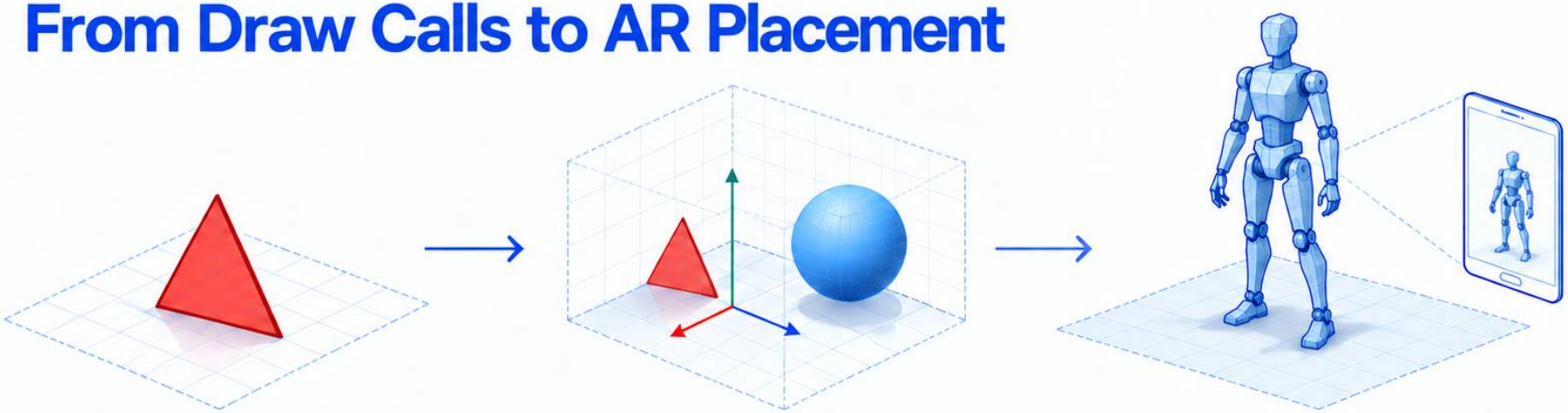
Virtual Worlds in the Age of AI

CG/XR Lab

From Draw Calls to AR Placement



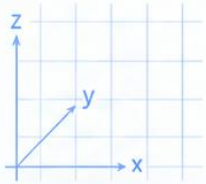
Draw

Compose

Place

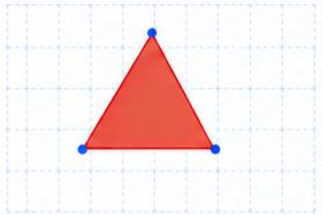
From a triangle to a model placed in your room

WebGL, Three.js, a GLB asset, and WebXR show a continuous set of responsibilities — not separate worlds.



1 LOW-LEVEL EXAMPLE: WebGL

Triangle data → GPU draw



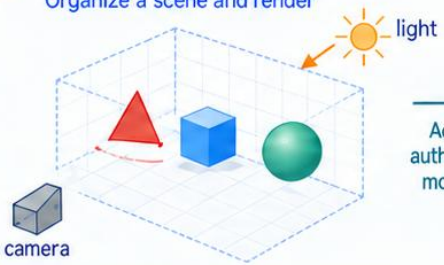
```
gl.bindBuffer(gl.ARRAY_BUFFER, buf);
gl.useProgram(program);
gl.drawArrays(gl.TRIANGLES, 0, 3);
```

- Expose data, shaders, state, and draw calls.
- You manage GPU resources.
- WebGL is one example; other low-level APIs include OpenGL and WebGPU.

Add scene and objects →

2 HIGHER-LEVEL EXAMPLE: Three.js

Organize a scene and render



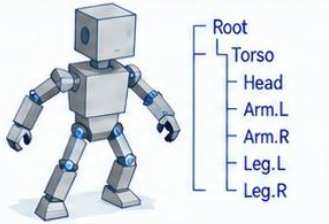

```
const scene = new THREE.Scene();
scene.add(triangle, cube, sphere);
renderer.render(scene, camera);
```

- Higher-level graphics library/API.
- Manage a scene, camera, lights, and renderable objects.
- Three.js is one example; Babylon.js is another higher-level library.

Add authored model →

3 AUTHORED ASSET: GLB

Model, materials, hierarchy, animation



animated model (GLB)

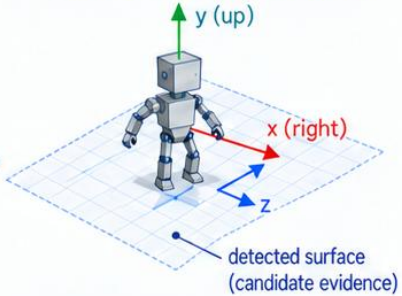
```
const loader = new GLTFLoader();
loader.load('model.glb', (gltf) => {
   scene.add(gltf.scene);
});
```

- GLB supplies geometry, materials, hierarchy, and animation data.
- GLTFLoader loads the model; Three.js can play its animation data.

Add tracked spatial view →

4 SPATIAL RUNTIME: WebXR

Place the model on a real surface



- WebXR adds a device-driven view and evidence about a detected surface.
- Uses a local reference space and explicit actions (e.g., tap to place).
- Immersive AR is a capability-dependent demonstration.
- Desktop is the common practice path.

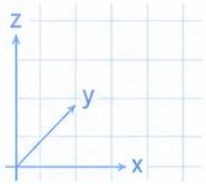
KEY TAKEAWAYS

1. WebGL is one low-level graphics API used here to expose data, shaders, state, and draw calls.
2. Three.js is one higher-level graphics library/API used here to organize scenes and renderable objects.
3. A GLB supplies authored geometry, materials, hierarchy, and animation data.
4. WebXR adds a device-driven view and evidence about a detected surface.
5. Desktop is the common practice path; immersive AR is a capability-dependent demonstration.

TOOLS ARE EXAMPLES; RENDERING RESPONSIBILITIES TRANSFER.

- WebGL and Three.js are representative graphics API examples at different levels.
- The same rendering responsibilities transfer to other APIs and engines (e.g., OpenGL, WebGPU, Babylon.js).
- This lab builds practical skills while leaving substantial original project work open.

Keep the rendering contract visible as abstractions change



LOW-LEVEL GRAPHICS APIs

Expose GPU resources and commands.

This lab: WebGL

Other examples: OpenGL · WebGPU

HIGHER-LEVEL GRAPHICS APIs

Organize objects and rendering.

This lab: Three.js

Another example: Babylon.js

This lab: one example stack

From scene objects to pixels (one of many possible stacks).

