Co-funded by the European Union

META QUEST

AR/VR APP DEVELOPMENT

KA220-VET - Cooperation partnerships in vocational education and training
Project Title: ***AI tools for VET schools***
Document Date: February 2026

Unity WebXR

Author: Bojan Ćirić,
co-authors:
Aleksandar Madić,
Boban Blagojević)

Introduction

The Meta Quest 3 marks a pivotal shift from pure Virtual Reality (VR) to high-fidelity Mixed Reality (MR). By leveraging high-resolution color passthrough and a significantly more powerful Snapdragon XR2 Gen 2 chipset, it allows digital content to coexist seamlessly with the physical world. With its slimmer "pancake" optics and 4K+ Infinite Display, it is currently the most accessible powerhouse for both consumers and developers.

APPLICATION DEVELOPMENT IN UNITY

Unity remains the industry leader for building "native" Quest applications. It provides the deepest access to the headset's hardware, making it the go-to for high-end games and complex enterprise simulations.

Key Development Pillars:

- The XR Interaction Toolkit (XRI): A high-level library that handles basic interactions like grabbing, hovering, and UI pointing across different controllers.
- Meta XR All-in-One SDK: This specific package grants access to Meta's unique features, such as Scene Discovery (understanding where your floor and walls are) and Spatial Anchors (saving virtual objects in a permanent physical location).
- Optimization: Since the Quest 3 is a mobile-based chipset, developers utilize the Universal Render Pipeline (URP) and techniques like foveated rendering to maintain high frame rates (72Hz–120Hz).

WEBXR: THE BROWSER-BASED ALTERNATIVE

WebXR allows developers to create immersive experiences that run directly in the Meta Quest Browser. It is built on standard web technologies, making "VR on the web" as easy to access as a website.

Why Choose WebXR?

- Frictionless Access: Users don't need to download large files from the Meta Store; they simply click a URL and hit "Enter VR."
- Frameworks: It utilizes powerful JavaScript libraries like Three.js, A-Frame (HTML-based), or Babylon.js.
- Cross-Platform: A single WebXR app can often run on a Quest 3, an Android phone (AR mode), or a desktop PC.

Meta QUEST

WebXR

Introduction to Meta Quest 3 and Mixed Reality

The Meta Quest 3 utilizes high-resolution Color Passthrough technology. Unlike traditional VR, where the world is entirely virtual, **Mixed Reality (MR)** uses onboard cameras to project the real environment and then overlays **WebGL** elements on top. Your code utilizes the immersive-ar mode, which is the core of this spatial experience.

System Architecture (The Web Stack)

The application functions as a "sandwich" of layers:

- **Layer 1 (Hardware)**: Quest 3 sensors, depth projectors, and RGB cameras.
- **Layer 2 (Browser)**: Meta Quest Browser (Chromium-based with WebXR support).
- **Layer 3 (API)**: WebXR Device API, which acts as the bridge between hardware and code.
- **Layer 4 (Logic)**: Your JavaScript logic using Three.js for rendering and TensorFlow.js for vision.

