Object Detection in Browser using TensorFlow.js

July 27, 2024

In order to import the TensorFlow.js library that enables Machine Learning in web applications, this line of code will be included in the header file of the HTML for the webpage where you can view the camera footage.

[ ]:
```
<script src="https:\/\/cdn.jsdelivr.net/npm/@tensorflow/tfjs@1.3.1/dist/tf.min.
↪js"> </script>
```

This next line loads the pre-trained models from the COCO dataset wherein **bottle** is one of the objects that can be detected from an image or from a frame.

[ ]:
```
<script src="https:\/\/cdn.jsdelivr.net/npm/@tensorflow-models/coco-ssd@2.1.
↪0"> </script>
```

A variable with the name *host* will be declared and the value assigned to it will allow connection to the **WebSocket** server hosted on the ESP32-CAM board which the ESP32 will later be used to transfer the result of the image processing from the web application to the ESP32CAM.

[ ]:
```
var host = `ws://${window.location.hostname}:100/ws`;
```

The line below enclosed in brackets is a placeholder automatically replaced by the IP address of the ESP32-CAM that is hosting the camera stream webpage. **100** is the port where it listens for new connections via WebSocket.

[ ]:
```
${window.location.hostname}:100
```

Declare a variable with the name *websocket* which will later store a value later.

[ ]:
```
var websocket;
```

In the line below, the **addEventListener()** method will be applied to the window or webpage that loads when the camera stream is opened on a browser. The arguments given to this method mean that the window listens for the *load* event or for the webpage to load, and then a callback function **onLoad()** will be executed.

[ ]:
```
window.addEventListener('load', onLoad);
```

The **onLoad** callback will execute the **initWebSocket()** function.

[ ]:
```
function onLoad(event) {
        initWebSocket();
    }
```

In the block of code below, the **initWebSocket()** function is defined. The statement logged into the console is mainly for testing during development to give the developer an indication that the function is now being called.

[ ]:
```
function initWebSocket() {
        console.log('Trying to open a WebSocket connection...');
        websocket = new WebSocket(host);
        websocket.onopen  = onConnect;
        websocket.onclose = onDisconnect;
        websocket.onmessage = onMessage;
    }
```

A **WebSocket object** is initialized to create a **WebSocket client** that can be addressed using the websocket variable. The argument *host* supplied to object means that this WebSocket client will attempt to the **WebSocket** host declared earlier in the code.

[ ]:
```
 websocket = new WebSocket(host);
```

Callback functions will be assigned to the *onopen*, *onclose*, and *onmessage* methods of the websocket object.

[ ]:
```
websocket.onopen  = onConnect;
websocket.onclose = onDisconnect;
websocket.onmessage = onMessage;
```

Inside the definition of the onConnect callback function, a message will be logged into the console to signal that a connection to the WebSocket server has been opened. The JavaScript program will proceed with sending a "Hello" message to the WebSocket server hosted in the ESP32-CAM.

[ ]:
```
function onConnect(event) {
        console.log('Connection opened');
        websocket.send("Hello")
    }
```

Likewise, a statement will be printed on the console after a 2 second delay if the connection to the WebSocket server closes.

[ ]:
```
function onDisconnect(event) {
        console.log('Connection closed');
        setTimeout(initWebSocket, 2000);
    }
```

Whenever the WebSocket server sends message to the websocket client object in this script and the onMessage callback function is executed, the data will be printed on the console.

