HAPI MANUAL

For version 1.3

CONTENTS

1 Introduction **4**

1.1 What is HAPI? 4

2 HAPI **5**

2.1 Overview 5

2.1.1 Device handling 5

2.1.2 Geometry based haptics 5

2.1.3 Free space haptic effects 6

2.1.4 Thread handling 6

3 Device handling **7**

3.1 Calibration 7

3.2 Layering 7

3.3 The HAPIHapticsDevice abstract base class 8

3.3.1 Device handling functions 9

3.3.2 Calibration functions 9

3.3.3 Haptics rendering functions 10

3.4 Example 11

3.5 Adding support for new devices 12

4 Haptics rendering algorithms **14**

4.1 Proxy based rendering 14

4.2 Available renderers 15

4.2.1 GodObject renderer . . . 15
4.2.2 Ruspini renderer . . . 15
4.2.3 Chai3D . . . 15
4.2.4 OpenHaptics . . . 16
4.3 Surfaces . . . 16
4.3.1 User defined surfaces . . . 17
4.3.2 The ContactInfo object . . . 18
4.3.3 Example surface . . . 19

5 Collision geometries **21**
5.1 Available shapes . . . 21
5.2 Using OpenGL to specify shapes . . . 22
5.3 Custom made collision geometries . . . 22

6 Force effects **24**
6.1 Available force effects . . . 24
6.2 Interpolation . . . 24
6.3 Creating new force effects . . . 24

7 Thread handling **26**
7.1 Thread handling classes . . . 26
7.2 Setting up a simple thread . . . 28
7.3 Thread safety . . . 29
7.3.1 Using locks . . . 29
7.3.2 Using callbacks . . . 30
7.4 Threads and haptics devices . . . 33

CHAPTER 1

INTRODUCTION

1.1 What is HAPI?

HAPI is an open-source, cross-platform, haptics rendering engine written entirely in C++. It is device-independent and supports multiple currently available commercial haptics devices. You can write your application once and not have to modify any code to use another haptics device. Choose between different rendering algorithms, different force effects and several kinds of surfaces to create the feeling that you want or create your own custom made effects. HAPI has been designed to be highly modular and easily extended.

Good Luck!
- The SenseGraphics Development Team.

CHAPTER 2

HAPI

2.1 Overview

HAPI has been designed to be a fully modular haptic rendering engine, allowing users to easily add, substitute or modify any component of the haptics rendering process.

HAPI consists of the following parts:

- Device handling
- Geometry based haptics
  - Collision handling
  - Haptics rendering algorithm
  - Surface interaction algorithm
- Free space haptics
  - Force effects
- Thread handling

2.1.1 Device handling

The device handling layer provides a device independent interface for various haptics devices. It handles device initialization, cleanup, access to device state such as position and orientation and output such as force and torque. HAPI can easily be extended for new devices by implementing abstract functions for performing those tasks. See section 3.5.

2.1.2 Geometry based haptics

Haptics rendering algorithm

A haptics rendering algorithm is needed in order to calculate forces and torques from the interaction of the haptics device with objects in the scene. An interface is provided to give a user the opportunity to implement their own algorithm. There are four algorithms already implemented. They are:

- God object algorithm - point proxy based
- Ruspini algorithm - sphere proxy based
- Chai3D - use Chai3D rendering library
- OpenHaptics - use OpenHaptics rendering library

Some of them have restrictions on what features of HAPI can be used, e.g. some does not support user defined surfaces. All these algorithms provide 3-DOF feedback only(i.e. no torque). 6-DOF algorithms are not currently available, however they can be generated by user force effects and rendering algoritms. Check out the details of each algorithm in chapter 4 to find out what is supported.

Collision handling

HAPI contains classes for collision handling used in the haptics rendering algorithms. A user have to create instances of these classes and feed them to the haptics device that they are to be rendered at. It also contains classes for building binary bound trees such as axis-aligned and oriented bounding box trees from triangles, that can be used in order to do faster collision detection.

Surface handling

When touching the surface of a geometry, the haptics rendering algorithm has to know what forces to generate depending on the penetration of the surface. This is handled by the surface classes. A user can define an arbitrary function for the force and proxy movement (does not work for all haptics rendering algorithms though as mentioned earlier).

2.1.3 Free space haptic effects

Force effects

Often a user wants to generate forces that are not based on touching geometries, but instead is only depending on the state of the haptics device, such as the position and orientation. Some exampels of such effects are force fields, gravity, springs, viscosity, etc. In these cases a force effect can be used.

2.1.4 Thread handling

HAPI manages high-priority threads running at 1000 Hz for the haptics rendering and provides mechanisms to communicate between different threads. The thread handling functions can also be used to create new user defined threads.

CHAPTER 3

DEVICE HANDLING

HAPI is device independent and can be used with a wide variety of haptics devices. The haptics devices currently supported by HAPI are:

- PHANToM Devices from SensAble Technologies (implemented in the PhantomHapticsDevice class) [5]
- Delta and Omega devices from ForceDimension (implemented in the ForceDimensionHapticsDevice class). [3]
- Falcon from Novint (implemented in the FalconHapticsDevice class). [4]
- HapticMaster from Moog FCS Robotics(implemented in the HapticMasterDevice class).[6] [2]

Each of these devices has a special class implementing the interface to the device. These classes also provide device specific functions that can be used. Most of the time the user does not care what haptics device is used and can then use the AnyHapticsDevice class when specifying the haptics device instead. The AnyHapticsDevice class will try to find a haptics device to use by trying to initialize all the devices that are supported. The haptics devices used will be the first one that initializes successfully.

