ChatGPT

Implementing Door Detection in AR (Unity + ARKit/ARFoundation + YOLOv5)

Building an AR system that recognizes **doors** and places a virtual **quad** (a rectangular plane) on each real door involves combining computer vision (YOLOv5 object detection) with AR spatial mapping (ARKit via ARFoundation). Below, we outline a step-by-step guide with best practices for **detecting doors**, **mapping detections to 3D space**, **spawning/updating quads**, and ensuring each door is covered by exactly one aligned quad. We also cover Unity-specific implementation tips (using ARFoundation components) and performance considerations for mobile AR.

Overview of the Approach

- **Object Detection with YOLOv5:** Use a YOLOv5 model (e.g. via Unity Barracuda or similar ML plugin) to detect doors in the camera feed in real time. YOLOv5 will provide either a 2D **bounding box** around the door or a segmentation mask of the door shape in the camera image.
- **2D to 3D Mapping:** Convert the 2D detection (box or mask in the camera image) to a 3D position and orientation in the AR world. This uses ARKit/ARFoundation features like **raycasting** with depth or **meshing** to find where the door is in physical space.
- **Anchoring a Quad:** Spawn a single **quad** (a flat rectangular GameObject) for each unique physical door. The quad is placed at the detected door's position, sized to match the door's frame, and rotated so it lies flush against the door (aligned with the wall or door surface).
- **Clustering & Persistence:** Ensure that if the same door is detected multiple times (from different angles or at different times), you **do not create duplicate quads**. Instead, recognize it as an already-detected door and update the existing quad.
- **Continuous Updates:** As ARKit refines its understanding of the environment (more feature points, mesh data, etc.) and as YOLO possibly re-detects the door from new viewpoints, update the quad's **position, size, and orientation** to improve accuracy.
- **Stability & Accuracy:** Use AR anchors and ARFoundation's tracking features to keep the quad stable in space. Leverage best practices (like parenting content to anchors or detected planes) so the quad doesn't jitter or drift as the device moves [1].
- **Performance Optimizations:** Running an object detector and AR tracking simultaneously on mobile is heavy. Use strategies to maintain frame rate: e.g. using a lightweight model or lower input resolution, throttling detection frequency, taking advantage of ARKit's hardware features (like the LiDAR **Depth API** for precise distance [2]), and efficient use of Unity's ARFoundation components.

Below, we break down the implementation into stages with Unity-specific guidance and code patterns.

1. Door Detection with YOLOv5 in Unity

Integrating the YOLOv5 Model: To run YOLOv5 in Unity, you can use Unity's **Barracuda** library or other inference engines. Convert the YOLOv5 model to an ONNX format that Barracuda can load. Note that Barracuda (as of 2020) officially supported up to YOLOv3-tiny [3]; by 2025, newer versions or Unity's Sentis

may support YOLOv5. If Barracuda lacks some YOLOv5 layers, consider using a smaller variant (e.g. YOLOv5n or YOLOv5s) or a different plugin. An alternative is to use Apple's **Core ML** if targeting iOS (by converting YOLOv5 to CoreML and using ARKit's Vision integration), but Barracuda offers cross-platform support.

Capturing Camera Frames: Use ARFoundation's **ARCameraManager** to access the camera image each frame. Subscribe to the `cameraFrameReceived` event and obtain the **XRCpuImage** (CPU camera image) or an RGBA `Texture2D` for the YOLO model input. Keep in mind the image may need transformations: ARFoundation provides the camera image in a particular orientation and format. You may need to **flip or rotate** the image to match the model's expected input orientation [4] (for example, ARKit might give a vertically flipped image, and YOLO expects a right-side-up image). Also crop/resize the image to the model's input resolution (e.g. 640x640). The Medium article by Deren Lei provides helper functions for ARCamera image preprocessing (flip, rotate, crop) for model input [4].

- *Unity Tip:* When using `TryAcquireLatestCpuImage`, **dispose** of the image after converting to a usable format to avoid memory leaks. Alternatively, use `ARCameraManager.captureCameraImage` with GPU textures. Unity's ARFoundation samples (e.g. the **CameraImage** sample) show how to convert the camera YUV image to an RGB `Texture2D` [5].

Running YOLOv5 Inference: Once you have the camera frame as a `Tensor` or `Texture`, run it through the YOLO model each frame (or periodically). This typically yields a list of detected objects with class labels, confidence scores, and bounding boxes (in normalized image coordinates). If using Barracuda, you would call `worker.Execute(inputTensor)` and then parse the output tensor(s) for bounding boxes. If using a higher-level solution or CoreML, you might get bounding boxes directly.

- *Example:* Using a TensorFlowSharp or Barracuda approach, Sha Qian's tutorial shows capturing a frame and running an SSD or YOLO model, then retrieving detections [6]. For instance:

```
var outputs = detector.Detect(cameraTexture, threshold: 0.6f);
```

  This returns a list of detections (each with a class name, confidence, and `rect` coordinates). You would filter for the `"door"` class (if your model is trained to recognize doors) and extract the bounding box.

Handling Detection Results: For each frame's detections, identify if a **door** is present. YOLOv5 might output multiple boxes if multiple doors are in view, or sometimes overlapping boxes for the same door (depending on NMS settings). Use a confidence threshold to ignore low-confidence detections. If multiple overlapping boxes for "door" appear, typically non-maximum suppression in YOLO should handle it, but you can also enforce a rule to take the highest-confidence box per door.

- If your YOLO model provides **segmentation masks** for doors (i.e. YOLOv5 segmentation variant), you get a pixel mask for the door region. This can improve accuracy in placement and sizing (we'll use it later for precise alignment), but it comes with more processing overhead. To keep things simple, you can start with bounding boxes and later refine with segmentation if needed.

