Writing augmented reality applications using JSARToolKit

- Demo
- Setting up JSARToolKit
- Using getUserMedia to access the webcam
- Detecting markers
- Matrix mapping
- Three.js integration
- Summary
- References

This article is about using the JSARToolKit library with the WebRTC getUserMedia API to do augmented reality applications on the web. For rendering, I'm using WebGL due to the increased performance it offers. The end result of this article is a demo application that puts a 3D model on top of an augmented reality marker in webcam video.

JSARToolKit is an augmented reality library for JavaScript. It's an open source library released under the GPL and a direct port of the Flash FLARToolKit that I made for the Mozilla Remixing Reality demo. FLARToolKit itself is port of the Java NyARToolKit, which is a port of the C ARToolKit. Long way, but here we are.

JSARToolKit operates on canvas elements. As it needs to read the image off the canvas, the image needs to come from the same origin as the page or use CORS to get around same-origin policy. In a nutshell, set the `crossOrigin`-property on the image or video element you want to use as a texture to `''` or `'anonymous'`.

When you pass a canvas to JSARToolKit for analysis, JSARToolKit returns a list of AR markers found in the image and the corresponding transformation matrices. To draw a 3D object on top of a marker, you pass the transformation matrix to whatever 3D rendering library you're using so that

your object is transformed using the matrix. Then, draw the video frame in your WebGL scene and draw the object on top of that and you're good to go.

To analyze video using the JSARToolKit, draw the video on a canvas, then pass the canvas to JSARToolKit. Do this for every frame and you've got video AR tracking. JSARToolKit is fast enough on modern JavaScript engines to do this in realtime even on 640x480 video frames. However, the larger the video frame, the longer it takes to process. A good video frame size is 320x240, but if you expect to use small markers or multiple markers, 640x480 is preferable.

Demo

To view the webcam demo, you need to have WebRTC enabled in your browser (on Chrome, go the about:flags and enable MediaStream). You also need to print out the AR marker below. You can also try opening the marker image on your phone or tablet and showing it to the webcam.

AR marker.

AR marker.

Setting up JSARToolKit

The JSARToolKit API is quite Java-like, so you'll have to do some contortions to use it. The basic idea is that you have a detector object that operates on a raster object. Between the detector and the raster is a camera parameter object that transforms raster coordinates to camera coordinates. To get the detected markers from the detector, you iterate over them and copy their transformation matrices over to your code.

The first step is to create the raster object, the camera parameter object and the detector object.

```
// Create a RGB raster object for the 2D canvas.
// JSARToolKit uses raster objects to read image data.
// Note that you need to set canvas.changed = true on every frame.
var raster = new NyARRgbRaster_Canvas2D(canvas);

// FLARParam is the thing used by FLARToolKit to set camera parameters.
// Here we create a FLARParam for images with 320x240 pixel dimensions.
var param = new FLARParam(320, 240);

// The FLARMultiIdMarkerDetector is the actual detection engine for marker detection.
```

```
// It detects multiple ID markers. ID markers are special markers that encode a number.
var detector = new FLARMultiIdMarkerDetector(param, 120);

// For tracking video set continue mode to true. In continue mode, the detector
// tracks markers across multiple frames.
detector.setContinueMode(true);

// Copy the camera perspective matrix from the FLARParam to the WebGL library camera matrix.
// The second and third parameters determine the zNear and zFar planes for the perspective
matrix.
param.copyCameraMatrix(display.camera.perspectiveMatrix, 10, 10000);
```

Using getUserMedia to access the webcam

Next, I'm going to create a video element that's getting webcam video through the WebRTC APIs. For pre-recorded videos, just set the source attribute of the video to the video URL. If you're doing marker detection from still images, you can use a image element in much the same way.

