Master Thesis

Virtual environments for a Magnetic Levitation Haptic Device

Martin Oberhuber

Prof. Dr. Ralph L. Hollis
The Robotics Institute
Carnegie Mellon University (CMU)
Adviser

Prof. Dr. Bradley J. Nelson
Institute of Robotics and Intelligent Systems
Swiss Federal Institute of Technology Zurich (ETH)

2008-04

IRIS
Institute of Robotics and Intelligent Systems

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Preface

The underlying project has been realized during my 6-month stay at the Robotics Institute of Carnegie Mellon University. My stay at CMU was one of the most exciting times during my studies. The project I worked on perfectly suited my interests and excited about the every days challenges I developed a passion for programming and further increased my interest for robotics. I am indebted to a couple of people who contributed to this wonderful experience.

First of all I want to thank my supervisor Prof. Ralph L. Hollis who gave me the opportunity to contribute to the Magnetic Levitation Haptic project. I want to thank my tutor at ETH Zurich, Prof. Bradley J. Nelson for setting up the connection with Carnegie Mellon University.

Special thanks are dedicated to Bertram Unger, who particularly helped me with initial problems on the software and for the fruitful discussions about haptic programming.

I am also greatly indebted to Kei Usui who was my mentor for programming and computer science questions. He furthermore helped me a lot in debugging my algorithms. I also thank Isabel Gardocki for the literary improvement of this thesis.

Finally, many thanks are dedicated to my parents for supporting me in such a great way during my studies.

Abstract

This thesis describes the implementation of 3D virtual environments for a magnetic levitation haptic device which has been developed at Carnegie Mellon University. This interface allows the user to interact with virtual and remote environments in six degrees of freedom with haptic (force/torque) feedback to the user's hand. The technology gives high motion resolutions (2-3 microns) at high position bandwidths (120 Hz), thereby providing a "high fidelity" interaction experience. The aim of the project was to create several demos for this specific device using an application programming interface which takes care of the low-level control of the hardware. Three different demos have been implemented with the purpose of highlighting the devices capabilities.

The first demo is a cube imprisoned in a box. The user has the impression that he or she holds a virtual cube in the hand. By manipulating it around and hitting the walls of the outer box the user feels forces and torques resulting from the interaction of the two virtual bodies. The second demo is an implementation of a virtual texture plate. Three sine wave texture patterns with different amplitudes and periods and a smooth surface can be felt by the user when moving a virtual probe across the surfaces. The third demo involves touching virtual objects with a complex geometry. The object is represented by a triangular mesh which can contain thousands of triangles.

The following thesis describes the implementation of these three demos and provides insight in the difficulties of creating a visio-haptic experience. This involves efficient integration of physical simulation of the virtual world, displaying the simulation graphically and finally controlling the haptic device. Altogether must operate at a high update rate and therefore programming is challenging.

Zusammenfassung

Die vorliegende Arbeit beschreibt die Implementierung von 3D virtuellen Umgebungen eines magnet-schwebe Haptik Gerätes, das an der Carnegie Mellon University entwickelt wurde. Dieses Gerät ermöglicht es dem Benutzer mit einer virtuellen oder fernliegenden Umgebung zu interagieren und stellt dabei eine Kraftrückkoppelung in 6 Freiheitsgraden zur Verfügung. Die verwendete Technologie garantiert hohe Bewegungsauflösung (2-3microns) bei gleichzeitig hoher Bandbreite von ungefähr 120 Hz. Diese überdurchschnittlich guten Eigenschaften bewirken ein sehr realitätsnahes Gefühl der virtuellen Umgebungen.

Ziel diese Projektes war es mehrere Anwendungen für diese Haptik-Gerät mit Hilfe einer Programmierschnittstelle, die für die direkte Hardware-Regelung zuständig ist, zu entwickeln. Dabei wurden 3 verschiedene Demonstrationsanwendungen programmiert die vor allem zugeschnitten sind um die einzigartigen Eigenschaften des magnet-schwebe Haptik Gerätes hervorzuheben.

Das erste Anwendungsbeispiel hat als virtuelle Umgebung einen Würfel der sich im Inneren eines größeren Würfels befindet. Der Benutzer hat das Gefühl den kleinen Würfel in seiner Hand zu halten. Wenn er nun diesen Würfel bewegt und ihn gegen die Kanten und Wände des größeren Würfels stoßt spürt er die Kräfte die durch die Zusammenstöße der beiden virtuellen Objekte entstehen.

Das zweite Demonstrationsprogramm simuliert die geometrische Beschaffenheit von Oberflächenmustern bestehend aus Sinuswellen mit verschiedenen Amplituden und Perioden. Der Benutzer hält dabei einen virtuellen Tastkopf in seiner Hand, bewegt diesen über die sinus-gewellten Oberflächen und spürt somit die verschiedenen Muster.

Das dritte Programmbeispiel macht es möglich virtuelle Objekte mit komplizierter Geometrie zu berühren. Die Objekte werden mittels eines Dreiecksmesh dargestellt, welches aus mehreren Tausend Dreiecken bestehen kann. Insbesondere stellt diese Tatsache eine große Herausforderung für die Programmierung von virtuellen Umgebungen dar.

In der folgenden Arbeit wird beschrieben wie diese drei Demonstrationsprogramme implementiert wurden und welche Schwierigkeiten bei der Erstellung von virtuellen Umgebungen zu beachten sind. Eine effiziente Integration von physikalischer Simulation, graphischer Darstellung und Regelung der Hardware ist unumgänglich, da alles zusammen bei einer hohen Aktualisierungsrate durchgeführt werden muss. Dieser Umstand stellt besondere Herausforderungen an die Programmierung solcher virtuellen Umgebungen.

Contents

List of Tables

List of Figures

1 Introduction

One of the new fields in computer interaction is called haptics and allows computer users to interact mechanically as well as visually with a virtual environment. The word haptics comes from the Greek root haptikos meaning "able to grasp or perceive" - which describes the scope of this field very precisely.

The field of haptics is very young, but nevertheless it is already very promising to have a big impact on many fields of human activity. For instance medical procedures could be practiced on virtual patients, or CAD modeling can be made much easier by mechanically manipulating the 3D objects, which would be very helpful in assembly. Effort-reflected flight or other vehicle control could be accurately simulated with reaction forces and vibration feedback to the operator.

In the future, these capabilities could be exercised over long distances, opening up new applications in remote diagnostics and repair and tele-medicine.

1.1 Critical consideration for haptic devices

Haptic devices are usually very complicated systems and therefore several considerations are essential when designing such devices. These important characteristics determine the performance of the device and have a big impact on the capability of generating reality like perception of rigid, soft and liquid bodies. The most important characteristics of haptic devices are:

Degrees of Freedom: Haptics interaction with 3D virtual environments is very restricted with a device having less than six degrees of freedom [19], [20]. The human arm and hand for example has much more than 6 DOF and is therefore considered as a redundant system. The additional degrees of freedom provide humans with high flexibility when it comes to manipulating real objects. 6 DOF are required at least to achieve any possible pose of the manipulandum in the device's workspace. On the other hand, providing 6-DOF capability traditionally has implied mechanical complexity and much higher cost.

Bandwidth: The bandwidth of a haptic device is an essential property for providing a realistic behavior of the virtual environment. A Low bandwidth makes the simulated environment feel sluggish. Unfortunately achieving high bandwidths in an electromechanical device is very challenging for the design and control of the hardware.

Resolution: In order to provide a realistic feel of the virtual environment, it is essential to be able to perceive small details, such as fine textures [3] or stick-slip friction. This is only possible if the device resolves extremely small changes in position and force.

Impedance (stiffness) range: When the user moves through free space, no friction should be felt, but if the virtual object attached to the manipulandum hits a hard surface, it should feel perfectly rigid. In practical systems, neither of these objectives is achieved, however the range of possible impedances felt by the hand should be maximized.

2 The Magnetic Levitation Haptic Device

Even if the field of haptics is relatively young, many well functioning devices have already been commercialized. A very popular device on todays market is the three-degree-of-freedom PHANToM device [1], which uses a parallelogram configuration with DC motors and cable transmission. A 6-DOF version of this device [2] has been developed as well, but was not very successful due to weak force and torque characteristics. Furthermore very high costs have limited its availability.

In 1998 an alpha version of a so-called magnetic levitation haptics device has been developed by Prof. Ralph Hollis at Carnegie Mellon University. This alpha version has been reengineered in the last 4 years, and remarkable improvements in performance have been achieved while lowering the cost, in order to make it suitable for commercialization (figure 2). Its applications will include CAD, medical training, tele-manipulation, hand controllers, vehicle simulation, and computer interfaces for blind users.

This device exhibits extremely high fidelity due to the fact that no motors, gears, bearings or linkages are used. Therefore artifacts of conventional haptic devices like cogging and friction are not present. Furthermore the magnetic levitation device stands out for its extremely high stiffness range [5, 6, 7, 8, 9, 10, 11, 12, 13].

A direct connection between the simulated environment and the user's hand is

Figure 1: Magnetic Levitation Haptics Device (Hollis, CMU).

given by a controlled magnetic field in which the handle levitates. The magnetic field is able to control the handle in 6 DOFs friction-free providing high bandwidth and high resolution motion. With no doubt this approach provides the highest fidelity haptic interaction currently achievable.

The handle is attached to a light weight bowl-shaped "flotor" containing six spherical coils. The magnetic field is created by NdFeB permanent magnets. The pose of the flotor is determined by tracking three LEDs which are attached to the flotor using optical position sensors (PSDs)

The MLHD system particularly stands out for its high -3dB position bandwidth of 130 Hz compared to the 10 Hz of the popular 3-DOF PHANToM device. Furthermore the stiffness of 40 N/mm cannot be achieved by any other 6DOF haptic device.

The disadvantage of the magnetic levitation haptic device is its small motion range of +-14 mm in translation and +-8 in rotation. However this can be overcome by scaling, using rate control or indexing.

Main performance characteristics of the device:

Attribute	**Value**	**Attribute**	**Value**
Degrees of Freedom	6	Maximum impedance	40.0 N/mm
Translational Workspace	spherical	Minimum impedance	0.002 N/mm
Translation range	+/-14	Impedance ration	20,000:1
Rotation range	+/-8	Flotor mass	503 g
Position bandwidth	120 Hz (-3dB)	Levitation power	4.5 W
Position resolution	$3\mu m$ (1σ)	Device pose	adjustable
Peak force	40 - 100 N	Manipulandum	interchangeable
Peak torque	4.5 - 8.8 Nm	Manipulandum buttons	2
RT operating system	QNX Neutrino 6.3	User interface	structured API

Table 1: Performance characteristics of 2nd generation magnetic levitation haptic interface.

3 The Maglev working principle

The user holds the freely-levitated handle of the haptic device, maneuvering it in free space to provide position and force/torque information to a physically-based 3D simulated environment. The running simulation provides 6-DOF force/torque output to the manipulandum, and consequently to the hand. Both, the proprioceptive (kinesthetic) senses of the fingers, hand, and wrist as well as the tactile senses in the skin are involved in the interaction.

The haptic controller is housed separately from the haptic device to increase flexibility. The position and orientation of the manipulandum is computed from information given by the optical sensors and actuator currents are computed from the commanded force torque wrench. An application programming interface (API) provides a straightforward interface to hardware capabilities. In most cases, haptic rendering algorithms are executed in the user's attached workstation, but the system also supports running the user's algorithms directly on the haptic controller.

The hemispherical shape of the levitated part, called flotor, allows an equal range of motion in all directions, and enables the user to easily grasp a manipulandum located at the center. The stator will enclose the flotor except for holes on the inner and outer surfaces; the manipulandum to be grasped by the user projects through the inside hole while all necessary wiring from the flotor hangs down through the outer hole [9] (figure 2).

The typical position of the user's hand is also shown in figure 2. As the device is manipulated by the user, the control system causes the dynamics of the flotor to reflect the calculated dynamics of a 3D virtual tool in the given simulated task. Reaction dynamics on the tool are felt by the user through the fingertip and hand.

3.1 Position/Orientation Sensing

An essential requirement for any haptic device is that the user's hand motion is accurately measured and sent to the application which is simulating the virtual environment. In our case, this means accurately measuring the position and orientation of the flotor in real time as it is controlled by the user through the manipulandum. In a further step the measured position is rigidly assigned to an object of the virtual world. To achieve a realistic performance, this position has to be measured as accurately as possible.

The sensor design for the magnetic levitation haptic device consists of three light-emitting diodes (LEDs) which are attached to the flotor and three 2-D lateral-effect position sensitive photodiodes (PSDs) situated orthogonally on the outer stator (figure 3). These provide six independent variables (x and y on each sensor) which can be transformed into the position and orientation of the flotor. A lens arrangement in front of each PSD will demagnify the motion of each LED by a ratio of approximately 10:1. Wide angle LEDs will be used to ensure that light from the LED always fills the lens system.

Once the x and y coordinates of each LED point is measured, the forward kinematic calculation has to be performed in order to achieve the position and orientation of the flotor.

The stator frame S is attached to the outer body of the haptic device and has its origin at the center of the flotor's workspace. The relation between this stator frame and the three frames of the PSDs, A, B and C, is a constant homogenous

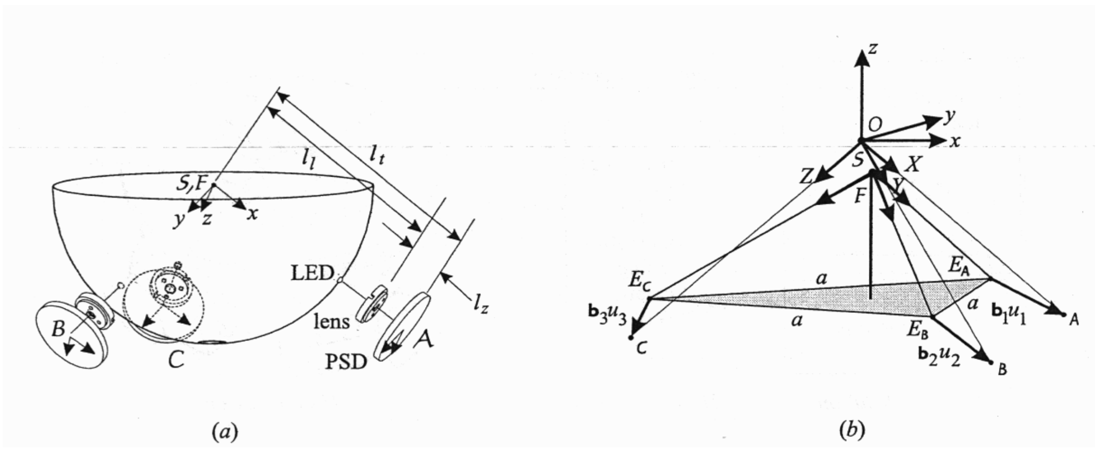


Figure 2: Optical sensors and related coordinate frames ([23] NFS Proposal, Hollis 2003).

transformation. The coordinate frame F is attached to the flotor and it's origin is coincident with the one of the stator frame S when the flotor is located in the center of the work space.

Calculating the inverse kinematics, e.g. finding the coordinates of the LED light spot on the PSDs as the flotor moves around, is straightforward. Since frame F is rigidly connected with the triangular base (shown in gray) to which LED emitters E_A, E_B, and E_C are attached, knowing the homogenous transformation between flotor and stator frame is sufficient to find the light spots on the PSDs. For example, for LED a (emitter E_A), with the constant vector A_f (the location of LED a with respect to the flotor frame) and the constant transformation matrix T_s^a, this is given by,

$$A^a = T_s^a T_f^s A^f \tag{3.1}$$

The light source is imaged on the PSD at a point ($s_{a,x}$, $s_{a,y}$) given by the lens equation

$$\begin{bmatrix} s_{a,x} \\ s_{a,y} \end{bmatrix} = (\frac{-l_z}{A_z^a}) \begin{bmatrix} {A^a}_x \\ {A^a}_y \end{bmatrix} \tag{3.2}$$

where l_z is the distance from the lens to the PSD (figure 3a).

The inverse kinematics is a function of six independent variables describing the position and orientation of the flotor frame with respect to the stator frame S. The representation used is n_1, n_2, θ, X, Y, Z, where n_1 and n_2 represent the x and y components of a normalized rotation axis, θ the rotation angle about this axis, and X, Y and Z the translational motion of the origin of F. Creating T_f^s as a function of these six variables plus n_3 (the z component of the rotation axis) and computing $s_{a,x}$ and $s_{a,y}$ from eqs. 3.1, 3.2 above gives the LED light spot position on PSD A:

$$s_{a,x} = \frac{l_z l_l [n_1 n_3 (1 - \cos\theta) - n_2 \sin\theta] + Z}{l_l [n_1^2 + (1 - n_1^2)\cos\theta] + X + l_z - l_t} \tag{3.3}$$

$$s_{a,y} = \frac{l_z l_l [n_1 n_2 (1 - \cos\theta) + n_3 \sin\theta] + Y}{l_l [n_1^2 + (1 - n_1^2)\cos\theta] + X + l_z - l_t} \tag{3.4}$$

where l_l is the distance from the origin of S to the lens and l_t the distance form the origin to the PSD surface. The computations for sensors B and C are equivalent.

Unfortunately, no closed-form solution exists for computing the forward kinematics, i.e. finding the flotor position and orientation from the light spot position. Therefore a fast numerical root-finding procedure, devised by Yu and Fries [21], is applied. It uses the flotor motion history to obtain initial position guesses. Referring to figure 3b the directions of the three vectors from the emitters to the lenses are known; i.e. they can be readily computed from the measured light spot positions on the PSDs and the (fixed) transformations of the sensor frames A, B, and C from the stator frame S. Only the magnitudes of the vectors are unknown, so the solution of the position and orientation of the flotor can be reduced to a third-order problem. The lens positions are given by S_i, and the LED positions are E_i; the LED-to-lens unit vector directions are b_i and their magnitudes are u_i.

For i, j= 1...3 and $i \neq j$ it can be seen from the figure that

$$E_i = S_i - b_i u_i \tag{3.5}$$

$$|E_i - E_j|^2 = a^2 \tag{3.6}$$

where a is the length of each side of the triangle determined by the three LEDs. Substituting for each E_i and E_j in eq. 3.5 and 3.6, the following system of three quadratic equations with three unknowns is obtained

$$u_1^2 + 2g_3 u_1 u_2 + u_2^2 + 2f_{12} u_1 + 2f_{21} u_2 + d^2 = 0 \tag{3.7}$$

$$u_2^2 + 2g_1u_2u_3 + u_3^2 + 2f_{23}u_2 + 2f_{32}u_3 + d^2 = 0 \tag{3.8}$$

$$u_3^2 + 2g_2u_3u_1 + u_1^2 + 2f_{31}u_3 + 2f_{13}u_1 + d^2 = 0 \tag{3.9}$$

where

$$g_k = -b_i \cdot b_j \tag{3.10}$$

$$f_{ij} = (S_i - S_j) \cdot b_i \tag{3.11}$$

$$d^2 = (S_i - S_j)^2 - a^2 > 0 \tag{3.12}$$

An approximate solution to this system is obtained by differentiating eqs. 3.7-3.9. After that, u_i is updated using the previous solution values and subsequently accuracy is improved with the iterative newton gradient method. Before this method can be applied, the large inherent pincushion nonlinearity of the PSDs must be compensated. This is done in an off-line automated calibration step which will generate a look-up table.

3.2 Actuation

Another key role in any haptic device is the actuation. It is responsible for sending forces and torques, calculated by the physical simulation of the application program, as accurately as possible to the user's hand. In this case, it means accurately computing currents to be applied to the six flotor coils to exert the correct "wrench" on the flotor, and hence on the manipulandum held by the user.

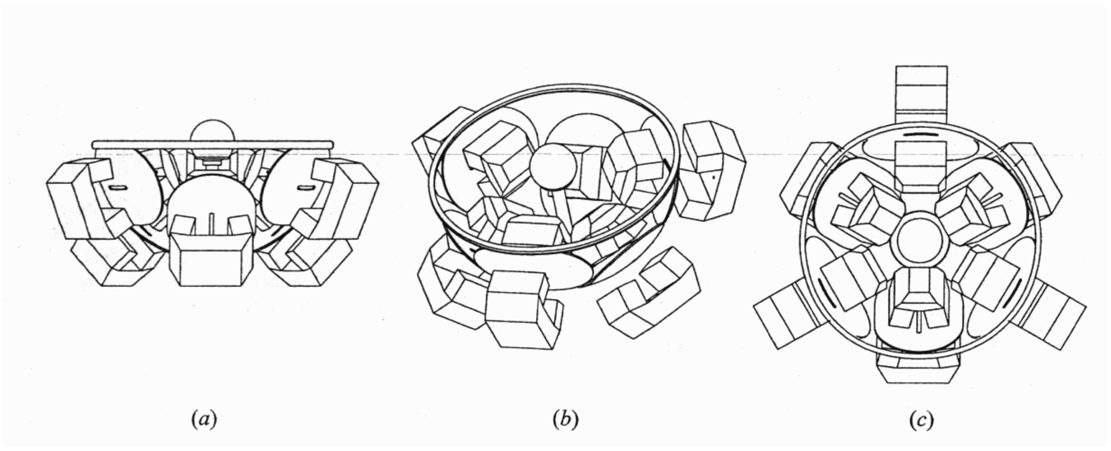


Figure 3: Magnet and coil arrangement ([23] NFS Proposal, Hollis 2003).

Figure 4 shows the arrangement of magnet assemblies and spherical coils. There are 6 magnet assemblies, each with 4 NdFeB magnets, arranged in inner and outer stators. Forces are generated in each of the 6 coils, producing an arbitrary 6-component wrench on the hemisphere and hence on the manipulandum. The flotor is free to move in translation within the magnetic gap, and is free to move in rotation over angles that allow the actuators to continue producing force.

To efficiently generate forces and torques in all directions, the actuators will be arranged in a tightly-packed configuration equally spaced around the circumference of the flotor with three actuators immediately below the rim and the other three centered 45°below the rim (figure 4). Coil positions are defined with respect to the flotor coil coordinate frame K which differs only by a constant rotation matrix from the flotor frame F (figure 3). The forces and torques generated on the flotor when it is in the centered position can be expressed as

$$F = AI \tag{3.13}$$

where F is a 6-vector force/torque wrench exerted on the flotor center, I is a 6-vector of the coil currents, and A the (purely geometric) 6x6 matrix which maps them. A can be calculated from the sums of the Lorentz forces and torques.

$$f_k = I_k \times B_k \tag{3.14}$$

$$\tau_k = c_k \times f_k \tag{3.15}$$

where B_k are the magnetic fields of each actuator magnet assembly and c_k are vectors to the coil centers. The air gap between the inner and outer magnet assemblies is 34 mm to accommodate the thickness of the flotor, its range of motion, and its curvature.

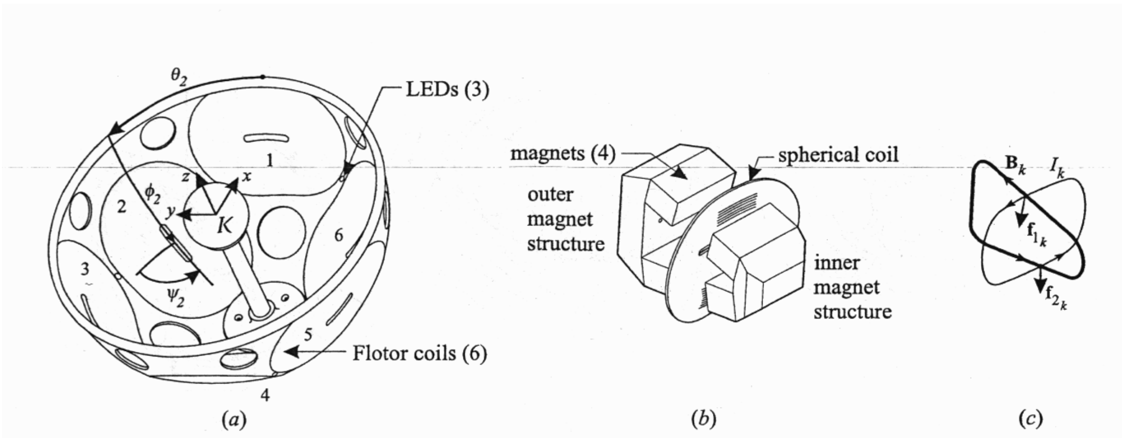


Figure 4: Flotor and magnet structure: (a) levitated flotor showing coils, (b) Lorentz actuator (one of 6), (c) force generation in each actuator ([23] NFS Proposal, Hollis 2003).

Figure 5 shows the Lorentz actuator design. The current I_k flows in the racetrack-shaped spherical coil k and interacts with the magnetic field B_k to produce the force f_k. Forces generated on one side of the loop add with forces generated on the other side, except for a small difference in direction due to the coil curvature.

3.3 Control

Once the relation between the flotor frame F and the stator frame S is calculated, simple control laws such as PD (proportional-derivative) can be used to stably levitate and control the motion of the flotor. Since flotor linear and angular velocities are not directly measurable, they must be estimated from filtered position measurements or from an observer. Feed-forward terms can be added for gravity cancellation. When the device is interacting with a virtual environment, stable control becomes much more complex.

3.4 Setup of the Magnetic Levitation Haptic Interface

Implementing virtual environments demands a very careful design of the architecture of the simulation. The biggest challenges is to integrate physical simulation, hardware control of the haptic device and graphical display of the simulation so that high update frequencies of the haptic forces can be achieved. These three main tasks are performed in different threads to guarantee the most possible independence. The three threads communicate with each other which further complicates the situation, since they are running at different update frequencies. The graphics loop, which is responsible for displaying the simulation of the virtual environment has to run at least at the screen update rate. We found out that 100 Hz is a reasonable value for the loop frequency.

The hardware control loop is basically responsible for calculating the currents for the coil mounted on the flotor. It transforms the force input coming from the physical simulation to the coil current performing the forward kinematics of the device. This control loop is actually an own application and runs on a separate computer like already explained in the previous section. It runs at a frequency rate of 1000 Hz in order to provide stable flotor control.

Calculating the currents for the coils is actually not very computationally intensive and since it is the only calculation that has to performed on the controller computer, which is a pentium 2.4 GHz, the execution time is not critical. It was actually tested that this control loop can easily run at 4000 Hz.

In order to calculate the coil currents the control loop must receive the force information from an additional application, which is the physical simulation of the virtual environment. The two applications communicate via Remote Procedure Call (RPC), where the device control loop employs a call back function which is executed regularly at 1000 Hz. This call back function executes the physical simulation for one time step and sends back the force information to the device controller.

In order to provide a realistic feel of the virtual environment the physical simulation for one time step must be executed within 1 ms, which can be a problem as the simulation of the virtual world gets complicated.

As can be seen from figure 6 the graphical scene rendering of the virtual environment and the physical simulation run on the same computer, which is a powerful pentium dual core with 3.2 GHz. The setup has been chosen this way to completely separate client application from low-level hardware control.