3.1 Calibration

By default the coordinates given by the haptics device classes are given in metres in the local coordinate space of the haptics device. Often this is not the coordinate system a user wants though, and therefore HAPI provides a means to change the coordinate system by adding a calibration matrix. The position given by the getPosition() function will then be pos = matrix * orig_pos. It is also possible to calibrate the orientation and the result will be in the same way, i.e. orn = orn_calib * orig_orn. In this case the calibration value will not be a matrix, but a Rotation object.

3.2 Layering

When adding shapes to be rendered at the haptics device, one can specify what "layer" in which it should be rendered. Each layer has its own haptics rendering algorithm, which makes it possible to render different shapes with different haptics rendering algorithms on one haptics device. The resulting force

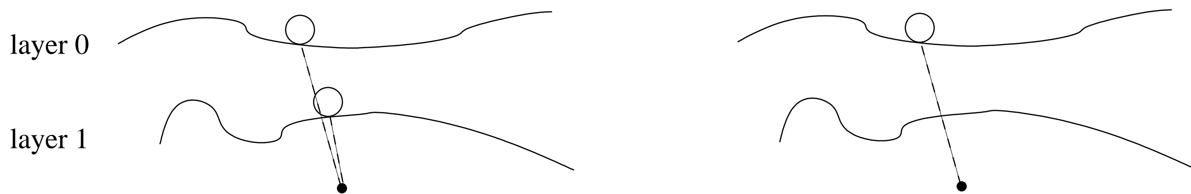

Figure 3.1: To the right all surface are part of layer 0. The proxy is then stopped at the first surface and the underlying surface will never be felt. To the left the two surfaces are put in two different layers which each has its own proxy, which means it will be felt.

will be the sum of the forces from each layer as illustrated in figure 3.1. A common use is, for example in medical simulation, to specify shapes for different tissue layers in different haptic layers. For example, if rendering a hand, one might use one layer for the skin and one layer for the bone of the hand. This will make it possible to feel the hard bone through the soft skin. If the shapes were specified in only one layer, the user would feel only the skin geometry, since the proxy stays on the surface. Specifying the bone in a second layer will add a new separate proxy for all the shapes in that layer. See the functions in section 3.3.3 for information on how to add shapes to a specific layer.

If no haptics rendering algorithm is specified for a layer, no forces will be generated from that layer.

By default all operations use layer 0.

3.3 The HAPIHapticsDevice abstract base class

The HAPIHapticsDevice abstract base class is the base class for all haptics device implementations. It contains the functions that are available for all haptics devices. The functions available can be divided into three categories:

- Device handling functions.
- Calibration functions.
- Haptic rendering functions.

The device handling functions are for communication with the haptics device, such as initializing/deinitializing the device, reading positions and sending forces.

The calibration functions lets the user manipulate the coordinate system the values of the haptics device are reported in.

The haptic rendering functions are used to define what to render on the haptics device, e.g. which effects, which shapes and what haptics renderer.

3.3.1 Device handling functions

Most common device handling functions:

- ErrorCode initDevice(int _thread_frequency = 1024) - Does all the initialization needed for the device before starting to use it.
- ErrorCode releaseDevice() - Releases all resources that were allocated in initDevice.
- ErrorCode enableDevice() - Enabling the device means that the positions and forces will be updated.
- ErrorCode disableDevice() - Stop the forces and position from being updated, but does not release the device.
- DeviceValues getDeviceValues() - Get a structure with all the device values that are recorded each frame (such as position, velocity,etc)
- Vec3 getPosition() - Get the position of the haptics device (in m)
- Vec3 getVelocity() - Get the velocity of the haptics device (in m/s)
- Rotation getOrientation() - Get the orientation of the haptics device stylus (if 6 DOF input).
- bool getButtonStatus(unsigned int button_nr) - get the status for a given button on the haptics device.

The ErrorCode returned for some of the above functions is an enumerator. The returned code will be HAPI::HAPIHapticsDevice::SUCCESS on success.

3.3.2 Calibration functions

Following are the most commonly used functions for modifying the local coordinate system the values of the haptics device are reported in.

- void setPositionCalibration(const Matrix4 &m) - Set the position calibration matrix
- const Matrix4 &getPositionCalibration() - Get the current position calibration matrix
- void setOrientationCalibration(const Rotation &r) - Set the orientation calibration
- const Rotation &getOrientationCalibration() - Get the current orientation calibration

3.3.3 Haptics rendering functions

Following are the most commonly used functions for changing what and how to render objects with the haptics device.

- void setHapticsRenderer(HAPIHapticsRenderer *r, unsigned int layer = 0) - Set the haptics rendering algoritm to use for a specific layer on the haptics device
- void addShape(HAPIHapticShape *shape, unsigned int layer = 0) - Render a shape on the given layer. The shape will be rendered until removed.
- void removeShape(HAPIHapticShape *shape, unsigned int layer = 0) - Remove a shape currently being rendered.
- void clearShapes(unsigned int layer = 0) - Remove all shapes on a layer.
- void addEffect(HAPIForceEffect *effect, HAPITime fade_in_time = 0) - Add a haptic effect to render by the haptics device. The effect will be rendered until removed.
- void removeEffect(HapticForceEffect *effect, HAPITime fade_out_time = 0) - Remove a haptic effect currently being rendered.
- void clearEffects() - Remove all haptic effects currently rendered.
- void transferObjects() - Transfer the current objects to the haptics loop.

The above functions works at a temporary local copy of effects and shapes. The haptics is rendered in a different loop and in order to transfer the changes made to that loop the user has to call the transferObjects() function.

Example:

```
// make changes to the objects you want to render
hd->clearEffects();
hd->addEffect( new HapticForceField );
hd->addEffect( new HapticSpring );

// nothing has changed on what is rendered yet, call transferObjects
// to transfer the changes to the haptics rendering thread.
hd->transferObjects();
```

See the doxygen documentation for a full listing of the functions available.

