ptc

vuforia™ studio

Metadata 201

Using JavaScript to Highlight Parts and Create Ionic Popups

Copyright © 2020 PTC Inc. and/or Its Subsidiary Companies. All Rights Reserved.

User and training guides and related documentation from PTC Inc. and its subsidiary companies (collectively "PTC") are subject to the copyright laws of the United States and other countries and are provided under a license agreement that restricts copying, disclosure, and use of such documentation. PTC hereby grants to the licensed software user the right to make copies in printed form of this documentation if provided on software media, but only for internal/personal use and in accordance with the license agreement under which the applicable software is licensed. Any copy made shall include the PTC copyright notice and any other proprietary notice provided by PTC. Training materials may not be copied without the express written consent of PTC. This documentation may not be disclosed, transferred, modified, or reduced to any form, including electronic media, or transmitted or made publicly available by any means without the prior written consent of PTC and no authorization is granted to make copies for such purposes. Information described herein is furnished for general information only, is subject to change without notice, and should not be construed as a warranty or commitment by PTC. PTC assumes no responsibility or liability for any errors or inaccuracies that may appear in this document.

The software described in this document is provided under written license agreement, contains valuable trade secrets and proprietary information, and is protected by the copyright laws of the United States and other countries. It may not be copied or distributed in any form or medium, disclosed to third parties, or used in any manner not provided for in the software licenses agreement except with written prior approval from PTC.

UNAUTHORIZED USE OF SOFTWARE OR ITS DOCUMENTATION CAN RESULT IN CIVIL DAMAGES AND CRIMINAL PROSECUTION.

PTC regards software piracy as the crime it is, and we view offenders accordingly. We do not tolerate the piracy of PTC software products, and we pursue (both civilly and criminally) those who do so using all legal means available, including public and private surveillance resources. As part of these efforts, PTC uses data monitoring and scouring technologies to obtain and transmit data on users of illegal copies of our software. This data collection is not performed on users of legally licensed software from PTC and its authorized distributors. If you are using an illegal copy of our software and do not consent to the collection and transmission of such data (including to the United States), cease using the illegal version, and contact PTC to obtain a legally licensed copy.

Important Copyright, Trademark, Patent, and Licensing Information:
See the About Box, or copyright notice, of your PTC software.

UNITED STATES GOVERNMENT RIGHTS

PTC software products and software documentation are “commercial items” as that term is defined at 48 C.F.

R. 2.101. Pursuant to Federal Acquisition Regulation (FAR) 12.212 (a)-(b) (Computer Software) (MAY 2014) for civilian agencies or the Defense Federal Acquisition Regulation Supplement (DFARS) at 227.7202-1(a) (Policy) and 227.7202-3 (a) (Rights in commercial computer software or commercial computer software documentation) (FEB 2014) for the Department of Defense, PTC software products and software documentation are provided to the U.S. Government under the PTC commercial license agreement. Use, duplication or disclosure by the U.S. Government is subject solely to the terms and conditions set forth in the applicable PTC software license agreement.

PTC Inc., 121 Seaport Blvd, Boston, MA 02210 USA

Prerequisite
Completed:
Metadata 101 – Using Attributes in Creo Illustrate

Intro
In today's world where so many companies are mostly virtual, it can prove at times to be hard to understand a physical product without having that part in front of a person. The sales, marketing, and service organizations of these companies need a way to digitally display these models for product familiarization, which is where AR and Vuforia Studio come in. Using Vuforia Studio, you can create AR representations of your products that are interactive and can display part data when clicked on. This allows you to give interactive product demonstrations and explain service procedures without ever needing the physical part in front of you.

This portion of the project will help you become familiar with the added functionality that JavaScript coding can add to a Vuforia Studio experience regarding populating Ionic popups with information and highlighting parts based on user clicks. It will cover the following topics:

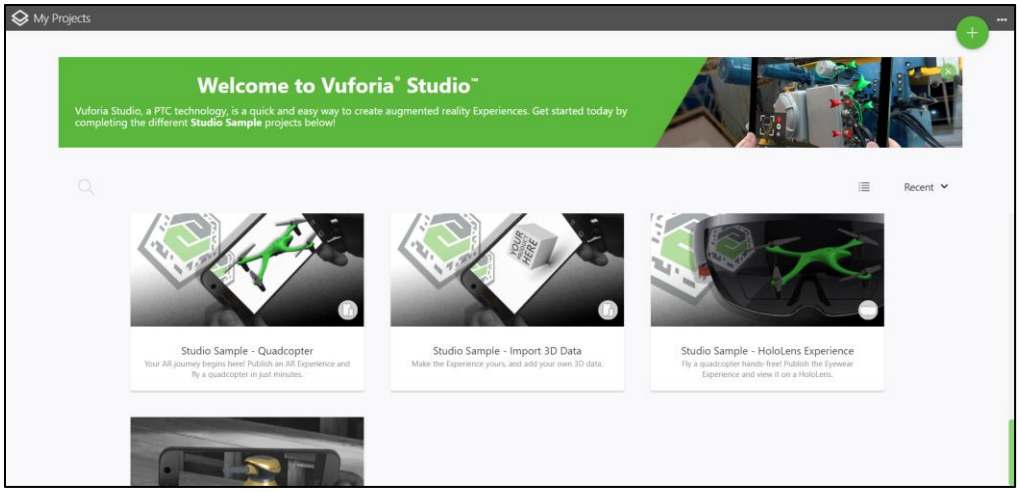
These sections can all be easily accessed using their hyperlinks.

There are also eight appendices at the end of the document for code explanation and copying.

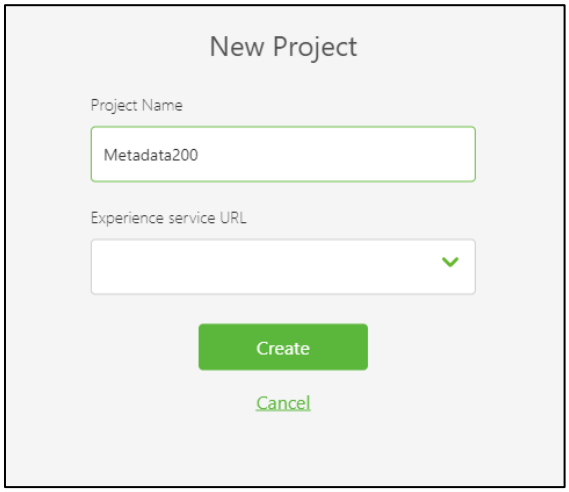
All important notes and UI areas are **Bold**
All non-code text to be typed is *italicized*
All code follows `this convention`
All code comments follow `this convention`

201.1 Set Up the Project

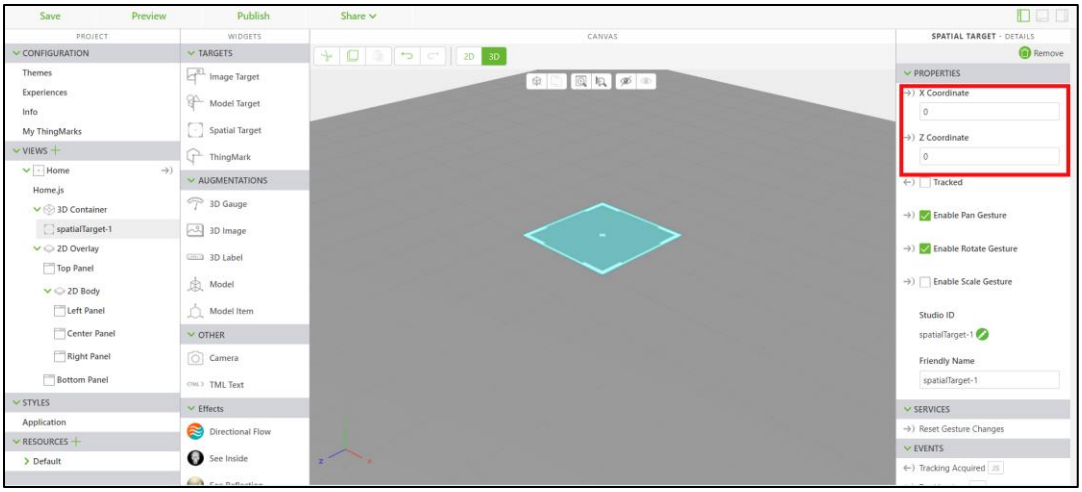
1. Open Vuforia Studio.



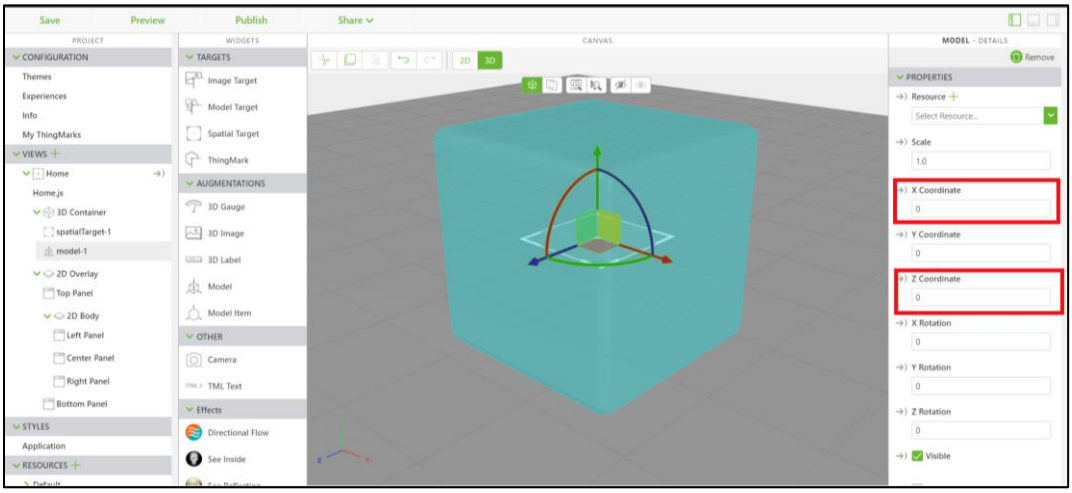
2. If not already configured, click the **…** at the top-right corner of the screen, select **Settings**, and add an **Experience Server** in the **Default Experience Server URL** box.
3. Click the **+** in the top-right corner to create a new project.
4. Select the **Mobile** template.
5. In the **New Project** box, enter *Metadata200* in the **Project Name** field and confirm that the correct **Experience service URL** is being used. This project will be used for both the Metadata 201 and Metadata 202 tutorials.