Library Analysis

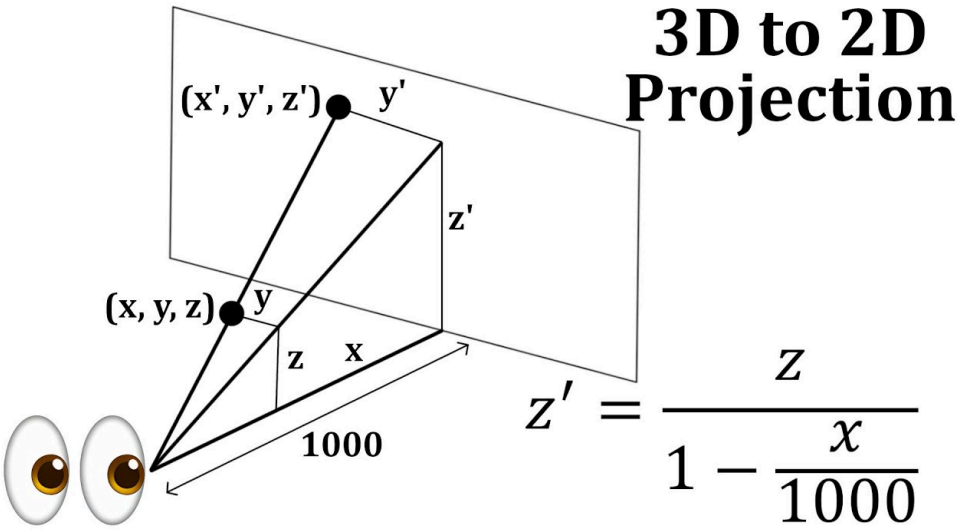
Three.js (The Render Engine)
Three.js handles the heavy lifting of WebGL. It manages the Scene, Camera, and Renderer. In your code, the renderer is set to alpha: true, which is vital for AR as it allows the "empty" parts of the browser to become a window into the real world.

TensorFlow.js & COCO-SSD
COCO-SSD (Common Objects in Context - Single Shot MultiBox Detector) is a model trained to recognize 80 classes of objects. The beauty of TF.js is its **WebGL backend**, which ensures that AI calculations are performed on the Quest's GPU rather than the CPU, preventing the device from overheating.

Mathematical Projection (2D to 3D)

This is the most technical part of the script. The AI model returns a bbox (bounding box) in pixels (e.g., x=100, y=200).
The detectionTo3D function performs un-projection:

1. **Normalization**: It converts screen coordinates to a range of -1 to $+1$.
2. **FOV Calculation**: It factors in the camera's Field of View.
3. **Vector Ray**: It creates a direction vector from the user's head position toward the object.
4. **Placement**: It places the 3D label on that vector at a defined distance (DIST).



Implementing Hit-Testing

Hit-testing allows the app to "fire" an invisible ray (raycast) into real space to detect intersections with floors or tables. When the ***hitTestSource*** returns a result, the cursor (green ring) is snapped to that pose (position and orientation).

AI Pipeline: Detection and Labeling

In the code, detection is not performed every frame (which would lag the headset), but every $2000ms$ (DETECT_INTERVAL_MS).

- **Label Sprite**: Since Three.js cannot render standard HTML fonts directly in a 3D scene, we use a CanvasTexture. We "draw" the text on an invisible 2D HTML canvas and apply it as a texture to a 3D Sprite that always faces the user (billboarding).

The Necessity of HTTPS

WebXR and camera access (getUserMedia) are classified as "Powerful Features" by browser vendors. They will only work in a **Secure** Context.

- **Localhost Exception**: You can test on your PC using http://localhost, but as soon as you try to access that server from your Quest 3 headset, the browser will block XR features because it sees an insecure network IP (e.g., http://192.168.1.10).
- **The Solution**: You must use a **Tunneling Service** or **Secure Hosting**.

Meta Quest Link & Developer Mode

To run and debug your code efficiently, your Quest 3 must be recognized as a developer device.

Step-by-Step Activation:

1. **Developer Account**: Register at **dashboard.oculus.com**. You will need to create an "Organization" (it can be any name).
2. Mobile App: Open the Meta Quest app on your phone, go to **Menu > Devices > Headset Settings > Developer Mode**, and toggle it **ON**.
3. **The Link**: Connect your Quest 3 to your PC using a high-quality USB-C 3.0 cable or via Air Link (High-speed Wi-Fi 6).

The Three.js Foundation (The Boilerplate)

Before entering AR, we must initialize a standard 3D environment. However, for Quest 3, two settings are non-negotiable:

- **Alpha & Antialias**:

```javascript
const renderer = new THREE.WebGLRenderer({ antialias: true, alpha: true });
```

- **XR Activation**:

```
enderer.xr.enabled = true;
```

This tells Three.js to listen for the Quest's head-tracking data ($6DOF$) and apply it to the `camera` object automatically.

