A Simple Haptic User Interface Library

Juraj Czigányi*

Faculty of Informatics
Masaryk University
Brno / Czech Republic

Abstract

This paper briefly describes a development of a simple Haptic User Interface (HUI) library. The HUI library provides two classes of haptic widgets, buttons and sliders, which can be used as basic control elements for assembling more complex haptic interfaces.

Keywords: haptic user interface, haptic control element, stereo rendering

1 Introduction

Haptic research has opened new areas for designing training and entertainment applications. In nowadays applications, we can combine visualization of 3D objects with their haptic exploration and modelling. This can be achieved by merging visual and haptical spaces using stereoprojection techniques and mirrored views. One such solution is Reachin Display™ (See Figure1) [2] that uses CRT display and shutter glasses for stereo rendering and a PHANTOM™ [1] as a force-feedback device.

Figure 1: Reachin Display™

Programing for Reachin Display™ is supported by development platform Reachin API that enables a usage of various programming laguages (C++, Phyton, VRML) and OpenGL. In my bachelor thesis, I have designed a simple Haptic User Interface (HUI) library for creating haptic widgets. It is not intended to replace sophisticated haptic libraries used by professional developers. My goal was to prepare a template solution where students could test various designs of haptic widgets.

*cziganyi@mail.muni.cz

2 Haptic rendering

A mechanoreceptor is a sensory receptor that responds to mechanical pressure or distortion. In a simplified way, they are placed in skin layers and they are capable of "feeling" deformation of skin about 10 μm, see [9]. Haptic rendering, assigning forces to each operational point, stimulates these receptors creating thus an illusion of objects which can be "touched". The operational point, or points, is the physical location on the haptic interface where position, velocity, acceleration, and sometimes force, are measured. With displaying a virtual environment, there are some problems that must be considered[8]:

Finding the point of contact – This is a problem of collision detection, which becomes more difficult and computationally expensive as the model of the virtual environment becomes more complex.

Generation of contact forces – This creates the "feel" of the object. Contact forces can represent the stiffness of the object, damping, friction, surface texture, etc.

Dynamics of the virtual environment – When the user manipulates objects in the virtual environment, they collide with each other and may move in a complicated way.

Computational rates must be high and latency low – Inappropriate values of both these variables can cause hard surfaces in the virtual environment to feel soft as well as create system instabilities. The update rate of the haptic feedback loop should be at least 1 kHz ([4]).

There are many applications based on this technology[3]:

Medicine – surgical simulators for medical training; manipulating micro and macro robots for minimally invasive surgery; remote diagnosis for telemedicine; aids for the disabled such as haptic interfaces for the blind

Entertainment – video games and simulators that enable the user to feel and manipulate virtual solids, fluids, tools and avatars

Education – giving students the feel of phenomena at nano, macro, or astronomical scales; "what if" scenarios for non-terrestrial physics; experiencing complex data sets

Industry – integration of haptics into CAD systems such that a designer can freely manipulate the mechanical components of an assembly in an immersive environment

Graphic Arts – virtual art exhibits, concert rooms, and museums in which the user can login remotely to play the musical instruments, and to touch and feel the haptic attributes of the displays; individual or cooperative virtual sculpturing across the internet

3 General Principles

While a user moves a generic probe, the haptic device "senses" it's new position and orientation. In case of a collision between the probe and a virtual object, the device calculates the reaction force from the penetration depth of the probe into the virtual object. The force vectors may then be modified depending on the objects surface details and the resulting force is excerted as a feed-back to the user.

Similarly to computer graphics, the representation of 3D objects to be explored haptically, can be surface-based or volume-based. In both cases, modelled objects can be eplored using two types of haptic interactions:

point-based – Only the haptic interface point (HIP), also known as the end effector point, interacts with the object. The collision detection algorithm checks whether the HIP is inside or outside the object. If it is inside the depth of penetration is calculated from the distance of the HIP and the nearest surface point.

ray-based – The generic probe is modeled as a finite ray with orientation and the interaction is calculated from the interaction between the ray and the virtual object.