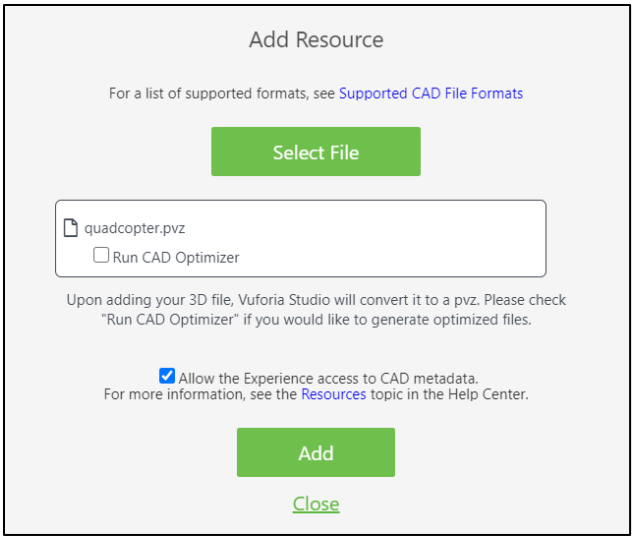
6. Drag and drop a **Spatial Target** onto the 3D canvas. Set the **X Coordinate** and **Z Coordinate** properties to *0***.**



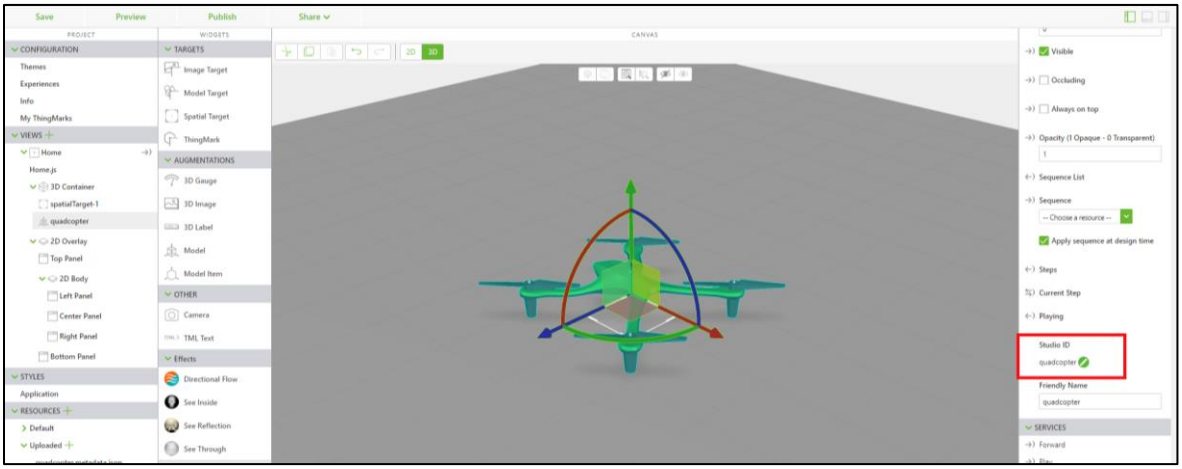
7. Drag and drop a **Model** widget onto the canvas. Set the **X Coordinate** and **Z Coordinate** properties to *0* so that the model widget is in line with the **Spatial Target**. This is important, as a misaligned model can cause issues when viewed as an AR experience.



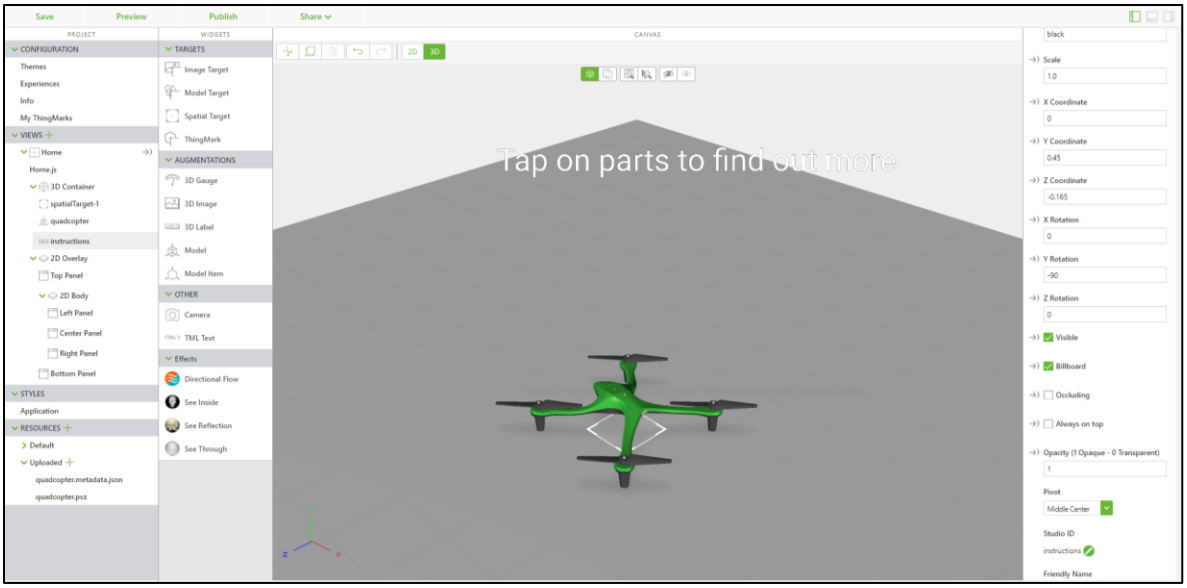
8. In the **Properties** pane for the **Model** widget, select **Resource +** to add a model to the widget. Browse to and select the completed quadcopter.pvz model that was created in Metadata 101. Check the box for **Allow the Experience access to CAD metadata**; this keeps the metadata attributes that are associated with the model from Creo Illustrate during the import process. Leave **Run CAD Optimizer** unchecked, as it will take away its underlying attributes. Click **Add**.



9. With the quadcopter model now visible on the canvas, change the **Studio ID** property to *quadcopter*.



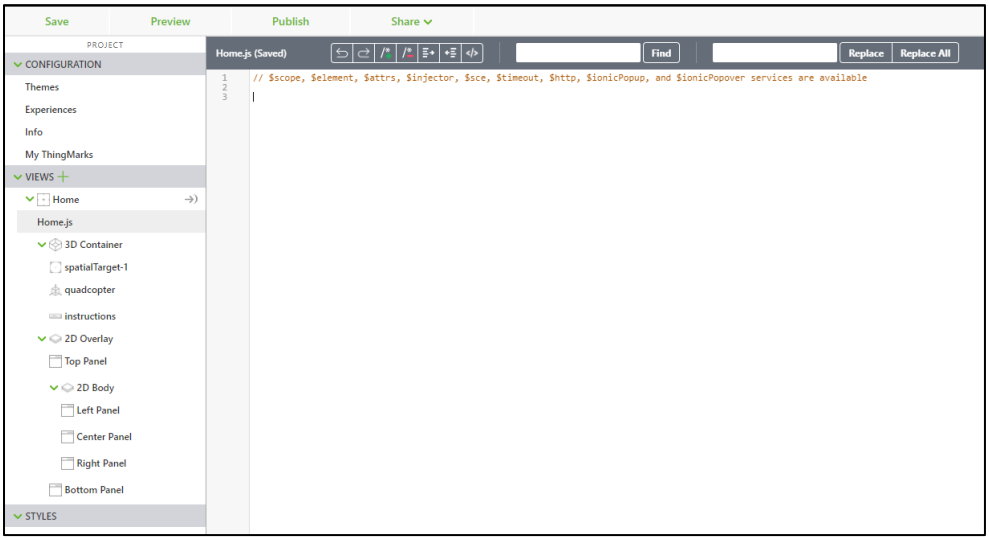
10. Add a **3D Label** widget onto the canvas.
    a. Change the **Text** property to say *Tap on parts to find out more*.
    b. Set the **Font Color** property to *white* and **Font Outline Color** to *black*.
    c. Change the **X, Y,** and **Z Coordinate** properties to *0*, *0.45*, and *-0.165*, respectively.
    d. Set the **X Rotation** property to *0* and **Y Rotation** to *-90*.
    e. Check the box for the **Billboard** property. This property ensures that the label is always flat to the viewing device, regardless of orientation of the viewer.
    f. Change the **Studio ID** for this widget to *Instructions*.



201.2 Userpick Events with an Ionic Popup

In addition to being able to bind widget events to custom JavaScript code in the **Events** section of the Property pane, JavaScript code can be written in the **Home.js** view for more complex custom coding. This section uses the **userpick** event listener function to signal an Ionic Popover to appear when a part of the quadcopter is selected.

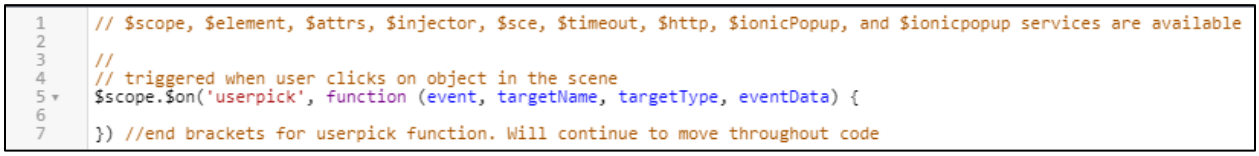
1. Open the **Home.js** view in the **Project** pane. This is where the custom JavaScript code can be written.