- **Coordinate conversions:** YOLO outputs bounding box coordinates relative to the image. Typically these are normalized [0,1] or pixel values relative to image size, and with origin at top-left of the

image. ARFoundation's screen coordinate system has origin at top-left for the camera image as well (if using raw camera image). However, if you use `ARRaycastManager.Raycast(Vector2 screenPoint, ...)`, it expects a **viewport coordinate** (origin at top-left, in pixels or normalized depending on the overload). Make sure to convert YOLO's output to the coordinate space Unity's ARRaycast expects. Usually, if you have the pixel `x,y` of the top-left of the bounding box and its width/height, you can compute the center as `centerX = x + w/2` and `centerY = y + h/2`. If using normalized coordinates, multiply by the screen pixel dimensions. Also note ARFoundation might use *bottom-left* origin for `Screen.width, Screen.height` in some contexts; double-check and flip the Y if needed. (Sha Qian notes that the YOLO output Y is top of screen whereas AR camera uses bottom origin, requiring an inversion [7]. Be mindful of this and adjust Y as `y_for_raycast = screenHeight - centerY` if needed.)

Bounding Box Grouping (Optional): At the detection stage, it's helpful to implement a simple **tracker** that associates boxes across frames. This can smooth out fluctuations and ensure persistent identification of the same door. For example, Deren Lei suggests grouping bounding boxes across frames that likely correspond to the *same object*, then picking the best (highest confidence) one to localize [8]. In practice, you can compare the new detection's 2D position with the previous frame's detection – if they overlap significantly (high IoU) and are the same class "door," treat them as the same object. This helps eliminate flicker (e.g., if one frame YOLO detects, next frame it misses, etc.). We will further enforce uniqueness at the 3D anchor stage.

2. Mapping 2D Detections to 3D World Positions

Once a door is detected in 2D, the next step is to find *where that door is in the real world* so we can place content there. We need to go from an image-space **bounding box** (or pixel mask) to a **3D pose** (position + orientation) in the AR scene. There are a few approaches:

(a) Raycasting with ARFoundation (Hit-Test): The simplest method is to perform a **raycast** from the camera through the center of the detected box and find where it intersects real-world geometry. ARKit's hit-testing can return a 3D point even if no explicit plane is detected, by using **estimated planes** or feature points [9]. In Unity ARFoundation, you do this via `ARRaycastManager`.

- **Raycast to Estimated Planes:** Use `ARRaycastManager.Raycast()` with a `TrackableType` mask that includes **estimated surfaces** and **feature points**. For example:

```
Vector2 screenCenter = new Vector2(centerX_pixels, centerY_pixels);
List<ARRaycastHit> hits = new List<ARRaycastHit>();
if (arRaycastManager.Raycast(screenCenter, hits, TrackableType.PlaneEstimated |
TrackableType.FeaturePoint | TrackableType.PlaneWithinInfinity))
{
    ARRaycastHit hit = hits[0];
    Pose hitPose = hit.pose;
    // Pose gives position + rotation in world space
}
```

In this raycast, we allow hitting **estimated planes** (surfaces ARKit thinks might exist even if not fully tracked) and **feature points** (sparse point cloud). We also include `PlaneWithinInfinity` so that if a vertical plane is anywhere along that ray it can hit it. ARKit's raycast with `.estimatedPlane` and alignment `.any` will return the closest intersection of the ray with any detected or inferred surface [9]. In practice, this means if the door (or wall) has been sensed by AR (via LiDAR or feature points), the raycast will hit it and give us a point in space. This technique is essentially ARKit's **hit-testing**: casting a ray and measuring distance to the first physical surface along that ray [10] [11].

- **Result Pose:** The `ARRaycastHit` yields a **Pose** (`hit.pose`) which includes the precise 3D position and a rotation alignment for the surface [12]. For example, if it hit a vertical wall, the pose's rotation will be oriented such that its Z-axis points out from the wall (ARFoundation aligns `Pose.forward` with the ray direction or surface normal depending on trackable type). ARKit's `worldTransform` for the hit includes the position (translation) in world coordinates [13]. In ARFoundation, this is encapsulated in the Pose. We can use this pose directly to place our door quad.

- **Depth-based Raycast:** ARFoundation (as of ARKit 4+) also allows using the **raw depth map** for raycasts. Setting the TrackableType to `Depth` (if supported) will consider the ARKit **scene depth** or LiDAR depth for intersections [14]. This means even if no plane or point is explicitly tracked, the depth buffer can provide a hit. Using `TrackableType.AllTypes` will include depth points as well. Ensure an **AROcclusionManager** is enabled with **Environment Depth** or **Scene Depth** so that ARKit is producing depth data.

Raycasting Example: In an ARFoundation+ARKit project, Sha Qian did exactly this for object placement: *"For simplicity, I use Raycast to do a hit test against the center point of the detected box, then place the AR object at the hit position."* [7] We will follow the same logic for doors.

(b) Using ARMesh or Plane Data: If the device has a LiDAR sensor (e.g. iPhone Pro or iPad Pro), ARKit will be building a **mesh** of the environment. You can use the **ARMeshManager** to access this mesh. One approach is to cast a Unity **Physics Ray** into the mesh collider to find the exact hit point on the mesh. For example, if ARMeshManager is generating MeshFilters with MeshColliders for the environment, you can do:

```
Ray ray = arCamera.ScreenPointToRay(screenCenter);
if (Physics.Raycast(ray, out RaycastHit hitInfo, maxDistance,
layerMaskForARMesh))
{
    Vector3 doorPoint = hitInfo.point;
    Vector3 surfaceNormal = hitInfo.normal;
    // Use this to place and orient the quad
}
```

This can give a very precise point on the actual scanned geometry of the door or wall. However, enabling mesh colliders for a large environment can be computationally heavy. A lighter alternative is to iterate through ARMeshManager meshes and do a math intersection with their vertices/triangles, but that's complex. For most cases, the ARRaycastManager approach (a) is sufficient, since ARKit internally may use the mesh/depth for hit-tests anyway.

(c) Using ARKit Plane Detection: Ensure that **vertical plane detection** is enabled via ARPlaneManager if you want ARKit to recognize walls. ARKit might **merge a closed door with the wall plane** (since it's a continuous flat surface), or it might detect an open door's surface as a separate plane if the door is partially closed. However, ARKit plane detection won't specifically identify door *openings* – an open doorway is an absence of plane. So rely on YOLO for detection and use plane data primarily for orientation (we'll cover that in alignment). If a vertical plane covering the door area is detected, a raycast with `TrackableType.PlaneWithinPolygon` would hit it and provide a stable reference. But even without an explicit plane, `PlaneEstimated` hits should work as described.