3.4 Example

Following is an example on a simple program that initialize a haptics device and sends a constant force of 1 N in the positive x-direction.

```
#include <HAPI/AnyHapticsDevice.h>
#include <HAPI/HapticForceField.h>

using namespace HAPI;

int main(int argc, char* argv[]) {
  // The force to render
  Vec3 force_to_render = Vec3( 1, 0, 0 );

  // Create a new haptics device, using any device connected.
  auto_ptr< AnyHapticsDevice > device( new AnyHapticsDevice );

  // initialize the device
  if( device->initDevice() != HAPIHapticsDevice::SUCCESS ) {
    // initilization failed, print error message and quit
    cerr << device->getLastErrorMsg() << endl;
    return 1;
  }

  // enable the device(forces and positions will be updated)
  device->enableDevice();

  // add the force effect to render
  device->addEffect( new HapticForceField( force_to_render ) );

  // transfer the effect to the haptics loop.
  device->transferObjects();

  // wait for keyboard ENTER press, then finish program
  string temp_string;
  getline( cin, temp_string );

  // release the device.
  device->releaseDevice();

  return 0;
```

}

3.5 Adding support for new devices

Support for new devices can easily be added to HAPI by subclassing the HAPIHapticsDevice class and implement the four abstract functions below.

- bool initHapticsDevice(int _thread_frequency = 1024) - initialize the haptics device. Return true on success, otherwise false and set an error message.
- bool releaseHapticsDevice() - releases all resources held by the haptics device. Return true on success, otherwise false and set an error message.
- void updateDeviceValues(DeviceValues &dv, HAPITime dt) - fill in the DeviceValues structures with the current values from the haptics device(values are e.g. position, velocity, etc). dt is the time in seconds since the last update.
- void sendOutput(DeviceOutput &dv, HAPITime dt) - render the output given in the DeviceOutput structure (such as force and torque) on the haptics device. dt is the time in seconds since the last update.

Look at the PhantomHapticsDevice and ForceDimensionHapticsDevice classes for example implementations.

Also in order for your new device to be recognised by the AnyHapticsDevice class, you will have to register the haptics device to a database of available devices. This is done by adding the a static member in your new class:

In header file:

```
static HapticsDeviceRegistration device_registration;
```

In .cpp file:

```
HAPIHapticsDevice::HapticsDeviceRegistration
MyNewHapticsDevice::device_registration(
                            "MyDevice",
                            &(newInstance< MyNewHapticsDevice >),
                            my_list_of_dlls
                            );
```

The string given as first argument is the name of your new device. The second argument is the function to use for creating a new instance of the device. The third argument is a list of names of dll-s that need to exist on the system to be able to start the device. This argument is only required for haptics devices that should be supported in Microsoft Windows.

CHAPTER 4

HAPTICS RENDERING ALGORITHMS

In HAPI a haptics rendering algorithm (or haptics renderer) is a class responsible for determining the forces and torque to render depending on position and other data from the haptics device and all the shapes the user has added to the scene.

4.1 Proxy based rendering

All the algorithms supported by HAPI (both internally implemented and external libraries such as Open-Haptics and Chai3D) use some variant of proxy based haptics rendering. The proxy based rendering technique involves having a virtual representation of the haptics device called the proxy. The proxy follows the position of the haptics device, but is not allowed to penetrate any surface. This means that when a shape is touched the proxy stays on the surface of the object, even if the haptics device actually has penetrated the surface. Forces are then generated to bring the haptics device out of the surface, towards the proxy, and usally some kind of spring force between the proxy and haptics device is used (see figure 4.1). When the user moves the haptics device inside the object the proxy follows the movement, but on the surface. How the proxy follows determines how the structure of the surface will feel. For example, if the proxy is always moved to a local minimum on the distance to the surface, the surface will feel completely smooth, but if the proxy is dragging behind friction effects are felt. In HAPI the user can control both the forces generated and the movement of the proxy.

In the algorithms in HAPI the shape of the proxy is either a point or a sphere. The problem with point proxies is that it can fall through small gaps in the haptic shape and makes it very hard to be able to touch small, thin objects. The problems with spheres, as with all other proxy-shapes than point, is that with increasing complexity of the proxy the problem of constraining the proxy to a surface increases. As the complexity increases the problem usually require more calculations to solve and as such it can be hard to make an algorithm fast enough for haptics rendering.

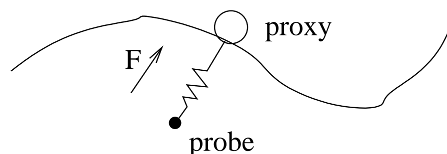

Figure 4.1: The proxy stays on the surface when the haptics device (probe) penetrates the surface. The resulting force is most commonly a spring force pushing the probe back towards the proxy.

4.2 Available renderers

There are four haptic renderers available in HAPI, where two are internally implemented in HAPI and two uses external haptics libraries. Each of them can work better than the others on a specific shape model or in a specific application. You will have to try and see which one works best in your case and with your requirements. In the listing below we point out some of the strength and weaknesses of each renderer in order for you to choose the algorithm.

4.2.1 GodObject renderer

The god-object algorithm is based on the paper "A constrained based god-object method for haptics display" by Zilles et al. [9]. It uses a point proxy and supports custom user defined surfaces.

Pros

- Open source.
- Device independent.
- User surfaces.

Cons

- Could be problems with the concave part of a curved surface.

4.2.2 Ruspini renderer

The RuspiniRenderer is based on the algorithm presented by Ruspini in "The Haptic Display of Complex Graphical Environments" [8]. It is different from all the other renderers in HAPI in that it uses a sphere proxy making it possible to have an interaction object with a size instead of just a point.

Pros

- Open source.
- Device independent.
- User surfaces.
- Sphere proxy.

Cons

- A little slower than other renderers in HAPI.

4.2.3 Chai3D

Chai3D [1] is an open source haptics library distributed under the GNU GPL license. It has been developed by a team at Stanford University in California, USA.

