```javascript
(function () {
'use strict';

const SVGNS = "http://www.w3.org/2000/svg";

(function() {
    const values = [.5, .7, .1, .2, .8, .4, .9, .3, .6, .01, .99, .68, .38, .18, .77, .91, .53, .22, .47];
    function r() {
        r.seed++;
        return values[r.seed % values.length];
    }
    r.seed = 0;
//  Math.random = r;
})();

function clamp(x, min, max) {
    return Math.max(min, Math.min(max, x));
}

function clampColor(x) {
    return clamp(x, 0, 255);
}

function distanceToDifference(distance, pixels) {
    return Math.pow(distance*255, 2) * (3 * pixels);
}

function differenceToDistance(difference, pixels) {
    return Math.sqrt(difference / (3 * pixels))/255;
}

function difference(data, dataOther) {
    let sum = 0, diff;
    for (let i=0;i<data.data.length;i++) {
        if (i % 4 == 3) { continue; }
        diff = dataOther.data[i]-data.data[i];
        sum = sum + diff*diff;
    }

    return sum;
}

function computeColor(offset, imageData, alpha) {
    let color = [0, 0, 0];
    let {shape, current, target} = imageData;
    let shapeData = shape.data;
    let currentData = current.data;
    let targetData = target.data;

    let si, sx, sy, fi, fx, fy; /* shape-index, shape-x, shape-y, full-index, full-x, full-y */
    let sw = shape.width;
    let sh = shape.height;
    let fw = current.width;
    let fh = current.height;
    let count = 0;

    for (sy=0; sy<sh; sy++) {
        fy = sy + offset.top;
        if (fy < 0 || fy >= fh) { continue; } /* outside of the large canvas (vertically) */

        for (sx=0; sx<sw; sx++) {
            fx = offset.left + sx;
            if (fx < 0 || fx >= fw) { continue; } /* outside of the large canvas (horizontally) */

            si = 4*(sx + sy*sw); /* shape (local) index */
            if (shapeData[si+3] == 0) { continue; } /* only where drawn */

            fi = 4*(fx + fy*fw); /* full (global) index */
            color[0] += (targetData[fi] - currentData[fi]) / alpha + currentData[fi];
            color[1] += (targetData[fi+1] - currentData[fi+1]) / alpha + currentData[fi+1];
            color[2] += (targetData[fi+2] - currentData[fi+2]) / alpha + currentData[fi+2];

            count++;
        }
    }

    return color.map(x => ~~(x/count)).map(clampColor);
}

function computeDifferenceChange(offset, imageData, color) {
    let {shape, current, target} = imageData;
    let shapeData = shape.data;
    let currentData = current.data;
    let targetData = target.data;

    let a, b, d1r, d1g, d1b, d2r, d2b, d2g;
    let si, sx, sy, fi, fx, fy; /* shape-index, shape-x, shape-y, full-index */
    let sw = shape.width;
    let sh = shape.height;
    let fw = current.width;
    let fh = current.height;

    var sum = 0; /* V8 opt bailout with let */

    for (sy=0; sy<sh; sy++) {
        fy = sy + offset.top;
        if (fy < 0 || fy >= fh) { continue; } /* outside of the large canvas (vertically) */

        for (sx=0; sx<sw; sx++) {
            fx = offset.left + sx;
            if (fx < 0 || fx >= fw) { continue; } /* outside of the large canvas (horizontally) */

            si = 4*(sx + sy*sw); /* shape (local) index */
            a = shapeData[si+3];
            if (a == 0) { continue; } /* only where drawn */

            fi = 4*(fx + fy*fw); /* full (global) index */

            a = a/255;
            b = 1-a;
            d1r = targetData[fi]-currentData[fi];
            d1g = targetData[fi+1]-currentData[fi+1];
            d1b = targetData[fi+2]-currentData[fi+2];

            d2r = targetData[fi] - (color[0]*a + currentData[fi]*b);
            d2g = targetData[fi+1] - (color[1]*a + currentData[fi+1]*b);
            d2b = targetData[fi+2] - (color[2]*a + currentData[fi+2]*b);

            sum -= d1r*d1r + d1g*d1g + d1b*d1b;
            sum += d2r*d2r + d2g*d2g + d2b*d2b;
        }
    }

    return sum;
}

function computeColorAndDifferenceChange(offset, imageData, alpha) {