Managing the AR Session (The Lifecycle)

The "Enter AR" button triggers **navigator.xr.requestSession**. This is where we define what "superpowers" our app needs from the Quest 3 hardware.

Required & Optional Features:

- **local-floor**: This tells the Quest to set the $Y=0$ coordinate at the actual physical floor level.
- **hit-test**: Enables the ability to raycast against real-world geometry.
- **plane-detection**: Requests the "Semantic" data (knowing which mesh is a 'table' vs. a 'wall').

```javascript
const session = await navigator.xr.requestSession('immersive-ar', {
    requiredFeatures: ['local-floor'],
    optionalFeatures: ['hit-test', 'plane-detection']
});
```

Meta QUEST

AR + AI Object Detection Demo

Project Overview

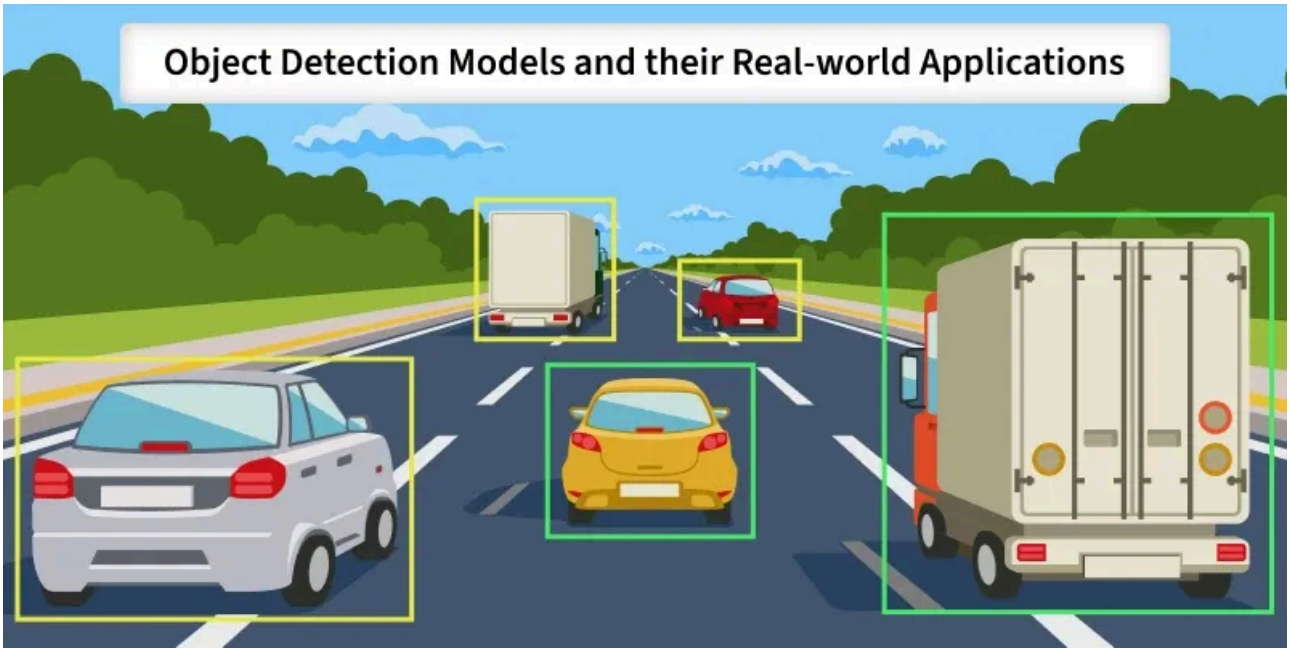
This is a WebXR Augmented Reality application for Meta Quest 3 that combines:

