A 3D VIEWER FOR COMPLEX HIERARCHICAL PRODUCT MODELS

hom3r API Documentation

(english version)

DOCUMENT IDENTIFIER:

Document	hom3r API
Status	Final
Editors	Daniel González-Toledo (UMA); María Cuevas-Rodríguez (UMA); Carlos Garre del Olmo (UMA); Luis Molina-Tanco (UMA); Arcadio Reyes-Lecuona (UMA)
Document description	**This document describes the hom3r 3D viewer. The document describes in detail the viewer with its API and its integration.**

REVISION TABLE:

Version	Date	Modified pages	Modified Sections	Comments
1.0	18/11/2016	-	-	First version of the hom3r API Documentation

TABLE OF CONTENTS

1 3D Viewer (hom3r)

HOM3R (Hierarchical prOduct Model 3D vieweR) is a 3D viewer designed to be integrated as a module in web applications. The goal of this module is to generate an interactive 3D render of a product model, allowing the user to navigate through its geometry while showing information coming from the web application.

Hom3r is also responsible of loading the geometric model of the product when requested by the application and administrates the product hierarchy. Hom3r provides a user interface, as seen in Fig 1, giving access to all implemented functionalities. Hom3r receives events from the interface as well as mouse and keyboard events to control navigation.

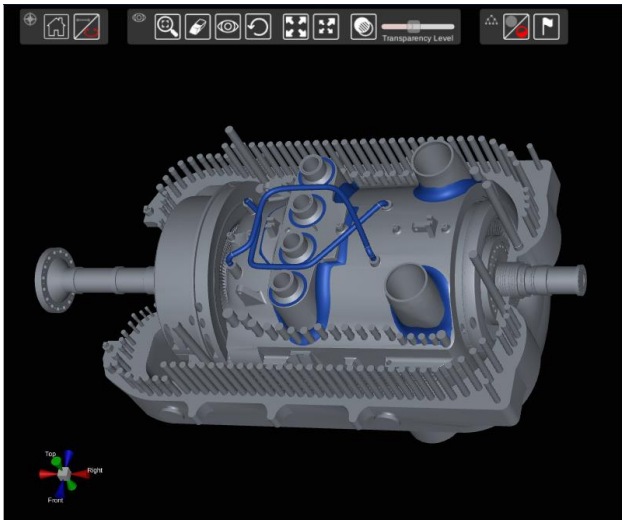

Fig 1. 3D viewer (hom3r)

1.1 Integrating hom3r in a web application

Hom3r consists of a 3D canvas and a Javascript API. The 3D canvas has been implemented using the videogame development platform Unity 3D, version 5 Pro, and built as WebGL for its integration in web applications. The JavaScript API is responsible of: (1) loading the 3D Canvas and (2) implementing the interface which allows information exchange between the 3D Canvas and the web application. This API consists of a set of JavaScript files with all interfaces and implementations.

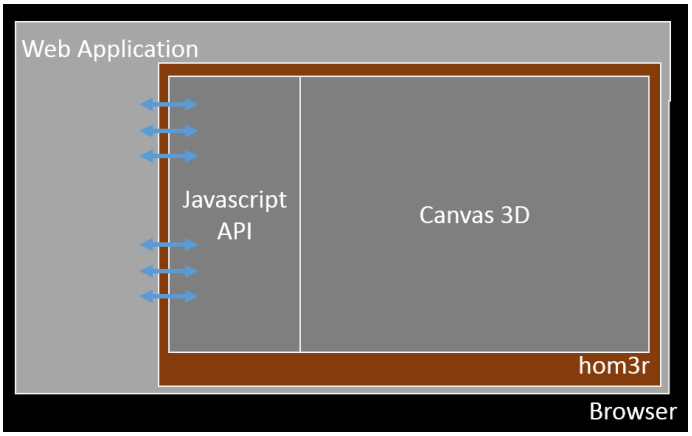

Fig 2. Block diagram of the developed system and its context

The functions for loading the 3D canvas are based in the templates and examples provided by Unity 3D. The API of hom3r, which will be described in the next section, has been implemented as a JavaScript container. The container is similar to a class where all needed methods and attributes have been implemented. These methods allow communication with the 3D canvas, adapting data types when needed. The API allows the web application to register a callback function through which the different events launched during 3D interaction will be reported to the web application.