[ ]:
```
function onMessage(event) {
    console.log(event.data);
   }
```

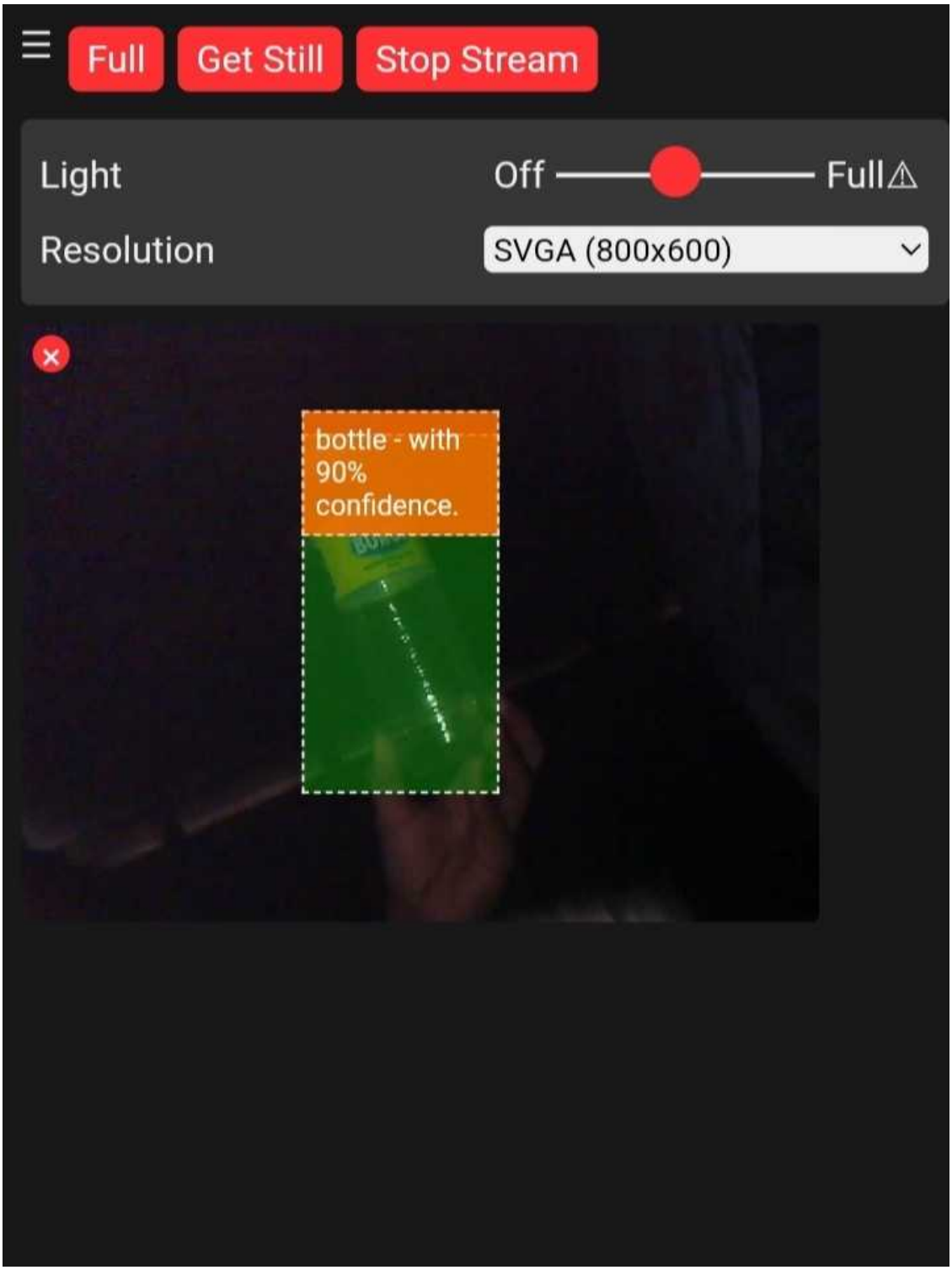
A **loadModels()** function will be defined to disable the display for the camera footage if the COCO models have not been loaded yet.

[ ]:
```
function loadModels() {
      cocoSsd.load().then(cocoSsd_Model => {
        Model = cocoSsd_Model;
        hide(loadingMessage);
        show(streamButton);
      });
    }
```