- **Passthrough AR** — the real room is visible through the headset cameras
- **Hit-test interaction** — a virtual cursor that snaps to real-world surfaces
- **AI object detection** — TensorFlow.js recognizes objects in the camera feed and displays labels in 3D space
- **Scene Understanding** — WebXR Plane Detection visualizes detected surfaces (floor, walls, table)

Live URL: http://92.113.18.92/

Single file:

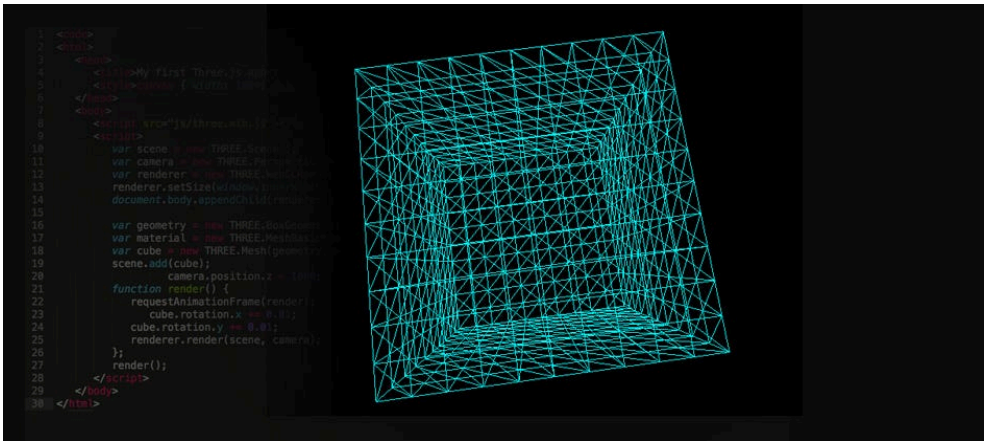
index.html



Project Architecture

📄 index.html (all-in-one)
│
├── HTML → canvas + UI button + hidden video element
├── CSS → transparent background, overlay UI
└── JavaScript
  ├── Three.js → 3D rendering engine (scene, camera, materials)
  ├── WebXR API → AR session, hit-test, plane detection
  └── TF.js → AI inference (coco-ssd model)

Why pure Three.js (no A-Frame)?
During development it was discovered that A-Frame has its own internal render loop that conflicts with an explicit **immersive-ar** WebXR session. A-Frame starts an **immersive-vr** session (opaque, no passthrough) instead of **immersive-ar**. Switching to pure **Three.js** gave us direct control over both the session type and the render loop.



Technology Stack

Technology	Version	Role
Three.js	0.157.0	3D rendering, geometry, materials
WebXR Device API	Browser-native	AR session, hit-test, plane detection
TensorFlow.js	4.15.0	In-browser ML inference engine
COCO-SSD	2.2.3	Pre-trained object detection model
getUserMedia API	Browser-native	Camera stream for TF.js analysis

Application Flow

Page loads
→ TF.js model (coco-ssd) loads asynchronously (~6MB)
→ Check: navigator.xr.isSessionSupported('immersive-ar')
→ "Enter AR" button becomes active

User clicks "Enter AR"
→ Request: navigator.xr.requestSession('immersive-ar', {...})
→ Quest enables Passthrough (camera visible through headset)
→ Simultaneously: getUserMedia() → camera stream for TF.js
→ renderer.setAnimationLoop() starts (XR render loop)

Every frame (~72fps on Quest 3)
├── scene.background = null (ensures transparency)
├── Hit-test → updates cursor position
├── Plane Detection → updates surface visualization
├── Every 2s → runDetection() → AI inference
└── renderer.render(scene, camera)

Detailed Code Explanation

1. HTML Structure (lines 1–78)

```HTML
<video id="tfvideo" playsinline muted autoplay></video>
```

A hidden <**video**> element receiving the ***getUserMedia***() stream. TensorFlow.js uses it as input for inference — it is never displayed to the user, only analyzed.

```
html
<div id="ui"> ... <button id="ar-button"> </div>
```

An overlay UI layer floating above the **Three.js** canvas. ***pointer-events: none*** on the container but ***pointer-events: all*** on the button only — so the overlay doesn't block 3D interactions.