The reaction force is usually calculated from the linear spring law,

$$F = k * x,$$

where k is the stiffness of the object (high value causes the feel of rigid objects) and x is the depth of penetration. For frictionless surface, the vector of the reaction force is a normal to the polygonal face the generic probe collides with.

The two main issues that any haptic interaction paradigm should specify, are the *collision detection* between the generic probe and the virtual object and the *collision response*, i.e. how to calculate the reaction force after a collision has been detected.

4 Collision detection

Collision detection is similar, but not the same in computer graphics and haptic rendering. While in computer graphics the main goal is to detect collision between the objects, in haptic rendering the main goal is to detect the collision between the generic probe and the object. If the generic probe is inside of a virtual object, it is still counted as a collision, and the depth of penetration is used to calculate the proper response.

The collision detection algorithm cyclically detects the position of the generic probe and checks if it is in collision with any point of the virtual objects. The naive algorithm compares each object with the position of the generic probe.

Algorithm 1 Naive collision detection algorithm for haptics

```
1: loop
2:     get the position of the generic probe
3:     for all virtual objects do
4:         if the generic probe is in collision with the virtual
           object then
5:             calculate the proper response from the depth
               of the penetration
6:         end if
7:     end for
8: end loop
```

If the virtual objects are only basic 3D shapes (box, cone, cylinder, sphere) the naive algorithm is "acceptable". However, for more complex objects the collision detection would be very time-consuming and the collision response would not be realistic. To speed up the collision detection, usually bounding volumes are used.

The algorithm checks the collision between the generic probe and the virtual object only when the generic probe is inside of the bounding volume containing the virtual object. Due to this only a small percentage of objects are tested for collision.

Algorithm 2 Collision detection with bounding volume

```
1: loop
2:     get the position of the generic probe
3:     for all virtual objects do
4:         if the generic probe is inside the bouding volume
           of the object then
5:             if the generic probe is in collision with the vir-
               tual object then
6:                 calculate the proper response from the depth
                   of the penetration
7:             end if
8:         end if
9:     end for
10: end loop
```

Another way to reduce the number of objects for test-

ing is to use a binary space partitioning. The main idea is to divide the space into smaller regions, and continue the testing only in the selected smaller region. After division the algorithm works only with the region containing the generic probe. If the region contains more than one virtual object the algorithm divides the space again, otherwise it checks the collision for the object in the region. A simplified example of space partitioning and it's binary tree is shown in Figure 2.

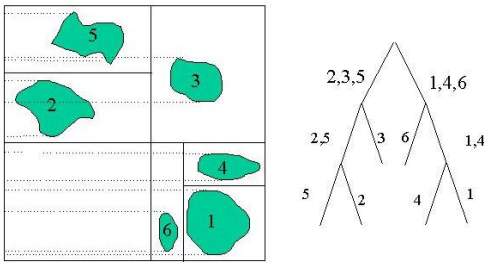

Figure 2: Binary space partitioning

5 Haptic User Interface Library

Control elements in the implementation are simple objects designated to create a simple HUI. They are not sufficient for creating professional HUIs but they can be used as a base for creating more complicated control elements. I have implemented them in an object oriented C++, theirs usage is specified in the HUI library documentation in [5]. The class diagram in Figure 3 shows the inheritance structure of HUI classes & objects.

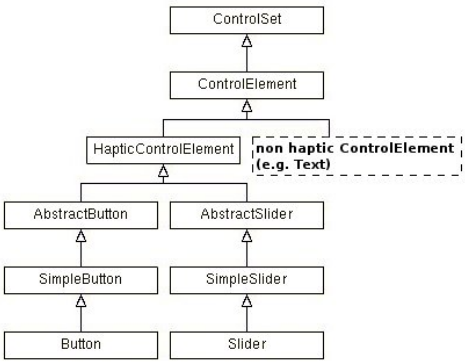

Figure 3: The class diagram of basic HUI classes

5.1 Control set

A *ControlSet* is a basic class, which is an "imaginary" object. Only this control element has absolute position, the other control elements have to be connected to a parent *ControlSet* class (or to a class derived from *ControlSet*) and their absolute position is calculated from their relative position and the absolute position of the parent object. Each of the control elements are inherited from the Control Set, so each of them can be a parent of other control elements.