2. The first lines of code to be added will be an event listener for when users tap on the model. An event listener is a function that is triggered when a certain event occurs. Vuforia Studio has a 3D object related event called `userpick` that is triggered when a user clicks on a part of the quadcopter in Vuforia View. Additional event listeners are available in Appendix 1, Appendix 2, and Appendix

3. For the arguments in this case, `event` is the act of clicking, `targetName` is referencing the name of the selected model, `targetType` corresponds to the type of widget that is being selected, and `eventData` is a JSON object with the occurrence property, which gives the **Model Tree** location of the selected part.

```
//
// triggered when user clicks on object in the scene
$scope.$on('userpick', function (event, targetName, targetType, eventData) {

}) //end brackets for userpick function. Will continue to move throughout code
```

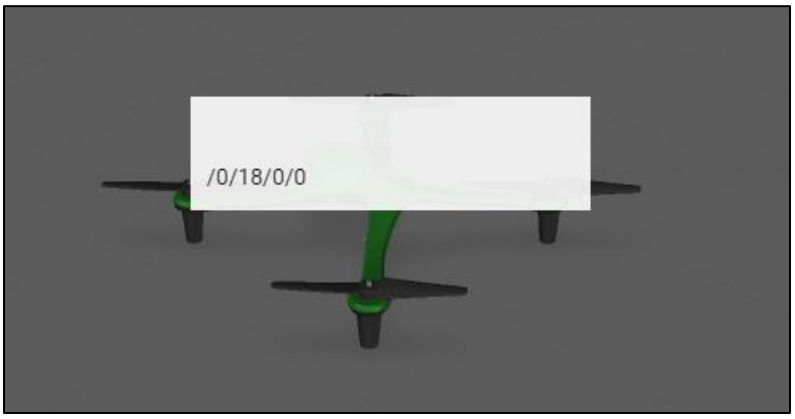


3. Next, inside the `userpick` event listener, add the code for what should happen when the user picks something. The `eventData` argument that is returned is a part specific JSON object. It contains data about the selected target, much like a **Model Item** would if it were added to a model. One of the object properties that is returned is the **occurrence** of the model, set to the `pathId` variable, which gives the Model Tree location of the selected part and is used to differentiate parts from one another. This is how Studio can differentiate between the parts that the user clicks on. The `currentSelection` variable will be created to house information about the part that is selected from its name and `pathId` and will be used throughout the code. In this case, an **Ionic popup** should appear when a part is clicked on. Ionic is a cross-platform SDK for web developers to assist with building applications. An Ionic popup is a dialog box that appears on the screen when a part is clicked. This is used to display part information in the next section.

```
  //
  // variable to pull the value for the occurrence property in the eventData JSON object
from the model. Create variable for the currently selected part
  var pathId = JSON.parse(eventData).occurrence
  $scope.currentSelection = targetName + "-" + pathId

  //
  // adds an ionic popup when a part is clicked. Show the pathId of the selected object
  var popup = $ionicPopup.show({
    template: '<div>' + pathId +  '</div>',
    scope: $scope
  }); //end of ionic popup
```

```
$scope.$on('userpick', function (event, targetName, targetType, eventData) {

  //
  // variable to pull the value for the occurrence property in the eventData JSON objec
  var pathId = JSON.parse(eventData).occurrence
  $scope.currentSelection = targetName + "-" + pathId

  //
  // adds an ionic popup when a part is clicked. Show the pathId of the selected object
  var popup = $ionicPopup.show({
    template: '<div>' + pathId +  '</div>',
    scope: $scope
  }); //end of ionic popup

}) //end brackets for userpick function. Will continue to move throughout code
```

4. **Progress Check:** Click **Save** to update the project and then **Preview** in Studio to open a preview of the experience so far. Click on any part of the quadcopter. A popup should appear in the preview window with the occurrence path of the part. If nothing pops up, double check that the code has been copied correctly.



5. Finish this step by adding in a function that will cause the popup to close on its own after a few seconds, which will come directly after the code for showing the popup

```
  // create a function to close the popup.
  var closePopup = function (popup) {

    //
    // function for returning that the popup will be closed using the .close() method
    return function() {

      //
      //close the popup
      popup.close()

    } // return end

  } // closepopup function end

  //
  //call the $timeout service which will call the function for closing the popup after 3
seconds (3000 ms)
  $timeout(closePopup(popup), 3000);
```

```
17    }); //end of ionic popup
18
19    // create a function to close the popup.
20    var closePopup = function (popup) {
21
22      //
23      // function for returning that the popup will be closed using the .close() method
24      return function() {
25
26        //
27        //close the popup
28        popup.close()
29
30      } // return end
31
32    } // closepopup function end
33
34    //
35    //call the $timeout service which will call the function for closing the popup after 3 seconds (3000 ms)
36    $timeout(closePopup(popup), 3000);
37
38  }) //end brackets for userpick function. Will continue to move throughout code
```

6. Click **Save** and open the **Preview** tab once again. Click on the quadcopter. If a popup appears and then disappears in 3 seconds, then this section has been completed correctly. The completed code for this section is available in Appendix 4.

201.3 Add Attributes from a Model to the Ionic Popup and Highlighting

Once an Ionic popup has been successfully created, you can add the Attribute metadata that was created for the model in Creo Illustrate in the Metadata 101 portion of this project. This is achieved by using the **PTC Metadata API** which is now included in versions 8.5.13 and beyond of Vuforia Studio.

1. Use the **PTC Metadata API** to call the Attributes from the JSON data for the model. This version of the API uses a `.then` method to that will use a callback function to retrieve the data if there is metadata for the model.
    a. Navigate to **Home.js** and add this code directly below the `userpick` function. **Note**: There will be an error message until the end brackets are added two steps from now.

```
  //
  //Look at model and see if it has metadata. If it does, then execute the below code and
create an object called metadata
  PTC.Metadata.fromId(targetName)
              .then  ( (metadata) => {
```

    b. Indent **lines 12-41** by selecting them and clicking the **Indent** button. These lines will be placed inside of the PTC API above, as they are dependent on the promise being successful.

    c. Add the below code to **line 43** to put some bracket ends on the PTC API function. This ensures that there are no errors with unfinished lines.

```
  }) //end brackets for PTC API and .then
```

```
  //
  //catch statement if the promise of having a part with metadata is not met
  .catch( (err) => { console.log('metadata extraction failed with reason : ' +err) })
```

```
  PTC.Metadata.fromId(targetName)
              .then  ( (metadata) => {

    //
    // variable to pull the value for the occurrence property in the eventData JSON object from the model. Create variable for the currently
    var pathId = JSON.parse(eventData).occurrence
    $scope.currentSelection = targetName + "-" + pathId

    //
    // adds an ionic popup when a part is clicked. Show the pathId of the selected object
    var popup = $ionicPopup.show({
      template: '<div>' + pathId +  '</div>',
      scope: $scope
    }); //end of ionic popup

    // create a function to close the popup.
    var closePopup = function (popup) {

      //
      // function for returning that the popup will be closed using the .close() method
      return function() {

        //
        //close the popup
        popup.close()

      } // return end

    } // closepopup function end

    //
    //call the $timeout service which will call the function for closing the popup after 3 seconds (3000 ms)
    $timeout(closePopup(popup), 3000);

  }) //end brackets for PTC API and .then

  //
  //catch statement if the promise of having a part with metadata is not met
  .catch( (err) => { console.log('metadata extraction failed with reason : ' +err) })

}) //end brackets for userpick function. Will continue to move throughout code
```

2. Uncovering the data from the metadata attributes requires the `get` function for the PTC Metadata API. Calling `metadata.get` retrieves all metadata for the model, which in this case is comprised of the attributes of the part that were added in Creo Illustrate. Using the `pathID` variable, which contains the occurrence data of the part, causes the `metadata.get` function to index the data for the selected part. A full list of metadata API functions can be found on the [PTC Support Website] of this document. This function is used to obtain values for the following variables: the display name of the part, the instruction name, and the part number. Add the following code below the `$scope.currentSelection` variable:

```
    //
    // create variables based on attribute names from Creo Illustrate for this model. use metadata.get to obtain the data from the JSON properties for this occurrence.
    var partName        = metadata.get(pathId, 'Display Name');
    var instructionName = metadata.get(pathId, 'illustration');
    var partNumber      = metadata.get(pathId, 'partNumber');
```

```
    $scope.currentSelection = targetName + "-" + pathId

    //
    // create variables based on attribute names from Creo Illustrate fo
    var partName        = metadata.get(pathId, 'Display Name');
    var instructionName = metadata.get(pathId, 'illustration');
    var partNumber      = metadata.get(pathId, 'partNumber');

    //
    // adds an ionic popup when a part is clicked. Show the part number
```

3. Change the definition for the `template` property of the popup. Now that the attribute data has been imported into Studio with `metadata.get`, the Ionic popup should display the `partNumber` and `partName` variable values when clicked on. The comment has also been edited to include more information.

```
    //
    // adds an ionic popup when a part is clicked. Show the part number and name of the
selected object.  </br> adds a line break between the two variables
    var popup = $ionicPopup.show({
      template: '<div>' + partNumber + ' </br>' + partName + '</div>',
      scope: $scope
    }); //end of ionic popup
```

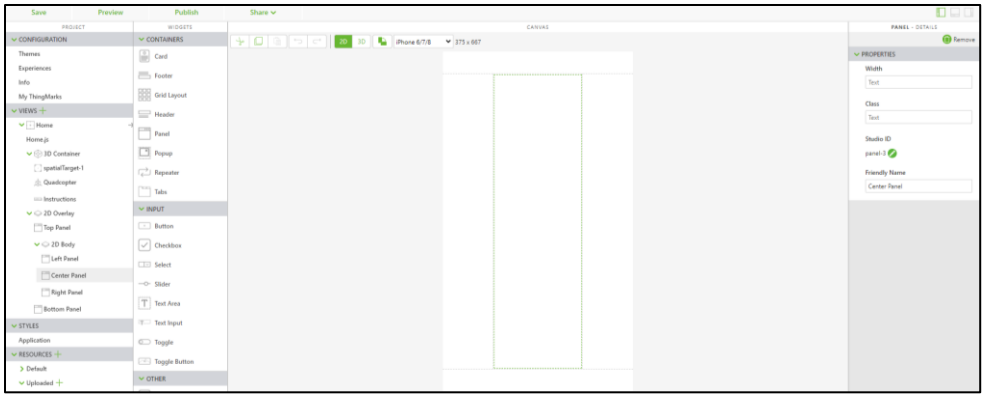
```
21      var partNumber      = metadata.get(pathId, 'partNumber');
22
23      //
24      // adds an ionic popup when a part is clicked. Show the part number and
25      var popup = $ionicPopup.show({
26        template: '<div>' + partNumber + ' </br>' + partName + '</div>',
27        scope: $scope
28      }); //end of ionic popup
```

4. Click **Save** and open **Preview** to verify that the new information was added into the popup. If the image looks like the one below, then this step has been completed correctly. The full code up until this section is provided in Appendix 5.