Mapping Using Depth Data: If ARRaycast doesn't find a hit (for example, if ARKit hasn't yet gathered info at that spot), you can fall back to manual depth calculation. If **AROcclusionManager** (scene depth) is enabled, you can get a depth texture where each pixel gives distance to the camera. Sample the depth at the bounding box center pixel: that gives you an approximate distance to the door. Then you can compute the 3D coordinates by unprojecting the 2D point with that depth. ARFoundation provides camera intrinsics or you can use `ARCameraManager.TryGetIntrinsics` to get focal length and principal point. The formula for unprojection: given normalized image coords (u,v) and depth Z, the 3D point in camera space is `(X = (u - cx)*Z/fx,  Y = (v - cy)*Z/fy,  Z = Z)`, where (cx, cy) is principal point, (fx, fy) focal length. Transform this by the camera pose to get world coordinates. This is more math-heavy; in practice the ARRaycastManager does something similar under the hood when you use TrackableType.Depth.

Validation: After getting a 3D point from either method, you may want to **validate the hit**. For example, if the raycast returns a point very far away (e.g. 10m away on a distant wall seen through an open door), and your expectation of a door is closer (say ~3m), you might have actually hit something beyond the door (like a wall in the next room because the door was open). You can use the YOLO bounding box size as a clue: a very large bounding box likely means the door is close, a small box means far. If YOLO's distance estimate conflicts with raycast distance, you might need to adjust (more on open doors in the alignment section). In most cases, ARKit's hit-testing will hit the nearest surface that aligns with the detection, which should be the door or wall itself.

3. Ensuring One Quad per Physical Door (Clustering & Tracking)

If the user moves around and the same door is detected multiple times (e.g., you scan the door from the front, then from an angle), we want to avoid spawning a new quad each time. Instead, recognize it's the *same real door* and update the existing quad. This requires a **clustering/tracking** mechanism for detected doors:

- **Maintain a List of Detected Doors:** Create a data structure to store active door objects with their world position (and perhaps other attributes like size or an ID). Each entry corresponds to one real door that has a quad spawned.

- **Distance-Based Matching:** When a new door detection comes in (with its 3D position from step 2), compare it to the positions of existing doors. If the distance between the new detection and an existing door's position is below a certain threshold, treat it as the **same door**. For example, if the new detection is within, say, 0.5 meters of an existing door anchor, and perhaps their surface normals are similar (both vertical, facing same direction), it's likely the same physical door. In that

case, do **not spawn** a new quad. Instead, you can **update** the existing door's quad (refine its position/orientation or adjust its size with the new info).

- **Clustering by ID or Visual Overlap:** If you expect multiple doors in view at once, YOLO might detect two or more distinct doors in one frame. To decide which detection corresponds to which existing door, the distance check suffices if they're spatially well-separated. If doors are close together (unlikely closer than ~1m though), you might incorporate the YOLO bounding box distinction. But generally, doors in one environment are separated enough.

- **Example (Frame-to-Frame grouping):** If the user scans a door, loses sight (detection drops out), then comes back, your stored door list will still have that door. Use a **temporal buffer** – e.g., keep the door in memory for some time or until you're sure it's gone. If a door isn't seen for a long time and tracking quality is high, you might remove the quad (but in many AR use cases, you keep it there as persistent augmentation). Deren Lei's approach grouped bounding boxes across frames to identify the same object [8]; in our case, we extend that to grouping across the 3D space.

- **One-time Spawn Flag:** Another simple method is to mark each door as "spawned" to avoid duplicating. For instance, if YOLO is robust and you expect only one door ever in view at a time, you could just not spawn a second quad if one already exists. However, this fails if there are actually multiple distinct doors (in different parts of the environment). So using spatial clustering is safer.

- **Anchor IDs:** If using ARKit's **ARWorldMap** or persistence, each ARAnchor has an ID. But since we are defining our own anchors, we manage them ourselves. There's no automatic merging of anchors – it's up to our logic.

Algorithm outline:

```
Vector3 newDoorPos = hitPose.position;
Quaternion newDoorRot = hitPose.rotation;
bool doorMatched = false;
foreach (DoorInfo door in activeDoors)
{
    float dist = Vector3.Distance(newDoorPos, door.anchor.transform.position);
    if (dist < 0.5f)  // within 50 cm (tweak as needed)
    {
        // Same door detected again
        doorMatched = true;
        // You could update door info here (e.g., refine size or orientation)
        door.UpdateFromNewDetection(newDoorPos, newDoorRot, boundingBox);
        break;
    }
}
if (!doorMatched)
{
    // New door found, spawn a quad for it
```

```
    SpawnDoorQuad(newDoorPos, newDoorRot, boundingBox);
}
```

In `UpdateFromNewDetection`, you might average the old and new positions for smoothing or update the orientation if the new one seems more accurate (for example, when the user moves to get a better view of the door's plane, yielding a better normal estimate).

- **Use YOLO's confidence and size:** Another clue for same object is the detection **confidence** or box size. If an object was already detected with high confidence, a subsequent detection with similar position might be redundant. Also, if YOLO outputs an **object ID** (some trackers like SORT can ID objects across frames) or if using `Ultralytics YOLOv8` tracking mode [15], that can help maintain identity. But in our scenario, implementing our own simple logic is sufficient.

- **Avoiding Multiple Quads on One Door:** Tuning the distance threshold is important. Too small and you might spawn duplicates (e.g., if AR's position jittered a bit between detections), too large and you might merge separate doors (e.g., two doors on the same wall might get merged if threshold is too high). For a typical door (~0.9 meter wide), a threshold around half that (0.4–0.5m) is reasonable, assuming doors are usually spaced more than that apart. You could also require orientation similarity (dot product of normals > 0.9) to ensure they face the same way.

- **UI/Debugging:** It helps to visualize or log when a new quad is spawned vs. an existing one updated. During testing, mark each door with an ID and maybe display it to verify the logic is grouping correctly.

