```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="utf-8" />
<meta name="viewport"
content="width=device-width,initial-
scale=1.0" />
<title>DOORWAYS</title>
<style>

html,body{height:100%;margin:0;back-
ground:#000;overflow:hidden;touch-
action:none;-webkit-user-select:none;}
  canvas{display:block;margin:0
auto;background:#000;image-
rendering:pixelated;touch-action:none;}
  /* small fallback UI for browsers that
block canvas focus */

#hint{position:fixed;left:8px;bottom:8px;
color:#fff;font-family:monospace;font-
size:12px;opacity:0.7}
</style>
</head>
<body>
<canvas id="game" width="800"
height="600"></canvas>
<div id="hint">Tap to start • Joystick left
• Buttons right</div>
<script>
/* ============================
   DOORWAYS (single-file)
   - Top-down, 100 doors
   - Rush / Blitz / Bound entities
   - Overseer, lockers, flashlight, battery
   - Touch + keyboard controls
   - Procedural high-quality sounds (Web
Audio API)
   - Save as doorways.html and open in
browser
   ============================
*/
```

```
// ---- Canvas & basic constants
const canvas =
document×getElementById('game');
const ctx = canvas×getContext('2d');
const W = canvas.width, H =
canvas×height;

const MAX_DOORS = 100;
const PLAYER_SPEED = 2.2;
const FLASH_DRAIN = 0.03; // tuned
const JOY_DEADZONE = 6;

// ---- Audio (Web Audio API) -
procedural, layered, stereo
let audioCtx = null;
let masterGain = null;

function ensureAudio() {
  if (audioCtx) return;
  audioCtx = new (window.AudioContext
|| window.webkitAudioContext)();
  masterGain = audioCtx×createGain();
  masterGain×gain×value = 0.9; //
immersive volume (user requested)

masterGain.connect(audioCtx.destina-
tion);
}

// short helper to play an AudioBuffer (for
noise bursts)
function playBuffer(buffer, when=0,
vol=1, pan=0, playbackRate=1) {
  try {
    const src =
audioCtx×createBufferSource();
    src×buffer = buffer;
    src×playbackRate×value =
playbackRate;
    const gain = audioCtx×createGain();
    gain×gain×value = vol;
```

```
    const panner =
audioCtx×createStereoPanner();
    panner×pan×value = pan;

src.connect(gain).connect(panner).con-
nect(masterGain);
    src.start(audioCtx.currentTime +
when);
    return src;
  } catch(e){/* ignore on older browsers
*/}
}

// make a short noise buffer (white noise)
for clanks / whoosh bases
function makeNoiseBuffer(duration=0.5)
{
  const sr = audioCtx.sampleRate;
  const len = Math.floor(duration*sr);
  const buf =
audioCtx×createBuffer(1,len,sr);
  const data = buf×getChannelData(0);
  for (let i=0;i<len;i++){
    data[i] = (Math.random()*2-1) *
Math.pow(1 - i/len, 1.25); // fade out
  }
  return buf;
}

// procedural whoosh (sweep + filtered
noise)
function playWhoosh({when=0, pan=0,
duration=0.9, vol=0.6}={}) {
  const t0 = audioCtx.currentTime +
when;
  // sweep oscillator
  const osc =
audioCtx×createOscillator();
  const oscGain = audioCtx.createGain();
  osc×type = 'sawtooth';
  osc.frequency.setValueAtTime(120, t0);
```

```
osc.frequency.exponentialRampToValue-
AtTime(600, t0 + duration*0.7);
  oscGain.gain.setValueAtTime(0.02*vol,
t0);

oscGain.gain.exponentialRampToValue-
AtTime(0.0001, t0 + duration);
  const panner =
audioCtx×createStereoPanner();
  panner×pan×value = pan;