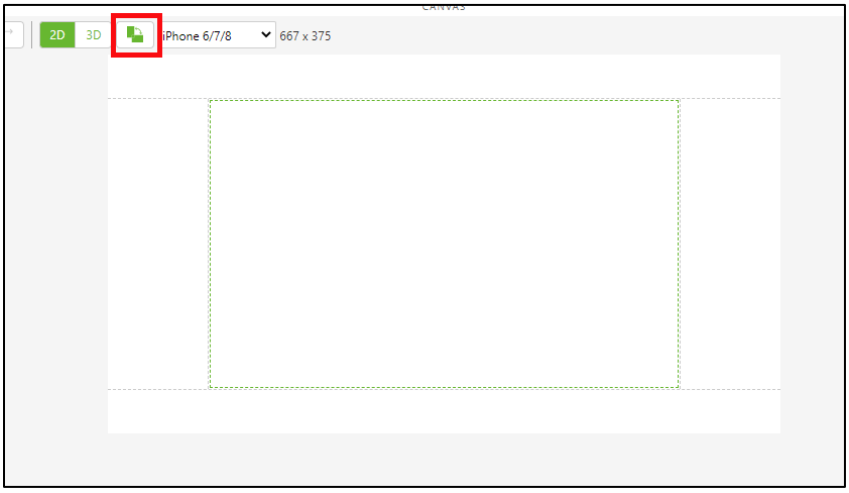
201.4 Bind Play and Reset Buttons

In Metadata 101, illustration sequences were created for the model and the **illustration** attribute was added to parts associated with the sequence. This attribute will be used to tell Studio which disassembly sequence to display on the model. In this experience, if a selected part has an associated illustration sequence with it, Studio will gather that **illustration** attribute value and you will be given the option to play the illustration sequence. There will also be a **Reset** button added to the experience to reset the model at any given time. These buttons will be added to the 2D canvas of the experience.

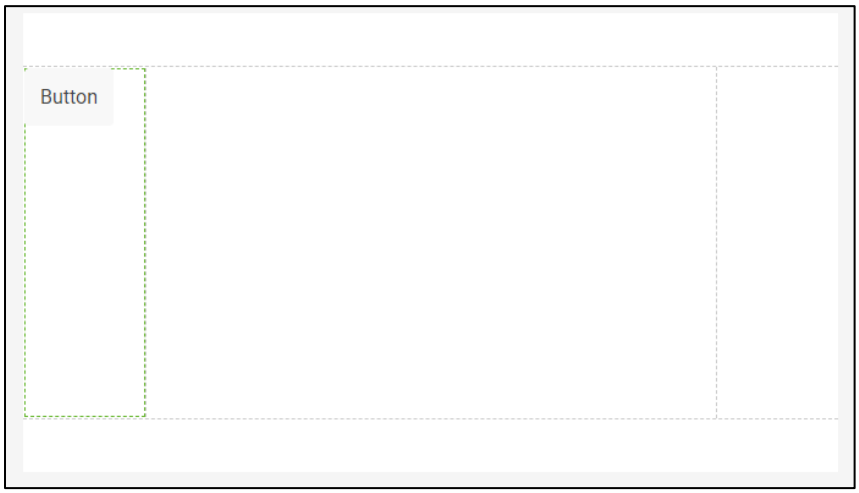
1. Navigate to the **Home** view and open the **2D** canvas.



2. Flip the orientation of the 2D canvas. It is suggested that you hold the phone or mobile device in a landscape orientation when using this experience.

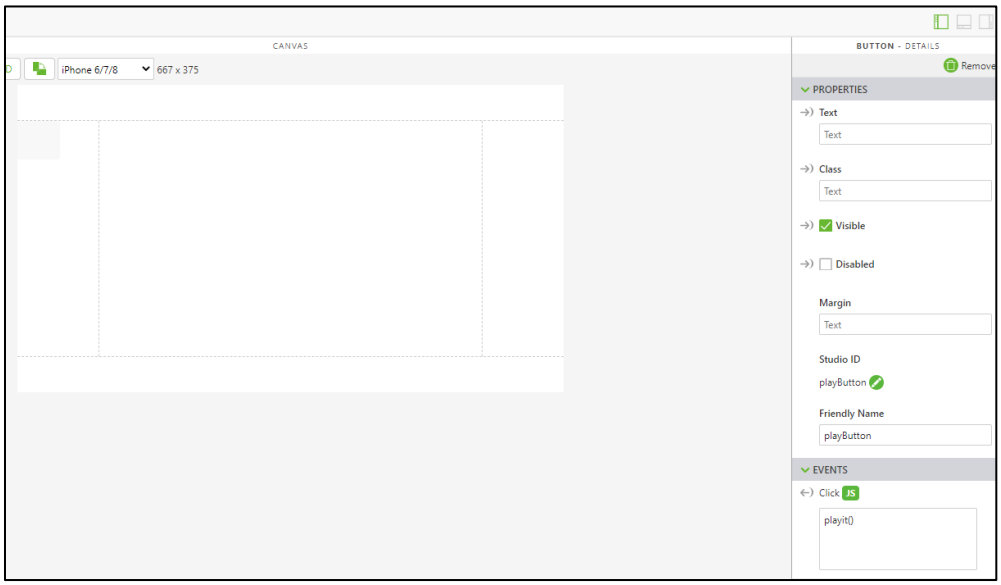


3. Add a **Button** widget to the **Left Panel** of the canvas.

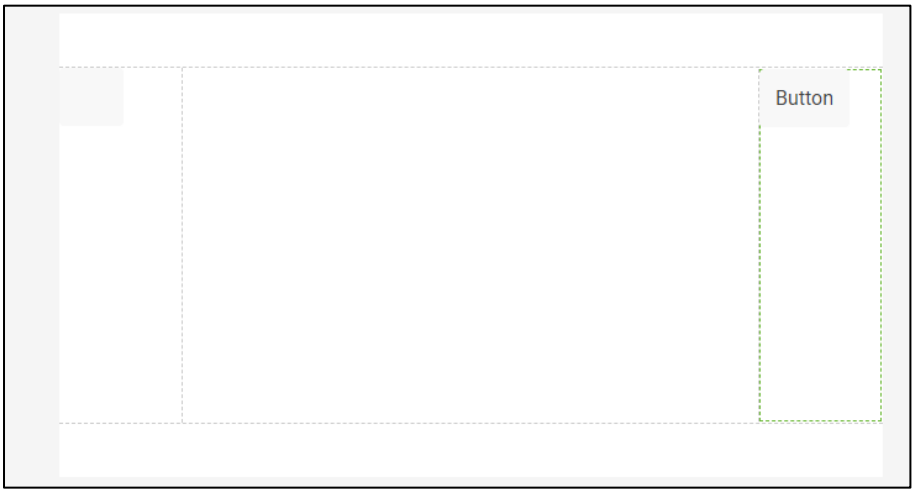


   a. Remove the text from the **Text** property so that the button is blank.
   b. Change the **Studio ID** of the button to *playButton***.**

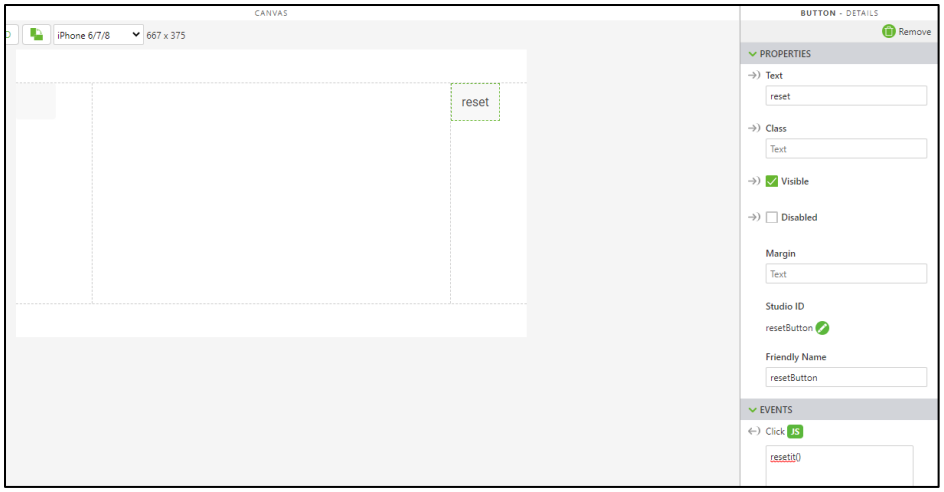
c. Open the **JS** dialog box for the **Click** event and enter *playit()*. This will bind the button to a function in the **Home.js** section that will be created in a later step.



4. Add another **Button** widget onto the **Right Panel** of the canvas.



a. In the **Text** property, change the value to *Reset***.**
b. Change the **Studio ID** of the button to be *resetButton***.**
c. In the **JS** box for the **Click** event, enter *resetit()*. This will bind the button to a function in the **Home.js** section that will be created in a later step.



5. Earlier, you created an Ionic popup to display the name and part number of a given part. Another attribute that some parts will have is the **illustration** attribute that was added in Metadata 101. This attribute contains the name of an animated sequence that was created in Creo Illustrate. A sequence is a combination of steps to perform a certain procedure, and it is transferred in with .pvz files when added into Vuforia Studio. The Play and Reset buttons will allow you to start or reset playback of the sequences associated with the quadcopter model.

   Note: The information in this section will only work if the **illustration** attribute has been filled out correctly for the model. Any part that does not have an **illustration** attribute will not be able to play a sequence.

