

```

1  (function () {
2  'use strict';
3
4  const SVGNS = "http://www.w3.org/2000/svg";
5
6  (function() {
7    const values = [.5, .7, .1, .2, .8, .4, .9, .3, .6, .01, .99, .68, .38, .18, .77, .91, .53, .22, .47];
8    function r() {
9      r.seed++;
10     return values[r.seed % values.length];
11   }
12   r.seed = 0;
13   // Math.random = r;
14   })();
15
16   function clamp(x, min, max) {
17     return Math.max(min, Math.min(max, x));
18   }
19
20   function clampColor(x) {
21     return clamp(x, 0, 255);
22   }
23
24   function distanceToDifference(distance, pixels) {
25     return Math.pow(distance*255, 2) * (3 * pixels);
26   }
27
28   function differenceToDistance(difference, pixels) {
29     return Math.sqrt(difference / (3 * pixels))/255;
30   }
31
32   function difference(data, dataOther) {
33     let sum = 0, diff;
34     for (let i=0; i<data.data.length; i++) {
35       if (i % 4 == 3) { continue; }
36       diff = dataOther.data[i] - data.data[i];
37       sum = sum + diff*diff;
38     }
39
40     return sum;
41   }
42
43   function computeColor(offset, imageData, alpha) {
44     let color = [0, 0, 0];
45     let {shape, current, target} = imageData;
46     let shapeData = shape.data;
47     let currentData = current.data;
48     let targetData = target.data;
49
50     let si, sx, sy, fi, fx, fy; /* shape-index, shape-x, shape-y, full-index, full-x, full-y */
51     let sw = shape.width;
52     let sh = shape.height;
53     let fw = current.width;
54     let fh = current.height;
55     let count = 0;
56
57     for (sy=0; sy<sh; sy++) {
58       fy = sy + offset.top;
59       if (fy < 0 || fy >= fh) { continue; } /* outside of the large canvas (vertically) */
60
61       for (sx=0; sx<sw; sx++) {
62         fx = offset.left + sx;
63         if (fx < 0 || fx >= fw) { continue; } /* outside of the large canvas (horizontally) */
64
65         si = 4*(sx + sy*sw); /* shape (local) index */
66         if (shapeData[si+3] == 0) { continue; } /* only where drawn */
67
68         fi = 4*(fx + fy*fw); /* full (global) index */
69         color[0] += (targetData[fi] - currentData[fi]) / alpha + currentData[fi];
70         color[1] += (targetData[fi+1] - currentData[fi+1]) / alpha + currentData[fi+1];
71         color[2] += (targetData[fi+2] - currentData[fi+2]) / alpha + currentData[fi+2];
72
73         count++;
74       }
75     }
76
77     return color.map(x => ~~(x/count)).map(clampColor);
78   }
79
80   function computeDifferenceChange(offset, imageData, color) {
81     let {shape, current, target} = imageData;
82     let shapeData = shape.data;
83     let currentData = current.data;
84     let targetData = target.data;
85
86     let a, b, d1r, d1g, d1b, d2r, d2b, d2g;
87     let si, sx, sy, fi, fx, fy; /* shape-index, shape-x, shape-y, full-index */
88     let sw = shape.width;
89     let sh = shape.height;
90     let fw = current.width;
91     let fh = current.height;
92
93     var sum = 0; /* V8 opt bailout with let */
94
95     for (sy=0; sy<sh; sy++) {
96       fy = sy + offset.top;
97       if (fy < 0 || fy >= fh) { continue; } /* outside of the large canvas (vertically) */
98
99       for (sx=0; sx<sw; sx++) {
100         fx = offset.left + sx;
101         if (fx < 0 || fx >= fw) { continue; } /* outside of the large canvas (horizontally) */
102
103         si = 4*(sx + sy*sw); /* shape (local) index */
104         a = shapeData[si+3];
105         if (a == 0) { continue; } /* only where drawn */
106
107         fi = 4*(fx + fy*fw); /* full (global) index */
108

```

```

109         a = a/255;
110         b = 1-a;
111         d1r = targetData[fi]-currentData[fi];
112         d1g = targetData[fi+1]-currentData[fi+1];
113         d1b = targetData[fi+2]-currentData[fi+2];
114
115         d2r = targetData[fi] - (color[0]*a + currentData[fi]*b);
116         d2g = targetData[fi+1] - (color[1]*a + currentData[fi+1]*b);
117         d2b = targetData[fi+2] - (color[2]*a + currentData[fi+2]*b);
118
119         sum -= d1r*d1r + d1g*d1g + d1b*d1b;
120         sum += d2r*d2r + d2g*d2g + d2b*d2b;
121     }
122 }
123
124 return sum;
125 }
126
