Mutual Calibration of a Co-Located Haptics Device and Stereoscopic Display

Daniel Evestedt
daniel@reachin.se

John McLaughlin
john@reachin.se

Reachin Technologies AB
www.reachin.se

Abstract

We describe a calibration technique for a hand-immersive virtual-reality environment – the Reachin Display [1]. The technique we present allows us to calibrate both the haptics device and the stereo display environment simultaneously where calibration of the stereo display utilises the haptics device and vice versa. The calibration procedure is described and we present a modular architecture for constructing calibration functions and show how fine control over the individual calibration parameters that are used can be obtained.

1. Introduction

The Reachin Display [1] comprises an articulated haptics device [2] and a stereoscopic visual display. The working volumes of these two components are arranged to coincide so as to effect a co-located hapto-visual display, i.e. the user can feel the object in the same place as he sees it. This is accomplished by using a semi-transparent mirror to reflect the scene being drawn giving the effect that the objects appear in space in the same place as the hands and the haptics device (see Figure 1). The greatest benefit of such a system is that it allows the user to utilise their sense of proprioception to quickly and accurately locate objects within the workspace. Also there is a much stronger sense of realism than the traditional set-up.

Co-located displays require accurate calibration in order for the user to perceive the combined haptic and visual displays as a single multi-sensory environment. Also, if the mirror in the display is semi-transparent then the user sees both the real objects in the workspace (such as their hands) as well as the virtual objects displayed by the monitor. In this *augmented reality* mode (which is used during calibration) any error in calibration will be obvious and disturbing. We would like to be able to acquire an accurate calibration in an easy way with as little work needed from the user as possible. For example, the task of setting the parameters manually requires a lot of work and any change to the set-up invalidates the calibration. Since the set-up can vary quite a lot we would like to be able to generate the calibration automatically.

Figure 1: User using the Reachin Display. The user looks through the mirror and thereby experiences the objects in the same space as his hands.

The calibration actually consists of two parts. One part is the translation of the arms' position of the haptics device into a position in the stereo display. This is not trivial since the haptics device in theory can be mounted in any position and orientation on the display. Hence a movement to the left in the haptics device coordinate space can be a movement in any direction in the stereo display environment. The other thing that needs to be

calibrated is the visual display of the scene. We need the objects to be the right size (the size specified by the user) and the perspective to be right so that the stereo environment does not differ from the real world. This means that a real object and a virtual object of the same size should appear to have same size anywhere in the stereo display work area.

We present a calibration technique where both these parts can be calibrated. The calibration of the stereo display environment utilises the haptics device and the calibration of the haptics device utilises the stereo display environment.

2. Calibration procedure

During the calibration process the user sees both the real stylus of the haptics device as well as a virtual model of the stylus moving within the stereoscopic display. When everything is calibrated correctly these two objects should coincide accurately in position, orientation and size. When they do not, the user may click the button on the stylus to freeze the virtual stylus, then move the real stylus so that it coincides accurately with the virtual stylus and then release the button. This records a *correction sample*. This sample consists, in our case, of the encoder angles of the arms of the haptics device and the screen coordinates for the left and right eye of the position of the virtual stylus. The user can specify several correction samples and these samples are then used by the software to calibrate some subset of more than 12 independent *calibration parameters*.

3. Calibration functions

A *calibration function* is a multi-valued vector function from a set of values (called *real* values) and a set of calibration parameters to another set of values (called *virtual* values). When the user clicks the button, they specify the virtual values of the correction sample and when they release the button they specify the corresponding real values.

In our case the calibration function consists of several stages. We begin with *relative encoder angles* read directly from the encoders on the arms of the haptics device. In our application there are 3 encoders measuring position and 3 for orientation of the stylus. These angles are *relative* to the angular positions of the encoders when the haptics device was last sent a *reset* command. Therefore we offset these values by the 6 *reset angle* calibration parameters to calculate the *absolute encoder angles*. Normally the device has a standard reset position in which the reset command is supposed to be called at. This defines the position where the reset angles are all zero and therefore the relative encoder angles also are the absolute encoder angles. However this reset position is not always reachable by the haptics device (e.g. the mirror in the display can be in the way) and even if it is it can be quite hard to put it in the reset position with good accuracy. So instead of doing this we include the reset angles in the calibration, which eliminates all such problems.

These encoder angles may then be transformed into a Cartesian position and stylus orientation according to known geometry information such as the lengths of the haptics device arms. This position is then further transformed into the *world coordinates* of the rendering software by means of a calibration translation, rotation and scaling. This is the coordinate space that a user of the Reachin Display uses to specify objects in.