osc.connect(oscGain).connect(panner).c
onnect(masterGain);
  osc.start(t0); osc.stop(t0 + duration);

  // noise burst
  const nb =
makeNoiseBuffer(duration×0.9);
  playBuffer(nb, when, 0.5*vol, pan, 1);
}

// procedural roar (lower, aggressive)
function playRoar({when=0, pan=0,
duration=1.0, vol=0.9}={}) {
  const t0 = audioCtx.currentTime +
when;
  // low oscillator
  const osc =
audioCtx×createOscillator();
  const gain = audioCtx×createGain();
  osc×type = 'triangle';
  osc.frequency.setValueAtTime(60, t0);

osc.frequency.linearRampToValueAtTime
(150, t0 + duration*0.4);
  gain.gain.setValueAtTime(0.0001, t0);

gain.gain.linearRampToValueAtTime(0.35
*vol, t0 + duration*0.2);

gain.gain.exponentialRampToValueAt-
Time(0.0001, t0 + duration);
```

```
  const biquad =
audioCtx×createBiquadFilter();
  biquad×type = 'lowpass';
  biquad.frequency.setValueAtTime(900,
t0);

biquad.frequency.exponentialRampTo-
ValueAtTime(300, t0 + duration);
  const panner =
audioCtx×createStereoPanner();
  panner×pan×value = pan;

osc.connect(gain).connect(biquad).con-
nect(panner).connect(masterGain);
  osc.start(t0); osc.stop(t0 + duration);

  // layered noise for aggression
  const nb =
makeNoiseBuffer(duration×0.9);
  playBuffer(nb, when, 0.25*vol, pan, 1);
}

// metallic clank chain sequence (Bound)
function playChainsClank({when=0,
pan=0, vol=0.8}={}) {
  const t0 = audioCtx.currentTime +
when;
  // a few pulses of filtered noise + ring
  for (let i=0;i<3;i++){
    const delay = i*0.25;
    // short noise hit
    const nb = makeNoiseBuffer(0.18);
    // process: bandpass to sound metallic
    const src =
audioCtx×createBufferSource();
    src×buffer = nb;
    const bp =
audioCtx×createBiquadFilter();
    bp×type = 'bandpass';
    bp.frequency.value = 1000 +
Math.random()*800;
    bp×Q×value = 6;
```

```
    const g = audioCtx.createGain();
    g×gain×value = 0.35*vol *
Math.pow(0.8, i);
    const p =
audioCtx×createStereoPanner();
    p×pan×value = pan +
(Math.random()*0.2-0.1);

src.connect(bp).connect(g).connect(p).c
onnect(masterGain);
    src.start(t0 + delay);
  }
  // small bell/ring
  const bell =
audioCtx×createOscillator();
  const bellGain = audioCtx.createGain();
  bell×type = 'sine';
  bell.frequency.setValueAtTime(700, t0
+ 0.35);
  bellGain.gain.setValueAtTime(0.0001,
t0 + 0.35);

bellGain.gain.linearRampToValueAtTime(
0.2*vol, t0 + 0.36);

bellGain.gain.exponentialRampToValue-
AtTime(0.0001, t0 + 1.0);

bell.connect(bellGain).connect(master-
Gain);
  bell.start(t0 + 0.35); bell.stop(t0 + 1.0);
}

// helper to kick audio when approach
starts (subtle rumble)
function playApproachTone({when=0,
pan=0, duration=6, vol=0.06}={}) {
  const t0 = audioCtx.currentTime +
when;
  const osc =
audioCtx×createOscillator();
  const g = audioCtx.createGain();
```

```
  osc×type='sine';
  osc.frequency.setValueAtTime(40, t0);

osc.frequency.linearRampToValueAtTime
(120, t0 + duration);
  g.gain.setValueAtTime(0.001*vol, t0);

g.gain.linearRampToValueAtTime(0.12*vo
l, t0 + duration);
  const bp =
audioCtx×createBiquadFilter();
  bp×type='lowpass';
bp.frequency.setValueAtTime(400, t0);
  const p =
audioCtx×createStereoPanner();
  p×pan×value = pan;

osc.connect(g).connect(bp).connect(p).