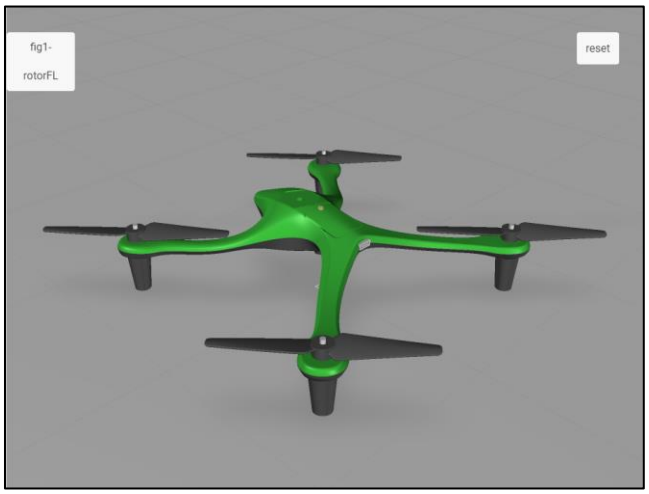
   a. When using the experience, after the popup disappears, the **playButton** widget should display the name of the sequence (if applicable) of the selected part. In **Home.js,** add the code below inside the **closePopup** function after `popup.close()` is called. The code will change the **Text** property of the **playButton** and edit an object called `toPlay` that will hold information about the model. The `toPlay` object will store the name of the model, which is defined by the `targetName` argument, and the illustration sequence of the model. Illustration sequences are stored as .pvi files inside .pvz files that are created in Creo Illustrate and then imported into Studio with the completed model from Illustrate. The format of these files is 1-Creo 3D- (figure name).pvi (unless you changed the Publish options, in which case Creo 3D may not be the string for the file format.). In this experience, the `instructionName` variable holds the value from the **illustration** attribute for the selected part for the **Figure** in Creo Illustrate that holds the animated sequence and is called into the string of text that is used to create the `instruction` property in the `toPlay` object.

```
        //
        //change the Text property of the playButton to the instructionName variable,
which was created from the JSON data of the model
         $scope.view.wdg.playButton.text = instructionName;
```

```
        //
        // create an object for the playButton called toPlay. This object will have
properties of model, which will be the name of the object that
        //is clicked on and instruction, which will add the proper syntax for calling a
sequence, based off the instructionName variable, into Studio
        $scope.view.wdg.playButton.toPlay = {       model: targetName,
                                                instruction: 'l-Creo 3D - ' +
instructionName + '.pvi' };
```

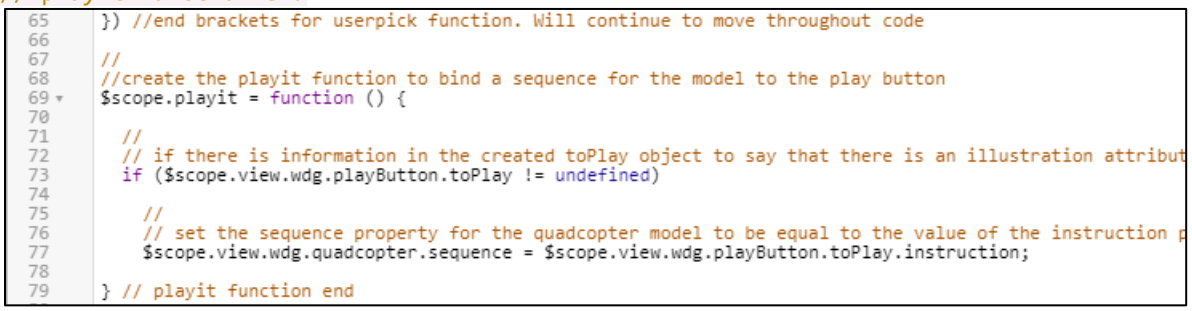
```
39          popup.close()
40
41          //
42          //change the Text property of the playButton to the instructionName variable, which was created from
43           $scope.view.wdg.playButton.text = instructionName;
44
45          //
46          // create an object for the playButton called toPlay. This object will have properties of model, whi
47          //is clicked on and instruction, which will add the proper syntax for calling a sequence, based off
48          $scope.view.wdg.playButton.toPlay = {       model: targetName,
49                                              instruction: 'l-Creo 3D - ' + instructionName + '.pvi' };
50
51      } // return end
```

b. Click **Save**, open the **Preview** tab, and click on the front-left rotor of the quadcopter when it appears. You should notice the change that happens to the **playButton** widget after the popup disappears. Try this for other parts and see the difference between ones that have sequences associated with them and the ones that don't.



c. Now that the text for the button is changed, a function needs to be bound to the button to play the sequence. If you remember, the function named *playit()* was added as a **Click JS** event for the **playButton** widget. This sets the value of the **Sequence** property of the **quadcopter** model to be equal to the `instruction` property of the `toPlay` object of **playButton**. In this case, `quadcopter` in the code below is referencing the **Studio ID** of the **model** widget that was added. If the **Studio ID** of the widget was not changed to *quadcopter* earlier in the project, this code will not work and the name of the model in the code will need to be changed. Place the following at the bottom of the code, as it is a separate function from `userpick`.

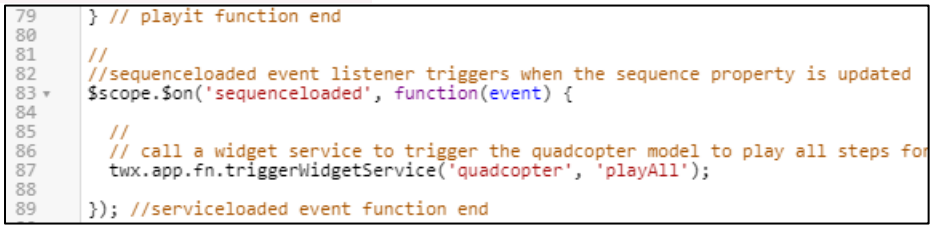
```
//
//create the playit function to bind a sequence for the model to the play button
$scope.playit = function () {

  //
  // if there is information in the created toPlay object to say that there is an
illustration attribute for the part
  if ($scope.view.wdg.playButton.toPlay != undefined)

    //
    // set the sequence property for the quadcopter model to be equal to the value of the
instruction property of the toPlay object
    $scope.view.wdg.quadcopter.sequence = $scope.view.wdg.playButton.toPlay.instruction;

} // playit function end
```



```
}) //end brackets for userpick function. Will continue to move throughout code

//
//create the playit function to bind a sequence for the model to the play button
$scope.playit = function () {

  //
  // if there is information in the created toPlay object to say that there is an illustration attribut
  if ($scope.view.wdg.playButton.toPlay != undefined)

    //
    // set the sequence property for the quadcopter model to be equal to the value of the instruction p
    $scope.view.wdg.quadcopter.sequence = $scope.view.wdg.playButton.toPlay.instruction;

} // playit function end
```

d. Once the **Sequence** property has been set for the model, the **playButton** widget needs to signal that sequence to start. This can be accomplished by using a new event listener, `sequenceloaded`, and widget service calls, which will start the playback of the sequence. Like in the last step, the **Studio ID** for the **model** widget needs to have been set to *quadcopter*, or else the code will need to be edited. Place this code below the `playit` function that you just created

```
//
//sequenceloaded event listener triggers when the sequence property is updated
$scope.$on('sequenceloaded', function(event) {

  //
  // call a widget service to trigger the quadcopter model to play all steps for the
given sequence

  twx.app.fn.triggerWidgetService('quadcopter', 'playAll');

}); //serviceloaded event function end
```



```
} // playit function end

//
//sequenceloaded event listener triggers when the sequence property is updated
$scope.$on('sequenceloaded', function(event) {

  //
  // call a widget service to trigger the quadcopter model to play all steps for
  twx.app.fn.triggerWidgetService('quadcopter', 'playAll');

}); //serviceloaded event function end
```

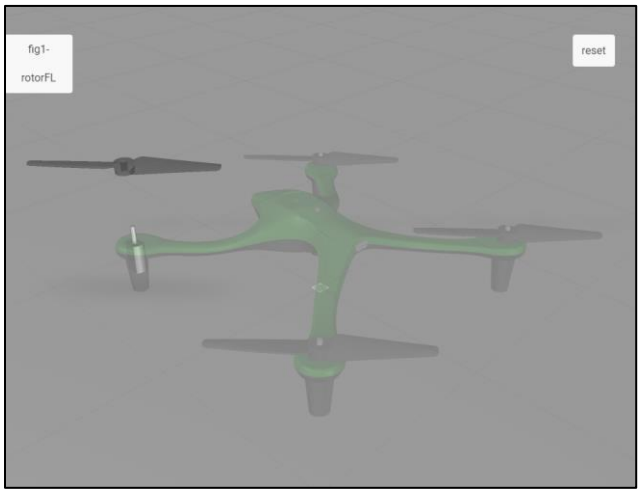
e. Similar to the **playButton** widget, the **resetButton** also has a JS event attached to it. This JS event is called *resetit()*. This standalone function

resets the model back to its original state, regardless if it is mid-playback or after a finished sequence. This happens by setting the **Sequence** property of the **quadcopter** model to being blank. This function will go below the `sequenceloaded` event listener

```
//
//resetit function
$scope.resetit = function () {

  //
  //set the sequence property of the quadcopter model to blank
  $scope.view.wdg.quadcopter.sequence = ''

} //resetit function end
```

```
89  }); //serviceloaded event function end
90
91  //
92  //resetit function
93  $scope.resetit = function () {
94
95    //
96    //set the sequence property of the quadcopter model to blank
97    $scope.view.wdg.quadcopter.sequence = ''
98
99  } //resetit function end
```

f. Once the code has been completely added to **Home.js**, click **Save** and open **Preview**. Click on the front-left rotor. If the popup appears, then the play button is populated with text, and when clicked, starts the repair sequence. Try pressing the **reset** button during the sequence. You should see the model return to its original state. If this happens, this has been completed successfully. The full code for this section is in Appendix 6.



201.5 Highlight Parts

This experience uses a dynamic structure for selecting parts on the model. In a Studio context, this means that parts of a model can be selected on their own without the need for a Model Item to separate the parts. Using the **tml3DRenderer,** a Studio renderer

based on WebGL, and the **occurrence** data offers users a set of services for changing 3D components based on their node IDs, which is what allows dynamic changes based on which part is selected by the user. This allows the model to be more interactive without needing to add widgets to the experience.

When selected, parts should not only have an Ionic popup appear with their part information, they should also be highlighted. Using JavaScript code and TML Text widgets to create shaders, this section will explain how to do just that.

1. In **Home.js**, a new function for highlighting parts called `hilite` will need to be created. This function will have inputs of `items`, which will be the part that is selected, and `hilite`, which is a Boolean that decides if a part needs to be highlighted. Inside the function, the `tml3DRenderer` object will be edited, which corresponds to the **TML Text** widget that will be added in the next step. The `tml3DRenderer` object calls the `.setProperties` service to set the properties of the object. This service intakes the item that is supposed to be highlighted and then checks to see if the Boolean value `hilite` that has been input to the function is true or not. The `?` conditional operator evaluates if `hilite` is true or false and then returns a set of object properties based on if it is true or false. If `hilite` is true, then the following properties will be set for the object: `{ shader:"green", hidden:false, opacity:0.9, phantom:false, decal:true }`. In this case, *green* is the name given to the **TML Text** widget that will be added to the next step, so this line of code is what calls the TML Text widget to activate. If `hilite` is false, then `{shader:"Default", hidden:false, opacity:1.0, phantom:false, decal:false}` will be set as the properties, which signal the parts to just stay their default views. This portion of the function is editing the graphic interface of Studio using WebGL. This code will not highlight the selected part until the **TML Text** widget is added.

