3Dconnexion JavaScript Framework

Author:	Nuno Gomes, Markus Bonk
Participant(s):	
Cc:	
Classification:	3Dconnexion Internal Only (Confidential)

Document history summary:

Version	Author	Date	Status	Comment
1.0	Nuno Gomes	2014-Mar-28	Draft	Preliminary documentation.
1.1	Markus Bonk	2014-Dec-15	Draft	Added description of new methods and updated the existing example code.
1.2	Markus Bonk	2015-Apr-14	Draft	Added putViewExtents description.
1.3	Markus Bonk	2015-Apr-20	Draft	Added console messages and pointer position.
1.4	Markus Bonk	2015-May-05	Draft	Added hit-testing and construction plane descriptions.
1.5	Markus Bonk	2015-May-29	Draft	Added the coordinate system and front view descriptions.
1.6	Markus Bonk	2015-Jun-09	Draft	Included the name of the properties being accessed by the getter and setter methods. Added missing property descriptions.
1.7	Markus Bonk	2015-Sep-14	Draft	Added rotatable property to view.
1.8	Markus Bonk	2016-Dec-08	Draft	Added commands and images.
1.9	Markus Bonk	2017-Jan-24	Draft	Added note about “Default” ActionSet. Corrected the ActionSet and Category constructors.
1.10	Markus Bonk	2017-Oct-19	Draft	Documented transaction property.
1.11	Markus Bonk	2018-Jan-01	Draft	Documented changes to matrix order and mutator naming.
1.12	Markus Bonk	2018-Nov-05	Draft	Documented frame timing properties.
1.13	Markus Bonk	2019-Feb-04	Draft	Fix incorrect coordinate system example.
1.14	Nuno Gomes	2020-Jul-08	Draft	Removed jQuery dependency
1.15	Markus Bonk	2022-Dec-06	Draft	Added the walk mode properties.
1.16	Markus Bonk	2024-Oct-11	Draft	Added module support and reworked some examples.

1. Introduction

1.1 Purpose

This document includes preliminary information on the 3DconnexionJS Framework API. The development of 3DconnexionJS is “work-in-progress”.

1.2 Target Audience

The target audience of this document is all 3Dconnexion employees involved in the development of 3DconnexionJS.

1.3 Document History

Version 1.0

- First document version. Includes preliminary information on the 3DconnexionJS framework API. The status of 3DconnexionJS as of document version 1.0 is “work-in-progress / experimental”.

Version 1.1

- Updated the example code for the update in the samples to glMatrix v2.2. Added description of the methods used by the navlib to support the views commands. The documentation reflects the methods implemented in the webglapp.html sample. Includes preliminary information on the 3DconnexionJS framework API. The status of 3DconnexionJS as of document version 1.1 is “work-in-progress / experimental”.

Version 1.2

- Added putViewExtents description.

Version 1.3

- Added console message flags description. Added description of the mouse pointer position.

Version 1.4

- Added section on hit-testing. Added an example for the construction plane.

Version 1.5

- Clarified that the methods implemented in the client interface are getter and setter methods for properties. Included the property names. Added selection and pivot properties.

Version 1.8

- Added on3dmouseCreated description
- Added classes used for command and image export

Version 1.9

- Added note on “Default’ action set identifier
- Corrected ActionSet and Category constructors
- Corrected create3dmouse example

Version 1.10

- Added ‘transaction’ property documentation

Version 1.11

- Documented changes to the matrix order
- Changed the naming to match 3DconnexionJS v0.5 mutators

Version 1.12

- Documented Frame properties.

Version 1.13

- Fix incorrect coordinate system example.

Version 1.14

- Removed dependency on jQuery.

Version 1.15

- Added documentation for the properties introduced for the walk and first-person motion models.

1.4 References

[1] 3Dconnexion, “3DxWare SDK 3.0 for Windows”; v. 3.0.2, revision 9261; 3Dconnexion Documentation; 11-Sep-2013.

[2] Tavendo1; Web Application Messaging Protocol (WAMP) version 1; http://wamp.ws/spec/wamp1/; Tavendo GmbH; 2014.

[3] Tavendo2; Autobahn|JS; http://autobahn.ws/js/. Tavendo GmbH; 2014.

2. Overview

The 3DconnexionJS Framework is 3Dconnexion's Application Programming Interface (API) for Javascript applications currently under active development. The information included in this document is subject to change and the documented API is likely to be modified without backward compatibility.

2.1 3Dconnexion's input devices

For a detailed explanation on the operation of 3Dconnexion's input device see reference [1]. Note that 3DconnexionJS does not require any additional 3Dconnexion Software Development Kit (SDK) packages.

2.2 3DconnexionJS and Navigation Library

Departing from a conventional Software Development Kit for an input device (see [1]), the 3DconnexionJS API uses 3Dconnexion's Navigation Library. The Navigation Library is a module that implements regular navigation modes used with a 3D mouse (object, camera, target-camera, "helicopter") freeing the JS program developers from having to re-implement the control features expected by 3D mouse users.

The 3DconnexionJS API provides a wrapper to the Navigation Library. The Navigation Library will be the base of 3Dconnexion's SDKs for multiple platforms.