127 function computeColorAndDifferenceChange(offset, imageData, alpha) {
128     let rgb = computeColor(offset, imageData, alpha);
129     let differenceChange = computeDifferenceChange(offset, imageData, rgb);
130
131     let color = `rgb(${rgb[0]}, ${rgb[1]}, ${rgb[2]})`;
132
133     return {color, differenceChange};
134 }
135
136 function getScale(width, height, limit) {
137     return Math.max(width / limit, height / limit, 1);
138 }
139
140 /* FIXME move to util */
141 function getFill(canvas) {
142     let data = canvas.getImageData();
143     let w = data.width;
144     let h = data.height;
145     let d = data.data;
146     let rgb = [0, 0, 0];
147     let count = 0;
148     let i;
149
150     for (let x=0; x<w; x++) {
151         for (let y=0; y<h; y++) {
152             if (x > 0 && y > 0 && x < w-1 && y < h-1) { continue; }
153             count++;
154             i = 4*(x + y*w);
155             rgb[0] += d[i];
156             rgb[1] += d[i+1];
157             rgb[2] += d[i+2];
158         }
159     }
160
161     rgb = rgb.map(x => ~~(x/count)).map(clampColor);
162     return `rgb(${rgb[0]}, ${rgb[1]}, ${rgb[2]})`;
163 }
164
165 function svgRect(w, h) {
166     let node = document.createElementNS(SVGNS, "rect");
167     node.setAttribute("x", 0);
168     node.setAttribute("y", 0);
169     node.setAttribute("width", w);
170     node.setAttribute("height", h);
171
172     return node;
173 }
174
175 /* Canvas: a wrapper around a <canvas> element */
176 class Canvas {
177     static empty(cfg, svg) {
178         if (svg) {
179             let node = document.createElementNS(SVGNS, "svg");
180             node.setAttribute("viewBox", `0 0 ${cfg.width} ${cfg.height}`);
181             node.setAttribute("clip-path", "url(#clip)");
182
183             let defs = document.createElementNS(SVGNS, "defs");
184             node.appendChild(defs);
185
186             let cp = document.createElementNS(SVGNS, "clipPath");
187             defs.appendChild(cp);
188             cp.setAttribute("id", "clip");
189             cp.setAttribute("clipPathUnits", "objectBoundingBox");
190
191             let rect = svgRect(cfg.width, cfg.height);
192             cp.appendChild(rect);
193
194             rect = svgRect(cfg.width, cfg.height);
195             rect.setAttribute("fill", cfg.fill);
196             node.appendChild(rect);
197
198             return node;
199         } else {
200             return new this(cfg.width, cfg.height).fill(cfg.fill);
201         }
202     }
203
204     static original(url, cfg) {
205         if (url == "test") {
206             return Promise.resolve(this.test(cfg));
207         }
208
209         return new Promise(resolve => {
210             let img = new Image();
211             img.crossOrigin = true;
212             img.src = url;
213             img.onload = e => {
214                 let w = img.naturalWidth;
215                 let h = img.naturalHeight;
216
217                 let computeScale = getScale(w, h, cfg.computeSize);

```

```

218         cfg.width = w / computeScale;
219         cfg.height = h / computeScale;
220
221         let viewScale = getScale(w, h, cfg.viewSize);
222
223         cfg.scale = computeScale / viewScale;
224
225         let canvas = this.empty(cfg);
226         canvas.ctx.drawImage(img, 0, 0, cfg.width, cfg.height);
227
228         if (cfg.fill == "auto") { cfg.fill = getFill(canvas); }
229
230         resolve(canvas);
231     };
232     img.onerror = e => {
233         console.error(e);
234         alert("The image URL cannot be loaded. Does the server support CORS?");
235     };
236     });
237 }
238
239 static test(cfg) {
240     cfg.width = cfg.computeSize;
241     cfg.height = cfg.computeSize;
242     cfg.scale = 1;
243     let [w, h] = [cfg.width, cfg.height];
244
245     let canvas = new this(w, h);
246     canvas.fill("#fff");
247     let ctx = canvas.ctx;
248
249     ctx.fillStyle = "#f00";
250     ctx.beginPath();
251     ctx.arc(w/4, h/2, w/7, 0, 2*Math.PI, true);
252     ctx.fill();
253
254     ctx.fillStyle = "#0f0";
255     ctx.beginPath();