To make use of the API, the script *Hom3rAPI.js* should be included by the web application through the appropriate HTML code. From that moment on, the web application can access the methods provided by the API by using the prefix "*Hom3rAPI*". For example, to register the function *MyCallbackFunction*, the web application should call: *Hom3rAPI.RegisterCallBack (MyCallBackFunction).*

Besides the script containing the API, the web application should integrate as well the rest of scripts containing the code allowing the load of the 3D canvas. This is the list of required scripts:

- **Hom3rAPI.js**: Contains the JavaScript API allowing communication with the 3D Canvas.
- **UnityConfigModule.js**: Configures the parameters needed for loading the 3D Canvas, including: (1) the location of the files containing the code of the hom3r and (2) the amount of memory reserved for execution; this parameter should be adjusted depending on the weight of the 3D models. This script is also responsible of checking potential compatibility issues with the browser and of exception handling when needed.
- **UnityProgress.js**: Contains the code for showing the splash screen and the progress bar while loading the 3D Canvas (WebGL canvas generated by Unity). This script can be adapted to customize the splash screen appearance.
- **UnityLoader.js**: Contains the Unity WebGL loader.

The steps for integrating hom3r in any web application are:

1. Include the JavaScript files in the HMTL code:

```
<script src="~/Hom3r/Hom3rAPI.js"></script>
<script src="~/Hom3r/UnityConfigModule.js"></script>
<script src="~/Hom3r/ProgressBar/UnityProgress.js"></script>
<script src="~/Hom3r/Release/UnityLoader.js"></script>
```

2. Add the 3D Canvas in the place of your HTML code where you want it to appear:

```
<div id="hom3rwebgl">
    <canvas class="emscripten" id="canvas"
oncontextmenu="event.preventDefault()" ></canvas>
</div>
```

The following is an example of the HTML code of a simple web page including hom3r:

```
<!doctype html>

<html lang="en-us">
<head>
<meta charset="utf-8">
<meta http-equiv="Content-Type" content="text/html; charset=utf-8">
<title>hom3r example</title>
</head>
<body>
<h3> home3r example</h3>
<div id="hom3rwebgl">
<canvas class="emscripten" id="canvas" oncontextmenu="event.preventDefault()"
height="700px" width="900px"></canvas>
</div>
<script src="Hom3rAPI.js"></script>
<script src="UnityConfigModule.js"></script>
<script src="ProgressBar/UnityProgress.js"></script>
<script src="Release/UnityLoader.js"></script>
</body>
</html>
```

1.2 hom3r API

This section defines the interface (API) that hom3r provides to web applications, with the goal of providing integration of the 3D Canvas with the web application and allowing information exchange between both. The API methods can be grouped in: (1) methods used by the web application to send information to hom3r (input interface) and (2) methods used by the web application to receive information from hom3r (callbacks), as seen in the following figure:

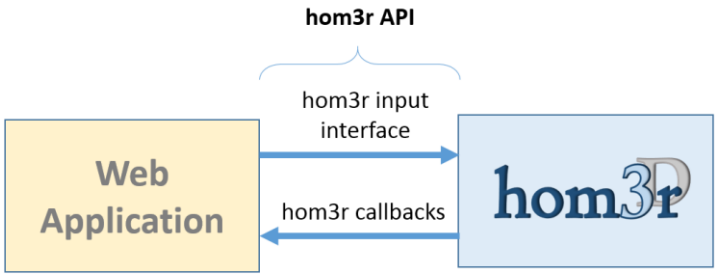

Fig 3. hom3r API

Input interface methods

The interface provides the methods shown in the diagram of Fig 4.