As WebRTC and getUserMedia are still new emerging technologies, you need to feature detect them. For more details, check out Eric Bidelman's article on Capturing Audio & Video in HTML5.

```
var video = document.createElement('video');
video.width = 320;
video.height = 240;

var getUserMedia = function(t, onsuccess, onerror) {
  if (navigator.getUserMedia) {
    return navigator.getUserMedia(t, onsuccess, onerror);
  } else if (navigator.webkitGetUserMedia) {
    return navigator.webkitGetUserMedia(t, onsuccess, onerror);
  } else if (navigator.mozGetUserMedia) {
    return navigator.mozGetUserMedia(t, onsuccess, onerror);
  } else if (navigator.msGetUserMedia) {
    return navigator.msGetUserMedia(t, onsuccess, onerror);
  } else {
```

```
    onerror(new Error("No getUserMedia implementation found."));
  }
};

var URL = window.URL || window.webkitURL;
var createObjectURL = URL.createObjectURL || webkitURL.createObjectURL;
if (!createObjectURL) {
  throw new Error("URL.createObjectURL not found.");
}

getUserMedia({'video': true},
  function(stream) {
    var url = createObjectURL(stream);
    video.src = url;
  },
  function(error) {
    alert("Couldn't access webcam.");
  }
);
```

Detecting markers

Once we have the detector running a-ok, we can start feeding it images to detect AR matrices. First draw the image onto the raster object canvas, then run the detector on the raster object. The detector returns the number of markers found in the image.

```
// Draw the video frame to the raster canvas, scaled to 320x240.
canvas.getContext('2d').drawImage(video, 0, 0, 320, 240);

// Tell the raster object that the underlying canvas has changed.
canvas.changed = true;

// Do marker detection by using the detector object on the raster object.
// The threshold parameter determines the threshold value
// for turning the video frame into a 1-bit black-and-white image.
//
var markerCount = detector.detectMarkerLite(raster, threshold);
```

The last step is to iterate through the detected markers and get their transformation matrices. You use the transformation matrices for putting 3D objects on top of the markers.

```
// Create a NyARTransMatResult object for getting the marker translation matrices.
var resultMat = new NyARTransMatResult();

var markers = {};

// Go through the detected markers and get their IDs and transformation matrices.
for (var idx = 0; idx &lt; markerCount; idx++) {
  // Get the ID marker data for the current marker.
  // ID markers are special kind of markers that encode a number.
  // The bytes for the number are in the ID marker data.
  var id = detector.getIdMarkerData(idx);

  // Read bytes from the id packet.
  var currId = -1;
  // This code handles only 32-bit numbers or shorter.
  if (id.packetLength <= 4) {
    currId = 0;
    for (var i = 0; i &lt; id.packetLength; i++) {
      currId = (currId << 8) | id.getPacketData(i);
    }
  }

  // If this is a new id, let's start tracking it.
  if (markers[currId] == null) {
    markers[currId] = {};
  }
  // Get the transformation matrix for the detected marker.
  detector.getTransformMatrix(idx, resultMat);

  // Copy the result matrix into our marker tracker object.
  markers[currId].transform = Object.asCopy(resultMat);
}
```

Matrix mapping

Here's the code to copy JSARToolKit matrices over to glMatrix matrices (which are 16-element FloatArrays with the translation column in the last four elements). It works by magic (read: I don't know how ARToolKit matrices are setup. Inverted Y-axis is my guess.) Anyway, this bit of sign-reversing voodoo makes a JSARToolKit matrix work the same as a glMatrix.

To use the library with another library, such as Three.js, you need to write a function that converts the ARToolKit matrices to the library's matrix format. You also need to hook into the FLARParam.copyCameraMatrix method. The copyCameraMatrix method writes the FLARParam perspective matrix into a glMatrix-style matrix.

```
function copyMarkerMatrix(arMat, glMat) {
  glMat[0] = arMat.m00;
  glMat[1] = -arMat.m10;
  glMat[2] = arMat.m20;
  glMat[3] = 0;
  glMat[4] = arMat.m01;
  glMat[5] = -arMat.m11;
  glMat[6] = arMat.m21;
  glMat[7] = 0;
  glMat[8] = -arMat.m02;
  glMat[9] = arMat.m12;
  glMat[10] = -arMat.m22;
  glMat[11] = 0;
  glMat[12] = arMat.m03;
  glMat[13] = -arMat.m13;
  glMat[14] = arMat.m23;
  glMat[15] = 1;
}
```

