three-algorithms

Claude Heiland-Allen

2018

Contents

1 .gitignore

```
mandelbrot
*.mkv
*.wav
*.pgm
*.png
*.aux
*.log
*.pdf
```

2 Makefile

```
all: three-algorithms.mkv three-algorithms.pdf

binaural: three-algorithms-binaural.mkv

clean:
	-rm *.mkv *.wav *.pgm *.png *.pdf *.aux *.log mandelbrot

three-algorithms.mkv: mandelbrot-algorithm.mkv
	ffmpeg -i mandelbrot-algorithm.mkv -vf pad=1920:1080:0:60 -pix_fmt ↙
	    ↳ yuv420p -profile:v high -level:v 4.1 -crf:v 20 -codec:a copy -↙
	    ↳ max_muxing_queue_size 99999 -y three-algorithms.mkv

three-algorithms.pdf: three-algorithms.md
	pandoc -Vgeometry=margin=2cm three-algorithms.md -o three-algorithms.pdf

three-algorithms-binaural.mkv: mandelbrot-binaural.mkv
	mv mandelbrot-binaural.mkv three-algorithms-binaural.mkv

mandelbrot-binaural.mkv: mandelbrot-binaural.wav three-algorithms.mkv
	ffmpeg -i mandelbrot-binaural.wav -i three-algorithms.mkv -map:a 0:0 -↙
	    ↳ map:v 1:0 -codec:a copy -codec:v copy -shortest mandelbrot-↙
	    ↳ binaural.mkv
```

```
mandelbrot-binaural.wav: mandelbrot-algorithm.pcm.wav
	ecasound -f:f32_le,2,48000 -G:jack,,send -i:jack,↙
	    ↳ ambix_binaural_standalone_o3 -o:mandelbrot-binaural.wav &
	ecasound -f:f32_le,16,48000 -G:jack,,recv -i:mandelbrot-algorithm.pcm.↙
	    ↳ wav -o:jack,ambix_binaural_standalone_o3
	killall ecasound

mandelbrot-algorithm.pcm.wav: mandelbrot-algorithm.wav
	sox mandelbrot-algorithm.wav mandelbrot-algorithm.wavpcm
	mv mandelbrot-algorithm.wavpcm mandelbrot-algorithm.pcm.wav

mandelbrot-algorithm.wav: three-algorithms.mkv
	ffmpeg -i three-algorithms.mkv -codec:a copy mandelbrot-algorithm.wav

mandelbrot-algorithm.mkv: titles.1.mkv titles.2.mkv titles.3.mkv mandelbrot-↙
    ↳ algorithm-0.mkv titles.4.mkv mandelbrot-algorithm-1.mkv titles.5.mkv ↙
    ↳ mandelbrot-algorithm-2.mkv titles.6.mkv
	ffmpeg -i mandelbrot-algorithm.txt -codec:a copy -codec:v copy ↙
	    ↳ mandelbrot-algorithm.mkv

mandelbrot: mandelbrot.c
	gcc-7 -std=c99 -Wall -Wextra -pedantic -O3 -fopenmp -o mandelbrot ↙
	    ↳ mandelbrot.c -lsndfile -lm

mandelbrot-algorithm-%.mkv: mandelbrot
	-rm out.wav video.mkv
	./mandelbrot $* 128 480 240 0 0 2 | ffmpeg -framerate 60 -f image2pipe -↙
	    ↳ i - -sws_flags neighbor -s 1920x960 -codec:v png video.mkv
	ffmpeg -i out.wav -i video.mkv -codec:v copy -codec:a copy $@

silence.wav:
	ecasound -f:f32_le,16,48000 -i:jack -o:silence.wav -t:4

titles.pdf: titles.tex
	pdflatex titles.tex

titles.1.png titles.2.png titles.3.png titles.4.png titles.5.png titles.6.png: ↙
    ↳ titles.pdf
	gs -sDEVICE=pngalpha -dTextAlphaBits=4 -o titles.%d.png -r960 titles.pdf

%.pgm: %.png
	convert $< -background white $@

%.mkv: %.pgm silence.wav
	-rm t.?.pgm
	ln -s $< t.0.pgm
	ln -s $< t.1.pgm
	ffmpeg -i silence.wav -r 1/4 -i t.%d.pgm -pix_fmt rgb24 -vf negate,fps=↙
	    ↳ fps=60,fade=in:0:60,fade=out:180:60 -r 60 -codec:v png -codec:a ↙
	    ↳ copy -t 4 $@
```

3 mandelbrot-algorithm.txt

```
ffconcat version 1.0
file titles.1.mkv
file titles.2.mkv
```

```
file titles.3.mkv
file mandelbrot-algorithm-0.mkv
file titles.4.mkv
file mandelbrot-algorithm-1.mkv
file titles.5.mkv
file mandelbrot-algorithm-2.mkv
file titles.6.mkv
```

4 mandelbrot.c

```