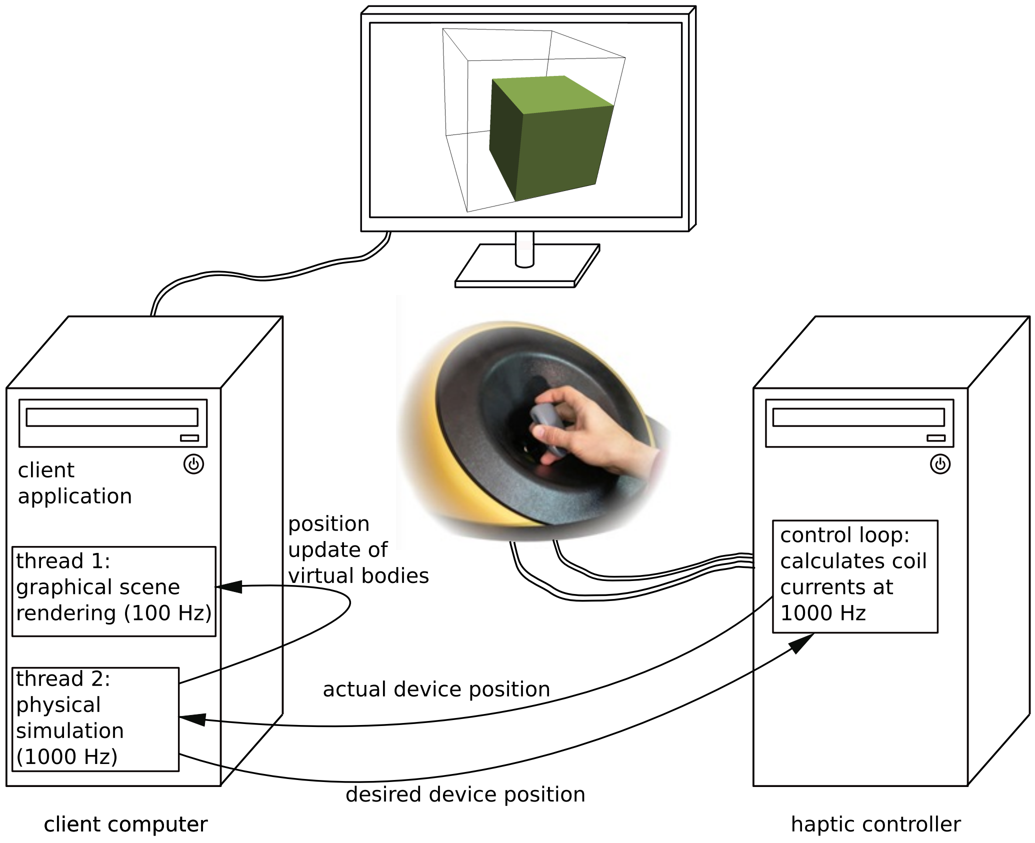

client application
thread 1: graphical scene rendering (100 Hz)
thread 2: physical simulation (1000 Hz)
position update of virtual bodies
actual device position
desired device position
control loop: calculates coil currents at 1000 Hz
client computer
haptic controller

Figure 5: Setup of the maglev haptic device.

4 The Cube in Cube Demo

The "Cube in Cube Demo" simulates a cube whose motion is constrained by the walls of a larger box. By hitting the virtual walls of the outer cube, forces and torques are generated which are applied to the flotor. This demo has already been implemented by Berkelman in 1998 on the first generation haptic device. At that time the demo was implemented by using force control, which means that the hardware controller of the haptic device gets the flotor force directly from the client program. However, to increase stability, all the demos on the second generation Maglev have been implemented using the so called virtual coupling method [14], in which the position of the flotor is controlled and not the force applied to it. This guarantees stability even in the case of communication delays between client program and hardware controller. By doing so, it is obvious that the way forces are calculated and applied to the flotor changes dramatically. In the next paragraphs it will be explained how this demo has been implemented and most importantly how the virtual coupling method can be applied to this case.

The green cube is floating in free-space and can be moved within the box without any constraints and, since the flotor is not connected to any mechanical device, this feels rather smooth and light. As soon as the cube touches the walls of the bounding box, forces and torques are calculated in x, y and z direction according to the penetration depth of the cube and the box. The forces and torques are calculated to simulate real behavior as accurate as possible. The penetration of the cube's vertices into the walls of the box are multiplied by the proportional gain of the PD controller. The resulting force, which subsequently will be applied to the flotor, is calculated by taking the sum of the single forces coming from all penetrations of the vertices in the x, y and z direction. To simplify the problem only point-penetration is considered, that means that only the penetration depth of the vertex is considered for calculating forces and torques.

For the case depicted in figure 8 the flotor force would be calculated as follows:

$$F_{1X} = g_X P_{1X} \qquad F_{4X} = g_X P_{4X} \tag{4.1}$$

$$F_X = F_{1X} + F_{4X} \tag{4.2}$$

$$F_y = F_{1Y} = g_Y P_{1Y} \tag{4.3}$$

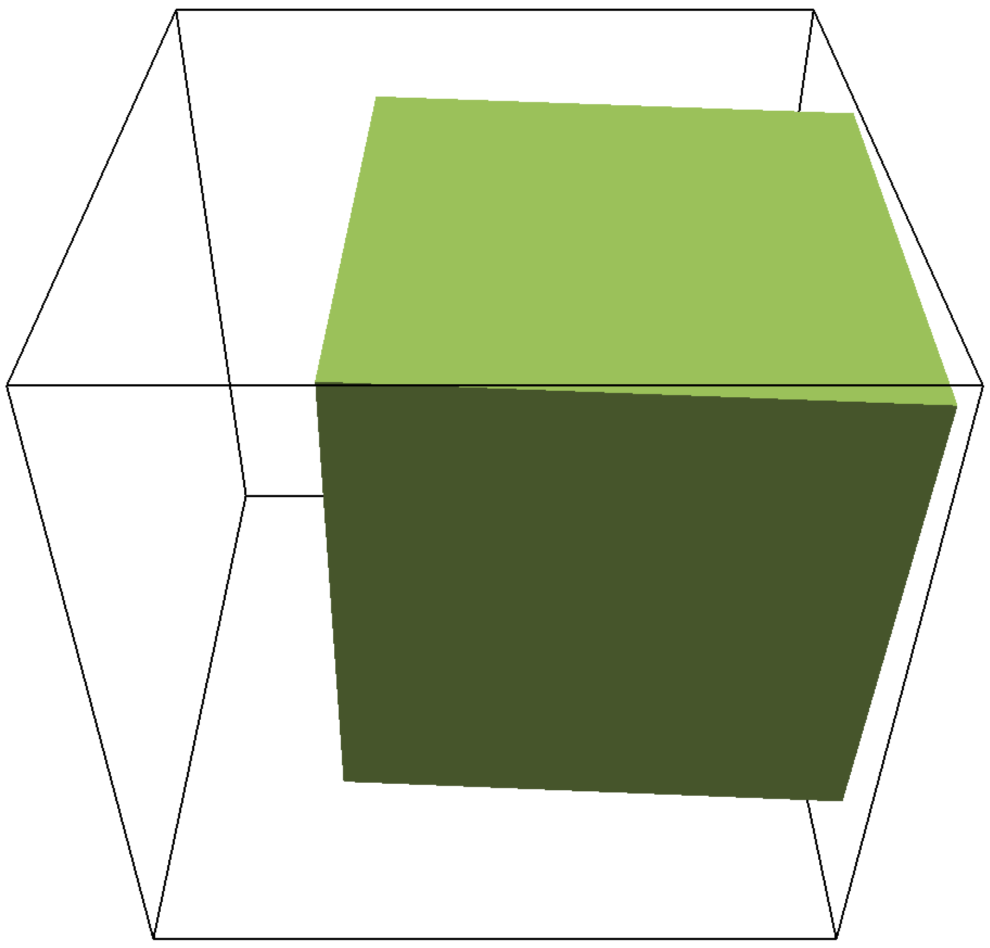


Figure 6: Screenshot of the "Cube in Cube Demo".

Torques are generated when the inner cube is touching the walls of the box in a tilted pose. They are calculated by multiplying the single forces in the vertices by their corresponding lever, which is the distance from the force's action line to the center of gravity of the cube.

$$T_Y = F_{1X}L_{1X} - F_{4X}L_{4X} - F_{1Y}L_{1Y} \tag{4.4}$$

In the end all the torque components in the corners of the inner cube sum up to the total torque sent to the haptic device. Note that in the depicted case (figure 9) only the vertices of the cube's front face are considered for calculating the torque. The forces F_{1X}, F_{1Y} and F_{4X} also generate a torque component on the x and z-axis. The formulas for calculating those are not stated here because

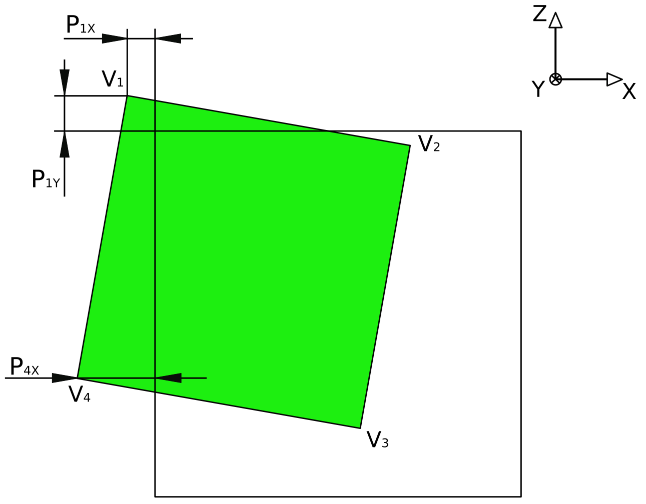


Figure 7: Translational forces.

the levers for the x and z axis cannot be seen in the view given by figure 9.

To calculate the positions of the vertices in each simulation loop a homogenous transformation has to be performed. This homogenous transformation can be determined by knowing the pose of the flotor in 3D space (X,Y,Z, α, β, γ).

$$A = \begin{bmatrix} \cos\alpha\cos\beta & \cos\alpha\sin\beta\sin\gamma - \sin\alpha\cos\gamma & \cos\alpha\sin\beta\cos\gamma + \sin\alpha\sin\gamma & X \\ \sin\alpha\cos\beta & \sin\alpha\sin\beta\sin\gamma + \cos\alpha\cos\gamma & \sin\alpha\sin\beta\cos\gamma - \cos\alpha\sin\gamma & Y \\ -\sin\beta & \cos\beta\sin\gamma & \cos\beta\sin\gamma & Z \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{4.5}$$

To make the user think the cube is weightless a feed-forward force in the vertical direction is always applied to the flotor. Nevertheless, this feed-forward force is not able to make the flotor feel inertia-free.

4.1 Balancing inertial effects of the flotor

The magnetic levitation haptic device is frictionless since it is floating in a magnetic field, but unfortunately the flotor has a small inertia, which contributes to

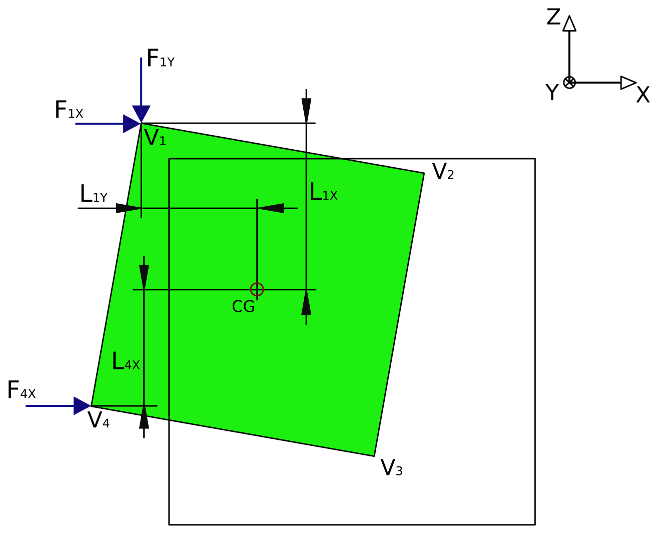


Figure 8: Scheme illustrating how torques are calculated.

the user's perception of the virtual world to a none negligible extend, particularly when the flotor's acceleration is significant. For these situations an balancing inertial effect might be desired.

There are basically two different ways to cancel the flotor's inertia. The first solution to this problem is to mount force sensors on the flotor. The difference between the forces applied to the flotor coming from the device, and the forces measured between the flotor's handle and its body, form the force coming from the acceleration of the flotor. This component has to be added to the force which is applied to the flotor in the subsequent time step.

$$F_t^F = F_t^H - F_t^{VE} \tag{4.6}$$

$$F_{t+1}^{VE'} = F_{t+1}^{VE} + F_t^F \tag{4.7}$$

F_t^F is the force needed to accelerate the flotor at time step t. F_t^{VE} denotes the force coming from the physical simulation of the virtual environment. $F_t^{VE'}$ is the force which finally has to be applied to the flotor in order to cancel its inertia.

This method seams to be straight forward, however it is not very easy to install force sensors at the flotor so that forces and torques can be measured in all three space dimensions.

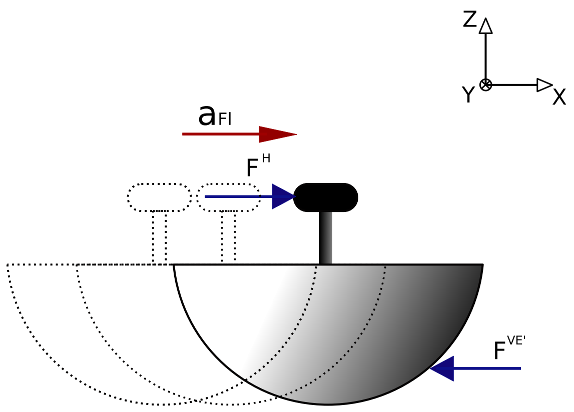


Figure 9: Balancing inertia of the flotor with force sensors.

A much easier solution to this problem does not involve any hardware changes of the device. By measuring the acceleration of the flotor and using Newton‘s law, one can determine what additional force has to be applied in order to balance inertial effect. However, for this method the inertia tensor of the flotor has to be determined as accurately as possible, which can be very difficult, since the flotor has a complicated geometry and its density is not homogenous.

$$\vec{F}_{xyz} = \vec{I}_{xyz}\vec{a}_{xyz} \tag{4.8}$$

$$\vec{T}_{\alpha\beta\gamma} = \vec{I}_{\alpha\beta\gamma}\vec{\dot{\omega}}_{\alpha\beta\gamma} \tag{4.9}$$

Unfortunately this method involves a further problem. Since there are no accelerometers installed on the flotor, the acceleration must be calculated by deriving twice the position data. The position data is noisy and deriving it twice can lead to an unacceptable amplification of this noise, which makes an accurate calculation of the flotor's inertial force problematic. Strong vibrations of the flotor can be the consequence.

This problem can be solved by filtering the position data, which has the disadvantage that it leads to a delay of the flotor's response. The more values are taken in consideration for determining the actual pose of the flotor, the more the update frequency of the flotor force decreases, and it reaches very soon an acceptable value. To compensate for this effect the control loop of the haptic device must run at a higher frequency. We encountered severe limitations in increasing the control loop frequency above 1000 Hz, because the communication between controller and client computer was not fast enough to run at more than 1000 Hz. A solution to this problem is to run the control loop at a higher frequency, let's say 4000 Hz, then apply a moving average filter with four values of the position data on the controller side and send the position information to the client computer with an update rate of 1000Hz. The forces of the device are still just updated at a rate of 1000 Hz, but the position data is filtered with 4 values which decreases noise significantly.

4.2 Torque balance

The forces which are applied to the flotor coming from the interaction of the coils and the magnetic field are not passing the center of the handle, which would be approximately the point where the forces applied by the user's hand are going through. When the user applies a force to the handle and receives a reaction force from the device, an unwanted torque is formed due to the misalignment of the two forces, also known as a force pair. This torque causes the flotor to tilt even if the user doesn't want the flotor to behave like this. In other words, when the user wants to push the flotor in a certain direction and the device responds with the same amount of force in the opposite direction, the flotor will be additionally tilted around the axis 90 degrees to the direction where the forces are applied. Figure 11 illustrates this effect with forces applied in x-direction causing a rotation of the flotor around the y-axis.

This effect can be counteracted by applying the same torque in the opposite direction. To implement this, the lever of the force pair has to be known. As a sufficient approximation, we consider the lever being the distance between the point where the forces coming from the magnetic field are passing through and the center point of the handle. This approach is only valid for slow motions, since the inertia of the flotor is not taken in consideration. A better model would have to include an inertia balance such as explained in the previous section. After some trial and error it was found out that $13.3cm$ is a good value for the lever L_{Fl}. However, in practice this value changes with the pose of the flotor.

$$T_y = F_x h \tag{4.10}$$

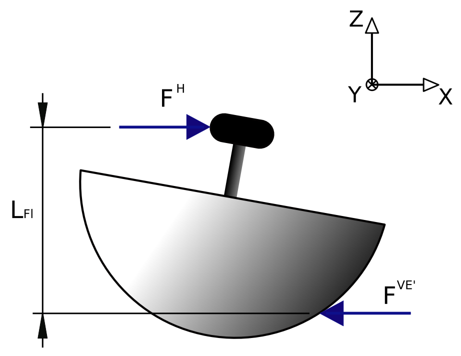


Figure 10: Image of tilted flotor with forces and torques.

$$h = 13,3cm \tag{4.11}$$

The CubeInCube demo feels much more convincing with this torque balance. When the cube is pushed horizontally against one of the walls of the bounding box, the flotor stays in an upright position and the user feels a pure horizontal force.

4.3 Gaining stiffness by means of graphical cheating

The forces coming from an interaction of two bodies are calculated using the impedance control method. Forces are proportional to the penetration level and for this reason rigid bodies must be allowed to penetrate into each other in order to get a measure of the force which has to be applied to them.

If the position of the cube given to the graphical representation is the real position of the flotor in 3D space, it can be seen by the user that the two rigid bodies, in this case the inner cube and the walls of the bounding box, can penetrate into each other. This looks very unnatural since the bodies do not deform while penetrating. A common method to "solve" this problem is by manipulating the actual pose of the two objects which are colliding. In our case the outer box is rigid and therefore only the inner cube's pose has to be modified before updated. The modification of the pose of the flotor has the purpose to give the impression that the cube is not able to penetrate into the walls of the bounding box. This is not a trivial problem, since rotation and translation of the cube are interconnected. There are actually infinitely many different combinations of rotation and translation which bring the cube to the borders of the bounding box. Due to these complicated circumstances we limited our modification on the translational part of the cube's pose. For doing so the maximum penetration in each direction x, y and z is taken to translate the cube accordingly. The following figure 12 shows the basic concept of this position modification.

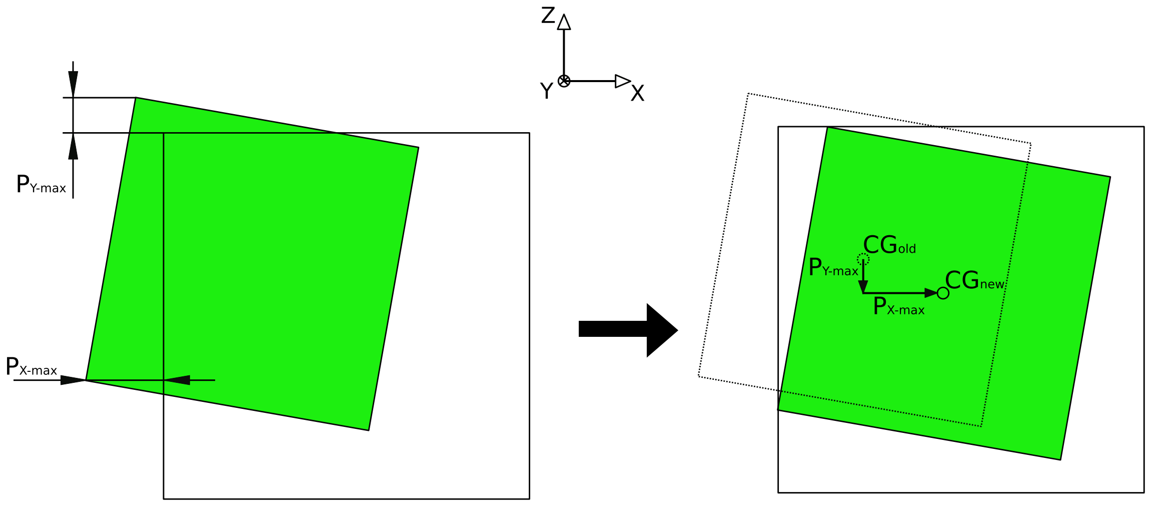


Figure 11: Graphical cheating.

It is very surprising how this little hack changes the user's perception of stiffness of the device. Walls are really felt as being rigid and not soft, as they actually are, due to the limited stiffness of the device.

5 The Texture Demo

5.1 Introduction

Simulating textures has been a main research area in the field of haptics for almost 2 decades. Many different models have been implement to simulate real texture as accurately as possible. One main issue in haptic texture rendering is the limit on the stiffness. It is very hard to have stable behavior of the haptic device when the virtual surface is very rigid. This is caused due to the fast changes of the force applied to the haptic device [16], [17]. Therefore the model representing the texture must chosen carefully.

This chapter explains the implementation of 2 different models simulating a texture plate with three different sine waves and one smooth surface. Additionally the impact of rigid bodies has been simulated using a feed-forward impulse.

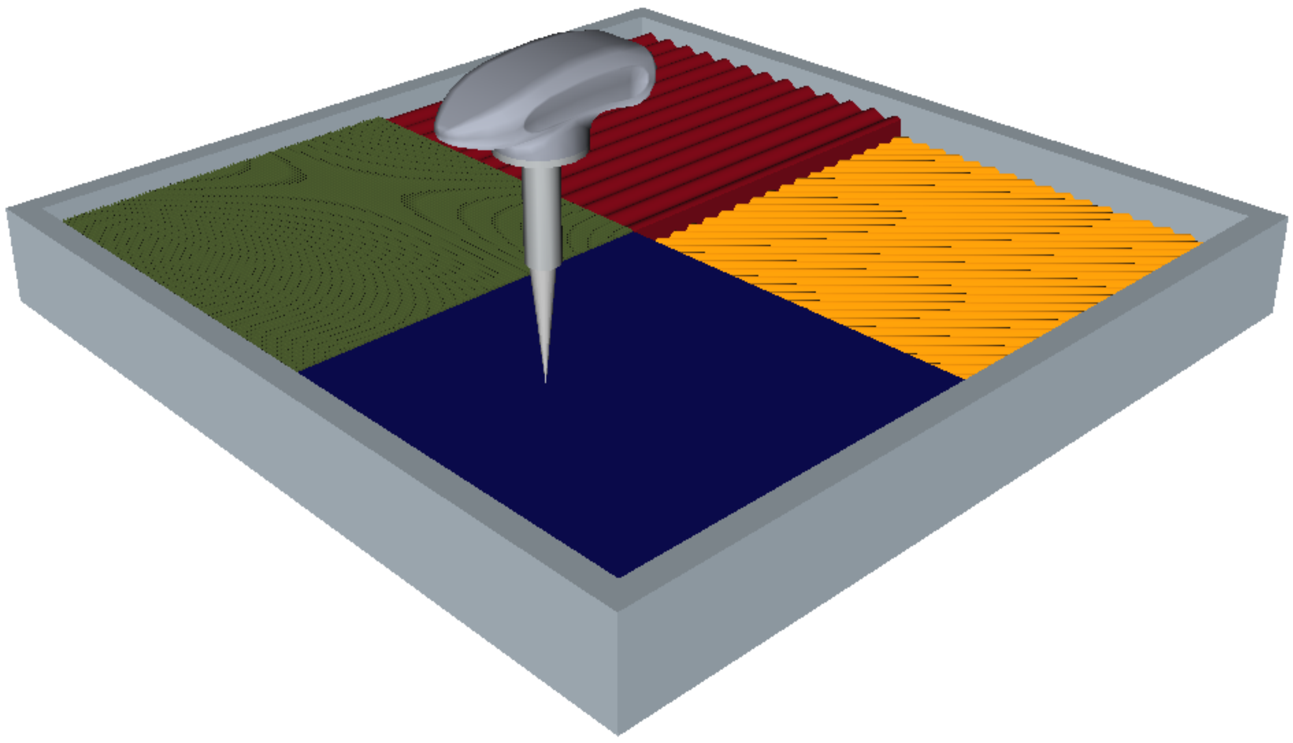


Figure 12: Screenshot of the "Touch the Bunny Demo".

We started from a CAD model of a texture plate which has 4 equal sized squares and each square has a different texture pattern (figure 13). Collision detection can be done with two different approaches in this case. The first approach would be a numerical method, where the triangular mesh structure of the CAD model is used in order to determine penetrations. This is a general way of solving the problem of collision detection with complicated bodies. It has the advantage that it can be applied to any geometry formed by a triangular mesh and is very flexible therefore.

The second and more customized way of solving this problem is considering the mathematical representation of a sine wave.

$$h = A\sin(\frac{x + x_{off}}{P}2\pi) + y_{off} \tag{5.1}$$

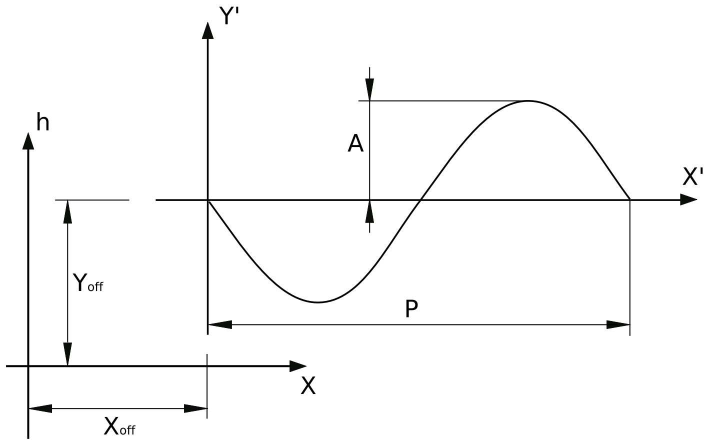


Figure 13: Parameters of the sine wave.

With this approach collision detection can be performed analytically by checking the position of the haptics interaction point with respect to the sine curve. We decided to implement this last method since it is very easy to get a first approximation of the forces coming from the penetration of the probe into the sine wave, by just considering the vertical penetration. This rather simple approach only involves forces in vertical direction. The penetration depth can be determined by simply taking the difference of the height of the virtual interaction point and the height of the value of the sine curve in that specific x-y position. This distance is multiplied in a second step by a proportional gain which gives the force applied to the flotor in z direction.

$$F_v = p_z(h - fl_z) \tag{5.2}$$

F_v is the vertical force applied to the flotor, p_z is the proportional gain for the PD-controller for the vertical axis and fl_z is the actual z-position of the flotor.

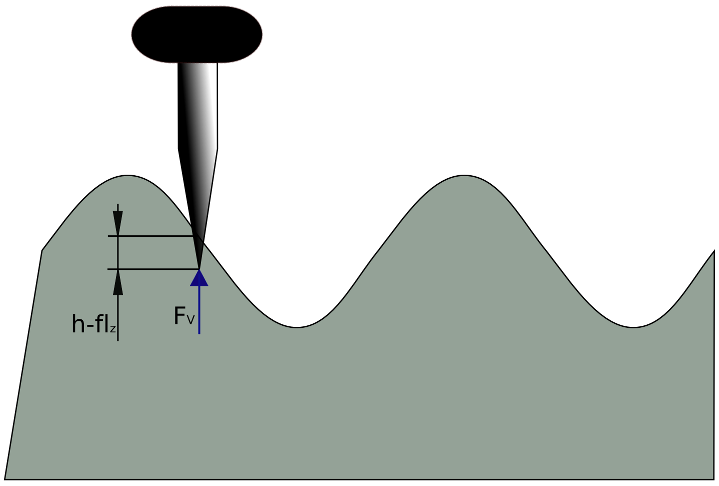


Figure 14: Vertical penetration.

