (object_reg,
     CS_REQUEST_VFS,
     CS_REQUEST_SOFTWARE3D,
     CS_REQUEST_ENGINE,
     ...
     CS_REQUEST_PLUGIN("crystalspace.plugins.phantom", iPhantom),
```

```
     CS_REQUEST_END))
{
  ...
}
...

csRef<iPhantom> phantom = CS_ SCF_CREATE_INSTANCE (object_reg,
iPhantom);
```

This is probably the best way to load the plugin because this gives the opportunity to override the plugin at the command line or in a configuration file. Another way is:

```
csRef<iPluginManager> plugin_mgr =
  CS_QUERY_REGISTRY (object_reg, iPluginManager);
csRef<iPhantom> phantom = CS_LOAD_PLUGIN (plugin_mgr,
  "crystalspace.plugins.phantom", iPhantom);
```

But this way there will be more references to clean up.

Another thing to think about before starting the plugin is the visible representation of the Phantom position. The Phantom position can be represented as any object that is movable in Crystal Space. However one should not forget to set the object's name to "PHANTOM". This way the object will be excluded in the conversion from a graphic to a haptic representation. The object's position should be set to the Phantom position every frame, which can be accomplished by using the plugin method getStylusPosition.

A.3 Using the plugin methods

This chapter describes the member functions of the plugin and where to use them in a typical Crystal Space application. It only describes the functions that are available through the interface and not the functions or classes that are outside this interface.

`SetMeshList(iMeshList* meshList)`
After the phantom plugin has been created and all meshes have been loaded and added, either in code or by loading maps or sprites, this function is the first to be called. The function will build up a completely similar world in haptics. Only meshes that implement the iPolygonMesh in Crystal Space are added. The meshes added as worldspawn or with no tag at all will always be checked for collision with the Phantom. Meshes added as things will be added to the octree and only checked when they are close to the cursor. Things tagged as movable or sprites will be added as dynamic haptic objects.

`start()`
Starts the servo loop. This should be called after the SetMeshList function has been called, but before the csDefaultRunLoop is started.

`stop()`
Stops the servo loop.

`getValidConstruction()`
This function checks if the phantom is correctly initialized. Returns true if this is the case.

`setWorkspacePosition(double x, double y, double z)`
This function updates the Phantom workspace position. This means that the origin of the workspace will be moved to the point `x,y,z`. Every time the camera position is updated in SetupFrame a call to setWorkspacePosition is needed to synch the haptics with the graphics.

`rotateWorkspace(double angle, double x, double y, double z)`
This function rotates the Phantom workspace with `angle` radians around the axis made up by `x,y,z`. Every time the camera is rotated in Crystal Space, a call to rotateWorkspace is needed to synch the haptics with the graphics.

`getStylusPosition()`
This function returns the position of the stylus in world coordinates. If one wants to display a graphic representation of the haptic tool this function should be called every frame to get its position.

`isCollision()`
Returns true if the Phantom stylus switch collides with any object.

`getStylusSwitch()`
This function can be used if one wants to utilize the stylus switch in an application. It returns true if the stylus switch is pressed.

`setForcesOn(boolean forcesOn)`
This function can be used to toggle the forces on or off.

`getGimbalAngles()`
Returns a vector with the current gimbal angles for the Phantom. The angles are returned in radians. The first value represent the rotation around the x-axis, the second represent rotation around the y-axis and the last represents rotation around the z-axis.

`getDynamicForceList()`
Returns a list containing pairs with indices to the correct mesh objects in the mesh list and the force that should be applied to them. Only the dynamic objects that are currently touched are returned.

For function descriptions of other classes used by the plugin, study the comments in the source code for the corresponding classes. Although general information about this can be found in chapter 5.

A.4 Changes to include files

During the implementation of the plugin we experienced a few conflicts between files included from different libraries. Because of this we had to make a few changes to some of the included files to be able to compile our plugin.