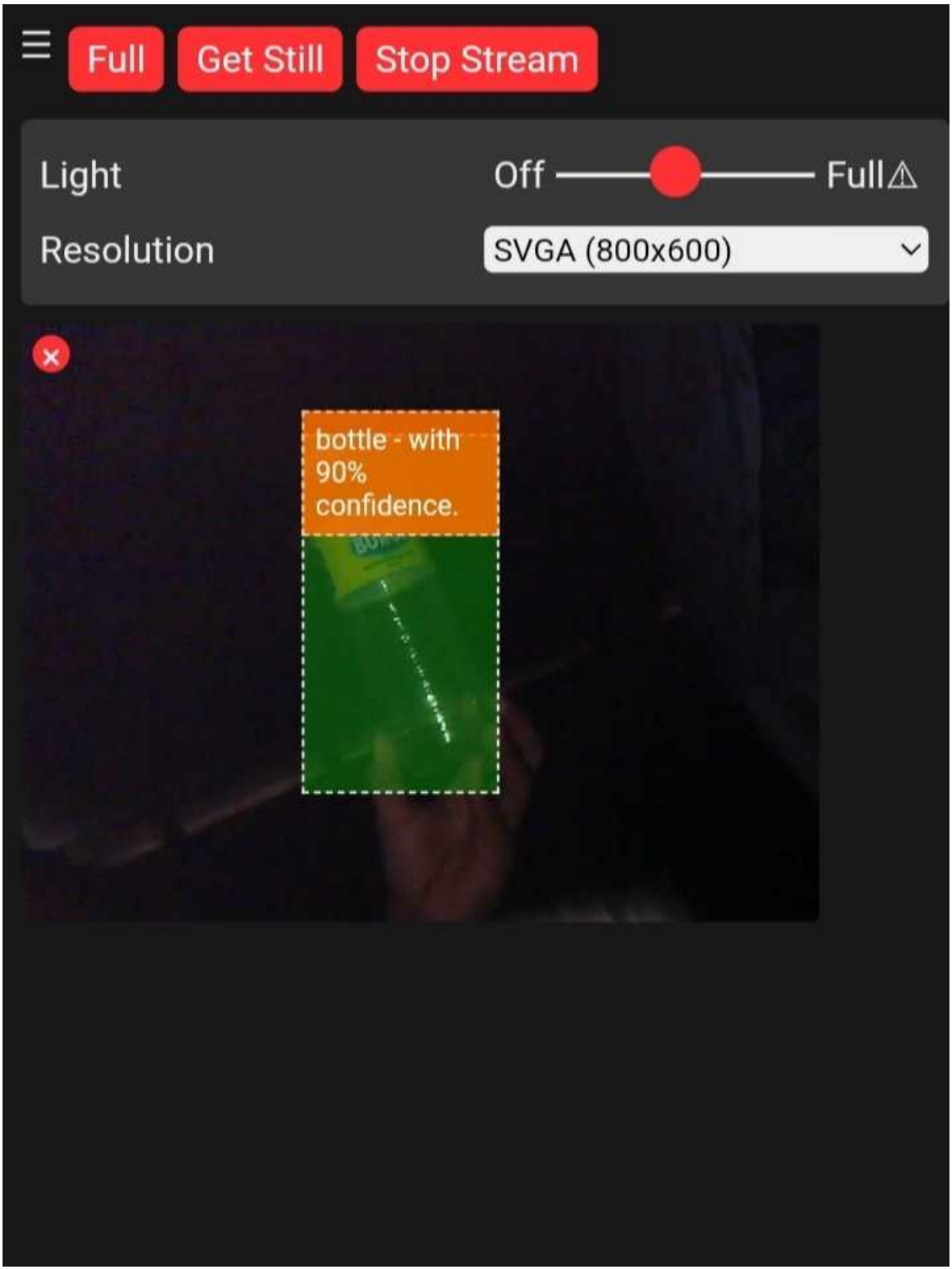
Once the cocoSsd_Model object to be used with TensorFlow.js for the image processing has finished being loaded, this will be stored in the Model variable declared earlier.

[ ]:
```
 Model = cocoSsd_Model;
```

The loading message below will disappear when the COCO machine learning or object detection models have been loaded.



The Start stream button below will also appear.



The **loadModels()** function is called when the webpage is opened in the browser and the initial values of buttons and other HTML elements are first loaded.

```
[ ]: fetch(`${baseHost}/status`)
       .then(function (response) {
         return response.json()
       })
       .then(function (state) {
         document
           .querySelectorAll('.action-setting')
           .forEach(el => {
             updateValue(el, state[el.id], false)
           })
         hide(waitSettings);
         show(settings);
         loadModels();
       })
```

The **getElementById()** method of the HTML DOM is used to address the elements containing the display for the camera stream within the HTML of the web page. The view variable stores the *img* element with the ID: **#stream** which displays the camera footage per frame, while the element of ID: **#stream-container** assigned to the viewContainer variable has a div type.

```
[ ]: const view = document.getElementById('stream')
     const viewContainer = document.getElementById('stream-container')
```