    let rgb = computeColor(offset, imageData, alpha);
    let differenceChange = computeDifferenceChange(offset, imageData, rgb);

    let color = `rgb(${rgb[0]}, ${rgb[1]}, ${rgb[2]})`;

    return {color, differenceChange};
}

function getScale(width, height, limit) {
    return Math.max(width / limit, height / limit, 1);
}

/* FIXME move to util */
function getFill(canvas) {
    let data = canvas.getImageData();
    let w = data.width;
    let h = data.height;
    let d = data.data;
    let rgb = [0, 0, 0];
    let count = 0;
    let i;

    for (let x=0; x<w; x++) {
        for (let y=0; y<h; y++) {
            if (x > 0 && y > 0 && x < w-1 && y < h-1) { continue; }
            count++;
            i = 4*(x + y*w);
            rgb[0] += d[i];
            rgb[1] += d[i+1];
            rgb[2] += d[i+2];
        }
    }

    rgb = rgb.map(x => ~~(x/count)).map(clampColor);
    return `rgb(${rgb[0]}, ${rgb[1]}, ${rgb[2]})`;
}

function svgRect(w, h) {
    let node = document.createElementNS(SVGNS, "rect");
    node.setAttribute("x", 0);
    node.setAttribute("y", 0);
    node.setAttribute("width", w);
    node.setAttribute("height", h);

    return node;
}

/* Canvas: a wrapper around a <canvas> element */
class Canvas {
    static empty(cfg, svg) {
        if (svg) {
            let node = document.createElementNS(SVGNS, "svg");
            node.setAttribute("viewBox", `0 0 ${cfg.width} ${cfg.height}`);
            node.setAttribute("clip-path", "url(#clip)");

            let defs = document.createElementNS(SVGNS, "defs");
            node.appendChild(defs);

            let cp = document.createElementNS(SVGNS, "clipPath");
            defs.appendChild(cp);
            cp.setAttribute("id", "clip");
            cp.setAttribute("clipPathUnits", "objectBoundingBox");

            let rect = svgRect(cfg.width, cfg.height);
            cp.appendChild(rect);

            rect = svgRect(cfg.width, cfg.height);
            rect.setAttribute("fill", cfg.fill);
            node.appendChild(rect);

            return node;
        } else {
            return new this(cfg.width, cfg.height).fill(cfg.fill);
        }
    }

    static original(url, cfg) {
        if (url == "test") {
            return Promise.resolve(this.test(cfg));
        }

        return new Promise(resolve => {
            let img = new Image();
            img.crossOrigin = true;
            img.src = url;
            img.onload = e => {
                let w = img.naturalWidth;
                let h = img.naturalHeight;

                let computeScale = getScale(w, h, cfg.computeSize);
                cfg.width = w / computeScale;
                cfg.height = h / computeScale;

                let viewScale = getScale(w, h, cfg.viewSize);

                cfg.scale = computeScale / viewScale;

                let canvas = this.empty(cfg);
                canvas.ctx.drawImage(img, 0, 0, cfg.width, cfg.height);

                if (cfg.fill == "auto") { cfg.fill = getFill(canvas); }

                resolve(canvas);
            };
            img.onerror = e => {
                console.error(e);
                alert("The image URL cannot be loaded. Does the server support CORS?");
            };
        });
    }

    static test(cfg) {
        cfg.width = cfg.computeSize;
        cfg.height = cfg.computeSize;
        cfg.scale = 1;
        let [w, h] = [cfg.width, cfg.height];

        let canvas = new this(w, h);
        canvas.fill("#fff");
        let ctx = canvas.ctx;

        ctx.fillStyle = "#f00";
        ctx.beginPath();
        ctx.arc(w/4, h/2, w/7, 0, 2*Math.PI, true);
        ctx.fill();

        ctx.fillStyle = "#0f0";
        ctx.beginPath();
        ctx.arc(w/2, h/2, w/7, 0, 2*Math.PI, true);
        ctx.fill();

        ctx.fillStyle = "#00f";
        ctx.beginPath();
        ctx.arc(w*3/4, h/2, w/7, 0, 2*Math.PI, true);
        ctx.fill();

        if (cfg.fill == "auto") { cfg.fill = getFill(canvas); }