5.2 Calculating flotor forces in vertical and horizontal direction

The approach for calculating the flotor force is very simplistic and is not able to give a very realistic behavior of the flotor. The reason for this is the fact that in reality, also forces in horizontal direction are present which cause the flotor to move along the x and y direction.

It turns out that calculating the horizontal forces for our specific case is not an easy task. An approach published by Basdogan et al [22] presents an approximated method which explains the calculation of forces when penetrating into a differentiable surface. In the first step the gradient of the sine curve is calculated in the point where the haptics interaction point is located. By knowing the gradient, the horizontal component of the force F_h can be determined as follows.

$$h = A\sin(\frac{x + x_{off}}{P}2\pi) + y_{off} \tag{5.3}$$

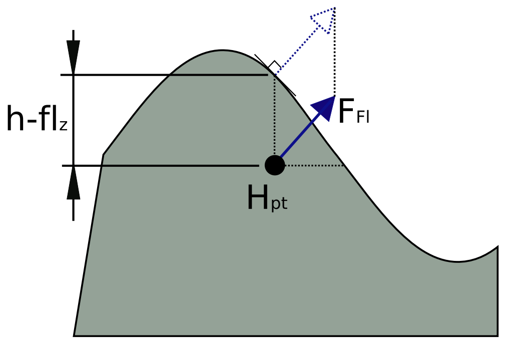


Figure 15: Model for calculating force direction, Basdogan et al [22].

$$h' = A\cos(\frac{x + x_{off}}{P}2\pi)\frac{2\pi}{P} \tag{5.4}$$

$$F_v = p_z(h - fl_z) \tag{5.5}$$

$$F_h = \frac{F_v}{h'} \tag{5.6}$$

A graphical representation of this gradient method is illustrated in figure 16. It is a rather simplified approach and can lead quickly to wrong results as we consider the following case: Unfortunately in some cases this approach is not very accurate, especially when the stiffness of the haptic device is small and haptic interaction point is allowed to penetrate the virtual body by a significant amount as we see in the following example.

Suppose the haptic interaction point is moves along a trajectory which is inclined 45 degrees with respect to the horizontal line and then hits the surface of the sine wave as can be seen in figure 17. Since we are using the impedance control method the virtual probe point is actually penetrating the sine wave.

If the point continues penetrating the sine wave, the force calculated with the gradient method changes direction and it finally ends up being totally vertical (last image of figure 17). The direction of the force should actually not change in this example because in reality we have to consider that we do not penetrate the sine wave, but rather deform it at a very small scale.

A more accurate model for calculating the flotor force considers the sine wave to be a soft object. By seeing the sine wave as a deformable body, we can imagine its behavior when touched by a point as depicted in figure 18.

Note that this model is only valid if friction is not neglected. Otherwise the

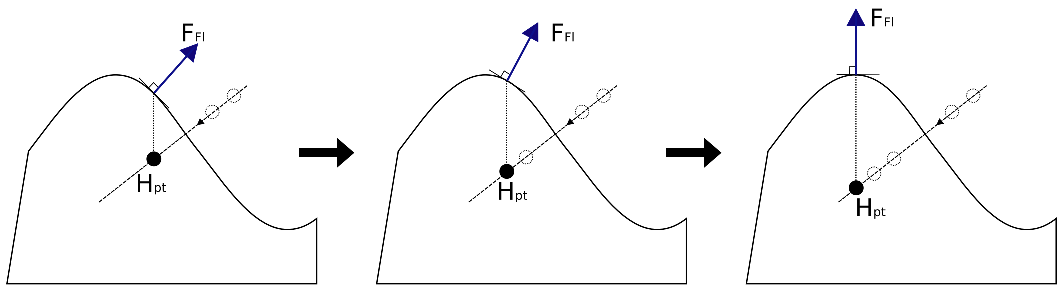


Figure 16: Sequence showing that the gradient method is imprecise in certain cases.

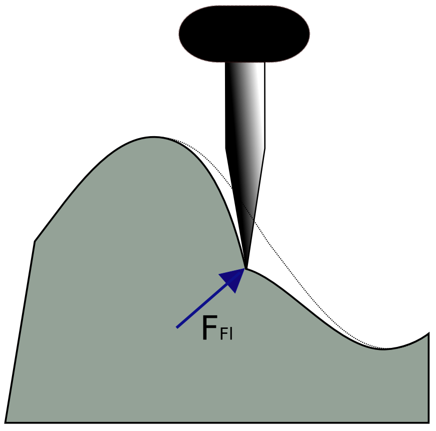


Figure 17: Deformable sine wave.

model gets much more complicated because the deformation of the sine wave also depends on the direction of where the probe is coming from.

To calculate the flotor force, the penetration depth d has to be determined, which is the smallest distance of the probe point to the surface of the sine wave. By definition, this distance d lies on a normal of the sine curve. It is not trivially easy to calculate this distance since no closed-form solution can be applied.

The normal of the sine curve is obtained by rotating its tangent by 90 degrees.

$$y' = \cos(x) \tag{5.7}$$

And therefore the slope m of the normal is given by

$$m = -\frac{1}{\cos(x)} \tag{5.8}$$

The normal represented as a line is given by

$$y = -\frac{1}{\cos(x)}x + b \tag{5.9}$$

This line has to pass through the point of the probe, therefore:

$$y_p = -\frac{1}{\cos(x)}x_p + b \tag{5.10}$$

The y-axis intercept b can be written as

$$b = \sin(x) + \frac{x}{\cos(x)} \tag{5.11}$$

Eqs. 5.10 and 5.11 give the final result

$$y_p = -\frac{x_p}{\cos(x)} + \sin(x) + \frac{x}{\cos(x)} \tag{5.12}$$

This last equation only contains the unknown variable x, which represents the x-coordinate of the intersecting point of the normal passing through the haptic interaction point with the sine curve (figure 19).

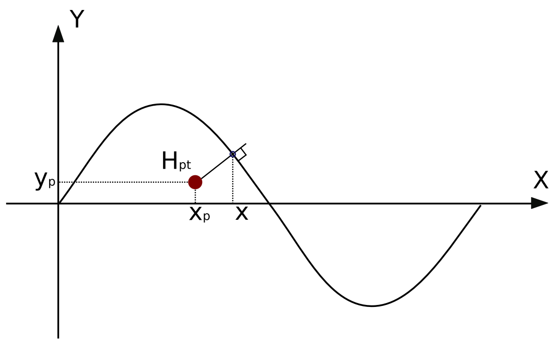


Figure 18: The shortest connection between the haptic interaction point and the sine curve is normal to the curve.

As it is clear now from equation 5.12 an iterative method has to be used in order to determine this intersecting point. Such an approach however, is computationally not efficient, which could be a problem, since calculating the forces for the flotor has to take less than 1 ms.

An alternative to the iterative method is the use of a look-up table. This approach is especially convenient for our case, because the sine wave is a repeating pattern. Only one period of the values of each sine wave has to be stored in advance in order to be able to perform collision detection. It is self-explaining that the more values our look-up table contains, the better the shape of the sine curve will be preserved and this results in a smoother behavior of the haptic device when moving across the texture patterns. By using more points in the look-up table almost an arbitrarily fine sine curve can be represented. However, it has to be kept in mind that a bigger look-up table implies the need of doing more comparisons during each simulation loop. For each point in the look-up table which is possibly the closest point to the probe, a series of calculations have to be performed in each simulation loop in order to calculate their distances from the haptic interaction point. Note that these calculations are all operations which are not computationally expensive compared to calculating trigonometric functions - which would be required by an iterative method.

A huge amount of points in the look-up table could lead to unstable behaviors hindering the control loop to run at 1000 Hz. In our case it was found out that a table with 10000 values leads to a very smooth surface and the computation time of the simulation loop is less than 0.05 ms on a Pentium dual core with 3.2 GHz. Alternatively using less values and then applying an interpolation to smooth the surface would give the same results.

For determining the closest point on the surface the distance of the considered points to the virtual probe point have to be calculated. It is actually easier to calculate the square of this distance.

$$d_i^2 = (x_p - x_i)^2 + (y_p - y_i)^2 \tag{5.13}$$

The number of points which are checked in every loop are fortunately not all 10000. It can be seen from figure 20 that in the worst case, half of all the points in the look-up table have to be checked for being the closest point to the probe.The interesting section of the look-up table depends on the position of the actual probe with respect to the sine wave.

There are two points (x_1, x_2) to be determined which describe the interval of the interesting points. One of these limiting points will always be the x-position

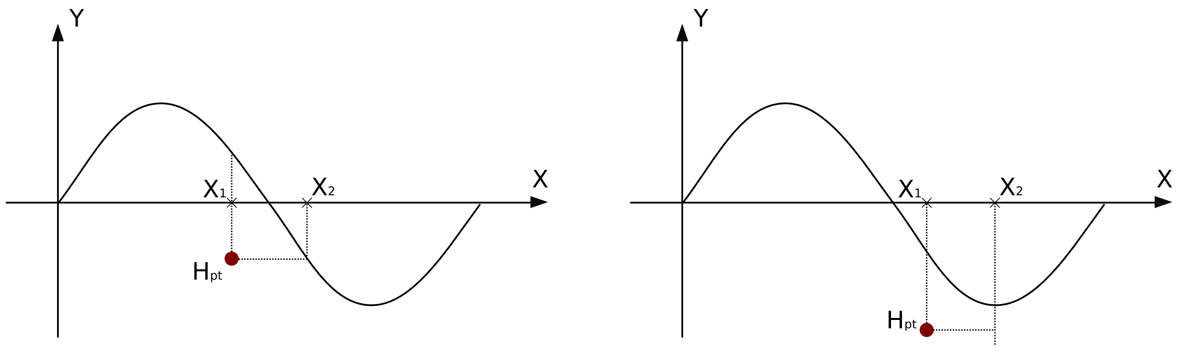


Figure 19: Image showing sine wave with different sections and possible points of being the closest.

of the probe point and the other one can be calculated by taking the *arcsin* of the probe's y-position.

$$x_1 = x_p \tag{5.14}$$

$$x_2 = \arcsin y_p \tag{5.15}$$

Since the look-up table only contains the values of a sine curve within one period, there arises a little problem when the probe is located between 0 and $\frac{\pi}{2}$ because the limiting value x_1 would be negative. The solution to this problem is to calculate the corresponding value for x_1 in the sector $\frac{3\pi}{2}$ and 2π as it is depicted in figure 21. The same happens if the probe point falls between $\frac{3\pi}{2}$ and 2π.

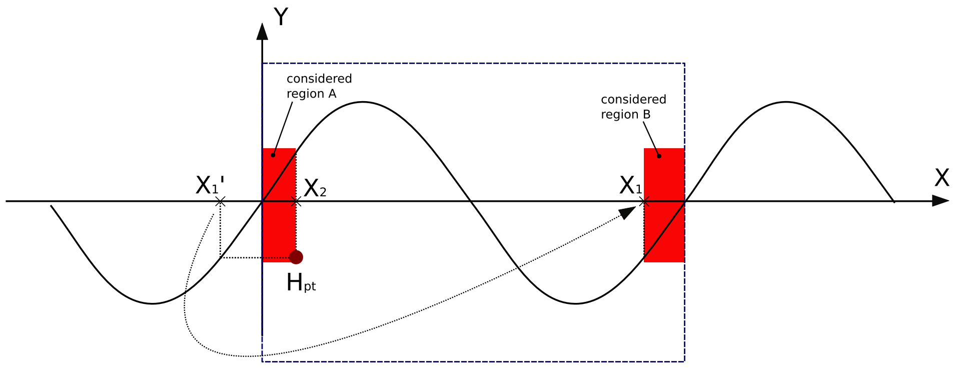


Figure 20: A point on a sine wave can be easily ported to its equivalent in a different period.

After determining the closest point on the sine wave, the force intensity is calculated by multiplying its distance from the probe point by the proportional gain of the PD controller. The force direction is given by the direction of the line connecting these two points, or can also be calculated by taking the normal to the tangent of the sine curve in the intersection point.

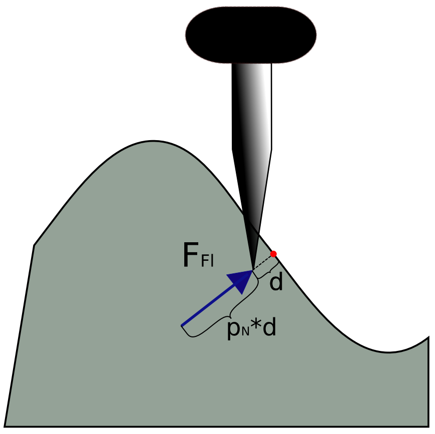


Figure 21: Image showing how the force is applied to the flotor.

5.3 Impact force

In spite of the remarkable high stiffness of the Magnetic Levitation Haptic Device, surfaces are not perceived as being totally rigid. No crisp impact is noticed when two hard bodies hit each other. When the probe is lifted up and tapped on the surface of the texture plate, the haptic device responds with the stiffness of 25 N/mm, which is actually way below the stiffness which one would encounter when tapping a hard surface in reality. For instance a surface made of steel would provide a stiffness around 1000 N/mm. Such a high value leads to vibrations of the body which in turn causes the sound of the impact. One main reason for these vibrations is the deformation of the body itself. It will be compressed and decompressed at a very high frequency. Different materials sound differently when hitting each other because their vibrational behavior depends on the specific material properties and most importantly the stiffness. Since the stiffness of the magnetic levitation haptics device is way below the stiffness of real rigid materials, a vibration of the flotor such as described above cannot be generated by using the impedance control method. However, by modifying the force output of the controller, impact behavior of rigid bodies can be approximated. We did this by applying a feed-forward force to the flotor whenever the virtual probe hits the texture plate.

The feed-forward force is proportional to the cube of the impact velocity.

$$F_{ff} = 50000 v_{imp}^3 \tag{5.16}$$

This particular function has been chosen, to make the difference between hitting the surface harder and lighter more evident. It is for sure just an approximation of a real impact, which is actually very complicated to model accurately. Nevertheless, it turned out that this very simple approach provides a very realistic feel and sound when the virtual probe is tapped against the texture plate.

By changing the feed-forward force, or even applying a profile of feed-forward forces, different sounds can be generated. This has been implemented by Kuchenbecker et.al [18].
The high feed-forward force can only be applied in the moment when the probe hits the texture plate. To avoid that this impulse is triggered while the probe is moving along the surface, it is essential to look at the history of the flotor's position. The condition for an impact force is that the probe penetrates the texture plate in the current loop, but did not touch it in the loop before. However, this is not enough to ensure a realistic and stable behavior of the flotor. Due to signal noise it happens that the flotor changes between the two states of being inside and outside the texture plate very quickly, sometimes even every simulation loop. As a consequence the impact force is turned on and off many times consecutively, which lead to vibrations of the flotor. Another reason for instabilities of the flotor is the fact that the high feed-forward force (up to 120N) pushes the flotor outside of the texture plate. Due to gravity and the users force applied to the flotor, the handle is pushed back into the texture plate, which triggers an impact force. This happens at quite high frequency 100-200 Hz and makes the flotor vibrate.

To overcome both problems just explained an additional timer has to be implemented, which forbids a frequent repetition of the impact feed forward force within a short period. After some trials it turned out, that a timeout of 70 control loops (70 ms) is an appropriate way to get rid of the vibrations.

However, this "hack" can cause an unrealistic behavior of the flotor. If the user is fast enough to make the flotor touch the surface twice within 70 ms, then no impact force would be applied at the second contact. An average person is not able to move its hand actively at a frequency bigger than 5 Hz. This frequency decreases if an object with significant inertia is hold in the hand, which applies to our case. However, the combination flotor and hand is not an active nor a passive system, but rather a mixture of both, since the flotor is controlled by two forces: the one generated by the magnetic field and the one generated by the users hand. A system is considered passive when the user doesn't actively apply a force to the flotor. In that case much higher frequencies can be accomplished. If the users force is governing the flotor then very low frequencies such as 5 Hz cannot be exceeded.

This behavior can be better explained considering a drummer. When doing a roll on a snare drum, the drummer holds his sticks very gently and barely applies a force, and most of the actuation force of the drumstick comes from the drum skin. This is considered to be a highly passive system. If the drummer tries to move the stick in free space, without hitting the drum skin, much lower frequencies can be achieved, because the system now is active.

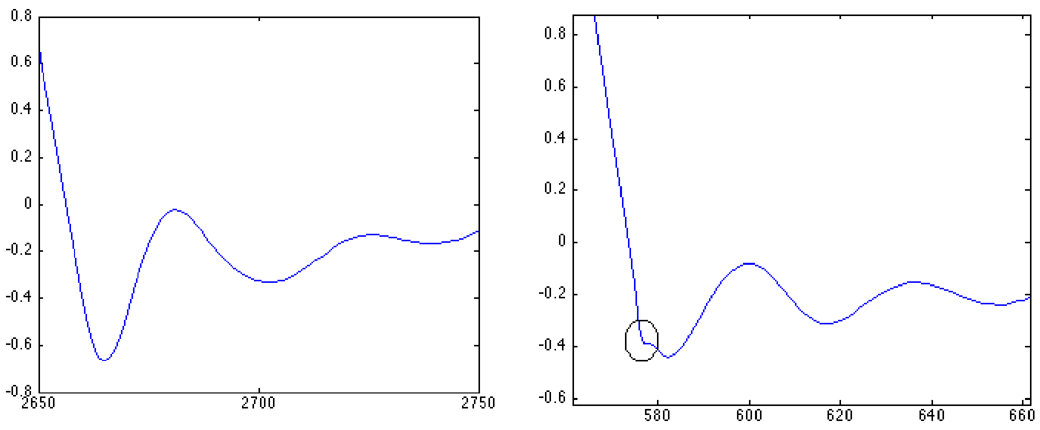


Figure 22: Flotor position with and without impact force

Figure 23 shows the flotor position in z-axis when the flotor is tapping the surface. The left image shows the flotor's position while the feed-forward force of the impact is turned off. The right image represents the case with the impact force turned on. The encircled region clearly shows a vibrational motion with relatively high frequency and low amplitude. This fairly short vibration causes the crisp sound which is not distinguishable from the impact of real rigid bodies.

The signal is very ragged because the sampling frequency of 1000 Hz is to low to represent such a high-frequency vibration.

5.4 Collision detection of bodies with non-differentiable surfaces

Surfaces which are not differentiable in certain areas lead to a problem in collision detection. For example lets consider the 90-degrees edge of the texture plate as depicted in figure 24. When sliding the probe along the horizontal part of the surface the vertical force which is applied to the flotor is calculated with the penetration depth d of the virtual probe. Unfortunately the distance d is not unique as can be seen in figure. Independent from the position of the probe there will be always two possibility for choosing d and unfortunately selecting the smaller one is not the right approach as it becomes clear in figure 24. Suppose we are touching the vertical part of the surface just slightly underneath the horizontal plane. As we penetrate more and more, the distance to the horizontal surface becomes smaller than the one to the vertical surface (last image of figure 24). By choosing always the smaller distance, we would suddenly get a harsh force in vertical direction which pushes the flotor out of the horizontal plane.

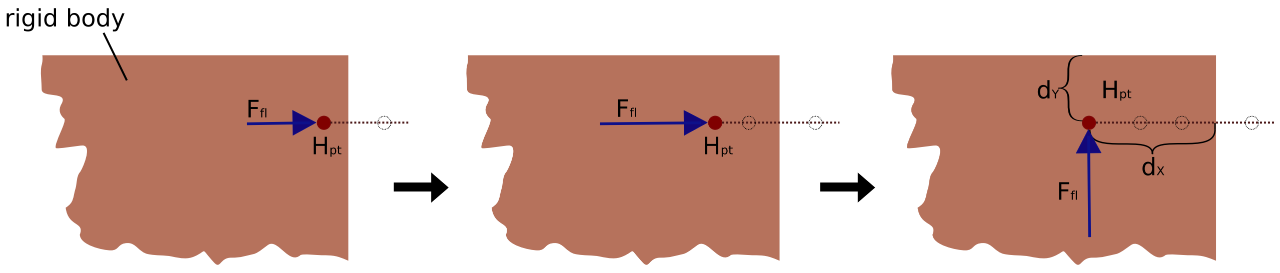


Figure 23: In the depicted case the penetration depth is not unique

This is an unwanted behavior and furthermore it leads to instabilities when the distance d changes between the horizontal and the vertical component. The harsh switch of force direction leads to vibrations of the flotor. A way to solve the problem of collision detection with locally non-differentiable surfaces is to keep track of the probe's position. The point where the probe first touches the surface determines the force applied to the flotor. If the probe point penetrates the side wall first, a flag is set and no matter how much the probe point penetrates the sidewall, the direction of the force applied to the flotor will not change. The flag will be reset to zero, when the point exits the surface on one of the sides.

This is actually very similar to the real interaction behavior of a point and a soft body. Even if we are trying to simulate interaction of undeformable objects it is not possible to get such behavior since the stiffness of the haptic device is limited.

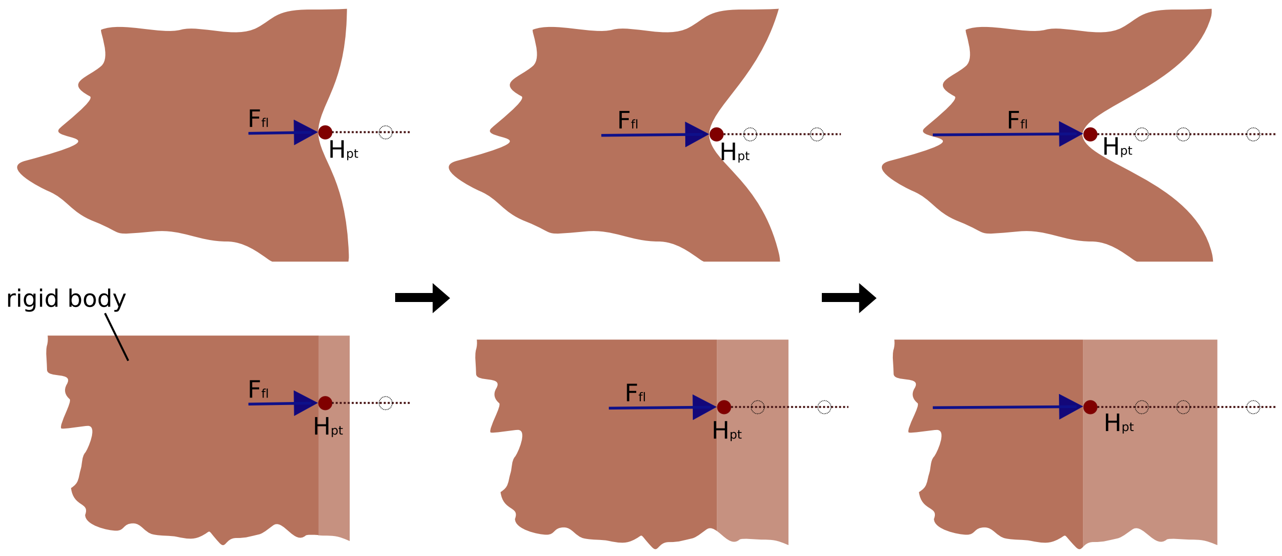


Figure 24: Probe penetrating a deformable body

Figure 25 shows the deformation of the soft object while the probe point pushes against its side wall. In this situation, if the probe would be moved out of the body in horizontal direction, it would return to its previous form. Now consider the case where the probe moves out of the body in vertical direction. As soon as the probe is outside the body the penetration flag is set to zero and the body could now be penetrated in the vertical direction.

The approach for collision detection with locally non-differentiable surfaces has been explained with a very special example, namely two flat surfaces which form an angle of 90 degrees. In this case it is straightforward to distinguish between the different penetration cases, since one is in vertical and the other one in horizontal direction. What happens if there is a combination of flat surfaces which are inclined less than 90 degrees or what if these surfaces are not flat but rather of class C1 or higher? Consider the following examples (figure 26):

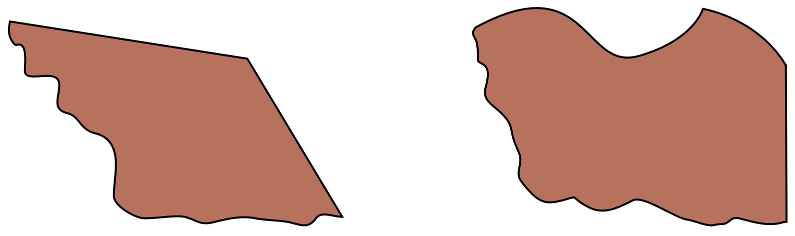


Figure 25: Examples of arbitrary connections

A solution for determining the force to be applied to the flotor can be found by generalizing the approach for the flat surfaces forming an angle of 90 degrees. By drawing normals to the tangents of the two surfaces at their connection(figure 27), an area can be individualized. Whenever the point of the probe is located within this area the direction of the force will depend on the history of the probe's position, e.g. the part of the surface which the probe point crossed most recently determines the force.

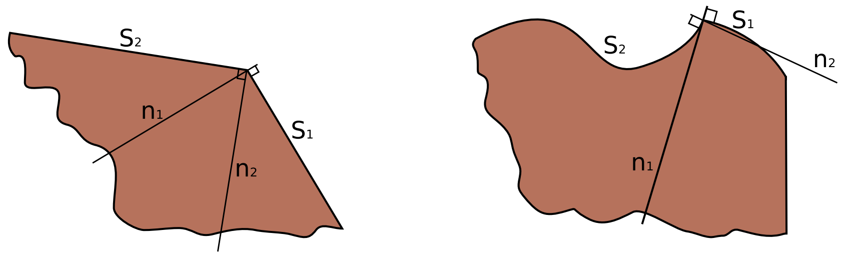


Figure 26: Image of the generalized case

In the case where two flat surfaces form a right angle, the area where the force depends on the history of the position is limited by the two surfaces itself, since the normal of the tangent of the horizontal surface lies on the plane of the vertical surface and vice versa.

5.5 Instabilities in the corners of the texture plate

In the corners of the texture plate instabilities of the flotor are experienced which tend to grow if the flotor is lifted up. The flotor is close to the workspace boundaries in the edges and comes even closer when it is lifted up. The position signal of the photo diode arrays has some noise which propagates through the calculation of the flotor position in 3D space. It has been measured that the standard deviation of the noise is around $1\mu m$ when the flotor is positioned at the center of its workspace. As the flotor is located further away from this center point, the noise of the position data increases and its standard deviation climbs up to $4\mu m$ when the flotor is close to the workspace boundary. This increasing signal noise causes the instabilities of the flotor in these extreme points. By lowering the gains of the PD controller these vibrations can be avoided.

