CONSTRUCTION OF CIRCLE PACKING WITH COMBINATORIAL RICCI FLOW

Ramil Aleskerov
Prof. Bianca Santoro

Abstract

Let *T* be a triangulated planar graph with a set of vertices, *V*. We start off with a random circle packing by assigning random positive integer values, *r* to each vertex, *v* which represent radiuses of spheres located at each vertex. We then run a Ricci flow, which calculates curvatures at every vertex and slightly adjusts radiuses. Each iteration linearly approximates new set of radiuses based on Ricci flow differential equation. Given enough iterations, the algorithm eventually produces a constant-curvature circle packing of the initial planar graph.

Introduction

The algorithm is based on the ideas presented in "Combinatorial Ricci Flows on Surfaces" by Bennett Chow & Feng Luo (2003).

Let *T* be a triangulation in a plane. Let $V = \{v_1, v_2, \ldots, v_N\}$ be a set of vertices of *T*. Let r_i be a randomly assigned positive integer assigned to v_i, representing a radius of an initial sphere at v_i. Let a_i be a sum of all interior angles at v_i. Let K_i be the curvature at v_i, and defined to $2\pi - a_i$ if v_i is interior and $\left(1 - \frac{2}{N}\right)\pi - a_i$ if v_i is on a boundary. Then we define Ricci flow as following:

$$\frac{dr_i}{dt} = -K_i r_i$$

Given that Ricci flow we can build an algorithm which in a limit would produce constant-curvature circle packing of a given triangulated planar graph.

Ricci Flow

Ricci flow (also known as Ricci-Hamilton flow) is a geometrical flow that deforms the metric of a Riemannian manifold and smoothens out irregularities similarly to the diffusion of heat. It was introduced by mathematician, Richard S. Hamilton, and is named after Gregorio Ricci-Curbastro. Ricci flow was heavily used by Perelman in his solution to Poincaré conjecture. Currently the area of work lies in how higher-dimensional Riemannian manifolds evolve under Ricci flow. It is also applicable to constructions of Kähler–Einstein metric.

Algorithm

The following code has been written on Java-based programming language, Processing 3. The whole code consists of two files: *main.pde* and *vertex.pde*.

In *vertex.pde* we initialize Vertex class. Each Vertex holds unique identifications such as radius *r*, curvature *k*, list of neighbour vertices in a counter clockwise order *neighbors*, location vector *location*, and others. Each vertex is initially assigned with a random positive radius, in this case in a (50, 100) range. *Adjust( )* method calculates the curvature of a given vertex and then linearly adjusts the radius.

There are three logical parts in *main.pde*. In the first part, the algorithm initializes the planar graph, types of vertices and edges. In the second part, the algorithm runs Ricci flow for a certain number of iterations (referred in the code as *evolutions*). In the third part, the algorithm reconstructs a visual image of the circle packing by the given radiuses. Lastly, the image is being produced to the screen.

The following is an example of a code for 2-Apollonian triangulated graph.