Three.js WebGLRenderer
Scene, materials, transforms

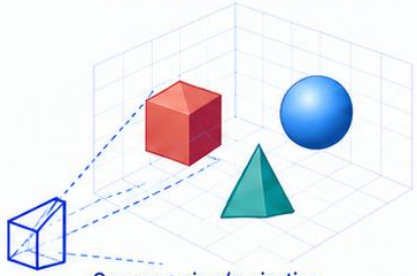
↓

WebGL
GPU resources and commands

↓

GPU
Rasterization and shading

Your scene (example)
Geometry + materials + transforms



Camera + view/projection (from app or XR runtime)

WebXR: tracked views + input + spatial evidence

Supplies poses and frame timing. Not a graphics renderer.

Poses + frame timing + input

App: camera views + frame timing + input

Use with WebGL or Three.js

The essentials remain the same

Geometry	Surface response	Transforms	Submission	Frame timing
vertices, indices	materials, textures	model · view · projection	draw calls	per-frame updates

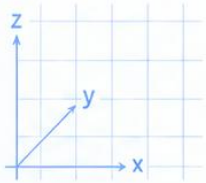
Edit → Run → Observe → Explain → Reset

Predict a change → **Edit one** parameter → **Run** in browser → **Observe** result → **Explain** what happened → **Reset** and try again

Categories describe responsibilities, not one mandatory toolchain.

WebGL and Three.js are representative low-level and higher-level graphics APIs. Other tools (OpenGL, WebGPU, Babylon.js) fit the same responsibilities.

WebGL turns typed arrays into primitives



Data → GPU → Primitives → Pixels
A low-level graphics API example

1 Clearing
Clearing establishes the starting framebuffer; it does not draw geometry.

2 Typed array → buffer
A typed array supplies positions; the bound buffer stores them for rendering.

3 Attribute setup
Attribute setup explains how the vertex shader reads that buffer.

4 Primitive mode
Primitive mode determines how vertices become points, lines, or triangles.

5 Uniform
A uniform can supply one shared color to all fragments in a draw.

WebGL pipeline (three vertices → triangle)

1 Position array (typed array)

```
const positions =
  new Float32Array([
  -0.6, -0.4,  0.0,
   0.6, -0.4,  0.0,
   0.0,  0.6,  0.0
]);
```

3 vertices (x, y, z)

2 Buffer + attribute layout

GPU buffer (ARRAY_BUFFER)

```
gl.bindBuffer(...)
gl.bufferData(...)
gl.vertexAttribPointer(...)
```

Tells the GPU how to read positions for each vertex.

3 Vertex shader (per vertex)

```
attribute vec3 aPosition;
...
gl_Position =
  vec4(aPosition, 1.0);
```

Transforms each vertex to clip space.

Rasterization (primitives → fragments)

Vertices become triangles (rasterized into fragments).

4 Fragment shader (per fragment)

```
uniform vec4 uColor;
...
gl_FragColor = uColor;
```

Computes a color for each fragment (e.g., one shared color).

5 Framebuffer (pixels)

Fragments write to the framebuffer (pixels on screen).

Key WebGL code (from Example 4)

```
1   gl.bindBuffer(gl.ARRAY_BUFFER, posBuffer);
2   gl.bufferData(gl.ARRAY_BUFFER, positions, gl.STATIC_DRAW);
3   gl.drawArrays(gl.TRIANGLES, 0, 3);
```

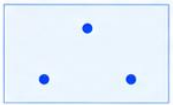
A single draw call uses 3 vertices (0, 1, 2) and the TRIANGLES mode to create one triangle.

Primitive mode comparison (same three vertices)

Clear (no geometry)

clear ≠ draw
`gl.clear(...)` sets the starting framebuffer.

Points (`gl.POINTS`)



3 vertices → 3 points

Lines (`gl.LINES`)

3 vertices → 1 line segment (0–1); vertex 2 unused

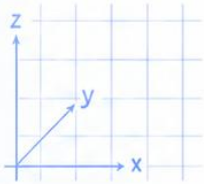
Triangles (`gl.TRIANGLES`)

3 vertices → one triangle (0–1–2)

Takeaway: A small set of steps turns your typed array data into pixels on the screen.

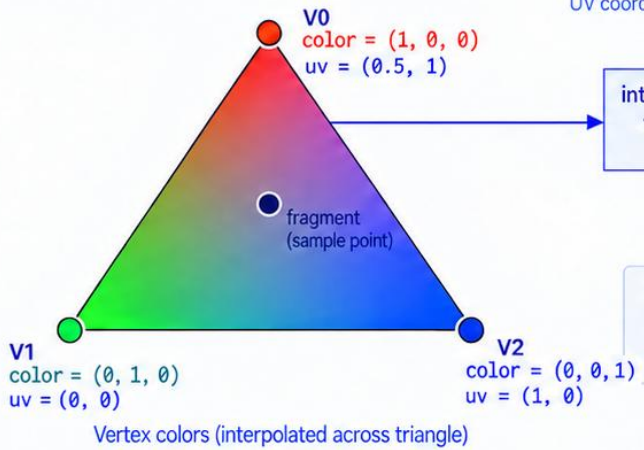
Clearing does not draw geometry.

Fragment appearance follows interpolated data



Triangle with vertex colors and UVs

Attributes are specified per vertex.



Interpolated UV at fragment

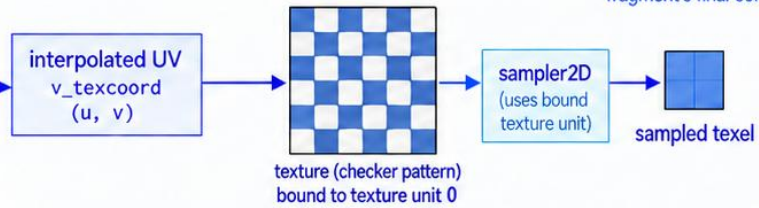
Rasterizer interpolates per-vertex UV coordinates.

Texture unit + sampler

The sampler reads from a texture bound to a texture unit.

Sampled texel

The sampled texel contributes to the fragment's final color.



Fragment shader (excerpt)

```
vec4 sample = texture2D(texture, v_texcoord);
```

KEY POINTS

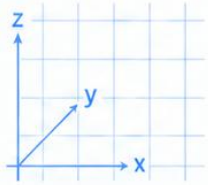
1. A per-vertex color varies across the triangle after rasterization.
2. UV coordinates locate samples in a texture; they are not world positions.
3. The sampler refers to a texture unit with a bound texture.
4. Attribute data, interpolation, and fragment operations jointly determine appearance.
5. Changing UVs changes mapping while leaving the triangle's positions unchanged.

Same geometry · different mapping

Change the UV coordinates (not the positions) to get a different texture mapping.

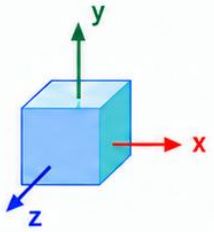
Demo cue: Alter a UV coordinate in example 6 and predict the mapping first.

