What do I need to do? – a short introduction to 3D navigation.

The navigation library TDxNavlib.dll provides all the math's to use a 3DMouse – SpaceMouse - to move around in a 3D application. After you are done, when a user grabs the SpaceMouse cap and displaces it, the navigation will library automatically receive the SpaceMouse data, query your application for view data, mangled the two together and then write back the changed view values to your application. How the 3d model moves in the view is chosen by the user by setting his preferred navigation mode in the 3Dconnexion properties panel.

The navigation library knows how to get 3D data from the SpaceMouse, how to mangle 3D data with view data to produce smooth movement in various shapes and sizes and the navigation library knows what data it needs to produce the smooth movement as well as other cool effects - you do not need to do this.

However, the navigation library does not know how to get that data from your application and how to update the changes. This is where the application developer steps in.

What you are going to do is implement a collection of functions to get specific values from and set specific values back to the 3D application, and when the application starts up tell the navigation library what exactly these functions are.

The next few pages will describe a way of implementing the code behind the accessor and mutator functions supplied to the navigation library, which is not dependent on the programming language used to code them. In this tutorial we will use c#, but it is not much different in c++ or any other language.

First, though, a short description of the sdk's c# assembly as well as the c++ classes…

The C# TDx.SpaceMouse.Navigation3D assembly.

The TDx.SpaceMouse.Navigation3D assembly is the managed interface between the 3DMouse and your application.

In a nutshell it is boilerplate code to take care of marshalling between managed and unmanaged code. It initializes the navigation library with the required accessor and mutator collection and exposes the callbacks to the C# application as an interface: the public interface TDx.SpaceMouse.Navigation3D.INavigation3D.

Using the TDx.SpaceMouse.Navigation3D assembly.

To use TDx.SpaceMouse.Navigation3D the public interface INavigation3D needs to be implement for your application and passed to an instance of TDx.SpaceMouse.Navigation3D.

A simple OpenGL sample that does this is the GettingStarted sample in the SDK. In the sample the interface TDx.SpaceMouse.Navigation3D.INavigation3D is implemented by the public class NavigationModel, which can serve as a blueprint for your application's interface to the navlib. How to go about implementing the interface methods is described later in the GettingStarted tutorial.

A more advanced example of an implementation is the 3DxTestNL sample in the SDK. This is a simple 3D viewer application which can load *.obj models, navigate with a SpaceMouse, execute application commands at the press of a 3DMouse button, as well as supporting the advanced 2D Mouse (CadMouse) zoom commands. The code pertinent to the SpaceMouse is grouped in the SpaceMouse folder. In the 3DxTestNL sample, the interface TDx.SpaceMouse.Navigation3D.INavigation3D is also implemented by the public class NavigationModel.

(Here, to help keep an overview of the portions of the code referring to specific interfaces, the implementations have been extracted to separate interface source code files i.e., the public interface IModel implementation is coded in the ModelCallbacks.cs code file. Etc.)

The C++ TDx::SpaceMouse::Navigation3D::CNavigation3D class.

The CNavigation3D class is the equivalent of the C# Navigation3D assembly, but for unmanaged c++ code, and can be used as a base class for the implementation of the interface to the navlib. The 3DxWare SDK inc/SpaceMouse directory contains the header files that make up the CNavigation3D class.

Using the TDx::SpaceMouse::Navigation3D::CNavigation3D class.

To use TDx::SpaceMouse::Navigation3D::CNavigation3D either the shared project in the 3DxWare SDK inc/SpaceMouse directory or the directory itself can be added to your solution. Then the interface IAccessors needs to be implemented for your application. A simple example of an IAccessors implementation is the 3DxTraceNL sample in the SDK. In the 3DxTraceNL sample the interface TDx::SpaceMouse::Navigation3D.IAccessors is implemented by the public class CNavigationModel, which can serve as a template for your application's interface to the navlib.

What follows is a step-by-step tutorial showing how to go about adding SpaceMouse support to an existing application.

Getting Started – First steps.

When this section is completed you will be able to perform basic navigation using the 3D Mouse, the Views as well as the Fit buttons on the 3D Mouse will be functional. See the FirstSteps.csproj C# sample for the code implemented in this step. The SpaceMouse specific code is grouped into the SpaceMouse subfolder. Code not in the SpaceMouse folder is assumed to be available as part of the 3D application the integration is being developed for.

