Building a real-time AR 8-ball pool assistant: complete implementation guide

This guide provides a complete, phase-by-phase technical blueprint for building a smartphone app that uses computer vision, physics simulation, and augmented reality to provide "god mode" shot guidance on a real pool table. The system detects all 16 balls, classifies them by type, tracks the cue stick angle, predicts full shot trajectories including spin and cushion rebounds, and overlays optimal aiming lines directly on the camera feed at 30fps. The approach combines a custom-trained YOLOv8-nano model for ball detection (~90%+ mAP50 achievable with 500–1,000 annotated images), an event-based physics engine for accurate trajectory prediction, and platform-native AR rendering through ARKit and ARCore. Prior academic work—notably PoolLiveAid (98% shot prediction accuracy) Springer +2 and Queen's University's ARPool—validates that real-time pool AR guidance is technically feasible. ResearchGate Semantic Scholar The commercial success of DrillRoom on iOS further proves mobile-only approaches work without external projectors or depth sensors. OrangeLoops Orangeloops

---

Prior art establishes clear technical feasibility

Before diving into implementation, understanding existing systems provides critical design constraints and proven approaches. **PoolLiveAid** (University of the Algarve, 2013/2016) used a ceiling-mounted Kinect 2 depth sensor plus projector, achieving **98% shot prediction accuracy** for non-bouncing shots. ResearchGate Springer The system used classical CV—color segmentation, background subtraction, and Hough line detection for the cue stick. Springer +3 ARPool from Queen's University (commercialized as ARxAI) demonstrated real-time trajectory projection onto table surfaces and spawned the first billiards-playing robot. Both systems relied on overhead-mounted hardware.

The mobile-only approach is validated by **DrillRoom** (iOS, launched 2021), which uses ARKit plus ML-based ball tracking from a phone on a tripod. OrangeLoops Open-source projects provide reusable components: **Pooltool** (Python, published in JOSS 2024) offers a peer-reviewed event-based physics engine; Evan Kiefl +2 **Cassapa** (C++/OpenCV) provides a complete camera-to-projection pipeline; Hackaday GitHub **pool-ball-tracking** demonstrates YOLOv4-Tiny achieving 30fps with only 210 training images. GitHub On Roboflow Universe, several datasets are immediately available— RoboflowBen Gann's 986-image set Roboflow and ipool's 746-image 12-class set Roboflow being the most useful starting points.

The CVZone "Pool Shot Predictor" contest (sponsored by Nvidia) produced multiple working implementations combining YOLOv8 with OpenCV trajectory prediction, providing reference architectures. GitHub Computer Vision Zone The academic paper "Billiards Sports Analytics: Datasets and Tasks" (Zhang et al., 2024) contributes 3,019 break shot layouts with trajectory data from professional 9-ball games. Zhengwang125

Architecture: a four-thread buffered pipeline

The system architecture must handle camera capture, ML inference, physics computation, and AR rendering simultaneously at 30fps. The recommended approach is a **buffered parallel pipeline** with four dedicated threads.

```
Thread 1 (Camera):    30fps capture → ring buffer (3-5 frames)
Thread 2 (ML/CV):     Latest frame → YOLO inference → ball detections (~10-30ms)
Thread 3 (Tracking):  Match detections → Kalman update → physics sim → game state (~1-3ms)
Thread 4 (Rendering): Consume game state → AR overlay → display (16ms target for 60fps)
```

The detection thread runs asynchronously—the renderer interpolates between detection results for smooth overlay at full frame rate. Based on production analysis, detection latency under **50ms** yields excellent tracking reliability, **100ms** is acceptable, and beyond **300ms** both tracking and UX degrade significantly. (Medium) For pool specifically, CV inference at 10–15fps is sufficient because game state changes slowly between shots.

Technology stack recommendation. For maximum performance, build dual-native: iOS (Swift + ARKit + CoreML + Metal) and Android (Kotlin + ARCore + LiteRT + OpenGL ES), with a shared C++ core library containing OpenCV processing, physics engine, SORT tracker, Kalman filters, game state management, and homography computation. If cross-platform from a single codebase is essential, Unity AR Foundation 6.x with Sentis for ML inference is the best alternative, (Unity) accepting a 15–30% performance overhead. (INAIRSPACE) Flutter and React Native are unsuitable— (Lucent Innovation)their bridge layers add unacceptable latency for real-time CV at 30fps.

The shared C++ core links OpenCV 4.13 (available via Maven Central on Android, CocoaPods on iOS, or the lightweight opencv-mobile package from nihui for minimal binary size). Platform-specific code handles camera access, AR framework integration, ML inference dispatch, and rendering.

Phase 1: Table and ball detection (computer vision pipeline)

1.1 Table detection and homography

Table detection combines color segmentation with geometric verification. The pool table's felt provides a strong color signal in HSV space.