connect(masterGain);
  osc.start(t0); osc.stop(t0 + duration +
0.1);
}

// ------------------------------
// Game state variables
// ------------------------------
let doorNum = 1;
let gameOver = false;
let inMenu = true;
let gameResult = "";
let lockers = [];
let overseer = null; // {x,y,r}
let lightsBroken = false;
let darkRoom = false;
let hidden = false;

let player = { x: 400, y: 300, w: 12, h: 12,
color: "#80ff80" };
let flashlight = true;
let flashlightBattery = 100;

let nextSpawnTimeout = null;
```

```
let spawnTimer = 0;
let entityActive = false; // whether Rush/
Blitz/Bound is currently passing
let currentVariant = null; //
"rush"|"blitz"|"bound"
let approaching = false; // flicker/warning
phase
let approachTimeout = null;
let passTimeout = null;

// joystick + touch UI
let keys = {};
let joystick = { active:false, startX:0,
startY:0, x:0, y:0, dx:0, dy:0 };
const buttons = {
  hide: { x: W-70, y: H-100, r: 34,
pressed:false },
  flash: { x: W-170, y: H-100, r: 34,
pressed:false },
  restart: { x: W-120, y: H/2, w: 100, h:
50 }
};

// input handlers
document.addEventListener('keydown',
(e)=>{
  const k = e.key.toLowerCase();
  keys[k] = true;
  if (inMenu && e×key === 'Enter')
startGame();
  if (gameOver && k === 'r') startGame();
});
document.addEventListener('keyup',
e=>{ keys[e×key×toLowerCase()] =
false; });

canvas.addEventListener('touchstart',
touchHandler, {passive:false});
canvas.addEventListener('touchmove',
touchHandler, {passive:false});
canvas.addEventListener('touchend',
touchEndHandler, {passive:false});
```

```
canvas.addEventListener('mousedown',
mouseDown, false);
canvas.addEventListener('mousemove',
mouseMove, false);
canvas.addEventListener('mouseup',
mouseUp, false);

// tap-to-start (also resumes audio)
canvas.addEventListener('touchstart',
(e)=>{
  if (inMenu) {
    // must resume audio on user gesture
    ensureAudio();
    if (audioCtx.state === 'suspended')
audioCtx.resume();
    startGame();
  }
}, {passive:false});

// mouse fallback to allow desktop
clicking on start
canvas.addEventListener('click', (e)=>{
  if (inMenu) {
    ensureAudio();
    if (audioCtx && audioCtx.state ===
'suspended') audioCtx.resume();
    startGame();
  }
});

function touchHandler(ev) {
  ev.preventDefault();
  if (!audioCtx) { ensureAudio(); if
(audioCtx.state === 'suspended')
audioCtx.resume(); }
  const rect =
canvas×getBoundingClientRect();
  if (inMenu) { startGame(); return; }
  for (let t=0; t<ev.touches.length; t++){
    const touch = ev×touches[t];
    const x = (touch×clientX - rect×left) /
rect×width × W;
```

```
    const y = (touch×clientY - rect×top) /
rect×height × H;
    if (x < W/2) {
      joystick×active = true;
      joystick×startX = x; joystick×startY =
y;
      joystick×x = x; joystick×y = y;
      joystick×dx = 0; joystick×dy = 0;
    } else {
      // buttons area
      const hb = buttons×hide, fb =
buttons×flash, rb = buttons.restart;
      if (!gameOver) {
        if (distance(x,y,hb.x,hb.y) < hb.r)
hb×pressed = true;
        if (distance(x,y,fb.x,fb.y) < fb.r)
fb×pressed = true;
      } else {
        if (x > rb.x && x < rb.x + rb.w && y >
rb.y && y < rb.y + rb.h) {
          startGame();
        }
      }
    }
  }
}

function touchEndHandler(ev) {
  ev.preventDefault();
  if (ev.touches.length === 0) {
    joystick×active = false;
    joystick×dx = joystick×dy = 0;
    buttons×hide×pressed = false;
    buttons×flash×pressed = false;
  }
}

function mouseDown(e){
  const rect =
canvas×getBoundingClientRect();
  const x = (e×clientX - rect×left) /
rect×width × W;
```

```
  const y = (e×clientY - rect×top) /
rect×height × H;
  if (inMenu) { startGame(); return; }
  if (x < W/2) {
    joystick×active = true; joystick×startX
= x; joystick×startY = y; joystick×x = x;
joystick×y = y;
  } else {
    const hb = buttons×hide, fb =
buttons×flash, rb = buttons.restart;
    if (!gameOver) {
      if (distance(x,y,hb.x,hb.y) < hb.r)
hb×pressed = true;
      if (distance(x,y,fb.x,fb.y) < fb.r)
fb×pressed = true;
    } else {
      if (x > rb.x && x < rb.x + rb.w && y >
rb.y && y < rb.y + rb.h) startGame();
    }
  }
}

function mouseMove(e){
  if (!joystick.active) return;
  const rect =
canvas×getBoundingClientRect();