Object and camera transforms meet in clip space



1. LOCAL (object space)

Vertices in model coordinates

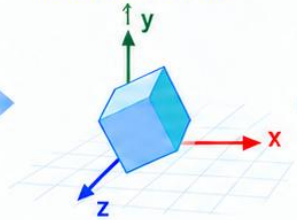


p_local

object origin and axes

2. WORLD (world space)

Model transform M moves to world coordinates

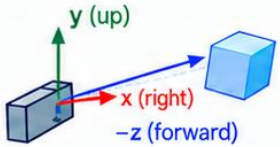


$p_{world} = M \times p_{local}$

object in world scene

3. VIEW (camera space)

View matrix V expresses world positions relative to camera

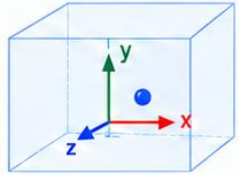


$p_{view} = V \times p_{world}$

positions relative to camera

4. CLIP (clip space)

Projection P → homogeneous clip coordinates



homogeneous clip coordinates

$-w \le x, y, z \le w$

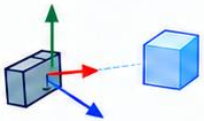
$p_{clip} = P \times p_{view}$

homogeneous coordinates

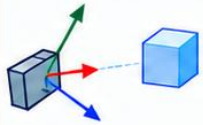
Camera up direction example

Changing camera up changes the view, not the object's geometry.

standard up = [0, 1, 0]



tilted up = [1, 1, 0]



Same object geometry; a different camera view.

p_{local} → **M** (M: object pose (model matrix)) → p_{world} → **V** (V: inverse camera pose (view matrix)) → p_{view} → **P** (P: projection (projection matrix)) → p_{clip}

$$p_{clip} = P \times V \times M \times p_{local}$$

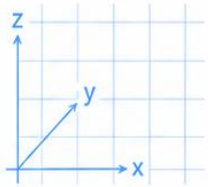
→ clip → divide by w → viewport → screen pixels

Key points

1. Model transforms move object-local positions into world coordinates.
2. The view matrix expresses world positions relative to the camera.
3. Projection maps camera-space positions to homogeneous clip coordinates.
4. With column vectors: $p_{clip} = P \times V \times M \times p_{local}$; perspective division follows clipping.
5. Changing the camera's up direction changes the view, not the object's geometry.

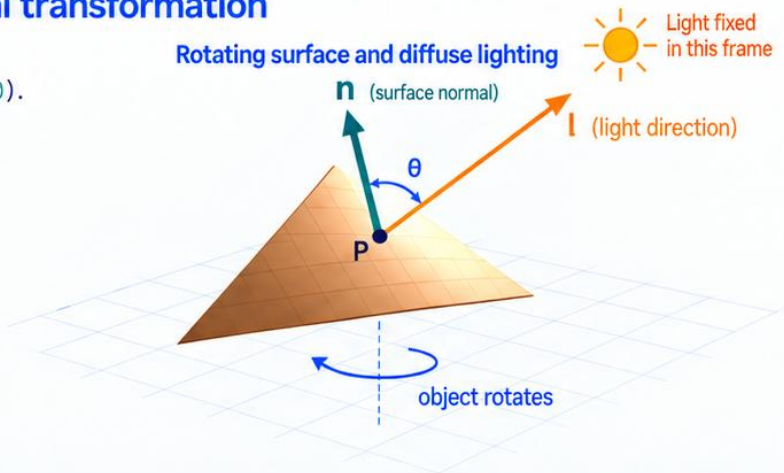
Apply M, then V, then P; clip first, then divide by w and map to the viewport.

Lighting works only when vectors share a frame



Diffuse response and normal transformation

- Diffuse response uses `max(dot(normalize(n), normalize(l)), 0)`.
- Normal transformation uses the inverse transpose of the relevant linear transform (3×3 part).
- **Example 8** transforms normals into view space and uses a view-space light direction.
- **Example 10** transforms normals into world space and uses a fixed world-space light direction.
- The shader shown computes diffuse light only. No ambient or specular term is present.



Diffuse equation (Lambert)

```
float diffuse = max(dot(normalize(n),
                        normalize(l)), 0.0);
```

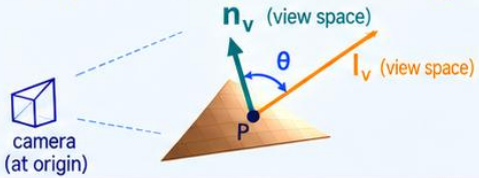
n and l must be in the same coordinate frame (view space, world space, etc.).

Normal transform: inverse transpose

$$N = \left(A^{-1}\right)^{T}$$

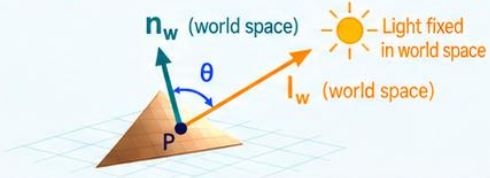
A = linear 3×3 part of model or model-view.
Transform, then normalize the normal.

Example 8: view-space normals + light



- Transform normals to view space.
- Use a view-space light direction (l_v).
- Compute `diffuse = max(dot(`n_v`, `l_v`), 0)`.
- Rotate the object: the normal changes relative to the light.

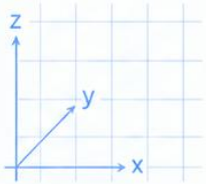
Example 10: world-space normals + light



- Transform normals to world space.
- Use a fixed world-space light direction (l_w).
- Compute `diffuse = max(dot(`n_w`, `l_w`), 0)`.
- Rotate the object: the normal changes relative to the light.

Key takeaway: ⚠ Lighting works only when the normal and light vectors are in the same coordinate frame.

Indices reuse vertices; scene instances reuse geometry



Indexed geometry: reuse vertices

Example 9: drawElements

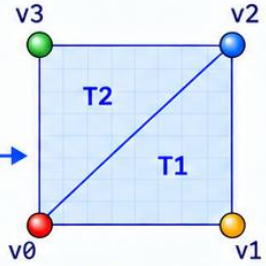
- An index buffer selects shared vertices to construct primitives.
- The quad uses indexed triangles instead of requiring every corner to be duplicated.

Position buffer
(4 unique vertices)

```
v0  (-1, -1, 0)
v1  ( 1, -1, 0)
v2  ( 1,  1, 0)
v3  (-1,  1, 0)
```

Index buffer
(6 indices → 2 triangles)

```
0, 1, 2,
0, 2, 3
```



Two indexed triangles form one quad.