6 Touch the Bunny Demo

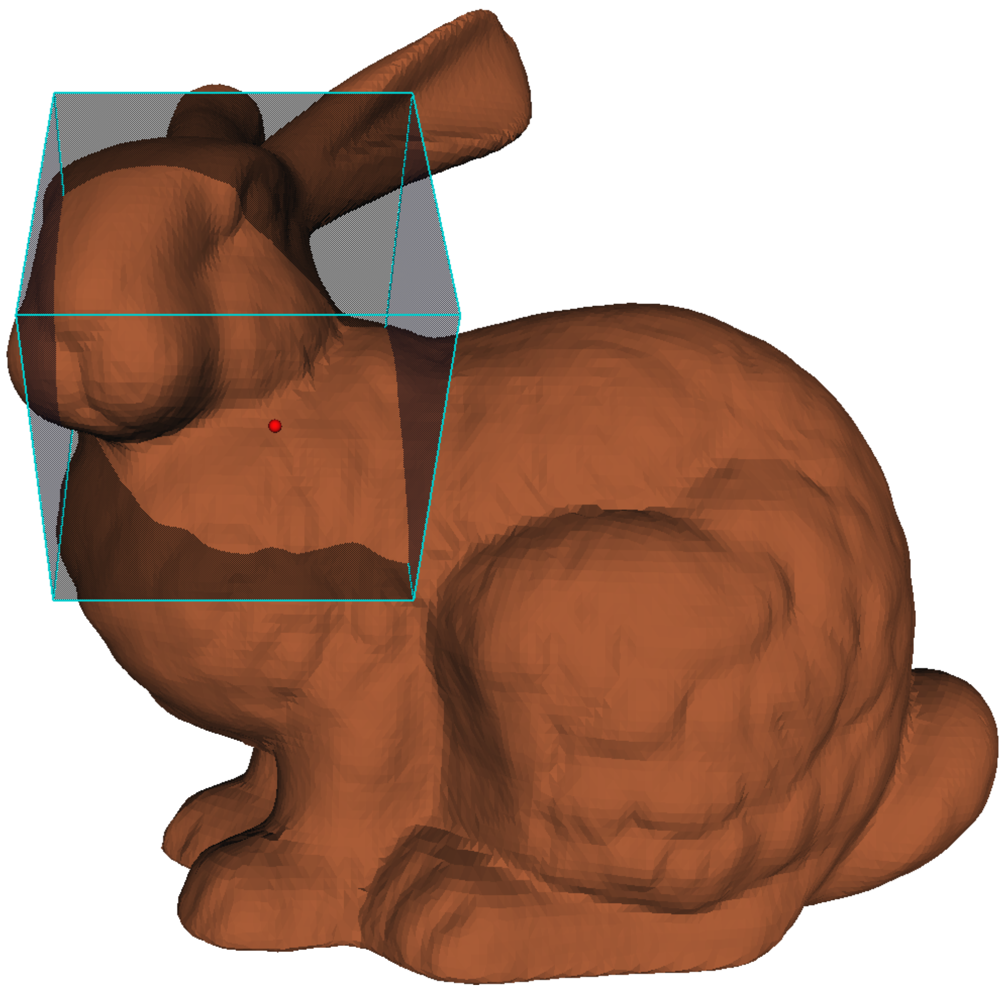


Figure 27: Screenshot of the "Touch the Bunny Demo".

This demo has been implemented to demonstrate haptics interaction of bodies with complex geometry. A triangular mesh is used to present these complex bodies and collision detection is performed on the basis of this mesh. Representing complex bodies with a triangular mesh has a big advantage on the flexibility of our application. Once the demo is implemented an arbitrary 3D geometry in the triangular mesh format can be imported easily and used as a touchable object. Note that, even though the texture plate was a complex body too, the regularity of the sine wave pattern permitted to implement this demo with an analytical approach. The texture plate can be imagined as "hard-coded" which makes this demo not flexible at all.

The scope of the ”Touch the Bunny” demo is to make it possible to explore complex geometries with the haptic device. A virtual point probe is used to scan the surface of the complex body. This probe is rigidly attached to the flotor and so the user can directly feel the resistance of the surface. The geometries we are using in this new demo can consist of thousands of triangles and the biggest challenge is to perform collision detection between the probe and the complex geometry at a frequency rate of 1000 Hz. Initially the physical simulation engine Open Dynamics Engine (ODE) was tried out to calculate the forces which had to be applied to the flotor. It turned out that ODE‘s collision detection algorithm for bodies with complex geometries is not reliable and not fast enough to keep up with the speed of the control loop. Collision detection of bodies with a complex geometry is in general computationally expensive and cannot be performed at a frequency of 1000 Hz. However, the problem we had to solve was special, in the sense that only one of the bodies had a very complex geometry and the other one was simply a point. This special case of collision detection is not supported by ODE, because a single point is not considered to be a 3 dimensional object. It is possible for the probe to use a 3D object like a sphere, but that again would make collision detection unreliable and very slow. For this reason we decided to implement our own collision detection algorithm which is specialized for collisions between a complex geometry and a single point.

The object we dealt with was a 3D model of a laser-scanned ceramic sculpture of a bunny, consisting of a triangular mesh with 70000 triangles. Since the virtual probe was just a simple point, only one triangle at the time can collide with the probe. Once this triangle is determined it is straight forward to calculate the force to be applied to the flotor. The orientation of the force is given by the surface normal of the triangle and its intensity is obtained by multiplying the distance between probe and triangle by a stiffness gain. However, it is not trivially easy to find the right triangle and the right direction of its normal.

6.1 Collision detection algorithm

The 3D model of the bunny is stored in an *.iv* file which contains information about triangle coordinates and their connection to each other. Unfortunately the triangular mesh we were provided with, did not contain information of the normals of the single triangles. This information is needed in order to calculate the forces applied to the probe. In particular the direction of the normal is essential for determining whether the probe is inside or outside the bunny. For example the scalar product of the normal and the vector which connects the middle point of the triangle with the probe will be negative if the probe is inside and the normal sticking outward, but it will positive if the probe point is not penetrating the bunny (figure 29). According to the sign of this scalar product and knowing the direction of the normal it can be inferred whether the probe and bunny are colliding with each other or not.

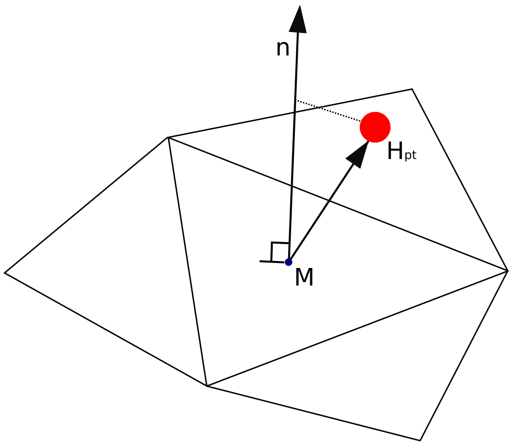


Figure 28: Scalar product between the normal of the surface and the vector connecting the probe point with the middle point of the triangle.

$$collision = true \qquad if \qquad sign(\overrightarrow{n} \cdot \overrightarrow{MH_{pt}}) = - \tag{6.1}$$

The orientation of the normal of each triangle is calculated simply by taking the cross product of the vectors given by two sides of the triangle (figure 30).

$$\vec{n} = \vec{v_1} \times \vec{v_2} \tag{6.2}$$

More challenging is the determination of the direction of the normal, since it can't be deduce from the cross product (eq. 6.2) whether the normal points inward or outward the bunny. It doesn't matter what direction is picked as long as all normals of the bunny follow the same convention, that means they have to point or all inward or all outward.

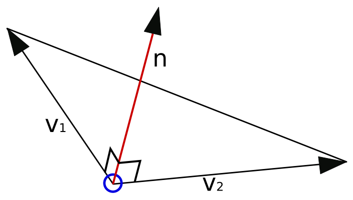


Figure 29: Calculating the orientation of the normal by taking the cross product of two vectors formed by the triangle edges.

An intuitive way to solve this problem is taking one triangle and set the normal to a certain direction. By doing some geometric calculations also the direction of its neighboring triangles can be determined so that they conform with each other. This method is then applied to the next neighboring triangles until the normals of all triangles are determined (figure 31). After doing so, all triangles conform with the normal of the first triangle. By changing the sign of the first triangle the directions of all other triangles will be inverted when performing the same algorithm again.

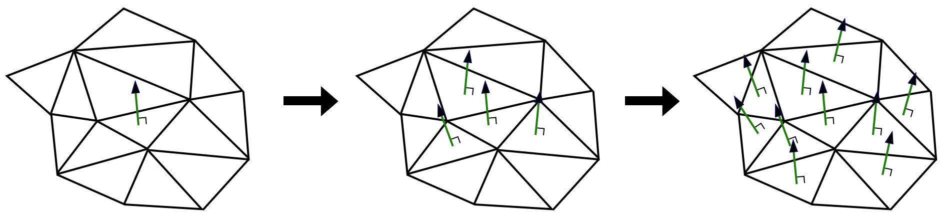


Figure 30: Determining the direction of the normal of one triangle affects the direction of its neighboring triangles.

The method for determining the direction of the normals explained above requires knowing the neighbors of each triangle in the mesh. This information is not given by the .iv-file of the 3-dimensional bunny. An algorithm was implemented which determined the neighbors of each triangle using the property that neighboring triangles share one side. As a consequence they also must have two points in common which is a favorable property.

The points of the triangular mesh of the bunny are organized in a big table, where each line contains the indices of the three points forming a triangle. Therefore this table has three columns and as many rows as the triangular mesh has triangles. Neighboring triangles can be find by looking for rows which have two numbers in common (table 2).

triangle n	**idx 1**	**idx 2**	**idx 3**
234	11238	11246	11248
235	11266	**11239**	**11302**
236	10456	10511	10501
⋮	⋮	⋮	⋮
1349	**11239**	11281	**11302**

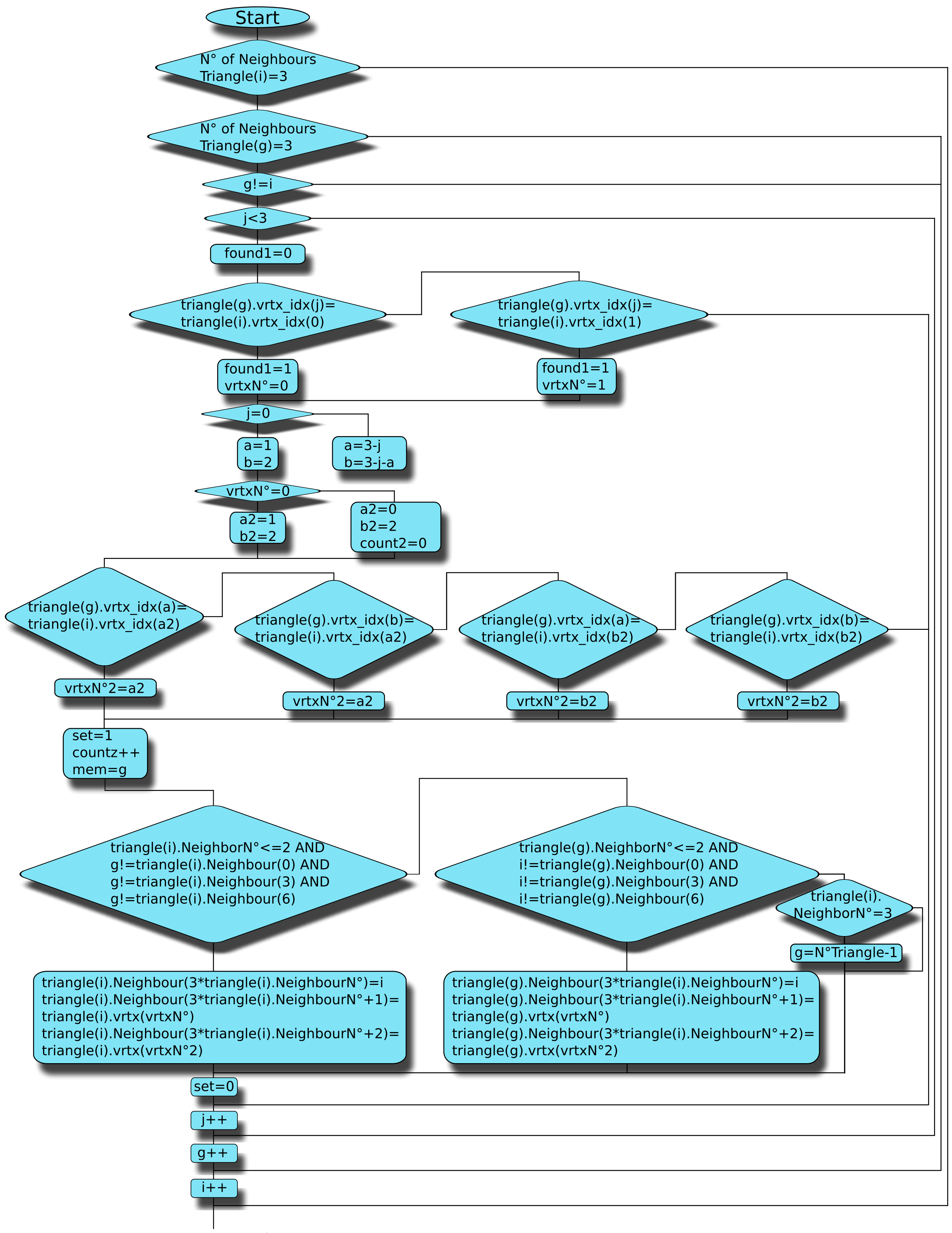
Table 2: Table showing how the triangular mesh is organized.

The .iv-file contains a second table which stores the coordinates of the different triangles. The number of lines of this array corresponds to the number of points in the triangular mesh. The values in the first column represent the indices of the mesh points used in the first array to describe which points form a triangle.

triangle n	**idx 1**	**idx 2**	**idx 3**
11241	0.5893	43.9867	13.2864
11239	0.6010	44.0451	13.3735
11240	0.5870	44.1353	13.4210
⋮	⋮	⋮	⋮
11302	0.6112	44.0538	13.3922

Table 3: Table showing how the mesh points are stored.

6.2 Algorithm for finding neighboring triangles


Start
N° of Neighbours Triangle(i)=3
N° of Neighbours Triangle(g)=3
g!=i
j<3
found1=0
triangle(g).vrtx_idx(j)= triangle(i).vrtx_idx(0)
triangle(g).vrtx_idx(j)= triangle(i).vrtx_idx(1)
found1=1 vrtxN°=0
found1=1 vrtxN°=1
j=0
a=1 b=2
a=3-j b=3-j-a
vrtxN°=0
a2=1 b2=2
a2=0 b2=2 count2=0
triangle(g).vrtx_idx(a)= triangle(i).vrtx_idx(a2)
triangle(g).vrtx_idx(b)= triangle(i).vrtx_idx(a2)
triangle(g).vrtx_idx(a)= triangle(i).vrtx_idx(b2)
triangle(g).vrtx_idx(b)= triangle(i).vrtx_idx(b2)
vrtxN°2=a2
vrtxN°2=a2
vrtxN°2=b2
vrtxN°2=b2
set=1 countz++ mem=g
triangle(i).NeighborN°<=2 AND g!=triangle(i).Neighbour(0) AND g!=triangle(i).Neighbour(3) AND g!=triangle(i).Neighbour(6)
triangle(g).NeighborN°<=2 AND i!=triangle(g).Neighbour(0) AND i!=triangle(g).Neighbour(3) AND i!=triangle(g).Neighbour(6)
triangle(i). NeighborN°=3
g=N°Triangle-1
triangle(i).Neighbour(3*triangle(i).NeighbourN°)=i triangle(i).Neighbour(3*triangle(i).NeighbourN°+1)= triangle(i).vrtx(vrtxN°) triangle(i).Neighbour(3*triangle(i).NeighbourN°+2)= triangle(i).vrtx(vrtxN°2)
triangle(g).Neighbour(3*triangle(i).NeighbourN°)=i triangle(g).Neighbour(3*triangle(i).NeighbourN°+1)= triangle(g).vrtx(vrtxN°) triangle(g).Neighbour(3*triangle(i).NeighbourN°+2)= triangle(g).vrtx(vrtxN°2)
set=0
j++
g++
i++

Flow chart 1: Algorithm for finding neighboring triangles.

Since we know that each pair of neighboring triangles has two points in common, we have to take each triangle and check in the array if there is another row which has two equal vertex indices. This algorithm would have a complexity of $O(n^2)$ which is not acceptable in our case, because the triangular mesh contains 70000 triangles. To reduce complexity the algorithm only checks against the triangles which do not yet have three neighbors, by giving each triangle in the array an additional integer number storing the number of neighboring triangles found. Only triangles with this number smaller than 3 are considered for further searching.

Note from the flow chart that only two points of the starting triangle (index i) are necessary to be compared with the vertices of the remaining triangles (index g). Since neighboring triangles share two points, at least one out of two points of each triangle has to occur in each set. From the following figure it can be deduced that, using this approach, finding the first common vertex of two triangles implies checking 6 different configurations, instead of 9 (figure 32).

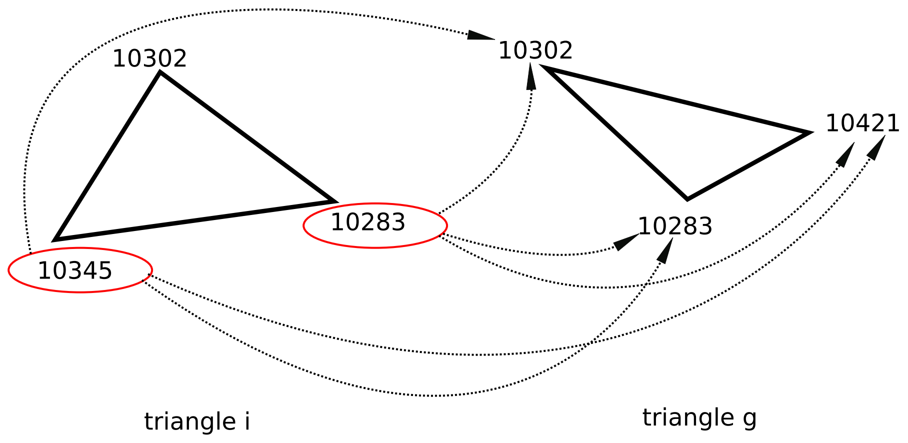


Figure 31: Scheme explaining how indices are used to find corresponding neighboring triangle.

Once two triangles which share one vertex, are found, a further examination for an eventual second common point is performed. In this second step four different possibilities have to be checked figure 33.

If finally a pair of neighboring triangles is identified, the information of the respective neighbor is stored next to each triangle's vertex information. That

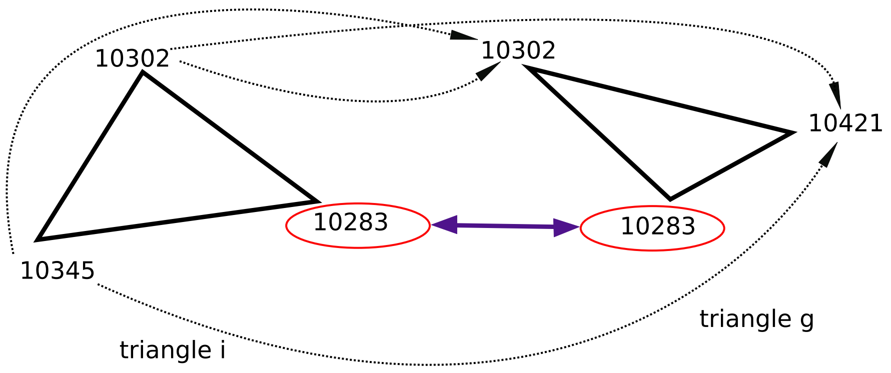


Figure 32: If one common vertex is found, 4 different checks are performed to see if there is a second point shared by both triangles.

includes the triangle number and the indices of the shared points. Additionally a counter variable is used, providing a termination condition of the algorithm, once all three neighbors of a triangle are found. In the end the array of the triangular mesh has been extended by 10 columns:

triangle n	idx 1	idx 2	idx 3	Neighb 1	Vrtx1 N1	Vrtx2 N1	Neighb 2	...
234	11238	11246	11248	1345	11346	11250	1350	...
235	11266	11239	11302	**1349**	**11239**	**11302**	1353	...
236	10456	10511	10501	1298	11001	11003	1323	...
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	...
1349	11239	11281	11302	**235**	**11239**	**11302**	876	...

Table 4: Table representing the triangular mesh containing information about triangle neighbors.

6.3 Determining normals of each triangle

After identifying the neighbors of each triangle, the next step is to calculate orientation and direction of the triangles' normal. Taking the cross product of the vectors determined by two edges of a triangle gives the orientation of the normal of that particular triangle.

However the cross product does not give information about the direction of the normal which is essential for the following collision detection. All normals should follow the same convention, e.g. they whether point all inwards or all outwards the bunny. Once the direction of the normal of one triangle has been determined, it is possible to determine also the normal's direction of its immediate neighbors. This was initially implemented using a recursive approach. Unfortunately it turned out that the stack of the computer was not big enough. Additionally we ran into memory problems, because up to 70000 copies of the same function were open at the same time. For these reasons a procedural approach has been implemented, since it has no disadvantage in terms of complexity with respect to a recursive approach. The idea of accomplishing the task with a procedural algorithm is to start from an arbitrary neighbor of the first triangle, determine the sign of its normal according to the direction of the first triangle's normal and then repeat the same with an arbitrary neighbor of this new triangle.

This approach is not very elegant since the forming chain of triangles with the normal determined is totally random, which implies that it could get stuck somewhere, before all normals are calculated. This happens for example when the chain closes itself like it is depicted in figure 34.

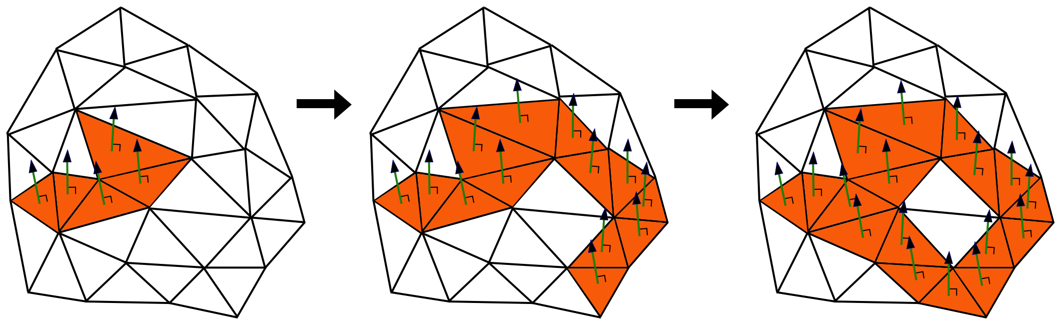


Figure 33: The forming chain can close itself.

In this case the algorithm checks for new neighbors of the last triangle and continues with spreading out. At one point the algorithm will get stuck when for example it hits a triangle whose neighbors have all their normals already determined. Consequently the first triangle in the list which hasn't the direction of its normal already set, but has at least one neighbor whose normal is determined, is taken as new starting triangle for the algorithm.

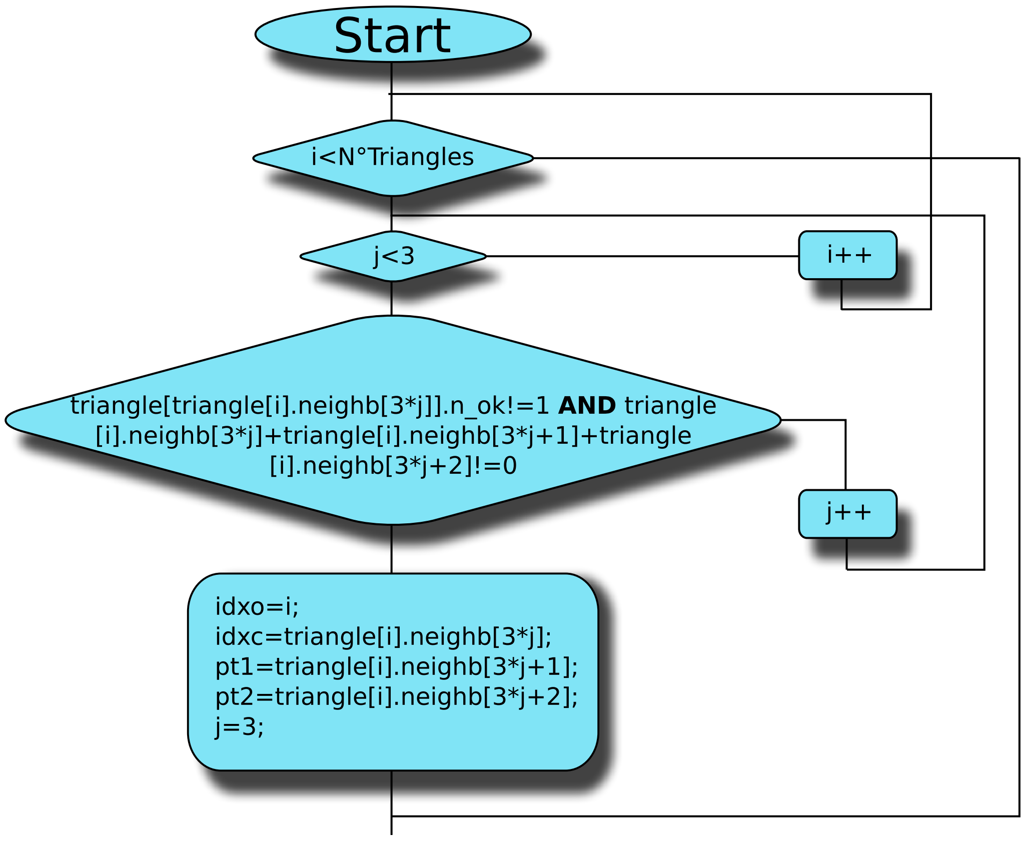

Start
i<N°Triangles
j<3
i++
triangle[triangle[i].neighb[3*j]].n_ok!=1 AND triangle [i].neighb[3*j]+triangle[i].neighb[3*j+1]+triangle [i].neighb[3*j+2]!=0
j++
idxo=i;
idxc=triangle[i].neighb[3*j];
pt1=triangle[i].neighb[3*j+1];
pt2=triangle[i].neighb[3*j+2];
j=3;

Figure 34: Flowchart explaining how the algorithm looks for new triangles when it gets stuck.

6.4 How to determine the direction of the normal

Once a pair of triangles is identified which has one triangle's normal determined, a certain procedure is followed to calculate the normal's direction of the second triangle.

The first step is to cut the surfaces of the two neighboring triangles by a plane which is perpendicular to the side connecting the two triangles (first image figure 36). All 4 points of the triangle pair are projected onto this plane.

The two points A and B which are shared by both triangles overlay on the plane, therefore the projection only shows three points. By connecting $P1'$ with A' and $P2'$ with A' a certain angle is formed (right image figure 36).