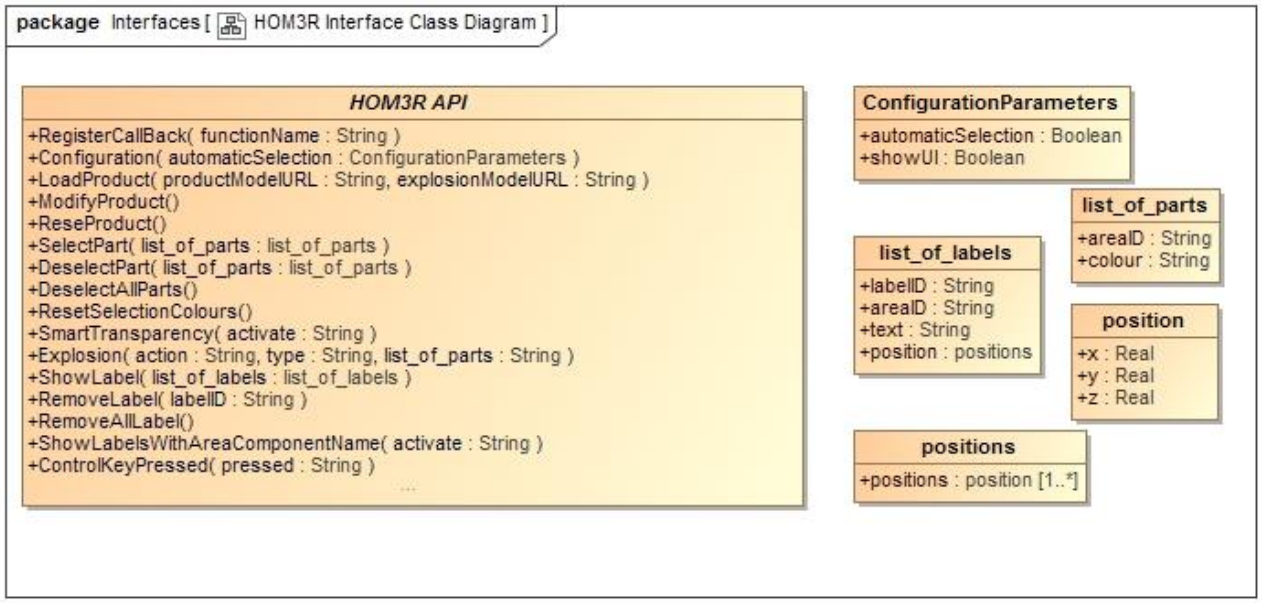

Fig 4. Methods in the interface between hom3r and the web application

A detailed description of each method follows, with two paragraphs for each method: (1) description of functionality and (2) arguments used by the method.

RegisterCallBack (functionName)

Description: Registers the callback for the web application to receive information coming from the 3D Canvas. The web application should define a function and register it with this interface. From that moment on, the 3D Canvas will send its information to the web application through the registered callback. The callback can be registered at any moment, but it is usually convenient to register it at the beginning of the execution of the web DOM. The next section, *Callbacks*, will provide deeper insight on how the callback should be implemented.

Arguments: *functionName* is a string with the name of the callback.

Configuration (config) * Not yet implemented

Description: Setups some parameters of the system. This method can be called at any moment. While the method is not called, a default configuration will be used.

Arguments: *config* is an object of class *ConfigurationParameters* containing the value of each configuration parameter. The system will use the default configuration for each parameter left blank.

The available configuration parameters are:

Parameter	Type	Description
automaticSelection	Boolean	When the user clicks with the mouse over a product part:

		• If *automaticSelection* is *true*, the 3D Canvas will show the part highlighted as selected and will report to the web application the part that has been selected. • If *automaticSelection* is *false*, the 3D Canvas will report only that the user might want to select the part. The web application will be responsible of actual selection through the appropriate command. Default value: true
showUI	Boolean	Parameter to show the user interface in hom3r. The interface will show or hide completely, not partially.

LoadProduct (productModelURL, explosionModelURL)

Description: Starts load of a product model, consisting of information such as geometry and disassembly for exploded views. This method can be called at any moment. If called more than once, only the first call will be attended and the rest ignored. It is the web application who is responsible of the validity of the model data. If the first URL passed as argument (*productModelURL*) is not valid, the model will be discarded. If the second URL (*explosionModelURL*) is not valid, the product will be loaded but exploded views will not be available. When the load of the product ends, the 3D Canvas will report the result through the registered callback.

```
productModelURL = "/Home/ProductModelHom3r";
explosionModelURL = "/Content/turbineexplosion.xml";
Hom3rAPI.LoadProduct(productModelURL, explosionModelURL);
```

Arguments: *productModelURL* is a string with the URL from which the system can download the Json file containing all data of the product model.

explosionModelURL is a string with the URL from which the system can download the XML file containing the additional data for the exploded view model of the product.