  const x = (e×clientX - rect×left) /
rect×width × W;
  const y = (e×clientY - rect×top) /
rect×height × H;
  joystick×dx = x - joystick.startX;
  joystick×dy = y - joystick.startY;
}
function mouseUp(e)
{ joystick×active=false; joystick×dx=0;
joystick×dy=0;
buttons×hide×pressed=false;
buttons×flash×pressed=false; }

// helpers
function distance(x1,y1,x2,y2){ return
Math.hypot(x1-x2,y1-y2); }
```

```
function randRange(a,b){ return a +
Math.random()*(b-a); }
function randInt(a,b){ return
Math.floor(randRange(a,b)); }
function chance(p){ return
Math.random() < p; }

// ------------------------------
// Room generation & reset
// ------------------------------
function generateRoom() {
  lockers×length = 0;
  overseer = null;
  lightsBroken = false;
  hidden = false;
  if (doorNum <= 50) darkRoom =
chance(0.10);
  else if (doorNum <= 90) darkRoom =
chance(0.30);
  else darkRoom = true;
  const numLockers = randInt(1,3);
  for (let i=0;i<numLockers;i++){
    lockers.push({ x: randInt(100, W-60),
y: randInt(100, H-60), w:30, h:20 });
  }
  if (darkRoom && chance(0.5)) {
    overseer = { x: randInt(100, W-100), y:
randInt(100, H-100), r: 25 };
  }
  // small battery top-up between rooms
  flashlightBattery = Math×min(100,
flashlightBattery + 20);
  // schedule spawn timer reset
  resetSpawnTimer();
}

// ------------------------------
// Spawn system (Rush/Blitz/Bound)
// ------------------------------
function resetSpawnTimer() {
  // Clear previous timers
  if (nextSpawnTimeout)
```

```
{ clearTimeout(nextSpawnTimeout);
nextSpawnTimeout = null; }
  spawnTimer = 0;
  entityActive = false;
  approaching = false;
}

// choose variant when spawn cycle
triggers:
// Blitz 5% chance, Bound 10% chance,
else Rush
function chooseVariant() {
  // Bound replaces Rush with 10%
  if (chance(0.10)) return 'bound';
  if (chance(0.05)) return 'blitz';
  return 'rush';
}

function scheduleNextSpawn() {
  // Called after previous pass finishes or
on room enter
  // spawn interval random 1-3 minutes
(ms)
  const delay =
Math.floor(randRange(60000, 180000));
  // but if player is in dark room and not in
last ten, skip spawn by rescheduling
  // We will set a timeout that checks the
dark room rule upon execution
  nextSpawnTimeout = setTimeout(() => {
    // check rule
    if (darkRoom && doorNum <= 90) {
      // skip and reschedule
      scheduleNextSpawn();
      return;
    }
    startApproach();
  }, delay);
}

function startApproach() {
  // Start the flicker & approach phase,
```

```
choose variant
  approaching = true;
  currentVariant = chooseVariant(); //
'rush' | 'blitz' | 'bound'
  // flicker count: blitz 2 quick flickers,
others 1
  const flicks = (currentVariant ===
'blitz') ? 2 : 1;
  // flicker visual & short buzz
  flickerLights(flicks, 300);
  // play flicker sound
  if (audioCtx) {
    // light flicker buzz
    playWhoosh({when:0, pan:0,
duration:0.18, vol:0.25});
  }
  // approach delay depends on variant
  let approachDelay = 0;
  if (currentVariant === 'blitz')
approachDelay =
Math.floor(randRange(5000,7000));
  else if (currentVariant === 'rush')
approachDelay =
Math.floor(randRange(10000,15000));
  else if (currentVariant === 'bound')
approachDelay =
Math×floor(randRange(14000,17000));
  // play approach ambient if long enough
  if (audioCtx) {
    // stereo pan left-right random small
    playApproachTone({when:0, pan:
(Math.random()*2-1)*0.3, duration:
Math.max(approachDelay/1000, 3),
vol:0.08});
  }
  // chain clanks for bound - play closer
to the end of approach
  if (currentVariant === 'bound' &&
audioCtx) {
    // schedule clank ~0.9s before pass
(but after half of approach)
    const clankWhen = Math×max(0,
```

```
(approachDelay - 900) / 1000);
    setTimeout(()=> {
      // clinking chains
      playChainsClank({when:0, pan:
(Math.random()*2-1)*0.5, vol:1.0});
    }, Math.max(0, approachDelay - 900));
  }
  // schedule actual pass
  approachTimeout = setTimeout(()=> {
    approaching = false;
    startPass();
  }, approachDelay);
}

function startPass() {
  entityActive = true;
  // play passing sound and visual
movement
  const variant = currentVariant; //
'rush'/'blitz'/'bound'
  const passDuration = (variant ===
'blitz') ? 500 : (variant === 'bound' ?