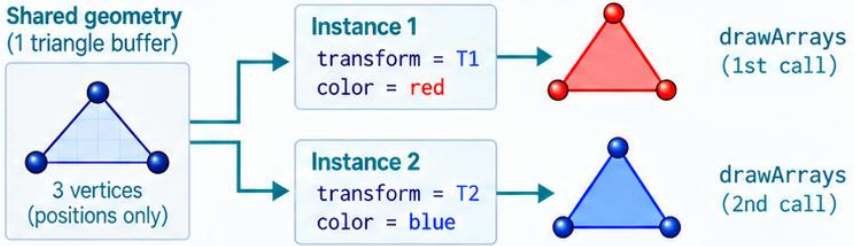
Conceptual sequence · Example 9

```
gl.bindBuffer(gl.ARRAY_BUFFER, posBuffer);
gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, indexBuffer);
gl.drawElements(gl.TRIANGLES, 6, gl.UNSIGNED_SHORT, 0);
```

Scene instances: reuse geometry

Example 11: two drawArrays calls

- Example 11 uses one position buffer for differently transformed and colored triangles.
- Its uniforms change between two separate drawArrays submissions.



Conceptual sequence · Example 11

```
gl.bindBuffer(gl.ARRAY_BUFFER, posBuffer);  // one geometry buffer
setUniforms(T1, red);
gl.drawArrays(gl.TRIANGLES, 0, 3);          // 1st triangle
setUniforms(T2, blue);
gl.drawArrays(gl.TRIANGLES, 0, 3);          // 2nd triangle
```

Shared geometry does not imply a GPU instanced draw.
Example 11 issues two separate drawArrays calls. GPU instanced drawing is a separate technique; this source does not implement it.

A scene composes several draws into one frame.

WEBGL • EXAMPLE 12

One frame submits multiple objects with the right state for each draw.

- Objects may share geometry with different surface calculations.
- Each draw needs the correct program, attributes, texture, and uniforms.
- Object transforms are independent.
- Depth testing resolves visibility.
- The frame loop updates state and submits the scene again.

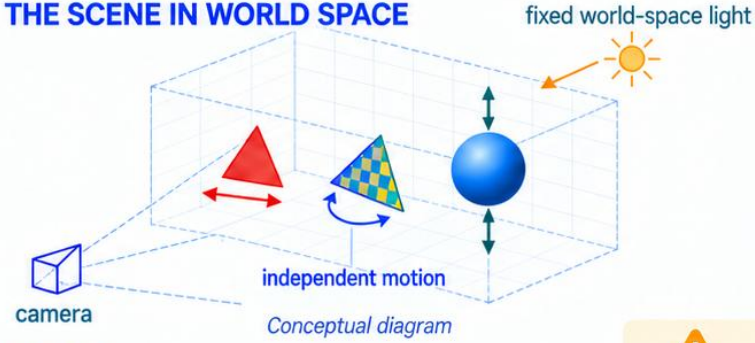
ONE FRAME ON THE GPU

1. **Clear buffers** — clear color + depth
2. **Draw red triangle** — program + transform + draw
3. **Draw textured triangle** — attributes + texture + draw
4. **Draw lit sphere** — normals + light + indexed draw

Depth testing during each draw resolves visibility

update → submit → repeat

THE SCENE IN WORLD SPACE



Conceptual diagram

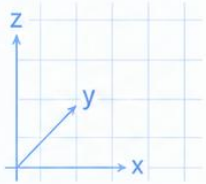
PER-OBJECT REQUIREMENTS

Object	Program / Surface	Key resources	Transform (model matrix)
Red triangle	program1 (unlit color)	position buffer (attributes)	M1 (independent motion)
Textured triangle	program2 (texture)	position buffer + texture (attributes + sampler)	M2 (independent motion)
Lit sphere	program3 (normals + lighting)	position + normal buffers (indexed geometry)	M3 (independent motion)

Demo: disable one object draw, then change only its motion and restore it.

Three.js takes over GPU plumbing while preserving scene intent

Both are choices within broader graphics API families.



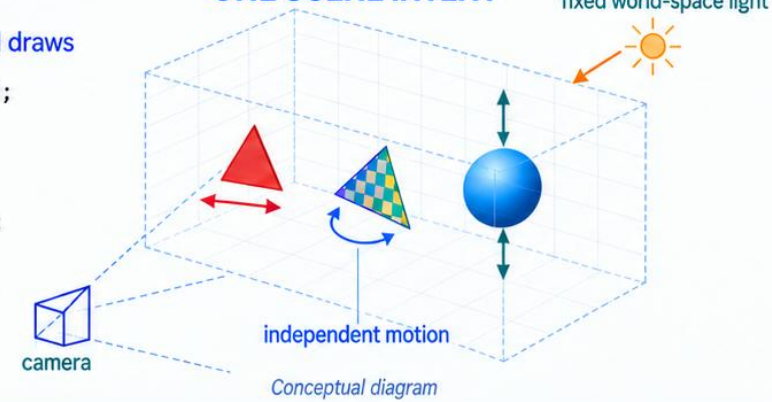
LOW-LEVEL EXAMPLE · WebGL

WebGL example 12 · explicit GPU resources and draws

```
gl.bindBuffer(gl.ARRAY_BUFFER, posBuffer3);
gl.useProgram(program3);
...
gl.bindBuffer(
    gl.ELEMENT_ARRAY_BUFFER, indexBuffer3);
gl.drawElements(gl.TRIANGLES,
    indicesCount3, gl.UNSIGNED_SHORT, 0);
```

Attributes + uniforms + textures are explicit.

ONE SCENE INTENT



Conceptual diagram

HIGHER-LEVEL EXAMPLE · Three.js

Three.js example 1 · objects and renderer

```
const sphereGeometry =
  new THREE.SphereGeometry(1, 30, 30);

const sphere = new THREE.Mesh(
  sphereGeometry, sphereMaterial);

scene.add(redTriangle,
      texturedTriangle, sphere);

renderer.render(scene, camera);
```

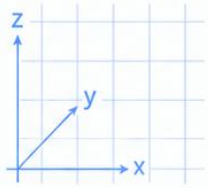
Geometry + material + transform form a mesh.