One problem we discovered when we compiled our project was that we got a conflict between the Standard Template Library (STL) used in Microsoft Visual Studio and a redesigned STL used in GHOST. To solve this we copied the redesigned STL directory and the GHOST include directory to our project directory. The conflict occurred because there were two different versions of vector.h and list.h. We changed the names

of these to gvector.h and glist.h in the copied GHOST STL directory. All files that included these had to be modified to include the renamed files. Changes were made to the files below.

"`vector.h`" was changed to "`gvector.h`" in these files:

```
gstBoundingBox.h
gstIncidentEdge.h
gstPoint2D.h
gstSpatialObject.h
gstTriPoly.h
gstTriPolyMeshHaptic.h
gstVector.h
bvector.h
hashtable.h
stack.h
```

"`list.h`" was changed to "`glist.h`" in these files:

```
gstBinTree.h
gstIncidentEdge.h
gstPoint2D.h
gstPolyPropertyContainer.h
gstVector.h
```

This way GHOST could use the STL files it required and we could use the Microsoft Visual Studio STL in our plugin.

When we designed our octree we used a class in Crystal Space for bounding boxes called csBox3. This class had a method to test if a box is adjacent to another box. The problem with this method was however that it was only considered adjacent when the boxes where overlapping each other in some way. This was not what we wanted because if the octree divided the world exactly at the side of an object's bounding box, this method used by our octree, would only return the object if the cursor was at the surface or penetrating the surface. The problem with this was that it could cause unwanted vibrations for the Phantom. These vibrations occurred since the position of the stylus is quite sensitive, so a change between being in an octree node that contained the object and one that did not, could easily happen. So for example when the user touched the side of the object he got pushed back by a force into the octree node that did not contain the object. Then he tried to reenter quickly and this caused vibration. (See figure 5.5). To solve this we added a method in the csBox3 class that returned true if the two boxes where close enough to each other.

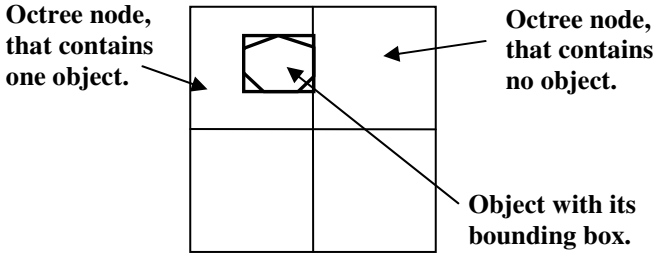

Figure A.1 *An object that has its bounding box side exactly in the same plane as a neighboring octree node caused vibrations for the Phantom.*

Appendix B: Using the demo applications

B.1 Requirements

The two demos have the same requirements as the haptic plugin. To check out these requirements see section A.1.1 and A.1.2.

B.2 Using the demo applications

This section will describe how to run the two applications and how to use some of their features.

B.2.1 Using demo application 1

The first demo application is run by executing the Demo1.exe file. There are a few startup parameters that can be used. The first parameter is the name of a level that will be loaded and used by the demo. Other parameters that can be of interest are the –video option. To use an OpenGL hardware accelerated card with the demo this option should be set as -video=opengl. Another thing that can be useful when running the demo is to recalculate the light maps. This needs to be done at least the first time the level is run or if the level has been modified in some way. The following example runs the level bighouse in OpenGL and recalculates the light maps.

```
Demo1.exe bighouse –video=opengl –relight
```

When demo1 starts up the user is situated in the world loaded from the command line. The user can now move around the world using the arrow keys on the keyboard. The arrow keys left and right rotates the user. The forward and backward keys are used to move the user forwards and backwards. The page down and page up key can be used to tilt the camera up and down. At the same time the user can feel the surroundings by moving around the Phantom stylus.

There is an option to temporarily turn off and on the forces by pressing the 'f' key. The application can be shut down by pressing the 'esc' key.