Step 1 — Green felt segmentation. Convert each frame to HSV and threshold for the felt color. (GitHub) Standard green felt falls in H: 35–85, S: 50–255, V: 50–255. For blue felt, shift hue range to H: 90–130. Apply morphological operations (erosion then dilation with a 5×5 kernel) to clean noise.

```
python
```

```
hsv = cv2.cvtColor(frame, cv2.COLOR_BGR2HSV)
lower_green = np.array([35, 50, 50])
upper_green = np.array([85, 255, 255])
mask = cv2.inRange(hsv, lower_green, upper_green)
mask = cv2.morphologyEx(mask, cv2.MORPH_CLOSE, np.ones((5,5), np.uint8))
contours, _ = cv2.findContours(mask, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
table_contour = max(contours, key=cv2.contourArea)
```

Step 2 — Corner extraction. Apply cv2.approxPolyDP to the table contour to find four corner points. Alternatively, use Canny edge detection (thresholds 50/150) followed by probabilistic Hough line transform (cv2.HoughLinesP with minLineLength=100, maxLineGap=10), OpenCV cluster detected lines by angle into horizontal/vertical groups, merge nearby parallel lines, and compute four intersection points.

Step 3 — Homography computation. Map detected corners to a standard top-down rectangle matching regulation table dimensions. GitHub A 9-foot table playing surface is **254 × 127 cm** (100 × 50 inches); 8-foot is 224 × 112 cm; 7-foot is 200 × 100 cm. The homography matrix H transforms any point on the table between image pixels and physical coordinates. ResearchGate

```
python

src_pts = np.float32([corner1, corner2, corner3, corner4])  # detected image corners
dst_pts = np.float32([[0,0], [2540,0], [2540,1270], [0,1270]])  # mm coordinates
H = cv2.getPerspectiveTransform(src_pts, dst_pts)
bird_eye = cv2.warpPerspective(frame, H, (2540, 1270))
```

The homography is a 3×3 matrix requiring exactly 4 point correspondences (8 equations for 8 unknowns). GitHub **If the camera is stationary (phone on tripod), compute H once**—the pix2pockets project achieved 333fps for coordinate transformation versus 10fps when recomputing per frame. GitHub If handheld, recompute every 10–30 frames or when AR tracking detects significant camera motion.

Step 4 — Pocket positions. Once table corners are known, pocket positions are deterministic. Corner pockets sit at the four corners with openings of **11.4–11.7 cm** (4.5–4.625 inches). Side pockets sit at the midpoints of the long rails with openings of **12.7–13.0 cm** (5.0–5.125 inches). Define each pocket as a position plus an acceptance cone (±15° for corners, ±10° for sides).

1.2 Ball detection with YOLOv8-nano

Model selection. YOLOv8n (nano) from Ultralytics is the optimal choice: **3.2M parameters, 8.7B FLOPs,** GitHub **~6MB model size**. On iPhone (CoreML + Neural Engine), it achieves ~33fps. GitHub On Android with GPU delegate, expect 15–30ms inference. For newer projects, YOLO11n offers improved small-object detection via the C3k2 block and C2PSA self-attention module; arXiv YOLO26 (September 2025) provides NMS-free inference for simpler deployment. arXiv

Class design. Two viable approaches exist:

- **17-class approach** (recommended for full identification): One class per ball number (1–15) plus cue ball plus 8-ball. Requires more training data (~100+ examples per class) but eliminates the need for a separate classification step.
- **4-class approach** (simpler, faster): Classes are {solid, stripe, cue_ball, eight_ball}. Requires a secondary color classifier to identify specific ball numbers. Achieves higher detection mAP with less data.

Training dataset. Start by combining existing Roboflow datasets: Ben Gann's 986 images, Roboflow ipool's 746 images (12 classes), Roboflow and the Billiard Pool 488-image set. Supplement with 200–500 custom photos from your target environment using various angles, lighting conditions, and ball configurations. The pix2pockets project achieved **AP50 of 91.2%** with only 195 images; arXiv ResearchGate the pool-ball-tracking project achieved 30fps with 210 images. For production quality, target **1,000–2,000 total annotated images**. Use Roboflow Annotate for AI-assisted labeling.

Training configuration:

python

```
from ultralytics import YOLO

model = YOLO("yolov8n.pt")  # pretrained on COCO
model.train(
    data="pool_balls.yaml",
    epochs=100,
    imgsz=640,        # or 1280 for better small-ball detection at distance
    batch=16,
    patience=20,      # early stopping
    augment=True,     # mosaic, HSV jitter, flips, scale
    hsv_h=0.015,      # hue augmentation (critical for lighting variation)
    hsv_s=0.7,        # saturation augmentation
    hsv_v=0.4,        # value augmentation
    degrees=15,       # rotation augmentation
    scale=0.5,        # scale augmentation
    mosaic=1.0,       # mosaic augmentation (disabled last 10 epochs automatically)
)
```

Export for mobile:

python

```
model.export(format="coreml")                # iOS → .mlpackage (uses Neural Engine)
model.export(format="tflite")                # Android → .tflite
model.export(format="tflite", int8=True, data="pool_balls.yaml")  # INT8 quantized
model.export(format="onnx")                  # Unity/cross-platform
```

Model sizes after quantization: FP32 ~6.2MB, FP16 ~3.2MB, INT8 ~1.8MB. **FP16 is the sweet spot for GPU inference** Medium (negligible accuracy loss, 1.5–2× faster). INT8 is optimal for NPU inference on Snapdragon (via QNN) or Apple Neural Engine.