Pros

- Open source.
- Device independent.

Cons

- No user defined surfaces.
- Some fallthrough problems on moving objects.

4.2.4 OpenHaptics

OpenHaptics [7] is a proprietary haptics library developed by SensAble Technologies. It uses a point proxy based approach and provides a stable haptic feedback. It is however not very extendable in terms of user defined surfaces and only works with haptics from SensAble Technologies.

Pros

- MagneticSurface available.
- Good with moving objects.

Cons

- Only works with devices from SensAble.
- No user defined surfaces.
- Closed source.

4.3 Surfaces

A surface object in HAPI is an object that defines the haptic properties of a geometric shape, such as stiffness and friction. It is responsible for generating forces at a local contact point on a shape. The base class of all such objects is HAPISurfaceObject and there are several surfaces available in HAPI.

- FrictionSurface - Allows the user to set stiffness, damping, static friction and dynamic friction parameters. The force produced is a linear spring-damper force between the proxy position and probe position, i.e. F = stiffness * (proxy_pos - probe_pos) - damping * probe_velocity. The friction parameters control how the proxy move over the surface during contact. The static friction parameter controls how hard it is to start moving across the surface from a resting contact while the dynamic friction parameter controls how hard it is to move across the surface when the proxy has started moving.
- DepthMapSurface - Use a texture to define the depth of the surface (only available for GodObjectRenderer and RuspiniRenderer). A minimization algorithm controls how the proxy move over the surface. If the proxy is in an area on the surface that is considered deep it will be harder to move the proxy to an area on the surface which is considered higher. The depth calculated also modifies the force so that deep (and high) parts of the surface can be felt.
- HapticTexturesSurface - A surface similar to FrictionSurface with the exception that the four parameters stiffness, damping, static_friction and dynamic_friction are modified by textures given to the surface.

There is also a surface specific for the OpenHaptics library which allows the user to set the parameters available in that library directly. That surface is:

- OpenHapticsSurface - Has all the parameters available in OpenHaptics, i.e. stiffness, staticFriction, etc, but instead of specifying them in the HAPI way in absolute units, they are all specified as a value between 0 and 1. A value of 0 for the stiffness means no stiffness at all and a value of 1 means the maximum stiffness the device can handle. See doxygen documentation for all parameters of this surface.

4.3.1 User defined surfaces

In addition to using the already defined surfaces HAPI provides an interface to develop custom surfaces. This gives the user full control over the forces generated when in contact with a surface. The base class for all surface classes is HAPISurfaceObject. The surface object is responsible for two things.

Moving the proxy

The movement of the proxy is often used to control the feeling of an object when moving across the surface, e.g. if the surface should feel smooth or rough. Letting the proxy move along the contact plane as to minimise the distance between proxy and probe will create a totally smooth surface since the forces will always just be normal forces and no tangential forces (see figure 4.2). Letting the proxy lag behind in different ways can be used to create different friction-like effects and not letting it move at all forces the proxy to be stuck at the first point of contact of the surface (unless the probe is lifted out of the surface). The proxy movement is a 2D movement along the xz-plane of the local contact coordinate system.

When defining a new surface the virtual function getProxyMovement(ContactInfo &ci) in HAPISurfaceObject is used to define the proxy movement. The user should call the ci.setLocalProxyMovement(Vec2f &pm) function here to set the proxy movement. The rendering system will try to move the proxy according to the specified movement but might be stopped by colliding with other shapes along the path to the new position. The proxy will then stay at this collision point instead of moving all the way to the specified position.

Calculating an interaction force

After the new position of the proxy has been calculated, the interaction force with the surface has to be determined. For the most common surface types this is usually a linear spring force pulling the device back towards the proxy. It is however totally up to the user to define the force profile of the interaction to model e.g. button clicks and other non-linear behaviour. To have the most stable force feedback it is recommended that the direction of the force is towards the proxy. The interaction force(and possible

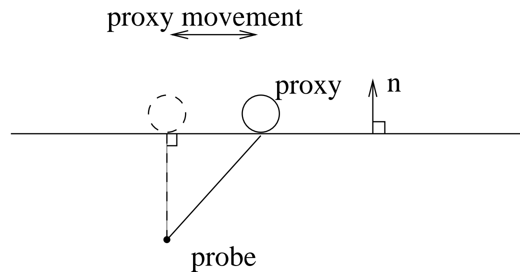

Figure 4.2: Local surface contact. Always moving the proxy to the dashed position(i.e. minimise the distance between proxy and probe) will give a surface without any friction.

torque) in a new surface is determined by defining the virtual function getForces(ContactInfo &ci) by calling the setGlobalForce or setLocalForce functions in the ci object.

4.3.2 The ContactInfo object

When calculating the proxy movement and the interaction force the user has access to a ContactInfo object. This object contains information about the surface contact that can be used in the calculations. It is also the object in which to return the calculated forces and proxy movement. The most commonly used values provided are:

- Contact point origin.
- Probe position.
- Probe velocity.
- Texture coordinate of contact.
- The haptics device that made the contact.
- The haptic shape the contact is made on.

All positions can be obtained in either the global coordinate system or in a local contact coordinate system. The local coordinate system is constructed with the proxy position as the origin, the contact normal as the y-axis and and two arbitrary perpendicular axis in the plane as x and z-axis (see figure 4.3). This means that e.g. the penetration depth is the same as -y in the local coordinate system. There are also functions in the ContactInfo object for converting between these two coordinate spaces.