The **onload** method is applied to the view variable in order to start the execution of the **detectObject()** callback function once the camera stream becomes available.

```
[ ]: view.onload = () => {
       detectObject();
     }
```

An empty array with variable name *children* is initialized to serve as the storage for the HTML elements to be injected or drawn on the webpage whenever an object is being detected from current camera stream frame.

```
[ ]: var children = [];
```

In the next block of code, the **detectObject()** will be defined which is the process mainly responsible for classifying or giving the class of the detected object in the camera footage based on the COCO dataset pre-trained models.

```
[ ]: function detectObject()
```

Inside the **detectObject()** function, a **detect** promise is created which will result in the execution of a function if an object that can be classified from the COCO models is successfully detected from the current camera stream frame.

```
[ ]: Model.detect(view).then(function (predictions)
```

The for loop in the following block of code removes each of the HTML elements that were stored in the *children* array when an object was detect detected previously. This will be replaced by the

graphical elements that will highlight and show the class of the new detected object in a stream frame.

```
[ ]: for (let i = 0; i < children.length; i++) {
       viewContainer.removeChild(children[i]);
     }
     children.splice(0);
```

Predictions can also be considered as the assumption of the existence of a person or object the data which have been classified in COCO SSD models in TensorFlow. A for loop will execute a series of actions for each exsistence of an instance that can be assigned a *class* in the current frame of the camera stream.

```
[ ]: for (let n = 0; n < predictions.length; n++)
```

For every section present in the current camera stream frame that is added to the set of instances that can be given a *prediction of its* ***class***, the if condition below will first determine if the **confidence score** for the class to be assigned is higher than 0.66 or 66% (pwede naman nating taasan itong minimum confidence score kung gusto niyo). As shown in the nested if block, if the class of the recognized object is equal to **'bottle'**, a message will be printed on the console to signal that an object with the class of bottle was detected.

```
[ ]: if (predictions[n].score > 0.66) {

       if (predictions[n].class == 'bottle') {
         console.log('A ' + predictions[n].class + ' was detected');
         websocket.send(JSON.stringify({'object':predictions[n].class}));
       }
```

The JavaScript program will then proceed to send this image processing result to the WebSocket server hosted on the ESP32-CAM. The data will be sent in the form of a JSON key-value pair with a **key** of **'object'**, and a **value** equal to the class of the detected or recognized object which in this case is **'bottle'**. This will enable the ESP32-CAM to transmit the result of the image processing to the wireless communication module connected to the Arduino UNO board in the system.