        return canvas;
    }

    constructor(width, height) {
        this.node = document.createElement("canvas");
        this.node.width = width;
        this.node.height = height;
        this.ctx = this.node.getContext("2d");
        this._imageData = null;
    }

    clone() {
        let otherCanvas = new this.constructor(this.node.width, this.node.height);
        otherCanvas.ctx.drawImage(this.node, 0, 0);
        return otherCanvas;
    }

    fill(color) {
        this.ctx.fillStyle = color;
        this.ctx.fillRect(0, 0, this.node.width, this.node.height);
        return this;
    }

    getImageData() {
        if (!this._imageData) {
            this._imageData = this.ctx.getImageData(0, 0, this.node.width, this.node.height);
        }
        return this._imageData;
    }

    difference(otherCanvas) {
        let data = this.getImageData();
        let dataOther = otherCanvas.getImageData();

        return difference(data, dataOther);
    }

    distance(otherCanvas) {
        let difference$$1 = this.difference(otherCanvas);
        return differenceToDistance(difference$$1, this.node.width*this.node.height);
    }

    drawStep(step) {
        this.ctx.globalAlpha = step.alpha;
        this.ctx.fillStyle = step.color;
        step.shape.render(this.ctx);
        return this;
    }
}

/* Shape: a geometric primitive with a bbox */
class Shape {
    static randomPoint(width, height) {
        return [~~(Math.random()*width), ~~(Math.random()*height)];
    }

    static create(cfg) {
        let ctors = cfg.shapeTypes;
        let index = Math.floor(Math.random() * ctors.length);
        let ctor = ctors[index];
        return new ctor(cfg.width, cfg.height);
    }

    constructor(w, h) {
        this.bbox = {};
    }

    mutate(cfg) { return this; }

    toSVG() {}

    /* get a new smaller canvas with this shape */
    rasterize(alpha) {
        let canvas = new Canvas(this.bbox.width, this.bbox.height);
        let ctx = canvas.ctx;
        ctx.fillStyle = "#000";
        ctx.globalAlpha = alpha;
        ctx.translate(-this.bbox.left, -this.bbox.top);
        this.render(ctx);
        return canvas;
    }

    render(ctx) {}
}

class Polygon extends Shape {
    constructor(w, h, count) {
        super(w, h);

        this.points = this._createPoints(w, h, count);
        this.computeBbox();
    }

    render(ctx) {
        ctx.beginPath();
        this.points.forEach(([x, y], index) => {
            if (index) {
                ctx.lineTo(x, y);
            } else {
                ctx.moveTo(x, y);
            }
        });
        ctx.closePath();
        ctx.fill();
    }

    toSVG() {
        let path = document.createElementNS(SVGNS, "path");
        let d = this.points.map((point, index) => {
            let cmd = (index ? "L" : "M");
            return `${cmd}${point.join(",")}`;
        }).join("");
        path.setAttribute("d", `${d}Z`);
        return path;
    }

    mutate(cfg) {
        let clone = new this.constructor(0, 0);
        clone.points = this.points.map(point => point.slice());

        let index = Math.floor(Math.random() * this.points.length);
        let point = clone.points[index];

        let angle = Math.random() * 2 * Math.PI;
        let radius = Math.random() * 20;
        point[0] += ~~(radius * Math.cos(angle));
        point[1] += ~~(radius * Math.sin(angle));

        return clone.computeBbox();
    }

    computeBbox() {
        let min = [
            this.points.reduce((v, p) => Math.min(v, p[0]), Infinity),
            this.points.reduce((v, p) => Math.min(v, p[1]), Infinity)
        ];
        let max = [
            this.points.reduce((v, p) => Math.max(v, p[0]), -Infinity),
            this.points.reduce((v, p) => Math.max(v, p[1]), -Infinity)
        ];

        this.bbox = {
            left: min[0],
            top: min[1],