5.2 Control element

A *ControlElement* class is derived from the *ControlSet* class. It serves as a base class for inheriting visible control elements. The boolean attribute *isVisible* designates the visibility of the control element. If *isVisible* is set to $false$ then the control element will not be rendered.

5.3 Haptic control element

A *HapticControlElement* class serves for inheriting control element classes with haptic feedback. It's boolean attribute *isActive* decides if a control element "can be touched" ($true$ value means that the object has a haptic response).

5.4 Control function

A *ControlFunction* virtual class serves as a base class for deriving objects, which can response to a control element event (these events will be explained at particular control elements). The class contains one virtual method $ControlFunction :: controlFunction(position, value)$ which is a response for an event. Where the *position* is the position of the generic probe at the moment, when the event arises and the *value* is float number from 0 to 1 (it's value will be explained at particular control elements).

5.5 Abstract Button

An *AbstractButton* class serves as a base class for inheriting button type control elements. This class contains two attributes *pressFunction* and *releaseFunction*. Both of them are instances of the *ControlFunction* class. Their meaning can be different from one button to the other.

5.6 Simple Button

A *SimpleButton* class represents a button, which is visualized as a thin rectangular plate. The class has attributes *height* and *width*, which are the height and width of the rectangle. Also the class has two color attributes: $color1$ is the color of the button, when it is not pressed, $color2$ is the color of the button while it is pressed. The button is pressed (the *pressFunction* is called with *value* attribute set to 1), when the user touches it and it get released (the *releaseFunction* is called with *value* attribute set to 1), when the user moves off the generic probe from the rectangle area.

5.7 Button

A *Button* class represents a button, which is a more sophisticated button than the Simple Button. This button is a solid object, it consists of two cuboid objects. One of them is the passive part of the button, which has a haptic response, but does not generates any events. The second one is an active part, which has a haptic response and can be ”pushed” into the passive part (it pushes out automatically to a basic position, if the user is not pushing it to the opposite direction). The user can push the active part into the passive part until it's front face get to the same plane as the front face of the passive part. The button have two states (pressed, released), which can be alternated by pushing the button. Pushing the button means to push the active part until it's front face reaches the plane of the passive button's front face. Before another state change the active part have to get back to it's basic position. When the button changes it's state an event is generated. Depending on the new state the *pressFunction* or the *releaseFunction* is called and the *value* attribute is set to 1. The button displays it's state by the colored a rectangle area placed in the middle of the passive parts front face. The class is derived from the *SimpleButton* class. The attribute *color*1 is the color of the passive part, the attribute *color*2 is the color of the active part. The attributes *height* and *width* sets the height and width of the passive part, height and width of the active part are 90 % of the attributes *height* and *width*. The height and width of the rectangle showing the buttons state is 95 % of the attributes *height* and *width*. The *Button* class also contains attributes:

*thickness*1 – Specifies the thickness of the passive part.

*thickness*2 – Specifies the thickness of the active part.

offColor – Specifies the color of the rectangle showing the buttons state, when the buttons state is equal to released.

onColor – Specifies the color of the rectangle showing the buttons state, when the buttons state is equal to pressed.

5.8 Abstract Slider

An *AbstractSlider* class serves as a base class for inheriting slider type control elements. This class contains two attributes *function* and *value*. The attribute *value* is the actual value set on the slider. The attribute *function* is an instance of *ControlFunction* class, which contains the function, which is called when, the actual value of the slider changes. The actual position of the generic probe and the actual value of the slider are handed over to this function.

5.9 Simple Slider

A *SimpleSlider* class instantiates a slider in the form of a thin rectangular panel. The area is split into a passive part and a movable marker, which shows the actual value of slider. Attributes *width* and *heigth* set the geometry of the passive part, the marker's width is initially set to 6 % of the attribute *width* and its height is 96 % of the attribute *heigth*. By touching the passive part the marker is immediately set to a new position and actual value is adjusted accordingly. The actual value depends on the relative distance from the ”lower” side of the passive part. The slider has three color attributes:

*color*1 – Specifies the color of the left side of the passive part.