256     ctx.arc(w/2, h/2, w/7, 0, 2*Math.PI, true);
257     ctx.fill();
258
259     ctx.fillStyle = "#00f";
260     ctx.beginPath();
261     ctx.arc(w*3/4, h/2, w/7, 0, 2*Math.PI, true);
262     ctx.fill();
263
264     if (cfg.fill == "auto") { cfg.fill = getFill(canvas); }
265
266     return canvas;
267 }
268
269 constructor(width, height) {
270     this.node = document.createElement("canvas");
271     this.node.width = width;
272     this.node.height = height;
273     this.ctx = this.node.getContext("2d");
274     this._imageData = null;
275 }
276
277 clone() {
278     let otherCanvas = new this.constructor(this.node.width, this.node.height);
279     otherCanvas.ctx.drawImage(this.node, 0, 0);
280     return otherCanvas;
281 }
282
283 fill(color) {
284     this.ctx.fillStyle = color;
285     this.ctx.fillRect(0, 0, this.node.width, this.node.height);
286     return this;
287 }
288
289 getImageData() {
290     if (!this._imageData) {
291         this._imageData = this.ctx.getImageData(0, 0, this.node.width, this.node.height);
292     }
293     return this._imageData;
294 }
295
296 difference(otherCanvas) {
297     let data = this.getImageData();
298     let dataOther = otherCanvas.getImageData();
299
300     return difference(data, dataOther);
301 }
302
303 distance(otherCanvas) {
304     let difference$$1 = this.difference(otherCanvas);
305     return differenceToDistance(difference$$1, this.node.width*this.node.height);
306 }
307
308 drawStep(step) {
309     this.ctx.globalAlpha = step.alpha;
310     this.ctx.fillStyle = step.color;
311     step.shape.render(this.ctx);
312     return this;
313 }
314 }
315
316 /* Shape: a geometric primitive with a bbox */
317 class Shape {
318     static randomPoint(width, height) {
319         return [~~(Math.random()*width), ~~(Math.random()*height)];
320     }
321
322     static create(cfg) {
323         let ctors = cfg.shapeTypes;
324         let index = Math.floor(Math.random() * ctors.length);
325         let ctor = ctors[index];
326         return new ctor(cfg.width, cfg.height);
327     }
328 }

```

```

327     }
328
329     constructor(w, h) {
330         this.bbox = {};
331     }
332
333     mutate(cfg) { return this; }
334
335     toSVG() {}
336
337     /* get a new smaller canvas with this shape */
338     rasterize(alpha) {
339         let canvas = new Canvas(this.bbox.width, this.bbox.height);
340         let ctx = canvas.ctx;
341         ctx.fillStyle = "#000";
342         ctx.globalAlpha = alpha;
343         ctx.translate(-this.bbox.left, -this.bbox.top);
344         this.render(ctx);
345         return canvas;
346     }
347
348     render(ctx) {}
349 }
350
351 class Polygon extends Shape {
352     constructor(w, h, count) {
353         super(w, h);
354
355         this.points = this._createPoints(w, h, count);
356         this.computeBbox();
357     }
358
359     render(ctx) {
360         ctx.beginPath();
361         this.points.forEach(([x, y], index) => {
362             if (index) {
363                 ctx.lineTo(x, y);
364             } else {
365                 ctx.moveTo(x, y);
366             }
367         });
368         ctx.closePath();
369         ctx.fill();
370     }
371
372     toSVG() {
373         let path = document.createElementNS(SVGNS, "path");
374         let d = this.points.map((point, index) => {
375             let cmd = (index ? "L" : "M");
376             return `${cmd}${point.join(",")}`;
377         }).join("");
378         path.setAttribute("d", `${d}Z`);
379         return path;
380     }
381
382     mutate(cfg) {
383         let clone = new this.constructor(0, 0);
384         clone.points = this.points.map(point => point.slice());
385
386         let index = Math.floor(Math.random() * this.points.length);
387         let point = clone.points[index];
388
389         let angle = Math.random() * 2 * Math.PI;
390         let radius = Math.random() * 20;
391         point[0] += ~(radius * Math.cos(angle));
392         point[1] += ~(radius * Math.sin(angle));
393
394         return clone.computeBbox();
395     }
396
397     computeBbox() {
398         let min = [