For C++, the procedure is effectively the same.

Determining Row or Column Vectors.

The first data structure we need to get from the application, if we can, is the camera or eye transform. This is the matrix that maps points in camera space to world space. In OpenGL, for example, this is the inverse of the *GL_MODELVIEW* matrix. In an application that does not supply a camera matrix, but stores the data as 'from, 'up' and 'to' vectors, the matrix needs to be built by hand. More about that later.

If a matrix is available, the camera transform matrix obtained from an application is generally a 4x4 matrix, where the last row or column is [0,0,0,1]. It can also be a 4x3 or 3x4 matrix where the missing row or column is taken to be [0,0,0,1]. If the last row is [0,0,0,1] then the navigation library needs to be configured for column vectors, otherwise, if the last column is [0,0,0,1], or the application doesn't use matrices for specifying the eye location, then the default (row vectors) is correct.

For the rest of the discussion row vectors will be assumed, if you have column vectors simply substitute column for row.

The Coordinate System Matrix

The coordinate system matrix maps points in application space into the 3D space used by the navlib. The navigation library's coordinate system has the y-axis pointing up, the x-axis pointing to the right and the z-axis pointing out of the screen. If you are lucky and the 3D application uses the same coordinate system then the matrix is the identity matrix and you are done determining this property, if not, then…

The position of the camera in world coordinates can be read from the last row vector of the camera transform matrix. If this is not the case, then the obtained matrix is very likely the world to camera transform and needs to be inverted. For an application that does not supply a camera matrix, the 'eye' position is all that is needed for now.

Once the camera position can be retrieved, the next task is to determine the coordinate system matrix the application uses by moving the camera or eye with the 2D Mouse.

The following method determines the inverse of the coordinate system matrix and then transposes the result: Start by looking at the front of an axis aligned box. The objective is to have two of the applications axes aligned with the sides of the viewport and the other pointing into or out of the screen. Take a note of the camera position (for example start = [-7.85, -54.37, 14.34, 1], then dolly the

camera/view quite a way away from the box and note the new camera position (end = [-7.84, -151.8, 18.98, 1]). Subtracting end-start = [0.01, -97.43, 4.64, 0], setting the small values to 0 and normalizing the vector gives [0, -1,0, 0]: This is the third row of the inverse matrix.

start = [-7.85, -54.37, 14.34, 1]
end = [-7.84, -151.8, 18.98, 1]
end-start = [0.01, -97.43, 4.64, 0] => 3rd row is [0, -1, 0, 0]
inv_m =[?, ?, ?, 0]
[?, ?, ?, 0]
[0,-1, 0, 0]
[0, 0, 0, 1]

For the first row pan the camera to the right while keeping the distance to the box, as well as the vertical position, constant, record the camera's position before (e.g. [-7.84, -151.8, 18.98, 1]) and after (e.g. [70.56, -151.15, 18.78, 1]) panning. Subtracting afterwards-before = [78.4, 0.65, -0.2, 0], then setting the small values to 0 and normalizing the vector gives us [1,0,0,0]: The first row of the inverse matrix. Note: When the camera is panned to the right, the box moves in the view to the left.

start = [-7.84, -151.8, 18.98, 1]
end = [70.56, -151.15, 18.78, 1]
end-start = [78.4, 0.65, -0.2, 0] => 1st row is [1, 0, 0, 0]
inv_m =[1, 0, 0, 0]
[?, ?, ?, 0]
[0,-1, 0, 0]
[0, 0, 0, 1]

Similarly, for the second row. move the camera up (the box moves down). In my case start=[-2.67, -150.62, 27.49], end=[-5.89, -145.43, 103.59], end-start=[-3.22, 5.19, 76.1] and the second row =[0,0,1,0].

start = [-2.67, -150.62, 27.49, 1]
end = [-5.89, -145.43, 103.59, 1]
end-start = [-3.22, 5.19, 76.1, 0] => 2nd row is [0, 0, 1, 0]
inv_m =[1, 0, 0, 0]
[0, 0, 1, 0]
[0,-1, 0, 0]
[0, 0, 0, 1]

Putting it all together, the inverse matrix is [1,0,0,0] [0,0,1,0] [0, -1,0,0] [0,0,0,1]. The coordinate system matrix is then its inverse (transpose in this case) and ISpace3D.GetCoordinateSystem() should return matrix =[1, 0, 0, 0]
[0, 0,-1, 0]
[0, 1, 0, 0]
[0, 0, 0, 1].