ModifyProduct (urlJson) * Not yet implemented

Description: Tells hom3r to update the product model currently loaded. Hom3r will modify the appropriate information from the loaded model with the new values contained in the new Json. This method can be called at any moment. It is the web application who is responsible of the validity of the model data. This method allows modification of existing parts, but do not allows adding new parts to the model. If the URL passed in argument *jsonUrl* is not valid, this method will have no effect. The system will report the result to the web application through the registered callback.

Arguments: *urlJson* is a string with the URL from which the system can download the Json file with the updated product model information.

ResetProduct () * Not yet implemented

Description: Tells the hom3r to unload the current product model. After calling this method, the 3D Canvas will not render any 3D geometry until a new model is loaded. The system will report the result to the web application through the registered callback.

SelectPart (list_of_parts)

Description: Sets the *selected* state (also known as *confirmed* state) for the parts of the product

referenced in the argument *list_of_parts*. This method can be called at any moment. The referenced parts can be at any level of the product tree, i.e. they can be either nodes, leaves or areas. If one of the newly *selected* parts has children (if it is a node or a leaf), all children will be recursively *selected* as well.

Arguments: *list_of_parts* is an array of parts to select, where each part to select contains: (1) *areaID,* string with the identifier of the part to be selected and (2) *colour,* string with the hexadecimal representation of the color* used for highlighting the selection of the part (for example, #ff0000).

***** If no color is specified, the system will use the default selection color for that part. Once a color is specified for one part, that color will be the default selection color for that part. If a part has no default selection color, a system default will be used. The default colors for all parts can be reset through the method *ResetSelectionColours*.

```
var newSelectedNodes = [];
newSelectedNodes.push({"areaID": partID, "colour": "#ff0000"});
…
Hom3rAPI.SelectPart(newSelectedNodes);
```

DeselectPart (list_of_parts)

Description: Sets the *not selected* state (also known as *not confirmed* state) for the parts of the product referenced in the argument *list_of_parts*. This method can be called at any moment. The referenced parts can be at any level of the product tree, i.e. they can be either nodes, leaves or areas. If one of the newly *not selected* parts has children (if it is a node or a leaf), all children will be recursively *not selected* as well.

Arguments: *list_of_parts* is an array of parts to unselect, where each part to select contains: (1) *areaID,* string with the identifier of the part to be selected and (2) *colour,* blank string that will be ignored (left for compatibility).

```
var DeSelectNodes = [];
DeSelectNodes.push({"areaID": partID, "colour": ""});
…
Hom3rAPI.SelectPart(DeSelectNodes);
```

DeselectAllParts ()

Description: Sets the *not selected / not confirmed* state for all parts of the product.

ResetSelectionColours()

Description: Sets the default selection color for all parts to the system default selection color.

SmartTransparency (activate)

Description: Switches on/off the smart transparency occlusion handling technique, depending on the value of the *activate* argument. This method can be called at any moment. If the value of *activate* is not valid, this method will have no effect.

Arguments: *activate* is a string indicating the new state for smart transparency. Valid values are: “true” (switches on smart transparency) or “false” (switches off smart transparency).

Explosion (action, type, list_of_parts) * Not yet implemented

Description: Starts the animation of an exploded view for occlusion handling. Exploded view animations may consist in explosion (disassembly of parts) or implosion (assembly back to collapsed or rest state of the product), depending on the value of the *action* argument. This method can be called at any time. If no parts are currently exploded, a call to this method with an *implode action* will have no effect. There are two types of exploded views: (1) *global*, where all parts are disassembled (exploded) and (2) *local*, where only a selection of parts is disassembled (exploded). A local explosion may imply disassembly of parts not selected for explosion in the case that those parts are occluding the disassembly path of other parts selected for explosion.

Arguments: *action* is a string indicating the direction of the exploded view animation. Valid values are: "explode", the parts are disassembled from the collapsed or rest state of the product or "implode", the parts are assembled back into the collapsed or rest state of the product.

type is a string indicating the type of exploded view to generate. Valid values are: "global", all parts of the product are disassembled or "local", only a selection of parts (referenced in the argument *list_of_parts*) is disassembled.

list_of_parts is an array of parts to select for explosion, where each part contains a string with its identifier. This argument is ignored if *type* is set to *global*.