B.2.2 Using demo application 2

The second demo application is run by executing the Demo2.exe file. The demo2 application also supports parameters to load different levels and set different options in the same way as the first demo. The following example runs the level labyrinth1 in OpenGL and recalculates the light maps.

```
Demo2.exe labyrinth1 –video=opengl –relight
```

The demo2 application starts with the camera looking at the ball from above in its start position. By using the arrow keys the player is able to move the camera sideways and forwards and backwards. Pushing the left or right arrow key moves the camera sideways. Pushing the up or down key moves the camera forwards or backwards. The

object of the game is to use the Phantom to roll the ball through the labyrinth and reach the exit position in as fast time as possible.
It is not just a matter of finding the correct way. There can be things in the way stopping the ball from entering certain areas. These objects must be removed with the Phantom. They can either be lifted away or pushed in a proper direction. Certain objects can also help the player to overcome holes in the labyrinth by filling them with these objects. When the player finds the way through the labyrinth and the ball reaches the x-marked goal position the game is finished. The time taken for this will be displayed for the player.

If the ball gets stuck in a situation that it is not possible to get out of by using the Phantom, the game can be restarted by pressing the 'r' key. The application can be shut down by pressing the 'esc' key.

Appendix C: Creating haptic worlds

There are a few things that one has to have in mind when constructing a world for the haptic plugin. These things mostly concern creating an object from different parts of geometry, without being able to push through the object where the different parts are joined together. However, there are also some possibilities to create a more realistic feeling world, which can be taken advantage of when creating the map. All these issues will be described in this appendix.

C.1 Multi part objects

C.1.1 Introduction

When creating multi part objects, it is very likely that it will be possible to push through the object where the different parts are joined together, even though the graphics does not show a hole through the object. This problem occurs because of the way that the haptic objects are constructed in the plugin or because of the way the forces are calculated. Since the Phantom cursor is a single point it can easily move between two adjacent objects.

There are two ways of creating a multi part object. One way is to simply create the different parts and then put them in their correct positions. This way we will actually get a different object for each part even though, on screen, it may look like they are joined together to create a single object. The other way is to actually group the parts together creating a single object. Both ways create the same visual result but the haptic result can be very different and both ways are actually useful, but for different types of joints.

When the haptic objects are created inside the plugin, an algorithm is used, which joins vertices together if they are located in the same position. For example, two different parts that share an edge will be joined together at this edge, making it impossible to push through it using the Phantom. However, this algorithm only works on a single object, so the parts have to be grouped together in the map. This approach can be used when joining parts where the adjacent edges are of the same length and their vertices are in the same positions.

If the adjacent edges should be of different lengths or if one part sticks out of the surface of the other part so that the vertices are not in the same position, the objects have to overlap. This will only work with objects that are not grouped together as a single object and the reason for this is the way that the forces are calculated. If two different objects overlap the forces will be calculated separately for each object and thus the resulting force will correspond to touching both objects. If overlapping is used within a single object the force calculation algorithm of that object will not be able to handle it and it will be possible to push into the object where the two parts intersect.

The rest of chapter C.1 will describe a number of difficulties related to building a map for the haptic plugin. Of course, all different situations that can occur will not be

presented, but there will be given some basic methods of solving problems, that will work in a number of situations that are similar to the ones described here. The examples used here mostly concern building some kind of house, but the solutions are not restricted to be used only with walls, floors and ceilings, but with all kinds of objects that present the same type of difficulties.

C.1.2 Corners

Walls are very common in a 3D environment and are a good example because they are often joined together with each other, or together with the floor and ceiling, creating corners. Corners can be made in three different ways.

The easiest way is to make the two walls overlap in the corner and keep them as two different objects. This way however, causes a problem with the graphics and that is when two surfaces overlap in the same plane, there will be a problem with the z-buffer causing a flicker as the pixels are sometimes taken from one surface and sometimes from the other surface. This phenomenon is called z-fighting. However, in a corner this problem only occurs on the outside so it can still be used if the outside of the corner is not visible.