// http://mathr.co.uk/blog/2014-11-02_practical_interior_distance_rendering.html

#include <complex.h>
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include <sndfile.h>

static inline double cnorm(double _Complex z)
{
  double x = creal(z);
  double y = cimag(z);
  return x * x + y * y;
}

// http://burtleburtle.net/bob/hash/integer.html
static uint32_t burtle_hash(uint32_t a)
{
    a = (a+0x7ed55d16) + (a<<12);
    a = (a^0xc761c23c) ^ (a>>19);
    a = (a+0x165667b1) + (a<<5);
    a = (a+0xd3a2646c) ^ (a<<9);
    a = (a+0xfd7046c5) + (a<<3);
    a = (a^0xb55a4f09) ^ (a>>16);
    return a;
}

// pseudo-random uniform number in [0,1)
static double uniform(uint32_t a, uint32_t b, uint32_t c)
{
  //return rand() / (double) RAND_MAX;
  return burtle_hash(a ^ burtle_hash(b ^ burtle_hash(c))) / (double) (0↙
      ↪ x100000000LL);
}

#define SR 48000
#define FPS 60
#define BUFLEN (SR/FPS)
#define CHANNELS 16

static float buffer[BUFLEN][CHANNELS];
static int bufptr = 0;
SNDFILE *sndfile;

unsigned char *image;
```

```
int width;
int height;

int frame = 0;

static inline void image_save_ppm(unsigned char *image, int width, int height, ↙
    ↳ const char *filename);

static void sonify(double _Complex Z, uint32_t n, uint32_t p, uint32_t q)
{
  double X = creal(Z);
  double Y = cimag(Z);
  if (isnan(X) || isnan(Y) || isinf(X) || isinf(Y)) { X = 0; Y = 0; }
  double R = X * X + Y * Y + 1;
  double x =   2 * Y / R;
  double y =  -2 * X / R;
  double z =  (2 - R)/ R;
  // https://blueripplesound.com/b-format
  double o = uniform(n, p, q) - 0.5;
#define b buffer[bufptr]
  // order 0
  b[ 0] = o * 1;
  // order 1
  b[ 1] = o * y;
  b[ 2] = o * z;
  b[ 3] = o * x;
  //  order 2
  b[ 4] = o * sqrt(3) * x * y;
  b[ 5] = o * sqrt(3) * y * z;
  b[ 6] = o * 1/2. * (3 * z * z - 1);
  b[ 7] = o * sqrt(3) * x * z;
  b[ 8] = o * sqrt(3/4.) * (x * x - y * y);
  // order 3
  b[ 9] = o * sqrt(5/8.) * y * (3 * x * x - y * y);
  b[10] = o * sqrt(15) * x * y * z;
  b[11] = o * sqrt(3/8.) * y * (5 * z * z - 1);
  b[12] = o * 1/2. * z * (5 * z * z - 3);
  b[13] = o * sqrt(3/8.) * x * (5 * z * z - 1);
  b[14] = o * sqrt(15/4.) * z * (x * x - y * y);
  b[15] = o * sqrt(5/8.) * x * (x * x - 3 * y * y);
#undef b
  if (++bufptr == BUFLEN)
  {
    bufptr = 0;
    sf_writef_float(sndfile, &buffer[0][0], BUFLEN);
    char filename[100];
    snprintf(filename, 100, "%08d.ppm", frame++);
    image_save_ppm(image, width, height, filename);
  }
}

static void sonify_finish(void)
{
  if (bufptr)
  {
    for (int i = bufptr; i < BUFLEN; ++i)
    {
```

```
        for (int c = 0; c < CHANNELS; ++c)
        {
          buffer[i][c] = 0;
        }
      }
      sf_writef_float(sndfile, &buffer[0][0], BUFLEN);
      char filename[100];
      snprintf(filename, 100, "%08d.ppm", frame++);
      image_save_ppm(image, width, height, filename);
    }
  }

  const double pi = 3.141592653589793;
  const double infinity = 1.0 / 0.0;
  const double phi = 1.618033988749895; // (sqrt(5.0) + 1.0) / 2.0;
  const double colour_modulus = 5.7581917135421046e-2; // (1.0 + 1.0 / (phi * phi)↙
      ↳ ) / 24.0;
  const double escape_radius_2 = 512.0 * 512.0;

  const int BIAS_UNKNOWN = 0;
  const int BIAS_INTERIOR = 1;
  const int BIAS_EXTERIOR = 2;

  const int ALGORITHM_PLAIN = 0;