ShowLabel(list_of_labels)

Description: Creates a set of 3D labels attached to product parts. The parts and the text to show on each label are specified in the argument *list_of_labels*. This method can be called at any time.

Arguments: *list_of_labels* is an array of labels, where each label contains: (1) *labelID*, is a string with the identifier of the label (2) *areaID*, is a string with the identifier of the product part to which the label is attached (3) *text*, is the text to show in the label and (4) *position*, is the (x, y, z) position in the local reference system of the product part to which the label will be attached. If position (0, 0, 0) is passed, the system will automatically compute an attachment point that connects the label with the geometric center of the part. Values other than (0, 0, 0) will use as reference the center of the 3D model of the part.

```
var position = [];
position.push({"x": 0, "y": 0, "z": 0});
var newlabels = [];
newlabels.push({ "labelID": labelID, "areaID": areaID, "text": text, "position":
position });

Hom3rAPI.ShowLabel(newlabels);
```

RemoveLabel(labelID)

Description: Removes one previously created 3D label. Once a label is removed, it can only be shown again with a new call to the method *ShowLabel*. This method can be called at any time.

Arguments: *labelID* is a string with the identifier of the label to be removed.

RemoveAllLabel()

Description: Removes all previously created 3D labels. Once a label is removed, it can only be shown again with a new call to the method *ShowLabel*. This method can be called at any time.

ShowLabelsWithAreaComponentName(activate)

Description: Switches on/off the mode "show labels with the name of areas or components" of hom3r. When this mode is switched on, the system will attach a new 3D label to each previously selected part (for example, after calls to method *SelectPart*). The text of the label will be the name of each part. If new parts are selected while in this mode, new 3D labels will be automatically attached. When the argument *activate* is *false*, all 3D labels previously created while in this mode will be deleted and no more 3D labels will be automatically created for newly selected parts.

Arguments: *activate* is a string indicating the new state for this mode. Valid values are: "true" (switches on this mode) or "false" (switches off this mode).

ControlKeyPressed(pressed)

Description: Tells hom3r that the *Control* key has been pressed/unpressed in the keyboard. This method should be called only when the pressed/unpressed state of the key has changed. Due to a limitation of Unity WebGL, hom3r cannot handle the Control key without interfering with the web application. Since hom3r needs this key for multiple parts selection, the web application should report hom3r the state of this key.

Arguments: *pressed* is a string with the pressed/unpressed state of the *Control* key. Valid values are: "true", the key has been pressed and "false ", the key has been unpressed (released).

```
document.onkeydown = function (data) {
    if (data.keyCode === 17 && lastKeyDown !== data.keyCode) {
        Hom3rAPI.ControlKeyPressed("true");
        lastKeyDown = data.keyCode;
    }
};
document.onkeyup = function (_data) {
    Hom3rAPI.ControlKeyPressed("false");
    lastKeyDown = "";
}
```

Callbacks

Communication from hom3r to the web application is achieved through the use of a callback function. This mechanism allows the application to define its own function to handle the events received from hom3r as desired. The application needs to define first the function and then call the hom3r API to register this functions as a callback. The web application decides the name and internal implementation of this function, but the function must have two string input arguments, following this pattern: *function_name(message, value)*. Hom3r tells which event or result wants to send to the web application through the argument *message*, and the value for that message in the argument *value*.

Example of callback use:

```
//Register Callback at the beginning.
Hom3rAPI.RegisterCallBack(MyCallBackFunction);

//Callback.
function MyCallBackFunction(message, value) {
    if (message === "hom3r") {
        if (value === "ok") {
            LoadProduct();                      //User function
        } else if (value === "error")
        {
            console.log(vale);
        }
    }
    if (message === "product") {
        if (value === "ok") {
            console.log("Product model loaded correctly!!!");
        } else if (value === "error") {
            console.log("Error: product model cannot be loaded.");
        }
    }
    if (message === "selectPart") {
        FromHom3r_SelectPartWithID(value);      //User function.
    }
    if (message === "deselectPart") {
        FromHom3r_DeselectPartWithID(value);    //User function
    }
    if (message === "deselectAllParts") {
        FromHom3r_DeselectAllParts();           //User function
    }
    if (message === "RemoveLabel") {
        FromHom3r_RemoveLabel(value);           //User function
    }
}
```

The following table shows the different types of messages sent by hom3r, with their possible values.