4. Spawning and Updating Door Quads in AR

When a new door is confirmed (not previously seen), create a **Quad** in the AR scene to represent it. Here's how to do that in Unity:

Creating the Quad GameObject: You can use a simple Unity **Quad** (GameObject with a MeshFilter/MeshRenderer and a Quad mesh) or a Plane. The quad will be oriented facing the camera by default when instantiated, but we will set its transform to the pose we got from the raycast. For example:

```
// Assume hitPose is the Pose from the AR raycast for the door center
ARAnchor anchor = anchorManager.AddAnchor(hitPose);
GameObject quad = Instantiate(doorQuadPrefab, hitPose.position,
hitPose.rotation, anchor.transform);
```

Here we create an **ARAnchor** at the door's pose, and parent the quad to this anchor. By parenting to an ARAnchor, we allow ARKit/ARCore to refine the quad's position as tracking improves – the anchor's position is updated by ARKit to "stick" to that real-world location [1]. (Without an anchor, content may drift if the AR

session origin gets reset or tracking quality changes.) Anchors ensure the quad stays at the desired physical coordinates even if the AR session's origin shifts [16] [1].

- *Anchor vs Plane:* If a **vertical ARPlane** for the wall is available at that location, you could also attach the quad to the ARPlane. For example, Unity's ARPlaneManager has `AttachAnchor(ARPlane, Pose)` which would create an anchor relative to that plane. Attaching to a plane can enhance stability (since ARKit already tracks the plane), but if the door is part of the plane, either approach works. Creating your own anchor is straightforward and decouples from ARPlane lifetimes (in case ARKit merges or adjusts planes).

Sizing the Quad: Initially, you need to decide the dimensions of the quad so that it matches the door's size. There are a few strategies:

- **Using Bounding Box and Depth:** If you have the 2D bounding box and an approximate depth (distance) to the door, you can estimate the real size. For instance, suppose YOLO's box width is `w_pixels` and height `h_pixels` and the depth from camera is `Z` (in meters). Using the camera focal length `f` (in pixels), the real width ≈ `(w_pixels / f) * Z` and real height ≈ `(h_pixels / f) * Z`. This comes from similar triangles in projection. ARFoundation provides focal length via `XRCameraIntrinsics.focalLength`. This method gives a rough size. However, it assumes the door is directly facing the camera; if the door is at an angle, the box appears narrower than actual. So consider this a starting estimate.

- **Raycast at Multiple Points:** A more robust method is to cast rays at the **extremities of the bounding box** (or segmentation mask) to directly get the 3D corners. For example, cast a ray through the top-left, top-right, bottom-left, bottom-right of the detected door in the image. If those hit the door or surrounding wall, you will get four 3D points. Then you can compute the bounds:

- The distance between the left and right hit points is the door's **width**.
- The distance between top and bottom hit (or using the known average door height if bottom might be occluded by maybe clutter) gives the **height**.
- These hit points also allow you to refine orientation: all points should lie roughly in a plane.

Implementation:

```
Vector2 topLeft = ... // pixel coords of top-left of box
Vector2 bottomRight = ... // pixel coords of bottom-right
// (You may convert these to viewport or normalized coordinates as needed)
arRaycastManager.Raycast(topLeft, hits, TrackableType.PlaneEstimated |
TrackableType.FeaturePoint);
Vector3 ptTL = hits.Count > 0 ? hits[0].pose.position : Vector3.zero;
// similarly for topRight, bottomLeft, bottomRight
```

Collect the valid hit points. If you get at least 3 distinct points on the same surface, you can fit a plane.

- **Using AR Mesh Data:** If ARKit's mesh is available, you could identify the door's mesh segment and get its dimensions. Interestingly, ARKit's **Mesh Classification** might label certain mesh triangles as

Door versus Wall, etc. (ARKit's scene understanding can classify surfaces as door, window, floor, etc. [17]). In ARFoundation's `ClassificationMeshes` sample, each triangle has a classification like Door/Wall [18]. This is advanced, but if accessible, you could directly query the mesh for classified "Door" area to get exact jamb-to-jamb, header-to-floor measurements. Without going that far, a few raycasts as above can suffice.

- **Initial Size Assumption:** If for some reason you can't get reliable size immediately (say segmentation not available and raycast at edges is iffy), you might spawn the quad with a **reasonable default size** (e.g. 2 meters high, 0.9m wide, a typical door) and then adjust it once you gather more info. Since we plan to update the quad as more data comes, an initial estimate is okay.

Setting the Quad's Transform: After instantiating the quad and parenting to an anchor, adjust its local scale to match the door. Unity's default Quad mesh is 1 unit by 1 unit (with mesh vertices from -0.5 to 0.5). If you want the scale in world units, you can set `quad.transform.localScale = new Vector3(width, height, 1)` where width and height are in meters. However, note Unity's Quad is oriented in the XY plane. If your quad's forward is facing the camera, scaling X and Y will change its size in those dimensions.

- Example: If the raycast orientation already aligns the quad with the surface, you can scale in that local space. Suppose the door width came out to 0.8m and height 2.0m:

```
quad.transform.localScale = new Vector3(0.8f, 2.0f, 1.0f);
```

This will stretch the quad mesh to that size. Ensure your quad's pivot is centered or adjust position if needed (by default, Unity's quad is centered at its middle).

Updating the Quad Over Time: With the door quad now in place, you should refine it as the AR session continues:

- **Position Refinement:** If later detections of the same door yield a slightly different position (maybe the initial was off by a bit due to limited depth data), you can move the anchor or quad toward the new position. Because the quad is anchored, you can either adjust the anchor position (by removing and creating a new anchor at a better estimate) or simply move the quad under the same anchor if the anchor is on the wall and only the quad's alignment needed tweaking. It might be easier to instantiate a fresh anchor if needed for significant adjustments.

- **Size and Orientation Updates:** As more of the door becomes visible, you might discover the door is wider or taller than initially placed. Update the quad's scale accordingly. Orientation usually should not change drastically (walls are vertical, after all), but the **yaw rotation** (the facing direction along the wall plane) might be adjusted if your initial normal was slightly off.

- **Plane Alignment:** If initially you placed the quad with only estimated planes, later ARKit might *confirm* a vertical plane for that wall or more feature points, solidifying the normal. You can then snap the quad's rotation to exactly match the wall's plane normal for accuracy. (We'll discuss flush alignment in the next section.)