2. Three.js Initialization (lines 84–109)

```
javascript
const renderer = new THREE.WebGLRenderer({ antialias: true, alpha: true });
renderer.xr.enabled = true;
```

alpha: true creates a WebGL canvas with a transparent alpha channel — this is a prerequisite for passthrough. ***xr.enabled = true*** activates **Three.js**'s built-in WebXR integration.

```
javascript
const camera = new THREE.PerspectiveCamera(70, aspect, 0.01, 20);
scene.add(camera);
```

The camera is **added to the scene**. This is important because objects attached to the camera (child entities) need to be in the scene hierarchy.

```
javascript
const cursorGeo = new THREE.RingGeometry(0.04, 0.06, 32);
cursorGeo.rotateX(-Math.PI / 2);
```

The ring geometry is rotated -90° on the X axis so it lies flat horizontally on detected surfaces. Without this rotation it would stand vertically.

3. makeLabel() — Text Sprite (lines 113–138)

```
javascript
function makeLabel(text, color, bgColor) {
  const canvas = document.createElement('canvas'); // 512x128 px
  // Draws rounded rect + text
  const texture = new THREE.CanvasTexture(canvas);
  const sprite  = new THREE.Sprite(material);
  sprite.scale.set(0.6, 0.15, 1); // 0.6m wide in 3D space
  return sprite;
}
```

THREE.Sprite is an object that always faces the camera (**billboarding**) — ideal for labels in 3D space. It is drawn on an HTML canvas, converted to a ***CanvasTexture***, and applied to a ***SpriteMaterial***.
depthTest: false ensures the label is always visible even if it is geometrically "behind" another 3D object.

4. detectionTo3D() — 2D → 3D Projection (lines 140–163)

This is the mathematical core of the AI-AR integration.

```
javascript
// Normalize bbox center to [-0.5, 0.5]
const nx = (bbox.x + bbox.width/2)  / videoWidth  - 0.5;
const ny = (bbox.y + bbox.height/2) / videoHeight - 0.5;
// Apply camera FOV (70° horizontal)
const fovH = 70 * Math.PI / 180;
const dir  = new THREE.Vector3(
  Math.tan(nx * fovH),
  Math.tan(-ny * fovV), // flip Y (image top-down, WebGL bottom-up)
  -1                    // forward in camera space (Three.js uses -Z)
).normalize();
// Rotate into world space using the current XR camera orientation
dir.applyQuaternion(camera.quaternion);
// World position = camera position + direction × distance
return camera.position.clone().addScaledVector(dir, placeDist);
```

*Example: If the AI detects a chair in the left third of the video, **nx ≈ -0.17**. This converts to a negative angle (left of the gaze axis). The label is placed 1.8m in front of the camera in that direction.*

5. makeBox3D() and bbox2dSize3D() — 3D Bounding Boxes (lines 165–182)

```
javascript
function makeBox3D(w, h, colorHex) {
  const edges = new THREE.EdgesGeometry(new THREE.BoxGeometry(w, h, 0.01));
  return new THREE.LineSegments(edges, material);
}
```

EdgesGeometry + LineSegments *renders only the edges of a box — the effect of a thin wireframe border with no filled surface.*

```
javascript
function bbox2dSize3D(bboxW, bboxH, videoW, videoH, dist) {
  const fovH   = 70 * Math.PI / 180;
  const totalW = 2 * dist * Math.tan(fovH / 2); // total visible width at given dist
  return {
    w: (bboxW / videoW) * totalW,  // bbox fraction → meters
    h: (bboxH / videoH) * totalH
  };
}
```

The ***totalW*** *formula is standard perspective projection:* ***at distance d****, the visible width depends on the FOV. From this we derive how many meters correspond to the pixels of the bounding box.*