Finally the world coordinates are projected onto the stereoscopic screen with one position for the left eye view and one for the right eye view. This transformation is controlled with OpenGL [4][7] parameters and can be divided into several parts. First the world coordinates are transformed by a *model view matrix* into *eye coordinates*. The eye coordinates are the coordinates of the objects looking into the world through a camera with a specified position and orientation. Since we use a stereoscopic display we must make two such transformations, one with the camera in the position of the left eye and one with the camera in the position of the right eye. After this transformation a *projection matrix* is applied and the coordinates are normalised. The projection matrix is set up by specifying an asymmetric view frustum for each eye. The view frustum is constructed from field-of-view angles, focal distance and inter-ocular distance (see Figure 2). This gives us the screen coordinates for the left and right eye (also called normalised device coordinates) and this is the end product of our calibration function since these define the perceived 3-dimensional position in the display.

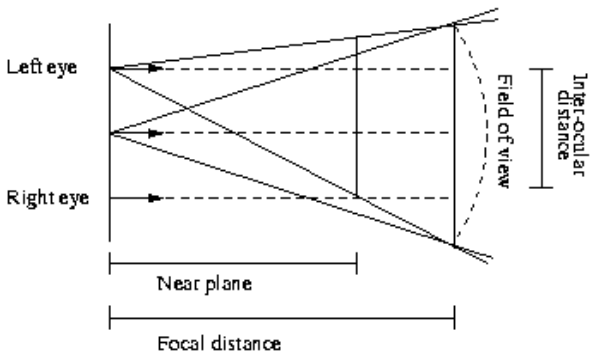

Figure 2: The asymmetric view frustum for the left eye. The centre view is the normal view when stereo is not used and the view from the left eye is the corresponding view used in stereo mode for the left eye. A similar frustum is used for the right eye.

We then define an *error function* based on the discrepancy between the measured real values transformed by the calibration function and the measured virtual

values. Specifically the discrepancy is defined to be the distance-squared between the virtual values and the transformed real values. So the error for a correction sample is

$$e(r,v,c) = \left(f(r,c) - v\right)^2$$

where r are the real values of the correction sample, v are the virtual values, c are the calibration parameters and f is the calibration function. If we have several correction samples the total error is the sum of the errors of the individual samples.

This gives us a single error value and the task of the calibration procedure is then to adjust nominated calibration parameters so as to minimise this error function. This is essentially a numerical minimisation function in multiple dimensions. We use a variable metric method to perform the minimisation. The method used is the Broyden-Fletcher-Goldfarb-Shanno variant of the Davidon-Fletcher-Powell minimisation algorithm [3]. This method requires us to provide the gradient of the function to minimise.

Let $J_c(r,c)$ be the matrix whose columns are the partial derivates of f with respect to the calibration parameters c. This is a sub-matrix of the full Jacobian of f. Using this function we may define the error gradient with respect to the calibration parameters c as

$$G(r,v,c) = 2\left[f(r,c) - v\right] \cdot J_c(r,c)$$

This is a sub-vector of the full gradient of e, containing only the partial derivatives of e with respect to c. In the case that the error is summed over several correction samples, the error gradient is simply the sum of the gradients for each sample. How to get hold of the Jacobian function is explained in the next section.

The problem of the minimisation is that it is easy to find a local minimum in the error function and hence we might not get the optimal solution. In order to prevent this we make several minimisations from different starting points and choose the smallest minimum we can find of all the minimisations. Of course this does not always find the global minimum either, but at least we have improved our chances to get a better minimum than if we had just used the first one we found.

5. Calibration modules

When implementing the calibration functions into a calibration program we would like to be able to make changes to them easily. We would, for example, like to be able to choose which calibration parameters to calibrate and which to leave unchanged. We would also like to be able to extend a calibration function with extra parameters without having to rewrite any code from the previous function. To accomplish this we use a number of *calibration modules* to construct the calibration function. A calibration function can consist of one or more of these calibration modules. If we want to do a change to the calibration function (e.g. add a rotation) we just construct a new calibration module defining this transformation and insert it into the calibration function.

A calibration module defines a function f from a set of values A to another set of values B. The values of A can be divided into *input parameters* and *calibration parameters*. The calibration parameters are the parameters that are to be adjusted to get the best calibration while input parameters are the positional/geometric values to be transformed by the module. For example, if the module specified a translation in Cartesian coordinates it would take six parameters. Three of them would be calibration parameters specifying the translation at each axis and three would be input parameters specifying the original coordinates which we want translated.

Each module also defines two Jacobian functions. One of the functions, J_i, returns the Jacobian matrix for f with respect to the input parameters and the other function, J_c, returns the Jacobian matrix of f with respect to the calibration parameters. These Jacobian functions can be quite complicated and in many cases they would be intractable to code by hand. Because of this we use the symbolic mathematics software Maple to automatically generate the code for the calibration modules. This software allows us to specify the function of the calibration module symbolically. Partial derivatives and Jacobian matrices can then easily be generated with the software and we use the automatic code generation tool to get optimised functions for evaluating both the Jacobian function and the normal function of the calibration module. This way the calibration modules become "black boxes" of code, which can be composed into larger calibration modules, which allows for an efficient and extensible run-time implementation.