- **Smoothing vs. Snapping:** When updating, consider smoothing changes to avoid visual popping. If a new measurement says the door is at X=2.1m but the old anchor was at X=2.0m, that's a 10cm shift – not huge, but the user might notice if it teleports. You could lerp the position slightly. However, AR anchors typically will be corrected by ARKit anyway if the tracking improves, so sudden jumps might also come from ARKit's own corrections. One approach is to trust ARKit/anchor to handle gradual refinement, and only intervene if you have a significantly better measurement.

- **Clustering Data:** If you accumulate multiple 3D points for the door (from different frames or angles), you can perform a **plane fit** to all those points. A simple way is to do an averaging of positions (for a stable center) and a robust estimate of normal. E.g., take two widest-apart points on the door and assume they form the horizontal direction, then cross with an up vector to get the normal (since doors are vertical). Or use three or more points and do a least-squares plane fit (solving for plane equation Ax+By+Cz+D=0). That gives a refined plane normal and center.

- **Open/Close state changes:** If the application runs long, consider if doors can open/close during the session. That's an edge case (most likely the environment is static during an AR scan). If a door moves, YOLO might detect it differently (or not at all if it swung out of view). Handling dynamic doors is very complex (would require removing or updating the quad's behavior), and typically not addressed unless your app specifically tracks moving doors.

5. Alignment and Orientation: Making Quads Flush with Doors

A critical part is ensuring the virtual quad **aligns perfectly with the real door** – it should be flush against the door or doorframe, not floating at an angle. There are a few things to get right: the plane (orientation) of the quad, and its exact position spanning from jamb to jamb (side to side) and correct height.

Vertical Orientation: Almost all doors are vertical surfaces (upright). We can take advantage of this known orientation: - Ensure the quad's plane is vertical. If your initial raycast hit gives a Pose with some rotation, verify it's vertical. For example, if using `TrackableType.PlaneEstimated`, ARKit might give a pose with alignment .any. If the door is on a wall, the pose rotation should already reflect a vertical plane (assuming ARKit recognized it as a vertical surface). If you suspect it didn't, you can enforce vertical by adjusting the rotation: e.g., set the quad's **pitch** and **roll** to 0 so it stands upright, only keeping the yaw (horizontal rotation). In Unity, you could do:

```
// Keep rotation around Y-axis only (assuming world-up is (0,1,0))
Quaternion rot = hitPose.rotation;
Vector3 euler = rot.eulerAngles;
euler.x = 0;
euler.z = 0;
quad.transform.rotation = Quaternion.Euler(euler);
```

This aligns the quad vertically while preserving the facing direction.

- If you have an ARPlane for the wall, simply use its rotation. ARPlane objects have a `alignment` (Horizontal or Vertical). For a vertical plane, the ARPlane's transform's rotation will be such that its **Y-**

axis is vertical and its plane surface is oriented correctly. Setting the quad's rotation = plane.transform.rotation will align it flush to that wall.

Surface Normal (Facing Direction): Flush with the wall means the quad's normal is exactly the wall's normal. We need to get the **direction** the door is facing. There are a couple ways: - **From Raycast Hit:** If your ARRaycastHit was on a plane or mesh, the `hit.pose.rotation` likely already has the correct normal alignment. ARFoundation's hit pose is defined such that the **pose's forward vector is the** ray *direction for feature points, but for plane hits it should align with plane's normal. To double-check, one can compare* `hit.hitType`*: if it's a Plane type, then* `hit.pose.up` *will be the plane's normal for horizontal surfaces, or* `hit.pose.forward` *might be the normal for vertical (depending on ARFoundation's convention). Documentation indicates ARRaycastHit gives a full pose including orientation if available* [19]*. In practice, many developers find that for vertical plane hits, the rotation's y-axis* aligns with plane normal (for ARCore it might differ). You may have to experiment, but often using the pose directly is fine.

- **Use multiple points:** If you raycasted the left and right edges of the door (on the wall), you can compute the wall's plane. The vector from left-hit to right-hit lies **along the wall**. Taking the cross product of that vector with the **world up vector (0,1,0)** will give you a vector pointing out of the wall. For example:

  ```
  Vector3 wallDir = rightHitPoint - leftHitPoint;
  Vector3 wallNormal = Vector3.Cross(Vector3.up, wallDir).normalized;
  ```

  Depending on the order of cross, you might get inward vs outward normal; choose the one that faces the camera (dot(wallNormal, camera.forward) > 0, if not, flip it). Now you have the normal. Construct a rotation for the quad such that its forward (or whichever axis of the quad's local space should point outward) equals this normal. You can do `Quaternion.LookRotation(wallNormal, Vector3.up)` which returns a rotation with given forward and world-up.

- **Open Doors vs Wall Plane:** If a door is **open**, the actual door surface (door slab) is at an angle relative to the wall. The question statement suggests we still want the quad "flush with the wall, matching jamb-to-jamb size, even for open or closet doors." This likely means that even if the door is open (or it's a doorway with no door), we want to cover the *doorway* on the wall. In other words, the quad should align with the wall plane (covering the door opening) rather than the door slab which is swung out. This is an important nuance: YOLO might detect an open door by the door frame or the door itself at an angle.

Handling open doors: If you raycast the center of an open doorway, the ray might go through the doorway and hit the far side of the next room (which is not what we want). To handle this, use the **door frame** for positioning: - Raycast near the door's **side edges** (left and right jambs) instead of the center. The left and right edges of the YOLO bounding box likely fall on the wall (door frame). Those raycasts will hit the wall plane at the edges of the door. Use those points to define the quad's plane on the wall. The quad's width should span between those two hits, and height from floor to top frame (you could raycast a point near the top of the bounding box as well to get the top of door frame on the wall). - If YOLO detection of an open door shows the door slab (which might be at, say, 90° to the wall), a center ray could hit that slab. How to detect this scenario? One clue: the orientation of the hit pose might not match a typical wall (i.e. the normal