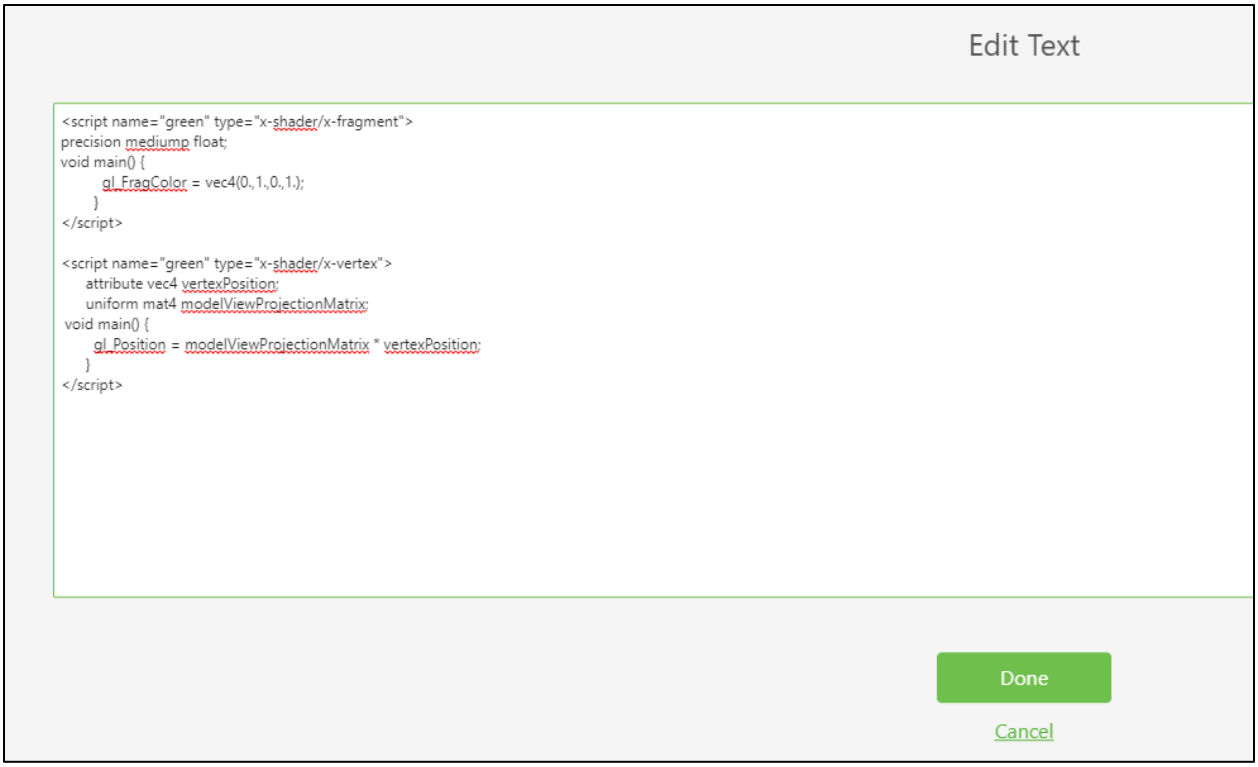
```
//
// highlighting function. Inputs are the selected part and a boolean for hilite
$scope.hilite = function (items, hilite) {

  //
  //iterate over each item that is used as an imported variable for the function using
.forEach to look at each value that comes in the items input
  items.forEach(function(item) {

    //
    //set the properties of the TML 3D Renderer to highlight the selected item using a
TML Text shader. "green" is the name of the script for the TML Text.
    tml3dRenderer.setProperties(item, hilite === true ? { shader: "green", hidden: false,
opacity: 0.9, phantom: false, decal: true }
                                                    : { shader: "Default", hidden:
false, opacity: 1.0, phantom: false, decal: false });

  }) //foreach end

} //hilite function end
```

```
99  } //resetit function end
100
101 //
102 // highlighting function. Inputs are the selected part and a boolean for hilite
103 $scope.hilite = function (items, hilite) {
104
105   //
106   //iterate over each item that is used as an imported variable for the function using .forEach to look at each value that comes in the ite
107   items.forEach(function(item) {
108
109     //
110     //set the properties of the TML 3D Renderer to highlight the selected item using a TML Text shader. "green" is the name of the script
111     tml3dRenderer.setProperties(item, hilite === true ? { shader: "green", hidden: false, opacity: 0.9, phantom: false, decal: true }
112                                                       : { shader: "Default", hidden: false, opacity: 1.0, phantom: false, decal: false });
113
114   }) //foreach end
115
116 } //hilite function end
```

2. Navigate back to **Home** in the **Project** pane to view the **3D** canvas. Drag a **TML Text** widget onto the canvas. Change the **Studio ID** of the widget to *shaders*. Click the green edit icon next to the **Text** property to open the **Edit Text** box, and enter ONLY the code below, not the comments. Click **Done**. This code will allow the shader to be applied whenever the function is called in the script.

   Note: The comments are only for explaining the code in this context. If included with the code, they will appear on the screen.

```
//name of the shader is green, the type is setting the color.
//
<script name="green" type="x-shader/x-fragment">
//
// setting the precision of the shader. medium is fine for this application.
  precision mediump float;
//
// function to set the color of the shader. Syntax is vec4(R, G, B, A) format and the
values are on a 0.0-1.0 scale
void main() {
          gl_FragColor = vec4(0.,1.,0.,1.);
        }
</script>
// name of the shader is green, this time the type sets the position
<script name="green" type="x-shader/x-vertex">
      attribute vec4 vertexPosition;
      uniform mat4 modelViewProjectionMatrix;
//
// sets the position of the vertex
 void main() {
        gl_Position = modelViewProjectionMatrix * vertexPosition;
      }
</script>
```

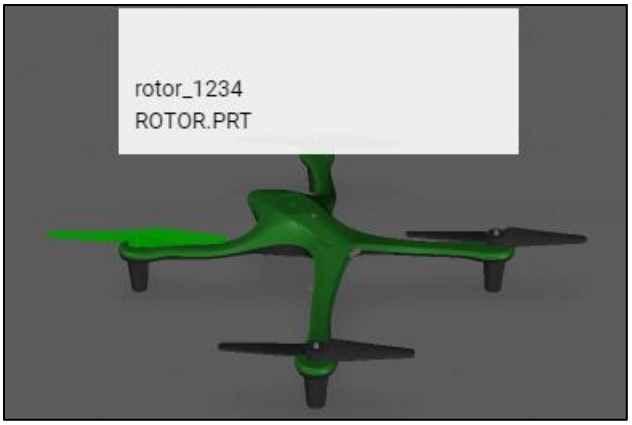


3. Now that the function is set up, it is time to add it to the code to be called when a part is clicked on. In the `userpick` function after the popup is called, add the code below. This uses the `currentSelection` variable that was created earlier to determine which part has been clicked on by inputting the name of the model that was selected along with the selected part occurrence in the model to determine the highlighted portion. For example, if the front-left rotor is selected, `[$scope.currentSelection]` becomes [quadcopter + "-" + /0/18/0/0], or quadcopter-/0/18/0/0 if the string is written out. `True` sets the Boolean to be true so that the highlight shader is applied.

```
//
//highlight the chosen item and set the shader to true
$scope.hilite([$scope.currentSelection], true);
```

```
28      }); //end of ionic popup
29
30      //
31      //highlight the chosen item and set the shader to true
32      $scope.hilite([$scope.currentSelection], true);
```

4. Click **Save** and then **Preview** the experience to make sure that the shader works correctly. If a part turns green when clicked, then the shader has been successfully added.



5. Notice, that the shader does not disappear when the popup does. This will also be added to the code in **Home.js** so that the shader disappears in the same function. `refitems` will be added as an input to the `closePopup` function, and the `hilite` function will be called inside that function. This time the Boolean input will be set to `false`, thus turning off the shader. Add the additional code to the existing `closePopup` function.

   Note: Make sure that `refitems` is added in as a new input for the `closePopup` function. Otherwise the experience will not work as intended.

```
    //
    // create a function to close the popup and turn off shading. popup is the popup,
refitems is the input for the part(s) that is being highlighted
    var closePopup = function (popup, refitems) {

      //
      //The function returns a method for removing the popup from the screen and turns
off the shader
      return function () {

        //
        //using the input parts, set the hilite function to be false, removing the
shading
        $scope.hilite(refitems, false)

        //
        //apply the .close method, which removes a certain section of a selected object,
to the popup variable
        popup.close()

        //
        //change the Text property of the playButton to the instructionName variable,
which was created from the JSON data of the model
        $scope.view.wdg.playButton.text = instructionName;

        //
        // create an object for the playButton called toPlay. This object will have
properties of model, which will be the name of the object that
        //is clicked on and instruction, which will add the proper syntax for calling a
sequence, based off the instructionName variable, into Studio
```

```
        $scope.view.wdg.playButton.toPlay = {       model: targetName,
                                                instruction: 'l-Creo 3D - ' +
instructionName + '.pvi' };

      } //return end

    } // closepopup function end
```

```
32      $scope.hilite([$scope.currentSelection], true);
33
34      //
35      // create a function to close the popup and turn off shading. popup is the popup, refitems is the input for the par
36      var closePopup = function (popup, refitems) {
37
38        //
39        //The function returns a method for removing the popup from the screen and turns off the shader
40        return function () {
41
42          //
43          //using the input parts, set the hilite function to be false, removing the shading
44          $scope.hilite(refitems, false)
45
46          //
47          //apply the .close method, which removes a certain section of a selected object, to the popup variable
48          popup.close()
49
50          //
51          //change the Text property of the playButton to the instructionName variable, which was created from the JSON d
52          $scope.view.wdg.playButton.text = instructionName;
53
54          //
55          // create an object for the playButton called toPlay. This object will have properties of model, which will be
56          //is clicked on and instruction, which will add the proper syntax for calling a sequence, based off the instruc
57          $scope.view.wdg.playButton.toPlay = {       model: targetName,
58                                                  instruction: 'l-Creo 3D - ' + instructionName + '.pvi' };
59
60        } //return end
61
62      } // closepopup function end
```

6. Update the target of the `closePopup` function to be the selected part.

```
$timeout(closePopup(popup, [$scope.currentSelection]), 3000);
```

```
62      } // closepopup function end
63
64      //
65      //call the $timeout service which will call the function for closing the popup after 3 se
66      $timeout(closePopup(popup, [$scope.currentSelection]), 3000);
```

7. Click **Save** the **Preview** tab and view the experience now. If the shader disappears when the popup disappears after a few seconds, then this step has been completed successfully. The full code for this section is available in Appendix 7. Save the project so that it can be used again to complete the following Metadata 202 tutorial. The completed project file for this tutorial is provided in the Metadata201 folder in GitHub.