main.pde

```
ArrayList<Vertex> bound_V;        // set of bound vertices
ArrayList<Vertex> inner_V;        // set of inner vertices
ArrayList<Vertex> V;              // set of all vertices

int[][]  E;                       // 2-D matrix of edge connections
ArrayList<Vertex> line;           // queue of Vertices used in setLocations()

int number_of_bound_v = 3;
int number_of_v = 7;
int evolutions = 0;

void setup() {
  size(800, 800);
  bound_V = new ArrayList<Vertex>();       // First part of the program
  inner_V = new ArrayList<Vertex>();       // Preparing to initialize lists
  V = new ArrayList<Vertex>();
  E = new int[number_of_v][number_of_v];

  E[0] = new int[]{2,6,5,3,0,0,0};   // E[i] is a list of neighbor vertices of v_(i+1)
  E[1] = new int[]{3,4,6,1,0,0,0};   // listed in the clockwise order
  E[2] = new int[]{1,5,4,2,0,0,0};   //
  E[3] = new int[]{2,3,5,7,6,0,0};   // zeroes indicate the end of the list of neighbors
  E[4] = new int[]{3,1,6,7,4,0,0};   //
  E[5] = new int[]{1,2,4,7,5,0,0};   // For example line 25 states that
  E[6] = new int[]{4,5,6,0,0,0,0};   // V7 is connected to V4, V5, and V6 in that order.

  for (int i = 0; i < number_of_v; i++) {                  // We fill in the list V with a parameter
    V.add(new Vertex((i >= number_of_bound_v)));   // i >= number_of_bound_v which indicates
  }                                    // if the vertex is bound or inner

  for (int i = 0; i < number_of_v; i++) {        // We are using matrix E to add information
    int j = 0;                                   // about neighbors to each vertex
    while (E[i][j] != 0) {
      V.get(i).addNeighbor(V.get(E[i][j] - 1));
      j++;
```

```
    }
  }

  for (int i = 0; i < number_of_bound_v; i++) {                    // initializing bound_V
    bound_V.add(V.get(i));
  }
  for (int i = 0; i < number_of_v - number_of_bound_v; i++) {  // initializing inner_V
    inner_V.add(V.get(number_of_bound_v + i));
  }
}

void draw() {
  evolutions++;
  if (evolutions < 300) {     // Second part of the program
    background(0);            // First 300 itterations
    for (Vertex v : V) {      // Going through each vertex
      v.adjust();             // Running Ricci flow and linearly adjusting radius
    }
  } else if (evolutions == 300) {      // Third part of the program
    line = new ArrayList<Vertex>();
    line.add(inner_V.get(0));

    inner_V.get(0).setLocation(new PVector(0,0));       // Fixing location of the first inner vertex
    inner_V.get(0).neighbors.get(0).setLocation(        // and its first neighbor as a point of reference
      new PVector(0, inner_V.get(0).r + inner_V.get(0).neighbors.get(0).r));
    setLocations();                                     // Setting locations of the rest of vertices
  } else {
    background(0);                    // Last part of the program
    translate(width/2, height/2);     // Centering the set of coordinates in the middle of the screen
    for (Vertex each : V) {           // Drawing vertices
      each.draw();
    }
  }
}

void setLocations() {                  // Method setLocations() is similar in its structure to
  PVector local;                       // Breadth-first search except the queue is only used
  ArrayList<Vertex> neighbors;         // for inner boundaries
  int index;

  while (line.size() != 0) {           // While ther are still vertices in the queue
    Vertex v = line.get(0);            // Take the first one and label it v
    line.remove(0);                    // Remove it from the list
    index = v.findIndexOfFixedNeighbor();  // Find a neighbor with fixed coordinates
    neighbors = v.getNeighbors();              // Get neighbor list of the vertex v

    local = new PVector(neighbors.get(index).getLocation().x - v.getLocation().x, neighbors.get(index).getLocation().y -
v.getLocation().y);
    local.normalize();             // Finds and normalizes the vector pointing to a fixed neighbor, which will be used
                                   // as a frame of reference for the rest of the neigbors
    for (int i = index; i < index + neighbors.size(); i++) {      // Running through the list of neighbors starting with the
fixed one
      Vertex u = neighbors.get(i % neighbors.size());                // Getting the indices of two neighbors u and w
      Vertex w = neighbors.get((i + 1) % neighbors.size());          // located next to each other
      local.rotate(v.innerAngle(u, w));                              // rotating local vector on the angle uvw
      if (!w.isFixed()) {                                            // If w is not yet assigned a set of coordinates
        local.mult(v.r + w.r);                                       // scale the vector so that it would point at the center of w
        w.setLocation(new PVector(v.getLocation().x + local.x, v.getLocation().y + local.y)); // assign this location to w
        if (w.isInner()) {          // if w is an inner vertex
```

```
        line.add(w);              // add w to the end of the queue
       }                          // repeat the process
       local.normalize();
     }
    }
  }
}
```

vertex.pde

```
class Vertex {
  float r;
  float K;

  ArrayList<Vertex> neighbors;

  boolean inner = false;
  boolean fixed = false;
  PVector location;

  Vertex(boolean a) {
    r = random(50, 100);
    neighbors = new ArrayList<Vertex>();
    inner = a;
  }

  void addNeighbor(Vertex u) {
    neighbors.add(u);
  }

  ArrayList<Vertex> getNeighbors() { return neighbors; }
  float getR() { return r; }

  float innerAngle( Vertex u, Vertex w) {
    float r1 = u.getR();
    float r2 = w.getR();
    return 2 * asin(sqrt(r1 * r2 / (r + r1) / (r + r2)));  // inner angle uvw
  }

  void setToInner() { inner = true; }
  boolean isInner() { return inner; }
  void setToFixed() { fixed = true; }
  boolean isFixed() { return fixed; }

  void adjust() {
    float total_inner_angle = 0;
    if (inner) {
      neighbors.add(neighbors.get(0)); // adds the first vertex to the end to simplify the code of the following loop
      for (int i = 0; i < neighbors.size() - 1 ; i++) {
        total_inner_angle += innerAngle(neighbors.get(i), neighbors.get(i+1));
      }
      neighbors.remove(neighbors.size() - 1); // removes the first vertex from the end
      K = 2 * PI - total_inner_angle;              // calculates the curvature

    } else {
      for (int i = 0; i < neighbors.size() - 1 ; i++) {
        total_inner_angle += innerAngle(neighbors.get(i), neighbors.get(i+1));
      }
```