            width: (max[0]-min[0]) || 1, /* fallback for deformed shapes */
            height: (max[1]-min[1]) || 1
        };

        return this;
    }

    _createPoints(w, h, count) {
        let first = Shape.randomPoint(w, h);
        let points = [first];

        for (let i=1;i<count;i++) {
            let angle = Math.random() * 2 * Math.PI;
            let radius = Math.random() * 20;
            points.push([
                first[0] + ~~(radius * Math.cos(angle)),
                first[1] + ~~(radius * Math.sin(angle))
            ]);
        }
        return points;
    }
}

class Triangle extends Polygon {
    constructor(w, h) {
        super(w, h, 3);
    }
}

class Rectangle extends Polygon {
    constructor(w, h) {
        super(w, h, 4);
    }

    mutate(cfg) {
        let clone = new this.constructor(0, 0);
        clone.points = this.points.map(point => point.slice());

        let amount = ~~((Math.random()-0.5) * 20);

        switch (Math.floor(Math.random()*4)) {
            case 0: /* left */
                clone.points[0][0] += amount;
                clone.points[3][0] += amount;
            break;
            case 1: /* top */
                clone.points[0][1] += amount;
                clone.points[1][1] += amount;
            break;
            case 2: /* right */
                clone.points[1][0] += amount;
                clone.points[2][0] += amount;
            break;
            case 3: /* bottom */
                clone.points[2][1] += amount;
                clone.points[3][1] += amount;
            break;
        }

        return clone.computeBbox();
    }

    _createPoints(w, h, count) {
        let p1 = Shape.randomPoint(w, h);
        let p2 = Shape.randomPoint(w, h);

        let left = Math.min(p1[0], p2[0]);
        let right = Math.max(p1[0], p2[0]);
        let top = Math.min(p1[1], p2[1]);
        let bottom = Math.max(p1[1], p2[1]);

        return [
            [left, top],
            [right, top],
            [right, bottom],
            [left, bottom]
        ];
    }
}

class Ellipse extends Shape {
    constructor(w, h) {
        super(w, h);

        this.center = Shape.randomPoint(w, h);
        this.rx = 1 + ~~(Math.random() * 20);
        this.ry = 1 + ~~(Math.random() * 20);

        this.computeBbox();
    }

    render(ctx) {
        ctx.beginPath();
        ctx.ellipse(this.center[0], this.center[1], this.rx, this.ry, 0, 0, 2*Math.PI, false);
        ctx.fill();
    }

    toSVG() {
        let node = document.createElementNS(SVGNS, "ellipse");
        node.setAttribute("cx", this.center[0]);
        node.setAttribute("cy", this.center[1]);
        node.setAttribute("rx", this.rx);
        node.setAttribute("ry", this.ry);
        return node;
    }

    mutate(cfg) {
        let clone = new this.constructor(0, 0);
        clone.center = this.center.slice();
        clone.rx = this.rx;
        clone.ry = this.ry;

        switch (Math.floor(Math.random()*3)) {
            case 0:
                let angle = Math.random() * 2 * Math.PI;
                let radius = Math.random() * 20;
                clone.center[0] += ~~(radius * Math.cos(angle));
                clone.center[1] += ~~(radius * Math.sin(angle));
            break;

            case 1:
                clone.rx += (Math.random()-0.5) * 20;
                clone.rx = Math.max(1, ~~clone.rx);
            break;

            case 2:
                clone.ry += (Math.random()-0.5) * 20;
                clone.ry = Math.max(1, ~~clone.ry);
            break;
        }

        return clone.computeBbox();
    }

    computeBbox() {
```

```javascript
        this.bbox = {
            left: this.center[0] - this.rx,
            top: this.center[1] - this.ry,
            width: 2*this.rx,
            height: 2*this.ry
        };
        return this;
    }
}

class Smiley extends Shape {
    constructor(w, h) {
        super(w, h);
        this.center = Shape.randomPoint(w, h);