The Camera Matrix

The camera matrix maps points in camera space to world space.

For cameras facing the scene, the default camera coordinate system has the x-axis to the right, the y-axis up and the z-axis towards the user. An alternative is the OpenCV coordinate system which has the x-axis to the right, the y-axis down and the z-axis towards the scene.

In OpenGL, the camera matrix is the inverse of the *GL_MODELVIEW* matrix.

To build the matrix from 'from, 'up' and 'to' vectors the following calculations are required.

Note: Positions always have a 1, i.e. [-7.85, -54.37, 14.34, 1], and directions such as 'up' i.e. [-3.22, 5.19, 76.1, 0] always have a 0 as the last value. Any combination of position or direction vectors can be added and/or subtracted if, and only if, the fourth value of the result is either 0 (a direction) or 1 (a position).

As mentioned above the 'from' (eye) position is the last (4^{th}) row of the matrix. The 3rd row is calculated by normalizing z-axis = 'from' – 'to'. The first row is calculated by normalizing x-axis = 'up' * z-axis, and the 2nd row y-axis = z-axis * x-axis.

For example:

With from = [-231.706528, -219.705536, 205.244751, 1], to = [19.2557735, 0.650770187, 18.0377884, 1], up = [0.365209579, 0.396455675, 0.842285514, 0] then the matrix returned by IView.GetCameraMatrix() is

[0.724287, -0.689419, 0.010457, 0]
[0.353287, 0.384093, 0.853031, 0]
[-0.592112, -0.614145, 0.521756, 0]
[-231.706528, -219.705536, 205.244751, 1]

A simple check is to orientate the camera so that it is looking at the front of an axis aligned box as in the text about the coordinate system matrix. The first three rows of the calculated camera matrix should then be the same the transpose of the coordinate system matrix.

When the navlib invokes IView.SetCameraMatrix() the cameras 'from, 'up' and 'to' vectors should be updated. If the new matrix supplied by the navlib is:

[0.7071068, -0.7071068, 0, 0]
[0.3535534, 0.3535534, 0.8660254, 0]
[-0.6123725, -0.6123725, 0.5, 0]
[-153.398, -153.398, 125.2489, 1]

Then, if the 'to' position has no side effects, the simplest is
from = [-153.398, -153.398, 125.2489, 1],
to = from – 3^{rd} row = [-153.398, -153.398, 125.2489, 1] - [-0.6123725, -0.6123725, 0.5, 0] x r,
up = 2^{nd} row = [0.3535534, 0.3535534, 0.8660254, 0]
where r>=1. However, if the 'to' position has no side effects, then r needs to be calculated so that the 'to' position is the nearest to where it used to be.

The Front View Matrix

The navlib uses this matrix to set the orientation of the camera when the user presses the 'Front View' button on a 3D Mouse. All other orientations are calculated from this matrix.

The simplest method to retrieve the matrix is to view the front face of an axis aligned box in the 3D application and then get the camera matrix. With the last row set to [0,0,0,1] the matrix will normally be all zeros (or very close to zero) except for one non-zero value in each row. This is the matrix to return from ISpace3D.GetFrontView().

Perspective and/or Orthographic

If the viewport projection is perspective, then the IView.IsViewPerspective implementation needs to return true and IView.GetViewFrustum needs to be implemented. Conversely, if the viewport projection is orthographic IView.IsViewPerspective must return false and both IView.GetViewExtents and IView.SetViewExtents require implementations.

The Model Extents

The model extents is the last piece of information that the navlib needs to be able to do a minimum of 3D navigation.

Updating and refreshing the view

In the C# implementation, before any of the methods are invoked that change a frame the Navigation3D.TransactionChanged event is invoked with the TransactionEventArgs.IsBegin property set to true. After all the values have been set for the frame the Navigation3D.TransactionChanged event is invoked with the TransactionEventArgs.IsEnd property set to true. The Navigation3D.TransactionChanged event can be used to trigger a view refresh.