399             this.points.reduce((v, p) => Math.min(v, p[0]), Infinity),
400             this.points.reduce((v, p) => Math.min(v, p[1]), Infinity)
401         ];
402         let max = [
403             this.points.reduce((v, p) => Math.max(v, p[0]), -Infinity),
404             this.points.reduce((v, p) => Math.max(v, p[1]), -Infinity)
405         ];
406
407         this.bbox = {
408             left: min[0],
409             top: min[1],
410             width: (max[0]-min[0]) || 1, /* fallback for deformed shapes */
411             height: (max[1]-min[1]) || 1
412         };
413
414         return this;
415     }
416
417     _createPoints(w, h, count) {
418         let first = Shape.randomPoint(w, h);
419         let points = [first];
420
421         for (let i=1; i<count; i++) {
422             let angle = Math.random() * 2 * Math.PI;
423             let radius = Math.random() * 20;
424             points.push([
425                 first[0] + ~(radius * Math.cos(angle)),
426                 first[1] + ~(radius * Math.sin(angle))
427             ]);
428         }
429         return points;
430     }
431 }
432
433 class Triangle extends Polygon {
434     constructor(w, h) {
435         super(w, h, 3);
436     }
437 }

```

```

436     }
437 }
438
439 class Rectangle extends Polygon {
440   constructor(w, h) {
441     super(w, h, 4);
442   }
443
444   mutate(cfg) {
445     let clone = new this.constructor(0, 0);
446     clone.points = this.points.map(point => point.slice());
447
448     let amount = ~~((Math.random()-0.5) * 20);
449
450     switch (Math.floor(Math.random()*4)) {
451       case 0: /* left */
452         clone.points[0][0] += amount;
453         clone.points[3][0] += amount;
454         break;
455       case 1: /* top */
456         clone.points[0][1] += amount;
457         clone.points[1][1] += amount;
458         break;
459       case 2: /* right */
460         clone.points[1][0] += amount;
461         clone.points[2][0] += amount;
462         break;
463       case 3: /* bottom */
464         clone.points[2][1] += amount;
465         clone.points[3][1] += amount;
466         break;
467     }
468
469     return clone.computeBbox();
470   }
471
472   _createPoints(w, h, count) {
473     let p1 = Shape.randomPoint(w, h);
474     let p2 = Shape.randomPoint(w, h);
475
476     let left = Math.min(p1[0], p2[0]);
477     let right = Math.max(p1[0], p2[0]);
478     let top = Math.min(p1[1], p2[1]);
479     let bottom = Math.max(p1[1], p2[1]);
480
481     return [
482       [left, top],
483       [right, top],
484       [right, bottom],
485       [left, bottom]
486     ];
487   }
488 }
489
490 class Ellipse extends Shape {
491   constructor(w, h) {
492     super(w, h);
493
494     this.center = Shape.randomPoint(w, h);
495     this.rx = 1 + ~~(Math.random() * 20);
496     this.ry = 1 + ~~(Math.random() * 20);
497
498     this.computeBbox();
499   }
500
501   render(ctx) {
502     ctx.beginPath();
503     ctx.ellipse(this.center[0], this.center[1], this.rx, this.ry, 0, 0, 2*Math.PI, false);
504     ctx.fill();
505   }
506
507   toSVG() {
508     let node = document.createElementNS(SVGNS, "ellipse");
509     node.setAttribute("cx", this.center[0]);
510     node.setAttribute("cy", this.center[1]);
511     node.setAttribute("rx", this.rx);
512     node.setAttribute("ry", this.ry);
513     return node;
514   }
515
516   mutate(cfg) {
517     let clone = new this.constructor(0, 0);
518     clone.center = this.center.slice();
519     clone.rx = this.rx;
520     clone.ry = this.ry;
521
522     switch (Math.floor(Math.random()*3)) {
523       case 0:
524         let angle = Math.random() * 2 * Math.PI;
525         let radius = Math.random() * 20;
526         clone.center[0] += ~(radius * Math.cos(angle));
527         clone.center[1] += ~(radius * Math.sin(angle));
528         break;
529
530       case 1:
531         clone.rx += (Math.random()-0.5) * 20;
532         clone.rx = Math.max(1, ~~clone.rx);
533         break;
534
535       case 2:
536         clone.ry += (Math.random()-0.5) * 20;
537         clone.ry = Math.max(1, ~~clone.ry);
538         break;
539     }