The output parameters that can be set are:

- Force - setGlobalForce/setLocalForce

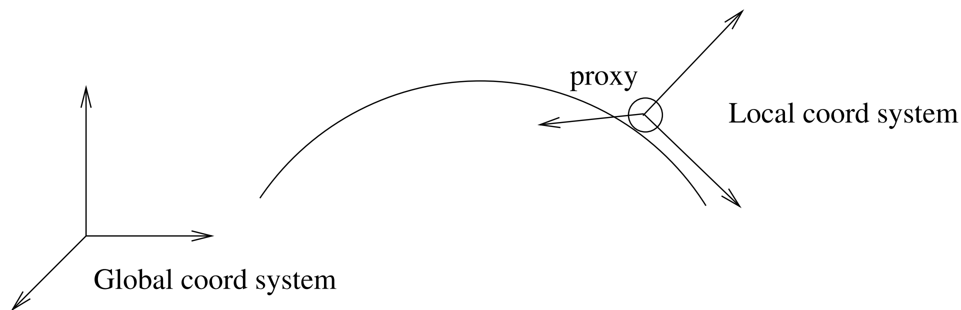

Figure 4.3: Local and global coordinate system. The local coordinate system is constructed with the proxy position as the origin, the contact normal as the y-axis and and two arbitrary perpendicular axis in the plane as x and z-axis.

- Torque - setGlobalForce/setLocalForce
- Proxy movement - setLocalProxyMovement

See the Doxygen documentation for a full listing of the functions available.

4.3.3 Example surface

Following is an example of an implementation of a surface with no friction and a linear spring force for the penetration of the surface.

```
// include the surface base class
#include <HAPI/HAPISurfaceObject.h>

namespace HAPI {

  // This class implements a surface without any friction and with
  // a linear spring interaction force pulling the probe back
  // towards the proxy. The force is defined as
  // F = stiffness * (proxy - probe)
  class HAPI_API SmoothSurface : public HAPISurfaceObject {
  public:

    // Constructor. Takes the spring constant.
    SmoothSurface( HAPIFloat _stiffness = 350 ):
      stiffness( _stiffness ) {}
```

```
    // Moves the proxy along the surface in the direction of the
    // probe.
    virtual void getProxyMovement( ContactInfo &contact_info ) {
      // get the position in the local contact coordinate system.
      Vec3 local_probe = contact_info.localProbePosition();

      // set the proxy movement to move to the projection of the probe
      // onto the local xz-plane.
      contact_info.setLocalProxyMovement( Vec2( local_probe.x ,
                                                local_probe.z ) );
    }

    // Calculate a force from the probe towards the proxy.
    virtual void getForces( ContactInfo &contact_info ) {
      // origin is proxy position after proxy movement. probe_to_origin
      // is the vector from the probe to the proxy.
      Vec3 probe_to_origin =
        contact_info.globalOrigin() - contact_info.globalProbePosition();

      // set the spring force
      contact_info.setGlobalForce(  stiffness*probe_to_origin );
    }

    // The stiffness of the surface.
    HAPIFloat stiffness;

  };
}
```

CHAPTER 5

COLLISION GEOMETRIES

In order for HAPI to do geometry based haptics rendering there must be a way to specify a geometry to do this haptic rendering on. In HAPI these collision geometries are called shapes.

5.1 Available shapes

The following shapes are general and can be used with any renderer although not all renderers support collision with the shape. RuspiniRenderer is for example the only implemented renderer for which lines and points can be felt. When referring to a "primitive" in the list below it means that the class inherit from GeometryPrimitive found in CollisionObjects.h.

- HapticLineSet - Contains a set of LineSegment primitives.
- HapticPointSet - Contains a set of Point primitives.
- HapticPrimitive - Contains one primitive.
- HapticPrimitiveSet - Contains a set of primitives.
- HapticPrimitiveTree - Contains a tree of primitives. The tree used allows for faster collision detection through grouping of the primitives into bounding boxes. Check the doxygen documentation (CollisionObject.h) for more information about the predefined tree classes in HAPI.
- HapticTriangleSet - Contains a set of Triangle primitives.
- HapticTriangleTree - Contains a tree of Triangle primitives. The tree used allows for faster collision detection through grouping of the triangles into bounding boxes. Check the doxygen documentation (CollisionObject.h) for more information about the predefined tree classes in HAPI.

The following two shapes are specific for OpenHapticsRenderer and will not work with any other renderer.

- HLDepthBufferShape - Use an instance of HAPIGLShape for easy implementation of a HL_SHAPE_DEPTH_BUFFER in HLAPI by using the render() function of HAPIGLShape.
- HLFeedbackShape - Use an instance of a HAPIGLShape for easy implementation of a HL_SHAPE_FEEDBACK_BUFFER in HLAPI.

5.2 Using OpenGL to specify shapes

If you are using OpenGL to do graphics rendering of your objects you can use the FeedbackBufferCollector class to get hold of the triangles that are used in your OpenGL graphics rendering and use them for haptics too.

```
// container to put the triangles in.
vector< HAPI::Collision::Triangle > triangles;

// start collecting OpenGL primitives
HAPI::FeedbackBufferCollector::startCollecting();

// draw your objects
draw();

// stop collecting and transfer the rendered triangles to the triangles vector.
HAPI::FeedbackBufferCollector::endCollecting( triangles );
```

The collected triangles could be used with a HapticTriangleSet or structured in a tree and used with HapticTriangleTree. The created shape can be sent to the haptics device to do geometry based haptic rendering.

5.3 Custom made collision geometries

All geometries sent to be rendered on a haptics device must inherit from the class HAPIHapticShape. These are the abstract functions that must be implemented when subclassing HAPIHapticShape.

- void closestPointOnShape(const Vec3 &p, Vec3 &cp, Vec3 &n, Vec3 &tc) - Get the closest point and normal on the object to the given point p.
- bool lineIntersectShape(const Vec3 &from, const Vec3 &to, Collision::IntersectionInfo &result, Collision::FaceType face) - Detect collision between a line segment and the object.
- bool movingSphereIntersectShape(HAPIFloat radius, const Vec3 &from, const Vec3 &to) - Detect collision between a moving sphere and the object.
- void getConstraintsOfShape(const Vec3 &point, Constraints &constraints, Collision::FaceType face, HAPIFloat radius) - Get constraint planes of the shape. A proxy of a haptics renderer will always stay above any constraints added.
- void getTangentSpaceMatrixShape(const Vec3 &point, Matrix4 &result_mtx) - Calculates a matrix which transforms from local space of the shape to texture space of the shape.