When two of these modules have been defined we can compose them into one single calibration module performing the composed transformation of the two individual modules. The only restriction is that the number of output parameters of the first module matches the number of input parameters of the other module. When composing two calibration modules into a single one we must create composed versions of the function and the two Jacobian matrices. For the function value it is quite straightforward.

$$f''_c(r,c'') = f''_c(r,c \mid c') = f'\left(f(r,c),c'\right)$$

where f is the function of the first module and c are the calibration parameters for the first module. The primed versions are same but for the second module. r is the input parameters for f. Note that the calibration parameters for

this new function are the composition of the two vectors of the calibration parameters for the individual modules.

To derive the composite Jacobian matrices we use the chain rule for vector valued functions. The Jacobian function with respect to the input parameters is given by

$$J_i''(r,c'') = J_i''(r,c \mid c') = J_i'\big(f(r,c),c'\big) \times J_i(r,c)$$

where J_i, f, and c are the Jacobian function, the function and the calibration parameters for the first module respectively. The primed versions are the same but for the second module. Note that the matrix multiplication enforces the restriction that the number of input parameters of the second module must be the same as the number of output parameters of the first one. The Jacobian function with respect to the calibration parameters is acquired by

$$J_c''(r,c'') = J_c''(r,c \mid c') = [A \mid B]$$

where

$$A = J_i'\big(f(r,c),c'\big) \times J_c(r,c)$$
$$B = J_c'\big(f(r,c),c'\big)$$

In this way we can compose as many modules as we want into the final calibration function and it is easy to make changes to it since we can just remove or add the calibration modules that we would like to use. If we want the calibration parameters for a certain module to remain the same during a calibration we just have to return a zero matrix as J_c. This way we can easily control which parameters to calibrate.

6. Implementation and results

We implemented the calibration program in the Reachin API [6] for use in the Reachin Display. We used the Python script language [5][8] for the implementation according to the model described. To make it run faster the black boxes of the calibration modules were implemented in C++.

We found that it was not good trying to calibrate all parameters at once unless the calibration from the start is acceptable. In our tests the minimiser only finds local minima with quite large errors if we do this. This results in some very bad calibrations. We found that the best calibration was acquired by first calibrating only the haptics device (transformation between encoder angles and world coordinates) and not change the OpenGL parameters. Most of the time this is enough since the default standard OpenGL parameters provided with the Reachin Display are general enough. However to achieve a better calibration and get the sizes to match better we can use the new calibration as starting point and let the minimiser find a minimum from there. This usually results in the best calibrations.

If this strategy is used the result is good. We get a precision to within a few millimetres in the recommended work area of the haptics device. The precision outside this area is also good but differs a little depending on the sample set used. As a worst case the calibration can be off by a centimetre but this is usually in the extreme positions of the haptics device and does not really matter since it should not be used in these positions anyway. To get a good calibration at least 5 correction samples are needed. Any less than that the calibration is fine at the sample points but is poor in the space between them. It is also important to have the samples spread out to cover as much as possible of the work area, otherwise the calibration could get bad in the area without any sample.

7. Discussion and conclusions

The simple calibration utility makes it easy to make individual calibration files for each user of the device. This is preferred to having a general calibration for all users. The problem of not having individual calibrations is that if, for example, two people of different heights were to use the display they would have a different distance from the eyes to the mirror and hence the same calibration would appear different for each of them. To make it work they both have to have their head in the same position and have the same inter-ocular distance, which will mean that at least one of the users will experience poor calibration.

The calibration can never be better than the sample set given to the calibration routine. Since these samples are based on a user's visual experience there is bound to be errors while specifying the samples. It is impossible to get the samples exactly right and hence the calibration will always have this extra error measure which it cannot compensate for. The calibration will always be adversely affected by any poor samples given by the user.

Currently the biggest problem is finding a global minimum in the presence of many local minima. The current strategy of randomly restarting minimisation sometimes results in slow performance that does not always find the global minimum. In future we intend to explore alternative minimisation routines that are better able to cope with these types of functions. Nevertheless the performance of the current system is satisfactory for our purposes since calibration only needs to be performed occasionally but should be as accurate as possible.

References

[1] D. Stevenson, K. Smith, J. McLaughlin, C. Gunn, J. Veldkamp, and M. Dixon. Haptic workbench: A multisensory virtual environment. In Proc. SPIE

Stereoscopic Displays and Virtual Reality Systems VI, volume 3639, pages 356--366, 1999.

[2] Thomas Harold Massie. Initial Haptic Explorations with the Phantom: Virtual Touch Through Pointer Interaction. Master 's thesis, Massachusetts Institute of Technology, February 1996.

[3] Numerical Recipes in C. Second edition. Chapter 10.7. p. 428 - 430.

[4] The OpenGL 1.2.1 specification. Section 2.10.

[5] David M. Beazly. Python Essential Reference. ISBN 0735709017.

[6] Thurfjell L, Lundin A, McLaughlin J, A Medical Platform for Simulation of Surgical Procedures, In Westwood et al. (Eds), Medicine Meets Virtual Reality, IOS Press, 2001; pp. 509-514.

[7] www.opengl.org.

[8] www.python.org.