Whenever the navlib starts or stops moving the camera, i.e., the user displaces the 3DMouse cap or lets go again, the Navigation3D.MotionChanged event is invoked with MotionEventArgs.IsNavigating set to either true or false, respectively. These are good points in time to initialize/finalize undo objects, adaptive degradation and possibly start or stop animation loops.

The above also applies equivalently for the unmanaged c++ implementation. Here the coinciding calls are IState.SetTransaction and IState.SetMotionFlag with the function parameters set to either !=0 and true or 0 and false.

The Rotation Pivot Cue (PivotCue)

This short section explains what needs to be done to display an icon in the 3D scene to represent the center of rotation of the view. Displaying a cue greatly advances the user's ability to easily navigate in the view. Due to the cue the user can see where in the scene the 3D mouse is acting. See the PivotCue.csproj C# sample, which takes the FirstSteps sample this step further by implementing the IPivot interface methods and writing to the pivot.* properties.

During normal operation, the navigation library controls the position of the pivot, however if required the application or the user can override the library's internal algorithms.

The position of the pivot.

The navigation library calls the interface implementation IPivot.SetPivotPosition with a point position in world (in OpenGL, *GL_MODELVIEW)* coordinates to set the pivot location. Normally, during view navigation, the position does not require updating. When the movement has finally completed (IState.SetMotionFlag(false)), a new position is calculated by the library and IPivot.SetPivotPosition is then invoked with the new world point.

Overriding the position of the pivot.

The navigation library allows the 3D application to gain control of the position of the rotation pivot. The application can then implement a command to allow the user to manually set the rotation pivot, typically using the mouse middle button and optionally a keyboard modifier.

To activate the manual pivot the application writes true to the library's "pivot.user" (PropertyNames.PivotUser) property. The side effect is that the library invokes IPivot.GetPivotPosition to acquire the world coordinates of the pivot, which it then uses as the center of rotation until the application writes the "pivot.user" property again with either true or false, or writes a new pivot position to the "pivot.position" (PropertyNames.PivotPosition) property.

Showing and hiding the pivot icon

IPivot.SetPivotVisible(bool show), where show is either true of false, is invoked whenever the visibility of the pivot icon should change. The call is asynchronous and is not limited to within a frame transaction.

Intelligent 3D Navigation (I3DN)

The navigation library implements advanced navigation algorithms and features that rely on the ability ‘see’ what the user is looking at. The heart of I3DN is hit testing. The I3DN.csproj C# sample continues where the PivotCue sample left off and implements the IHit interface methods.

Hit testing for the navigation library

Whether the application uses ray casting or any other algorithm for hit testing is not known by the navigation library and it assumes ray casting. The parameters the navigation library supplies define either a cylindrical beam for an orthographic projection or a cone beam for a perspective projection. These are set by the interface methods IHit.SetLookFrom(): the origin of the beam, IHit.SetLookDirection(): the direction of the beam, and IHit.SetLookAperture() : the diameter of the beam on the near plane. To query the result of the hit-test the navigation library invokes the interface method IHit.GetLookAt().

Depending on the active algorithm, the navigation library may want to limit the result of the hit test to the user’s currently selected objects. In this case the interface method IHit.SetSelectionOnly() is invoked with a value of true.

Exporting Commands and Images for the 3D Mouse buttons.

The final step is to export the application commands and their image representations to allow the user to execute them from 3D mouse buttons and radial menus. The GettingStarted.csproj C# sample incorporates all the previous steps and adds exporting commands. Because the 3D viewer the sample represents is very simple and does not have any useful commands, the sample pops up a message box for the executed command.

The command/action set.

The 3D Mouse and the 3DxWare driver groups commands into command or action sets, where only one command set can be active at a time. This allows the application exporting the commands, when required to, to group commands according to the task the user is performing and the user to map them to the 3D Mouse buttons per task. The application sets the active command set by writing the PropertyNames.CommandsActiveSet property. In the sample this is eventually done by NavigationModel.ActiveCommandSet.

The command set structure.

A command or action set can be considered to represent a menu bar, which has menus (command categories) and menu items (commands). In other words, a command set is a tree structure where commands are grouped together and allows the user to easily find his favorite command when presented with the complete set.