1500 : 1000); // ms
  const color = (variant === 'blitz') ?
'rgba(255,20,20,0.9)' : (variant ===
'rush' ? 'rgba(255,205,40,0.92)' :
'rgba(160,160,160,0.85)');
  // play roar/whoosh aligned with pass
  if (audioCtx) {
    if (variant === 'blitz')
playRoar({when:0, pan:0, duration:0.6,
vol:1.2});
    else if (variant === 'rush')
playWhoosh({when:0, pan:0,
duration:0.9, vol:1.0});
    else if (variant === 'bound')
{ playRoar({when:0.2, pan:0,
duration:1.5, vol:1.0}); }
  }
  // animate pass across screen (we will
handle drawing in render loop by
marking pass start)
```

```
  passStartTime = performance.now();
  passInfo = { variant, color, duration:
passDuration, start: passStartTime };
  // when pass ends:
  passTimeout = setTimeout(()=> {
    // If player not hidden -> lose
    if (!hidden) {
      endGame(`$
{variant.charAt(0).toUpperCase() +
variant.slice(1)} passed and caught you!
`);
    } else {
      // survived; lights break permanently
      lightsBroken = true;
      // schedule next spawn if not end
game
      if (!gameOver)
scheduleNextSpawn();
    }
    entityActive = false;
    currentVariant = null;
  }, passDuration);
}

// flicker helper
function flickerLights(times=1,
speed=300) {
  let c = 0;
  let state = lightsBroken;
  const doF = ()=>{
    state = !state;
    lightsBroken = state;
    c++;
    if (c < times*2) setTimeout(doF,
speed);
    else {
      // restore after short pause (unless it
should be broken later)
      lightsBroken = false;
    }
  };
  doF();
```

```
}

// variables for drawing the passing
entity
let passStartTime = 0;
let passInfo = null;

// ------------------------------
// Game lifecycle: start/restart
// ------------------------------
function startGame(){
  // ensure audio (first user gesture)
  ensureAudio();
  if (audioCtx.state === 'suspended')
audioCtx.resume();
  // reset state
  inMenu = false;
  gameOver = false;
  gameResult = '';
  doorNum = 1;
  playerx = 20; playery = H/2;
  flashlight = true;
  flashlightBattery = 100;
  generateRoom();
  scheduleNextSpawn();
}

// end game
function endGame(msg) {
  gameOver = true;
  gameResult = msg + ` You reached
door ${doorNum}.`;
  // clear timers
  if (nextSpawnTimeout)
{ clearTimeout(nextSpawnTimeout);
nextSpawnTimeout = null; }
  if (approachTimeout)
{ clearTimeout(approachTimeout);
approachTimeout = null; }
  if (passTimeout)
{ clearTimeout(passTimeout);
passTimeout = null; }
```

```
}

// ------------------------------
// Update loop
// ------------------------------
let lastTS = 0;
function update(ts) {
  if (!lastTS) lastTS = ts;
  const dt = ts - lastTS;
  lastTS = ts;
  if (!inMenu && !gameOver && !