1.3 Ball detection fallback: Hough circles plus color filtering

As a lightweight alternative or validation layer alongside YOLO, use OpenCV's Hough circle transform:

```
python

gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
gray_blurred = cv2.GaussianBlur(gray, (9, 9), 2)
circles = cv2.HoughCircles(
    gray_blurred, cv2.HOUGH_GRADIENT,
    dp=1, minDist=50, param1=50, param2=30,
    minRadius=10, maxRadius=40  # adjust based on camera distance
)
```

The minDist parameter is the most critical to tune— PyImageSearch it prevents detecting the same ball twice. Known limitations: false positives from specular highlights, degraded accuracy with motion blur, and inability to classify balls. Ucsd Use this as a secondary validation: if YOLO detects a ball and Hough confirms a circle at the same location, confidence increases.

1.4 Ball classification by color

For the 4-class YOLO approach (or to distinguish specific ball numbers within solid/stripe categories), implement HSV-based color classification on detected ball patches. The POOL-AID system (UCSD) achieved **98.64% classification accuracy** with this approach. Ucsd

Color mapping for standard pool balls:

Ball(s)	Color	HSV Hue Range	Key Discriminators
1, 9	Yellow	H: 20–40	Bright, high saturation
2, 10	Blue	H: 100–130	Distinct hue range
3, 11	Red	H: 0–10, 170–180	Wraps around hue cylinder
4, 12	Purple	H: 130–160	Unique hue
5, 13	Orange	H: 10–25	Between red and yellow
6, 14	Green	H: 40–80	Overlaps with table felt—use ROI masking
7, 15	Maroon	H: 0–15, low S	Low saturation distinguishes from red
8	Black	V < 50	Very low value channel
Cue	White	S < 30, V > 200	Low saturation, high value

Solid vs. stripe discrimination. Compute the white-pixel ratio within each ball patch. Stripes have a band of white that significantly increases their white-to-color ratio. Apply `cv2.inRange(hsv, [0, 0, 200], [180, 30, 255])` for white pixels. If `white_ratio > 0.30`, classify as stripe; otherwise, solid. This threshold is robust and the core discriminator between solids and stripes.

Global consistency with bipartite matching. After computing per-ball classification probabilities, apply the Hungarian algorithm (scipy.optimize.linear_sum_assignment) across all detected balls to ensure globally consistent assignments—no two balls should have the same identity.

Phase 2: Game state recognition and tracking

2.1 Ball tracking between frames

Implement a lightweight SORT (Simple Online and Realtime Tracking) variant. Full DeepSORT is unnecessary for pool balls because ball appearances are distinctive and balls are mostly stationary between shots.

Per-frame tracking pipeline:

1. YOLO produces detections: `List[Detection(x, y, w, h, class_id, confidence)]`
2. Kalman filter predicts each existing track's position forward one timestep
3. Hungarian algorithm matches detections to tracks using IoU-based cost matrix
4. Color-based identity verification confirms or corrects class assignments

5. Kalman state updates with matched measurements
6. Unmatched detections create new tracks; unmatched tracks increment their "lost" counter

Kalman filter configuration for pool balls. State vector: `[x, y, vx, vy]`. Measurement vector: `[x, y]`. Use a constant-velocity motion model. Set process noise low (balls are mostly stationary) and measurement noise moderate (detection jitter). During active shots (detected ball motion), temporarily increase process noise. After collisions, reset Kalman state for affected balls since the linear assumption breaks down.

Position smoothing for AR display. Apply exponential moving average: `display_pos = α × detected_pos + (1-α) × prev_display_pos` with α = 0.3–0.5. This eliminates jitter in the AR overlay without introducing perceptible lag.

2.2 Pocketed ball detection

Define six pocket zones as circular regions (radius ~8cm) centered on each pocket position. Track balls entering these zones over consecutive frames. If a ball enters a pocket zone and then disappears for 3+ consecutive frames, mark it as pocketed. Maintain a persistent game state dictionary: `{ball_id: "on_table" | "pocketed" | "in_hand"}`.

2.3 Game state machine

The 8-ball game state machine tracks:

- **Assignment phase:** Before any ball is pocketed after the break, both solids and stripes are unassigned. The first legally pocketed ball (not the 8-ball) determines the player's group.
- **Player assignment:** `{player_1: "solids"|"stripes"|None, player_2: opposite}`
- **Legal target balls:** Based on assignment, the player must hit their group first. Once all group balls are pocketed, the 8-ball becomes the legal target.
- **Ball-in-hand detection:** If the cue ball is pocketed (enters any pocket zone) or is not detected on the table, flag ball-in-hand state.
- **Turn tracking:** Detect shot events (cue ball velocity spike followed by deceleration) and shot outcomes (ball pocketed or not) to track whose turn it is.