For getting consistency of the normal's direction, their projections (which are actually represented in real length on the plane) have to lie in the same circle sector formed by the previously determined vectors $P1'A'$ and $P2'A'$. In other

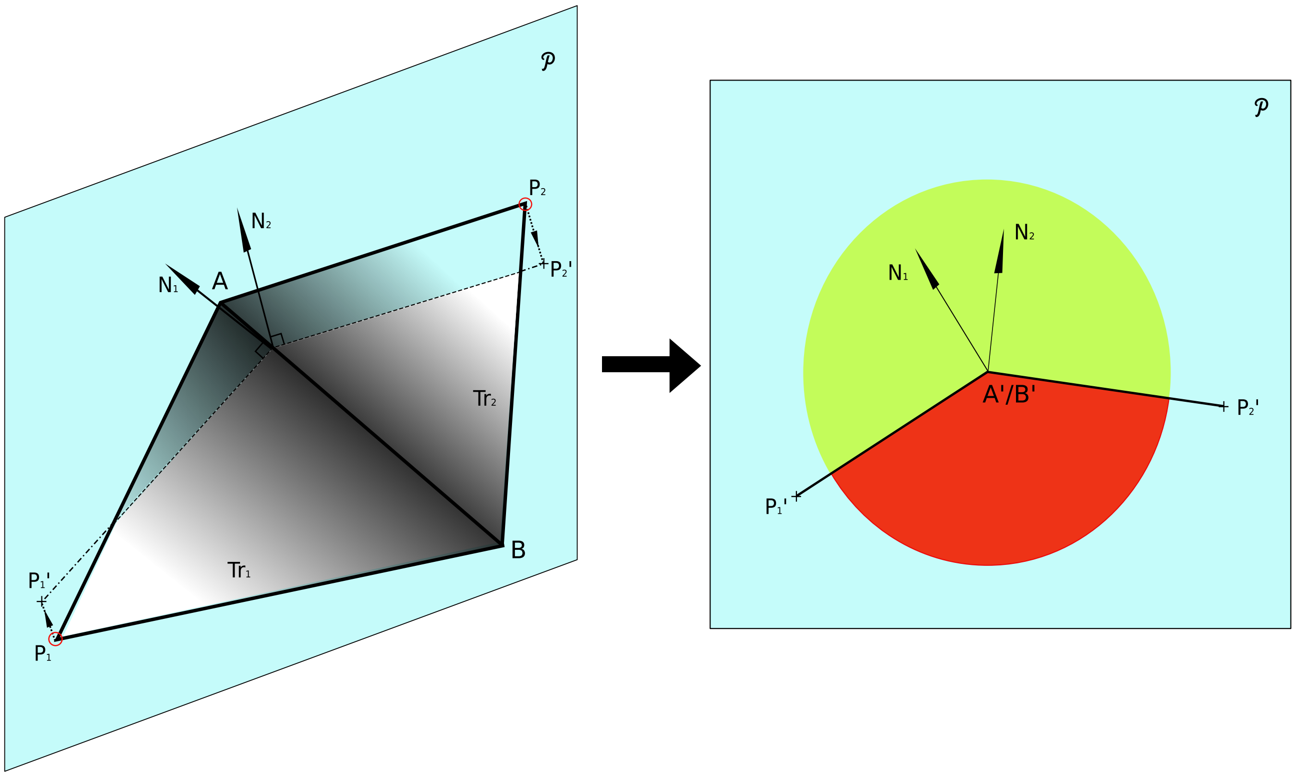


Figure 35: The normal of two neighboring triangles have to lie in the same circle sector.

words if the normal of the first triangle lies in the green circle sector, then the normal of its neighbor has to be also in the green sector. If the new calculated normal lies in the opposite angle sector, its sign must be inverted. To determine whether the normals are in the same angle sector, the scalar product between the normal of one triangle with the vector representing the cut of the surface of the other triangle is calculated for both combinations. If the scalar products have the same sign, then the normals are located in the same angle sector, otherwise they are not.

$$sign(\overrightarrow{N1} \cdot \overrightarrow{P2'A'}) = sign(\overrightarrow{N2} \cdot \overrightarrow{P1'A'}) \tag{6.3}$$

6.5 Runtime collision detection algorithm

The previous sections described the pre-calculation steps which have to be performed when the "Touch the Bunny" demo is started. Once the normal of the triangles are determined, the simulation is ready to start. Collision detection is performed during runtime, which means that the performance of the algorithm has to be very high in order to guarantee a fast calculation of the flotor forces (less than 1 ms). This implies that it has to be highly optimized since collision detection is very expensive when the shape of the bodies are complicated. Our case involves collision detection of a complex object with a simple point. This circumstance simplifies the problem significantly and it remains to find the right triangle with which the probe is colliding. An intuitive solution for this problem would be calculating the distances of the middle points of all triangles to the point of the probe and pick the closest triangles for further investigation. This is too expensive in terms of computation, since the complex object we are dealing with is formed by 70000 triangles. The algorithm would not be fast enough to keep up with the 1000 Hz of the control loop. To overcome this problem only a few triangles namely the ones which are in the immediate proximity of the probe are checked for eventual collision. This is done by considering only triangles which are located in the inside of a virtual cube around the probe point (figure 37). Picking the right size of this cube is very critical for the performance of our algorithm. If the cube is too big and too many triangles are located at its inside, the computation time becomes quickly unacceptable. On the other hand, if the size of the cube was chosen too small, and too few triangles are considered at each time step, the algorithm could fail at detecting an eventual collision. Even if the cube has at least the size to bound the biggest triangle of the mesh, the algorithm could still fail detecting collisions, because of the limited update rate of the position information (1000 Hz). If the probe is moved at relatively high velocity so that its position varies a lot from one time step to another, it could happen that in one time step the probe is outside the bunny and no middle point of a triangle is located inside the checking box. In the consequent time step the probe has penetrated the bunny, so that again no middle point of the triangle is located in the inside of the bunny, then the algorithm will not trigger the signal for collision. In other words, while crossing the surface no triangle has ever been considered to be colliding with the probe.

To summarize choosing the box too big is critical for computation time. Choosing it too small can lead to the failure of the collision detection algorithm. After a while of trial and error we found an appropriate size of the cube which worked in all cases. The computer we were implementing this demo actually allowed us to use a very big cube size, so that 1000 triangles can easily be checked in every loop - which was not necessary.

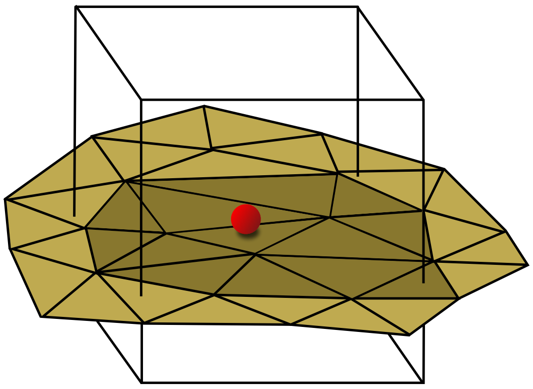


Figure 36: Only triangles falling into the bounding box around the point probe are considered for collision detection.

By using the approach described above, at each loop the middle points of all triangles would have to be checked, if they fall into this bounding box or not. This is obviously computationally too intense since we have to check x, y and z coordinate of 70000 triangle middle points. Additionally there are many other operations which have to be executed altogether in less than 1 ms. An easy but very effective approach of making this procedure remarkably faster is achieved by sorting one coordinate of the triangle's middle points beforehand. On this sorted list of, lets say x-coordinates, a binary sorting algorithm is applied twice in order to find the lower and upper bound of the x-coordinate which fall into the bounding box. Now the remaining part of the body which is further investigated for collision, can be imagined as a slice.

All the triangles falling into this slice are further investigated for satisfying the condition of being within the y and z bound of the checking box. Unfortunately the list of y and z coordinates falling into this sliced section are not sorted and therefore all points have to be checked. But there is good news, only one of the two remaining coordinates has to be checked first. By doing so, the search space has again dramatically decreased. After this, the last coordinate of just a few triangles has to be examined if they are located in the inside of the bounding box.

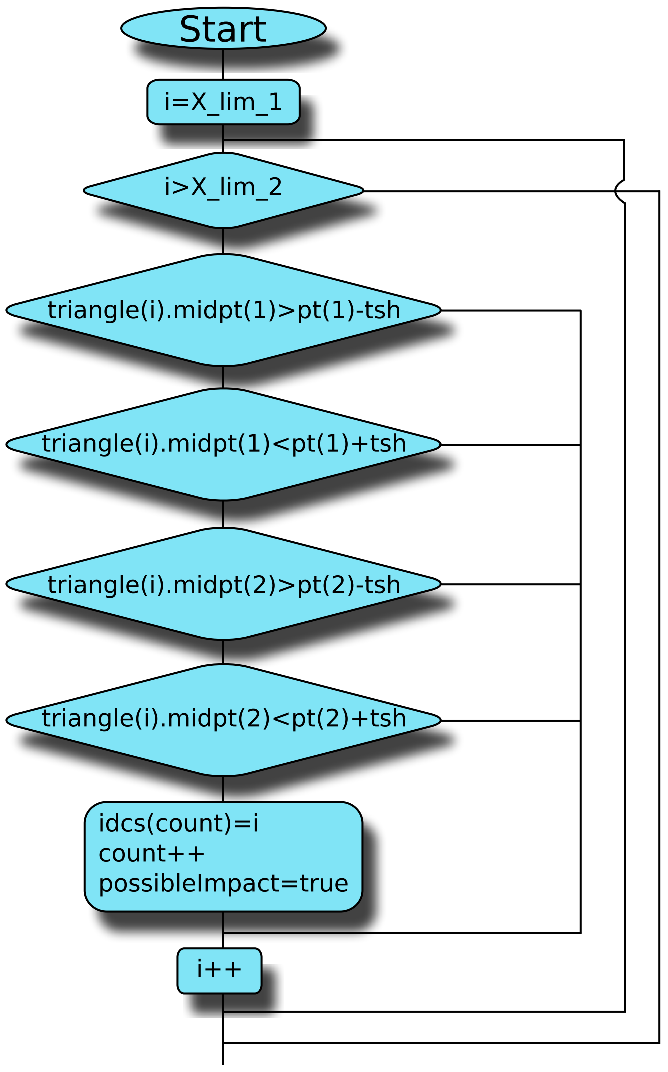

Start
i=X_lim_1
i>X_lim_2
triangle(i).midpt(1)>pt(1)-tsh
triangle(i).midpt(1)<pt(1)+tsh
triangle(i).midpt(2)>pt(2)-tsh
triangle(i).midpt(2)<pt(2)+tsh
idcs(count)=i
count++
possibleImpact=true
i++

Figure 37: Flow chart showing the efficient selection of triangles which fall into the bounding box

Now the very few triangles (5-10), which satisfy the condition of being located within the box, can be checked for collision. The further examination involves projecting the point of the probe onto all planes which are defined by these triangles located in the bounding box. Thereby the 3 points of each triangle form an individual plane.

The probe point has to be projected normally to each plane which is done by first projecting the line connecting middle point of the triangle with the point of the probe, onto the normal of the triangle. This projection is found by calculating the scalar product of the triangle's normal N and the vector MP connecting the probe point and the middle point of the same triangle.

A new vector SN is formed by multiplying the scalar calculated before with the normal of the triangle. The final projection of the probe point onto the plane is obtained by subtracting this vector from MP and by adding the vector OT connecting the triangle's middle point with the origin.

Once we know the exact coordinates of the projected point, it remains to check whether it falls into the region of the triangle on the plane or not (figure 39). If the projection is located in the inside of the triangle, the force would be determined by its normal in the case of a collision.

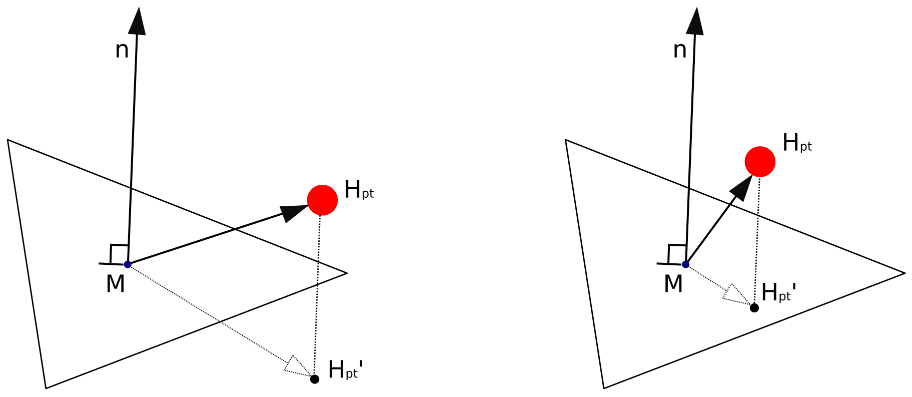


Figure 38: The point projected onto the plane of the triangles can be located inside or outside the triangle.

By connecting this projected point with the three vertices of the triangle 6 angle are formed. Summing up these angles gives a conclusion about the location of the projected point. This sum gives exactly 180 degrees if the point stays inside the triangle. For points which stay outside the triangle the sum is bigger than 180, as becomes clear with figure 40.

Due to numerical discretization the summed angle is not perfectly 180 degrees for points which stay inside a triangle. Therefore the triangle with the smallest angle is chosen to be the one potentially colliding with the probe point. For this triangle finally the scalar product between the normal and the vector connecting the middle point of the triangle and the probe point is calculated. If the sign of this scalar product changes within two subsequent simulation steps, then a collision of the probe and the bunny has occurred. The magnitude of the force

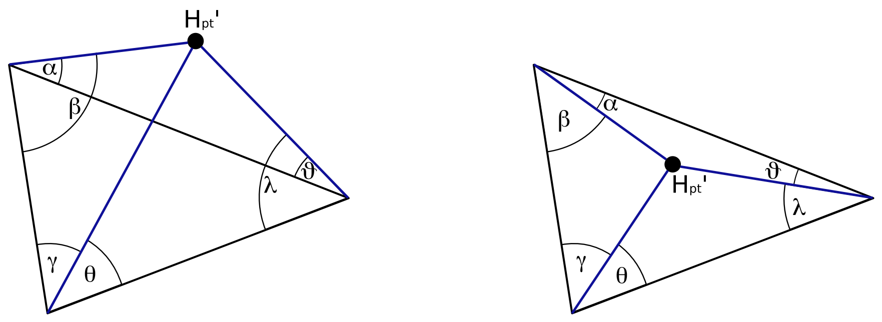


Figure 39: By summing up the depicted angles it is easy to determine whether the point is inside or outside the triangle.

applied to the flotor is proportional to this scalar product. The orientation and direction of the force is given by the normal of the triangle.

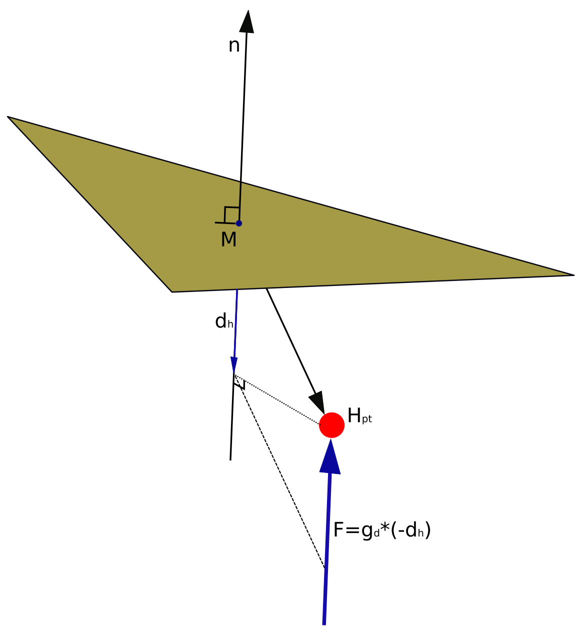


Figure 40: Scheme showing how the penetration force is finally calculated.

Unfortunately the projected point could fall in more than one triangle. But the good thing is that the scalar product between the normal and the line connecting middle point and the haptic interaction point will only change its sign when applied to the triangle with which the probe is actually colliding.

6.6 Smoothing the surface

The bunny is formed by triangles which are connected to each other at their edges. This forms angular transitions between the triangles, which cause a not smooth behavior of the force applied to the flotor when the probe is sliding from on triangle to its neighbor.

To avoid the bunny's surface to appear rough, due to this phenomena described above, an interpolation is applied to the colliding triangle and its neighbors. This approach can be also imagined as a Gaussian smoothing of an image, where each pixel value is determined by its original value and the values of its neighbors.

Whenever the probe penetrates the body, the force which is applied to the flotor is a linear combination of the the normals given by the triangle currently colliding with the probe and its three neighbors (figure 42).

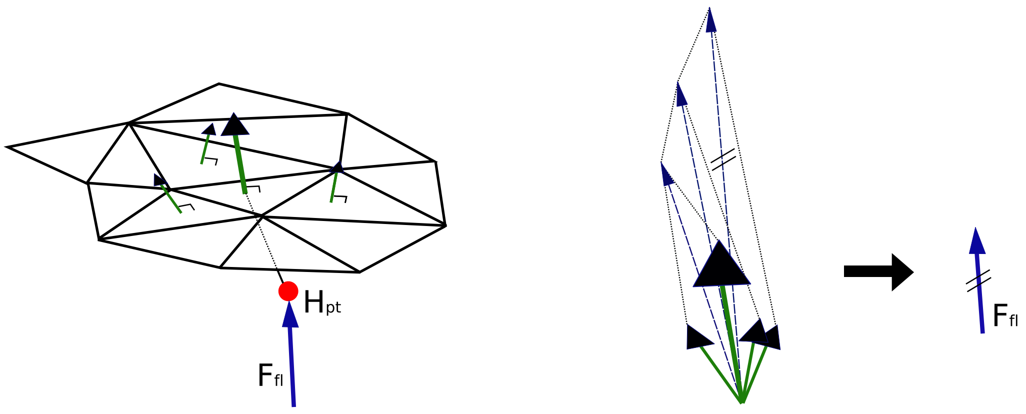


Figure 41: Scheme showing how the surface is smoothed with a linear combination of the normals of the neighboring triangles.

$$\vec{F}_{Fl} = a\vec{N}_0 + b\vec{N}_{N1} + c\vec{N}_{N2} + d\vec{N}_{N3} \tag{6.4}$$

This method can be also extended to a larger number of neighbors. However, it has the drawback that the surface of the bunny is smoothed too much and fine details such as small "valleys" are not perceived anymore. After doing some trial and error, it was found out that taking only the direct neighbors of each triangle is the best tradeoff between smoothing and keeping the details of the object's surface.

6.7 Enlarging the workspace of the haptic device

The workspace of the magnetic levitation haptic device is approximately a cube with 14 mm side length. Scaling the bunny down so that it fits this rather small region would be disadvantageous for the resolution. It would be impossible to feel fine details of the bunny's surface. To overcome this issue an algorithm has been implemented which allows changing the relative pose of the device's workspace with respect to the pose of the complex body. This makes it possible that the complex object, in this case the bunny, can be explored in different locations. For doing so the user chooses between positioning mode and touching mode by hitting the left button of the flotor. The positioning mode first navigates the flotor to the zero position of its workspace. It controls this zero position with rather low proportional gains (1 N/mm for translation and 3 Nm/rad for rotation). When the user now pushes the handle away from the zero position, the amount of offset is taken to calculated the motion speed for the axis which has the offset, in rotation and translation. An appropriate deadband has to be chosen in order to prevent the body from moving when no force is applied. This would happen because of the steady-state error of the pose of the flotor, since no integral gain is used in the controller. A further reason for the deadband is the fact that the position signal of the flotor has some noise ($2 - 4\mu m$ standard deviation).

The amount of offset is packed into a homogeneous transformation matrix as follows:

$$T_{new} = T_{old} * T_{HB} = T_{old} * \begin{bmatrix} c\alpha c\beta & c\alpha s\beta s\gamma - s\alpha c\gamma & c\alpha s\beta c\gamma + s\alpha s\gamma & X \\ s\alpha c\beta & s\alpha s\beta s\gamma + c\alpha c\gamma & s\alpha s\beta c\gamma - c\alpha s\gamma & Y \\ -s\beta & c\beta s\gamma & c\beta s\gamma & Z \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{6.5}$$

$$T_{old} = T_{new} \tag{6.6}$$

The angles α, β and γ represent the angle offset of the flotor with respect to the upright position and X, Y and Z denote the translational offset. The matrix T_{HB} is multiplied by the old homogenous transformation, which represented the

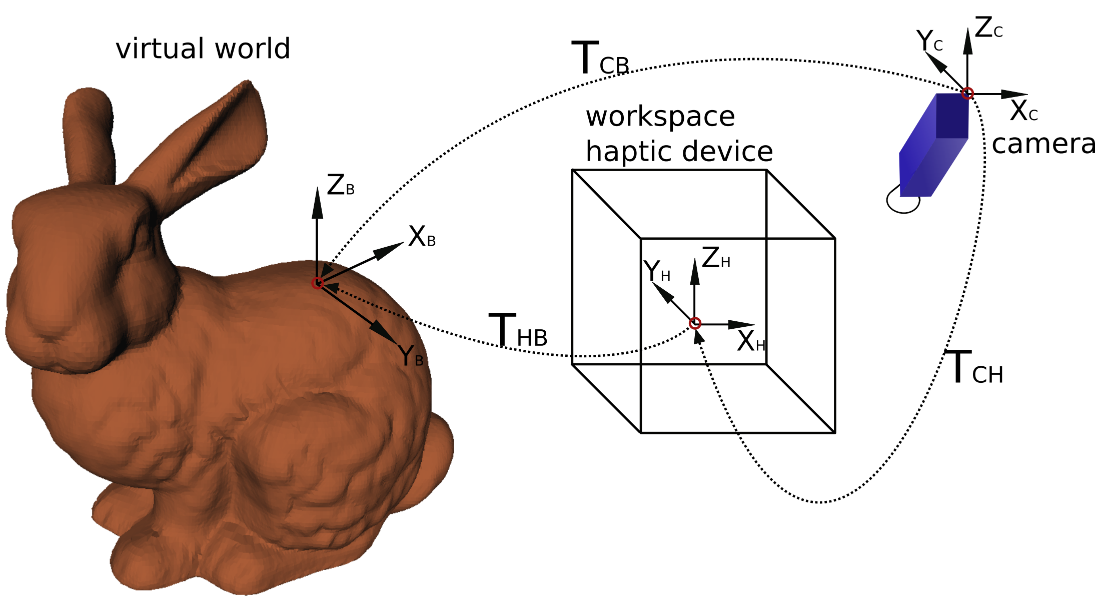


Figure 42: This scheme shows the relation between the bunny coordinate frame, the work space and the camera coordinate frame.

position of the haptic's workspace with respect to the world frame in the previous simulation step. Figure illustrates the relation between the single coordinate frames. T_{CH} represents the relation between the coordinate frame of the camera and the work space of the haptic device. It is obviously a constant homogenous transformation.

The triangular mesh of the bunny is actually fixed and the pose of its inertial coordinate frame expressed in the world frame is a constant homogeneous transformation. Instead the homogeneous transformation matrix, which describes the relative pose of the flotor's workspace with respect to the world frame and the bunny T_{HB} is not constant and changes due to the multiplication of $T_{old} * T$. This convention has been chosen, because it wouldn't be feasible to rotate the actual bunny around. In this case in each time step all the coordinates of the triangle vertices and the triangle middle points would have to be recalculated according to the new transformation. Note that this process doesn't have to be performed at 1000 Hz, because there is no force feedback in the positioning mode. To guarantee a smooth visualization of the demo when the device's workspace changes pose, the homogenous transformation of the mesh would have to be performed at 100 Hz, which is double the update frequency of the screen. However, recalculating the coordinates of the triangular mesh is computationally so intense that it wouldn't be even close to such a required speed.

7 Summary and Contributions

Three virtual environments have been implemented for the purpose to show the performance of the new magnetic levitation haptic device. A major new aspect of the implementation of these demos was the use of a structured application programming interface, which takes care of the low-level hardware control. This allows the user to concentrate on the higher level issues of the virtual environment to be implemented. Since the API was still in development when these three demos have been implemented the purpose of the underlying project was also to test and debug this API.

Contemporaneous with the completion of this project, the new magnetic levitation haptic device has been presented officially at a press release at Carnegie Mellon University. Furthermore we exposed the device at the international haptic symposium in Reno. At both events the demos were used to highlight the special characteristics of the new maglev haptic device.

7.1 Future Work: Physical simulation using a physics engine

Simulating the physical phenomena occurring in a virtual environment is one of the most challenging components of haptic rendering. Collision detection is very complicated when virtual environments contain a large number of bodies, or when the bodies have a complicated geometry. If both is the case, then the problem gets almost impossible to solve at high update rates, which are needed in haptic rendering. When simulating complex environments, it is very useful using a physical simulation engine such as Open Dynamics Engine (ODE) or Ageia PhysX. These simulation engines have the advantage that their collision detection algorithm is optimized in terms of computational complexity. A further convenience is their integrability in a haptic rendering loop. It is fairly straight forward to implement the calculation of the forces which are applied to the haptic device, even if the virtual environment is sophisticated or contains a numerous amount of bodies interacting with each other. However, the performance of haptic rendering demos using such a physics engine is not as good as in demos which use a customized implementation of the physical simulation. This is caused by the fact that virtual environments using physics engines cannot make use of very high stiffness values. To find the reason for this problem one has to analyze the way physics engines simulate the dynamics of bodies in the virtual environment.

When two bodies collide with each other a force is calculated on the basis of the penetration depth of the two bodies. In a further step this force is applied to the two bodies. All the forces applied to a body are summed together and cause the body to translate and rotate. Depending on the inertial properties of the body, an acceleration of the body determines velocity and position when integrated over time.

$$a = F/m \tag{7.1}$$

$$v = \frac{1}{m}\int F dt + v_0 \tag{7.2}$$

This implies that the position of a bodies in the virtual world is indirectly given by the forces applied to it and that means that the flotor's position cannot be directly assigned to the position of a virtual object.

However in haptics simulation there should be a rigid connection between the handle of the device and the virtual object this handle represents in the virtual world. In other words, it is desired to assign the position of the bodies in the virtual world directly. This is not possible with todays physics engines and therefore a virtual spring-damper system has to be inserted between the flotor and the object in the virtual world. Figure () illustrates this connection; note that the damping value d_v and spring constant k_v do not have anything to do with the controller gains of the haptic device.

In a first moment this configuration seams to be the perfect solution to our problem, because apparently the stiffness of this virtual spring can be set in theory to an infinitely high value and we would end up with a rigid connection between handle and the corresponding body in the virtual world.

By moving the flotor around, the virtual spring of this system will be compressed due to the inertia of the object in the virtual world to which the handle is attached. The compression of the spring induces a force which "pushes" the object in the desired direction. The higher the stiffness of this virtual spring the better the virtual object follows the motion of the handle of the haptic device. By setting it to infinite, the virtual body is rigidly attached to the handle and

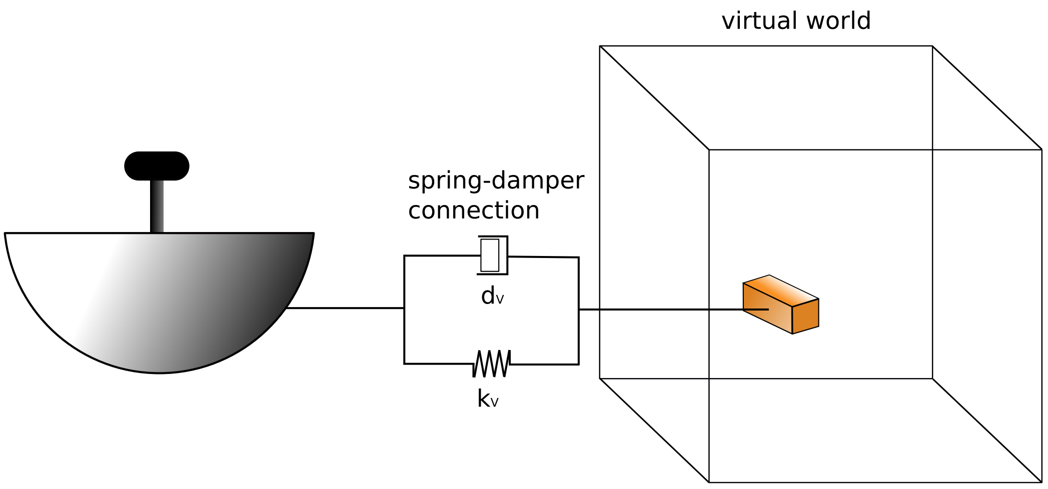


Figure 43: Spring-damper connection between flotor and virtual world object.

follows perfectly the motion of the handle, which is exactly the behavior we want to achieve. Even by setting the stiffness of the virtual spring to a very high value, would make the connection so rigid, that no real difference to the approach where the position of the virtual object is set directly, would be recognizable. Unfortunately this is not possible, since the damping of this virtual coupling would have to be incremented too, otherwise high frequency vibrations of both, the virtual body and the handle would be the consequence.