3. Usage

The 3DconnexionJS framework is implemented in file `3dconnexion.cjs` and uses version 1 of the WebSocket Application Messaging Protocol (WAMP, see [2]) to transport 3D mouse data into the JavaScript context of a web application. The messaging data is formatted in JavaScript Object Notation (JSON).
The WebSocket connection occurs over the system's loopback adapter with the framework only accepting secure socket connections. Web applications can be delivered by HTTP or HTTPS.

A web application includes the 3DconnexionJS framework either by including the script `3dconnexion.min.js` in a <script/> statemen (see the webglapp.html sample in the SDK):

```
<script type="text/javascript" src="3dconnexion.min.js"></script>
```

Module support is available using the `3dconnexion.module.min.js f`or example :

```
<script type="importmap">
  {
    "imports": {
      "3dconnexion": "./3dconnexion.module.min.js"
    }
  }
</script>
<script type="module">
  import * as TDx from "3dconnexion";
  // Navigation model
  var navigationModel = {};
  var spaceMouse = null;
</script>
```

This is demonstrated in the web_threejs.html and ThreeJSMultiCanvas.html samples.

3.1 3DconnexionJS Framework Dependencies

In addition to 3Dconnexion's driver and Navigation Library software, the 3DconnexionJS Framework depends on:

- Autobahn|JS
  Autobahn|JS Version 0.8.2.1 is an open-source implementation of WAMP v1 and copyright of Toledo GmbH. See [2] for additional information.

- CryptoJS JavaScript library of crypto standards.
  Currently Version 4.2.0. is used.

- Support for the JavaScript ES2020 feature globalThis is required.

  Both 3DconnexionJS packages include the AutobahnJS and CryptoJS dependencies.

3.2 Coordinate System

Internally, the 3DconnexionJS Framework uses a right-handed coordinate system with the Y-axis up and the camera looking down its negative Z-Axis. The Framework will query the client for the client's coordinate system and orientation of the front view when a connection is established (see the client interface properties, later) so that the positions, etc., can be passed to the 3DconnexionJS Framework in the client's coordinate system.

3.3 Console Messages

Two flags control the output of console messages during runtime. _3DCONNEXION_DEBUG controls the output of the 3DconnexionJS Framework messages and AUTOBAHN_DEBUG controls the output of the websocket protocol layer messages. To enable output of one or the other the corresponding flag needs to be defined as true before the 3DconnexionJS Framework is included.

3.4 The _3Dconnexion Class

Web applications create a `_3Dconnexion` object to use the 3DconnexionJS framework.

3.4.1 Constructor

```
_3Dconnexion(object);
```

Arguments:

`object`

  Required. This is the client object which supplies the callbacks the _3Dconnexion instance uses to pass data back to the client, i.e. the navigation model. The methods that the client object is required to define are described below.

3.4.2 Methods

3.4.2.1 connect()

Establishes a connection to the 3Dconnexion Navigation Server using WAMP as a WebSocket sub-protocol. On a successful connection the client object's `onConnect()` method is invoked. If the client object used to construct the _3Dconnexion object has defined an `onDisconnect()` method, this will be called on a disconnect (see the description of the Client methods below).

```
connect()
```

Arguments:
None.

Example:

```
function init3DMouse() {
  spaceMouse = new TDx._3Dconnexion(navigationModel);
  if (!spaceMouse.connect()) {
    if (TRACE_MESSAGES)
      console.log("Cannot connect to 3Dconnexion NL-Proxy");
  }
}
```

3.4.2.2 create3dmouse()

Creates a connection to a 3D mouse and a navigation controller for the view. On a successful creation the clients `on3dmouseCreated()` method is invoked.

```
create3dmouse (canvas, appname, option)
```

Arguments:

`canvas`

Required. This is the canvas/window/view instance, typically the canvas used to draw the scene. The canvas must be focusable as the _3Dconnexion object adds `onfocus` and `onblur` call-backs to the canvas instance to be able to determine keyboard and 3D mouse focus.

`appname`

Required. The `appname` argument is used to identify the application for the 3D mouse property panels and displayed to the user.

`options`

Optional. The `option` argument is used to specify configuration options to the navigation library. The default value is _3Dconnexion.nlOptions.none. The only other supported value is _3Dconnexion.nlOptions.rowMajorOrder which is used to specify that the matrices are in row major order. By row major order we mean that the vectors are column vectors i.e. transforms are multiplied on the left: v' = T * v; as normally taught in school.

Example:

```
navigationModel.onConnect = function () {
  // create the 3dmouse connection.
  spaceMouse.create3dmouse(canvas, appName);
};
```

Remarks:

The default matrix representation uses column major order. Should row major order be required then the property _3Dconnexion.columnMajorOrder must be set to false before invoking create3dmouse().

3.4.2.3 delete3dmouse()

Deletes the 3D mouse controller.

```
delete3dmouse()
```

Arguments:
None.

3.5 The _3Dconnexion.ActionTree Class

Container class used to build a tree of commands to be exposed to the 3Dconnexion Properties Buttons panel

3.5.1 Constructor

```
_3Dconnexion.ActionTree();
```

Arguments:
None.