Implement this as a finite state machine with states: `BREAK → OPEN_TABLE → ASSIGNED_PLAY → SHOOTING_8 → GAME_OVER`. Transitions are triggered by detected events (ball pocketed, foul, 8-ball pocketed).

Phase 3: Physics engine and shot calculation

3.1 Physical constants for regulation play

Hardcode these constants (sourced from Dr. Dave Alciatore, Colorado State University, and Mathavan et al.,

2010):

```
BALL_RADIUS        = 0.028575  # meters (1.125 inches, standard 2.25" diameter)
BALL_MASS          = 0.170     # kg (6 oz)
BALL_MOMENT_INERTIA = (2/5) * BALL_MASS * BALL_RADIUS²  # solid sphere

TABLE_LENGTH       = 2.54      # meters (100 inches, 9-foot table)
TABLE_WIDTH        = 1.27      # meters (50 inches)
CORNER_POCKET_WIDTH = 0.117    # meters (4.625 inches)
SIDE_POCKET_WIDTH   = 0.130    # meters (5.125 inches)

MU_BALL_BALL       = 0.06      # ball-ball friction coefficient
MU_SLIDE           = 0.20      # ball-cloth sliding friction
MU_ROLL            = 0.01      # ball-cloth rolling resistance
MU_CUSHION         = 0.14      # ball-cushion sliding friction

COR_BALL_BALL      = 0.93      # coefficient of restitution, ball-ball
COR_BALL_CUSHION   = 0.75      # coefficient of restitution, ball-cushion

GRAVITY            = 9.81      # m/s²
CUSHION_HEIGHT     = 7 * BALL_RADIUS / 5  # = 0.040 meters
```

3.2 Ball motion equations

A ball on the table exists in one of four states: **sliding** (contact point slipping), **rolling** (pure roll, no slip), **spinning** (stationary but spinning in place), or **stationary**.

Sliding phase. The contact velocity $u = v - R \times \omega$ is nonzero. Friction acts opposite to the sliding direction with constant magnitude:

```
acceleration:          a = -μ_slide × g × û       (where û = unit vector of contact velocity)
angular acceleration:  α = -(5μ_slide × g)/(2R) × û
```

The trajectory during sliding is **parabolic** (constant acceleration vector). Sliding transitions to rolling when the contact velocity reaches zero. Transition time for a center-struck ball: $t_1 = 2|v_0|/(7 \times \mu\_slide \times g)$.

Rolling phase. Pure rolling constraint: $v = R \times \omega$. Deceleration is:

```
a = -μ_roll × g × v̂    (where v̂ = unit velocity vector)
```

Trajectory is a straight line with constant deceleration. Distance to stop: $d = v^2/(2 \times \mu\_roll \times g)$. The ball stops when velocity falls below a threshold (~0.001 m/s).

3.3 Event-based simulation engine

The discrete time-stepping approach (advancing positions by Δt each frame) misses collisions at high speeds and requires very small timesteps (~0.1ms) for accuracy. The **event-based algorithm** (Leckie & Greenspan, used in Pooltool) is far superior for billiards: github Evan Kiefl

pseudocode

```
function simulateShot(balls, initial_cue_velocity):
    applyCueImpact(cue_ball, initial_cue_velocity)

    while any ball is moving:
        // Calculate time to every possible next event
        t_min = infinity
        next_event = null

        for each pair (ball_i, ball_j):
            t = timeToCollision(ball_i, ball_j)  // solve quadratic equation
            if t < t_min: t_min = t; next_event = BallCollision(i, j)

        for each (ball, cushion):
            t = timeToCushion(ball, cushion)  // solve linear equation
            if t < t_min: t_min = t; next_event = CushionCollision(ball, cushion)

        for each ball:
            t = timeToTransition(ball)  // sliding→rolling or rolling→stationary
            if t < t_min: t_min = t; next_event = StateTransition(ball)

        for each (ball, pocket):
            t = timeToPocket(ball, pocket)
            if t < t_min: t_min = t; next_event = Pocketed(ball, pocket)

        // Advance ALL balls analytically to t_min
        for each ball:
            advanceAnalytically(ball, t_min)

        // Resolve the event
        resolveEvent(next_event)

    return all_trajectory_points
```

Between events, ball motion is analytically solvable (parabolas for sliding, straight lines for rolling), so no numerical integration errors accumulate. The **time-to-collision between two balls** requires solving a quadratic equation: given positions $p_1(t)$, $p_2(t)$ as functions of time, find t where $|p_1(t) - p_2(t)| = 2R$.