A very high proportional gain demands a relatively high differential gain. In a first consideration this does not seem to be a problem, but the truth is that we are actually limited in setting this damping gain. The reason for the limitation is the noise of the position information of the handle. The damping force is calculated by multiplying the first derivative of the position with the damping gain. The differentiation of a signal with noise leads to value which is actually even noisier then the source signal. It is straight forward that when multiplying this "jumpy" signal by a high damping gain, which is needed when using high stiffness, the noise of the force output to the flotor is not acceptable.

There is an additional aspect which causes instabilities when setting very high values for the pd-controller gains. Unfortunately the proportional force and the damping force are calculated in two different time instances of the system. The velocity is the mean velocity between the actual position and the position in the previous time step. Besides from being affected by noise, this is not the velocity of the flotor in the current loop, but the proportional force calculated is related to the current state of the system.

Both problems just explained can be improved by increasing the frequency of the control loop. Position information can be filtered better and the approximation of the current velocity of the flotor is more precise.

Due to the limitation explained above, virtual environments implemented using physics engines are "softer" than demos in which the implementation of the physical simulation is done by the user. In the second case the user can rigidly attach the flotor to the body in the virtual world and a spring-damper connection, between the flotor and the virtual body system, which is sensitive to signal noise, is not needed anymore.

If a physical simulation engine would include the possibility to directly control the pose of bodies in its virtual world, then the problem explained above is overcome. In that case implementing virtual environments can be done using physical simulation engines without giving up performance.

However, if the position of virtual bodies is set directly, it is not allowed to apply additional forces to these bodies. More specifically, in one simulation loop it would be forbidden applying forces and setting positions to a body at the same time. This is hard to imagine since bodies apply forces to each other when colliding, if at the same time the position of a body is changed, because it is rigidly attached to the flotor, then the system is considered to be over-constrained. This contradiction leads to the fact that the problem can only be solved if the forces calculated in the physics engine are applied to the flotor and not to the body to which the flotor is attached. The virtual body attached to the flotor should get the position of the flotor directly. Velocity and acceleration of this body is determined by the motion of the flotor and is not directly influenced by the other bodies in the virtual world. Forces coming from the body attached to the flotor and applied to other bodies are calculated by the change of impulse of the flotor.

$$F = \frac{mdv}{dt} \tag{7.3}$$

As can be seen from eq. 7.3 the inertia tensor of the flotor has to be known in order to calculate F.

The approach described above demands significant changes to the physical simulation engine, which is unfortunately not possible with Ageia PhysX, since this library is not open source. Open Dynamic Engine on the other hand is an open source library and could be accustomed to the use in haptics rendering.

References

[1] T. H. Massie and J. K. Salisbury, "The PHANToM haptic interface: A device for probing virtual objects", *Proceedings of the ASME Dynamic Systems and Control Division*, pp. 295-299, American Society of Mechanical Engineers, 1994.

[2] A. Cohen and E. Chen, "Six degree-of-freedom haptic system as a desktop virtual prototyping interface", *Proceedings of the ASME Dynamic Systems and Control Division*, vol. DSC-67, pp. 401-2, ASME, 1999.

[3] M. Minsky, M. Ouh-young, O. Steele, F. P. Brooks, Jr. and M. Behensky, "Feeling and seeing: issues in force display", *Computer Graphics*, vol. 24(2), pp. 235-243, 1990,

[4] J. E. Colgate, J. M. Brown, M. C. Stanley, G. G. Schenkel, P. A. Millman, and K. W. Grace, "The fundamentals of haptic display - a study of virtual wall implementation", *IEEE Int'l Conf. on Robotics and Automation - video proceedings*, May 1994,

[5] R. L. Hollis, "Six DOF magnetically levitated fine motion device with programmable compliance", U.S. Patent N4,874,998, October 17 1989.

[6] R. L. Hollis, S. E. Salcudean, and D. W. Abraham, "Toward a telenanorobotics manipulation system with atomic scale force feedback and motion resolution", *Proc. Third IEEE Workshop on Micro Electro Mechanical Systems*, (Napa Valley, CA), pp. 115-119, Feb. 12-14 1990.

[7] S. Salcudean, N. M. Wong, and R. L. Hollis, "A force-reflecting teleoperation system with magnetically levitated master and wrist", *Proc. IEEE Int'l Conf. on Robotics and Automation*, (Nice, France), May 10-15, 1992.

[8] P. J. Berkelman, R. L. Hollis, and S. E. Salcudean, "Interacting with virtual environments using a magnetic levitation haptic interface", *Proc. IEEE Int'l Conf. on Intelligent Robots and Systems*, vol. I, (Pittsburgh, PA), pp. 117-122, August 1995.

[9] P. J. Berkelman, Z. J. Butler, and R. L Hollis, "Design of a hemispherical magnetic levitation haptic interface device", *Proc. ASME Symposium*

on Haptic Interfaces, vol. DSC-58, (Atlanta), pp. 483-488, November 17-22 1996.

[10] P. Berkelman and R. L. Hollis, "Dynamic performance of a magnetic levitation haptic device", *Proc. SPIE Conf. on Telemanipulator and Telepresence Tecnologies IV*, vol. 3206, (Pittsburgh, PA), October 14-17 1997.

[11] P. J. Berkelman, R. L. Hollis, and D. Baraff, "Interaction with a realtime dynamic environment simulating using a magnetic levitation haptic interface device", *IEEE Int'l Conf. on Robotics and Automation*, (Detroit), May 1999.

[12] P. J. Berkelman, "Tool-Based Haptic Interaction with Dynamic Physical Simulations using Lorentz Magnetic Levitation", *PhD thesis, Carnegie Mellon University, 1999.*

[13] P. J. Berkelman and R. L. Hollis, "Lorentz magnetic levitation for haptic interaction: Device design, performance and interaction with physical simulations", *Int'l J. of Robotics Research*, vol. 19, pp. 644-667, July 2000.

[14] R. J. Adams and B. Hannaford, "Stable haptic interaction with virtual environments", *IEEE Trans. on Robotics and Automation*, vol. 15, pp. 465-474, June 1999.

[15] R. L. Klatzky, D. K. Pai, and E. P. Krotkov, "Hearing material: Perception of material from contact sounds", *PRESENCE: Teleoperators and Virtual Environments*, vol. 9, pp. 399-410, August 2000.

[16] S. Choi and H. Z. Tan, "An analysis of perceptual instability during haptic texture rendering", *Proceedings of the 10th Internal Symposium on Haptic Interfaces for Virtual Environment and Teleoperator Systems*, pp. 129-136, 2002.

[17] S. Choi and H. Z. Tan, "A parameter space for perceptually stable haptic texture rendering", *Proceedings of the fifth PHANToM Users Group Workshop*,2002.

[18] K. J. Kuchenbecker, J. P. Fiene, and G. Niemeyer, "Improving contact realism through event-based haptic feedback", *IEEE Transactions on Visu-*

alization and Computer Graphics, vol. 12, no. 2, pp. 219-230, March/April 2006.

[19] K. Waters and S. Wang, "A 3D interactive physically-based micro world", *Proc. SPIE (Extracting meaning from complex data: processing, display and interaction*, vol. 1259, (Santa Claram CA), pp. 91-98, Feb. 14-16 1990.

[20] S. Choi and H. Z. Tan, "An analysis of perceptual instability during haptic texture rendering", *Proc. 10th Int'l Symp. on Haptic Interfaces for Virtual Environments and Teleoperator Systems*, pp. 129-136, March 24-25 2002.

[21] S. Yu and G. Fries "Target location from three direction sensors", *tech. rep., Microdynamic Systems Laboratory*, December 17 1996.

[22] Ho, C., Basdogan, C., Srinivasan, "An Efficient Haptic Rendering Technique for Displaying 3D Polyhedral Objects and Their Surface Details in Virtual Environments", *Presence: Teleoperators and Virtual Environments, MIT Press*, Vol. 8, No. 5, pp. 477-491 October 1999.

[23] R. L. Hollis, "Instrument Development: Magnetic Levitation Haptic Interface System", *NSF Proposal: Major Research Instrumentation Program CISE Experimental and Integrative Activities*, January 23, 2003.

A Cube in Cube demo

The following code includes all the functions necessary to run the demo. It consists of two threads which are running contemporaneously and initialization of the haptic device which is executed at the beginning of the demo.

The graphics thread is responsible for updating the scene at a frequency of 100 Hz. The control loop (”getPos”) is called by the controller callback function (”tickCallback”) 1000 times a second. It provides the new forces to be applied to the flotor.

```
/********************** Cube in Cube demo  *************************/

#include <pthread.h>
#include <sys/time.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <math.h>
#include <time.h>
#include <stdint.h>
#include <ctype.h>

// Graphics libraries needed
#include <Inventor/sensors/SoTimerSensor.h>
#include <Inventor/Xt/SoXt.h>
#include <Inventor/Xt/viewers/SoXtExaminerViewer.h>
#include <Inventor/nodes/SoCube.h>
#include <Inventor/nodes/SoDrawStyle.h>
#include <Inventor/nodes/SoLightModel.h>
#include <Inventor/nodes/SoDirectionalLight.h>
#include <Inventor/nodes/SoMaterial.h>
#include <Inventor/nodes/SoPerspectiveCamera.h>
#include <Inventor/nodes/SoOrthographicCamera.h>
#include <Inventor/nodes/SoSeparator.h>
#include <Inventor/nodes/SoTranslation.h>
#include <Inventor/nodes/SoScale.h>
#include <Inventor/nodes/SoTransform.h>
#include <Inventor/nodes/SoText2.h>
#include <Inventor/nodes/SoFont.h>
#include <Inventor/nodes/SoEventCallback.h>
#include <Inventor/events/SoKeyboardEvent.h>

// Haptic device library
#include "ml_api.h"

// haptic controller ip-address
char * HOST_NAME = "192.168.0.3";
```

```
ml_device_handle_t device_hdl;

// half side length of the cube
const float sc=4.0;

// half side length of the outer box
const float sb=6.0;

// translation and rotation of inner cube
SoTransform *transCube;
SoSFRotation *rot;
SoSeparator *txt, *txt2;
SoSeparator *rootCube;
SoXtExaminerViewer *myViewer;
int setflag=0;
// For some unknown reason the device applies a torque on the z-axis
// whenever the device outputs a force in z direction. I have
// implemented a hack to correct this unwanted torque. The user can
// decide whether using this torque correction or not. As default the
// correction is active.
bool torque_flag=true;

// Actual and desired pose of the device.
ml_position_t position, despos;
ml_gain_vec_t gains;


// position of the inner cube
float b[3];

// Client control loop
void *controlBox(void *arg);
void stopDemo();
// graphics callback function to update the scene
void graphics_callback(void *data, SoSensor *sense);

void myKeyPressCB(void *userData, SoEventCallback *eventCB)
{
   const SoEvent *event = eventCB->getEvent();
   if (SO_KEY_PRESS_EVENT(event, P)) {
     stopDemo();
     //printf("you pressed the button");
          eventCB->setHandled();
   }
}


SoSeparator *text2disp(const char *str[], int sizestr)
  {
  SoSeparator *txt =new SoSeparator;
  SoOrthographicCamera *ocam = new SoOrthographicCamera;
  //ocam->position.setValue(10,100,10);
  txt->addChild(ocam);
  SbViewVolume vv=ocam->getViewVolume();
  SbVec3f pt = vv.getPlanePoint(0.0f, SbVec2f(0.0f, 0.95f));


  SoText2 *text = new SoText2;
  //const char *str[]={"Press ’Esc’ to change camera view manually","", "Press ’i’ to disable impact force","", "Press ’
  //const char * str2[3] = {str[0],str[1],str[2]};
  text->string.setValues(0, sizestr/sizeof(char*), str);
  text->justification = SoText2::LEFT;
  SoTransform *texttrans = new SoTransform;
  texttrans->translation.setValue(pt[0]-0.5,pt[1],0);
  //printf("this pt0 %f\n", pt[0]);
  txt->addChild(texttrans);
  txt->addChild(text);
  return txt;
  }


int main(int argc, char **argv)
{
  if (argc>1){
  char * arg_str = argv[1];
  if (!strcmp(arg_str, "notorque"))
    torque_flag=false;
  }

  //initialization of the control loop thread
  pthread_t thread;
  pthread_create(&thread,NULL,controlBox,NULL);

  //initializing graphics window
  Widget myWindow = SoXt::init(argv[0]);
  if (myWindow == NULL) exit (1);

  //graphics callback for graphics update (100 Hz)
  SoTimerSensor *tSense = new SoTimerSensor(graphics_callback,NULL);
  tSense->setInterval(0.01);
  tSense->schedule();

  //scene definition
  rootCube = new SoSeparator;
  rootCube->ref();

  SoEventCallback *myEventCB = new SoEventCallback;
  myEventCB->addEventCallback(SoKeyboardEvent::getClassTypeId(),myKeyPressCB);
  rootCube->addChild(myEventCB);
  //camera
  SoPerspectiveCamera *myCamera = new SoPerspectiveCamera;
  rootCube->addChild(myCamera);
  myCamera->position.setValue(0.0, -20.0, 15.0);
  myCamera->orientation.setValue(1.0, 0.0, 0.0, 2*M_PI/3);

  //light
  rootCube->addChild(new SoDirectionalLight);

SoFont *myFont = new SoFont;
  myFont->name.setValue("Arial");
  myFont->size.setValue(24.0);
  rootCube->addChild(myFont);

 const char *str[]={"Press 'p' to quit"};
 txt = text2disp(str, sizeof(str));

  //graphics for the bounding box
  SoSeparator *boxSep = new SoSeparator;
  SoDrawStyle *boxSty = new SoDrawStyle;
  boxSty->style = SoDrawStyle::LINES;
  boxSty->lineWidth = 2.0;
  SoLightModel *boxLight = new SoLightModel;
  boxLight->model = SoLightModel::BASE_COLOR;
  SoCube *boxCube = new SoCube;
  boxCube->width=2*sb;
  boxCube->height=2*sb;
  boxCube->depth=2*sb;
  boxSep->addChild(boxSty);
  boxSep->addChild(boxLight);
  boxSep->addChild(boxCube);


  //graphics for the inner cube
  SoSeparator *cubeSep = new SoSeparator;
  SoMaterial *cubeMaterial = new SoMaterial;
  cubeMaterial->diffuseColor.setValue(0.0, 0.9, 1.0);
  transCube = new SoTransform;
  rot = new SoSFRotation;
  SoCube *cubeCube = new SoCube;
  cubeCube->width=2*sc;
  cubeCube->height=2*sc;
  cubeCube->depth=2*sc;
  cubeSep->addChild(transCube);
  cubeSep->addChild(cubeMaterial);
  cubeSep->addChild(cubeCube);

  //adding bounding box and inner cube to the root scene
  rootCube->addChild(boxSep);
  rootCube->addChild(cubeSep);
  rootCube->addChild(txt);

  //viewing the scene with the Examiner Viewer
  //SoXtExaminerViewer *
    myViewer = new SoXtExaminerViewer(myWindow);
  SbVec2s wsizer;
  wsizer.setValue(1900, 1200);
  myViewer->setSceneGraph(rootCube);
  myViewer->setTitle("Cube in the Box");
  myViewer->setSize(wsizer);
  myViewer->show();
  myViewer->viewAll();

  SoXt::show(myWindow);
  SoXt::mainLoop();

}


// this is the callback function for the graphics (100 Hz update frequency)
void graphics_callback(void *data, SoSensor *sense)
{
  static int activetxt=1;
  static const char *str[]={"Press 'p' to quit"};
  static const char *str2[]={"Press 'Esc' and then 'p' to quit"};

  //position update of the inner cube
  transCube->translation.setValue(b[0], b[1], b[2]);

  //orientation update of the inner cube
  transCube->rotation.setValue(rot->getValue());

  if ((myViewer->isViewing()==true) && (activetxt!=2)){
txt2 = text2disp(str2, sizeof(str2));
    rootCube->replaceChild(6, txt2);
    //printf("number of children after first switch %d \n", rootSinus->getNumChildren());
    activetxt=2;}
  if ((myViewer->isViewing()==false) && (activetxt!=1)){
txt = text2disp(str, sizeof(str));
  rootCube->replaceChild(6, txt);
  activetxt=1;}

}

// getPos calculates the desired pose of the flotor. This demo uses virtual coupling and therfore
// forces and torques are communicated to the server in terms of a position offset and gains for
// the PD controller. Virtual coupling has been chosen for this demo because it is more stable
// than conventional force control. When using the force control method the controller for the
// haptic device has to get force information from the client precisely at 1000 Hz, otherwise
// the device would get unstable. But when the controller is provided with desired position
// information for the flotor, the client side does not necessarily have to run at 1000 Hz.
// In that case the haptic device's update rate is just lower, but does not cause any instabilities.
// In this function virtual coupling is not used directly. First I calculate all the forces
// which are applied to the flotor, so that components from the single axes can be added together.
// In a second step these force values and a set of controller gains are used to calculate the
// new desired position of the flotor.

int getPos(ml_position_t & position)
{
  static int i, j;
  static float orient[3];
  static float sgam, cgam, sbet, cbet, salp, calp;
  static float a11, a12, a13, a21, a22, a23, a31, a32, a33;
  static float theta, inv2st;
  static float w;
  static float p[8][3];
  static float pt[8][3];
  static float ptotal[3];
  static float torquebalance=0;
  static ml_position_t force;

  static SbVec4f locVert[8];
  static SbVec4f globVert[8];

for(i=0; i<3; i++)
 position.values[i]*=1000;

for(i=3; i<6; i++)
  position.values[i]*=(360/(2*M_PI));

  // maxpos and maxneg contain the maxium penetration values of the cube in the bounding box for
  // each axis, on the positive and negative side.
  float maxpos[3]={0,0,0};
  float maxneg[3]={0,0,0};

  // Transformation of angle from degrees to radians with additional scale factor to amplify rotation,
  // so that the user thinks the rotation of the flotor is bigger than it actually is.
  for (i=0; i<3; i++)
    orient[i]=4*M_PI*position.values[i+3]/360;

  // Calculation of the rotation matrix. For some reason the homogeneous transformation doesn't work
  // in Coin3d, the translational part is not added to the position. Therefore rotation and
  // translation has to be treated separately. The values which represent the position in the matrix
  // are set to zero (a14=a24=a34=0).
  sgam = sin(orient[0]);
  cgam = cos(orient[0]);
  sbet = sin(orient[1]);
  cbet = cos(orient[1]);
  salp = sin(orient[2]);
```

```
calp = cos(orient[2]);

a11 = calp*cbet;
a12 = calp*sbet*sgam-salp*cgam;
a13 = calp*sbet*cgam+salp*sgam;
a21 = salp*cbet;
a22 = salp*sbet*sgam+calp*cgam;
a23 = salp*sbet*cgam-calp*sgam;
a31 = -sbet;
a32 = cbet*sgam;
a33 = cbet*cgam;

// Conversion of a rotation from three given rotation angles to a rotation vector.
// This conversion has to be done because Coin3d only accepts this kind of notation
// for rotations.
theta = acos((a11+a22+a33-1)/2.0);
inv2st = 1/(2*sin(theta));
rot->setValue(SbVec3f((a32-a23)*inv2st,(a13-a31)*inv2st,(a21-a12)*inv2st),theta);

SbMatrix mat(a11, a12, a13, 0, a21, a22, a23, 0, a31, a32, a33, 0, 0, 0, 0, 1);

// Generating the vertices' positions of the inner cube with respect to its coordinate frame.
// The origin is located at the center of gravity of the cube.
locVert[0].setValue(sc, -sc, sc, 1);
locVert[1].setValue(sc, -sc, -sc, 1);
locVert[2].setValue(-sc, -sc, -sc, 1);
locVert[3].setValue(-sc, -sc, sc, 1);
locVert[4].setValue(sc, sc, sc, 1);
locVert[5].setValue(sc, sc, -sc, 1);
locVert[6].setValue(-sc, sc, -sc, 1);
locVert[7].setValue(-sc, sc, sc, 1);

// Computation of the coordinates of the 8 vertices with respect to the world frame using
// the rotation matrix "mat" set up previously.
for (i=0; i<=7; i++)
  {
    mat.multVecMatrix(locVert[i], globVert[i]);
    globVert[i].getValue(p[i][0], p[i][1], p[i][2], w);

    // As mentioned above, the position of the center of gravity of the cube
    // has to be added to the coordinates of the vertices since the matrix 'mat'
    // doesn't contain this information.
    for (j=0; j<3; j++)
      p[i][j]+=position.values[j];

  }

// Setting all force values to zero.
for(i=0; i<=5; i++)
  force.values[i]=0;
```

```
// Setting single penetration values of inner cube into bounding box to zero.
for(i=0; i<=7; i++)
{
    for(j=0; j<3; j++)
  pt[i][j]=0.0;
}
// Setting total penetration value to zero.
for(j=0; j<3; j++)
  ptotal[j]=0.0;


//Calculation of the penetration of each vertice in x, y and z
for(i=0; i<=7; i++)
{
  for(j=0; j<=2; j++)
    {
  if(p[i][j]>sb)
    {
      pt[i][j]=-p[i][j]+sb;
      //Calculating the maximum penetration on the positive side.
      if (pt[i][j]<maxpos[j])
        maxpos[j]=pt[i][j];
    }
  if(p[i][j]<-sb)
    {
      pt[i][j]=-p[i][j]-sb;
      //Calculating the maximum penetration on the negative side.
      if (pt[i][j]>maxneg[j])
        maxneg[j]=pt[i][j];
        }
    }
}

// Calculating the sum of all penetrations in x, y and z
for(i=0; i<=7; i++)
{
for(j=0; j<3;j++)
  ptotal[j]+=pt[i][j];
}

// Initializing gain values of the controller.
for (i=0; i<6; i++)
{
  gains.values[i].p = 0;
  gains.values[i].d = 0;
  gains.values[i].ff = 0;
}
gains.values[2].ff=5.0;
```

```
// Setting gains for PD controller only if there is a penetration.
if(fabs(ptotal[0])!=0.0)
   {
     gains.values[0].p=15.0;
     gains.values[0].d=0.03;
     gains.values[4].d=0.2;
   }

if(fabs(ptotal[1])!=0.0)
   {
     gains.values[1].p=15.0;
     gains.values[1].d=0.03;
     gains.values[3].d=0.2;
   }

if(fabs(ptotal[2])!=0.0)
   {
     gains.values[2].p=30.0;
     gains.values[2].d=0.03;
   }

// Calculating forces for x, y and z axes of the flotor. It has to be divided by four because
// the penetration of the four vertices were summed together previously.
for(j=0; j<3;j++)
   force.values[j]=ptotal[j]*gains.values[j].p/4;

// The torque for each axis is computed by the sum of the penetrations of the vertices multiplied
// by half of the side length of the inner cube. The different signs for the single penetration
// values are due to the fact that penetrations in different positions and different orientations
// cause either negative or positive torque on the relative axis.

ptotal[3]=(-pt[0][1]-pt[0][2]+pt[1][1]-
          pt[1][2]+pt[2][1]-pt[2][2]-
          pt[3][1]-pt[3][2]-pt[4][1]+
          pt[4][2]+pt[5][1]+pt[5][2]+
          pt[6][1]+pt[6][2]-pt[7][1]+
          pt[7][2]);


ptotal[4]=(pt[0][0]-pt[0][2]-pt[1][0]-
          pt[1][2]-pt[2][0]+pt[2][2]+
          pt[3][0]+pt[3][2]+pt[4][0]-
          pt[4][2]-pt[5][0]-pt[5][2]-
          pt[6][0]+pt[6][2]+pt[7][0]+
          pt[7][2]);


ptotal[5]= (pt[0][0]+pt[0][1]+pt[1][0]+
          pt[1][1]+pt[2][0]-pt[2][1]+
          pt[3][0]-pt[3][1]-pt[4][0]+
          pt[4][1]-pt[5][0]+pt[5][1]-
          pt[6][0]-pt[6][1]-pt[7][0]-
          pt[7][1]);

// The gains for rotation are only turned on if the torque is different than zero.
if(fabs(ptotal[3])!=0.0)
  { gains.values[3].p=10.0;
gains.values[3].d=0.2;}
if(fabs(ptotal[4])!=0.0)
  {gains.values[4].p=10.0;
gains.values[4].d=0.2;}
if(fabs(ptotal[5])!=0.0)
  {gains.values[5].p=10.0;
gains.values[5].d=0.2;}


// Computing torques. The unit of the gain values for rotation is by default Nm/radians. For consistency
// they have to be transformed to Nm/degrees. 0.7 is an additional scale factor.
for(j=3; j<6; j++)
  force.values[j]=-0.7*ptotal[j]*gains.values[j].p*2*M_PI/360;

// Since the forces on the flotor are not applied on the center of the handle,
// but rather 13.3 mm underneath that point, an additional torque has to balance this effect.
force.values[4]+=force.values[0]*0.133;
force.values[3]-=force.values[1]*0.133;

// To make sure that the flotor isn't getting out of range of the optical sensors a virtual
// limit is set to the flotor's orientation.
static const float virtlim=4.0;
 for (i=3; i<6; i++)
  {
    if (position.values[i]>virtlim)
  {
    force.values[i]-=(position.values[i]-virtlim);
    gains.values[i].d=0.15;
    gains.values[i].p=50.0;
  }
    if (position.values[i]<-virtlim)
  {
    force.values[i]-=(position.values[i]+virtlim);
    gains.values[i].d=0.15;
    gains.values[i].p=50.0;
  }
  }

  // As mentioned above, the device generates a small torque around the z axis whenever
    // there is a force in that direction. It has been found out that the relation between
    // this force and torque can be described very precisely by a linear function. The
    // parameters for this linear function might vary from device to device.
 if (torque_flag){
     if (force.values[2]>0.0){
       torquebalance=force.values[2]*0.045+0.2;
       gains.values[5].p=10;
     }
     else if (force.values[2]<0.0){
       torquebalance=force.values[2]*0.018+0.2;
       gains.values[5].p=10;
     }
     else{
       torquebalance=0.2;
       gains.values[5].p=10;}
   }

  // Virtual coupling: The forces calculated above are now converted to a position offset,
  // given the gains of the PD controller.
  for (i=0; i<3; i++)
   despos.values[i]=position.values[i]+force.values[i]/gains.values[i].p;

  for (i=3; i<6; i++)
   despos.values[i]=position.values[i]+force.values[i]/(gains.values[i].p*2*M_PI/360);

  despos.values[5]+=torquebalance;

  // Calculating the position of the cube for the graphical representation. The position of the cube
  // in the graphics doesn't always correspond to the flotor's position. When the cube is penetrating
  // the outer box, the graphics show the cube just on the boundary of the box, so the user has the
  // impression that he or she is not able to penetrate the walls of the box. Since the haptic
  // device has a finite stiffness it is not possible to achieve such a behaviour.

  for(i=0; i<6; i++)
     {
       if (gains.values[i].p<0.01)
         despos.values[i]=position.values[i];
     }
  for (i=0; i<3; i++)
    b[i]=position.values[i]+maxpos[i]+maxneg[i];

 for(i=0; i<3;i++)
 despos.values[i]/=1000;

for(i=3; i<6;i++)
  despos.values[i]*=(2*M_PI/360);

 for(i=0; i<3;i++){
   gains.values[i].p*=1000;
   gains.values[i].d*=1000;
 }


}
```

```
// This callback function is called by the controller at a frequency rate of 1000 Hz
int tick_callback_handler(ml_device_handle_t device_hdl, ml_position_t *position2)
{
  static int result;
   if(setflag==0){
  // Calculation of the new desired pose of the flotor
  getPos(* position2);

  // Set gains for the pd controller and set the desired position.
  ml_SetGainVecAxes(device_hdl, ML_GAINSET_TYPE_NORMAL, gains);
  result = ml_SetDesiredPosition(device_hdl, despos);
   }
  return 0;
}


static const int BUF_SIZE = 256;
static char cmdbuf[BUF_SIZE];

// This is the control function. It initializes communication between client and host computer.
// Furthermore the flotor is floated to the origin of the haptic device.
void *controlBox(void *arg)
{
  int status = 0;
  int result;
  int i;

// Gains of all axes are set to zero.
for (i=0; i<=5; i++)
{
    gains.values[i].p = 0;
    gains.values[i].d = 0;
    gains.values[i].ff = 0;
}

  //connect to the device
  if(ml_Connect(&device_hdl, HOST_NAME) != ML_STATUS_OK)
    {
      printf("Unable to connect to MLHI device! Check name or IP address!");
      exit(-1);
    }
  printf("Connected. Device is taking off...");

  // Flotor takes off, e.g. moves to zero position. The gains which are used are preset in the function.
  if ((result = ml_Takeoff(device_hdl)) != ML_STATUS_OK)
    {
      printf("Error while taking off");
      ml_Disconnect(device_hdl);
      exit(-1);
    }
  // Waiting for the flotor to take off
  sleep(2);
  printf("Success");

  bool done =false;

// Menu loop. The user can choose one of two options: start demo and stop demo and land. When stopping
// the demo the flotor should be held by one hand and the cube shouldn't touch the walls of the
// cube(zero gains), otherwise instabilities could occur.

//do
  //{
    printf("Press 1 to start demo \n");
    printf("Press 2 to stop and shut down \n");
    printf("\n");

    //reads keyboard input
    /* if (fgets(cmdbuf, BUF_SIZE, stdin) == NULL)
      {
       printf("Invalid input\n");
       continue;
      }
    char key =cmdbuf[0];
    */
    //if(key == '1')
    //{
       // Register the callback function which is called by the controller at
       // a frequency rate of 1000 Hz.
       ml_RegisterCallbackTick(device_hdl, tick_callback_handler);
       //}
}
    // Setting flag for stopping the program
       /*    else if (key == '2')
        done=true;
    else
        printf("Not a valid input!!");

      } while(!done);
       */
void stopDemo()
{
int status = 0;
  int result;
    int i;
setflag=1;
// usleep(2000);
    // When the program is shutting down the tick callback should be unregistered

 gains.values[0].p =gains.values[1].p =2000;
gains.values[2].p = 8000.0;
 gains.values[3].p =gains.values[4].p =gains.values[5].p = 30.0;
 gains.values[3].d =gains.values[4].d =gains.values[5].d = 0.30;
    gains.values[0].d = gains.values[1].d =  gains.values[2].d =30;
    gains.values[2].ff = 4.8;

    ml_SetGainVecAxes(device_hdl, ML_GAINSET_TYPE_NORMAL, gains);

    printf("here1\n");

    ml_UnregisterCallbackTick(device_hdl);
    sleep(1);

    printf("here2\n");

    result = ml_Takeoff(device_hdl);
    //printf("** FINISHING cube in cube demo **\n\n");

    // "liftoff" again just to zero position

    //printf("taking off...\n");

    //if ((result = ml_Takeoff(device_hdl)) != ML_STATUS_OK)
    //printf("MLHI Error occurred while landing flotor.\n");

    sleep(2);

    //printf("Success.\n\n");
    //printf("6. Done! Disconnect from MLHI..."); fflush(stdout);

    // When the process ends the device has to be disconnected. The controller can not be
    // connected to more than one process at a time.
printf("ends successful 1\n");
    if ((result = ml_Disconnect(device_hdl)) != ML_STATUS_OK)
    {
      printf("MLHI Error occurred while disconnecting from MLHI.\n");
      status = -1;
    }
printf("ends successful\n");
    exit(status);

}
```