  const int ALGORITHM_UNBIASED = 1;
  const int ALGORITHM_BIASED = 2;
  const int ALGORITHM_ANALYSE = 4;

  static inline double cabs2(complex double z) {
    return creal(z) * creal(z) + cimag(z) * cimag(z);
  }

  static inline unsigned char *image_new(int width, int height) {
    return malloc(width * height * 3);
  }

  static inline void image_clear(unsigned char *image, int width, int height)
  {
    memset(image, 0, width * height * 3);
  }

  static inline void image_delete(unsigned char *image) {
    free(image);
  }

  static inline void image_save_ppm(unsigned char *image, int width, int height, ↙
      ↳ const char *filename) {
    FILE *f = stdout;//fopen(filename, "wb");
    if (f) {
      fprintf(f, "P6\n%d %d\n255\n", width, height);
      fwrite(image, width * height * 3, 1, f);
      //fclose(f);
    } else {
      fprintf(stderr, "ERROR saving '%s'\n", filename);
    }
  }
```

```
static inline void image_poke(unsigned char *image, int width, int i, int j, int r, int g, int b) {
  int k = (width * j + i) * 3;
  image[k++] = r;
  image[k++] = g;
  image[k  ] = b;
}

static inline void colour_hsv_to_rgb(double h, double s, double v, double *r, double *g, double *b) {
  double i, f, p, q, t;
  if (s == 0) { *r = *g = *b = v; } else {
    h = 6 * (h - floor(h));
    int ii = i = floor(h);
    f = h - i;
    p = v * (1 - s);
    q = v * (1 - (s * f));
    t = v * (1 - (s * (1 - f)));
    switch(ii) {
      case 0: *r = v; *g = t; *b = p; break;
      case 1: *r = q; *g = v; *b = p; break;
      case 2: *r = p; *g = v; *b = t; break;
      case 3: *r = p; *g = q; *b = v; break;
      case 4: *r = t; *g = p; *b = v; break;
      default:*r = v; *g = p; *b = q; break;
    }
  }
}

static inline void colour_to_bytes(double r, double g, double b, int *r_out, int *g_out, int *b_out) {
  *r_out = fmin(fmax(255 * r, 0), 255);
  *g_out = fmin(fmax(255 * g, 0), 255);
  *b_out = fmin(fmax(255 * b, 0), 255);
}

static inline void colour_mandelbrot(unsigned char *image, int width, int i, int j, int period, double distance) {
  double r, g, b;
  colour_hsv_to_rgb(period * colour_modulus, 0.5, tanh(distance), &r, &g, &b);
  int ir, ig, ib;
  colour_to_bytes(r, g, b, &ir, &ig, &ib);
  image_poke(image, width, i, j, ir, ig, ib);
}

static inline void colour_analysis(unsigned char *image, int width, int i, int j, int bias, int outcome) {
  int ir = 0, ig = 0, ib = 0;
  if (bias == outcome) {
    ig = 255;
  } else if (bias == BIAS_INTERIOR && outcome == BIAS_EXTERIOR) {
    ib = 255;
  } else if (bias == BIAS_EXTERIOR && outcome == BIAS_INTERIOR) {
    ir = 255;
  }
  image_poke(image, width, i, j, ir, ig, ib);
}
```

```
210
    static inline int attractor(complex double *z_out, complex double *dz_out, ↙
        ↳ complex double z0, complex double c, int period) {
      double epsilon_2 = 1e-20;
      complex double zz = z0;
      for (int j = 0; j < 64; ++j) {
215     complex double z = zz;
        complex double dz = 1;
        for (int i = 0; i < period; ++i) {
          dz = 2.0 * z * dz;
          z = z * z + c;
220       sonify(z, period, i, -1);
        }
        complex double zz1 = zz - (z  - zz) / (dz - 1.0);
        if (cabs2(zz1 - zz) < epsilon_2) {