3.5.2 Methods

3.5.2.1 push()

The push() method adds a new item to the end of the array of child nodes, and returns the node.

```
push(node)
```

Arguments:
`node`.
    Required. The `node` must be a `_3Dconnexion.ActionSet` instance.

Example:

```
let actionTree = new TDx._3Dconnexion.ActionTree();
let buttonBank = actionTree.push(
  new TDx._3Dconnexion.ActionSet('ACTION_SET_ID', 'Custom action
set'));
```

3.5.3 Properties

3.5.3.1 nodes[]

Array of child nodes.

3.6 The _3Dconnexion.ActionSet Class

Class that describes a set of commands. An action set can also be considered to be a button bank, a menubar, or a set of toolbars.

3.6.1 Constructor

`_3Dconnexion.ActionSet(actionSetId, label)`

Arguments:

`actionSetId`

Required. The `actionSetId` is an application defined unique string identifier for the application command set. This is the value which is updated to the `commands.activeSet` property for the 3Dconnexion UI to select the correct command context.

`label`

Required. This is displayed to the user in the 3Dconnexion UI to identify the action set.

Note: An action set with actionSetId === `'Default'` will not have its label displayed to the user. `'Default'` is the recommended actionSetId in applications that only require a single action set.

3.6.2 Methods

3.6.2.1 push()

The push() method adds a new item to the end of the array of child nodes, and returns the node.

`push(node)`

Arguments:

`node`.

Required. The `node` can be a `_3Dconnexion.Category or _3Dconnexion.Action` instance.

Example:

```
let actionTree = new TDx._3Dconnexion.ActionTree();
let buttonBank = actionTree.push(
  new TDx._3Dconnexion.ActionSet('ACTION_SET_ID', 'Custom action set')
);
buttonBank.push(
  new TDx._3Dconnexion.Action('ID_OPEN', 'Open', 'Open file')
);
```

3.6.3 Properties

3.6.3.1 id

Application defined unique string identifier.

3.6.3.2 label

User readable label.

3.6.3.3 type

Node type = `SI_ACTIONSET_NODE`

3.6.3.4 nodes[]

Array of child nodes.

3.7 The _3Dconnexion.Category Class

Class that allows commands to be grouped by category so that the user can find commands easily.

3.7.1 Constructor

`_3Dconnexion.Category(categoryId, label)`

Arguments:

`categoryId`

Required. The `categoryId` is an application defined unique string identifier for the category.

`label`

Required. This is used in the 3Dconnexion UI to identify the category.

3.7.2 Methods

3.7.2.1 push()

The push() method adds a new item to the end of the array of child nodes, and returns the node.

`push(node)`

Arguments:

`node`.

Required. The `node` can be a `_3Dconnexion.Category` or `_3Dconnexion.Action` instance.

Example:

```
let actionTree = new TDx._3Dconnexion.ActionTree();
let buttonBank = actionTree.push(
  new TDx._3Dconnexion.ActionSet('ACTION_SET_ID', 'Custom action set')
);
let fileNode = buttonBank.push(
  new TDx._3Dconnexion.Category('CAT_ID_FILE', 'File')
);
```

3.7.3 Properties

3.7.3.1 id

Application defined unique string identifier.

3.7.3.2 label

User readable label.

3.7.3.3 type

Node type = `SI_CATEGORY_NODE`

3.7.3.4 nodes[]

Array of child nodes.

3.8 The _3Dconnexion.Action Class

Class that describes an application command

3.8.1 Constructor

`_3Dconnexion.Action(commandId, label, description)`

Arguments:

`commandId`

Required. The `commandId` is an application defined unique string identifier for the application command. This is the value which is set to the `commands.activeCommand` property when the user presses the button which has the command assigned.

`label`

Required. This is used in the 3Dconnexion UI to identify the command.

`description`

Optional. Text that allows the user to understand what the command does.

3.8.2 Methods

None

3.8.3 Properties

3.8.3.1 id

Application defined unique string identifier.

3.8.3.2 label

User readable label.

3.8.3.3 description

User readable text describing the action.

3.8.3.4 type

Node type = `SI_ACTION_NODE`

3.9 The _3Dconnexion.ImageCache Class

Container class for the command images.

3.9.1 Constructor

`_3Dconnexion.ImageCache()`

Arguments:
None.

3.9.2 Methods

3.9.2.1 push()

The push() method adds a new item to the end of the array of images.

`push(item)`

Arguments:
`item`.
    Required. The `item` must be an instance of `_3Dconnexion.ImageItem`.

Example:
```
let images = new TDx._3Dconnexion.ImageCache();
images.push(
  _3Dconnexion.ImageItem.fromURL('images/open.png', 'ID_OPEN')
);
```

3.9.3 Events

3.9.3.1 onload()

The onload() event is called when all outstanding http get requests for the images have been satisfied.

`onload()`

Arguments:
None.

Example:
```
let cache = new TDx._3Dconnexion.ImageCache();
cache.onload = function () {
  spaceMouse.update3dcontroller({ images: cache.images });
};
```

3.9.4 Properties

3.9.4.1 images[]

Array of `_3Dconnexion.ImageItem` items

3.9.4.2 outstanding_requests

number of image load requests which are outstanding

3.10 The _3Dconnexion.ImageItem Class

Container class for a command image.

