Pixel to circles project

Mats Hoem Olsen, Katla Mária Meyer, Oda Johanne Matre Simensen

December 5, 2022

Contents

List of Algorithms

Abstract

We describe our implementation in modern C++ of converting raster images of overlapping discs to a vector-based format, using Moore's neighbour tracing algorithm for finding contours, Harris' corner detector, and Postscript®.

1 Problem

As input, we are to generate a square two-dimensional landscape of overlapping circular discs. We chose to use only binary colour data. We are then to create an image processing implementation which can read the image and output a representation of the discs as sets of origin coordinates, radii, and colour indices.

1.1 Assumptions

We assumed all raster points to have values 1 or 0. Visually, this is rendered as white and black respectively. We used the .pbm file format[1] to store our raster image, making sure our image reading module did account for the variable amount of information a .pbm image's header can contain.

We also assumed circles could not be centered outside the image, as our generator makes no such circles, and we have not proofed our processing stage to reliably reconstruct such circles either.

2 Image generation

The image generation module of the project (`image_gen/`) was made to generate plain white square images, and then draw a given number of superimposed discs of random radii and positions. This allowed us to quickly and conveniently generate however many varied images we needed. We decided to leave this module separate and independently executable from the rest of the project, as the origin of the image used did not in principle matter to the function of the project as a whole.

We did consider parallelizing the image generator using some MPI solution, such as openMPI. We may have opted to use some dynamic system of load balancing, allowing separate processes to write separate circles so long as they did not immediately intersect (thus avoiding a race condition), but the scale of the project and the relatively small number of circles we were intending to use meant this would not save any significant amount of time or resources.

The image class (`image.cpp`) allocates a continuous space for the image in the heap upon construction, and deallocates it again afterwards. The image is initialized full of '0', such that it appears white. This space gives each raster point one char to fill, which we deemed reasonably effective as chars typically only take up a byte each. Writing bitwise was an option and would have been even more space-efficient, but we avoided it as we would have had to create our own memory traversal method, possibly at the cost of efficiency. In addition to the constructor and destructor, it features functions for setting any raster point to a value 0/1, drawing a circle at a given origin, and saving the raster image to a .pbm file. From `image.cpp`:

```
image::image(int width, int height){
        this->pixels = new char[width*height];
        this->width = width;
```

```
        this->height = height;

        for (int i = 0; i < (width*height); i++){
                this->pixels[i] = '0';}
}

image::~image(void){
        delete[] this->pixels;}
```

As the image buffer in heap (here called "pixels") is allocated simply as a space in memory, we do our own quick calculations to get the abstraction of two-dimensional space when allocating a point. For image generation, we do not require retrieving information from any specific point, but the same logic would be used. From `image.cpp`:

```
void image::set_point (int x, int y, char colour){
        this->pixels[(y * this->width + x)] = colour;
}
```

The `draw_circle()` function scans line-by-line through the radius^2 square area around the circle's origin, checking the distance from each pixel to the circle origin. This leaves us with a constant 21.5% inefficiency checking points we do not manipulate, which scales with the circle size instead of the unrelated image size.

For the randomization of disc position and radius, we scaled seeded pseudo-random values to be within minimum and maximum constraints: For position, the contours of the image, and for radius an arbitrary min/max size. We seeded `rand()` using the current microsecond `tv_usec`, retrieved from `sys/time.h`, which we thought to be sufficiently unpredictable for this application. We did not randomize colour, drawing discs alternatingly in black and in white.

3 Contour detector

The contour detection algorithm comprises the following:

Algorithm 1 Contour algorithm

$P_0 \leftarrow$ First pixel of a shape
$P_{-1} \leftarrow$ Previous pixel
$P_n \leftarrow P_0$
C is the contour of the shape
while $P_n \neq P_0$ **do**
    $P_{-1} \leftarrow$ Look anti-clockwise from P_{-1}.
    if P_{-1} is foreground **then**
        $C \leftarrow P_{-1}$
        Swap label P_{-1} and P_n
    end if
end while
return C

This algorithm will trace the contour of one blob at a time, and we apply the algorithm repeatedly until we hit the lower right corner of the image. We assume that pixels of value 1 belong to the foreground, and 0 to the background. Pixels are stored in a vector coordinate structure that contains x and y coordinates for all foreground pixels. Depending on image size and how much of the given image is 0/1, the resulting memory use will vary. A more advanced implementation could index both 0 and 1 pixels initially, treating the colour with the least pixels as the foreground and saving only this.

3.1 Exceptions for enclosed shapes

There are certain cases that our code does not account for. Namely, any hole or shape in a whole enclosed within a blob is going to be ignored because the seeking algorithm for finding further blobs skips from the left to the right side of the current blob when re-encountering it.

4 Harris' corner detector

The Harris corner detector algorithm is used to find all the corners in an image. We apply it to one blob contour at a time. The algorithm uses the eigenvalues from the first directional derivative matrix

$$M = \begin{bmatrix} I_x^2 & I_x I_y \\ I_y I_x & I_y^2 \end{bmatrix}$$

After calculating the smallest eigenvalue for every point along the path, we apply a threshold to determine if the eigenvalue is large enough to be considered a corner.

The algorithm we use is:

Algorithm 2 Harris corner detector

```
λ ← {}
C ← {}
for all Points in picture do
    Calculate I_x and I_y
    Make M matrix
    Find the smallest Eigenvalue
    λ ← Eigenvalue
end for
for all points in λ do
    if point > limit then
        C ← point
    end if
end for
return C
```

5 Image viewer

As our final step, after we have extracted origin points and radii from our circles and stored them as 3 values, we wish to also store and draw them more permanently. For this purpose, we chose to use the language Postscript, since it has simple syntax and is relatively easy to convert to other file formats that support vector graphics. We use the command ”arc fill”, which takes x,y coordinates, radii, and arc angles. The ”fill” command just uses a watershed algorithm to fill inn the circle with solid colour. The colour can be specified through the function `setgray`, which allows us to reconstruct both black and white discs just like we had during generation.

At this point, we have reduced each circle down to just 3 data points, using a Circle struct derived from our coords struct with an added radius data point, placed inside vectors for potential mass storage. As we do not need to find any specific circles, only list all of them, the difficulties of traversing a mutable list do not stand out as major a issue.

References

[1] *The PBM Format*, https://netpbm.sourceforge.net/doc/pbm.html, retrieved 2022-12-05.