          *z_out = z;
225       *dz_out = dz;
          return 1;
        }
        zz = zz1;
      }
230   return 0;
    }

    static inline double interior_distance(complex double z0, complex double c, int ↙
        ↳ period) {
      complex double z = z0;
235   complex double dz = 1;
      complex double dzdz = 0;
      complex double dc = 0;
      complex double dcdz = 0;
      for (int p = 0; p < period; ++p) {
240     dcdz = 2 * (z * dcdz + dz * dc);
        dc = 2 * z * dc + 1;
        dzdz = 2 * (dz * dz + z * dzdz);
        dz = 2 * z * dz;
        z = z * z + c;
245     sonify(z, period, p, -1);
      }
      return (1 - cabs2(dz)) / cabs(dcdz + dzdz * dc / (1 - dz));
    }

250 struct partial {
      complex double z;
      int p;
    };

255 static inline void render(unsigned char *image, int algorithm, int maxiters, int↙
        ↳  width, int height, complex double center, double radius) {
      //double pixel_spacing = radius / (height / 2.0);
    //  #pragma omp parallel for schedule(dynamic, 1)
      for (int j = 0; j < height; ++j) {
        struct partial *partials = 0;
260     int bias = BIAS_EXTERIOR, new_bias, npartials;
        if (algorithm & ALGORITHM_BIASED) {
          partials = malloc(maxiters * sizeof(struct partial));
        }
```

```
    for (int i = 0; i < width; ++i) {
      image_poke(image, width, i, j, 255, 255, 255);
      new_bias = BIAS_UNKNOWN;
      npartials = 0;
      double theta0 = 2 * pi * (i + 0.5) / width;
      double theta1 = 2 * pi * (i + 1.5) / width;
      double phi0 = pi * (height / 2.0 - j - 0.5) / height;
      double phi1 = pi * (height / 2.0 - j - 0.5 + (j >= height / 2 ? -1 : 1)) /↙
          ↳ height;
      double X0 = cos(theta0) * cos(phi0);
      double Y0 = sin(theta0) * cos(phi0);
      double Z0 = sin(phi0);
      double X1 = cos(theta1) * cos(phi1);
      double Y1 = sin(theta1) * cos(phi1);
      double Z1 = sin(phi1);
      double x0 = X0 / (1 - Z0);
      double y0 = Y0 / (1 - Z0);
      double x1 = X1 / (1 - Z1);
      double y1 = Y1 / (1 - Z1);
      double pixel_spacing = hypot(x0 - x1, y0 - y1);
      complex double c = x0 + I * y0;
      complex double z = 0;
      complex double dc = 0;
      double minimum_z2 = infinity;
      int period = 0;
      for (int n = 1; n <= maxiters; ++n) {
        dc = 2 * z * dc + 1;
        z = z * z + c;
        sonify(z, n, -1, -1);
        double z2 = cabs2(z);
        if (z2 < minimum_z2) {
          minimum_z2 = z2;
          period = n;
          if (algorithm & (ALGORITHM_UNBIASED | ALGORITHM_BIASED)) {
            if (algorithm & ALGORITHM_UNBIASED || bias == BIAS_INTERIOR) {
              complex double z0 = 0, dz0 = 0;
              if (attractor(&z0, &dz0, z, c, period)) {
                if (cabs2(dz0) <= 1.0) {
                  if (algorithm & ALGORITHM_ANALYSE) {
                    colour_analysis(image, width, i, j, bias, BIAS_INTERIOR);
                  } else {
                    double distance = interior_distance(z0, c, period) / ↙
                        ↳ pixel_spacing;
                    colour_mandelbrot(image, width, i, j, period, distance);
                  }
                  new_bias = BIAS_INTERIOR;
                  break;