Appendix 1: Sequence & Figured Related Vuforia Studio Events

Event Name (evt.name)	Description	Arguments/Usage Notes	Example
newStep	Triggered by going to a new step in an animation sequence	arg1 (text) of the following form: (<step #>/<total steps>) <step name>, e.g. "(4/8) Step 4 -remove case"	`$scope.$on('newStep', function(evt, arg) { $scope.view.wdg['label-1']['text'] = $scope.view.wdg['label-1']['text'] + " evt: " + evt.name + " arg: " + arg ;});`
playstarted	Triggered by play or play all of an animation sequence	No arguments returned	`$scope.$on('playstarted', function(evt, arg) { $scope.view.wdg['label-1']['text'] = $scope.view.wdg['label-1']['text'] + " evt: " + evt.name;});`
stepstarted	Similar to playstarted but with more flexible argument data returned	Arg1 is model name (e.g. "model-1") Arg2 is the type of object (twx-dt-model) Arg3 is an JSON object containing stepName,duration (in ms), acknowledge (boolean), totalSteps (int), nextStep (int). Note: Studio Preview for 8.3.2 now also includes stepDescription (the 'notes' from the step in Creo Illustrate) and acknowledgeMessage. However these two fields are not yet exposed for use in Vuforia View.	`$scope.$on('stepstarted', function(evt, arg, arg2, arg3) { var parsedArg3 = JSON.parse(arg3); $scope.view.wdg['label-1']['text'] = $scope.view.wdg['label-1']['text'] + " evt: " + evt.name + " arg: " + arg + " arg2: " + arg2 + " arg3 fields: " + parsedArg3.stepName + " " + parsedArg3.duration + " " + parsedArg3.totalSteps ;});`
playstopped	Triggered by stop of an animation sequence	arg1 (object) contains stepName,duration (in ms), acknowledge (boolean), acknowledgeMessage,totalSteps (int), nextStep (int)	`$scope.$on('playstopped', function(evt, arg) { $scope.view.wdg['label-1']['text'] = $scope.view.wdg['label-1']['text'] + " evt: " +`

			`evt.name + " arg: " + arg.stepName ;});`
stepstopped	Similar to playstarted but with more flexible argument data returned	See ‘stepstarted’	`Same as ‘stepstarted’ except with event name of ‘stepstopped’`
sequenceloaded	Triggered by sequence loading (when a model with a sequence is loaded, or the sequence property is updated for a model).	E.g. arg1 "model-1", arg2 is the type of object (twx-dt-model), arg3 is the model's current 'sequence' property (e.g. app/resources/Uploaded/mypvzfile/mysequence-name.pvi)	`$scope.$on('sequenceloaded', function(evt, arg, arg2, arg3) { $scope.view.wdg['label-1']['text'] = $scope.view.wdg['label-1']['text'] + " evt: " + evt.name + " arg: " + arg + " arg2: " + arg2 + " arg3: " + arg3 ;});`
sequenceacknowledge	Triggered by a figure/sequence that was defined in Creo Illustrate as having an acknowledgement	See ‘playstopped’	`Same as ‘playstopped’ except with event name of ‘sequenceacknowledge’`
sequencereset	Triggered by the model ‘reset’ (of the sequence/figure) event	Arg1 is model name (e.g. “model-1”) Arg2 is the type of object (twx-dt-model)	`$scope.$on('sequencereset', function(evt, arg, arg2) { $scope.view.wdg['label-1']['text'] = $scope.view.wdg['label-1']['text'] + " evt: " + evt.name + " arg1: " + arg + " arg2: " + arg2;});`

Appendix 2: 3D Object Related Studio Events

Event Name (evt.name)	Description	Arguments/Usage Notes	Example
modelLoaded	Triggered when a model is loaded (can be multiple times if an experience includes multiple models) as well as when a model's	Arg1 is the model name (e.g. “model-1”)	`$scope.$on('modelLoaded', function(evt, arg) { $scope.view.wdg['label-1']['text'] = $scope.view.wdg['label-1']['text'] + " evt: " + evt.name + " arg: " + arg ;});`

	'Resource' property is updated.		
Loaded3DObj	Triggered when any 3D widget is loaded in an experience. Similar to 'modelLoaded' event but applies to all 3D widgets.	See 'modelLoaded'	`Same as 'modelLoaded' except with event name of 'loaded3DObj'`
userpick	Triggered by users clicking on 3D objects in the experience (e.g. model items, models, 3D labels)	Arguments returned include event (name), target (e.g. model-1 or modelItem-1, or 3DImage-1, etc.), parent (null), edata (JSON object containing 'occurrence' property value for modelItems as defined in the PVZ, e.g. /0/0/18)	`$scope.$on('userpick', function(event,target,parent,edata){ if (edata) { console.log('my console of userpick evt: '+ event.name + " target: " + target + " and parent:" + parent + " edata.occurence: " + JSON.parse(edata).occurrence); } });`
click	Similar to userpick	Event includes the widget's Studio ID in tergetScope._widgetId. No args data returned.	`$scope.$on('click', function(evt, arg) { $scope.view.wdg['debug-label']['text'] = " evt: " + evt.name + " event targetScope Widget ID: " + evt.targetScope._widgetId;});`
trackingacquired	Triggered when a ThingMark, Spatial, or Model Target is acquired ('locked onto' by Vuforia View	Arg1 for ThingMark Targets is the ThingMark ID (e.g. 555:10), no arguments for other target types	`$scope.$on('trackingacquired', function(evt, arg) { $scope.view.wdg['label-1']['text'] = "evt: " + evt.name + " arg: " + arg ;});`
trackinglost	Triggered when a ThingMark, Spatial, or Model Target is lost ('locked has been lost') by Vuforia View	See 'trackingacquired'	`Same as 'trackingacquired' except with event name of 'trackinglost'`

Appendix 3: ThingWorx External Data Services Studio Events

Event Name (evt.name)	Description	Arguments/Usage Notes	Example

(servicename)-begin	Triggered by start of a TWX service defined in the Studio project	No arguments returned, just event.name.Note: event is broadcast to root scope, therefore must listen with $scope.$root.$on, not just $scope.$on).	`$scope.$root.$on('myTWXService-begin', function(evt, arg) { $scope.view.wdg['label-1']['text'] = $scope.view.wdg['label-1']['text'] + " event name: " + evt.name;});`
(servicename)-complete	Triggered by successful completion of a TWX service defined in the Studio project	See (servicename)-begin	`See (servicename)-begin`
(servicename)-end	Triggered by completion of a TWX service (whether successful or not)	See (servicename)-begin	`See (servicename)-begin`
(servicename)-failed	Triggered by failure when calling a TWX service	Returns arg that is JSON {service: logicalName, params: serviceParams, reason: reason}	

Appendix 4: Section 201.2 Code

```
//
// triggered when user clicks on object in the scene
$scope.$on('userpick', function (event, targetName, targetType, eventData) {

  //
  // variable to pull the value for the occurrence property in the eventData JSON object from the model. Create variable for the currently selected part
  var pathId = JSON.parse(eventData).occurrence
  $scope.currentSelection = targetName + "-" + pathId

  //
  // adds an ionic popup when a part is clicked. Show the pathId of the selected object
  var popup = $ionicPopup.show({
    template: '<div>' + pathId +  '</div>',
    scope: $scope
  }); //end of ionic popup

  // create a function to close the popup.
  var closePopup = function (popup) {
```

```
    //
    // function for returning that the popup will be closed using the .close() method
    return function() {

      //
      //close the popup
      popup.close()

    } // return end

  } // closepopup function end

  //
  //call the $timeout service which will call the function for closing the popup after 3 seconds (3000 ms)
  $timeout(closePopup(popup), 3000);

}) //end brackets for userpick function. Will continue to move throughout code
```

Appendix 5: Section 201.3 Code

```
//
// triggered when user clicks on object in the scene
$scope.$on('userpick', function (event, targetName, targetType, eventData) {

  //
  //Look at model and see if it has metadata. If it does, then execute the below code and create an object called metadata
  PTC.Metadata.fromId(targetName)
              .then  ( (metadata) => {

    //
    // variable to pull the value for the occurrence property in the eventData JSON object from the model. Create variable
for the currently selected part
    var pathId = JSON.parse(eventData).occurrence
    $scope.currentSelection = targetName + "-" + pathId

    //
    // create variables based on attribute names from Creo Illustrate for this model. use metadata.get to obtain the data
from the JSON properties for this occurrence.
    var partName        = metadata.get(pathId, 'Display Name');
    var instructionName = metadata.get(pathId, 'illustration');
    var partNumber      = metadata.get(pathId, 'partNumber');
```

```
    //
    // adds an ionic popup when a part is clicked. Show the part number and name of the selected object.  </br> adds a
line break between the two variables
    var popup = $ionicPopup.show({
      template: '<div>' + partNumber + ' </br>' + partName + '</div>',
      scope: $scope
    }); //end of ionic popup

    // create a function to close the popup.
    var closePopup = function (popup) {

      //
      // function for returning that the popup will be closed using the .close() method
      return function() {

        //
        //close the popup
        popup.close()

      } // return end

    } // closepopup function end

    //
    //call the $timeout service which will call the function for closing the popup after 3 seconds (3000 ms)
    $timeout(closePopup(popup), 3000);

  }) //end brackets for PTC API and .then

  //
  //catch statement if the promise of having a part with metadata is not met
  .catch( (err) => { console.log('metadata extraction failed with reason : ' +err) })

}) //end brackets for userpick function. Will continue to move throughout code
```