        this.text = "☺";
        this.fontSize = 16;
        this.computeBbox();
    }

    computeBbox() {
        let tmp = new Canvas(1, 1);
        tmp.ctx.font = `${this.fontSize}px sans-serif`;
        let w = ~~(tmp.ctx.measureText(this.text).width);

        this.bbox = {
            left: ~~(this.center[0] - w/2),
            top: ~~(this.center[1] - this.fontSize/2),
            width: w,
            height: this.fontSize
        };
        return this;
    }

    render(ctx) {
        ctx.textAlign = "center";
        ctx.textBaseline = "middle";
        ctx.font = `${this.fontSize}px sans-serif`;
        ctx.fillText(this.text, this.center[0], this.center[1]);
    }

    mutate(cfg) {
        let clone = new this.constructor(0, 0);
        clone.center = this.center.slice();
        clone.fontSize = this.fontSize;

        switch (Math.floor(Math.random()*2)) {
            case 0:
                let angle = Math.random() * 2 * Math.PI;
                let radius = Math.random() * 20;
                clone.center[0] += ~~(radius * Math.cos(angle));
                clone.center[1] += ~~(radius * Math.sin(angle));
            break;

            case 1:
                clone.fontSize += (Math.random() > 0.5 ? 1 : -1);
                clone.fontSize = Math.max(10, clone.fontSize);
            break;
        }

        return clone.computeBbox();
    }

    toSVG() {
        let text = document.createElementNS(SVGNS, "text");
        text.appendChild(document.createTextNode(this.text));

        text.setAttribute("text-anchor", "middle");
        text.setAttribute("dominant-baseline", "central");
        text.setAttribute("font-size", this.fontSize);
        text.setAttribute("font-family", "sans-serif");
        text.setAttribute("x", this.center[0]);
        text.setAttribute("y", this.center[1]);

        return text;
    }
}

const numberFields = ["computeSize", "viewSize", "steps", "shapes", "alpha", "mutations"];
const boolFields = ["mutateAlpha"];
const fillField = "fill";
const shapeField = "shapeType";
const shapeMap = {
    "triangle": Triangle,
    "rectangle": Rectangle,
    "ellipse": Ellipse,
    "smiley": Smiley
};

function fixRange(range) {
    function sync() {
        let value = range.value;
        range.parentNode.querySelector(".value").innerHTML = value;
    }

    range.oninput = sync;
    sync();
}

function init$1() {
    let ranges = document.querySelectorAll("[type=range]");
    Array.from(ranges).forEach(fixRange);
}

function lock() {
    /* fixme */
}



```

```
function getConfig() {
    let form = document.querySelector("form");
    let cfg = {};

    numberFields.forEach(name => {
        cfg[name] = Number(form.querySelector(`[name=${name}]`).value);
    });

    boolFields.forEach(name => {
        cfg[name] = form.querySelector(`[name=${name}]`).checked;
    });

    cfg.shapeTypes = [];
    let shapeFields = Array.from(form.querySelectorAll(`[name=${shapeField}]`));
    shapeFields.forEach(input => {
        if (!input.checked) { return; }
        cfg.shapeTypes.push(shapeMap[input.value]);
    });

    let fillFields = Array.from(form.querySelectorAll(`[name=${fillField}]`));
    fillFields.forEach(input => {
        if (!input.checked) { return; }

        switch (input.value) {
            case "auto": cfg.fill = "auto"; break;
            case "fixed": cfg.fill = form.querySelector("[name='fill-color']").value; break;
        }
    });

    return cfg;
}

/* State: target canvas, current canvas and a distance value */
class State {
    constructor(target, canvas, distance = Infinity) {
        this.target = target;
        this.canvas = canvas;
        this.distance = (distance == Infinity ? target.distance(canvas) : distance);
    }
}

/* Step: a Shape, color and alpha */
class Step {
    constructor(shape, cfg) {
        this.shape = shape;
        this.cfg = cfg;
        this.alpha = cfg.alpha;