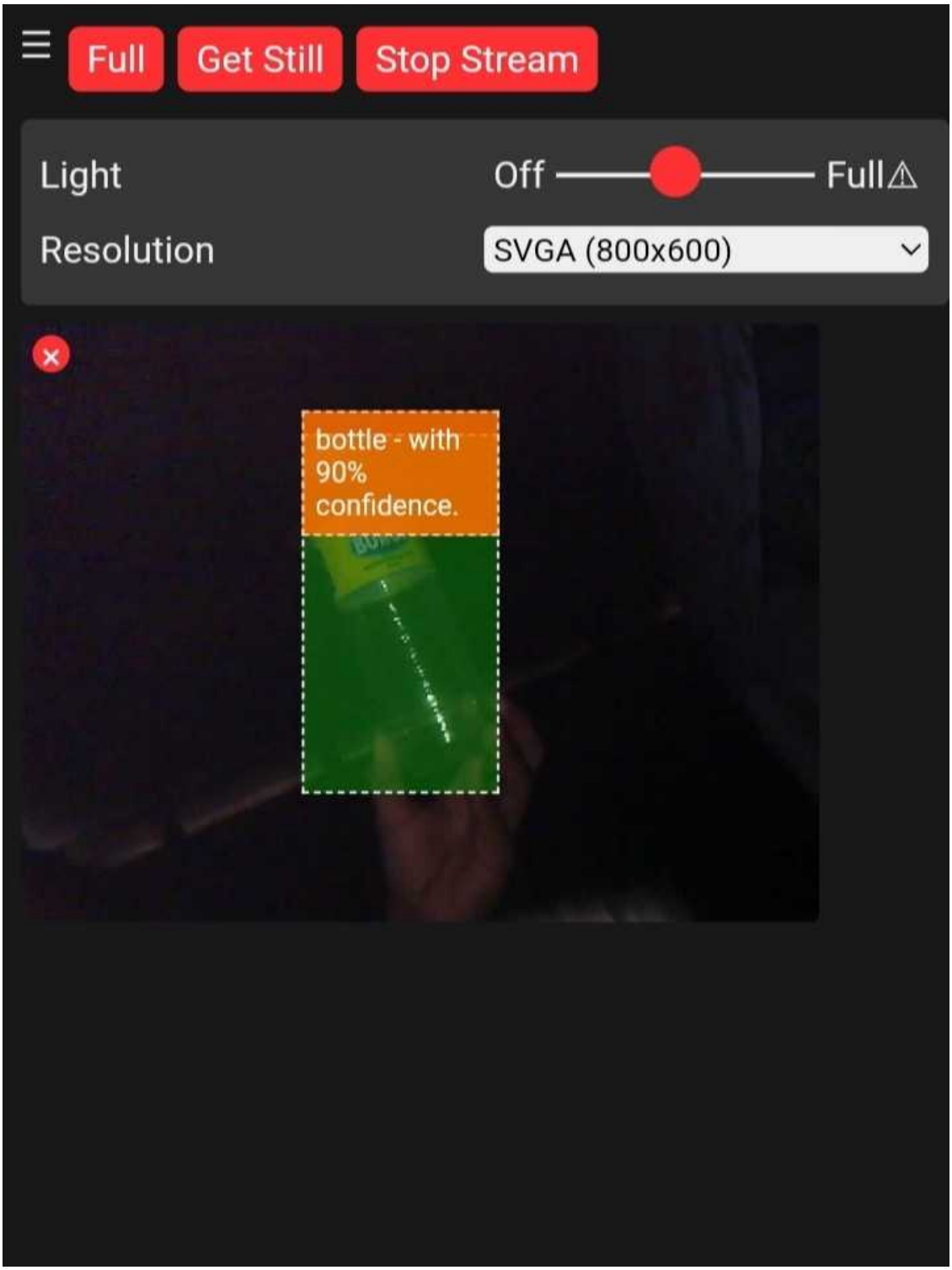
```
[ ]: websocket.send(JSON.stringify({'object':predictions[n].class}));
```

The next block of code simply utilizes the **createElement()** method of the HTML DOM to construct a p or paragraph HTML element that will show a text in the camera footage display stating the class of a detected object along with the confidence score in percentage.

```
[ ]: const p = document.createElement('p');
       p.innerText = predictions[n].class  + ' - with '
           + Math.round(parseFloat(predictions[n].score) * 100)
           + '% confidence.';
       p.style = 'margin-left: ' + predictions[n].bbox[0] + 'px; margin-top: '
           + (predictions[n].bbox[1] - 10) + 'px; width: '
           + (predictions[n].bbox[2] - 10) + 'px; top: 0; left: 0;';
```

A div HTML element referenced as highlighter in the JavaScript code is also drawn by the **createElement()** method to highlight the recognized objects from the camera view.

[ ]:
```
const highlighter = document.createElement('div');
        highlighter.setAttribute('class', 'highlighter');
        highlighter.style = 'left: ' + predictions[n].bbox[0] + 'px; top: '
            + predictions[n].bbox[1] + 'px; width: '
            + predictions[n].bbox[2] + 'px; height: '
            + predictions[n].bbox[3] + 'px;';
```



The created HTML elements will be appended to the div element referenced as viewContainer that serves as the wrapper to allow them to appear in the appropriate position in the camera stream display.

[ ]:
```
viewContainer.appendChild(highlighter);
        viewContainer.appendChild(p);
        children.push(highlighter);
        children.push(p);
```

The **requestAnimationFrame()** method from the Web API is continuously called to execute the **detectObject()** function for an indefinite amount of time and update the visual indicators of the detected object from the camera footage as long as this program is running on a browser.

[ ]:
```
window.requestAnimationFrame(detectObject);
```