Appendix 6: Section 201.4 Code

```
//
// triggered when user clicks on object in the scene
$scope.$on('userpick', function (event, targetName, targetType, eventData) {

  //
  //Look at model and see if it has metadata. If it does, then execute the below code and create an object called metadata
```

```
  PTC.Metadata.fromId(targetName)
              .then  ( (metadata) => {

    //
    // variable to pull the value for the occurrence property in the eventData JSON object from the model. Create variable
for the currently selected part
    var pathId = JSON.parse(eventData).occurrence
    $scope.currentSelection = targetName + "-" + pathId

    //
    // create variables based on attribute names from Creo Illustrate for this model. use metadata.get to obtain the data
from the JSON properties for this occurrence.
    var partName        = metadata.get(pathId, 'Display Name');
    var instructionName = metadata.get(pathId, 'illustration');
    var partNumber      = metadata.get(pathId, 'partNumber');

    //
    // adds an ionic popup when a part is clicked. Show the part number and name of the selected object.  </br> adds a
line break between the two variables
    var popup = $ionicPopup.show({
      template: '<div>' + partNumber + ' </br>' + partName + '</div>',
      scope: $scope
    }); //end of ionic popup

    // create a function to close the popup.
    var closePopup = function (popup) {

      //
      // function for returning that the popup will be closed using the .close() method
      return function() {

        //
        //close the popup
        popup.close()

        //
        //change the Text property of the playButton to the instructionName variable, which was created from the JSON data
of the model
         $scope.view.wdg.playButton.text = instructionName;

        //
```

```
        // create an object for the playButton called toPlay. This object will have properties of model, which will be the
name of the object that
        //is clicked on and instruction, which will add the proper syntax for calling a sequence, based off the
instructionName variable, into Studio
        $scope.view.wdg.playButton.toPlay = {       model: targetName,
                                              instruction: 'l-Creo 3D - ' + instructionName + '.pvi' };

      } // return end

    } // closepopup function end

    //
    //call the $timeout service which will call the function for closing the popup after 3 seconds (3000 ms)
    $timeout(closePopup(popup), 3000);

  }) //end brackets for PTC API and .then

  //
  //catch statement if the promise of having a part with metadata is not met
  .catch( (err) => { console.log('metadata extraction failed with reason : ' +err) })

}) //end brackets for userpick function. Will continue to move throughout code

//
//create the playit function to bind a sequence for the model to the play button
$scope.playit = function () {

  //
  // if there is information in the created toPlay object to say that there is an illustration attribute for the part
  if ($scope.view.wdg.playButton.toPlay != undefined)

    //
    // set the sequence property for the quadcopter model to be equal to the value of the instruction property of the toPlay
object
    $scope.view.wdg.quadcopter.sequence = $scope.view.wdg.playButton.toPlay.instruction;

} // playit function end

//
//sequenceloaded event listener triggers when the sequence property is updated
$scope.$on('sequenceloaded', function(event) {
```

```
  //
  // call a widget service to trigger the quadcopter model to play all steps for the given sequence
  twx.app.fn.triggerWidgetService('quadcopter', 'playAll');

}); //serviceloaded event function end

//
//resetit function
$scope.resetit = function () {

  //
  //set the sequence property of the quadcopter model to blank
  $scope.view.wdg.quadcopter.sequence = ''

} //resetit function end
```

Appendix 7: Section 201.5 Code

```
//
// triggered when user clicks on object in the scene
$scope.$on('userpick', function (event, targetName, targetType, eventData) {

  //
  //Look at model and see if it has metadata. If it does, then execute the below code and create an object called metadata
  PTC.Metadata.fromId(targetName)
              .then  ( (metadata) => {

    //
    // variable to pull the value for the occurrence property in the eventData JSON object from the model. Create variable
for the currently selected part
    var pathId = JSON.parse(eventData).occurrence
    $scope.currentSelection = targetName + "-" + pathId

    //
    // create variables based on attribute names from Creo Illustrate for this model. use metadata.get to obtain the data
from the JSON properties for this occurrence.
    var partName        = metadata.get(pathId, 'Display Name');
    var instructionName = metadata.get(pathId, 'illustration');
    var partNumber      = metadata.get(pathId, 'partNumber');

    //
```

```
    // adds an ionic popup when a part is clicked. Show the part number and name of the selected object.  </br> adds a
line break between the two variables
    var popup = $ionicPopup.show({
      template: '<div>' + partNumber + ' </br>' + partName + '</div>',
      scope: $scope
    }); //end of ionic popup

    //
    //highlight the chosen item and set the shader to true
    $scope.hilite([$scope.currentSelection], true);

    //
    // create a function to close the popup and turn off shading. popup is the popup, refitems is the input for the part(s)
that is being highlighted
    var closePopup = function (popup, refitems) {

      //
      //The function returns a method for removing the popup from the screen and turns off the shader
      return function () {

        //
        //using the input parts, set the hilite function to be false, removing the shading
        $scope.hilite(refitems, false)

        //
        //apply the .close method, which removes a certain section of a selected object, to the popup variable
        popup.close()

        //
        //change the Text property of the playButton to the instructionName variable, which was created from the JSON data
of the model
        $scope.view.wdg.playButton.text = instructionName;

        //
        // create an object for the playButton called toPlay. This object will have properties of model, which will be the
name of the object that
        //is clicked on and instruction, which will add the proper syntax for calling a sequence, based off the
instructionName variable, into Studio
        $scope.view.wdg.playButton.toPlay = {       model: targetName,
                                              instruction: 'l-Creo 3D - ' + instructionName + '.pvi' };

      } //return end
```

```
    } // closepopup function end

    //
    //call the $timeout service which will call the function for closing the popup after 3 seconds (3000 ms)
    $timeout(closePopup(popup, [$scope.currentSelection]), 3000);

  }) //end brackets for PTC API and .then

  //
  //catch statement if the promise of having a part with metadata is not met
  .catch( (err) => { console.log('metadata extraction failed with reason : ' +err) })

}) //end brackets for userpick function. Will continue to move throughout code

//
//create the playit function to bind a sequence for the model to the play button
$scope.playit = function () {

  //
  // if there is information in the created toPlay object to say that there is an illustration attribute for the part
  if ($scope.view.wdg.playButton.toPlay != undefined)

    //
    // set the sequence property for the quadcopter model to be equal to the value of the instruction property of the toPlay
object
    $scope.view.wdg.quadcopter.sequence = $scope.view.wdg.playButton.toPlay.instruction;

} // playit function end

//
//sequenceloaded event listener triggers when the sequence property is updated
$scope.$on('sequenceloaded', function(event) {

  //
  // call a widget service to trigger the quadcopter model to play all steps for the given sequence
  twx.app.fn.triggerWidgetService('quadcopter', 'playAll');

}); //serviceloaded event function end

//
//resetit function
```

```
$scope.resetit = function () {

  //
  //set the sequence property of the quadcopter model to blank
  $scope.view.wdg.quadcopter.sequence = ''

} //resetit function end

//
// highlighting function. Inputs are the selected part and a boolean for hilite
$scope.hilite = function (items, hilite) {

  //
  //iterate over each item that is used as an imported variable for the function using .forEach to look at each value that comes in the items input
  items.forEach(function(item) {

    //
    //set the properties of the TML 3D Renderer to highlight the selected item using a TML Text shader. "green" is the name of the script for the TML Text.
    tml3dRenderer.setProperties(item, hilite === true ? { shader: "green", hidden: false, opacity: 0.9, phantom: false, decal: true }
                                                       : { shader: "Default", hidden: false, opacity: 1.0, phantom: false, decal: false });

  }) //foreach end

} //hilite function end
```

Appendix 8: CAD Metadata API Functions (from PTC support site)

Declaration	Parameters	Description
`get` (idpath, propName, categoryName)	`string|string[]} idpath`— id path such as `'/0/1'`, or array of id paths `['/0/1', '/0/2']`. `string|string[]} propName`— (Optional) For example, `'Display`	Gets a metadata object representing the id path or property value(s) for the given `idpath` and `propName`. This function returns the metadata object representing the given `idpath`, or if `propName` is given then the value of the property on the component. Example:

	Name' or ['Display Name', 'Part ID Path'] string|string[]} categoryName—(Optional) For example, 'PROE Parameters'. If propName was string [], then categoryName must also be an array of matching length (or undefined).	PTC.Metadata.fromId('model-1').then((metadata) => { var result = metadata.get('/0/6', 'Display Name') });

getProp (propName, categoryName)	string|string[]} propName—(Optional) For example, 'Display Name' or ['Display Name', 'Part ID Path']
	string|string[]} categoryName—(Optional) For example, 'PROE Parameters'.
	If propName was string [], then categoryName must also be an array of matching length (or undefined).
		This function returns all string property values from a single component, or undefined if no data/components available. If the given propName was an array, it returns string[] of values.
		Example:
		PTC.Metadata.fromId('model-1').then((metadata) => {
		var result = metadata.get('/0/1').getProp('Display Name');
		});

getCategory (categoryName)	string} categoryName	This function returns object with all property names and values from given category.
		Example:
		PTC.Metadata.fromId('model-1').then((metadata) => {
		var result = metadata.get('/0/6'). getCategory ('__PV_SystemProperties');
		});

`getSelected` (selectFunc)	`function} selectFunc`—(Optional) Function that controls the values put into the returned array. The function is given `idpath` and an argument and current metadata as: `` `this` function(idpath) { return [idpath, this.get(idpath, 'Display Name')]; }); ``	This function returns an array of whatever is returned by the given `selectFunc`, or if `selectFunc` is undefined, then it returns string[] of id paths. Example: `PTC.Metadata.fromId('model-1').then( (metadata) => { var selectFunc = function(idpath) { return metadata.get(idpath, 'Display Name'); } var result = metadata.getSelected(selectFunc); });`

`find` (propName, category)	`string} propName`—(Required)
	`string} category`—(Optional)
		Finds components based on property values. Also see `findCustom` below.
		Returns a finder for components based on given the `propName` and category.
		Example:
		`PTC.Metadata.fromId('model-1').then( (metadata) => { var displayName = metadata.find('Display Name').like('BOLT'); }); PTC.Metadata.fromId('model-1').then( (metadata) => { var result = metadata.find('Part Depth').lessThan(3).find('Display Name').like('PRT'); });`
		`PTC.Metadata.fromId('model-1').then( (metadata) => { var selectFunc = function(idpath) { return metadata.get(idpath, 'Display Name') var result = metadata.find('Part Depth').greaterThan(4, selectFunc) });`
		The comparison can be as follows:
		`- startsWith,like,sameAs,unlike : string comparison - equal,notequal,greaterThanEq,lessThanEq,lessThan,greaterThan : numeric comparison - in,out : numeric range comparison - before,after : date/time comparison`

`findCustom` (whereFunc, selectFunc)	`function} whereFunc`— (Required) `function} selectFunc`— (Optional)	Also see `find` above. This function returns a finder for components based on custom `whereFunc`. The following example finds all components with `depth<2` or has a name like `'ASM'`. Example: `PTC.Metadata.fromId('model-1').then((metadata) => { var whereFunc = function(idpath) { const depth = metadata.get(idpath, 'Part Depth') const name = metadata.get(idpath, 'Display Name') return parseFloat(depth) >= 4 || (name && name.search('ASM') >= 0) } var result = metadata.findCustom(whereFunc); });`