B Texture Plate Demo

The following function (”getPos”) is the most important part of the the Texture Plate demo. It calculates all the forces to be applied to the flotor and is called by the ”tickCallback” function at a frequency rate of 1000 Hz. The structure of the program is similar to the one used in the Cube in Cube demo and therefore the initialization of the haptic device and the graphics threads are not present in this appendix.

```
/************************  calculate flotor forces  ***************************/

int getPos(ml_position_t & position)
{
    static int i=0;
    static float pg=8.34;
    static int qnow;
    static ml_position_t positionold=position;

    pb.pos[0]=b[0]=pg*position.values[0];
    pb.pos[1]=b[1]=pg*position.values[1];
    pb.pos[2]=b[2]=position.values[2];

    impact[0]=impact[1]=impact[2]=0;

    for (i=0; i<6; i++)
        {
        pb.vel[i]=(position.values[i]-positionold.values[i])*1000;
        pb.counter[i]++;
        pb.force[i]=0.0;
        }

    positionold=position;

    //filter();
    qnow=findquad();
    sideimpact();

    if (pen(qnow))
      insideimpact(&qnow);

    setsinepar(qnow);

    calcpenforce(qnow);
    //calcfric(&pb, qnow);


    gains.values[0].p=gains.values[1].p=gains.values[2].p=0.0;
    gains.values[0].d=gains.values[1].d=gains.values[2].d=0.0;
    gains.values[3].p=gains.values[4].p=gains.values[5].p=50.0;
    gains.values[3].d=gains.values[4].d=gains.values[5].d=0.3;
    gains.values[2].ff=3.0;

    for (i=0; i<3; i++)
        {
        if (pb.force[i]!=0.0)
          {
            gains.values[i].p=5.0;
            gains.values[i].d=0.02;
            despos.values[i]=position.values[i]+pb.force[i]/gains.values[i].p;

          }
        }

    for (i=0; i<2; i++)
        {
        if (impact[i])
            {
            gains.values[4-i].d=0.7;
            gains.values[i].d=0.07;
            }
        }

    if (impact[2])
        gains.values[2].d=0.065;
    qold=qnow;
    despos.values[3]=despos.values[4]=despos.values[5]=0.0;

}

/**************************** moving average filter for position data ***********************/

void filter()
{
    static float v[fil][3];
    static int count=0;
    static float velsum[3]={0,0,0};
    int i, j;

    if (count>=pb.num)
        {
        for(i=0; i<3; i++)
            {
            pb.vel[i]+=(pb.vel[i]-v[pb.num-1][i])/pb.num;
            v[0][i]=pb.vel[i];
            }

        for(i=pb.num-1; i>0; i--)
```

```
            {
            for(j=0; j<3; j++)
                v[i][j]=v[i-1][j];
            }
        }
        count+=1;
}

/**************************** finding the pattern on which the probe is located ********************/

int findquad()
{
    static int quad;
    if ((pb.pos[0]>=0.0) && (pb.pos[1]>=0.0))
        quad=1;
    else if ((pb.pos[0]<0.0) && (pb.pos[1]>=0.0))
        quad=2;
    else if ((pb.pos[0]<0.0) && (pb.pos[1]<0.0))
        quad=3;
    else if ((pb.pos[0]>=0.0) && (pb.pos[1]<0.0))
        quad=4;
    return quad;
}

/*************************** detecting if the probe is penetrating the surface *********************/

bool pen(int qnow)
{
    static float penet=0.0;
    static float h;
    if (qnow==2){
        h=0.15*sin((pb.pos[0]+5.6)*2*M_PI/4.0)-0.16;
        penet=h-pb.pos[2];}
    else if (qnow==3){
        h=0.2*sin((pb.pos[1]+0.375)*2*M_PI)-0.16;
        penet=h-pb.pos[2];}
    else if (qnow==4){
        h=0.0;
        penet=h-pb.pos[2];}

    if (penet>0.0)
        return true;
    else
        return false;
}

/********** calculates the impact force when the flotor is hitting the inner walls of the plate **********/

void insideimpact(int *qnow)
{
```

```
static float iimpgain=1.0;
static float sgain=1.0;

    if (qold!=*qnow)
        {
        if ((qold==1) && (*qnow==4))
            {
              pb.force[1]+=-sgain*pb.pos[1];
              b[1]=0.0;
              if ((pb.counter[1]>implim) && (pb.outside[1]==1))
                {
                pb.force[1]+=iimpgain*fabs(pb.vel[1]*pb.vel[1]*pb.vel[1]*pb.vel[1]/scale);
                pb.counter[1]=0;
                impact[1]=1;
                }
            *qnow=1;
            pb.outside[1]=0;
            }

        else if ((qold==1) && (*qnow==2))
            {
              pb.force[0]+=-sgain*pb.pos[0];
              b[0]=0;
            if ((pb.counter[0]>implim) && (pb.outside[0]==1))
                {
                pb.force[0]+=iimpgain*fabs(pb.vel[0]*pb.vel[0]*pb.vel[0]*pb.vel[0]/scale);;
                pb.counter[0]=0;
                impact[0]=1;
                }
            *qnow=1;
            pb.outside[0]=0;
            }
        else if ((qold==1) && (*qnow==3))
            {
              pb.force[0]+=-sgain*pb.pos[0];
              pb.force[1]+=-sgain*pb.pos[1];
              b[0]=b[1]=0.0;
            if ((pb.counter[0]>implim) && (pb.outside[0]==1))
                {
                pb.force[0]+=iimpgain*fabs(pb.vel[0]*pb.vel[0]*pb.vel[0]*pb.vel[0]/scale);
                pb.counter[0]=0;
                impact[0]=1;
                }
            if ((pb.counter[1]>implim) && (pb.outside[1]==1))
                {
                pb.force[1]+=iimpgain*fabs(pb.vel[1]*pb.vel[1]*pb.vel[1]*pb.vel[1]/scale);
                pb.counter[1]=0;
                impact[1]=1;
                }
            *qnow=1;
```

```
                pb.outside[0]=0;
                pb.outside[1]=0;
                }
            }

    qold=*qnow;
}

/******* coordinate transformation for the pattern rotated by 45 degrees with respect to the other patterns *********/

void cotrans(float alpha)
{
  pb.pos[0]=pb.pos[0]*cos(alpha)+pb.pos[1]*sin(alpha);
  pb.pos[1]=-pb.pos[0]*sin(alpha)+pb.pos[1]*cos(alpha);
}

void setsinepar(int qnow)
{
    static float ab=16.3*cos(-M_PI/4)+23*sin(-M_PI/4);

    if (qnow==2)
        {
        sine.xoff=5.6;
        sine.yoff=0.16;
        sine.amp=0.15;
        sine.per=4.0;
        sine.n=1;
        }

    else if (qnow==3)
        {
        sine.n=1;
        sine.xoff=0.375;
        sine.yoff=0.16;
        sine.amp=0.2;
        sine.per=1.0;
        cotrans(M_PI/2);
        sine.n=2;
        }

    else if (qnow==1)
    {
         sine.xoff=-ab+0.5;
        sine.yoff=2.53;
        sine.amp=0.1;
        sine.per=2.0;
        cotrans(-M_PI/4);
        sine.n=0;
    }
}
```

```
/******************* calculates the look-up table for sine waves *****************/

void calclookup()
{
  static int i, j;
  static float a, b;
      for (j=0; j<3; j++)
      {
        setsinepar(j+1);
        for (i=0; i<lookt.sizet; i++)
          {
        a=lookt.x[j][i]=sine.per*(float)i/((float)lookt.sizet-1);
        b=lookt.y[j][i]=sine.amp*sin(lookt.x[j][i]*2*M_PI/sine.per);
        lookt.ds[i]=10000;
          }
      }


}

/************** calculates the flotor force according to its penetration into the surface **************/

void calcpenforce(int quad)
{
    static float h, pen, xnew, ynew, x1, x2, min, rep;
    static int idx1, idx2, idxmin, i;
    static float it2=(float)lookt.sizet;
    static float vimpgain=20.0;
    static float vergain=10.0;
    static float horgain=1.0;
    static float pforce;
    int g=sine.n;
    if (quad==4)
      h=0.0;

    else
      h=sine.amp*sin((pb.pos[0]+sine.xoff)*2*M_PI/sine.per)-sine.yoff;

    pen=h-pb.pos[2];
      if (pen>0.0)
        {
        if (quad==4)
        {
        min=pen;
        pb.force[2]=vergain*min;
        }
        else
          {
        ynew=pb.pos[2]+sine.yoff;
```

```
b[2]=h;
xnew=fmod(pb.pos[0]+sine.xoff, sine.per);
if (xnew<0.0)
    xnew=sine.per+xnew;
if (xnew<=sine.per/4)
    {
    x2=xnew;
    x1=sine.per*asinf((h-pen+sine.yoff)/sine.amp)/(2*M_PI);
    }
if ((xnew>sine.per/4) && (xnew<=sine.per/2))
    {
    x1=xnew;
    x2=sine.per/2-sine.per*asinf((h-pen+sine.yoff)/sine.amp)/(2*M_PI);
    }

if ((xnew>sine.per/2) && (xnew<=3*sine.per/4))
    {
    x1=xnew;
    x2=sine.per/2+fabsf(sine.per*asinf((h-pen+sine.yoff)/sine.amp)/(2*M_PI));
    }

if ((xnew>3*sine.per/4) && (xnew<sine.per))
    {
    x2=xnew;
    x1=sine.per-fabsf(sine.per*asinf((h-pen+sine.yoff)/sine.amp)/(2*M_PI));
    }

if (pb.pos[2]<-(sine.amp+sine.yoff))
    {
    if (xnew<=sine.per/4)
        {
        x1=-sine.per/4;
        x2=xnew;
        }
    if ((xnew<=3*sine.per/4) && (xnew>sine.per/4))
        {
        x1=xnew;
        x2=3*sine.per/4;
        }
    if ((xnew<=sine.per) && (xnew>3*sine.per/4))
        {
        x1=3*sine.per/4;
        x2=xnew;
        }
    }

idx1=(int)floor(x1*it2/sine.per);
idx2=(int)ceil(x2*it2/sine.per);
if (idx2>=lookt.sizet)
    idx2=lookt.sizet-1;
```

```
if (idx1>=0)
    {
    for (i=idx1; i<=idx2;i++)
        lookt.ds[i]=(lookt.x[g][i]-xnew)*(lookt.x[g][i]-xnew)+(lookt.y[g][i]-ynew)*(lookt.y[g][i]-ynew);

    min=lookt.ds[idx1];
    idxmin=idx1;
    for (i=idx1+1; i<=idx2; i++)
        {
        if (lookt.ds[i]<min)
            {
            min=lookt.ds[i];
            idxmin=i;
            }
        }
    }

else if (idx1<0)
    {
    idx1=lookt.sizet+idx1;

    for (i=idx1; i<lookt.sizet;i++)
        lookt.ds[i]=(lookt.x[g][i]-sine.per-xnew)*(lookt.x[g][i]-sine.per-xnew)+(lookt.y[g][i]-ynew)*(lookt.y[g]

    for (i=0; i<=idx2;i++)
        lookt.ds[i]=(lookt.x[g][i]-xnew)*(lookt.x[g][i]-xnew)+(lookt.y[g][i]-ynew)*(lookt.y[g][i]-ynew);

    min=lookt.ds[idx1];
    idxmin=idx1;

    for (i=idx1+1; i<lookt.sizet; i++)
        {
        if (lookt.ds[i]<min)
            {
            min=lookt.ds[i];
            idxmin=i;
            }
        }
    for (i=0; i<=idx2; i++)
        {
        if (lookt.ds[i]<min)
            {
            min=lookt.ds[i];
            idxmin=i;
            }
        }
    }

rep=sqrt(1+sine.amp*cos(lookt.x[g][idxmin]*2*M_PI/sine.per)*sine.amp*cos(lookt.x[g][idxmin]*2*M_PI/sine.per));
```

```
        pforce=horgain*(-min*sine.amp*cos(lookt.x[g][idxmin]*2*M_PI/sine.per)/rep);
        pb.force[2]+=vergain*min/rep;

        min=0.0;

        if (quad==2)
                pb.force[0]+=pforce;

        if (quad==3)
            pb.force[1]+=pforce;

        if (quad==1)
            {
            pb.force[1]+=-pforce/sqrt(2);
            pb.force[0]+=pforce/sqrt(2);
            }
          }
        if ((pb.counter[2]>implim) && (pb.outside[2]==1))
            {
            pb.force[2]+=vimpgain*fabs(pb.vel[2]*pb.vel[2]*pb.vel[2]*pb.vel[2]/scale);
            pb.counter[2]=0;
            impact[2]=1;
            }

        pb.outside[2]=0;
        }

          else
        pb.outside[2]=1;
}

/********** calculates the impact force when the probe hits the outer side walls of the plate **********/

void sideimpact()
{
    static float simpgain=1.0;
    static float sgain=1.0;
    static int i;
    static float edge=8.34*6.0;
    for(i=0; i<2; i++)
        {
        if (pb.pos[i]>edge)
            {
            pb.force[i]+=sgain*(edge-pb.pos[i]);
            b[i]=edge;
            if ((pb.outside[i]==1) && (pb.counter[i]>implim))
                {
                pb.force[i]+=-simpgain*fabs(pb.vel[i]*pb.vel[i]*pb.vel[i]*pb.vel[i]/scale);
                pb.counter[i]=0;
                impact[i]=1;
                }
            pb.outside[i]=0;
            }
        else if (pb.pos[i]<-edge)
            {
            pb.force[i]+=sgain*(-edge-pb.pos[i]);
            b[i]=-edge;
            if ((pb.outside[i]==1) && (pb.counter[i]>implim))
                {
                pb.force[i]+=simpgain*fabs(pb.vel[i]*pb.vel[i]*pb.vel[i]*pb.vel[i]/scale);
                pb.counter[i]=0;
                impact[i]=1;
                }
            pb.outside[i]=0;
            }
        else
            pb.outside[i]=1;
       }
}
```

C Touch the Bunny Demo

The following code contains only the characteristic parts of the Touch the Bunny demo. It is therefore not sufficient to run the demo. The section of the main function represents how the triangular mesh is extracted from the .iv-file and stored in a data structure. The "graphicsCallback" function is responsible for the scene update. The most important thing is the calculation of the workspace and camera positions as they are changed during the positioning mode.