6. runDetection() — AI Inference Pipeline (lines 205–247)

```
javascript
async function runDetection() {
  const predictions = await cocoModel.detect(tfVideo);
  // Clear old labels and boxes
  activeLabels.forEach(({ sprite, box }) => { scene.remove(sprite); scene.remove(box); });
  activeLabels = [];
  predictions.forEach(pred => {
    if (pred.score < 0.5) return; // Confidence threshold: 50%
    // ...create sprite + box...
    activeLabels.push({ sprite, box, expireAt: Date.now() + 4000 });
  });
}
```

cocoModel.detect(tfVideo) *accepts a <video> element directly — TF.js internally reads pixel data from the current video frame.*
pred.bbox *format:* ***[x, y, width, height]*** *in pixels.*
Each detection creates a ***JS(sprite, box)*** *pair that lives for 4 seconds.*

7. WebXR Session — Key Details (lines 274–285)

```
javascript
const session = await navigator.xr.requestSession('immersive-ar', {
  requiredFeatures: ['local-floor'],
  optionalFeatures: ['hit-test', 'plane-detection']
});
renderer.xr.setReferenceSpaceType('local-floor');
await renderer.xr.setSession(session);
```

Why **immersive-ar** and not **immersive-vr**?

- **immersive-vr** = opaque black background (VR helmet experience)
- **immersive-ar** = passthrough camera + virtual content overlaid on top

local-floor reference space fixes the coordinate system to the room floor — **y=0** is at floor level.
hit-test and **plane-detection** are optional because they are not supported on all devices and browser versions — declaring them as optional prevents the session from failing if they're unavailable

8. Render Loop (lines 335–395)

```
javascript
renderer.setAnimationLoop(function(time, frame) {
  scene.background = null;              // Ensures transparency every frame
  renderer.setClearColor(0x000000, 0); // Alpha = 0 (fully transparent)
  // ...hit-test, plane detection, AI...
  renderer.render(scene, camera);
});
```

Why **setAnimationLoop** and not **requestAnimationFrame**?
The **Three.js** XR render loop must be integrated with the WebXR frame callback. **renderer.setAnimationLoop()** automatically binds to the XR session when **renderer.xr.enabled = true**. The alternative (manually calling **session.requestAnimationFrame()**) requires calling **renderer.render()** manually but risks missing the correct XR view matrices that **Three.js** applies internally.
scene.background = null must be called **every frame** because Three.js may reset this value internally during certain operations.

9. Plane Detection (lines 355–378)

```
javascript
session.detectedPlanes.forEach(plane => {
  if (!detectedPlanes.has(plane)) {
    // New surface detected — create a mesh
    const label = plane.semanticLabel; // 'floor', 'wall', 'table'...
    const mesh = new THREE.Mesh(PlaneGeometry, transparentMaterial);
    scene.add(mesh);
    detectedPlanes.set(plane, mesh); // store reference
  }
  // Every frame, update the surface pose
  const pose = frame.getPose(plane.planeSpace, xrRefSpace);
  mesh.position.copy(pose.transform.position);
  mesh.quaternion.copy(pose.transform.orientation);
});
```

plane.planeSpace is an XRSpace that tracks the physical surface. **frame.getPose()** returns its pose in the chosen reference space.
The **detectedPlanes** map **(Map<XRPlane, THREE.Mesh>)** prevents creating duplicate meshes for the same surface across frames.

Known Limitations

Limitation	Reason
AI labels are not pixel-perfect aligned with objects	The Quest passthrough cameras and getUserMedia deliver different streams with different FOV and optical calibration
Plane detection requires Quest Room Setup to be completed	The headset must have previously scanned the room
getUserMedia may be denied on some devices	Depends on browser permissions
AI inference is not real-time (runs every 2s)	coco-ssd is a relatively heavy model; lighter alternatives (MobileNet SSD) would give higher throughput

Source code

https://arvr.cirasoft.net