Three.js integration

Three.js is a popular JavaScript 3D engine. I'm going to go through how to use JSARToolKit output in Three.js. You need three things: a full screen quad with the video image drawn onto it, a camera with the FLARParam perspective matrix and an object with marker matrix as its transform. I'll walk you through the integration in the code below.

```
// I'm going to use a glMatrix-style matrix as an intermediary.
// So the first step is to create a function to convert a glMatrix matrix into a Three.js
Matrix4.
```

```
THREE.Matrix4.prototype.setFromArray = function(m) {
  return this.set(
    m[0], m[4], m[8], m[12],
    m[1], m[5], m[9], m[13],
    m[2], m[6], m[10], m[14],
    m[3], m[7], m[11], m[15]
  );
};

// glMatrix matrices are flat arrays.
var tmp = new Float32Array(16);

// Create a camera and a marker root object for your Three.js scene.
var camera = new THREE.Camera();
scene.add(camera);

var markerRoot = new THREE.Object3D();
markerRoot.matrixAutoUpdate = false;

// Add the marker models and suchlike into your marker root object.
var cube = new THREE.Mesh(
  new THREE.CubeGeometry(100,100,100),
  new THREE.MeshBasicMaterial({color: 0xff00ff})
);
cube.position.z = -50;
markerRoot.add(cube);

// Add the marker root to your scene.
scene.add(markerRoot);

// Next we need to make the Three.js camera use the FLARParam matrix.
param.copyCameraMatrix(tmp, 10, 10000);
camera.projectionMatrix.setFromArray(tmp);



// To display the video, first create a texture from it.
var videoTex = new THREE.Texture(videoCanvas);

// Then create a plane textured with the video.
```

```
var plane = new THREE.Mesh(
  new THREE.PlaneGeometry(2, 2, 0),
  new THREE.MeshBasicMaterial({map: videoTex})
);

// The video plane shouldn't care about the z-buffer.
plane.material.depthTest = false;
plane.material.depthWrite = false;

// Create a camera and a scene for the video plane and
// add the camera and the video plane to the scene.
var videoCam = new THREE.Camera();
var videoScene = new THREE.Scene();
videoScene.add(plane);
videoScene.add(videoCam);

...

// On every frame do the following:
function tick() {
  // Draw the video frame to the canvas.
  videoCanvas.getContext('2d').drawImage(video, 0, 0);
  canvas.getContext('2d').drawImage(videoCanvas, 0, 0, canvas.width, canvas.height);

  // Tell JSARToolKit that the canvas has changed.
  canvas.changed = true;

  // Update the video texture.
  videoTex.needsUpdate = true;

  // Detect the markers in the video frame.
  var markerCount = detector.detectMarkerLite(raster, threshold);
  for (var i=0; i&lt;markerCount; i++) {
    // Get the marker matrix into the result matrix.
    detector.getTransformMatrix(i, resultMat);

    // Copy the marker matrix to the tmp matrix.
    copyMarkerMatrix(resultMat, tmp);
```

```
    // Copy the marker matrix over to your marker root object.
    markerRoot.matrix.setFromArray(tmp);
  }

  // Render the scene.
  renderer.autoClear = false;
  renderer.clear();
  renderer.render(videoScene, videoCam);
  renderer.render(scene, camera);
}
```

Summary

In this article we went through the basics of JSARToolKit. Now you are ready to build your own webcam-using augmented reality applications with JavaScript.

Integrating JSARToolKit with Three.js is a bit of a hassle, but it is certainly possible. I'm not 100% certain if I'm doing it right in my demo, so please let me know if you know of a better way of achieving the integration. Patches are welcome :)

References

- JSARToolKit
- Magi
- Three.js

□□□ #3
□ □□□ □□□ 2026-10-31 22:28:31 UTC
□ cocobook-sync □□□ 2026-11-04 17:38:30 UTC