Framework: Cone Tree V2

State: August 02th , 2019

Note:
This writing explains how the current Cone Tree visualization is implemented.
It is the second version of it and differs from the initial publication in the point,
that it tries to place the nodes so that we get a minimal tree.

1. Sources

The following material was used to acquire the knowledge and get the equations.

[1] Sébastien Grivet, David Auber, Jean-Philippe Domenger, Guy Melançon. ***Bubble Tree Drawing Algorithm***. ICCVG: International Conference on Computer Vision Graphics, Sep 2004, Warsaw, Poland. pp.633-641, 10.1007/1-4020-4179-9_91. lirmm-00108872

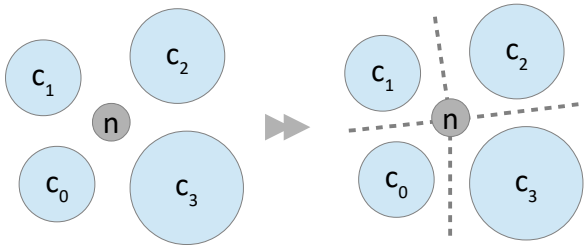
[2] Welzl, Emo. (1997). ***Smallest Enclosing Disks (balls and Ellipsoids)***. Proceedings of New Results and New Trends in Computer Science. 555. 10.1007/BFb0038202.

[3] ***Perpendicular Bisector Calculator.*** http://www.meracalculator.com/graphic/perpendicularbisector.php

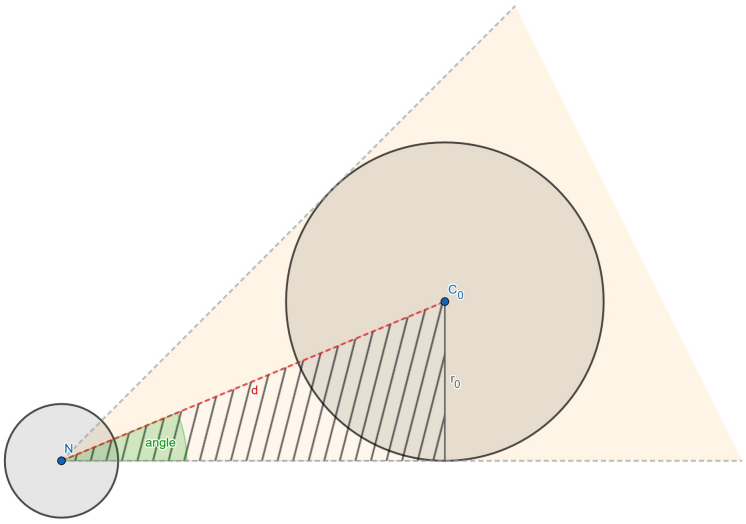
1. Bubble Tree Drawing Algorithm

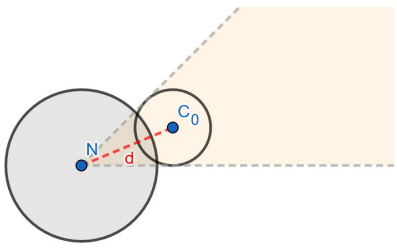
The so called *Bubble Tree Layout* is used to position a group of nodes properly. Please refer to Sébastien Grivet et al. [1] for a detailed explanation, as the following sections will only cover the most important steps briefly.
Our algorithm consists of mainly two steps. One to calculate the relative positions of nodes and a second one to calculate the final position of the *GameObject* they represent. As the algorithm traverses the directory structure recursively for the first time, you will either stumble across a leaf node or a group of nodes. For each leaf node, we can easily tell that its relative position is in the center (0, 0) and we can simply assign a radius to it. The tricky part comes up as soon as we process a group of nodes.

How to pack our set of nodes C = {c_0, c_1, c_2, c_3} around a center ***n***
inside a circle, so that we do not waste much space,
can have space between them and do not have any overlapping nodes?



For a basic positioning around the center node *n*, [1] suggests to use *angular sectors* that are proportional to the radii r_i .
So if we imagine a circle surrounding everything and having its center at the position of the node *n*, we can subdivide it into single parts (the angular sectors). This is shown by the figure above. So to calculate the angle for each node of C, we can sum their radii up, see how much a nodes radius takes from it and multiply it by 2π (or 360 to get it in degree). We can now check how far away the node must be placed from the center node *n*. To calculate the distance ***d***, we use the triangle that is formed by the angular sector of the node. This is shown by the figure below.