- void glRenderShape() - Render a graphical representation of the shape using OpenGL. This function only needs to be implemented if HAPI is built with OpenGL support.

The first three functions are used for collision detection. The second and fourth are used by the custom implemented renderers in HAPI. The fifth function must calculate something useful if DepthMapSurface should work with the shape. The last function have its uses for debugging purposes and also for one specific renderer (OpenHapticsRenderer). For more information about the arguments to the functions see the doxygen documentation.

CHAPTER 6

FORCE EFFECTS

There are two ways to generate forces in HAPI. One is the one previously described, by specifying shapes and surfaces and letting a haptics rendering algorithm determine forces when colliding with the shapes. Another way to generate forces is by using force effects. A force effect is a global force function which calculates what force to generate at any given moment. It can, among other things, be used to simulate viscosity and springs.

6.1 Available force effects

The force effects available in HAPI are:

- HapticForceField
- HapticPositionFunctionEffect
- HapticShapeConstraint
- HapticSpring
- HapticTimeFunctionEffect
- HapticViscosity

6.2 Interpolation

Force effects does not have an internal property to decide whether they should be interpolated between frames. If this is needed it is assumed that this is taken care of by the force calculation itself. However, when adding or removing a force effect from a haptics device there is a parameter to make the force effect fade in or out smoothly. See section 3.3.3 for details.

6.3 Creating new force effects

A user can easily create their own force effects by subclassing from HAPIForceEffect and implementing the calculateForces() method. The method takes the struct EffectInput as input. This struct contains:

- hd - A pointer to the HAPIHapticsDevice on which the force is rendered. Through this pointer the force effect can access device values such as the current position, velocity and orientation of the HAPIHapticsDevice.
- deltaT - The time difference from the last haptic frame to the current haptic frame.

The output of the calculateForces() method is an EffectOutput structure. It contains:

- force - The force calculated by the calculateForces() function.
- torque - The torque calculated by the calculateForces() function.

The force and torque outputed should always be given in global coordinates. Here follows an example of a very simple class that generates a constant force.

```
#include <HAPI/HAPIForceEffect.h>

namespace HAPI {
  /// This is a HAPIForceEffect that generates a constant force.
  class HAPI_API HapticForceField: public HAPIForceEffect {
  public:

    // Constructor.
    HapticForceField( const Vec3 &_force ):
      force( _force ) {}

    /// The force of the EffectOutput will be the force of the force field.
    EffectOutput virtual calculateForces( const EffectInput &input ) {
      return EffectOutput( force );
    }

  protected:
    Vec3 force;
  };
}
```

CHAPTER 7

THREAD HANDLING

Sometimes it is desirable to do calculations in a thread separate from an applications main thread. Thread handling in HAPI is needed because haptics rendering should be done at (at least) 1000 Hz which is considerably higher than main loop threads of most applications. A typical application with haptics added through HAPI would have thread communication like in figure 7.1. The purpose of the haptics thread is to calculate forces from all force effects and geometries that are rendered on a haptics device (figure 7.2).

7.1 Thread handling classes

All classes and everything needed to do proper thread handling in HAPI can be found in the H3DUtil library. The most commonly used classes for thread handling are:

- MutexLock - Used to lock/unlock access to data.
- ConditionLock - A MutexLock with extra features.
- SimpleThread - The simplest thread possible. Runs one function in a separate thread.
- PeriodicThread - Thread has a main loop and it is possible to add and remove callbacks.

See doxygen documentation for Threads.h and Threads.cpp (in H3DUtil) for more information and classes.