The function "calcnorm" calculates orientation and direction of the normals. "Searchpart" is responsible for finding the three neighboring triangles of each triangle. "Mergesort" sorts the x-coordinate of the triangle middle points.

```
/**********************************************************/
int main(int argc, char **argv)
{
...

SoInput myInput;
if (! myInput.pushFile("/usr0/obmartin/bunny.iv"))
    printf("Cannot open file!\n");

fileContents = SoDB::readAll(&myInput);
```

```
if (fileContents == NULL)
    printf("File doesn't contain anything!\n");

float val[3];
float v1[3], v2[3], no[3];
const SbVec3f *Vector=new SbVec3f;
triangles tmp;
int a,b,g,i,j;
int stmp, s1, s2;

SoCoordinate3 *Coords=new SoCoordinate3;
Coords=(SoCoordinate3 *) fileContents->getChild(0);
SoIndexedFaceSet *Idx=(SoIndexedFaceSet *)fileContents->getChild(3);
const int32_t * pointer1=Idx->coordIndex.getValues(0);

int NumVert2=Coords->point.getNum();
int NumTri2=(3*Idx->coordIndex.getNum()/4)/3;

for(i=0; i<NumVert2; i++)
    {
    Vector=Coords->point.getValues(i);
    Vector->getValue(val[0], val[1], val[2]);
    for(j=0; j<3; j++)
        vert[i][j]=val[j];
    }


for(i=0; i<NumTri2; i++)
    {
    for(j=0; j<3; j++)
        {
        tri[i].vidx[j]=*pointer1;
        pointer1++;
        }
        pointer1++;
    }

for(i=0; i<NumTri2; i++)
    {
    for(j=0; j<3; j++)
        tri[i].mid[j]=(vert[tri[i].vidx[0]][j]+vert[tri[i].vidx[1]][j]+vert[tri[i].vidx[2]][j])/3;
    }

for(i=0; i<NumTri2; i++)
    {
    numbers[i].val=tri[i].mid[0];
    numbers[i].idx=i;
    }

mergesort(0, NumTri2-1);
```

```
triangles temp2[NumTri2];

for(i=0; i<NumTri2;i++)
    temp2[i]=tri[numbers[i].idx];

for(i=0; i<NumTri2;i++)
    tri[i]=temp2[i];

searchpart();
calcnorm();

pthread_t thread;
pthread_create(&thread,NULL,controlBox,NULL);


...


}


// this is the callback function for the graphics (100 Hz update frequency)
void graphics_callback(void *data, SoSensor *sense)
{

static float dbandp=0.0005;
static float srotX, srotY, srotZ;
static int flagf=0;
static float scalexy=10.0;
static float scalez=50.0;
static float dband=0.04;
static float sgam, cgam, sbet, cbet, salp, calp;
static float a11, a12, a13, a21, a22, a23, a31, a32, a33;
static int count, hflag;
static SbVec4f srcrpoint;
static  SbVec4f dstrpoint;
srcrpoint.setValue(-0.002760, -0.043475, 0.137835, 1);

static SbMatrix matoldrot(1,0,0,0,
                         0,1,0,0,
                         0,0,1,0,
                         0,0,0,1);
SbMatrix camtrans(1,0,0,0,
                   0,cos(-0.8),-sin(-0.8),0,
                   0,sin(-0.8),cos(-0.8),0,
                   -0.002760, -0.143475, 0.117835, 1);
SbMatrix camtrans2(1,0,0,0,
                    0,1,0,0,
                    0,0,1,0,
                    0,0,0,1);
SbMatrix camtrans3(1,0,0,0,
                    0,1,0,0,
                    0,0,1,0,
                    0,0,0,1);
SbMatrix trans(1,0,0,0,
                0,1,0,0,
                0,0,1,0,
                0,0,0,1);

transsphere->translation.setValue(projcheat[0], projcheat[1], projcheat[2]);

if (penc==1)
    matsph->diffuseColor.setValue(1.0, 0.0, 0.0);
else
    matsph->diffuseColor.setValue(1.0, 1.0, 0.0);

if (penc2==1)
    bxmat->diffuseColor.setValue(1.0, 0.0, 0.0);
else
    bxmat->diffuseColor.setValue(0.0, 0.8, 0.8);

static int i;
static float posimp[3];
for(i=0;i<3;i++)
    posimp[i]=tri[numbertri].mid[i]*50;

transspherei->translation.setValue(posball2[0], posball2[1], posball2[2]);

if (fabs(posX)<dbandp)
     posX=0.0;
else
    {
    if(posX>dbandp)
    posX-=dbandp;
    else if(posX<-dbandp)
    posX+=dbandp;
    }

if (fabs(posY)<dbandp)
    posY=0.0;
else
    {
    if(posY>dbandp)
    posY-=dbandp;
    else if(posY<-dbandp)
    posY+=dbandp;
    }

if (fabs(posZ)<dbandp)
```

```
    posZ=0.0;
else
    {
    if(posZ>dbandp)
    posZ-=dbandp;
    else if(posZ<-dbandp)
    posZ+=dbandp ;
    }

if (fabs(rotX)<dband)
    rotX=0.0;

if (fabs(rotY)<dband)
    rotY=0.0;
if (fabs(rotZ)<dband)
    rotZ=0.0;

sgam = sin(rotX/scalexy);
cgam = cos(rotX/scalexy);
sbet = sin(rotY/scalexy);
cbet = cos(rotY/scalexy);
salp = sin(rotZ/scalez);
calp = cos(rotZ/scalez);
float sdf=Fraum[0];
a11 = calp*cbet;
a12 = calp*sbet*sgam-salp*cgam;
a13 = calp*sbet*cgam+salp*sgam;
a21 = salp*cbet;
a22 = salp*sbet*sgam+calp*cgam;
a23 = salp*sbet*cgam-calp*sgam;
a31 = -sbet;
a32 = cbet*sgam;
a33 = cbet*cgam;
static SbVec3f tr, sc, cr, csc;
static SbRotation ro, sb, co, csb;
static float scdown=0.5;
SbMatrix mat(a11, a12, a13, 0, a21, a22, a23, 0, a31, a32, a33, 0, -scdown*posX, -scdown*posY,-scdown*posZ, 1);

if (trsf2==0)
    mat.multRight(matold);

matold.getTransform(tr, ro, sc, sb);
SbRotation sbrot2=SbRotation(matold);
matoldglob=matold;
matoldrot=matold;
matold=mat;
camtrans.multRight(mat);
camtrans2.multRight(matoldrot);
camtrans.getTransform(cr, co, csc, csb);
SbRotation sbrotc=SbRotation(camtrans);
```

```
transbox->translation.setValue(tr);
tr.getValue(trv[0], trv[1], trv[2]);
transbox->rotation.setValue(sbrot2.getValue());
sbrot2.getValue(sbr[0], sbr[1], sbr[2], sbr[3]);
pltr->translation.setValue(tr);
pltr->rotation.setValue(sbrot2.getValue());
pltr2->translation.setValue(tr);
pltr2->rotation.setValue(sbrot2.getValue());
pltr3->translation.setValue(tr);
pltr3->rotation.setValue(sbrot2.getValue());
pltr4->translation.setValue(tr);
pltr4->rotation.setValue(sbrot2.getValue());
tr.getValue(offb[0], offb[1], offb[2]);

matold.multVecMatrix(srcrpoint, dstrpoint);
myCamera->position.setValue(cr);//dstrpoint.getValue());
myCamera->orientation.setValue(sbrotc.getValue());
   count++;
}

int getPos(ml_position_t & position)
{

static float edge[3];
static float force[3];
static float tsh=0.005;
static int val, val2, midd, idxmin, count, i, j, countold, idxfound, idxminold;
static int low = 0;
static int high = 69451;
static float distsq[NumTri];
static float d[3][3];
static float penv[3];
static float scap, scale;
static float fn[3];
static bool stop;
static bool probimphist;
static typedef struct {
                int id;
                float sc;
                } shist;
static shist idcs[NumTri];
static shist idcsold[NumTri];
static float dmin=1000;
static int counterha, counterha1, counterha2;
idxmin=0;
count=0;
static float w;
static int first;
static float sgam, cgam, sbet, cbet, salp, calp;
static float a11, a12, a13, a21, a22, a23, a31, a32, a33;
static float balancefx, balancefy;
static float xyrlim=0.1;
static float zrlim=0.1;

rotX=rotY=rotZ=0.0;
sgam = sin(rotX);
cgam = cos(rotX);
sbet = sin(rotY);
cbet = cos(rotY);
salp = sin(rotZ);
calp = cos(rotZ);
float sdf=Fraum[0];
a11 = calp*cbet;
a12 = calp*sbet*sgam-salp*cgam;
a13 = calp*sbet*cgam+salp*sgam;
a21 = salp*cbet;
a22 = salp*sbet*sgam+calp*cgam;
a23 = salp*sbet*cgam-calp*sgam;
a31 = -sbet;
a32 = cbet*sgam;
a33 = cbet*cgam;

for(i=0; i<6; i++)
    despos.values[i]=0.0;

balancefx=balancefy=0.0;
static float ab[4][4];
static  float ac[4][4]={{1,0,0,0}, {0,1,0,0}, {0,0,1,0}, {offset[0], offset[1], offset[2], 1}};
static int ctrsf;
static int firstt=1;

if(trsf2==1)
    {
    ctrsf++;
    matold.setValue(ac);
    if(ctrsf==1)
        {
        trsf2=0;
        ctrsf=0;
        }
    }

if (trsf==1)
    {
    posX=0.5*position.values[0];
    posY=0.5*position.values[1];
    posZ=0.5*position.values[2];
    rotX=1.5*position.values[3];
    rotY=1.5*position.values[4];
    rotZ=4*position.values[5];
    for(i=0; i<6;i++)
      despos.values[i]=0.0;

    gains.values[3].p=gains.values[4].p=1.5;
    gains.values[5].p=5.0;
    gains.values[5].d=gains.values[3].d=gains.values[4].d=0.150;
    gains.values[2].ff=4.8;

    if(first==1)
        gains.values[0].p=gains.values[1].p=gains.values[2].p=0.0;

    first=0;
    for(i=0;i<2; i++)
        {
        if (gains.values[i].p<1500)
            {
            gains.values[i].p+=5;
            gains.values[i].d=30;
            gains.values[4-i].d=0.6;
            }
        }

    if (gains.values[2].p<3000)
        {
        gains.values[2].p+=10;
        gains.values[2].d=30;
        }

    balancefx=position.values[0]*gains.values[0].p*0.13/gains.values[4].p;
    balancefy=position.values[1]*gains.values[1].p*0.13/gains.values[3].p;
}

if(trsf==0)
    {
    posX=posY=posZ=0.0;
    rotX=rotY=rotZ=0.0;
    first=1;
    static SbVec3f tr, sc;
    static SbRotation ro, sb;
    SbMatrix transw(a11, a12, a13, 0,
               a21, a22, a23, 0,
               a31, a32, a33, 0,
               offb[0], offb[1], offb[2], 1);

    static SbMatrix transwinv(a11, a12, a13, 0,
               a21, a22, a23, 0,
               a31, a32, a33, 0,
               0, 0, 0, 1);

    static SbVec4f srcpoint;
    static  SbVec4f dstpoint;
    srcpoint.setValue(position.values[0]*100*boxscale,position.values[1]*100*boxscale, position.values[2]*100*boxscale,1

    for(i=0; i<3; i++)
        position.values[i]*=1000;

    static SbVec3f fr, fsc;
    static SbRotation fo, fsb;
    static float fa, fb, fc;
    SbMatrix frotat;
    matoldglobinv=matoldglob;
    fr.setValue(0,0,0);
    fsc.setValue(1,1,1);
    matoldglob.multVecMatrix(srcpoint, dstpoint);
    matoldglobinv=matoldglobinv.inverse();
    SbRotation sbrotf=SbRotation(matoldglobinv);
    frotat.setTransform(fr, sbrotf, fsc);
    dstpoint.getValue(pt[0], pt[1], pt[2], w);
    stop=false;
    val=val2=-1;
    dmin=1000;
    low=0;
    high=69451;
    static int countw1;
    static int countw2;
    countw1=countw2=0;

    while (1)
        {
        if(low>high)
            break;
        countw1++;
        midd = (low+high)/2;
        if (tri[midd].mid[0]>=(pt[0]-tsh))
            {
            if (tri[midd-1].mid[0]>(pt[0]-tsh))
                high = midd-1;
            else
                {
                val=midd;
                break;
                }
            }
        else if (tri[midd].mid[0]<(pt[0]-tsh))
            {
            if(tri[midd+1].mid[0]<(pt[0]-tsh))
                low = midd+1;
            else
                {
                val=midd;
```

```
                break;
                }
            }
       break;
       }
    }

if (val<0)
    val=0;
stop=false;
low = val;
high = 69451 - 1;

while (1)
    {
    if(low>high)
        break;
    countw2++;
    midd = (low+high)/2;
    if (tri[midd].mid[0] >= (pt[0]+tsh))
        {
        if (tri[midd-1].mid[0]>=(pt[0]+tsh))
            high = midd-1;
        else
            {
            val2=midd;
            break;
            }
        }
    else if (tri[midd].mid[0]<(pt[0]+tsh))
        {
        if(tri[midd+1].mid[0]<(pt[0]+tsh))
            low = midd+1;
        else
            {
            val2=midd;
            break;
            }
        }
    break;
    }
}

if (val2<0)
    {
    if (tri[NumTri2].mid[0]<=pt[0]+tsh)
        val2=69451;
    }

static bool probimp=false;
static bool fidx=false;
probimp=false;
count=0;

if (val2!=-1)
    {
    for(i=val; i<=val2; i++)
        {
        if(tri[i].mid[1]>pt[1]-tsh)
            {
            if(tri[i].mid[1]<pt[1]+tsh)
                {
                if(tri[i].mid[2]>pt[2]-tsh)
                    {
                    if(tri[i].mid[2]<pt[2]+tsh)
                        {
                        idcs[count].id=i;
                        count++;
                        probimp=true;
                        }
                    }
                }
            }
        }
    }

static int impflag=0;
static float bunny[3];
static float sign=1;
static float sctot[3][2], lengthn2n;
static float vett[3][3];
static int vt, countproj, k;
static float tempp[3];
static float norm2norm[3];
static float pointproj[3];
static float pprojsc;
static int n2n;
static float ptd[3];
static float ptdl, diffang, mindiffang;
static int countin;
static float vtv[3][3];
static float ang[3][2];
static float angtot[3];
static float triang[3];
static float vtvl[3];
static float fpointproj[3], fpointprojt[3];
static float dl[3];
static float ptvf[3], ptvft[3];
static float scp;
static bunnyears diffarr[10];
static int countdiff=0;
static int check;
static int idxminvor;
countdiff=0;
dmin=10000000;
countin=0;
idxmin=-1;
static int okflag1, okflag2;
okflag1=okflag2=0;

if ((impflag==1) && (probimp==false))
    printf("lost track!!!!!!\n");

static float mc=10000;
for(i=0; i<count; i++)
    {
    for(n2n=0; n2n<3; n2n++)
        ptd[n2n]=pt[n2n]-tri[idcs[i].id].mid[n2n];

    idcs[i].sc=ptd[0]*tri[idcs[i].id].nl[0]+ptd[1]*tri[idcs[i].id].nl[1]+ptd[2]*tri[idcs[i].id].nl[2];
    ptdl=sqrt(ptd[0]*ptd[0]+ptd[1]*ptd[1]+ptd[2]*ptd[2]);
    tempp[0]=ptd[1]*tri[idcs[i].id].nl[2]-ptd[2]*tri[idcs[i].id].nl[1];
    tempp[1]=-ptd[0]*tri[idcs[i].id].nl[2]+ptd[2]*tri[idcs[i].id].nl[0];
    tempp[2]=ptd[0]*tri[idcs[i].id].nl[1]-ptd[1]*tri[idcs[i].id].nl[0];
    norm2norm[0]=tempp[1]*tri[idcs[i].id].nl[2]-tempp[2]*tri[idcs[i].id].nl[1];
    norm2norm[1]=-tempp[0]*tri[idcs[i].id].nl[2]+tempp[2]*tri[idcs[i].id].nl[0];
    norm2norm[2]=tempp[0]*tri[idcs[i].id].nl[1]-tempp[1]*tri[idcs[i].id].nl[0];
    lengthn2n=sqrt(norm2norm[0]*norm2norm[0]+norm2norm[1]*norm2norm[1]+norm2norm[2]*norm2norm[2]);

    for(n2n=0; n2n<3; n2n++)
        norm2norm[n2n]/=lengthn2n;

    pprojsc=ptd[0]*norm2norm[0]+ptd[1]*norm2norm[1]+ptd[2]*norm2norm[2];

    for(n2n=0; n2n<3; n2n++)
        pointproj[n2n]=tri[idcs[i].id].mid[n2n]+pprojsc*norm2norm[n2n];

    for(vt=0;vt<3;vt++)
        {
        for(j=0; j<3; j++)
            d[vt][j]=pointproj[j]-vert[tri[idcs[i].id].vidx[vt]][j];
        }

    for(j=0; j<3; j++)
        dl[j]=sqrt(d[j][0]*d[j][0]+d[j][1]*d[j][1]+d[j][2]*d[j][2]);

    for(j=0; j<3; j++)
        {
        for(n2n=0; n2n<3; n2n++)
            d[j][n2n]/=dl[j];
        }

    for(j=0; j<3; j++)
        {
        vtv[0][j]=vert[tri[idcs[i].id].vidx[1]][j]-vert[tri[idcs[i].id].vidx[0]][j];
        vtv[1][j]=vert[tri[idcs[i].id].vidx[2]][j]-vert[tri[idcs[i].id].vidx[0]][j];
        vtv[2][j]=vert[tri[idcs[i].id].vidx[2]][j]-vert[tri[idcs[i].id].vidx[1]][j];
        }

    for(j=0; j<3; j++)
        vtvl[j]=sqrt(vtv[j][0]*vtv[j][0]+vtv[j][1]*vtv[j][1]+vtv[j][2]*vtv[j][2]);

    for(j=0; j<3; j++)
        {
        for(n2n=0; n2n<3; n2n++)
            vtv[j][n2n]/=vtvl[j];
        }

    sctot[0][0]=d[0][0]*vtv[0][0]+d[0][1]*vtv[0][1]+d[0][2]*vtv[0][2];
    sctot[0][1]=d[0][0]*vtv[1][0]+d[0][1]*vtv[1][1]+d[0][2]*vtv[1][2];
    sctot[1][0]=-d[1][0]*vtv[0][0]-d[1][1]*vtv[0][1]-d[1][2]*vtv[0][2];
    sctot[1][1]=d[1][0]*vtv[2][0]+d[1][1]*vtv[2][1]+d[1][2]*vtv[2][2];
    sctot[2][0]=-d[2][0]*vtv[1][0]-d[2][1]*vtv[1][1]-d[2][2]*vtv[1][2];
    sctot[2][1]=-d[2][0]*vtv[2][0]-d[2][1]*vtv[2][1]-d[2][2]*vtv[2][2];

    for(j=0; j<3; j++)
        {
        for(k=0; k<2; k++)
            ang[j][k]=acos(sctot[j][k]);

        angtot[j]=ang[j][0]+ang[j][1];
        }
    triang[0]=acos(vtv[0][0]*vtv[1][0]+vtv[0][1]*vtv[1][1]+vtv[0][2]*vtv[1][2]);
    triang[1]=acos(vtv[0][0]*vtv[2][0]+vtv[0][1]*vtv[2][1]+vtv[0][2]*vtv[2][2]);
    triang[2]=acos(vtv[1][0]*vtv[2][0]+vtv[1][1]*vtv[2][1]+vtv[1][2]*vtv[2][2]);
    diffang=0.0;
    for (j=0; j<3;j++)
        diffang+=angtot[j];

    if (i==0)
        {
        idxmin=1;
        mindiffang=diffang;
        }
    check=0;
    if (diffang<mindiffang)
        {
        if(probimphist==true)
            {
            if (tri[idxminvor].nl[0]*tri[idcs[i].id].nl[0]+tri[idxminvor].nl[1]*tri[idcs[i].id].nl[1]+tri[idxminvor].nl[
                check=1;
            }
        else
            check=1;
        if (check==1)
            {
            idxmin=i;
            mindiffang=diffang;
            ptvft[0]=ptd[0];
            ptvft[1]=ptd[1];
            ptvft[2]=ptd[2];
            fpointprojt[0]=pointproj[0];
            fpointprojt[1]=pointproj[1];
            fpointprojt[2]=pointproj[2];
            }
        }
    }

if (probimp==true)
    probimphist=true;

if (countdiff>10)
    printf("this is countdiff %d\n", countdiff);

ptvf[0]=ptvft[0];
ptvf[1]=ptvft[1];
ptvf[2]=ptvft[2];
fpointproj[0]=fpointprojt[0];
fpointproj[1]=fpointprojt[1];
fpointproj[2]=fpointprojt[2];
posball2[0]=tri[idcs[idxmin].id].mid[0];
posball2[1]=tri[idcs[idxmin].id].mid[1];
posball2[2]=tri[idcs[idxmin].id].mid[2];
numbertri=i;
fidx=false;
static float add[3];
static int inverted;
static float div;
static float amp=1.0;
static int counttouch;
scp=tri[idcs[idxmin].id].nl[0]*ptvf[0]+tri[idcs[idxmin].id].nl[1]*ptvf[1]+tri[idcs[idxmin].id].nl[2]*ptvf[2];

if((scp<0.0000)&&(probimp==true))
    {
    counttouch++;
    if (counttouch==100)
        counttouch=0;

    div=4;
    if (tri[idcs[idxmin].id].part[3]+tri[idcs[idxmin].id].part[4]+tri[idcs[idxmin].id].part[5]==0)
```

```
        div-=1;
    if (tri[idcs[idxmin].id].part[6]+tri[idcs[idxmin].id].part[7]+tri[idcs[idxmin].id].part[8]==0)
        div-=1;

    for(i=0; i<3; i++)
        {
        bunny[i]=(-scp*tri[idcs[idxmin].id].nl[i]-
            amp*scp*tri[tri[idcs[idxmin].id].part[0]].nl[i]-
            amp*scp*tri[tri[idcs[idxmin].id].part[3]].nl[i]-
            amp*scp*tri[tri[idcs[idxmin].id].part[6]].nl[i])/div;
        add[i]=-scp*tri[idcs[idxmin].id].nl[i];
        projcheat[i]=pt[i]+(0.001-scp)*tri[idcs[idxmin].id].nl[i];
        penc=1;
        }
    }

else
    {
    add[i]=0.0;
    penc=0;
    bunny[0]=bunny[1]=bunny[2]=0.0;
    gains.values[0].d=gains.values[1].d=gains.values[2].d=0.0;
    for(i=0;i<3;i++)
        projcheat[i]=pt[i];
    }

for(i=0; i<count;i++)
    {
    idcsold[i].id=idcs[i].id;
    idcsold[i].sc=idcs[i].sc;
    }

countold=count;
static float sicflag=1.5;
float wall=5.0;
float wallscale=1.0;
penc2=0;
for(i=0; i<3; i++)
    {
    if (position.values[i]>wall)
        {
        edge[i]=wallscale*(wall-position.values[i]);
        penc2=1;
        }
    else if(position.values[i]<-wall)
        {
        penc2=1;
        edge[i]=wallscale*(-wall-position.values[i]);
        }
    }
```

```
for(i=0; i<6;i++)
    {
    gains.values[i].p=0.0;
    gains.values[i].d=0.0;
    }

static float bunnyf[3];
static float scalebf[3];
for(i=0; i<3;i++)
    scalebf[i]=200;

if(probimp==true)
    {
    srcpoint.setValue(bunny[0],bunny[1], bunny[2],1);
    frotat.multVecMatrix(srcpoint, dstpoint);
    dstpoint.getValue(bunnyf[0],bunnyf[1], bunnyf[2], w);
    }

else
    bunnyf[0]=bunnyf[1]=bunnyf[2]=0.0;

static float tsh2=5.0;
for (i=0; i<3; i++)
    {
    force[i]=edge[i]+scalebf[i]*bunnyf[i];
    if (force[i]>tsh2)
        force[i]=tsh2;
    else if(force[i]<-tsh2)
        force[i]=-tsh2;
    despos.values[i]=(position.values[i]+2*force[i])/1000;
    }

// Set the rotational gains
gains.values[3].p=gains.values[4].p=50.0;
gains.values[5].p=10.0;
gains.values[5].d=gains.values[3].d=gains.values[4].d=0.40;
gains.values[2].ff=4.8;

// Set translational gains
for(i=0;i<2; i++)
    {
    if(force[i]!=0.0)
        {
        gains.values[i].p=5000;
        gains.values[i].d=30;
        gains.values[4-i].d=0.6;
        }
    }
if(force[2]!=0.0)
    {
```

```
    gains.values[2].p=10000;
    gains.values[2].d=30;
    }

idxmin=idxminold;
idxminvor=idcs[idxmin].id;
}
despos.values[3]+=balancefy;
despos.values[4]-=balancefx;
}


/**************   mergesort2 **************/
void mergesort(int low, int high) {

int i = 0;
int length = high - low + 1;
int pivot  = 0;
int merge1 = 0;
int merge2 = 0;
int merge21 = 0;
int merge22 = 0;
float working[length];
int tmptri[length];

if(low == high)
    return;

pivot  = (low + high) / 2;
mergesort(low, pivot);
mergesort(pivot + 1, high);

for(i = 0; i < length; i++)
    {
    working[i] = numbers[low + i].val;
    tmptri[i]=numbers[low + i].idx;
    }

merge1 = 0;
merge2 = pivot - low + 1;
merge21 = 0;
merge22 = pivot - low + 1;

for(i = 0; i < length; i++)
    {
    if(merge2 <= high - low)
        if(merge1 <= pivot - low)
            if(working[merge1] > working[merge2])
                {
                numbers[i + low].val = working[merge2++];
                numbers[low + i].idx=tmptri[merge22++];
                }
            else
                {
                numbers[i + low].val = working[merge1++];
                numbers[low + i].idx=tmptri[merge21++];
                }
        else{
            numbers[i + low].val = working[merge2++];
            numbers[low + i].idx=tmptri[merge22++];
            }
    else
        {
        numbers[i + low].val = working[merge1++];
        numbers[low + i].idx=tmptri[merge21++];
        }
    }
}

/****************************** searchpart ******************************/
void searchpart()
{
bool set=false;
NumTri2=69451;
int weird=0;
int countz=0;
int setcons=0;
int count2=0;
int mem=0;
int fok=0;
int code;
int a2, b2, zg, i, g,j,a,b;

for(i=0; i<NumTri2; i++)
    {
    zg=i;
    if(tri[i].partc<=2)
        {
        countz=0;
        for(g=i; g<NumTri2;g++)
            {
            if(tri[g].partc<=2);
                {
                if (g!=i)
                    {
                    for(j=0; j<3;j++)
                        {
                        fok=0;
                        if (tri[g].vidx[j]==tri[i].vidx[0])
                            {
```

```
        fok=1;
        code=0;
        }
    if (tri[g].vidx[j]==tri[i].vidx[1])
        {
        fok=1;
        code=1;
        }
    if (fok==1)
        {
        if(j==0)
            {
            a=1;
            b=2;
            }
        if (j!=0)
            {
            a=3-j;
            b=3-j-a;
            }
        count2=0;
        a2=0;
        b2=2;
        if(code==0)
            {
            a2=1;
            b2=2;
            }
        if(tri[g].vidx[a]==tri[i].vidx[a2])
            {
            set=true;
            setcons=a2;
            countz++;
            mem=g;
            }
        else if (tri[g].vidx[b]==tri[i].vidx[a2])
            {
            set=true;
            setcons=a2;
            countz++;
            mem=g;
            }
        else if (tri[g].vidx[a]==tri[i].vidx[b2])
            {
            set=true;
            setcons=b2;
            countz++;
            mem=g;
            }
        else if(tri[g].vidx[b]==tri[i].vidx[b2])
```

```
                                {
                                set=true;
                                setcons=b2;
                                countz++;
                                mem=g;
                                }
                            if(set==true)
                                {
                                if((tri[i].partc<=2)  && (g!=tri[i].part[0])
                                    && (g!=tri[i].part[3]) && (g!=tri[i].part[6]))
                                        {
                                        tri[i].part[3*tri[i].partc]=g;
                                        tri[i].part[3*tri[i].partc+1]=tri[i].vidx[code];
                                        tri[i].part[3*tri[i].partc+2]=tri[i].vidx[setcons];
                                        tri[i].partc++;
                                        }
                                if((tri[g].partc<=2) && (i!=tri[g].part[0])
                                    && (i!=tri[g].part[3]) && (i!=tri[g].part[6]))
                                        {
                                        tri[g].part[tri[g].partc*3]=i;
                                        tri[g].part[3*tri[g].partc+1]=tri[i].vidx[code];
                                        tri[g].part[3*tri[g].partc+2]=tri[i].vidx[setcons];
                                        tri[g].partc++;
                                        }
                                if (tri[i].partc>2)
                                    g=NumTri2-1;

                                set=false;
                                }
                            }
                        }
                    }
                }
            }
        }
    }
}

 /************************** calcnorm ****************************/

void calcnorm()
{

static int i, g, j;
static int ptr[2];
static float corrv2[3];
static float corrv1[2][3];
static float corrv11[2][3];
static float scalcorr[2];
static float newp[2][3];
static float cut[2][3];
static float vec[3];
static float norm2[3];
static float scal1, scal2, lcorrv2, norml;
static int idxc=0;
static int sf, cng;
static int nofound=0;
static int NumbTri2=69451;
static int idxo, cwhile, cwi, nc;
static bool again;
static int pt1, pt2, sok1, sok2, naco, okco;
static float vec1[3], vec2[3], ang[4];
static int ctnz, cpb;
static int ok, innof;
again=true;
idxc=0;
sf=1;
for(i=0; i<3; i++)
    {
    vec1[i]=vert[tri[idxc].vidx[1]][i]-vert[tri[idxc].vidx[0]][i];
    vec2[i]=vert[tri[idxc].vidx[2]][i]-vert[tri[idxc].vidx[0]][i];
    }

norm2[0]=vec1[1]*vec2[2]-vec1[2]*vec2[1];
norm2[1]=-vec1[0]*vec2[2]+vec1[2]*vec2[0];
norm2[2]=vec1[0]*vec2[1]-vec1[1]*vec2[0];
norml=sqrt(norm2[0]*norm2[0]+norm2[1]*norm2[1]+norm2[2]*norm2[2]);

for(i=0;i<3;i++)
    tri[idxc].nl[i]=norm2[i]/norml;

tri[idxc].nok=1;
idxo=idxc;
idxc=tri[idxc].part[0];
pt1=tri[idxo].part[1];
pt2=tri[idxo].part[2];
nofound=0;

do
    {
    innof=0;
    if (nofound==1)
        {
        innof=1;
        for (i=0; i<NumbTri2; i++)
            {
            if (tri[i].nok==1)
                {
                for (j=0; j<3; j++)
                    {
```

```
                    idxo=i;
                    idxc=tri[i].part[3*j];
                    pt1=tri[i].part[3*j+1];
                    pt2=tri[i].part[3*j+2];
                    nofound=0;
                    j=3;
                    }
                }
                if(nofound==0)
                    i=NumbTri2;
            }
        }
        if(nofound==1)
            again=false;
    }

if(again==true)
    {
    ptr[0]=tri[idxc].vidx[0]+tri[idxc].vidx[1]+tri[idxc].vidx[2]-pt1-pt2;
    ptr[1]=tri[idxo].vidx[0]+tri[idxo].vidx[1]+tri[idxo].vidx[2]-pt1-pt2;
    sok1=sok2=0;
    for(i=0; i<3; i++)
        {
        if(tri[idxc].vidx[i]==ptr[0])
            sok1=1;
        if(tri[idxo].vidx[i]==ptr[1])
            sok2=1;
        }

    for(i=0;i<3;i++)
        corrv2[i]=vert[pt2][i]-vert[pt1][i];

    lcorrv2=sqrt(corrv2[0]*corrv2[0]+corrv2[1]*corrv2[1]+corrv2[2]*corrv2[2]);
    for(i=0;i<3;i++)
        corrv2[i]/=lcorrv2;

    for(i=0; i<3;i++)
        {
        corrv1[0][i]=vert[ptr[0]][i]-vert[pt1][i];
        corrv1[1][i]=vert[ptr[1]][i]-vert[pt1][i];
        }

    for(j=0;j<2;j++)
        {
        scalcorr[j]=corrv1[j][0]*corrv2[0]+corrv1[j][1]*corrv2[1]+corrv1[j][2]*corrv2[2];
        for(i=0; i<3;i++)
        corrv11[j][i]=scalcorr[j]*corrv2[i];
        }

    for(j=0; j<2;j++)
```

```
        {
        for(i=0; i<3;i++)
            cut[j][i]=corrv1[j][i]-corrv11[j][i];
        }

    for(i=0; i<3; i++)
        vec[i]=vert[ptr[0]][i]-vert[pt1][i];

    norm2[0]=vec[1]*corrv2[2]-vec[2]*corrv2[1];
    norm2[1]=-vec[0]*corrv2[2]+vec[2]*corrv2[0];
    norm2[2]=vec[0]*corrv2[1]-vec[1]*corrv2[0];
    norml=sqrt(norm2[0]*norm2[0]+norm2[1]*norm2[1]+norm2[2]*norm2[2]);

    for(i=0;i<3;i++)
        norm2[i]/=norml;

    scal1=cut[0][0]*tri[idxo].nl[0]+cut[0][1]*tri[idxo].nl[1]+cut[0][2]*tri[idxo].nl[2];
    scal2=cut[1][0]*norm2[0]+cut[1][1]*norm2[1]+cut[1][2]*norm2[2];
    if(scal2>=0.0)
        {
        if(scal1<0.0)
            {
            for(i=0;i<3;i++)
                norm2[i]=-norm2[i];
            }
        }
    else if(scal2<0.0)
        {
        if(scal1>=0.0)
            {
            for(i=0;i<3;i++)
                norm2[i]=-norm2[i];
            }
        }

    for(i=0;i<3;i++)
        tri[idxc].nl[i]=norm2[i];

    ang[0]=atan2(tri[idxc].nl[2], tri[idxc].nl[1]);
    ang[1]=atan2(tri[idxo].nl[2], tri[idxc].nl[1]);
    ang[2]=atan2(cut[0][2], cut[0][1]);
    ang[3]=atan2(cut[1][2], cut[1][1]);
    for(i=0; i<4; i++)
        {
        if (ang[i]<0.0)
            ang[i]=2*M_PI+ang[i];
        }

    if(ang[2]<ang[3])
        {
```

```
        if ((ang[1]<=ang[3]) && (ang[1]>=ang[2]))
            {
            if ((ang[0]<=ang[3]) && (ang[0]>=ang[2]))
                ok=1;
            else
                cpb++;
            }
        else if (((ang[1]<=ang[3]) && (ang[1]<=ang[2])) || ((ang[1]>=ang[3]) && (ang[1]>=ang[2])))
            {
            if ((ang[0]<=ang[3]) && (ang[0]<=ang[2]) || (ang[0]>=ang[3]) && (ang[0]>=ang[2]))
                ok=1;
            else
                cpb++;
            }
        }
    else
        {
        if ((ang[1]<=ang[2]) && (ang[1]>=ang[3]))
            {
            if ((ang[0]<=ang[2]) && (ang[0]>=ang[3]))
                ok=1;
            else
                cpb++;
            }
        else if (((ang[1]<=ang[3]) && (ang[1]<=ang[2])) || ((ang[1]>=ang[3]) && (ang[1]>=ang[2])))
            {
            if (((ang[0]<=ang[3]) && (ang[0]<=ang[2])) || ((ang[0]>=ang[3]) && (ang[0]>=ang[2])))
                ok=1;
            else
                cpb++;
            }
        }
    tri[idxc].nok=1;
    nofound=1;
    for(i=0; i<3; i++)
        {
        if((tri[tri[idxc].part[3*i]].nok==0) && (tri[idxc].part[3*i]+tri[idxc].part[3*i+1]+tri[idxc].part[3*i+2]!=0))
            {
            idxo=idxc;
            idxc=tri[idxc].part[3*i];
            pt1=tri[idxo].part[3*i+1];
            pt2=tri[idxo].part[3*i+2];
            nofound=0;
            i=3;
            }
        }
    }
}while(again);
}
```