RESPONSIBILITY	WEBGL	THREE.JS
Geometry	vertex + index buffers	geometry
Surface response	shader + texture state	material
Object pose	transform uniforms	mesh transform
Frame submission	draw calls	renderer.render(scene, camera)

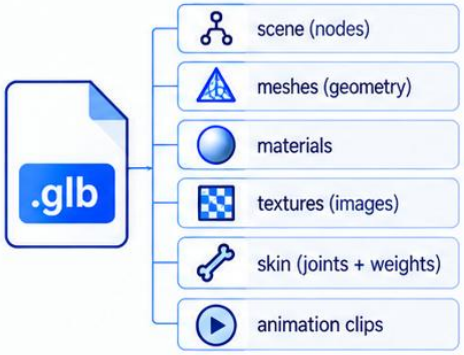
Preserve: camera · motion · UV mapping · light frame · material response · output settings

⚠ **One render call can issue several GPU draws.**

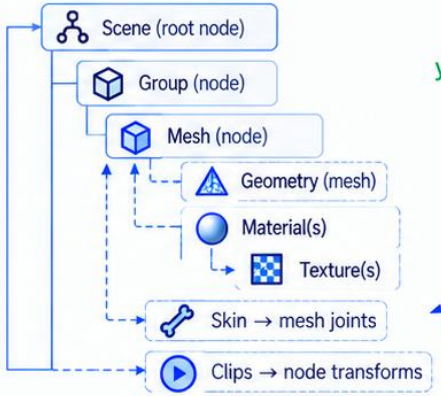
A GLB reconstructs an authored scene, not just a mesh



GLB: scene + geometry + materials + textures + skin + clips

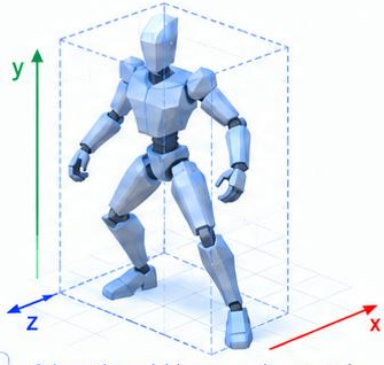


Reconstructed scene hierarchy (in memory)



Framed animated model (schematic)

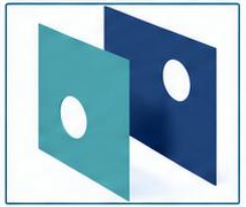
Initial pose + world matrices before bounds.



Schematic model (conceptual geometry)

Overlapping surfaces: cutout vs. blended

Cutout + depth writes (this asset)



- ✓ Fragments with alpha < 0.5 discarded (this asset)
- ✓ Remaining fragments depth-tested and written
- ✓ `transparent = false` `depthWrite = true`
- ✓ `alphaTest = 0.5` (asset-specific)

Blending can expose sorting artifacts

- ⓘ `transparent = true`
- ⓘ order-dependent blending
- ⓘ Overlapping surfaces can expose sorting artifacts (draw order)

Load → parse → attach → pose → bounds → frame

```
const gltf = await loader.parseAsync(data, "");
const model = gltf.scene;
model.updateMatrixWorld(true);
```

1. Load and parse reconstruct objects, geometry, materials, textures, skinning, and clips in this asset.
2. Attach the loaded scene only after the asynchronous load completes.
3. Update the initial animated pose and world matrices before computing framing bounds.

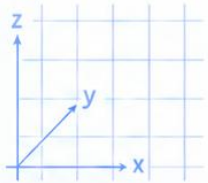
This asset: material correction for overlapping surfaces

```
material.transparent = false;
material.depthWrite = true;
material.alphaTest = 0.5;
```

ⓘ **Alpha sorting, face sidedness, and handedness are different questions.**

load .glb → parse GLB bytes → attach to scene → initial pose + world matrices → compute bounds → frame camera

Animation changes the model; orbit changes the view



1. A clip action starts playback; `AnimationMixer.update(delta)` advances animated properties.
2. `OrbitControls` changes the camera and target through orbit, zoom, and pan.
3. **These updates are independent:** moving the viewpoint does not move the skeleton.
4. Use elapsed seconds, with a bounded delta after an interruption.
5. Stop callbacks and dispose controls, mixer state, model resources, and renderer on replacement.

ANIMATION UPDATES MODEL

`clip action → animated pose`

skeleton pose (changes over time)

Conceptual model

model pose

animated mesh + skeleton

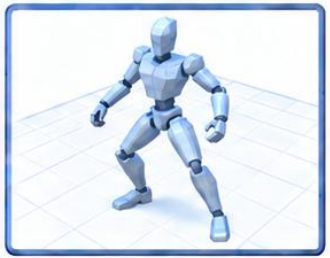
ORBIT CHANGES CAMERA

`orbit / zoom / pan → camera pose`

camera pose

camera position and target

ONE FRAME COMBINES BOTH



rendered frame: animated model from chosen view

FRAME UPDATE (`T/02-animated-model.js`)

```
mixer.update(delta);
controls.update();
renderer.render(scene, camera);
```

```
renderer.render(
scene, camera)
```

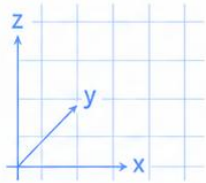
DELTA IN SECONDS

Use elapsed seconds.
Bound a long delta after an interruption.

REPLACE EXAMPLE → STOP + DISPOSE

- stop animation callbacks
- dispose controls and mixer state
- dispose model resources
- dispose renderer

AR changes who owns the camera and frame timing



1 CHECK CAPABILITIES + PERMISSIONS

Feature support, secure context (HTTPS), and permission are required.

2 USER ACTION → REQUEST SESSION

Request `immersive-ar` with required `hit-test`; `Dom overlay` is optional.

3 ENABLE XR ON THE RENDERER

Use the renderer's XR manager and `renderer.setAnimationLoop`

4 RUNTIME OWNS CAMERA + TIMING

The runtime supplies tracked views and projection; desktop orbit control is suspended.

5 TRANSPARENT BACKGROUND FOR AR

Remove the opaque scene background for AR; desktop preview remains the fallback.

CAPABILITY GATE

THREE.JS CODE (ENABLE XR)

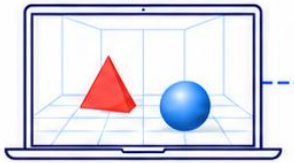
```
renderer.xr.enabled = true;
renderer.xr.setReferenceSpaceType("local");
renderer.setAnimationLoop(render);
```

```
navigator.xr.requestSession("immersive-ar", {
  requiredFeatures: ["hit-test"],
  optionalFeatures: ["dom-overlay"]
})
```

```
immersive-ar
required: hit-test
optional: dom-overlay
```

DESKTOP PREVIEW

no AR support → desktop preview



desktop camera (e.g., OrbitControls)

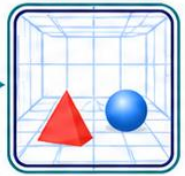
THREE.JS RENDERER

(WebGL)

- scene
- renderer
- XR manager

IMMERSIVE AR SESSION

runtime owns tracked views + XR frame timing



XR camera (from runtime)

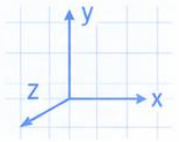
- runtime provides camera views and projection
- desktop orbit control suspended
- scene background is transparent for AR

EXIT → RETURN TO DESKTOP CAMERA

Conceptual view

KEY TAKEAWAY | WebXR does not replace WebGL or Three.js — it adds tracked spatial views to the same renderer.