3.10.1 Constructor

```
_3Dconnexion.ImageItem()
```

Arguments:
None.

3.10.2 Methods

None.

3.10.3 Properties

3.10.3.1 id

The `id` is an application defined string identifier. To associate an image with a `_3Dconnexion.Action` item, the ids need to be identical.

3.10.3.2 type

The ImageItem instance type. Currently only `_3Dconnexion.SiImageType_t.e_image` and `_3Dconnexion.SiImageType_t.e_none` are supported.

3.10.3.3 index

Defines the image index when using a collection of sub-images contained in a wider image (image list). The default is 0.

3.10.3.4 buffer

Used internally to store the images data.

3.10.3.5 data

Accessor to the internal buffer

3.10.3.6 status

Status of the XMLHttpRequest to load the image

3.10.4 Events

3.10.4.1 onload()

The onload() event is called when an outstanding XMLHttpRequest get request for the image has completed.

```
onload()
```

3.10.5 Functions

3.10.5.1 fromURL()

The `fromURL` function creates a `_3Dconnexion.ImageItem` object from an URL.

```
_3Dconnexion.ImageItem.fromURL(url, id)
```

Arguments:
`url`

Required. The `url` of the image

`id`

Required. The `id` of the command the image is to be associated with.

Example:

```
var images = new TDx._3Dconnexion.ImageCache();
images.push(
  TDx._3Dconnexion.ImageItem.fromURL('images/open.png', 'ID_OPEN')
);
```

3.10.5.2 fromImage()

The `fromImage` function creates a `_3Dconnexion.ImageItem` object from an arrayBuffer.

```
_3Dconnexion.ImageItem.fromImage(arrayBuffer, id)
```

Arguments:

`arrayBuffer`

Required. The `arrayBuffer` containing the image

`id`

Required. The `id` of the command the image is to be associated with.

3.11 Client interface

The client interface consists of methods implemented by the web application (the client of the `_3Dconnexion` object) and used by the framework to access client properties or signal events. The Navigation Server is a program running in the user host context and implements the WebSocket server and the wrapper code to access the Navigation Library and 3D mouse driver. The 3Dconnexion Navigation Library implements the scene navigation models by querying the client for properties of the scene or the view/camera, applying transformations derived from the input 3D mouse data to the acquired values and then setting the updated properties in the client. The 3Dconnexion Navigation Library accessing the client properties through getter and setter methods implemented by the client.

3.11.1 Event onConnect()

Mandatory method which is called when a successful connection has been made to the 3Dconnexion Navigation Server allowing the client to finish initialization. Typically the client will call the _3Dconnexion object’s create3dmouse() method.

```
onConnect()
```

Implementation:
Mandatory.

Arguments:
None.

Example:
```
      navigationModel.onConnect = function () {
        const namePos = window.location.pathname.lastIndexOf("/");
        const name = window.location.pathname.substring(namePos + 1);
        const canvas = document.getElementById('container');
        spaceMouse.create3dmouse(canvas, name);
      };
```

3.11.2 Event onDisconnect()

Optional method which is called after an unsuccessful connection attempt to the 3Dconexion Navigation Server

```
onDisconnect(reason)
```

Implementation:
Required for any post disconnect processing.

Arguments:
`reason`
    An error number.

Example:
```
      navigationModel.onDisconnect= function (reason) {
        console.log("3Dconnexion NL-Server disconnected " + reason);
      };
```

3.11.3 Event on3dmouseCreated()

Optional method which is called when the 3DMouse controller instance is successfully created. Typically the client will initialize the application\web page specific button data here.

```
On3dmouseCreated()
```

Implementation:
Optional.

Arguments:
None.

Example:
```
      navigationModel.on3dmouseCreated = function () {
        let actionTree = new TDx._3Dconnexion.ActionTree();
        let actionImages = new TDx._3Dconnexion.ImageCache();
        actionImages.onload = function () {
          spaceMouse.update3dcontroller({
            images: actionImages.images
          });
        };
        let buttonBank = actionTree.push(
          new TDx._3Dconnexion.ActionSet('ACTION_SET_ID', 'Custom action
      set')
        );
        getApplicationCommands(buttonBank, actionImages);

        spaceMouse.update3dcontroller({
          commands: {
            activeSet: 'ACTION_SET_ID', tree: actionTree
        } });
      };
```

3.11.4 Property ‘motion’

Event `onStartMotion()`
Method called by the Navigation Server once when it is about to start moving the view and updating the view matrix. The client can use this event to initialize the animation loop. When the Navigation Library has finished moving the view onStopMotion() will be invoked (see below). onStartMotion() and onStopMotion() occur in pairs.

```
onStartMotion()
```

Implementation:
Required to start the animation frame loop.

Associated Navlib property: “motion”

Arguments:
None.

Example:
```
      navigationModel.onStartMotion = function () {
        if (!animating) {
          animating = true;
          window.requestAnimationFrame(render);
        }
      };
```

Event `onStopMotion()`
Method called by the Navigation Server after it has invoked onStartMotion() when it will no longer update the view. onStartMotion() and onStopMotion() occur in pairs.

```
onStopMotion()
```

Implementation:
Required to stop the animation frame loop

Associated Navlib property: “motion”

Arguments:
None.