*color*2 – Specifies the color of the right side of the passive part. The color of the passive part between the right and left side changes smoothly and it is calculated using linear interpolation between *color*1 and *color*2.

*color*3 – Specifies the color of the marker.

5.10 Slider

The *Slider* class represents a slider similar to the slider created by the *SimpleSlider* class, but this slider has thickness. There are two thickness attributes *thickness*1 and *thickness*2, which specify the thickness of the passive part, and the thickness of marker. The marker is positioned on the passive part. The actual value of this slider can be changed by pushing the marker from the left or right side to the wanted value.

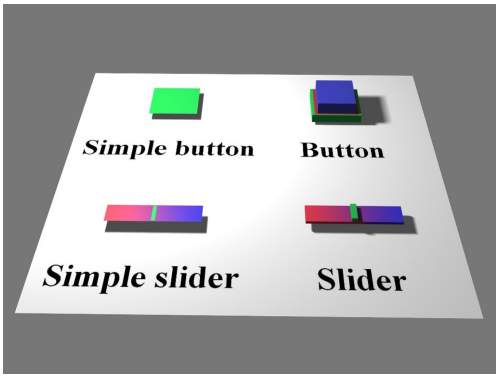

Figure 4: Button, slider, etc.

6 Stereo rendering with OpenGL

Stereoscopic rendering is any technique capable of reproducing three-dimensional visual information or creating the illusion of depth in an image. The illusion of depth in photographs, movies, or other two-dimensional images is created by presenting a slightly different image to each

eye. The stereopsis is a process, which allows humans to calculate the distance of the objects from two images. The eyes, being approximately 7 cm far from each other, observe the same objects from a different angle. By differentiating the angles the brain is capable of deciding the objects position and its distance from the observer. In stereo rendering we need to reverse the process – to create two different images from the knowledge of the objects position and distance.

For creating the depth illusion on the Reachin Display device with the help of the OpenGL quick alternating between two slightly different pictures are needed. Drawing of the two pictures requires a special function for setting up the projection matrix and another one for setting up the viewing matrix. I have implemented these two special functions from the procedural base [7] into object oriented form, they are represented in the implementation by the method *SetStereoPerspective*() and method *SetStereoLookAt*() of the class *Stereo*.

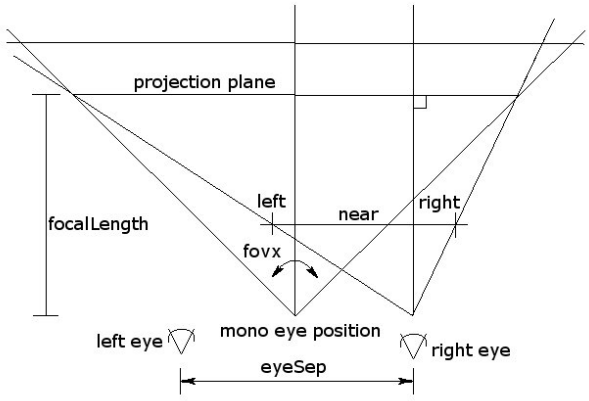

Figure 5: Rendering for stereo display from the top. (source: http://pds5.egloos.com/pds/200706/07/74/stereo_viewing_coding.ppt)

6.1 SetStereoPerspective()

SetStereoPerspective() is built on function *glFrustum*(), which is used to set the "mono" perspective projection matrix. *glFrustum*() has six attributes[6]:

$left$, *right* – Specifies the coordinates for the left and right vertical clipping planes.

bottom, top – Specifies the coordinates for the bottom and top horizontal clipping planes.

near, far – Specifies the distances to the near and far depth clipping planes.

To set pair of projection matrixes for stereo, method *SetStereoPerspective*() uses the following attributes, see Fig.5:

near, far – same as in the mono rendering.

$fovy$ – Specifies the field of view angle, in degrees, in the y-direction.

aspect – Specifies the aspect ratio that determines the field of view in the x-direction. The aspect ratio is the ratio of x (width) to y (height).

eyeSep – Specifies the distance between the two eyes of the spectator.