3.4 Ball-ball collision resolution

For two equal-mass balls with coefficient of restitution e:

```
// At collision, compute normal vector between centers
n̂ = normalize(ball_2.pos - ball_1.pos)

// Decompose velocities into normal and tangential components
v1n = dot(v1, n̂)    // normal component
v2n = dot(v2, n̂)

// Exchange normal components (with COR)
v1n_new = ((1-e)/2) × v1n + ((1+e)/2) × v2n
v2n_new = ((1+e)/2) × v1n + ((1-e)/2) × v2n

// Reconstruct velocity vectors (tangential unchanged)
v1_new = v1 + (v1n_new - v1n) × n̂
v2_new = v2 + (v2n_new - v2n) × n̂
```

For a stun shot (no spin), the cue ball and object ball separate at approximately **85°** (theoretical 90° reduced by COR of 0.93). The object ball speed is `v_OB = v_CB × cos(cut_angle)`. The cue ball residual speed is `v_CB_after = v_CB × sin(cut_angle)`. Real World Physics Problems

Throw effect. Ball-ball friction during the ~250μs contact duration pushes the object ball off the expected line. AzBilliards Dr. Dave Pool Info Maximum throw angle ≈ `arctan(μ_ball) ≈ arctan(0.06) ≈ 3.4°` at very low speeds. At typical game speeds, throw is approximately **1–2°**. BIZU BILLIARDS For the first implementation, this can be omitted and added as a refinement.

3.5 Cushion collision resolution

The mirror-image method handles basic bank shots: reflect the target pocket across the rail to find the aim point.

```
pseudocode

function bankShotAimPoint(ball_pos, pocket_pos, rail):
    mirror_pocket = reflectAcrossLine(pocket_pos, rail)
    aim_point = lineIntersection(ball_pos, mirror_pocket, rail)
    return aim_point
```

For physics accuracy, the cushion bounce uses: `v_normal_out = -COR_cushion × v_normal_in` (normal component reverses with energy loss) and `v_tangential_out ≈ v_tangential_in` (tangential preserved, minus cushion friction). Running english (sidespin in the direction of ball travel along the rail) widens the rebound angle; reverse english shortens it. Dr. Dave Pool Info Pool Table Portfolio

3.6 The ghost ball aiming system

The ghost ball is the central aiming concept. Pool Table Portfolio App Store Given target ball position OB and target pocket position P:

```
direction = normalize(P - OB)
ghost_ball_center = OB - direction × 2R    // one ball diameter behind OB
contact_point = OB - direction × R         // surface contact point
```

The cue ball must arrive at ghost_ball_center to send the object ball toward the pocket. Cornilleau The cut angle is the angle between the cue ball approach line and the line of centers: cut_angle = arccos(dot(normalize(GB - CB), normalize(OB - GB))).

Cut angle difficulty table (from Dr. Dave Alciatore):

Hit Fraction	Cut Angle	Difficulty
Full ball	0°	Easiest—straight in
¾ ball	14.5°	Easy
½ ball	30°	Medium
¼ ball	48.6°	Hard
⅛ ball	~60°	Very hard—thin cut

3.7 Cue ball path prediction after contact

The 30° rule (Coriolis/Dr. Dave): For a rolling cue ball (natural topspin from rolling), the cue ball deflects approximately **30° from the impact line** for medium cut angles. Maximum deflection is **33.7°** at a cut angle of **28.1°**. The actual path is parabolic during the sliding phase (as spin-induced friction curves the ball), then straightens once pure rolling resumes.

For draw shots (backspin), the cue ball reverses along the tangent line. Sfbilliards For follow shots (topspin), the cue ball continues forward. For stun shots (no spin at contact), the cue ball travels along the pure tangent line at 90° to the object ball direction. Wikipedia

---

Phase 4: AR overlay and guidance rendering

4.1 Coordinate mapping: 2D detections to 3D AR world

Ball detection produces 2D pixel coordinates. AR rendering requires 3D world coordinates on the table plane. Since the table is a known planar surface (effectively Z=0 in table coordinates), the full 3D projection simplifies to the 3×3 homography matrix computed in Phase 1.

From pixel to table coordinates: Given a detected ball at pixel (u, v), compute table position as $[X, Y, 1]^T = H^{-1} \times [u, v, 1]^T$, then normalize by the third component.

From table coordinates to AR world: Use the AR framework's raycasting. On ARCore: `Frame.hitTest(x, y)` returns a `HitResult` with a 3D world `Pose`. (Google) On ARKit: `session.raycast(from:allowing:alignment:)` returns `ARRaycastResult.worldTransform`. In Unity AR Foundation: `ARRaycastManager.Raycast(screenPoint, hits, TrackableType.Planes)`. The AR framework handles the camera-to-world transform using its internal tracking.

Hybrid approach (recommended). Compute homography for accurate ball positioning in table-space physics. Use AR framework raycasting to anchor the table plane in 3D world space. Combine both: run physics in 2D table coordinates, then project results back to screen coordinates via the homography for rendering.