Example:
```
      navigationModel.onStopMotion = function () {
        animating = false;
      };
```

3.11.5 Property ‘coordinateSystem’

Getter Method `getCoordinateSystem()`

Method called by the Navigation Library when a connection is established to determine the pose of the application’s world coordinate system described in the Navigation Library’s coordinate system (x to the right, y-up, and z out of the screen). In most cases the matrix will be the inverse of the front view’s matrix. The matrix is returned as a 1 x 16 array.

```
getCoordinateSystem()
```

Implementation:
Required to convert between the client coordinate system to the navigation library coordinate system

Arguments:
None.

Example:
```
      navigationModel.getCoordinateSystem = function () {
        // The following is for X to the right, Z-up, Y into the screen
        return [1,0,0,0, 0,0,-1,0, 0,1,0,0, 0,0,0,1];
      };
```

3.11.6 Property ‘view.constructionPlane’

Getter Method `getConstructionPlane()`

The method returns a 1x4 java array containing the general form of the construction plane. This property is used by the Navigation Library to distinguish views used for construction in an orthographic projection: typically the top, right left etc. views. The library assumes that when the camera’s look-at axis is parallel to the plane normal the view should not be rotated.

```
getConstructionPlane()
```

Implementation:
Required for the orthographic 2d view algorithms.

Arguments:
None.

Example:
```
      navigationModel.getConstructionPlane = function () {

        // a point on the construction plane
        var origin = new THREE.Vector3(0., 0., 0.);
        origin.applyMatrix4(grid.matrixWorld);

        // In this sample the up-axis is the y-axis
        var yAxis = new THREE.Vector3();
        grid.matrixWorld.extractBasis(new THREE.Vector3()
             , yAxis
             , new THREE.Vector3());
        var d0 = yAxis.dot(origin);
        // return the plane equation as an array
        return [yAxis.x, yAxis.y, yAxis.z, -d0];
      };
```

Note:
If there is no construction plane return null.

3.11.7 Property 'view.fov'

Getter Method `getFov()`
Method called by the Navigation Library when it needs to know the field-of-view (fov) of the camera/view.The method returns the vertical fov in radians.

`getFov()`

Implementation:
Required for zooming in perspective projections.

Arguments:
None.

Example:
```
      navigationModel.getFov = function () {
        return gl.fov;
      };
```

Setter Method setFov()
Method called by the Navigation Library when it wants to update the fov of the camera/view.

`setFov(fov)`

Implementation:
Required for zooming in perspective projections.

Arguments:
`fov`
    The fov in radians.

Example:
```
      navigationModel.setFov = function (fov) {
        gl.fov=fov;
      };
```

3.11.8 Property 'view.perspective'

Getter Method `getPerspective()`
The getPerspective method returns if the view is a perspective projection.

`getPerspective()`

Implementation:
Required to distinguish between orthographic and perspective navigation algorithms.

Arguments:

None.

Example:
```
      navigationModel.getPerspective= function () {
        return true;
      };
```

3.11.9 Property 'view.extents'

Getter Method `getViewExtents()`

The method returns a 1x6 java array containing the min and max extents of the view bounding box. The view extents defines the visible viewing volume of an orthographic projection in view coordinates as min = [left, bottom, -far] and max = [right, top, far].

`getViewExtents()`

Implementation:
Required for orthographic projections.

Arguments:
None.

Example:
```
      navigationModel.getViewExtents = function () {
        return [view.left, view.bottom, -view.farVal
          , view.right, view.top, view.farVal];
      };
```

Setter Method `setViewExtents()`

The method receives a 1x6 java array containing the min and max extents of the view bounding box. The view extents defines the visible viewing volume of an orthographic projection in view coordinates as min = [left, bottom, far] and max = [right, top, near]. The Navigation Library will call setViewExtents when navigating in an orthographic projection in order to zoom the view.

`setViewExtents(extents)`

Implementation:
Required for zooming orthographic projections.

Arguments:
`extents`
    1x6 array containing the min and max extents of the view.

Example:
```
      navigationModel.setViewExtents = function (data) {
        // Only change the size of the view,
        // ignore the back and front clipping planes
        view.left = data[0];
        view.right = data[3];
        view.bottom = data[1];
        view.top = data[4];
      };
```

3.11.10 Property 'view.frustum'

Getter Method `getViewFrustum()`

The method returns a 1x 6 java array containing the frustum of the perspective view/camera in camera coordinates. This may be called by the Navigation Library when it needs to calculate the field-of-view of the camera, or during algorithms that need to know if the model is currently visible within the frustum. The Navigation Library will not write to the 'view.frustum' property, instead it will write to the ‘view.fov’ property and leave the client to change the frustum as required.

`getViewFrustum()`

Implementation:
Required for perspective projections.

Arguments:
None.

Example:
```
      navigationModel.getViewFrustum = function () {
        const tan_halffov = Math.tan(gl.fov * Math.PI / 360);
        const bottom = -camera.near * tan_halffov;
        const left = bottom *camera.aspect;
        if (TRACE_MESSAGES)
          console.log('frustum=[' + left + ', ' + -left + ', ' + bottom + ',
      ' + -bottom + ', ' + camera.near + ', ' + camera.far + ']');
        return [left,-left,bottom,-bottom, camera.near, camera.far];
      };
```

Remarks:
The values in the array are the same as those passed to glFrustum in OpenGL.