540
541     return clone.computeBbox();
542   }
543
544   computeBbox() {

```

```

545     this.bbox = {
546       left: this.center[0] - this.rx,
547       top: this.center[1] - this.ry,
548       width: 2*this.rx,
549       height: 2*this.ry
550     };
551     return this;
552   }
553 }
554
555 class Smiley extends Shape {
556   constructor(w, h) {
557     super(w, h);
558     this.center = Shape.randomPoint(w, h);
559     this.text = "O";
560     this.fontSize = 16;
561     this.computeBbox();
562   }
563
564   computeBbox() {
565     let tmp = new Canvas(1, 1);
566     tmp.ctx.font = `${this.fontSize}px sans-serif`;
567     let w = ~~(tmp.ctx.measureText(this.text).width);
568
569     this.bbox = {
570       left: ~~(this.center[0] - w/2),
571       top: ~~(this.center[1] - this.fontSize/2),
572       width: w,
573       height: this.fontSize
574     };
575     return this;
576   }
577
578   render(ctx) {
579     ctx.textAlign = "center";
580     ctx.textBaseline = "middle";
581     ctx.font = `${this.fontSize}px sans-serif`;
582     ctx.fillText(this.text, this.center[0], this.center[1]);
583   }
584
585   mutate(cfg) {
586     let clone = new this.constructor(0, 0);
587     clone.center = this.center.slice();
588     clone.fontSize = this.fontSize;
589
590     switch (Math.floor(Math.random()*2)) {
591       case 0:
592         let angle = Math.random() * 2 * Math.PI;
593         let radius = Math.random() * 20;
594         clone.center[0] += ~~(radius * Math.cos(angle));
595         clone.center[1] += ~~(radius * Math.sin(angle));
596         break;
597
598       case 1:
599         clone.fontSize += (Math.random() > 0.5 ? 1 : -1);
600         clone.fontSize = Math.max(10, clone.fontSize);
601         break;
602     }
603
604     return clone.computeBbox();
605   }
606
607   toSVG() {
608     let text = document.createElementNS(SVGNS, "text");
609     text.appendChild(document.createTextNode(this.text));
610
611     text.setAttribute("text-anchor", "middle");
612     text.setAttribute("dominant-baseline", "central");
613     text.setAttribute("font-size", this.fontSize);
614     text.setAttribute("font-family", "sans-serif");
615     text.setAttribute("x", this.center[0]);
616     text.setAttribute("y", this.center[1]);
617
618     return text;
619   }
620 }
621
622 const numberFields = ["computeSize", "viewSize", "steps", "shapes", "alpha", "mutations"];
623 const boolFields = ["mutateAlpha"];
624 const fillField = "fill";
625 const shapeField = "shapeType";
626 const shapeMap = {
627   "triangle": Triangle,
628   "rectangle": Rectangle,
629   "ellipse": Ellipse,
630   "smiley": Smiley
631 };
632
633 function fixRange(range) {
634   function sync() {
635     let value = range.value;
636     range.parentNode.querySelector(".value").innerHTML = value;
637   }
638
639   range.oninput = sync;
640   sync();
641 }
642
643 function init$1() {
644   let ranges = document.querySelectorAll("[type=range]");
645   Array.from(ranges).forEach(fixRange);
646 }
647
648 function lock() {
649   /* fixme */
650 }
651
652
653

```

```

654 function getConfig() {
655   let form = document.querySelector("form");
656   let cfg = {};
657
658   numberFields.forEach(name => {
659     cfg[name] = Number(form.querySelector(`[name=${name}]`).value);
660   });
661
662   boolFields.forEach(name => {
663     cfg[name] = form.querySelector(`[name=${name}]`).checked;
664   });
665
666   cfg.shapeTypes = [];
667   let shapeFields = Array.from(form.querySelectorAll(`[name=${shapeField}]`));
668   shapeFields.forEach(input => {
669     if (!input.checked) { return; }
670     cfg.shapeTypes.push(shapeMap[input.value]);
671   });
672
673   let fillFields = Array.from(form.querySelectorAll(`[name=${fillField}]`));
674   fillFields.forEach(input => {
675     if (!input.checked) { return; }
676
677     switch (input.value) {
678       case "auto": cfg.fill = "auto"; break;
679       case "fixed": cfg.fill = form.querySelector("[name='fill-color']").value; break;
680     }
681   });
682
683   return cfg;
684 }
685
686 /* State: target canvas, current canvas and a distance value */