4.2 Camera intrinsics

Both ARCore and ARKit provide pre-calibrated camera intrinsics per frame—**no manual camera calibration is needed**. On ARCore: `Camera.getImageIntrinsics()` provides focal length (fx, fy) and principal point (cx, cy). On ARKit: `ARCamera.intrinsics` provides a 3×3 intrinsic matrix. In Unity: `ARCameraManager.TryGetIntrinsics()` returns focal length, principal point, and resolution. (Unity)

If bypassing the AR framework for custom processing, use OpenCV camera calibration with a checkerboard pattern (`cv2.calibrateCamera` using 10+ checkerboard images from different angles), then undistort frames with `cv2.undistort()` (OpenCV-Python Tutorials) or, for better performance, pre-compute remap matrices with `cv2.initUndistortRectifyMap()`. (OpenCV)

4.3 Rendering the AR overlay

Trajectory lines. Render the predicted cue ball path as a green semi-transparent line, the object ball path as a yellow line, and the ghost ball position as a transparent circle at the aiming point. Use line widths of 3–5px for visibility without obscuring the table.

On iOS (Metal/SceneKit): Create `SCNNode` objects with custom `SCNGeometry` from vertex data. For lines with controllable width (since `GL_LINES` renders 1px only), generate screen-space quads: expand line endpoints into triangulated strips using the line direction normal in the vertex shader. Set `SCNMaterial.transparency = 0.6` and `.blendMode = .alpha`.

On Android (OpenGL ES): Render the camera background texture on a full-screen quad using `GL_TEXTURE_EXTERNAL_OES`. Draw trajectory quads with custom vertex/fragment shaders. Enable alpha

blending: `glEnable(GL_BLEND); glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA)`. Use a fragment shader outputting `vec4(0.0, 1.0, 0.0, 0.6)` for semi-transparent green guide lines.

In Unity (URP): Use Unity's `LineRenderer` component for trajectory paths with custom material (Unlit/Transparent shader). For the ghost ball, instantiate a semi-transparent sphere prefab at the computed world position. Use Unity's Universal Render Pipeline (URP) with custom shader graph materials for glow effects on guide lines. Canvas UI overlays show shot recommendations, scores, and game state.

Key rendering principle: Decouple detection framerate (10–15fps) from rendering framerate (30–60fps). Between detections, interpolate ball positions using Kalman-predicted states and gyroscope-compensated camera motion. This ensures smooth, jitter-free AR overlay regardless of ML inference speed.

4.4 Visual guidance elements to render

- **Ghost ball:** Semi-transparent circle at the required cue ball contact position
- **Aiming line:** Line from current cue ball position through ghost ball, extending to the cue stick direction
- **Object ball path:** Line from object ball through pocket, showing expected trajectory
- **Cue ball path after contact:** Predicted post-collision cue ball trajectory (parabolic for rolling shots, straight for stun shots)
- **Cushion reflection paths:** For bank shots, show the reflection point on the rail and resulting trajectory
- **Pocket highlights:** Glow effect on the recommended target pocket
- **Ball labels:** Small overlay showing ball numbers/colors when detection confidence is high
- **Shot difficulty indicator:** Color-coded (green/yellow/red) based on cut angle and distance

---

Phase 5: Cue stick tracking and real-time aiming

5.1 Cue detection approaches

Approach 1 — YOLO detection (recommended). Include "cue_stick" and "cue_tip" as classes in the YOLO training data. The cue appears as an elongated bounding box; the tip position is extracted from the end nearest the cue ball. Yang & Yan (2024, IGI Global) achieved **95% detection accuracy** for cue stick tip and bridge hand. Key challenge: the detection "bounces around" when the player is moving the cue during aiming—apply temporal smoothing (EMA with α=0.2).

Approach 2 — Hough line detection. After cue ball detection, search for long lines in the region around the cue ball using `cv2.HoughLinesP`. Filter candidates by: length > 150px, one endpoint within 50px of cue ball center, brown/black color consistent with a cue stick. Compute the cue angle from the filtered line's endpoints.

```
python
```

```
lines = cv2.HoughLinesP(edges, 1, np.pi/180, threshold=80, minLineLength=100, maxLineGap=15)
for line in lines:
    x1, y1, x2, y2 = line[0]
    length = np.sqrt((x2-x1)**2 + (y2-y1)**2)
    dist_to_cue = min(distance(x1,y1,cue_x,cue_y), distance(x2,y2,cue_x,cue_y))
    if length > 150 and dist_to_cue < 50:
        cue_angle = np.arctan2(y2-y1, x2-x1)
```

Approach 3 — Hybrid. Use YOLO for initial cue detection at 10–15fps, then use Hough line detection as lightweight inter-frame tracking at full camera framerate. The YOLO result anchors the search region; Hough refines the angle estimate frame-to-frame.

5.2 Real-time aim line projection

Once the cue stick angle is detected, project the aiming line forward from the cue ball center in the cue direction. Compute where this line intersects with each object ball (circle intersection test: find the distance from the line to each ball center; if distance < 2R, a collision will occur). Show the first ball the cue ball will hit, the resulting trajectories for both balls, and whether the target ball will reach a pocket.