3.11.11 Property ‘view.affine’

Getter Method `getViewMatrix()`
The method returns a java array containing the view 4x4 column major matrix. This may be called by the Navigation Library when calculating a next frame. The view matrix describes the transform from view coordinates to world coordinates of the view/camera position. Generally getViewMatrix() is called by the library at the beginning of motion.

`getViewMatrix()`

Implementation:
Required for navigation.

Arguments:
None.

Example:
```
      navigationModel.getViewMatrix = function () {
        return Array.prototype.slice.call(cameraMatrix);
      };
```

Remarks:
OpenGL and WebGL have both implemented matrices as column major.

Setter Method `setViewMatrix()`
Method called by the Navigation Library when it has calculated the position of the view for the next frame.

`setViewMatrix(columnMajorMatrix)`

Implementation:
Required for navigation.

Arguments:
`columnMajorMatrix`

The 4x4 view matrix as a column major order array.

Example:
```
	navigationModel.setViewMatrix = function (data) {
		cameraMatrix = data;
		updateScene();
	};
```

3.11.12 Property 'view.rotatable'

Getter Method `getViewRotatable()`

The method returns a true of false. This may be called by the Navigation Library when calculating a next frame. The rotatable describes whether the view in an orthographic project can be rotated.

```
getViewRotatable()
```

Implementation:
Required for 2d navigation.

Arguments:
None.

Example:
```
	navigationModel.getViewRotatable = function () {
	  return false;
	};
```

Remarks:
The Navigation Library will treat non-rotatable views as pure pan-zoom views and also disable the views keys (Right, Left, Top, etc.). To allow the navigation library to rotate the view to an alternative orientation such as Top, define the view as rotatable and a corresponding construction plane.

3.11.13 Property 'views.front'

Getter Method `getFrontView()`

Method called by the Navigation Library when a connection is established to determine the pose of the front view. When the user presses the 'Front' button on a 3D Mouse this will be the pose the Navigation Library switches to. All other view orientations are calculated from this.

```
getFrontView()
```

Implementation:
Required for the named views algorithms

Arguments:
None.

Example:
```
	navigationModel.getFrontView = function () {
	  // In this sample the front view corresponds to the world pose
	  return [1,0,0,0, 0,1,0,0, 0,0,1,0, 0,0,0,1];
	};
```

3.11.14 Property 'model.extents'

Getter Method `getModelExtents()`

Method called by the Navigation Library when it needs to know the bounding box of the model. One example of when this will happen is when the library needs to zoom the view to the extents of the model. The extents are returned as an array containing the min and max values of the bounding box in world coordinates.

```
getModelExtents()
```

Implementation:
Required for the pivot and zoom algorithms.

Arguments:
None.

Example:
```
      navigationModel.getModelExtents = function () {
        const boundingBox = model.geometry.boundingBox;
             return [boundingBox.min.x, boundingBox.min.y, boundingBox.min.z,
             boundingBox.max.x, boundingBox.max.y, boundingBox.max.z];
      };
```

3.11.15 Property ‘model.floorPlane’

Getter Method `getFloorPlane()`

Method called by the Navigation Library when it needs to know the location of the floor plane for example in the ‘Walk’ motion model. This allows the motion model to keep the view point a fixed distance above a plane as well as the ability to automatically walk up stairs.

```
getFloorPlane()
```

Implementation:
Required for the walk motion model algorithms.
Introduced in 3DxWare 10 version 10.8.12.

Arguments:
None.

Example:
```
      navigationModel.getFloorPlane = function () {

        // a point on the floor plane
        const point = new THREE.Vector3(0., 1., 0.);
        point.applyMatrix4(grid.matrixWorld);

        // In this sample the up-axis is the y-axis
        const yAxis = new THREE.Vector3();
        grid.matrixWorld.extractBasis(new THREE.Vector3()
             , yAxis
             , new THREE.Vector3());
        const d0 = yAxis.dot(point);
        // return the plane equation as an array
        return [yAxis.x, yAxis.y, yAxis.z, -d0];
      };
```

Note:
If there is no floor plane return null.

3.11.16 Property ‘model.unitsToMeters’

Getter Method `getUnitsToMeters()`

Method called by the Navigation Library when it requires the conversion factor to calculate the height above the floor in walk mode as well as the speed in the first-person motion model. The Navigation Library assumes that this value does not change, and it is only queried once. The value specifies the length of the world/model units in meters.

`getUnitsToMeters()`

Implementation:
Required for the first-person motion model algorithms.
Introduced in 3DxWare 10 version 10.8.12.

Arguments:
None.

Example:
```
      navigationModel.getUnitsToMeters = function () {
        // 1 unit is 10cm
        return 0.1;
      };
```

3.11.17 Property 'pivot.position'

Getter Method `getPivotPosition()`
Method called by the Navigation Library when it wants to know the position of the 2D mouse pivot or of a pivot manually set by the user. The position is returned as a 1x3 array in world coordinates.