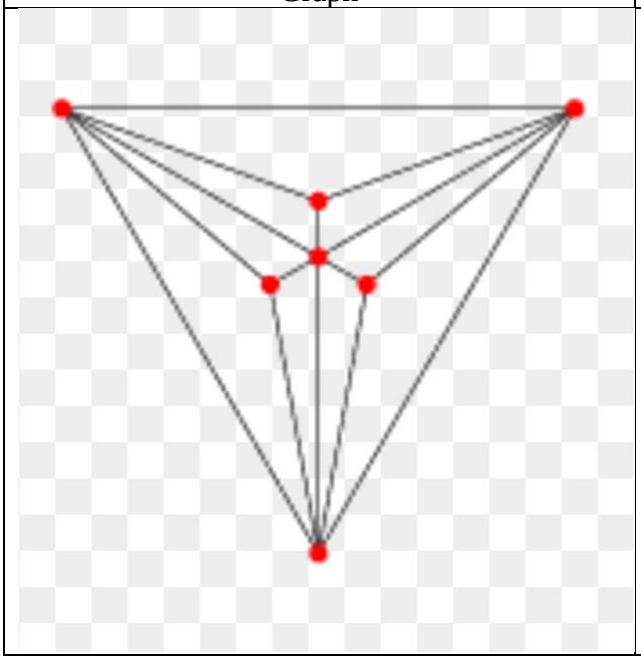
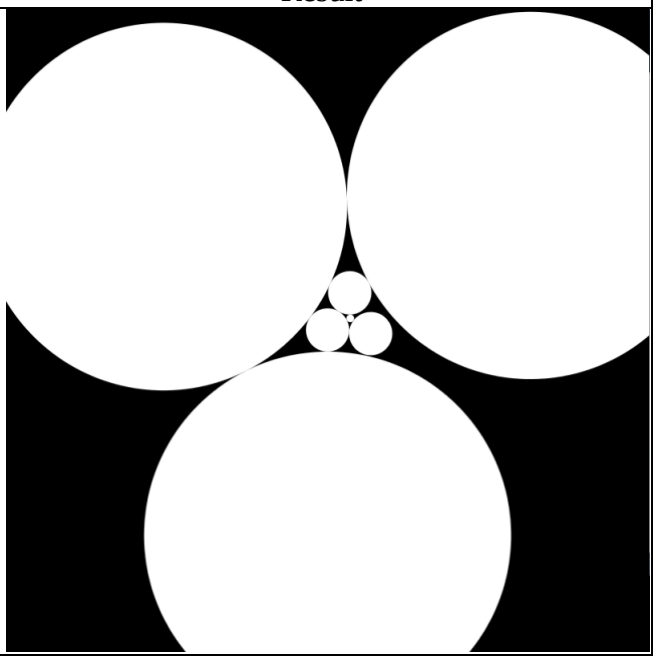
```
    K = (number_of_bound_v - 2) * PI / number_of_bound_v - total_inner_angle; // calculates the curvature for the
bound case
   }
   float dr = - round(1000 * K * r) / 1000;        // approximates and rounds dr
   r = r + dr / 100;                               // lineraly approximates r
  }

  void setLocation( PVector v ) {
   location = v;              // whenever the location is set
   fixed = true;              // the vertex becomes fixed
  }

  PVector getLocation( ) { return location; }

  int findIndexOfFixedNeighbor() {                 // Finds a fixed neighbor which would
   for (int i = 0; i < neighbors.size(); i++)  {   // serve a point of reference for the rest
    if (neighbors.get(i).isFixed()) {
     return i;
    }
   }
   println("Undefined case");
   return -1;
  }

  void draw() {                  // Displays the circle on the screen
   fill(255,255,255);
   noStroke();
   ellipse(location.x, location.y, r*2, r*2);
  }
}
```

Graph	Result
	

References

Chow, Bennett, and Feng Luo. “Combinatorial Ricci Flows on Surfaces.” Journal of Differential Geometry, Lehigh University, projecteuclid.org/euclid.jdg/1080835659.

Colling, Charles R, and Kenneth Stephenson. A circle packing algorithm. Department of Mathematics, University of Tennessee, 18 Feb. 2002.

Thurston, William (1978–1981), The geometry and topology of 3-manifolds, Princeton lecture notes.