Problem:
Large angle or small radius
results in overlap!

Based on the triangle that is marked in the left figure above, we want to calculate *d* and have given the angle and $\mathbf{r_0}$.
Using the trigonometric formula **sin = opp / hyp** , we can get the following equation to solve for **hyp** (which is *d* in our case).

$$\delta_i = \frac{r_i}{\sin(\theta_i/2)}$$

As shown by the previous figure on the right, there is a problem when the radius of the node $\mathbf{c_i}$ is smaller than the radius of *n*. This problem also occurs if the angle is too large for the size of the node (e.g. imagine two small nodes – both will receive an angle of 180 degree and thus, this problem would always occurs in this scenario). To address this problem, we can simply ensure that the distance can not be less than the sum of the radius of *n* and the radius of c_i and we update our equation as follows.

$$\delta_i = \max\left(\text{radius}(n) + r_i, \frac{r_i}{\sin(\theta_i/2)}\right)$$

The position $\mathbf{pos}_i$ of the node can be determined easily by using distance $\boldsymbol{\delta}_i$, cosine and sine to place it on a circle and taking the sum of the angles of previous nodes $\mathbf{ap}_i$ into account. Here are the according equations for it.

$$\text{pos}_i = \begin{cases} x_i = \delta_i \cos(\text{ap}_i + \theta_i/2) \\ y_i = \delta_i \sin(\text{ap}_i + \theta_i/2) \end{cases}, \quad \text{ap}_i = \sum_{j=0}^{i-1} \theta_j$$

As there will be unused space, [1] suggest a way to optimize the angular sectors. We won't cover this approach here, as it is just briefly described and does not seem to offer much improvements for the final layout. So we will keep it simple.

Having this basic tree layout calculated for a group of nodes now, the next step consists of calculating their smallest enclosing circle ***SEC***. It is required to get the "center of mass" of our node group as well as the radius. The group radius will be used by the parent node as soon as the recursion "travels up" in the hierarchy. An approach proposed by Welzl [2] is used to find the *SEC.*

2. Smallest Enclosing Circle Algorithm

As previously mentioned, this algorithm is proposed by Welzl [2]. We need the center of mass of our group of nodes to determine how to position them relative to their parent node. Also, the radius of the smallest enclosing disk (***SED***) is required by the parent node to create the layout for its layer in the tree accordingly. The recursive algorithm runs in expected linear time and is based on Seidel's Linear Programming algorithm. It basically breaks down the problem to a set of at most 3 points for which a so called "trivial" algorithm is used to calculate the circumscribed circle (or circumcircle). As the following section covers only the most important points, required for our implementation, please refer to the original paper for detailed information and proofs.
("Disk" is used instead of "circle" for easier linking of relevant information when referring to the original publication.)

The basic idea.

While computing the final result in an incremental way, we maintain the smallest enclosing disk of the points considered so far. Consider that we have already a *SED* called ***D***. We randomly select a point $\boldsymbol{p}_i$ from our set of points ***P***. In case that *D* already contains p_i , we do not need to consider this point anymore and can continue with the next one that we also select randomly.
In case that *D* does not contain p_i , we know that this point must lie on the boundary of another disk ***D'***. As *D'* will contain our selected point p_i , as well as the points that *D* already covered, it maintains the smallest enclosing disk that we have so far.
By fixing a point to be on the boundary of the disk, the problem becomes easier. Welzl [2] also shows, that there are at most three points of the boundary to form the *SED*. Therefore, a trivial algorithm creates the *SED* from one, two or three points.

The algorithm in pseudocode.

Input: Finite sets *P* and *R* (hold points in the plane)
Output: *D* = *SED* (enclosing *P* and has *R* on boundaries)

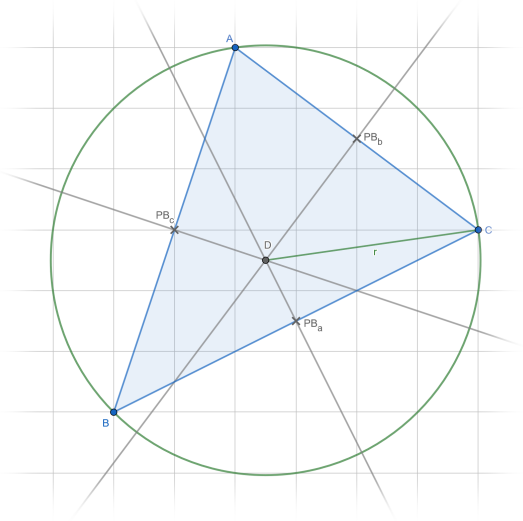
```
1  function welzl(P, R):
2     if P is empty or |R| = 3 then return trivial(R);
3     choose random p ∈ P;
4     D := welzl(P – {p}, R);
5     if p is in D then return D;
6     return welzl(P – {p}, R ∪ {p});
```