687 class State {
688   constructor(target, canvas, distance = Infinity) {
689     this.target = target;
690     this.canvas = canvas;
691     this.distance = (distance == Infinity ? target.distance(canvas) : distance);
692   }
693 }
694
695 /* Step: a Shape, color and alpha */
696 class Step {
697   constructor(shape, cfg) {
698     this.shape = shape;
699     this.cfg = cfg;
700     this.alpha = cfg.alpha;
701
702     /* these two are computed during the .compute() call */
703     this.color = "#000";
704     this.distance = Infinity;
705   }
706
707   toSVG() {
708     let node = this.shape.toSVG();
709     node.setAttribute("fill", this.color);
710     node.setAttribute("fill-opacity", this.alpha.toFixed(2));
711     return node;
712   }
713
714   /* apply this step to a state to get a new state. call only after .compute */
715   apply(state) {
716     let newCanvas = state.canvas.clone().drawStep(this);
717     return new State(state.target, newCanvas, this.distance);
718   }
719
720   /* find optimal color and compute the resulting distance */
721   compute(state) {
722     let pixels = state.canvas.node.width * state.canvas.node.height;
723     let offset = this.shape.bbox;
724
725     let imageData = {
726       shape: this.shape.rasterize(this.alpha).getImageData(),
727       current: state.canvas.getImageData(),
728       target: state.target.getImageData()
729     };
730
731     let {color, differenceChange} = computeColorAndDifferenceChange(offset, imageData, this.alpha);
732     this.color = color;
733     let currentDifference = distanceToDifference(state.distance, pixels);
734     if (-differenceChange > currentDifference) debugger;
735     this.distance = differenceToDistance(currentDifference + differenceChange, pixels);
736
737     return Promise.resolve(this);
738   }
739
740   /* return a slightly mutated step */
741   mutate() {
742     let newShape = this.shape.mutate(this.cfg);
743     let mutated = new this.constructor(newShape, this.cfg);
744     if (this.cfg.mutateAlpha) {
745       let mutatedAlpha = this.alpha + (Math.random()-0.5) * 0.08;
746       mutated.alpha = clamp(mutatedAlpha, .1, 1);
747     }
748     return mutated;
749   }
750 }
751
752 class Optimizer {
753   constructor(original, cfg) {
754     this.cfg = cfg;
755     this.state = new State(original, Canvas.empty(cfg));
756     this._steps = 0;
757     this.OnStep = () => {};
758     console.log("initial distance %s", this.state.distance);
759   }
760
761   start() {
762     this._ts = Date.now();

```

```

763     this._addShape();
764 }
765
766 _addShape() {
767     this._findBestStep().then(step => this._optimizeStep(step)).then(step => {
768         this._steps++;
769         if (step.distance < this.state.distance) { /* better than current state, epic */
770             this.state = step.apply(this.state);
771             console.log("switched to new state (%s) with distance: %s", this._steps, this.state.distance);
772             this.onStep(step);
773         } else { /* worse than current state, discard */
774             this.onStep(null);
775         }
776         this._continue();
777     });
778 }
779
780 _continue() {
781     if (this._steps < this.cfg.steps) {
782         setTimeout(() => this._addShape(), 10);
783     } else {
784         let time = Date.now() - this._ts;
785         console.log("target distance %s", this.state.distance);
786         console.log("real target distance %s", this.state.target.distance(this.state.canvas));
787         console.log("finished in %s", time);
788     }
789 }
790
791 _findBestStep() {
792     const LIMIT = this.cfg.shapes;
793
794     let bestStep = null;
795     let promises = [];
796
797     for (let i=0; i<LIMIT; i++) {
798         let shape = Shape.create(this.cfg);
799
800         let promise = new Step(shape, this.cfg).compute(this.state).then(step => {
801             if (!bestStep || step.distance < bestStep.distance) {
802                 bestStep = step;
803             }
804         });
805         promises.push(promise);
806     }
807
808     return Promise.all(promises).then(() => bestStep);