Another way is to position the walls so that they share one edge, leaving a square indentation on the outside of the corner, and then group the two parts together. This will cause them to be joined together at their common edge. If the indentation on the outside is not desirable it can be filled with a thin square part of the same height and with as the walls. This part will share vertices with both walls and therefore not create any gaps.

The third way is the most time consuming but also the one that creates the best visible results and we recommend that it should be used where it is possible. By making a 45° angle at the end of each wall using the vertex manipulation tool and grouping the walls together, they will form a perfect corner since both their inner and outer edges will be joined together.

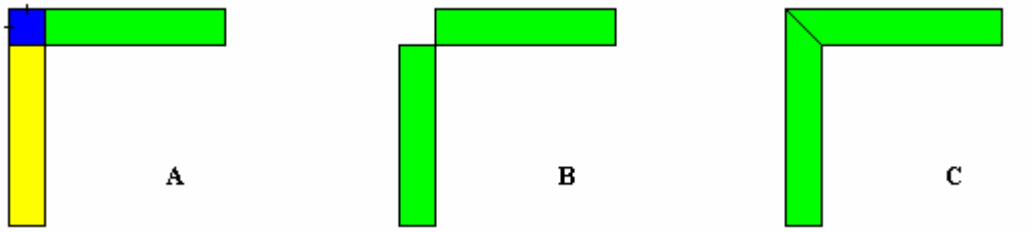

Figure C.1 *The figure shows three different ways of joining two walls together in a corner. Method A differs from the other two in that the walls are not grouped into a single object, but are simply two objects overlapping. The crossing lines in the corner show the surfaces where so called z-fighting occurs. All three ways will prevent the Phantom cursor to be pushed through the walls at the joint.*

C.1.3 Doors and windows

When creating holes through objects, for example when making doors and windows, there are a couple of ways to get a good result. Figure C.2 shows three different ways of constructing a wall with a window that looks good in Crystal Space. However, it is only method B and C that will work with the haptic plugin. The reason for this is that all adjacent parts share their entire edges, which is not the case if method A is used.

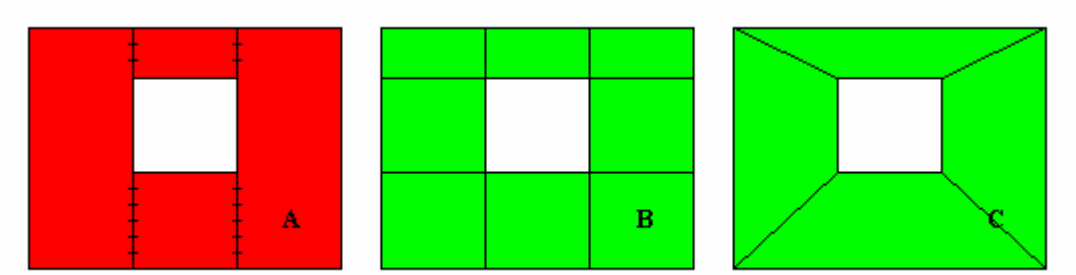

Figure C.2 *The figure shows three different ways of constructing a window. Method A will cause the Phantom cursor to be able to push through the wall at the joints marked with crossing lines. Method B and C will work fine because all different parts share edges with their neighbors.*

C.1.4 Large objects

To speed up the haptic calculations the plugin is able to handle large objects without adding them to the octree containing the rest of the static objects. The reason for this is that an object that covers a large share of the world space will become part of a lot of nodes within the octree and possibly cause the octree to become much larger than optimal. To prevent this from happen, the object can be marked with a certain class name in the map (see chapter C.2). One should have in mind that it is the bounding box of the object that is used for the octree and that the bounding box is aligned with the coordinate axes.