Update this projection at camera framerate (30fps) by tracking the cue angle with temporal smoothing. As the player adjusts their aim, the projected trajectory updates in real-time—providing immediate visual feedback on shot accuracy.

Phase 6: Shot recommendation AI and optimal play engine

6.1 Shot scoring algorithm

For each legal target ball, evaluate every possible pocket (6 pockets per ball = up to 42 candidate shots for 7 remaining balls). Score each candidate using a weighted multi-factor evaluation:

Pocketability (50% weight). Compute the ghost ball position and cut angle. Calculate the effective pocket opening at the approach angle: `effective_width = pocket_width × cos(pocket_angle)`. Compute the angular margin of error: `margin = arctan(effective_width / (2 × OB_to_pocket_distance)) - ball_radius_angle`. Factor in distance penalties—longer CB-to-OB and OB-to-pocket distances reduce accuracy. Cut angles beyond 60° are nearly impossible; above 45° are difficult.

Path clearance (15% weight). Check if the CB-to-ghost-ball path and OB-to-pocket path are unobstructed by other balls. For each ball on the path, compute the perpendicular distance from the line; if less than 2R, the path is blocked. A blocked path scores zero.

Position play (25% weight). Simulate the shot at medium speed and predict the cue ball's final resting position. From that position, count the number of high-scoring available next shots. Better position play yields more

options for the next shot. This is where the physics engine (Phase 3) provides critical value—accurate cue ball position prediction after contact.

Safety margin (10% weight). Assess scratch risk: predict whether the cue ball might reach a pocket after hitting the object ball. Evaluate whether a missed shot leaves the opponent with an easy run. Penalize shots where the cue ball trajectory passes near a pocket.

```pseudocode
function recommendBestShot(game_state):
    candidates = []
    for ball in game_state.legal_target_balls:
        for pocket in game_state.pockets:
            ghost = computeGhostBall(ball.pos, pocket.pos)
            if isPathClear(cue_ball.pos, ghost, all_balls):
                if isPathClear(ball.pos, pocket.pos, all_balls):
                    score = evaluateShot(cue_ball, ball, pocket, game_state)
                    candidates.append((ball, pocket, score, ghost))

    candidates.sort(by=score, descending=True)
    return candidates[0]  // highest-scoring shot
```

6.2 Multi-shot planning

For advanced guidance, implement a depth-limited search that considers sequences of 2–3 shots. After evaluating the immediate shot, simulate the resulting table state and recursively evaluate the best next shot from the predicted cue ball position. This captures the critical skill of "playing position"—choosing a slightly harder current shot that leaves a much better next shot. Limit search depth to 2–3 moves to keep computation under 100ms on mobile.

6.3 Bank shot and kick shot computation

When no direct pocket shot is available, evaluate bank shots (object ball off one or more cushions into a pocket) and kick shots (cue ball off cushions to reach the object ball). For single-rail banks, use the mirror method. For multi-rail kicks, apply mirror reflections iteratively—mirror the pocket across the second rail, then mirror that result across the first rail. Score these alternatives with a higher difficulty penalty than direct shots.

Phase 7: Integration, optimization, and polish

7.1 Complete data pipeline

The integrated system processes each frame through this pipeline:

```
Camera (YUV_420_888, 1080p, 30fps)
  → Ring buffer (3 frames)
  → [ML Thread] YUV→RGB → Crop to table ROI → Resize 320×320 → YOLO inference (~10-30ms)
  → [Tracking Thread] Hungarian matching → Kalman update → Color verification → Game state update
  → [Physics Thread] Ghost ball computation → Trajectory simulation → Shot scoring
  → [Render Thread] Table-to-screen projection → Line rendering → UI overlay → Display
```

Total pipeline latency target: **<50ms from frame capture to AR overlay update** on flagship phones (2024+). Mid-range devices may achieve 15–20fps processing with tracking-based interpolation maintaining smooth 30fps display.

7.2 Performance optimization techniques

Adaptive frame skipping. Monitor inference latency dynamically. If <50ms, process every frame. If 50–100ms, process every 2nd frame. If >100ms, process every 3rd frame. Always interpolate between detection results for smooth rendering.

ROI tracking. After initial table detection, crop subsequent frames to the table region only (saves 40–60% of pixels processed). Recompute full-frame detection periodically or when AR tracking detects significant camera motion.

Model warm-up. Run a dummy inference during the splash screen. The first inference allocates GPU buffers, compiles shaders, and uploads weights—causing several seconds of delay if done during gameplay.

Memory management. Pre-allocate all input/output tensors before the inference loop. Use a ring buffer for camera frames. Recycle OpenCV Mat objects. On Android, keep the hot path in native C++ to avoid garbage collection pauses.