Although this algorithm looks relatively small and easy, the recursive steps are hard to imagine. It also does not explain what exactly is done in the "trivial" part of the algorithm, using the set *R* to create the actual *SED*. This is why we will cover that step as well in the following section, by having a look at circumscribed circles of triangles and how to calculate them.

3. Circumscribed Circle of a Triangle

Covered by the previous section, we now know that the set *R* holds all the points on the boundary of the enclosing disk and that it has at most three points. These points are what we use to calculate the circumscribed circle of the triangle they form together.

The figure on the right shows an example for *R* = {A, B, C} .
What we have to do now, is to calculate the perpendicular bisectors of the triangles edges (lines between the points). These are named ***PB*** in the figure and shown by the gray lines. As one can see, all three possible perpendicular bisectors meet at one point ***D*** .
This point is the *circumcenter* of the circumcircle and represents the position that we need to find. Two of the three perpendicular bisectors are enough to determine *D* . After we have done so, we can get the distance between that point and any other point of our triangle. This distance represents the *radius* of our circumcircle. The figure shows it by drawing a line between *D* and C, labeled with the letter **r** .



There exist several ways to accomplish the aforementioned task.
In the following section, we will mainly use just one of them.
Anyway, some other possibilities will be mentioned in it.
So feel free to use any other method that may fit your needs better.

First of all, we need to find at least two perpendicular bisectors *BP* and will briefly mention two possible ways.
Which technique you choose at the end is totally up to you. ***We will now refer to the circumcenter when using the letter D***.

Finding the perpendicular bisectors. Approach 1.

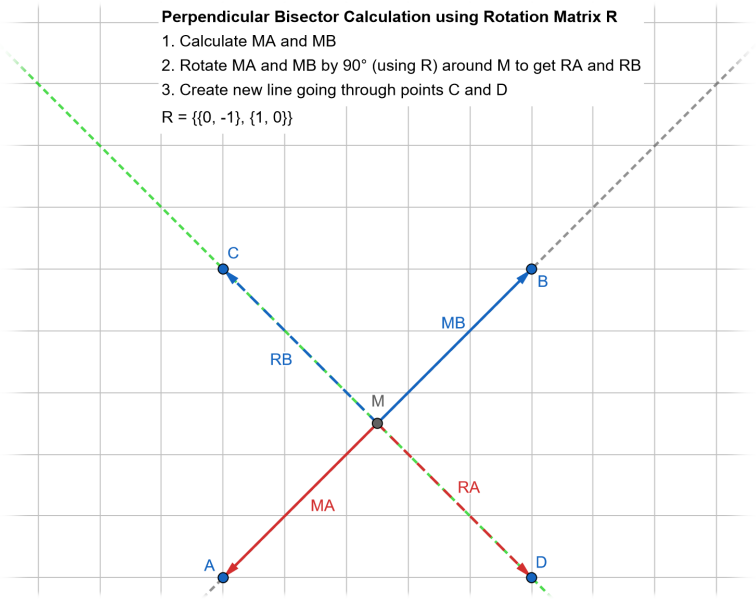
As our first approach, we will do so by using the following steps, proposed by [3]:

1. Determine the midpoint ***m*** between two points (e.g. A and B) of the triangle
2. Calculate the slope ***s*** of the two triangle points and use it to determine the slope of the perpendicular bisector ***s'***
3. Use the **point-slope form**, because we have ***m*** and ***s'*** given, to retrieve the final equation of the perpendicular bisector
4. Calculate the intersection of two perpendicular bisectors, e.g. by using *homogeneous coordinates* to get ***D***

In case you do use the previous steps, make sure to check step 1 for the case, where the x- or y-values of A and B are equal. The reason for this is, that calculating the slopes would then result in a division by zero that you need to handle manually.

Approach 2.