might be way different from other vertical surfaces around). Another clue is distance: the door slab might be slightly offset from the wall (maybe a few centimeters if open against a door stop) or the slab might be partially out of frame. It's tricky, but a safe approach is to **always prefer the wall alignment** by default for doors. So even if you get a hit on a door slab, you might override the orientation to align with the wall's plane if you can infer it. - You can find the wall's normal by other nearby data: e.g., ARPlane of the wall (if the door is open, ARKit likely detected the wall segments on either side as one plane or two separate planes). If ARKit plane detection is active, check for a vertical plane whose boundary encompasses the door area. If found, use that plane's normal for the quad. - In summary, for an open door, you effectively place the quad covering the *doorway*. That means the quad will be slightly larger than the door slab (covering the whole frame). The left/right raycast approach gives you jamb-to-jamb width. For height, ideally raycast near the top center of the doorway (just below the top of the bounding box) – that may hit the top door frame or wall above the door. If it hits wall above, fine; if it hits nothing (if doorframe has nothing behind it because it's open to another room's void), consider using a standard door height or detecting the door header via other means (maybe the YOLO segmentation mask if it outlines the door shape including the open part).

Matching Jamb-to-Jamb Size: This phrase means the quad should exactly span the door opening horizontally (from left door frame to right door frame), and vertically from the bottom (floor or threshold) to the top frame. Achieving this: - Use the raycast hits on the left and right door frames (as described) to set the quad's horizontal span. - For vertical: if the floor is within view and ARKit detected a horizontal plane for the floor, you could raycast downward from the door detection to find the floor at the door's location (or use ARPlane extent if the wall plane intersects the floor). However, floor detection might not precisely align with the door due to slight elevation differences or if user hasn't scanned the floor there. Alternatively, assume a standard height (~2m). Many interior doors are ~2.0 to 2.1 meters high. If YOLO's bounding box covers the door fully, that plus depth can yield height as discussed. If YOLO segmentation gives the top arc of an open door, you can detect the top.

- If the door is a **closet door** (often double doors or sliding), flush with wall likely means the closet is an opening in the wall as well. The approach remains similar: detect the door panels or opening, and align the quad with the wall.

Orientation Summary: We want each quad either on the door surface (if closed) or on the door's wall plane (if open). To ensure this: - If the door is closed (or mostly closed), the first raycast likely hit the door itself (which is essentially part of the wall plane if flush). The orientation from that hit is usually correct. - If the door is open, skip using the possibly weird hit on the door slab; instead use the wall's orientation. Typically, your left/right frame hits or known wall plane will provide `wallNormal`. Use that to set the quad's rotation via `LookRotation` as shown, and position the quad anchored at the midpoint of those two frame hits. - Double-check that the quad is **not intersecting the wall**. If you place it exactly flush, half the quad geometry might go inside the wall depending on its pivot. If using center pivot and exact flush positioning, half might embed. If using one face of the quad as the visual side, you might offset it a few centimeters outwards along the normal so it doesn't z-fight with the wall. However, since it's flush, z-fighting shouldn't occur if it's exactly on the plane, unless you also render the AR wall. Typically, AR content on the wall is fine with no offset, but if you see flickering, offset by a tiny epsilon.

Example workflow for alignment (open doorway case):
1. YOLO detects a door (which is open) – bounding box spans the doorway.
2. Raycast center – hits far wall 5m away (too far).
3. Raycast left edge – hits wall at left jamb (distance 2.5m). Raycast right edge – hits wall at right jamb

(distance 2.5m). Now you have two points on the wall.
4. Compute wallNormal via cross with up. This normal points outward through the doorway.
5. Anchor a quad on the wall: position it at the midpoint of those two hits (midpoint horizontally, and vertically maybe use the y of one of those hits or a known door height). Set quad rotation to `Quaternion.LookRotation(wallNormal, Vector3.up)`.
6. Set quad width = distance between left/right hits, height = standard door height or use YOLO's vertical bounds (adjusted by depth).
7. The quad now covers the doorway, flush with wall.

Refining with Segmentation: If YOLOv5 provides a segmentation mask for the door, you can get a more exact outline. One could convert the mask into a 3D **point cloud** by taking many mask pixels and projecting them with depth (from ARKit). Then run a plane fit on those 3D points: the result gives a very accurate plane (which will be the door's plane if closed, or wall plane if it mainly sees the frame). The convex hull of those 3D points would outline the door shape. From that you could set the quad's corners exactly. This is the most precise but also computationally heavy. For real-time, you might not need this level – a few key rays and known constraints often suffice.

6. Anchor Stability and Spatial Accuracy

To keep the quads **locked in place** on the doors, leverage ARFoundation's anchors and tracking features:

- **Use ARAnchor for Each Door:** As shown, attaching the quad to an ARAnchor (or ARPlane Anchor) ensures ARKit will continually correct the anchor's transform to keep it in the same real-world spot [1]. If the tracking system experiences slight drift or needs to relocalize, it will adjust anchors rather than letting your content float away. *Unity Tip:* Use `ARAnchorManager.AddAnchor(pose)` to create a world anchor at a given pose. Store references to these anchors (e.g., in your `DoorInfo` struct/class) so you can update or remove them later if needed.

- **World Tracking Quality:** Remember that ARKit's understanding of the environment improves as the user moves around. Features like **ARKit's Session** provide quality metrics. You might query `ARSession.notTrackingReason` or monitor `ARSessionState`. If tracking quality is poor (e.g., in low light or fast motion), anchor accuracy may suffer. One way to mitigate sudden jumps is to delay spawning a quad until ARKit has a decent fix (for example, after the ARCoaching overlay finishes, or when `ARSessionState == Tracking`). However, usually by the time YOLO detects something, ARKit has some tracking.

- **Spatial Mapping Considerations:** If using **ARMeshManager** (LiDAR), the mesh data can add stability because anchors can latch onto the mesh's features. ARKit may internally use feature points or mesh geometry for anchors. There's nothing specific you need to do except ensure **collision** or **classification** if you want to use them (e.g., adding a MeshCollider to mesh as earlier described for raycasts). If using ARPlanes, each ARPlane has an **ARAnchor** implicitly. You could attach your door quad to the **wall's ARPlane** if that plane exists, which might yield even more stable results (the plane is updated as ARKit refines its shape).