Thermal throttling mitigation. Monitor `PowerManager.getCurrentThermalStatus()` on Android. At `THERMAL_STATUS_LIGHT`, reduce inference frequency. At `THERMAL_STATUS_MODERATE`, switch to lower input resolution (320→256). At `THERMAL_STATUS_SEVERE`, pause ML inference and rely on tracking-only mode. Target sustained power draw under 7W. Research shows thermal throttling can reduce frame rate by **34%** (from 35 to 23fps) on sustained load.

Battery optimization. FP16 on GPU is the optimal tradeoff for CNNs (per Ollion, 2025). NPU inference is the most power-efficient per TOPS. Reduce camera resolution to 720p capture when possible. Consider an "eco mode" that reduces inference to 5fps for casual use.

7.3 Testing and debugging pipeline

CV pipeline testing. Record raw camera frames plus sensor data (gyroscope, accelerometer) to video files. Build a replay mode that feeds recorded video through the detection→tracking→physics pipeline without a live camera. This enables deterministic regression testing and offline development.

Performance profiling. On Android, use Android Studio Profiler for CPU/memory/GPU/energy analysis, Perfetto for system-wide traces, and the TFLite Benchmark Tool for model-specific latency measurement. On iOS, use Xcode Instruments (Time Profiler, Metal System Trace, Core Animation FPS) and Core ML Profiler. Set up automated benchmarks that alert on >10% latency regression.

Debug overlay. Render detection bounding boxes, tracking IDs, confidence scores, inference time, and current game state directly on the camera preview. Toggle this overlay via a developer settings menu. Log per-frame metrics (detection count, inference ms, tracking confidence) to JSON for offline analysis.

7.4 Platform-specific deployment notes

iOS deployment. Export the YOLO model as `.mlpackage` via CoreML. CoreML automatically schedules across CPU, GPU, and Apple Neural Engine for optimal performance. Use the Vision framework (`VNCoreMLRequest`) for seamless integration. ARKit provides world-class surface detection—configure `ARWorldTrackingConfiguration` with `planeDetection = [.horizontal]`. Use SceneKit (`ARSCNView`) for flexible custom overlay rendering.

Android deployment. Export the YOLO model as `.tflite` with FP16 or INT8 quantization. Use LiteRT (the new name for TensorFlow Lite) with GPU delegate for inference. Important caveat: ARCore and the ML model compete for GPU resources—on devices with NPU support (Snapdragon 8 Gen 2+), route ML inference to the NPU via the QNN delegate to avoid GPU contention. Use CameraX for zero-copy camera frame access (`ImageProxy`).

Cross-platform via Unity. Use AR Foundation 6.x for unified ARCore/ARKit abstraction. Use Unity Sentis (v2.4+) for ONNX model inference. Export YOLO without NMS in the model graph and implement NMS in C# (Sentis CPU-based NMS causes frame drops). Use GPU compute backend for inference. Render overlays with URP LineRenderer and custom shader graph materials.

7.5 Why not use Gemini Vision API for real-time processing

Cloud APIs are fundamentally unsuitable for real-time AR. Gemini 2.5 Flash averages **8–12 seconds per image** with 15-second P95 latency. Even the Live API supports only 1fps video with 10-minute session limits. The minimum requirement for AR is 10fps detection (~100ms budget). On-device YOLO inference at 10–30ms is **100–1000× faster** than any cloud API. Use Gemini only for offline tasks: training data generation, one-time scene understanding in photo mode, or post-game analysis. All real-time detection must run on-device.

Conclusion: key technical decisions and novel insights

The most critical architectural decision is running **all ML inference on-device** rather than in the cloud—this is non-negotiable for real-time AR. YOLOv8-nano provides the best accuracy-to-speed ratio for pool ball detection, achieving >90% mAP50 with as few as 200 training images while running in 5–30ms on modern phones. The event-based physics engine (pioneered by Leckie & Greenspan, implemented in Pooltool) is

essential for accurate trajectory prediction—discrete time-stepping misses collisions and accumulates numerical errors unacceptably for a guidance system.

The **ghost ball aiming system** is the core UX innovation: by showing exactly where the cue ball needs to arrive at contact, it translates abstract physics into intuitive visual guidance that beginners can immediately act on. The shot recommendation engine's value comes less from pocketability calculation (which experienced players can assess visually) and more from **position play prediction**—showing where the cue ball will end up after the shot, which is the hardest skill for human players to develop intuitively.

Three non-obvious technical challenges will determine production quality. First, **solid-vs-stripe discrimination** requires computing white pixel ratios within detected ball patches—color alone is insufficient since solids and stripes share the same hue ranges. Second, **cue stick detection instability** during player movement requires aggressive temporal smoothing (EMA with α=0.2) and potentially a separate lightweight tracking model for the cue tip. Third, **GPU contention between ARCore and ML inference** on Android necessitates routing ML to the NPU where available, or carefully time-slicing GPU access with frame-aligned scheduling. These challenges are solvable with the approaches documented above, and the commercial success of DrillRoom proves the entire system is achievable on today's smartphone hardware.