A surface hit is evidence for a candidate



1. **Viewer-space hit-test source** casts toward the environment using the runtime's observations.
2. **Read hit poses in the local reference space** for the current XR frame.
3. **The hit-pose +Y axis** represents the surface normal.
4. **Accept near-horizontal surfaces** using `normal.y ≥ 0.9`.
5. **Show the ring only with a usable hit** and tracked view; missing evidence cannot authorize placement.

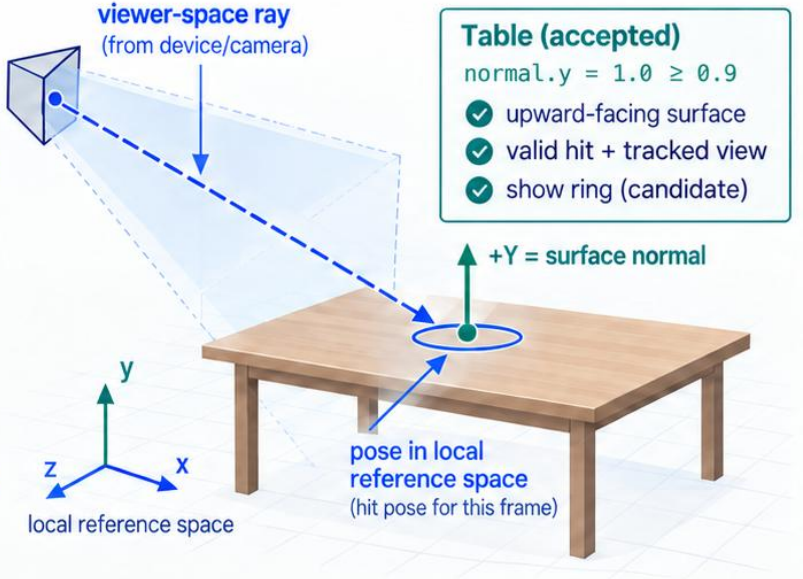


Table (accepted)

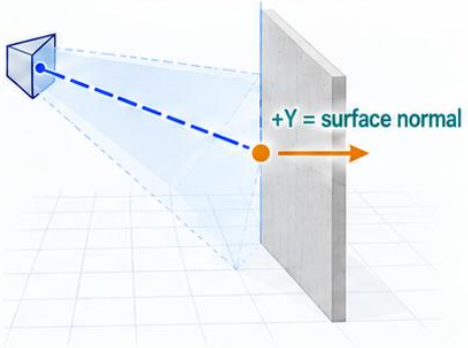
`normal.y = 1.0 ≥ 0.9`

- ✓ upward-facing surface
- ✓ valid hit + tracked view
- ✓ show ring (candidate)

Wall (rejected)

`normal.y = 0.0 < 0.9`

- ✗ vertical surface
- ✗ does not meet predicate
- ✗ no ring (no placement)



HIT-TEST RESULT (XR FRAME)

```
const hits = frame.getHitTestResults(hitSource);
const pose = hits[0]?.getPose(referenceSpace);
```

SURFAOE ACCEPTANCE PREDICATE

```
accept if normal.y ≥ 0.9
```

Demonstrates near-horizontal, upward-facing surfaces such as tables or floors.

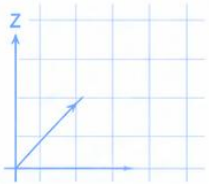
MISSING EVIDENCE

No valid hit or untracked view → no ring (no placement).

A surface hit is candidate evidence, not a placement.

Conceptual diagram — not device test results.

Placement, scale, and animation belong to different transforms



1 Tap → request
User taps to place or move the model (one instance).

2 Next XR frame
Validate fresh hit evidence immediately.

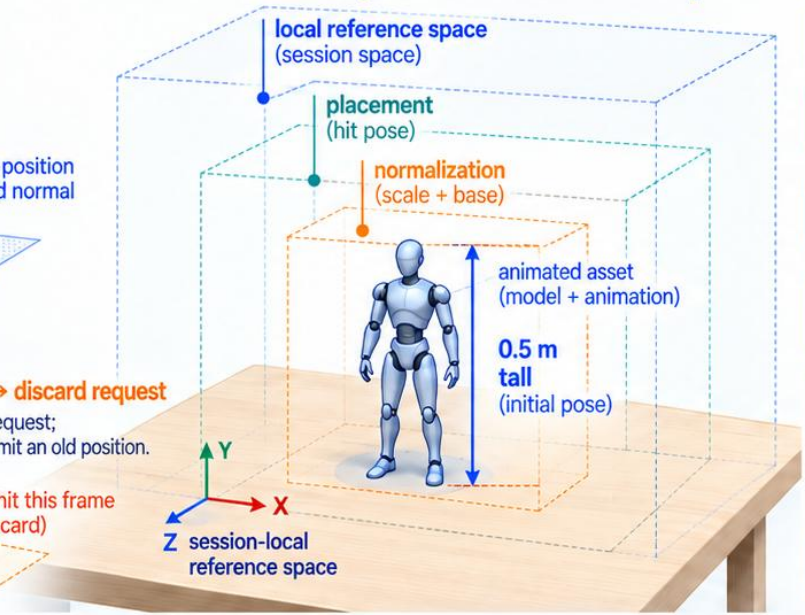
hit position and normal

3 Valid → place or move
Update the placement transform.

4 Missing → discard request
Clear the request; do not commit an old position.

no hit this frame (discard)

Transform hierarchy and coordinate ownership



Transform hierarchy (parent → child)

local reference space	XR session's local reference space
placement (Group)	Receives hit transform from fresh XR frame (in session space).
normalization (Group)	Scales model to 0.5 m tall and places base at surface (chosen display size).
animated asset (model + animation)	Imported scene hierarchy and animation retained.

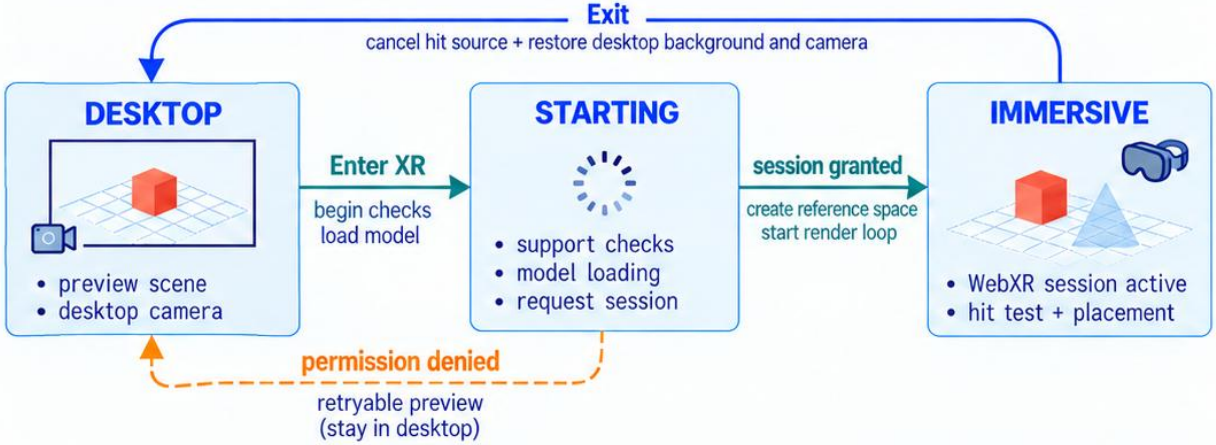
A later tap moves the same model instance

Key points

- A tap requests placement; the next XR animation frame must still contain valid hit evidence.
- A missing hit discards that request rather than committing an old marker position.
- The placement group receives the hit transform in the session's local reference space.
- A normalization group makes the initial model pose 0.5 m tall with its base at the surface (chosen display size, not a real measurement).
- Imported scene hierarchy and animation are retained; a later tap moves the same model.