- **Persistent Anchors:** If the app needs to remember door positions across sessions (not explicitly asked here, but worth noting), you could save the anchor positions (e.g., in ARWorldMap on iOS, or

some cloud anchor). That's beyond our immediate scope, but if it's a requirement, ARFoundation's ARAnchorManager has methods to persist anchors.

- **Avoiding Anchor Overload:** Each anchor has a cost (tracking and memory). For a typical scenario with a few doors, this is fine. If someone scans dozens of doors, be mindful not to create anchors unnecessarily (don't double-anchor the same door, etc., which our clustering already avoids). If an anchor is no longer needed (e.g., user leaves that area and you won't see that door again), you could remove it to free resources.

- **Anchor Placement Accuracy:** The initial placement of the anchor can have some error if the depth estimate was rough. ARKit will try to reconcile this over time – usually, if the anchor is slightly off-plane, it might correct it when better data comes in. However, ARKit does not drastically move anchors once placed; it might be up to you to adjust if you determine the anchor should be somewhere else. For example, if you realize the quad is misaligned, you might destroy and re-add the anchor at a corrected pose. This could momentarily detach from ARKit's tracking, but if placed on a detected plane, ARKit can snap it. Choose anchor re-placement carefully to avoid oscillation.

- **Smoothing Movements:** As mentioned, you can interpolate updates to the quad. But because we parent to an anchor, large-scale smoothing might conflict with ARKit's own updates. A common approach is to trust ARKit's anchor and just update scale/rotation gently if needed. If the anchor jumps (due to tracking reset), you might not be able to smooth that – it's abrupt because ARKit relocated it. But these jumps should be rare if tracking remains good.

Accuracy Tips: - **Use high-resolution features**: ARKit's **collaborative mapping** or **mesh classification** can give more context (like distinguishing door vs wall) – useful if you expand functionality. - **Lighting and Texture:** ARKit relies on visual features for tracking (in addition to LiDAR if present). A plain white door might have fewer features, so the area could be less robust for anchors. If you notice drift around feature-poor doors, that's an ARKit limitation. Encourage the user (via UI) to scan the area around the door (including textures on the wall or door frame) to help ARKit. In ARKit, you could enable a **coaching overlay** to guide the user to move the device for better mapping. - **Gravity alignment:** ARKit auto-aligns its world coordinate Y-axis to gravity. Thus vertical doors should consistently have a normal perpendicular to a gravity-aligned plane. This means your assumption that up-vector (0,1,0) is perpendicular to walls holds true (unless the user tilts their head; but AR coordinates remain gravity aligned). So you can confidently use `Vector3.up` in calculations for normal as we did.

7. Performance and Optimization for Mobile AR

Running YOLOv5 and ARKit concurrently on a mobile device (phone/tablet) can be demanding. Here are some optimizations and best practices to keep the experience smooth:

- **Use a Lightweight Model:** If full YOLOv5 (which can be large) is too slow, consider using a smaller model variant or even a different model specialized for doors. YOLOv5n (nano) or YOLOv5s are much faster than YOLOv5x, with some accuracy trade-off. Since doors are relatively easy to recognize (large, rectangular), a lighter model might suffice. You could also train a **tiny YOLO** model just for doors (one class) which could yield smaller networks. Unity Barracuda runs models on the GPU (or CPU if fallback), so ensure it's using GPU if available.

- **Reduce Input Resolution:** Running detection on a downscaled image can greatly improve speed. For example, you might use a 416×416 or 320×320 input to YOLO instead of full camera resolution. Doors can still be detected at lower res if within a reasonable range. Find the lowest resolution that reliably detects doors in your testing. This directly reduces computation in the model.

- **Lower Detection Frequency:** You likely do not need to run YOLO on **every single frame** (which could be 30 or 60 FPS). You could run it, say, every 5th frame, or twice per second, especially if the user is moving slowly scanning an environment. ARKit's tracking will still be running at full frame rate, keeping anchors updated. The detection only needs to catch when a new door comes into view or if significant changes occur. Using a Timer or coroutine to throttle the `detector.Detect` calls can help maintain frame rate. For example, run YOLO inference at 5-10 FPS. This also gives time for each inference to complete if it's slightly heavy.

- **Asynchronous Inference:** If the ML inference can be done asynchronously (in another thread or using Barracuda's asynchronous API), utilize that. Unity's Barracuda allows async execution via `StartManualSchedule` or similar patterns, so the main thread isn't stalled. Make sure to handle thread safety when feeding in the camera image and getting results. If using Core ML via Xamarin/Objective-C plugin, leverage Apple's neural engine by setting the model's `usesCPUOnly = false` (so it uses GPU/ANE).

- **Efficient Camera Image Handling:** Accessing the camera image can be a bottleneck if done incorrectly. Use the **fastest path**: for example, use `XRCpuImage.GetPlane` data directly to construct input tensor without unnecessary conversions. Avoid `Texture2D.ReadPixels` every frame (slow). If using GPU texture, perhaps use a compute shader or fragment shader to resize and normalize the image for the model input, then use `AsyncGPUReadback` to get it as tensor data. Unity's sample (Sha Qian's GitHub) shows capturing camera image and converting to RGB [5] efficiently – they borrowed code from ARFoundation samples which use unsafe code to copy YUV to RGB.

- **Optimize YOLO Post-processing:** Decoding YOLO boxes (applying NMS, etc.) in pure C# might be slow if many boxes. Try to do as much as possible in the model (some YOLO ONNX include NMS as a layer). If not, at least limit the number of predictions (e.g., YOLOv5 outputs 25200 boxes at 640x640 – perhaps use a smaller grid or accept only the top k confidences before NMS). Since you only care about the "door" class, you can filter other classes early if the model is multi-class.

- **Limit ARMesh Density:** If using ARMeshManager on LiDAR, consider reducing **mesh density** to lighten the load. ARFoundation's ARMeshManager doesn't allow adjusting density via Unity (as per docs, ARKit's meshing density can't be changed from Unity [20]), but you can limit how often you render or update the mesh. If you're only using the mesh for occasional raycasts, you might disable continuous updates after you get the needed data (e.g., stop ARMeshManager once doors found). Similarly, **plane detection** can be disabled after initial use to save some processing (planes updates are not too expensive generally, but every bit helps).