                }
              }
            } else if (algorithm & ALGORITHM_BIASED) {
              partials[npartials].z = z;
              partials[npartials].p = period;
              npartials++;
            }
          }
        }
        if (z2 >= escape_radius_2) {
```

```
            if (algorithm & ALGORITHM_ANALYSE) {
              colour_analysis(image, width, i, j, bias, BIAS_EXTERIOR);
            } else {
              double distance = sqrt(z2) * log(z2) / (cabs(dc) * pixel_spacing);
              colour_mandelbrot(image, width, i, j, period, distance);
            }
            new_bias = BIAS_EXTERIOR;
            break;
          }
        }
        if (algorithm & ALGORITHM_BIASED) {
          if (bias == BIAS_EXTERIOR && new_bias == BIAS_UNKNOWN) {
            for (int n = 0; n < npartials; ++n) {
              complex double z = partials[n].z;
              int period = partials[n].p;
              complex double z0 = 0, dz0 = 0;
              if (attractor(&z0, &dz0, z, c, period)) {
                if (cabs2(dz0) <= 1.0) {
                  if (algorithm & ALGORITHM_ANALYSE) {
                    colour_analysis(image, width, i, j, bias, BIAS_INTERIOR);
                  } else {
                    double distance = interior_distance(z0, c, period) / ↙
                        ↳ pixel_spacing;
                    colour_mandelbrot(image, width, i, j, period, distance);
                  }
                  new_bias = BIAS_INTERIOR;
                  break;
                }
              }
            }
          }
        }
        if (new_bias == BIAS_UNKNOWN) {
          if (algorithm & ALGORITHM_ANALYSE) {
            colour_analysis(image, width, i, j, bias, BIAS_UNKNOWN);
          } else {
            if (algorithm & (ALGORITHM_UNBIASED | ALGORITHM_BIASED)) {
              colour_mandelbrot(image, width, i, j, period, 0.0);
            } else {
              colour_mandelbrot(image, width, i, j, period, 10.0);
            }
          }
          new_bias = BIAS_EXTERIOR;
        }
        bias = new_bias;
      }
      if (algorithm & ALGORITHM_BIASED) {
        free(partials);
      }
    }
  }

  int main(int argc, char **argv) {
    if (argc != 8) {
      fprintf(stderr,
        "usage: %s algorithm maxiters width height creal cimag radius\n"
        "values for algorithm:\n"
```

```
        "    0  plain (exterior only)\n"
        "    1  unbiased (exterior and interior, unoptimized)\n"
        "    2  biased (exterior and interior optimized by local connectedness)\n"
        "    6  analysis map of biased rendering\n"
        , argv[0]);
      return 1;
    }
    SF_INFO info = { 0, SR, CHANNELS, SF_FORMAT_WAV | SF_FORMAT_FLOAT, 0, 0 };
    sndfile = sf_open("out.wav", SFM_WRITE, &info);
    int algorithm = atoi(argv[1]);
    int maxiters = atoi(argv[2]);
    width = atoi(argv[3]);
    height = atoi(argv[4]);
    complex double center = atof(argv[5]) + I * atof(argv[6]);
    double radius = atof(argv[7]);
    image = image_new(width, height);
//  for (int zoom = 0; zoom < 64; ++zoom)
    int zoom = 8;
    {
      image_clear(image, width, height);
      render(image, algorithm, maxiters, width, height, center, radius * pow(0.5, ↙
          ↪ zoom - 8));
