collision-detector

Step-by-Step Explanation

A custom A-Frame component that prevents the camera from walking through walls. It uses invisible rays, snap-back logic, and boundary clamping. Works on desktop and Meta Quest VR.

`THREE.Raycaster`	`lastPosition`	`tock()`	`class="wall"`
Fires invisible rays towards walls	Remembers the last safe player spot	Runs after movement each frame	Tags any element as collidable

1 Create the custom component

```
AFRAME.registerComponent('collision-detector', {
```

This tells A-Frame: *"Create a new custom component called* **`collision-detector`***."* You then attach it to the player rig:

```
<a-entity id="rig" collision-detector>
```

2 Set adjustable values (schema)

```
schema: {
  distance: { type: 'number', default: 0.6 },
  xLimit:   { type: 'number', default: 3.8 },
  zLimit:   { type: 'number', default: 19.0 }
}
```

These are the three settings you can tune:

- `distance: 0.6` — player must stay at least **0.6 metres** from any wall
- `xLimit` — limits movement left and right
- `zLimit` — limits movement forward and backward

3 Save the player's safe position

```
this.lastPosition = new THREE.Vector3();
this.el.object3D.getWorldPosition(this.lastPosition);
```

At startup the code remembers where the player is standing. This becomes the first **safe position**. If the player later hits a wall, they are moved back here.

4 Prepare helper objects

```
this.origin   = new THREE.Vector3();
this.yAxis    = new THREE.Vector3(0, 1, 0);
this.camQuat  = new THREE.Quaternion();
this.camEuler = new THREE.Euler();
```

These are Three.js helper objects created once and reused every frame (efficient). They help calculate:

- Where the player is in the world
- Which direction the player is facing
- How to rotate the rays to match the player's view

5 Find all the walls

```
document.querySelectorAll('.wall')
```

This searches the scene for every element tagged with **`class="wall"`**. Only these objects are tested for collision — everything else is ignored, which keeps performance fast.

```
<a-box class="wall" position="-4 1.6 -9" ...></a-box>
```

> **TIP** Any A-Frame element can be a wall — just add class="wall" to it.

6 Find the camera (important for Quest VR)

```
this.cameraEl = this.el.querySelector('[camera]') ||
                this.el.querySelector('a-camera');
```

In a VR headset like the Meta Quest, the *rig* entity never rotates — the *camera* inside it rotates as you turn your head. This line finds the camera so the component can read the real facing direction.

> **TIP** Without this, the rays would always fire forward along world Z — ignoring head turns.

7 tock() runs every frame

```
tock() {
```

tock() is called after all movement has already happened. This is the critical difference from **tick()**:

- **tick()** → runs at the same time as movement — sees the OLD position ✗
- **tock()** → runs AFTER movement — sees the NEW position ✓

> **TIP** Using tock() is what makes the collision actually work.

8 Get the direction the player is facing

```
this.cameraEl.object3D.getWorldQuaternion(this.camQuat);
this.camEuler.setFromQuaternion(this.camQuat, 'YXZ');
yRot = this.camEuler.y;
```

This reads the camera's current Y rotation in world space — i.e., which way the player is looking left or right. The rays are then rotated to match, so collision always detects walls relative to the player's view.

9 Create four direction vectors

```
const directions = [
  forward,
  forward.clone().multiplyScalar(-1),  // backward
  right.clone().multiplyScalar(-1),    // left
  right                                // right
];
```

Four rays spread around the player like compass points — front, back, left, right. This creates a circular protective zone so the player cannot sneak through a wall from any angle.

10 Cast invisible rays towards walls

```
rc.set(this.origin, dir.normalize());
const hits = rc.intersectObjects(this.wallObjects, true);
```

A **ray** is like an invisible laser line fired from the player outward. `intersectObjects()` returns every wall the ray passes through, along with the distance to each hit.

11 Check if the wall is too close

```
if (hits.length > 0 &&
    hits[0].distance < this.data.distance) {
  collision = true;
}
```

If the nearest wall is closer than **`0.6 metres`**, the collision flag is set to **`true`**. The loop then breaks early — one hit is enough.

12 Snap the player back

```
if (collision) {
  position.copy(this.lastPosition);  // go back to safety
}
```

If a collision was detected, the player is instantly returned to their last safe position. From the user's perspective this feels like hitting an invisible wall — they simply cannot move further in that direction.

13 Save the new safe position

```
else {
  this.lastPosition.copy(position);  // save as new safe spot
}
```

If there is no collision, the player's current position becomes the new lastPosition. Every frame the safe point is updated — so snap-back always returns to *just before* the wall, not all the way back to the start.

14 Add a hard boundary clamp

```
position.x = THREE.MathUtils.clamp(
  position.x, -this.data.xLimit, +this.data.xLimit
);
position.z = THREE.MathUtils.clamp(
  position.z, -this.data.zLimit, +this.data.zLimit
);
```

This is a second safety net. Even if the raycasting fails to catch a very fast movement through a thin wall, the clamp guarantees the player can never leave the defined scene rectangle.

> **TIP** Defence in depth: raycasting is the main check, clamping is the backup.

15 Update the debug display

```
document.getElementById('debug').textContent = ...
```

While testing, this line shows useful information on screen:

- Number of walls loaded
- Player X and Z position
- Whether a collision is currently active

> **TIP** Remove or hide this in the final live version.

Simple Summary

Player moves → system checks nearby walls using invisible rays → if too close, player snaps back → if safe, new position is saved.

This is a custom collision detector built in A-Frame. It uses invisible rays to check whether the player is close to walls. If a wall is detected too near, the player is moved back to the last safe position — stopping them from passing through the wall.

Schema attribute	What it does
`class="wall"`	Tag any element to make it collidable
`distance`	Min metres from wall before snap-back (default: 0.6)
`xLimit`	Max left/right boundary ± in metres (default: 3.8)
`zLimit`	Max forward/backward boundary ± in metres (default: 19.0)