```
// --- UI helpers ---
const $ = (id) => document.getElementById(id);

const file = $("file");
const btnProcess = $("btnProcess");
const btnExport = $("btnExport");

const srcCanvas = $("src");
const outCanvas = $("out");
const srcCtx = srcCanvas.getContext("2d");
const outCtx = outCanvas.getContext("2d");

const edge = $("edge"), blur = $("blur"), thick = $("thick"), shade = $("shade"), hatch =
$("hatch"), angle = $("angle");
const vEdge = $("vEdge"), vBlur = $("vBlur"), vThick = $("vThick"), vShade = $("vShade"),
vHatch = $("vHatch"), vAngle = $("vAngle");

let img = new Image();
let cvReady = false;

function syncLabels() {
  vEdge.textContent = edge.value;
  vBlur.textContent = blur.value;
  vThick.textContent = thick.value;
  vShade.textContent = shade.value;
  vHatch.textContent = hatch.value;
  vAngle.textContent = angle.value;
}
[edge, blur, thick, shade, hatch, angle].forEach(r => r.addEventListener("input", () => {
  syncLabels();
  if (img.src) process(); // live update
}));
syncLabels();

// Wait for OpenCV.js to be ready
const waitCV = setInterval(() => {
  if (typeof cv !== "undefined" && cv.Mat) {
    clearInterval(waitCV);
    cvReady = true;
    if (img.src) btnProcess.disabled = false;
  }
}, 50);

file.addEventListener("change", (e) => {
  const f = e.target.files?.[0];
  if (!f) return;
  const url = URL.createObjectURL(f);
  img = new Image();
```

```
  img.onload = () => {
    // Fit canvases to image (limit max side for speed)
    const maxSide = 1800;
    let w = img.naturalWidth, h = img.naturalHeight;
    const scale = Math.min(1, maxSide / Math.max(w, h));
    w = Math.round(w * scale); h = Math.round(h * scale);

    srcCanvas.width = w; srcCanvas.height = h;
    outCanvas.width = w; outCanvas.height = h;

    srcCtx.clearRect(0,0,w,h);
    srcCtx.drawImage(img, 0, 0, w, h);

    btnProcess.disabled = !cvReady;
    btnExport.disabled = false;

    process();
  };
  img.src = url;
});

btnProcess.addEventListener("click", process);

btnExport.addEventListener("click", () => {
  const a = document.createElement("a");
  a.download = "stencil.png";
  a.href = outCanvas.toDataURL("image/png");
  a.click();
});

// --- Core processing ---
function process() {
  if (!cvReady || !img.src) return;

  const w = srcCanvas.width, h = srcCanvas.height;

  // Read from src canvas into OpenCV
  const srcMat = cv.imread(srcCanvas);
  const gray = new cv.Mat();
  cv.cvtColor(srcMat, gray, cv.COLOR_RGBA2GRAY, 0);

  // Blur for cleanup
  const k = parseInt(blur.value, 10);
  if (k > 0) {
    const ksize = new cv.Size(k, k);
    cv.GaussianBlur(gray, gray, ksize, 0, 0, cv.BORDER_DEFAULT);
  }
```

```
// Edge detection (Canny)
const edges = new cv.Mat();
const t1 = parseInt(edge.value, 10);
const t2 = Math.min(255, t1 * 2);
cv.Canny(gray, edges, t1, t2);

// Thicken lines (dilate)
const t = parseInt(thick.value, 10);
if (t > 1) {
  const kernel = cv.getStructuringElement(cv.MORPH_ELLIPSE, new cv.Size(t, t));
  cv.dilate(edges, edges, kernel);
  kernel.delete();
}

// Convert edges to black lines on white
// We'll draw on outCanvas combining: lines + hatch
outCtx.setTransform(1,0,0,1,0,0);
outCtx.clearRect(0,0,w,h);
outCtx.fillStyle = "#fff";
outCtx.fillRect(0,0,w,h);

// Draw edges
// Create an ImageData from edges mat: edges is 0..255 (white edges on black)
// We want black lines: invert when drawing.
const edgeRGBA = new cv.Mat();
cv.cvtColor(edges, edgeRGBA, cv.COLOR_GRAY2RGBA);
// Put to a temp canvas via cv.imshow, then invert via composite
const tmp = document.createElement("canvas");
tmp.width = w; tmp.height = h;
cv.imshow(tmp, edgeRGBA);

// Invert edge colors to get black lines on transparent background
const tctx = tmp.getContext("2d");
const id = tctx.getImageData(0,0,w,h);
const d = id.data;
for (let i=0;i<d.length;i+=4) {
  const v = d[i]; // 0..255
  // edges are white where line: v=255
  // We'll set alpha to v, color black.
  d[i] = 0; d[i+1] = 0; d[i+2] = 0;
  d[i+3] = v; // alpha
}
tctx.putImageData(id,0,0);
outCtx.drawImage(tmp,0,0);

// Hatch (shadows) based on original luminance
drawHatchingFromGray(gray, w, h);
```

```
  // Cleanup mats
  srcMat.delete(); gray.delete(); edges.delete(); edgeRGBA.delete();
}

function drawHatchingFromGray(grayMat, w, h) {
  const intensity = parseInt(shade.value, 10) / 100; // 0..1
  if (intensity <= 0) return;

  // Get grayscale pixels
  const gray = new Uint8Array(grayMat.data);

  // Create hatch layer
  const hatchCanvas = document.createElement("canvas");
  hatchCanvas.width = w; hatchCanvas.height = h;
  const hctx = hatchCanvas.getContext("2d");
  hctx.clearRect(0,0,w,h);

  // Hatch spacing: lower => denser
  const spacing = parseInt(hatch.value, 10); // 4..24
  const ang = parseInt(angle.value, 10) * Math.PI / 180;

  // We draw many diagonal lines across the canvas, then mask by darkness.
  // Approach: per-pixel mask is heavy; instead we draw hatch, then apply alpha mask via
ImageData.
  hctx.save();
  hctx.translate(w/2, h/2);
  hctx.rotate(ang);
  hctx.translate(-w/2, -h/2);

  hctx.strokeStyle = "rgba(0,0,0,1)";
  hctx.lineWidth = 1;

  for (let y = -w; y < h + w; y += spacing) {
    hctx.beginPath();
    hctx.moveTo(0, y);
    hctx.lineTo(w, y + w);
    hctx.stroke();
  }
  hctx.restore();

  // Convert hatch to ImageData and set alpha based on darkness (from gray)
  const hid = hctx.getImageData(0,0,w,h);
  const hd = hid.data;

  for (let i=0, p=0; i<hd.length; i+=4, p++) {
    // If not a hatch pixel, skip
    if (hd[i+3] === 0) continue;
```

```
    const g = gray[p]; // 0..255
    // darkness: 0 (white) .. 1 (black)
    const dark = (255 - g) / 255;

    // Boost control: only apply where dark is significant
    const a = Math.max(0, (dark - 0.15) / 0.85); // clamp-ish
    const alpha = Math.round(255 * a * intensity);

    hd[i] = 0; hd[i+1] = 0; hd[i+2] = 0;
    hd[i+3] = alpha;
  }

  hctx.putImageData(hid, 0, 0);

  // Draw hatch layer over the lines
  outCtx.drawImage(hatchCanvas, 0, 0);
}
```