`getPivotPosition()`

Implementation:
Required for rotating about the pivot position

Arguments:
None.

Setter Method `setPivotPosition()`
Method called by the Navigation Library to set the position of the pivot. The pivot position is in world coordinates.

`setPivotPosition(position)`

Implementation:
Required for changing the pivot position.

Arguments:
`position`
  position of the 3D mouse pivot in world coordinates

3.11.18 Property 'pivot.visible'

Setter Method `setPivotVisible()`
Method called by the Navigation Library when it wants to set the visibility of the pivot used for the 3D mouse. This will be dependent on the user setting for the pivot visibility in the 3Dconnexion 3D mouse settings panel and whether the Navigation Library is actively navigating the scene.

`setPivotVisible(visible)`

Implementation:

Required for changing the 3D mouse pivot visibility.

Arguments:
`visible`
    true if the pivot should be visible, otherwise false

3.11.19 Property 'pointer.position'

Getter Method `getPointerPosition()`
Method called by the Navigation Library when it requires the world position of the mouse pointer on the projection/near plane.

```
getPointerPosition()
```

Implementation:
Required for the quick zoom algorithms.

Arguments:
None

Example:

```
navigation.getPointerPosition = function () {

  const rect = canvas.getBoundingClientRect();

  // position of the mouse in the canvas (windows [0,0] is at the top-
  // left of the screen, opengl[0,0] is at the bottom-left)
  // the position is tracked relative to the window so we need to
  // subtract the relative position of the canvas
  // Setting z=0 puts the mouse on the near plane.
  const pos_opengl = new THREE.Vector3(window.mouseX - rect.left,
  gl.viewportHeight - (window.mouseY - rect.top), 0.0);
  if (TRACE_MESSAGES)
       console.log('Mouse Position =[' + pos_opengl.x + ', ' +
       pos_opengl.y + ', ' + pos_opengl.z + ']');

  // three.js has screen coordinates that are (-1,-1) bottom left and
  // (1,1) top right.
  const pos = new THREE.Vector3(pos_opengl.x / gl.viewportWidth *
       2.0 - 1.0, pos_opengl.y / gl.viewportHeight * 2.0 - 1.0
       , pos_opengl.z * 2.0 - 1.);

  // make sure the matrices are up to date
  camera.updateProjectionMatrix();
  camera.updateMatrixWorld();

  pos.unproject(camera);
  return pos.toArray();
};
```

3.11.20 Frame Properties

3.11.20.1 Property 'transaction'

Setter Method `setTransaction()`
Method called by the Navigation Library at the beginning and end of a frame change. At the beginning of a frame change the Navigation Library will set the 'transaction' property to a value >0. When the Navigation Library has completed the changes to the frame, the value is

reset to 0. Clients that do not use an animation loop and need to actively refresh the view can trigger the update when the value is set to 0.

`setTransaction(transaction)`

Implementation:
Required to refresh the view

Arguments:
`transaction`
    The current transaction

Example:
```
      navigationModel.setTransaction = function (transaction) {
        if (transaction === 0) {
          // request a redraw
          updateScene();
        }
      };
```

3.11.20.2 Property 'frame.timingSource'

By setting the 'frame.timingSource' property to 1, the client application informs the Navigation Library that the client has an animation loop and will be the source of the frame timing. (See the 'frame.time' property below).

Example:
```
      navigationModel.on3dmouseCreated = () => {
        // set ourselves as the timing source for the animation frames
        spaceMouse.update3dcontroller({
          frame: { timingSource: 1 }
        });
      };
```

3.11.20.3 Property 'frame.time'

When the 'frame.timingSource' property is set to 1, the client initiates a frame transaction by informing the Navigation Library of the new frame time. The frame time type is double and is the time stamp of the frame.

Example:
```
      // the callback that results in the scene being rendered
      navigationModel.render = function (now) {
        if (animating) {
          // Initiate a new frame transaction by updating the controller
          // with the frame time.
          spaceMouse.update3dcontroller({
            frame: { time: now }
          });
          // Request an animation frame for the next incoming transaction
          // data.
          window.requestAnimationFrame(render);
        }

        // Render the current scene.
        renderer.render(scene, camera);
      };
```

3.11.21 Hit-testing Properties

The Navigation Library includes navigation algorithms that are dependent on the size of and the distance to the model being viewed. This can be at the view centre or at the position of the mouse cursor. When these algorithms are activated, the Navigation Library may require the client to perform hit-testing. In this case the Navigation Library will set up a ray and query the nearest hit.

3.11.21.1 Property 'hit.lookfrom'

Setter Method `setLookFrom()`
Method called by the Navigation Library that defines the origin of the ray used in hit-testing.

`setLookFrom(origin)`

Implementation:
Required for the auto pivot algorithms.

Arguments:
`origin`
The position of the origin of the ray in world coordinates.

Example:
```
      navigationModel.setLookFrom = function (data) {
        ray.origin = data;
      };
```

3.11.21.2 Property 'hit.direction'

Setter Method `setLookDirection()`
Method called by the Navigation Library that defines the direction of the ray used in hit-testing.

`setLookDirection(direction)`

Implementation:
Required for the auto pivot algorithms.

Arguments:
`direction`
The direction of the ray in world coordinates.