Session-local placement; no native anchor.

A session is finished only when its work is released



Key lifecycle behaviors

1. Support checks and model loading may finish after the selected example changes.
2. Denied permission returns to a usable desktop preview with a retry path.
3. Session exit cancels the hit source and restores the desktop background and camera.
4. **Rerun/Reset** ends an active session, stops its loop, and disposes owned resources.
5. Late session grants or model results are released rather than reviving an old run.

Run / Reset cleanup (ends the current session)

End session
request session end

→ **Stop render loop**
stop scheduling frames

→ **Release XR state**
cancel hit source
restore camera + background

→ **Wait for pending work**
renderer attachment
and session end

→ **Dispose owned resources**
controls, geometry,
materials, textures, renderer

Late result after Run / Reset (do not revive old run)

Support check completes later

Model load completes later

Session grant completes later

release result
do not revive old run

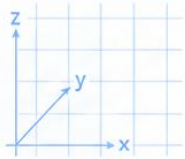
Discard result
free resources
ignore for old run

Automated checks vs. physical-device evidence

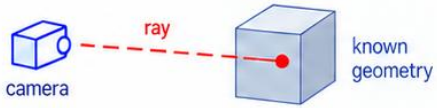
Automated lifecycle checks
- permission denied path
- session start / end
- resource cleanup
- late-result release

Mock checks ≠ physical-device verification
- do not prove real AR tracking
- not a substitute for device tests
- still need physical-device evidence for tracking accuracy

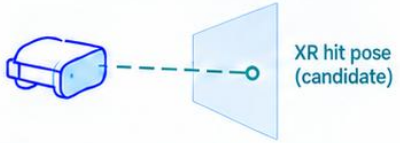
Desktop rays, XR hits, and native anchors answer different questions



1 Desktop ray: known virtual geometry
Intersects geometry the application already knows.



2 XR hit: current environment evidence
Provides a pose derived from the runtime's current environment evidence.



3 Native anchor: resolve each frame
Provides an anchor space whose pose must be resolved in later frames.

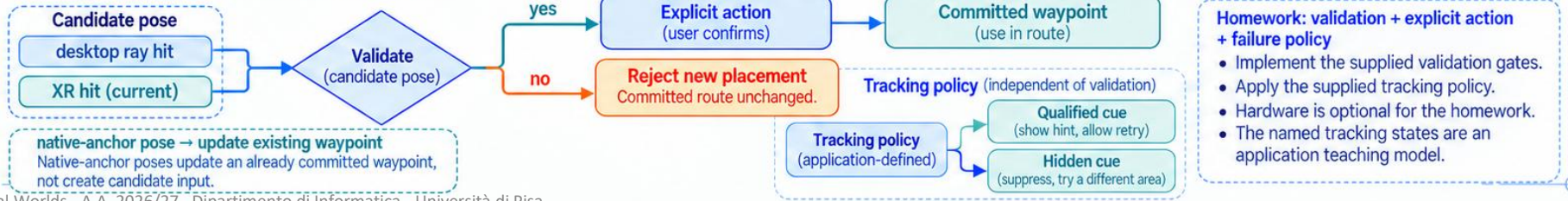
4 AR sample: one session-local pose
Keeps one placement for this session. Does not create native anchors or promise persistence.

Method	Evidence source	Coordinate ownership	Pose update requirement	Key limitations
Desktop ray (virtual geometry)	Geemetry the application already knows.	Application (its scene coordinate system).	Recompute intersection as camera/ray or geometry changes.	Only tests known virtual geometry. Does not use real-world environment.
XR hit (environment)	Runtime's current environment evidence.	Provided by the XR runtime (in a reference space).	A new hit test each frame to get a current candidate pose.	Not a physical identity proof. Environment may change.
Native anchor (anchorSpace)	Anchor space created by the XR runtime.	Provided by the XR runtime (relative to a reference space).	Resolve anchorSpace pose each frame relative to a reference space.	**No permanence or relocalization guarantee.**
AR sample placement	A single placement from an XR hit in this session.	Stored by the application (in the current session).	Keep and use this pose during the session.	Does not create native anchors. Does not promise persistence.

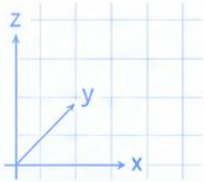
From candidate evidence to a committed waypoint

Add explicit validation and a failure policy (homework).

Application states are not WebXR enum values.

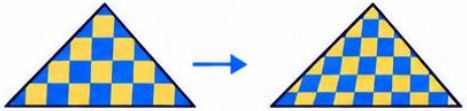


Change one mechanism and explain the resulting behavior



1 UV change: mapping, not geometry

Keep the triangle geometry fixed.
Change UV mapping and predict what changes.

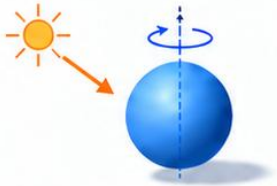


UV mapping A → UV mapping B

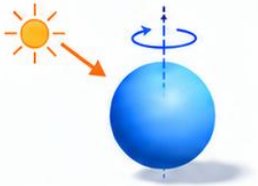
Geometry (vertices) is the same; only the texture mapping changes.

2 Light change: inspect its reference frame

Move the light's parent or direction.
Explain its frame relative to a rotating object.



World-space light (stays fixed)



Light parented to object (moves with it)

PREDICT
What will change? Why?

EDIT
Change one parameter or state.

RUN
View the result in the scene.

EXPLAIN
What changed? Which state explains it?

RESET
Return to the starting state and try another variation.

PREDICT → EDIT → RUN → EXPLAIN → RESET

3 Pause mixer: orbit still moves the view

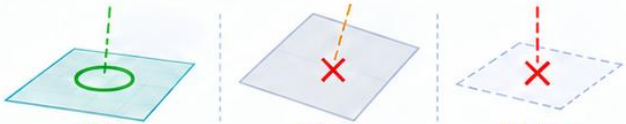
Pause model animation while orbiting.
Identify the state that still changes.