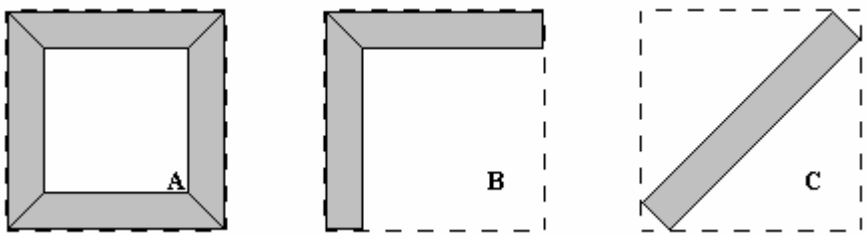

Figure C.3 *Objects that are of different shapes and sizes may still have equally large bounding boxes. This can be good to remember when constructing large objects. Since the bounding boxes are aligned with the coordinate axes, objects that are diagonal to the coordinate axes get larger bounding boxes than objects of the same shape that are axis aligned. If the bounding box covers a large amount of the world space it is wise not to add it to the octree.*

It is also important to remember that the bounding boxes are created from a single object. This means that two walls that are put together by overlapping, and are not grouped together to a single object, will each get their own bounding box. A single wall object may very well be considered as a small object, especially if it is aligned with the coordinate axes. Therefore it can be good to use overlapping for some objects, providing that z-fighting can be avoided so that the graphics are not ruined.

C.2 Class-names and keys

When creating a map it is possible to customize it to make it more efficient and more realistic from a haptic viewpoint. To take advantage of this, the objects have to be assigned class names and keys, describing what kind of an object it is and what properties it has. This information can be added by converting an object into one of the entities listed below and then assigning the different keys. It is not necessary to use any of these entities, but this will result in that there will be no optimizations and only the default values will be used for the key values.

Here is the list of the entities that get special treatment from the haptic plugin:

- Class name: **worldspawn**

  Using this entity will cause the object to be considered for collision detection every loop of the force calculations. In other words, it is not put into the octree. This entity should therefore be used for very large objects, taking up a lot of space. If no keys are used together with this entity it is equivalent to not using an entity at all.

- Class name: **thing**

  The thing entity should be used for all objects that should be part of the octree. This entity will most likely be the most common entity in any map since it should be used for all smaller objects and walls.

  — Key name: **cs_name**

    This key is used to define a custom name for the thing. With the haptic plugin there are a few names that are reserved and should be used with care.

    — Value: **liquid**

      If the name liquid is assigned to a thing a certain liquid object will be created, so this name should only be used when it is intended that the object should feel like a liquid.

      The following three tags are only used for dynamic objects that should not be represented as mesh objects.

    — Value: **dynamic_box**

      If the name dynamic_box is used the object will be added as a box in the plugin.

    — Value: **dynamic_sphere**

      If the name dynamic_sphere is used the object will be added as a sphere in the plugin.

    — Value: **dynamic_cylinder**

      If the name dynamic_cylinder is used the object will be added as a cylinder in the plugin.

  — Key name: **viscosity**

    This key can be used to create liquids of different viscosity. The viscosity value will only have effect if it is used together with the cs_name **liquid**.

- Class name: **All class names**

  These keys work together with all kinds of entities.

  — Key name: **stiffness**

    The stiffness of an object determines if the object feels hard or soft.

  — Key name: **friction**

    This key determines the amount of friction for the object.

Appendix D: Dictionary

Avatar - An interactive representation of a human in a virtual reality environment

Beziér curves - Curved lines or paths described by mathematical equations.

Bump mapping - A technique to simulate bumpy surfaces by varying the surface normals and still maintain a flat surface.

First-person game - A 3D game played from a first-person point of view. This means that the player is able to navigate in the world and see what is around him, but he is generally not able to see a representation of himself.

GLUT (OpenGL Utility Toolkit) - An independent window system toolkit for writing OpenGL applications.

GNU copyleft license - A software license giving all users the freedom to redistribute and change the software. GNU (GNU's Not Unix) is an acronym for the Free Software Foundation (FSF).

Light map - A pre-calculated two-dimensional texture map designed to provide lighting effects to another texture.