$focalLength$ – Specifies the distance between the eyes of the spectator and the projection plane. The spectator is focused on the projection plane, so this distance is the focal length.

eyeMode – Specifies the active eye (for which is the program rendering).

The attribute top is calculated as

$$top = near * \tan(DTOR * \frac{fovy}{2})$$

where *DTOR* is a constant used for conversion from degrees to radians and vice versa. The value *bottom* is simply calculated as $-top$. The attributes $left$ and *right* depend on the eye for which is the projection is used. For this difference a new variable is defined, the *stereoAdjustment*, which is calculated as

$$stereoAdjustment = \frac{eyeSep * near}{2 * focalLength}.$$

With the variable *halfNearWidth*, which is the half of the width of the field of the view at the near clipping plane

$$(halfNearWidth = aspect * top),$$

, the attributes $left$ and *right* for the right eye are calculated as:

$$\begin{aligned} left &= -halfNearWidth - stereoAdjustment \\ right &= halfNearWidth - stereoAdjustment \end{aligned}$$

6.2 SetStereoLookAt()

SetStereoLookAt() is built on the function *gluLookAt*(), which is used to set "mono" viewing matrix. *gluLookAt*() has nine attributes[6]:

$eyeX$, $eyeY$, $eyeZ$ – Specifies the position of the eye point (the point between the two eyes of the spectator).

$centerX$, $centerY$, $centerZ$ – Specifies the position of the reference point (the point the spectator is looking at).

upX, upY, upZ – Specifies the direction of the up vector.

Algorithm 3 GLvoid Stereo::SetStereoPerspective()

```
1:  GLdouble left;
2:  GLdouble right;
3:  GLdouble top = near*tan( DTOR * fovy/2 );
4:  GLdouble bottom = -top;
5:  GLdouble halfNearWidth = aspect * top;
6:  GLdouble stereoAdjustment = eyeSep/2 * near / focalLength;
7:  if RIGHT == eyeMode then
8:      left = -halfNearWidth - stereoAdjustment;
9:      right = halfNearWidth - stereoAdjustment;
10: else
11:     left = -halfNearWidth + stereoAdjustment;
12:     right = halfNearWidth + stereoAdjustment;
13: end if
14: glFrustum( left, right, bottom, top, near, far );
```

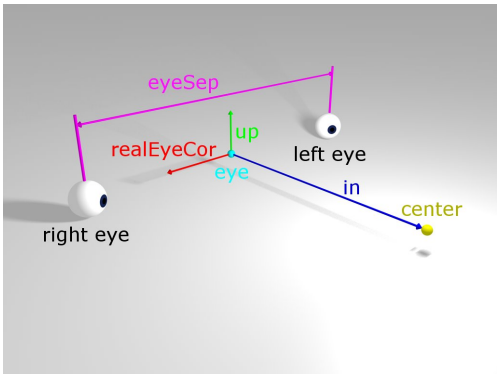

Figure 6: Calculation of the eyes position

The *SetStereoLookAt*() works with the same attributes and with the attributes *eyeMode* and *eyeSep*, which are the same as in the subsection SetStereoPerspective() on the page 6. For stereo rendering only the position of the eye has to be changed according to the *eyeMode*. The subtraction of the point *centre* and *eye* adds the vector *in*, which is pointing from the *eye* to the *centre*.

$$in = centre - eye$$

The cross product of the vector *in* and the point *up* is the vector *realEyeCor* pointing from the *eye* towards the spectators right eye (the vector $-realEyeCor$ is pointing towards the spectators left eye).

$$realEyeCor = in \times up$$

The position of the right (left) eye is calculated as the addition of the *eye* and the relative position of the right (left) from the *eye*. The relative position of the right (left) eye from the *eye* is the multiplication of the vector *realEyeCor* ($-realEyeCor$) with the proportion of the length of the vector *realEyeCor* and the half of the attribute *eyeSep*.