809 }
810
811 _optimizeStep(step) {
812     const LIMIT = this.cfg.mutations;
813
814     let totalAttempts = 0;
815     let successAttempts = 0;
816     let failedAttempts = 0;
817     let resolve = null;
818     let bestStep = step;
819     let promise = new Promise(r => resolve = r);
820
821     let tryMutation = () => {
822         if (failedAttempts >= LIMIT) {
823             console.log("mutation optimized distance from %s to %s in (%s good, %s total) attempts", arguments[0].distance, k
824                 , bestStep.distance, successAttempts, failedAttempts);
825             return resolve(bestStep);
826         }
827
828         totalAttempts++;
829         bestStep.mutate().compute(this.state).then(mutatedStep => {
830             if (mutatedStep.distance < bestStep.distance) { /* success */
831                 successAttempts++;
832                 failedAttempts = 0;
833                 bestStep = mutatedStep;
834             } else { /* failure */
835                 failedAttempts++;
836             }
837
838             tryMutation();
839         });
840     };
841
842     tryMutation();
843
844     return promise;
845 }
846
847 const nodes = {
848     output: document.querySelector("#output"),
849     original: document.querySelector("#original"),
850     steps: document.querySelector("#steps"),
851     raster: document.querySelector("#raster"),
852     vector: document.querySelector("#vector"),
853     vectorText: document.querySelector("#vector-text"),
854     types: Array.from(document.querySelectorAll("#output [name=type]"))
855 };
856
857 let steps;
858
859 function go(original, cfg) {
860     lock();
861
862     nodes.steps.innerHTML = "";
863     nodes.original.innerHTML = "";
864     nodes.raster.innerHTML = "";
865     nodes.vector.innerHTML = "";
866     nodes.vectorText.value = "";
867
868     nodes.output.style.display = "";
869     nodes.original.appendChild(original.node);
870
871     let optimizer = new Optimizer(original, cfg);

```



```

872     steps = 0;
873
874     let cfg2 = Object.assign({}, cfg, {width:cfg.scale*cfg.width, height:cfg.scale*cfg.height});
875     let result = Canvas.empty(cfg2, false);
876     result.ctx.scale(cfg.scale, cfg.scale);
877     nodes.raster.appendChild(result.node);
878
879     let svg = Canvas.empty(cfg, true);
880     svg.setAttribute("width", cfg2.width);
881     svg.setAttribute("height", cfg2.height);
882     nodes.vector.appendChild(svg);
883
884     let serializer = new XMLSerializer();
885
886     optimizer.onStep = (step) => {
887         if (step) {
888             result.drawStep(step);
889             svg.appendChild(step.toSVG());
890             let percent = (100*(1-step.distance)).toFixed(2);
891             nodes.vectorText.value = serializer.serializeToString(svg);
892             nodes.steps.innerHTML = `(${++steps} of ${cfg.steps}, ${percent}% similar)`;
893         }
894     };
895     optimizer.start();
896
897     document.documentElement.scrollTop = document.documentElement.scrollHeight;
898 }
899
900 function onSubmit(e) {
901     e.preventDefault();
902
903     let inputFile = document.querySelector("input[type=file]");
904     let inputUrl = document.querySelector("input[name=url]");
905
906     let url = "test";
907     if (inputFile.files.length > 0) {
908         let file = inputFile.files[0];
909         url = URL.createObjectURL(file);
910     } else if (inputUrl.value) {
911         url = inputUrl.value;
912     }
913
914     let cfg = getConfig();
915
916     Canvas.original(url, cfg).then(original => go(original, cfg));
917 }
918
919 function init$$1() {
920     nodes.output.style.display = "none";
921     nodes.types.forEach(input => input.addEventListener("click", syncType));
922     init$1();
923     syncType();
924     document.querySelector("form").addEventListener("submit", onSubmit);
925 }
926
927 function syncType() {
928     nodes.output.className = "";
929     nodes.types.forEach(input => {
930         if (input.checked) { nodes.output.classList.add(input.value); }
931     });
932 }
933
934 init$1();
935
936 }());

```