Mip-mapping - Using multiple images, with different resolutions, of a texture map to present textures at varying distance from the viewer's perspective.

Object space - The local coordinate system of a three dimensional object.

OpenGL (Open Graphics Library) - An operating system independent low-level graphics library specification.

Particle system - Dynamic simulation of a group of moving objects. Often used for simulating things like, fountains, fire, smoke, snow, rain etc.

Phong shading - A lighting model that uses interpolation of vertex normals to calculate smooth shading of surfaces.

Procedural textures - Programmable graphic textures that can be rendered in real-time.

Python - An interpreted, interactive, object-oriented programming language that is portable to many different operating systems.

SCF (Shared Class Facility) - A library intended to separate C++ class implementations from the programs that use them.

STL (Standard Template Library) - A collection of predefined algorithms and data structures for C++.

View plane - The plane, onto which the image is projected. The view plane is orthogonal to the direction in which the camera is pointing.

VRML (Virtual Reality Modeling Language) - A markup language for describing interactive 3D objects and worlds.

World space - The coordinate system of a three dimensional world.

References

Literature

[1] J. Tyberghein, A. Zabolotny, E. Sunshine, T. Hieber, M. Ewert, S. Galbraith, 2003, "*Crystal Space Open source 3D Game Toolkit Documentation*", Edition 96.003.1 for Crystal Space 96.003.

[2] 2000, "*GHOST SDK Programmer's guide*", version 3.1, Sensable Technologies, Inc.,Woburn MA

[3] 2001, "*e-Touch Programmer's Guide*", beta release 1.0, Novint Technologies, Albuquerque NM.

[4] 2001, "*GHOST SDK API Reference*", version 3.1, Sensable Technologies, Inc., Woburn MA.

[5] 2001, "*e-Touch reference manual*", version 1.0.0 beta 3, Novint Technologies, Albuquerque NM.

[6] T. Anderson, N. Brown, 2001, "*The ActivePolygon Polygonal Algorithm for Haptic Force Generation*", Novint Technologies, Albuquerque NM.

[7] G. C. Burdea, 1996, "*Force and Touch Feedback for Virtual Reality*", John Wiley & Sons, Inc., New York NY.

[8] N. Reistad, 1998, "*Naturvetenskaplig problemlösning*", 2nd edition, Lund Institute of Technology, Lund.

[9] A. Gosline, 2001, "*Evaluation of Friction Models with a Haptic Interface*".

[10] R. J. Stone, 2000, "*Haptic Feedback: A Potted History, From Telepresence to Virtual Reality*", Chester House, Sale.

[11] M. Basdogan, M. A. Srinivasan, 2002, "*Haptic Rendering in Virtual Environments*", Handbook of Virtual Environments, Lawrence Earlbaum, Inc., London.

[12] Jan Skansholm, 2000, "*C++ Direkt*", Studentlitteratur AB, Lund.

URLs

[13] "*Crystal Space 3D*", http://crystal.sourceforge.net/drupal/

[14] "*SourceForge.net: Project Info - Crystal Space 3D SDK*", http://sourceforge.net/projects/crystal

[15] "*Sensable Technologies - Leading provider of touched based applications and tools for your 3D design and product development process*",
http://www.sensable.com

[16] "*e-Touch*",
http://www.etouch3d.org/

[17] R. Smith, "*Open Dynamics Engine - ODE*",
http://opende.sourceforge.net/

[18] E. W. Weisstein, "*Friction - from Eric Weisstein's World of Physics*",
http://scienceworld.wolfram.com/physics/Friction.html

[19] "*Haptik*",
http://www.nada.kth.se/kurser/kth/2D1413/AGI02-haptik-OH.pdf

[20] "*Immersion Corporation*",
http://www.immersion.com/

[21] "*Haptics-e The Electronic Journal of Haptics Research*",
http://www.haptics-e.org/

[22] "*Reachin - The leading haptic software solution company*",
http://www.reachin.se/