Animation playing (model moves)

Animation paused (orbit still moves view)

4 Hit fixtures: valid / tilted / missing

Trace different hit fixtures and predict the ring and placement behavior (no device required).



Valid
Horizontal surface shows a placement ring.

Tilted
Sloped surface rejected; no ring.

Missing
No surface hit no ring.

No headset required for fixture traces.

Bounded experiments on supplied code.

Reset between trials and record a short prediction, observation, and explanation.

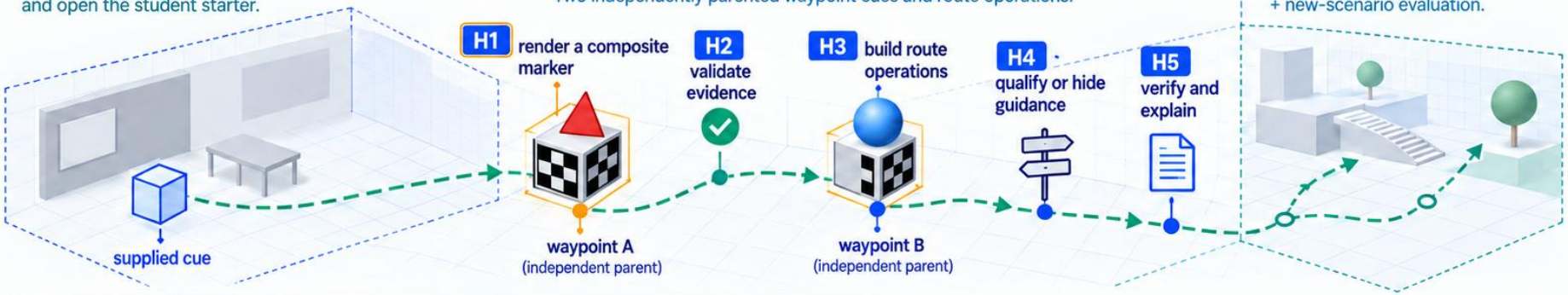
Extend the supplied cue into an evidence-aware route

SUPPLIED: room + cue + fixtures
Run the single-cue demonstration and open the student starter.

STUDENT-OWNED ROUTE EXTENSION (H1 – H5)
Two independently parented waypoint cues and route operations.

Homework scope ends here

FINAL PROJECT
Original design + integration + new-scenario evaluation.



BOUNDED HOMEWORK

- **H1** Build a different composite marker and explicit rendering setup.
- **H2–H3** Validate supplied evidence; implement independent waypoints and route operations.
- **H4** Qualify or hide guidance using the supplied application policy.
- **H5** Add an edge-case test and explain the engineering choices.

SUBMIT: code · test · screenshot · route JSON · explanation

code · edge-case test · screenshot · route JSON · explanation

⚠ **Unchanged scaffolding does not earn implementation credit.**
Supplied examples are common scaffolding; original project design, integration, and evaluation remain open.

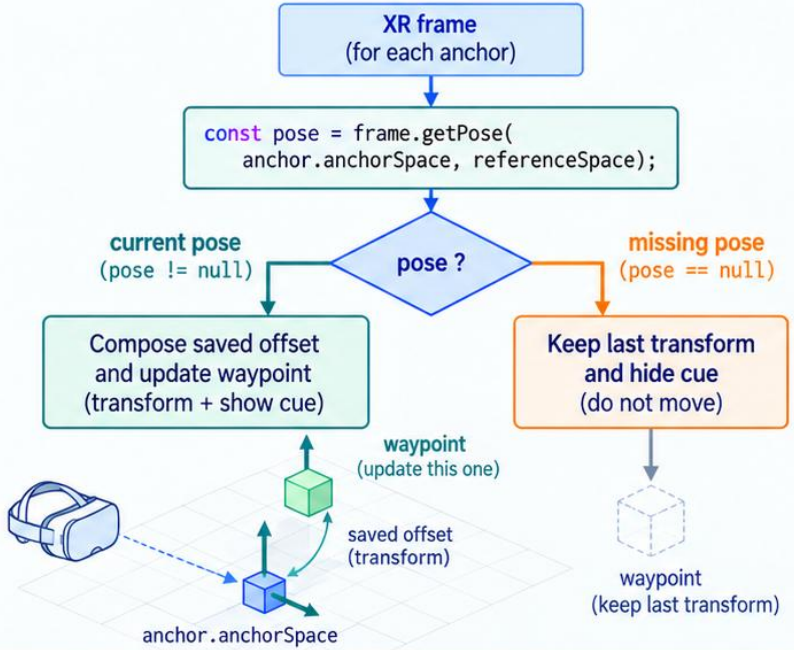
Takeaway: use the supplied single cue to build a bounded, evidence-aware route, then design a substantially original final project.

OPTIONAL APPENDIX

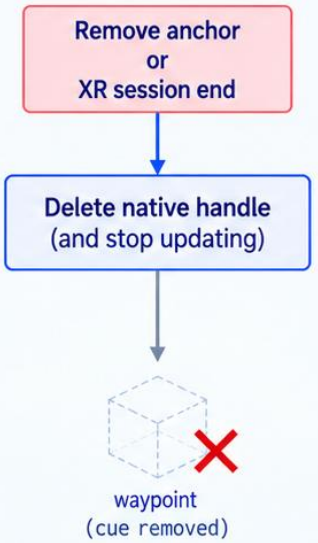
Native anchors require a per-frame lifecycle

- Native anchors are separately negotiated and are absent from the simple AR-placement example.
- Resolve `frame_getPose(anchor.anchorSpace, referenceSpace)` on each XR frame.
- With a current pose, compose the saved offset and update only the corresponding waypoint.
- With no pose, retain its last transform and withdraw the cue; release native handles on removal/session end.
- Student scope remains one isolated synchronization hook and two mocks after the required homework.
  No device required.

Per-frame anchor update loop (WebXR)



Cleanup on removal or session end



Instructor reference implementation

- Richer native anchor lifecycle with per-frame synchronization.
- Not part of the simple AR-placement example.
- Use the two mocked pose cases only if time allows.
- This is not a new implementation assignment.

Student extension (after required homework)

- One isolated synchronization hook (per-frame update).
- Two mocks: one with a current pose, one with a missing pose.
- **No device required.**

References

WebGL API — MDN	developer.mozilla.org/en-US/docs/Web/API/WebGL_API
Three.js — documentation	threejs.org/docs/
glTF 2.0 — Khronos registry	registry.khronos.org/glTF/
WebXR — Device API	w3.org/TR/webxr/
WebXR — Hit Test Module	w3.org/TR/webxr-hit-test-1/
Native anchors — optional reading	immersive-web.github.io/anchors/

Lab version: Three.js r186

Full links and course sources: sources.md

Model credit: lab/examples/assets/README.md