    }
    sonify_finish();
    image_delete(image);
    sf_close(sndfile);
    return 0;
}
```

5 README.md

```
- `jackd` must be running

- run `make`

- needs ~2.5GB space

- `make binaural` version may be out of sync and require editing for latency,
  also kills ecasound processes.  FIXME

- `ambix_binaural_standalone_o3` must be running with JACK with a preset loaded
```

6 three-algorithms.md

```
# Three Algorithms

## Space Concept

The Mandelbrot set is a fractal formed by repeated iterations of the complex
number formula $z \to z^2 + c$.  A first naive algorithm is to simply iterate
repeatedly to check if the sequence diverges.  An improvement is to check for
convergence, using Newton's method for root finding to check that a periodic
limit cycle has derivative less than $1$ in magnitude.  A third optimisation is
to optionally postpone those more expensive checks, depending on the outcome
of the previous pixel - if a pixel diverges, its neighbour is likely to
diverge also, while convergent pixels cluster together too.
```

```
These three algorithms are sonified by stereographic projection of the complex
z coordinate (after each $z \to z^2 + c$ iteration) to the unit Riemann sphere
in $3$D space, whereafter Ambisonics takes over. The waveform so placed in ↙
    ↳ space
is a hash noise function of the current iteration number of the pixel together
with the candidate period of the cycle under investigation for interior checks.

## Technical Desciption

The piece is a fixed multimedia video composition. The soundtrack is encoded
in $16$ channel ACN/SN3D 3rd order ambisonics, with a separate binaural render
provided for preview purposes only. The sound is synchronized to the image,
both are generated by the same fractal iterations. One audio sample frame is
calculated per iteration at $48000$ Hz, a pixel may take many iterations,
thus the image appears in scanline fashion.

The video file is rendered at $480\times 240$ pixels, upscaled $4\times$, which
gives a total audio length of approximately $4$ minutes $30$ seconds.

- audio codec: f32_le 16ch 48000Hz PCM
- video codec: yuv420p h264 profile high level 4.1 crf 20
- media container: Matroska (MKV)

## Requirements

- video playback and projection ($1920\times1080p60$), synchronized to
- sound reproduction (Ambisonics decode to Cube speakers)
- setup time: a simple 5-10mins playthrough (a/v line check)

## Download Links

- <https://mathr.co.uk/three-algorithms/three-algorithms.mkv> 800MB
- <https://mathr.co.uk/three-algorithms/three-algorithms-binaural.mkv> 100MB

## About mathr

Claude Heiland-Allen (<https://mathr.co.uk>) is an artist from London
interested in the complex emergent behaviour of simple systems, and
mathematical aesthetics.

\hfill$\square$
\pagestyle{empty}
```

7 titles.tex

```
\documentclass{article}
\usepackage[paperwidth=2in,paperheight=1in,margin=0.4in]{geometry}
\usepackage{lmodern}
\usepackage{adjustbox}

\renewcommand{\familydefault}{\sfdefault}
\pagestyle{empty}

\begin{document}
\centering
\(z \mapsto z^2 + c\)
\newpage
```

```
three algorithms
\newpage
1. exterior checking
\newpage
2. interior checking
\newpage
3. locally connected
\newpage
mathr.co.uk 2018
\end{document}
```