Of course, the previous can be achieved as well by using a rotation-matrix to rotate the edges of the triangle 90° around the midpoints. You could do the calculation using vectors and matrices (see https://www.xarg.org/puzzle/codesignal/circumcircle/). The figure below shows the basic idea behind this approach. The intersection calculation to get ***D*** can be done using the vectors or converting them into a general line equation form and using *homogeneous coordinates* again. The former is more difficult but also mentioned as part of the referenced article: https://www.xarg.org/2016/10/calculate-the-intersection-point-of-two-lines/.
Using this approach is especially useful if you can use pre-implemented vectors, matrices and e.g. intersection calculations.



As the intersection calculation of approach 2 seemed to be very complex, I decided to use only its way of determining the perpendicular bisectors (using the rotation-matrix). This way, we would also not have any trouble with the slopes in case the x- or y-values of the points are the same. Furthermore, matrix-vector multiplication is very efficient nowadays. Anyway, for the intersection part I did not use the suggestion of the second approach as it seemed very complex and cluttered. Therefore, I would create the general form of both lines instead.

$$ax + by + c = 0$$

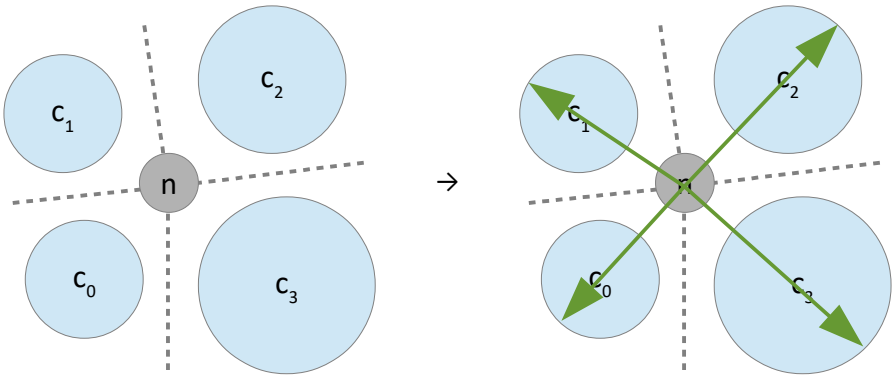
Using this form, we can simply turn each line in a three-dimensional vector and apply the *cross-product* to them. This is why the concept of *homogeneous coordinates* in terms of line-line intersection was mentioned earlier. So if we apply the cross-product in homogeneous coordinates, we will get a 2D point that represents the intersection of our two lines. We can then "convert" the point back to "correct" coordinates, ***dividing*** the x- and y-value by the z-value (see *perspective division*) and retrieve our ***D***.

$$H = \begin{pmatrix} a_1 \\ b_1 \\ c_1 \end{pmatrix} \times \begin{pmatrix} a_2 \\ b_2 \\ c_2 \end{pmatrix} = \begin{pmatrix} b_1 * c_2 - b_2 * c_1 \\ c_1 * a_2 - c_2 * a_1 \\ a_1 * b_2 - a_2 * b_1 \end{pmatrix} \quad \rightarrow \quad \boldsymbol{D} = \begin{pmatrix} H_x \ / \ H_z \\ H_y \ / \ H_z \end{pmatrix}$$

As already mentioned, we can then create our smallest enclosing disk/circle using the circumcenter ***D*** and the radius ***r*** , which we can simply get by calculating the distance between ***D*** and any other point of our triangle.

4. Wait, isn't there something missing?

You may have asked yourself at this point: "Why are we always using ***POINTS*** instead of ***CIRCLES***?", as our initial goal was to create the cone tree layout where each node has a specific radius that also depends on its child-nodes. This is a very important question. To answer it, let's have a look at the bubble-tree layout algorithm again. Specifically, we go back to the point at which we had our nodes positioned relative to their parent-node, using the angular sectors. Here is one more figure for this purpose.



As shown by the figure above, one can see that we are able to use the relative position that we already calculated to answer this question. Following the direction from the center of node *n* to the center of the other nodes c_i , we can easily get the position of the point we need to use by extending the ray by the radius. Once we have the points, we can use the algorithm to find the SED.

Another question you may have asked is: "What are we going to do when we have ***less than 3 points*** in ***R*** and ***P*** is empty?". This question basically refers to the algorithm line 2. The answer is simple, just draw a circle using just two- or a single point. In case a single point is left, we simply use its radius as the radius of our smallest enclosing disk/circle and that's it.

For more details, optimizations and improvements to the layout, please refer to the referenced sources.