Table 1: CallBack function. List of messages and their values.

Message	Description	Values

hom3r	This message is sent when hom3r has been loaded and is waiting for the product model. If the callback is not registered on time, this message may be lost.	Values: *ok / error.*
		Example of function to execute when this message is received (callback implementation). This function loads the Json file containing the product model and the XML file containing the exploded view model:
		`function LoadProduct() {`
		`  var productModelURL = "model/ProductModelHom3r_Saturn5.json";`
		`  var explosionModelURL = "model/Hom3r_Saturn5_Explosion.xml";`
		`  Hom3rAPI.LoadProduct(productModelURL, explosionModelURL);`
		`}`

product	This message is sent when hom3r has finished loading the product model (with or without errors).	Values: *ok / error.*
		Example of callback implementation:
		`if (message === "product") {`
		`  if (value === "ok") {`
		`  console.log("Product model loaded correctly!!!");`
		`  } else if (value === "error") {`
		`  console.log("Error: product model cannot be loaded.");;`
		`  }`
		`}`

selectPart	This message is sent when the user selects one or more parts of the product through 3D interaction.	Values: array containing all product parts that the user has newly selected. Each part contains: (1) *areaID,* string with the identifier of the part and (2) *colour,* blank string (ignored, but kept for compatibility).
		Example of callback implementation:
		`function FromHom3r_SelectPartWithID(list_of_parts) {`
		`  for (var i = 0; i < list_of_parts.data.length; i++) {`
		`  console.log("This part has been selected: " + list_of_parts.data[i].areaID);`
		`  }`
		`}`

deselectPart	This message is sent when the user unselects one or more parts of the product through 3D interaction.	Values: array containing all product parts that the user has newly selected. Each part contains: (1) *areaID,* string with the identifier of the part and (2) *colour,* blank string (ignored, but kept for compatibility).

		Example of callback implementation:
		`function FromHom3r_DeselectPartWithID(list_of_parts) {`
		`  for (var i = 0; i < list_of_parts.data.length; i++) {`
		`  console.log("This part has been de-selected: " + list_of_parts.data[i].areaID);`
		`  }`
		`}`

deselectAllPart	This message is sent when the user unselects all parts of the product through 3D interaction.	Values: none
removeLabel	This message is sent when the user has removed (through 3D interaction) a 3D label that was previously added by the web application through the method *ShowLabel.*	Values: string with the identifier of the 3D label to be removed. The identifier was previously set by the web application in the call to the method *ShowLabel*.

Behaviour of the interface

This section describes the behavior of the system through message exchange in the most typical or main scenarios, showing different sequence diagrams.

Main scenario: initialization and load of models

This scenario describes how the system is loaded inside the web application. The system is loaded through a specific code contained in the set of files of the API. In parallel to the load of the 3D Canvas, the callback should be registered with the API.

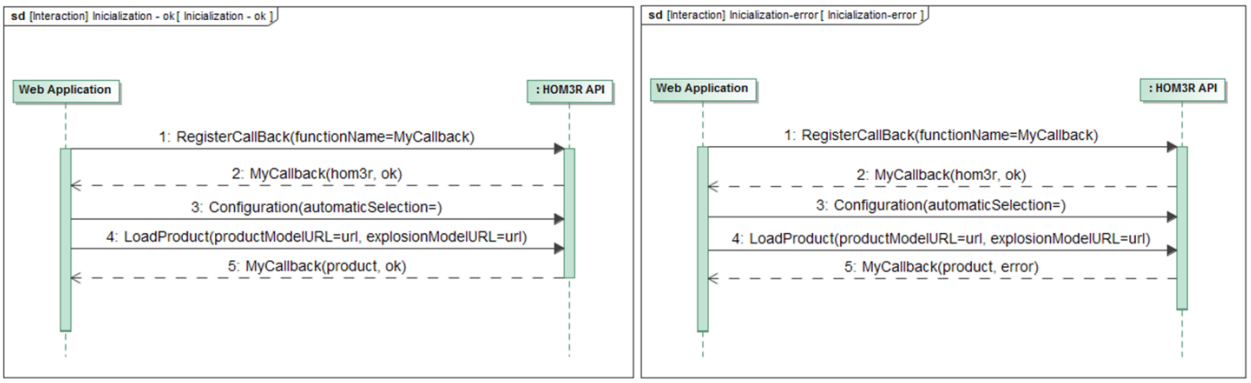

Fig 5. Sequence diagram of the initialization and load of models scenario. (a) Successful initialization. (b) Initialization attempt, with an error in the load of the product model.