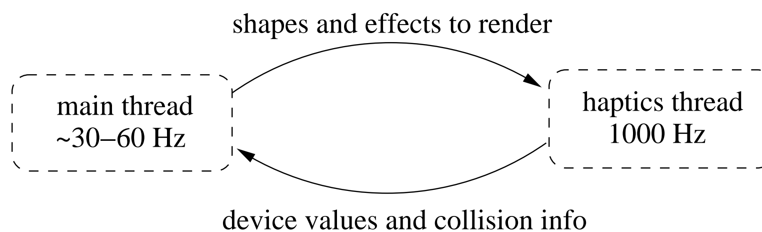

Figure 7.1: Thread communication

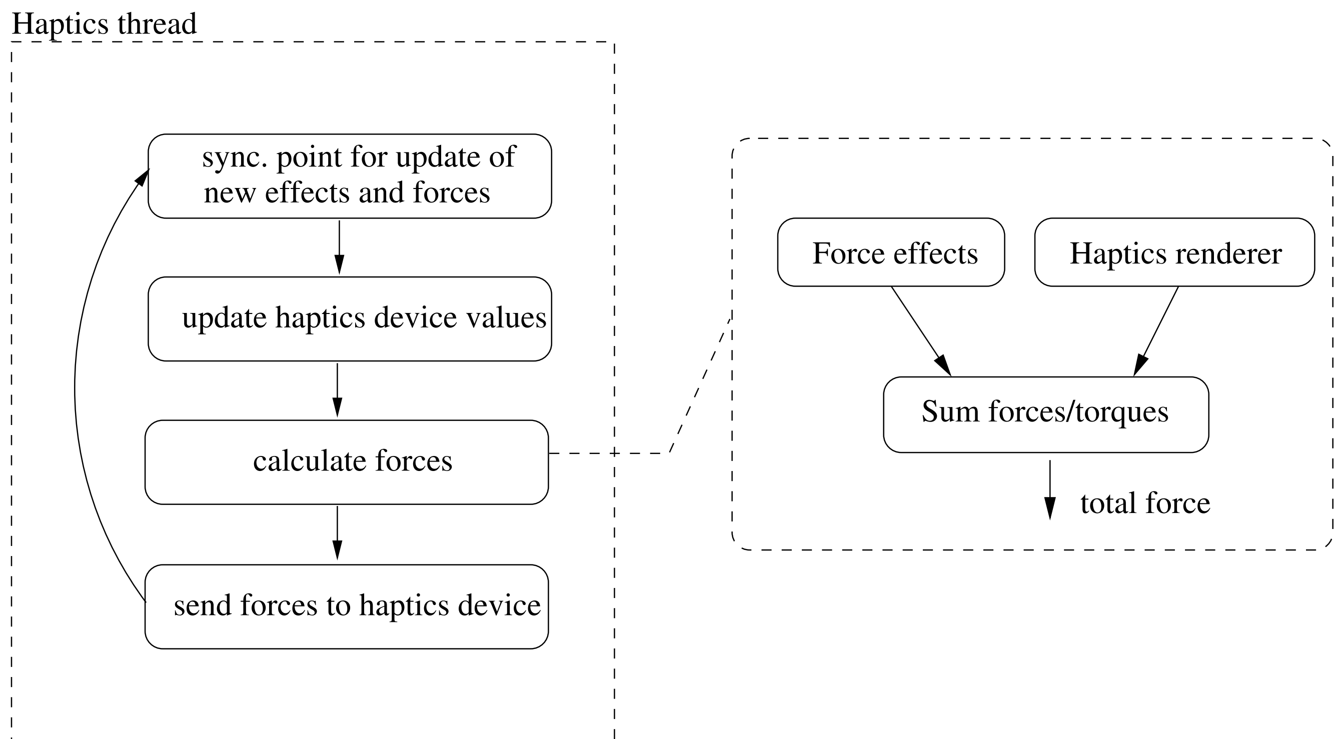

Figure 7.2: The flow of execution in the haptics thread.

7.2 Setting up a simple thread

Setting up a thread using the SimpleThread class is very easy. The arguments to SimpleThread are what function to run in a separate thread and arguments to that function if there are any. If the function depends on more than one argument these should be collected in a custom made struct and a pointer to an instance of the struct should be sent as argument.

A simple example of setting up a thread could be to print "SimpleThread" over and over again to the console in a separate thread while the main thread prints "Press ENTER to exit" once to the same console. The main thread will wait for input from the user and when ENTER is pressed the program will exit.

```
#include <H3DUtil/Threads.h>

// To be able to use std functions.
#include <iostream>
#include <string>
#include <vector>
using namespace std;

// The function to run in a separate thread.
void * printFunction(void *argument) {
  string * to_print = (string *)argument;
  while( true )
    cerr << *to_print << endl;
  return 0;
}

// Main function.
int main(int argc, char* argv[]) {
  // Creating the thread. The thread will start printing to the console.
  // The second argument contains what should be printed. It is done
  // in this way to show how to send arguments to the function.
  string to_print = "SimpleThread";
  H3DUtil::SimpleThread a_simple_thread( printFunction,
                                        (void *)(&to_print) );
  // while at the same time this is printed.
  cerr << "Press ENTER to exit" << endl;

  string temp_string;
  getline( cin, temp_string );
}
```

7.3 Thread safety

When adding new features which requires data to be accessed by more than one thread care must be taken so that only one thread at a time tries to access critical data. In HAPI this can be done in two ways. One approach is to use callbacks, the other one is to use MutexLocks to lock access for all other threads except the one that is currently modifying or reading data. These approaches are not mutually exclusive. It is possible to mix callbacks and locks.

7.3.1 Using locks

The concept of locks is easy to understand and use. The two most important functions which exists in both the MutexLock class and ConditionLock class are:

- lock() - Used to lock access to data.
- unlock() - Used to release the lock.

These two functions are used in pairs to protect code from being accessed by more than one thread at a time. If a variable is used in two or more threads there should be a lock/unlock pair around all code that use this variable. Similary, if a function will be called from two or more threads all code in the function should be enclosed by a lock/unlock pair. It is important to remember that calls to the lock() function only prevent access to data surrounded by lock/unlock pairs in those places where the same instance of MutexLock (or ConditionLock) is used to lock. This means that in an application which contains "MutexLock A" and "MutexLock B" calls to A.lock() will not prevent access to code after calls to B.lock().

In the example in section 7.2 one might want to change what is printed to the console by the SimpleThread without setting up the thread again. In the following example locks will be used in order to safely change the content of the string variable used for printing since it is not possible to set up callbacks for the class SimpleThread. Removing the locks from this example will most likely crash the program when pressing ENTER.

```
#include <H3DUtil/Threads.h>

// To be able to use std functions.
#include <iostream>
#include <string>
#include <vector>
using namespace std;

H3DUtil::MutexLock my_lock;
```

```
// The function to run in a separate thread.
void * printFunction(void *argument) {
  string * to_print = (string *)argument;
  while( true ) {
    // Added locks around the print.
    my_lock.lock();
    cerr << *to_print << endl;
    my_lock.unlock();
  }
  return 0;
}

// Main function.
int main(int argc, char* argv[]) {
  // Creating the thread. The thread will start printing to the console.
  // The second argument contains what should be printed. It is done
  // in this way to show how to send arguments to the function.
  string to_print = "SimpleThread";
  H3DUtil::SimpleThread a_simple_thread( printFunction,
                                         (void *)(&to_print) );
  // while at the same time this is printed.
  cerr << "Press ENTER to change what is printed" << endl;

  string temp_string;
  getline( cin, temp_string );

  // Thread safety locks.
  my_lock.lock();
  to_print = "Thread";
  my_lock.unlock();

  cerr << "Press ENTER to exit" << endl;
  getline( cin, temp_string );
}
```

7.3.2 Using callbacks

Using callbacks is another way to handle thread safety. All threads that inherit from the class PeriodicThreadBase must have the following functions implemented.