entityActive) {
    // movement keyboard or joystick
    let mvx = 0, mvy = 0;
    if (!hidden) {
      // keyboard
      if (keys['w']) mvy -= PLAYER_SPEED;
      if (keys['s']) mvy += PLAYER_SPEED;
      if (keys['a']) mvx -= PLAYER_SPEED;
      if (keys['d']) mvx += PLAYER_SPEED;
      // joystick
      if (joystick.active) {
        const dx = joystick×dx || (joystick.x
- joystick.startX);
        const dy = joystick×dy || (joystick.y
- joystick.startY);
        // map to movement with deadzone
        if (Math.abs(dx) > JOY_DEADZONE)
mvx += (dx > 0 ? 1 : -1)*PLAYER_SPEED;
        if (Math.abs(dy) > JOY_DEADZONE)
mvy += (dy > 0 ? 1 : -1)*PLAYER_SPEED;
      }
    }
    player×x = Math.max(0, Math.min(W -
player.w, player.x + mvx));
    player×y = Math.max(0, Math.min(H -
player.h, player.y + mvy));
  }

  // flashlight toggle via keyboard or
button
  if ((keys['f'] && !inMenu) ||
```

```
buttons.flash.pressed) {
    // only toggle on press; to avoid
repeated toggles, clear keys/button
immediately
    flashlight = !flashlight;
    keys['f'] = false;
    buttons×flash×pressed = false;
  }
  // hide/unhide
  if ((keys['e'] && !inMenu) ||
buttons.hide.pressed) {
    // toggle hide (only when near a
locker)
    if (hidden) hidden = false;
    else {
      for (let l of lockers) {
        if (distance(player.x + player.w/2,
player.y + player.h/2, l.x + l.w/2, l.y + l.h/
2) < 40) {
          hidden = true; break;
        }
      }
    }
    keys['e'] = false;
    buttons×hide×pressed = false;
  }

  // battery drain
  if (flashlight && flashlightBattery > 0
&& !hidden) {
    flashlightBattery -= FLASH_DRAIN *
(dt/16.67); // scale with frame time
    if (flashlightBattery <= 0)
{ flashlightBattery = 0; flashlight =
false; }
  }

  // Overseer detection
  if (overseer && flashlight &&
flashlightBattery > 0 && !hidden) {
    const d = distance(player.x + player.w/
2, player.y + player.h/2, overseer.x,
```

```
overseer.y);
    if (d < overseer.r + 120) {
      endGame('The Overseer reacted to
your light!');
    }
  }

  // movement reaching door (right edge)
  if (!inMenu && !gameOver && player.x >
W - 24) {
    doorNum++;
    if (doorNum > MAX_DOORS) {
      endGame('You escaped!');
    } else {
      // next room
      player×x = 20; player×y = H/2;
      generateRoom();
      // cancel active entity & timers in
previous room
      if (nextSpawnTimeout)
{ clearTimeout(nextSpawnTimeout);
nextSpawnTimeout=null; }
      if (approachTimeout)
{ clearTimeout(approachTimeout);
approachTimeout=null; }
      if (passTimeout)
{ clearTimeout(passTimeout);
passTimeout=null; }
      entityActive = false;
      currentVariant = null;
      aproachStarted = false;
      // schedule next spawn
      scheduleNextSpawn();
    }
  }
}

// ------------------------------
// Drawing
// ------------------------------
function render(ts) {
  // clear
```

```
  ctx.clearRect(0,0,W,H);

  if (inMenu) {
    // Title screen
    ctx×fillStyle = "#111";
    ctx.fillRect(0,0,W,H);
    ctx×fillStyle = "#fff";
    ctx×textAlign = "center";
    ctx×textBaseline = "middle";
    ctx×font = "56px monospace";
    ctx.fillText("DOORWAYS", W/2, H/2 -
40);
    ctx×font = "20px monospace";
    ctx.fillText("TAP ANYWHERE TO
START", W/2, H/2 + 30);
    ctx×font = "14px monospace";
    ctx.fillText("Mobile: left joystick • right
buttons", W/2, H/2 + 70);
    return;
  }

  // background (darkness depending)
  const darkBg = darkRoom && !