Example:
```
      navigationModel.setLookDirection = function (data) {
        ray.direction = data;
      };
```

3.11.21.3 Property 'hit.aperture'

Setter Method `setLookAperture()`
Method called by the Navigation Library that defines the diameter of the ray used in hit-testing.

`setLookAperture(aperture)`

Implementation:
Required for the auto pivot algorithms.

Arguments:
`aperture`
The diameter of the ray in world coordinates.
Example:
```
      navigationModel.setLookAperture = function (data) {
        ray.aperture = data;
      };
```

3.11.21.4 Property 'hit.selectionOnly'

Setter Method `setSelectionOnly()`

Method called by the Navigation Library that defines whether hit-testing should include all the objects or be limited to the current selection set.

```
setSelectionOnly(selection)
```

Implementation:
Required for the auto pivot algorithms.

Arguments:
`selection`

If hit-tetsing should only be performed on the current selection set then this value is true, otherwise false.

Example:

```
navigationModel.setSelectionOnly = function (data) {
  navigationModel.hittest.selectionOnly = data;
};
```

3.11.21.5 Property 'hit.lookat'

Getter Method `getLookAt()`

Method called by the Navigation Library when it requires the world position of the nearest intersection of the "look" ray with the scene or selection set.

```
getLookAt()
```

Implementation:
Required for the auto pivot algorithms.

Arguments:
None.

Example:

```
navigationModel.getLookAt = function () {
  // Create a raycaster
  const raycaster = new THREE.Raycaster(ray.origin,
          ray.direction, gl.frustumNear,
          gl.frustumFar);
  raycaster.precision = ray.aperture /2.;
  raycaster.linePrecision = ray.aperture / 2.;

  // do the hit-testing
  var intersects = raycaster.intersectObjects(scene.children);
  if (intersects.length > 0) {
       const lookAt = new THREE.Vector3();
       lookAt.copy(ray.direction);
       lookAt.multiplyScalar(intersects[0].distance);
       lookAt.add(ray.origin);
       if (TRACE_MESSAGES)
          console.log('lookAt=[' + lookAt.x + ', ' + lookAt.y + ', ' +
          lookAt.z + ']');
       return lookAt.toArray();
  }
  // If nothing was hit return nothing
  return null;
};
```

3.11.22 Selection Properties

Dependent on the user's action and settings, the Navigation Library pivot and zoom algorithms will be modified if a subset of the model in the scene is selected. The Navigation Library will query if the current selection set is empty and if not then request the extents of the selection set.

3.11.22.1 Property 'selection.empty'

Getter Method `getSelectionEmpty()`
Method called by the Navigation Library to determine if the current selection set is empty. The method returns true if the selection is empty.

```
getSelectionEmpty()
```

Implementation:
Required for the auto pivot algorithms and zoom algorithms

Arguments:
None.

3.11.22.2 Property 'selection.extents'

Getter Method getSelectionExtents()
Method called by the Navigation Library when it needs to know the bounding box of the selection set. One example of when this will happen is when the library needs to zoom the view to the extents of the selection. The extents are returned as an array containing the min and max values of the bounding box in world coordinates.

```
getSelectionExtents()
```

Implementation:
Required for the auto pivot algorithms and zoom algorithms

Arguments:
None.

3.11.23 Command Properties

Application commands are exposed to the 3Dconnexion property panel using the commands properties. These can be assigned to buttons on the 3D Mouse by the user to create custom 3D Mouse configurations for when the web page is in focus.

3.11.23.1 Property 'commands.activeCommand'

Setter Method `setActiveCommand()`
Method called by the Navigation Library when the user presses a 3D Mouse button that has been assigned an application command.

```
setActiveCommand(commandId)
```

Implementation:
Required for application commands

Arguments:
`commandId`
This is an application define string used to identify the command. See the `_3Dconnexion.ActionTree` class for further details.

Example:
```
navigationModel.setActiveCommand = function (commandId) {
  if (commandId)
```

```
    console.log("Id of command to execute= " + commandId);
};
```

3.11.23.2 Property ‘commands.tree’

The client exposes commands to the 3Dconnexion property panel by updating the 3Dmouse controller’s ‘commands.tree’ property with a `_3Dconnexion.ActionTree` instance. See the _3Dconnexion.ActionTree class for further details.

Example:

```
spaceMouse.update3dcontroller({
    commands: {
      tree: actionTree
    }
});
```

3.11.23.3 Property ‘commands.activeSet’

The commands exposed to the 3Dconnexion property panel may contain more than one action set/button bank. This allows context sensitive button configurations. The client application uses this property to set which action set is active. If the property is not set, the 3Dconnexion driver will use the action set which is defined as default in the profile for the application.

Example:

```
spaceMouse.update3dcontroller({
    commands: {
      activeSet: 'ACTION_SET_ID'
    }
});
```

3.11.23.4 Property ‘images’

Images can be associated with commands. These are exposed to the 3Dconnexion UI elements by updating the ‘images’ property with an array of `_3Dconnexion.ImageItem` instances. See the `_3Dconnexion.ImageCache` class for details.

Example:

```
spaceMouse.update3dcontroller({
    images: actionImages.images
});
```