- virtual void synchronousCallback(CallbackFunc func, void *data) - Adds a callback function to

be executed in this thread. The calling thread will wait until the callback function has returned before continuing.

- virtual int asynchronousCallback(CallbackFunc func, void *data) - Adds a callback function to be executed in this thread. The calling thread will continue executing after adding the callback and will not wait for the callback function to execute. Returns a handle to the callback that can be used to remove the callback.
- virtual bool removeAsynchronousCallback(int callback_handle) - Attempts to remove a callback. returns true if succeded. returns false if the callback does not exist. This function should be handled with care. Callbacks are removed if they return CALLBACK_DONE or a call to this function is made.

The argument "func" in the above functions contains the callback function to execute. The callback function has to be of the form "CallbackCode myCallbackFunc(void *data)" where myCallbackFunc is the name of the function. A callback function used with synchronousCallback shall return the CallbackCode CALLBACK_DONE. A callback functions used with asynchronousCallback should return either CALLBACK_DONE or CALLBACK_CONTINUE. As long as the return code is CALLBACK_CONTINUE the callback will be called in the next thread loop.

Here follows an example on how to create a thread and set up different callbacks.

```
#include <H3DUtil/Threads.h>

// To be able to use std functions.
#include <iostream>
#include <string>
#include <vector>
using namespace std;

// The function we will use for our asynchronousCallback.
H3DUtil::PeriodicThreadBase::CallbackCode printAsynchronous(
                                                void *argument ) {
  string * to_print = (string *)argument;
  cerr << *to_print << endl;
  return H3DUtil::PeriodicThreadBase::CALLBACK_CONTINUE;
}

// The function we will use for our synchronousCallback. It will change
// the argument to the asynchronousCallback.
H3DUtil::PeriodicThreadBase::CallbackCode printSynchronous(
                                               void *argument ) {
  string * to_print = (string *)argument;
  cerr << "synchronousCallback" << endl;
```

```
  *to_print = "asynchCallback";
  return H3DUtil::PeriodicThreadBase::CALLBACK_DONE;
}

// Main function.
int main(int argc, char* argv[]) {
  // Creating the PeriodicThread.
  H3DUtil::PeriodicThread periodic_thread( DEFAULT_THREAD_PRIORITY, 3 );
  cerr << "Press ENTER to add an asynchronousCallback "
       << "that will be called each loop in the thread. "
       << "The number of loops per second is 3." << endl;

  string temp_string;
  getline( cin, temp_string );

  string to_print = "asynchronousCallback";
  int asynch_callback_handle =
    periodic_thread.asynchronousCallback( printAsynchronous,
                                          (void *)(&to_print) );

  cerr << "This could be printed before "
       << "\"asynchronousCallback\" prints. Press ENTER to add"
       << " a synchronousCallback." << endl;
  getline( cin, temp_string );

  periodic_thread.synchronousCallback( printSynchronous,
                                       (void *)(&to_print) );

  cerr << "This line will never be printed until after"
       << " synchronousCallback has been printed. "
       << "Press ENTER to remove the asynchronousCallback." << endl;
  getline( cin, temp_string );

  periodic_thread.removeAsynchronousCallback( asynch_callback_handle );

  cerr << "Press ENTER to exit" << endl;
  getline( cin, temp_string );
}
```

For thread safety reasons it does not matter if we add the function printSynchronous as a synchronous-Callback or asynchronousCallback. The program will not crash. The only difference is that the calling thread will wait for the callback to finish before continuing. However, if the main thread would change

the value of the variable "to_print" without the use of callback a crash may occur. It is more likely to happen for high update frequencies of the PeriodicThread.

7.4 Threads and haptics devices

Haptics rendering have to be done for each device which means that each device will have its own haptic thread. Most of the time it is enough to use the functions in section 3.3.3 but sometimes one might want to access the thread of the haptics device. This can be done through the function "H3DUtil::PeriodicThreadBase *getThread()" defined in the HAPIHapticsDevice class. The thread returned from this function can be used to set up new callbacks.

By default the thread used is a HapticThread. Most of the time when adding a new device to HAPI by subclassing HAPIHapticsDevice the default thread is sufficient. If however the haptics device provides some way of adding callbacks through calls to other functions a wrapper class should be implemented to override the virtual functions of PeriodicThreadBase for adding and removing callbacks. For example implementation of this see HLThread and PhantomHapticsDevice.

CHAPTER 7

BIBLIOGRAPHY

[1] Chai3D. http://www.chai3d.org. 15

[2] R. Van der Linde, P. Lammertse, E. Frederiksen, and B. Ruiter. The hapticmaster, a new highperformance haptic interface, 2002. 7

[3] ForceDimension. http://www.forcedimension.com. 7

[4] Novint Technologies Inc. http://www.novint.com. 7

[5] SensAble Technologies Inc. http://www.sensable.com. 7

[6] Moog/FCS. http://www.fcs-cs.com/robotics. 7

[7] OpenHaptics. http://www.sensable.com. 16

[8] Diego C. Ruspini, Krasimir Kolarov, and Oussama Khatib. The haptic display of complex graphical environments. In *Computer Graphics (SIGGRAPH 97 Conference Proceedings)*, pages 345–352. ACM SIGGRAPH, 1997. 15

[9] C. Zilles and J. Salisbury. A constraint based god-object method for haptic display. In *Proceedings of the IEE/RSJ International Conference on Intelligent Robots and Systems, Human Robot Interaction, and Cooperative Robots*, volume 3, pages 146–151, 1995. 15