flashlight && !lightsBroken;
  ctx×fillStyle = darkBg ? "#040404" :
"#333";
  ctx.fillRect(0,0,W,H);

  // if flickering approaching (visual
effect)
  if (approaching) {
    // subtle global flicker effect by tint
overlay
    const flick = Math×floor(ts/120) % 2;
    if (flick === 0) {
      ctx×fillStyle =
'rgba(255,255,255,0.02)';
ctx.fillRect(0,0,W,H);
    } else {
      ctx×fillStyle = 'rgba(0,0,0,0.03)';
ctx.fillRect(0,0,W,H);
    }
```

```
  }

  // lockers (pixel-art rectangles)
  ctx×fillStyle = "#6b4f2f";
  for (let l of lockers) {
    ctx.fillRect(Math.round(l.x),
Math.round(l.y), l.w, l.h);
    // locker door line
    ctx×strokeStyle = "#403020";
ctx×lineWidth = 1;
    ctx.beginPath(); ctx.moveTo(l.x + 4, l.y
+ 2); ctx.lineTo(l.x + l.w - 4, l.y + 2);
ctx.stroke();
  }

  // overseer glow if present (draw before
lighting so it softly shows in dark)
  if (overseer) {
    const grad =
ctx.createRadialGradient(overseer.x,
overseer.y, 0, overseer.x, overseer.y,
overseer.r*3);
    grad.addColorStop(0,
"rgba(0,160,255,0.9)");
    grad.addColorStop(1,
"rgba(0,160,255,0)");
    ctx×fillStyle = grad;
    ctx.beginPath(); ctx.arc(overseer.x,
overseer.y, overseer.r*3, 0, Math.PI*2);
ctx.fill();
    // small orb center
    ctx×fillStyle = "rgba(0,200,255,0.9)";
ctx.beginPath(); ctx.arc(overseer.x,
overseer.y, overseer.r, 0, Math.PI*2);
ctx.fill();
  }

  // player
  if (!hidden) {
    ctx×fillStyle = player.color;
    ctx.fillRect(Math.round(player.x),
Math.round(player.y), player.w, player.h);
```

```
  } else {
    // show subtle hidden indicator
    ctx.fillStyle = "#888";
ctx.fillRect(Math.round(player.x),
Math.round(player.y), player.w, player.h);
  }

  // flashlight lighting (destination-out
trick)
  if ((darkRoom || lightsBroken) &&
flashlight && flashlightBattery > 0 && !
hidden) {
    // dark overlay
    ctx.fillStyle = "rgba(0,0,0,0.92)";
    ctx.fillRect(0,0,W,H);

    // radial hole
    ctx.save();
    ctx.globalCompositeOperation =
'destination-out';
    const radius = 100 +
(flashlightBattery/100)*120;
    const grad =
ctx.createRadialGradient(player.x +
player.w/2, player.y + player.h/2, 20,
player.x + player.w/2, player.y + player.h/
2, radius);
    grad.addColorStop(0,
'rgba(255,255,220,0.95)');
    grad.addColorStop(1, 'rgba(0,0,0,0)');
    ctx.fillStyle = grad;
    ctx.beginPath();
    ctx.arc(player.x + player.w/2, player.y +
player.h/2, radius, 0, Math.PI*2);
    ctx.fill();
    ctx.restore();
  } else if ((darkRoom || lightsBroken)) {
    // fully dark overlay
    ctx.fillStyle = "rgba(0,0,0,0.9)";
ctx.fillRect(0,0,W,H);
  }
```

```
  // if passing entity active, draw it as a
sweeping colored bar
  if (entityActive && passInfo) {
    const now = performance×now();
    const elapsed = now - passInfo.start;
    const pct = Math×min(1, elapsed /
passInfo.duration);
    // sweep left -> right across screen
    const sweepWidth = 200;
    const x = -sweepWidth + pct × (W +
sweepWidth×2);
    ctx×fillStyle = passInfo.color;
    // add blur-like gradient
    ctx.fillRect(x, -20, sweepWidth,
H+40);
  }

  // HUD
  ctx×fillStyle = "#fff";
  ctx×font = "14px monospace";
  ctx×textAlign = "left";
  ctx.fillText("Door: " + doorNum, 16, 20);
  ctx.fillText("Battery: " + Math.max(0,
Math.floor(flashlightBattery)) + "%", 16,
40);
  if (hidden) ctx.fillText("[HIDDEN]", 16,
60);
  if (darkRoom) ctx.fillText("(Dark