$$rightEye = eye + realEyeCor * \frac{eyeSep/2}{|realEyeCor|}$$

$$leftEye = eye - realEyeCor * \frac{eyeSep/2}{|realEyeCor|}$$

Algorithm 4 GLvoid Stereo::SetStereoLookAt()

```
1:  GLdouble inX = centreX - eyeX;
2:  GLdouble inY = centreY - eyeY;
3:  GLdouble inZ = centreZ - eyeZ;
4:  GLdouble factor = (eyeMode==RIGHT)?1:-1;
5:  GLdouble realEyeCorX = factor * (inY*upZ - upY*inZ);
6:  GLdouble realEyeCorY = factor * (inZ*upX - upZ*inX);
7:  GLdouble realEyeCorZ = factor * (inX*upY - upX*inY);
8:  GLdouble length = sqrt( realEyeCorX*realEyeCorX +
    realEyeCorY*realEyeCorY + realEyeCorZ*realEyeCorZ);
9:  GLdouble corLengthFactor = eyeSep/(2*length);
10: realEyeCorX *= corLengthFactor;
11: realEyeCorY *= corLengthFactor;
12: realEyeCorZ *= corLengthFactor;
13: gluLookAt( eyeX + realEyeCorX, eyeY + realEyeCorY,
    eyeZ + realEyeCorZ, centreX, centreY, centreZ, upX, upY, upZ);
```

7 Conclusion

The HUI library can be used for creating haptic interfaces with haptic widgets such as buttons and sliders for the Reachin Display device. The user can use the basic OpenGL (also GLU and GLUT) functions for drawing. Solid objects are rendered in stereo.

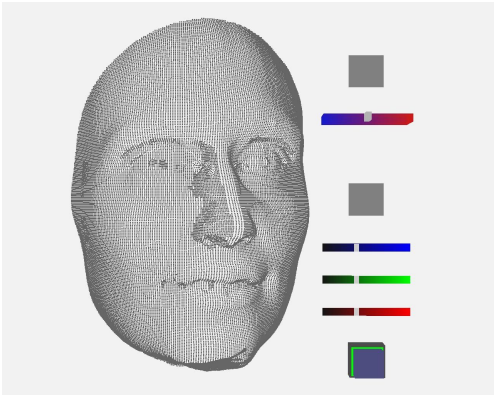

Figure 7: Simple application for exploration of point-based models

HUI library is simple and serves only as a base for creating more complex haptic widgets. In the future, I plan to extend it with the following functionality:

multiplatform and multilanguage – The library should provide interfaces for various platforms (Windows, Linux) and languages (C++, Java, Python)

compatibility update – The header file could support different methods of stereo rendering.

improved stereo–rendering – Stereo rendering with the consideration of parallax.[7]

additional control elements – A broader repertoir of control elements, and new physically based features (colors, textures, stiffnes, mechanics) for creating complex HUIs.

HUI toolkit – a toolkit for assebly of basic haptic controls similar to GUI toolkits.

This work is a part of my Bachelor's thesis, February 2008, Masaryk university, Brno, the Czech Republic.

References

[1] The PHANTOM Desktop device home page. `http://www.sensable.com/haptic-phantom-desktop`.

[2] The Reachin Display device home page. `http://www.reachin.se/products/`.

[3] C. Basdogan and C. Ho. Principles of haptic rendering for virtual environments. `http://network.ku.edu.tr/~cbasdogan/Tutorials/haptic_tutorial.html`, 2007.

[4] C. Basdogan, C. Ho, and M. Srinavasan. NASA tech briefs - algorithms for haptic rendering of 3D objects. `http://www.techbriefs.com/content/view/1063/32/`, 2007.

[5] J. Czigányi. GUI for point-based haptic rendering. Bachelor's thesis, Faculty of Informatics Masaryk University, 2008.

[6] Khronos Group. OpenGL - the industry standard for high performance graphics. `http://www.opengl.org/`, 2007.

[7] K. Hyun-Chul. Stereo viewing coding. `http://pds5.egloos.com/pds/200706/07/74/stereo_viewing_coding.ppt`, 2007.

[8] A. M. Okamura. Literature survey of haptic rendering. `http://www.haptics.me.jhu.edu/publications/old/hapticlit.html`, 2007.

[9] S. Trojan et al. *Medical Physiology (in Czech)*. Grada Publishing, 2003.