        /* these two are computed during the .compute() call */
        this.color = "#000";
        this.distance = Infinity;
    }

    toSVG() {
        let node = this.shape.toSVG();
        node.setAttribute("fill", this.color);
        node.setAttribute("fill-opacity", this.alpha.toFixed(2));
        return node;
    }

    /* apply this step to a state to get a new state. call only after .compute */
    apply(state) {
        let newCanvas = state.canvas.clone().drawStep(this);
        return new State(state.target, newCanvas, this.distance);
    }

    /* find optimal color and compute the resulting distance */
    compute(state) {
        let pixels = state.canvas.node.width * state.canvas.node.height;
        let offset = this.shape.bbox;

        let imageData = {
            shape: this.shape.rasterize(this.alpha).getImageData(),
            current: state.canvas.getImageData(),
            target: state.target.getImageData()
        };

        let {color, differenceChange} = computeColorAndDifferenceChange(offset, imageData, this.alpha);
        this.color = color;
        let currentDifference = distanceToDifference(state.distance, pixels);
        if (-differenceChange > currentDifference) debugger;
        this.distance = differenceToDistance(currentDifference + differenceChange, pixels);

        return Promise.resolve(this);
    }

    /* return a slightly mutated step */
    mutate() {
        let newShape = this.shape.mutate(this.cfg);
        let mutated = new this.constructor(newShape, this.cfg);
        if (this.cfg.mutateAlpha) {
            let mutatedAlpha = this.alpha + (Math.random()-0.5) * 0.08;
            mutated.alpha = clamp(mutatedAlpha, .1, 1);
        }
        return mutated;
    }
}

class Optimizer {
    constructor(original, cfg) {
        this.cfg = cfg;
        this.state = new State(original, Canvas.empty(cfg));
        this._steps = 0;
        this.onStep = () => {};
        console.log("initial distance %s", this.state.distance);
    }

    start() {
        this._ts = Date.now();
```

```
            this._addShape();
        }

        _addShape() {
            this._findBestStep().then(step => this._optimizeStep(step)).then(step => {
                this._steps++;
                if (step.distance < this.state.distance) { /* better than current state, epic */
                    this.state = step.apply(this.state);
                    console.log("switched to new state (%s) with distance: %s", this._steps, this.state.distance);
                    this.onStep(step);
                } else { /* worse than current state, discard */
                    this.onStep(null);
                }
                this._continue();
            });
        }

        _continue() {
            if (this._steps < this.cfg.steps) {
                setTimeout(() => this._addShape(), 10);
            } else {
                let time = Date.now() - this._ts;
                console.log("target distance %s", this.state.distance);
                console.log("real target distance %s", this.state.target.distance(this.state.canvas));
                console.log("finished in %s", time);
            }
        }

        _findBestStep() {
            const LIMIT = this.cfg.shapes;

            let bestStep = null;
            let promises = [];

            for (let i=0;i<LIMIT;i++) {
                let shape = Shape.create(this.cfg);

                let promise = new Step(shape, this.cfg).compute(this.state).then(step => {
                    if (!bestStep || step.distance < bestStep.distance) {
                        bestStep = step;
                    }
                });
                promises.push(promise);
            }

            return Promise.all(promises).then(() => bestStep);
        }

        _optimizeStep(step) {
            const LIMIT = this.cfg.mutations;

            let totalAttempts = 0;