- **Memory Management:** Running a large model can use a lot of memory. Monitor memory usage to avoid crashes on older devices. Dispose of `Tensor` objects after use (Barracuda Tensors), and reuse allocated buffers if possible. If using Compute Shaders or GPU resources, release them properly when not needed.

- **Testing on Device:** Always profile on the actual target devices. What runs fine in Editor (with powerful desktop GPU) will be much slower on mobile. Use Unity's Profiler connected to the phone to see if the bottleneck is on rendering, script (likely the model), or AR subsystem. This will guide further optimization.

- **Dual-camera or LiDAR usage:** If on an iPhone with LiDAR, ARKit can provide a **segmented depth** and also **People Occlusion**. These typically won't interfere with our door logic, but be mindful if both Occlusion and your own depth usage are running, that's additional GPU load. Only enable features you need.

- **UI and Feedback:** A smooth experience might involve guiding the user. For example, showing a scanning indicator while YOLO is processing could be helpful (to set expectations if a half-second lag occurs during detection). Also, once a door quad is placed, you might highlight it and not run YOLO again on it unless needed (you can mask out that region or just skip detection if camera is focused on an already-detected door).

- **Parallelizing AR and ML:** The ARKit tracking and rendering runs on its own threads internally, but your script and ML may run on the main thread by default. Offload heavy work to background threads where possible. But be careful with Unity's API: ARFoundation calls (like raycast or anchor creation) must be on the main thread. You can however do the ML image preprocessing and inference on another thread, then return results to main thread to update AR content.

By combining these optimization strategies, you can achieve near real-time performance even on mobile. For instance, using a tiny YOLO model at ~5 FPS and ARKit depth-based raycasts could allow detecting and marking a door in under a second of it coming into view, which is usually acceptable.

Final Notes: This system leverages the strengths of both **AI detection** and **AR tracking**. YOLOv5 gives the semantic insight (where the doors are in the image), and ARKit gives the spatial understanding (where that corresponds in 3D space, and how to keep content aligned there). The result is an AR application that, for example, could highlight doorways for navigation or interactive content on doors.

Throughout development, test in a variety of scenarios: different door colors (to ensure YOLOv5 is robust), different wall textures (for ARKit tracking), open vs closed doors, etc. Use the ARKit **debug visuals** (point clouds, planes, meshes) to understand what ARKit "sees" of the door – it will help tune your raycasting and anchor placement logic. And always maintain the citations and references for any external solutions or code you integrate.

By following the above guide, you should be able to spawn **exactly one well-aligned quad per physical door**, updating dynamically as the user scans, all while maintaining anchor stability and mobile performance. Good luck with your AR door detection project!

Sources:

- Sha Qian, *"Object Detection and ARFoundation in Unity"* – example of detecting an object (apple) and placing an AR object via raycast [7] .

- Kinto Tech Blog (Viacheslav), *"Extracting 3D Coordinates of Objects Detected by Vision Framework in AR"* – describes hit-testing a 2D detection to 3D using ARKit's raycast (estimated planes, alignment any) [9] [13].
- Deren Lei, *"Object detection with localization using Unity Barracuda and ARFoundation"* – discusses grouping 2D detections across frames to identify the same object [8] and using Barracuda for YOLO models [3].
- Unity ARFoundation Documentation – ARRaycastHit provides the world-space pose of a raycast intersection [12]; Anchors usage and importance for stability [1]; ARMesh classification can identify doors vs walls in LiDAR scans [17].
- ARKit 4 Depth API news – highlights availability of precise scene depth for better 2D->3D mapping [2].

---

[1] [16] Introduction to anchors | AR Foundation | 6.0.6
https://docs.unity3d.com/Packages/com.unity.xr.arfoundation@6.0/manual/features/anchors/introduction.html

[2] Apple announces ARKit 4 with new Depth API, Location Anchors ...
https://www.auganix.org/apple-announces-arkit-4-with-new-depth-api-location-anchors-and-expanded-face-tracking-support/

[3] [4] [8] Object detection with localization using Unity Barracuda and ARFoundation | by Deren Lei | Medium
https://derenlei.medium.com/object-detection-with-localization-using-unity-barracuda-and-arfoundation-794b4eff02f8

[5] [6] [7] Object Detection and ARFoundation in Unity | by Sha Qian | Medium
https://medium.com/@shaqian629/object-detection-and-arfoundation-in-unity-8782b1ee6ea3

[9] [10] [11] [13] Extracting 3D Coordinates of Objects Detected by Vision Framework in ARSCNView Video Feeds | KINTO Tech Blog | キントテックブログ
https://blog.kinto-technologies.com/posts/2024-12-21_ar-coordinates-extraction/

[12] Struct ARRaycastHit | AR Foundation | 4.0.12
https://docs.unity3d.com/Packages/com.unity.xr.arfoundation@4.0/api/UnityEngine.XR.ARFoundation.ARRaycastHit.html

[14] Enum TrackableType | AR Foundation | 5.0.7
https://docs.unity3d.com/Packages/com.unity.xr.arfoundation@5.0/api/UnityEngine.XR.ARSubsystems.TrackableType.html

[15] Custom trained model detecting one object multiple times · Issue #5814 · ultralytics/ultralytics · GitHub
https://github.com/ultralytics/ultralytics/issues/5814

[17] [PDF] ARTIFICIAL INTELLIGENCE - ResearchGate
https://www.researchgate.net/publication/356985391_Artificial_Intellgence_--_Application_in_Life_Sciences_and_Beyond_The_Upper_Rhine_Artificial_Intelligence_Symposium_UR-AI_2021/fulltext/61b6b8b01d88475981e6b6af/Artificial-Intellgence--Application-in-Life-Sciences-and-Beyond-The-Upper-Rhine-Artificial-Intelligence-Symposium-UR-AI-2021.pdf

[18] [20] Meshing | ARKit XR Plugin | 4.0.12
https://docs.unity3d.com/Packages/com.unity.xr.arkit%404.0/manual/arkit-meshing.html

[19] Does ARFoundation have a method to fetch point surface normals in ...
https://discussions.unity.com/t/does-arfoundation-have-a-method-to-fetch-point-surface-normals-in-a-point-cloud/820390