If the 3D Canvas is successfully loaded, as seen in Fig 5, hom3r will send a message (message 2 in the figure) to the web application through the registered callback (*MyCallback*) indicating that it is ready and waiting for commands. If an error happens, this message may either not be sent or be sent with an *error* value.

After receiving the *(hom3r, ok)* message, the web application can send the configuration parameters. If these parameters are not sent, the system will work with the default configuration parameters. After (optional) configuration, the web application should tell hom3r to load the product model, including geometry and attached data, with the command *LoadProduct*. This command provides the URLs where hom3r can download the Json and XML files of the model. Hom3r will answer with a *(product, ok)* message if load of the models was successful or *(product, error)* in case of error.

Main scenario: Interaction

If the 3D Canvas and the product model have been loaded successfully, the system will be in the state after Fig 5 (a). From that moment on, hom3r starts the visualization of the geometric model of the product and the user can interact with both the web application and the 3D Canvas of hom3r.

In this scenario, message exchange is bidirectional; the web application sends commands to hom3r using the methods defined in the API and hom3r sends messages to the web application through the registered callback. The previous section described in detail the different messages, with their arguments, that can be sent on each situation. The following diagram shows an example of message exchange between hom3r and the web application during user interaction.

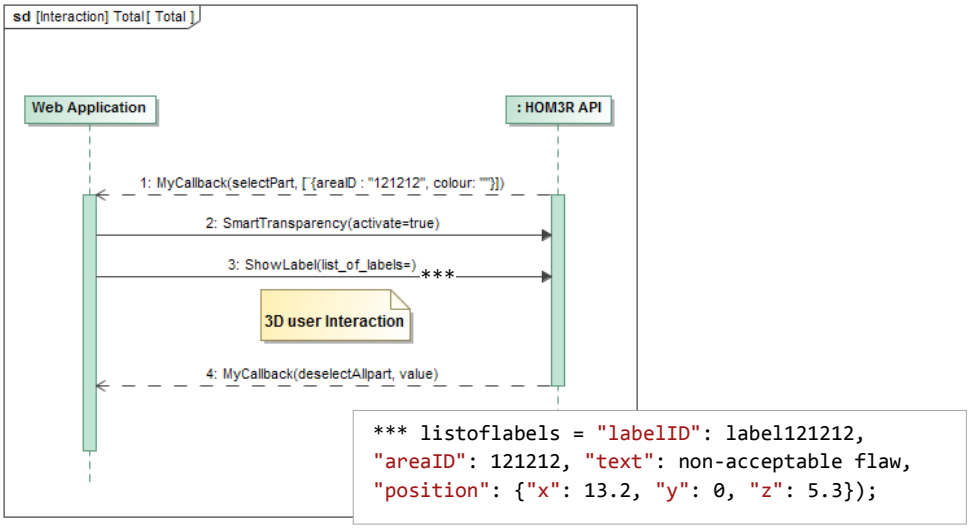

Fig 6. *Sequence diagram of the interaction scenario, showing an example of message exchange between hom3r and the web application*

Fig 6 shows an example of message exchange during user interaction. The sequence of messages could be completely different depending on user interaction and the diagrams shows only a few of the many possible messages.

In this example, the user selects through 3D interaction (using the mouse) a part with identifier *121212.* Hom3r then shows the part highlighted in the 3D Canvas and reports this event to the web application through the registered callback.

The web application then sends a command (*2. SmartTransparency*) to hom3r to activate the smart transparency occlusion handling technique, so that hom3r makes the part visible through occluding parts in the 3D render.

After a while, the web application tells hom3r to attach a label (*3. ShowLabel*) to the selected part, with identifier 121212, in a specific local position with text “non-acceptable flaw”.

To conclude the example, the user clear the selection of parts and hom3r reports the event to the web application through the registered callback.