            let successAttempts = 0;
            let failedAttempts = 0;
            let resolve = null;
            let bestStep = step;
            let promise = new Promise(r => resolve = r);

            let tryMutation = () => {
                if (failedAttempts >= LIMIT) {
                    console.log("mutation optimized distance from %s to %s in (%s good, %s total) attempts", arguments[0].distance, b
                    return resolve(bestStep);
                }

                totalAttempts++;
                bestStep.mutate().compute(this.state).then(mutatedStep => {
                    if (mutatedStep.distance < bestStep.distance) { /* success */
                        successAttempts++;
                        failedAttempts = 0;
                        bestStep = mutatedStep;
                    } else { /* failure */
                        failedAttempts++;
                    }

                    tryMutation();
                });
            };

            tryMutation();

            return promise;
        }
    }

    const nodes = {
        output: document.querySelector("#output"),
        original: document.querySelector("#original"),
        steps: document.querySelector("#steps"),
        raster: document.querySelector("#raster"),
        vector: document.querySelector("#vector"),
        vectorText: document.querySelector("#vector-text"),
        types: Array.from(document.querySelectorAll("#output [name=type]"))
    };

    let steps;

    function go(original, cfg) {
        lock();

        nodes.steps.innerHTML = "";
        nodes.original.innerHTML = "";
        nodes.raster.innerHTML = "";
        nodes.vector.innerHTML = "";
        nodes.vectorText.value = "";

        nodes.output.style.display = "";
        nodes.original.appendChild(original.node);

        let optimizer = new Optimizer(original, cfg);
        steps = 0;

        let cfg2 = Object.assign({}, cfg, {width:cfg.scale*cfg.width, height:cfg.scale*cfg.height});
        let result = Canvas.empty(cfg2, false);
        result.ctx.scale(cfg.scale, cfg.scale);
        nodes.raster.appendChild(result.node);

        let svg = Canvas.empty(cfg, true);
        svg.setAttribute("width", cfg2.width);
        svg.setAttribute("height", cfg2.height);
        nodes.vector.appendChild(svg);

        let serializer = new XMLSerializer();

        optimizer.onStep = (step) => {
            if (step) {
                result.drawStep(step);
                svg.appendChild(step.toSVG());
                let percent = (100*(1-step.distance)).toFixed(2);
                nodes.vectorText.value = serializer.serializeToString(svg);
                nodes.steps.innerHTML = `(${++steps} of ${cfg.steps}, ${percent}% similar)`;
            }
        };
        optimizer.start();

        document.documentElement.scrollTop = document.documentElement.scrollHeight;
    }

    function onSubmit(e) {
        e.preventDefault();

        let inputFile = document.querySelector("input[type=file]");
        let inputUrl = document.querySelector("input[name=url]");

        let url = "test";
        if (inputFile.files.length > 0) {
            let file = inputFile.files[0];
            url = URL.createObjectURL(file);
        } else if (inputUrl.value) {
            url = inputUrl.value;
        }

        let cfg = getConfig();

        Canvas.original(url, cfg).then(original => go(original, cfg));
    }

    function init$$1() {
        nodes.output.style.display = "none";
        nodes.types.forEach(input => input.addEventListener("click", syncType));
        init$1();
        syncType();
        document.querySelector("form").addEventListener("submit", onSubmit);
    }

    function syncType() {
        nodes.output.className = "";
        nodes.types.forEach(input => {
            if (input.checked) { nodes.output.classList.add(input.value); }
        });
    }

    init$$1();

    }());
```