Room)", 16, 80);

  // draw touch UI (joystick indicator)
  if (joystick.active) {
    const cx = joystick×startX, cy =
joystick×startY;
    const dx = joystick×dx || 0, dy =
joystick×dy || 0;
    // background circle
    ctx×strokeStyle = "#fff";
ctx×lineWidth = 2;
    ctx.beginPath(); ctx.arc(cx, cy, 42, 0,
Math.PI*2); ctx.stroke();
    // thumb
```

```
    const tx = cx + Math×max(-32,
Math.min(32, dx));
    const ty = cy + Math×max(-32,
Math.min(32, dy));
    ctx×fillStyle = "#ccc"; ctx.beginPath();
ctx.arc(tx, ty, 18, 0, Math.PI*2); ctx.fill();
  } else {
    // draw a faint joystick hint on left
bottom
    ctx×strokeStyle =
"rgba(255,255,255,0.12)"; ctx×lineWidth
= 1;
    ctx.beginPath(); ctx.arc(80, H-110, 36,
0, Math.PI*2); ctx.stroke();
  }

  // action buttons
  drawButton(buttons.hide, "E");
  drawButton(buttons.flash, "F");

  // game over overlay & restart button
  if (gameOver) {
    ctx×fillStyle = "rgba(0,0,0,0.6)";
ctx.fillRect(0,0,W,H);
    ctx×fillStyle = "#fff"; ctx×textAlign =
"center"; ctx×font = "36px monospace";
    ctx.fillText(gameResult, W/2, H/2 -
40);
    ctx×font = "16px monospace";
    ctx.fillText("Tap RESTART or press R",
W/2, H/2 + 6);
    // restart button
    const rb = buttons.restart;
    ctx×fillStyle = "#444";
ctx.fillRect(rb.x, rb.y, rb.w, rb.h);
    ctx×strokeStyle = "#fff";
ctx.strokeRect(rb.x, rb.y, rb.w, rb.h);
    ctx×fillStyle = "#fff"; ctx×font = "16px
monospace"; ctx.fillText("RESTART",
rb.x + rb.w/2, rb.y + rb.h/2 + 2);
  }
}
```

```
function drawButton(b, label) {
  ctx.beginPath();
  ctx.arc(b.x, b.y, b.r, 0, Math.PI*2);
  ctx×fillStyle = (b.pressed) ?
"rgba(255,255,255,0.25)" :
"rgba(255,255,255,0.12)";
  ctx.fill();
  ctx×strokeStyle = "#fff"; ctx×lineWidth
= 1; ctx.stroke();
  ctx×fillStyle = "#fff"; ctx×font = "16px
monospace"; ctx×textAlign = "center";
ctx×textBaseline = "middle";
  ctx.fillText(label, b.x, b.y+1);
}

// ------------------------------
// Main loop
// ------------------------------
function mainLoop(ts) {
  update(ts);
  render(ts);
  requestAnimationFrame(mainLoop);
}
requestAnimationFrame(mainLoop);

// start scheduling initial spawn only after
entering first room
function scheduleNextSpawn() {
  // clear any existing
  if (nextSpawnTimeout)
clearTimeout(nextSpawnTimeout);
  // pick 1-3 minutes
  const delay =
Math.floor(randRange(60000, 180000));
  nextSpawnTimeout = setTimeout(() => {
    // rule: cannot spawn in dark rooms
unless last 10 doors
    if (darkRoom && doorNum <= 90) {
      scheduleNextSpawn(); return;
    }
    startApproach();
```

```
  }, delay);
}

// ------------------------------
// simple helpers and small polish
// ------------------------------

function distance(a,b,c,d){ return
Math.hypot(a-c,b-d); }

// make sure UI positions adjust with
canvas size (if any)
function layoutButtons() {
  buttons×hide×x = W - 70;
buttons×hide×y = H - 100;
  buttons×flash×x = W - 170;
buttons×flash×y = H - 100;
  buttons×restart×x = W - 120;
buttons×restart×y = H/2 - 25;
}
layoutButtons();

// initial menu draw
render();

// Ensure we cancel timers on page
unload (cleanup)
window.addEventListener('beforeunload'
, ()=> {
  if (nextSpawnTimeout)
clearTimeout(nextSpawnTimeout);
  if (approachTimeout)
clearTimeout(approachTimeout);
  if (passTimeout)
clearTimeout(passTimeout);
});
</script>
</body>
</html>
```