Circle Rendering

Idea: Pigeonhole principle to find nearest neighbour points :

Principle:

“If n items are put in m boxes where n > m, then one box must contain more than one item”.

In circle rendering algorithm, the first objective is to find the nearest grid points for a given point of a circle. Consider the circle shown in Figure 1. Suppose the length of the square region of search area is l_1 and the length of the square formed by using four nearest grid points is l_2. Now, if $l_1 == l_2$ then, according to pigeonhole principle, the maximum number of grid points inside the search area will always < = 4 (constant). In the algorithm, the circle points are calculated by varying angle in the first quadrant starting with 0 and interval of 10. The other points of a circle points are calculated by changing the sign of sine and cosine functions and using symmetric distance criteria.

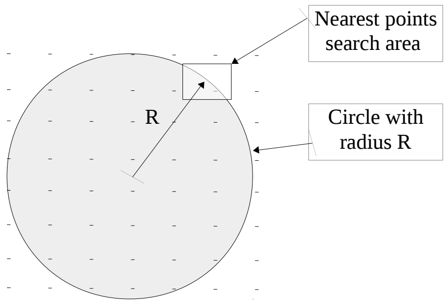

Figure 1. Sample example of circle with radius R.

Algorithms :

Algorithm : Circle_rendering(image)

Initialization:

```
/*  Initialize image with RGB(0,0,0)
    divide image by grid of equal size
     center_xy = get_xy on mouse_press */
if (mouse_event == true) do:
     while(mouse_drag == true) do:
          current_xy = get_xy on mouse_drag
          curr_radius =  sqrt ( (current_xy-center_xy) ^ 2)
          inner_radius= outer_radius=curr_radius;
          Point P;
          for theta in 0-90 do:
                P.x= r * cos(theta);
                P.y= r * sin(theta);
             /* call  the following function */
                calculate_nearest_neighbours(P);
                update_inner_outer_radius(P);
                theta = theta + 10;
                draw_circle(xy_center,radius);
                draw_innercircle(xy_center,inner_radius)
                draw_outercircle(xy_center,outer_radius)
                update Qlabel with image
          if(mouse_release==true) do:
               delete circles;
                 show_nearest neighbour points
```

```
Algorithm_1_1 : calculate_nearest_neighbours ( Point )

    xloc[ ]= locations of first row grid points
    yloc[ ]= locations of first column grid points
    for i in 0 to grid_size-1 do:
        if (Point.x >= xloc[i] && Point.x <= xloc[i+1])
             found_x= xloc[i];
        if (Point.y >= yloc[i] && Point.y <= yloc[i+1])
            found_y=yloc[i];
      /* calculate four grid points starting with  p (found_x,found_y)  and store in points[ ] */
     for i in 0 to 4 do:
          if (search_area contains points[i])
                 store point[i] as candidate nearest point
                 update_inner_outer_radius(Point)
```

```
Algorithm Condition : update_inner_outer_radius(Point)

   distance   = /*euclidean distance from Point to the center of curr_circle */
  if ( distance < radius && distance < inner_radius )
          inner_radius=dist;
  if ( distance  > radius && distance > outer_radius )
          outer_radius=dist;
```

Running Time Analysis :

- In the circle rendering algorithm, the for loop executes 10 times with increment of theta by 10(constant). Suppose this constant is c_1.

- Now, for each point calculated in the circle rendering algorithm, one need to find the nearest points of it. Here, inside the nearest_neighbour algorithm, the first for loop executes *grid_size* times. Note that, in the given problem, the *grid_size* = 20. One need to consider the *grid_size* = n for running time analysis.
- The other *for* loop of the same function executes 4 times(constant) and suppose this constant is c_2. The update condition of inner and outer radius takes constant time O(1).

- Thus the running time of an algorithm is $O(c_1*c_2*n)= O(n)$ where n=*grid_size*

Implimentation Detail :

- *This algorithm is developed in C++ with QT 5.7 framework in Ubantu 16.04 LTS.*
- *To Display and update the image Qlabel is used with custom propery.*

Results :

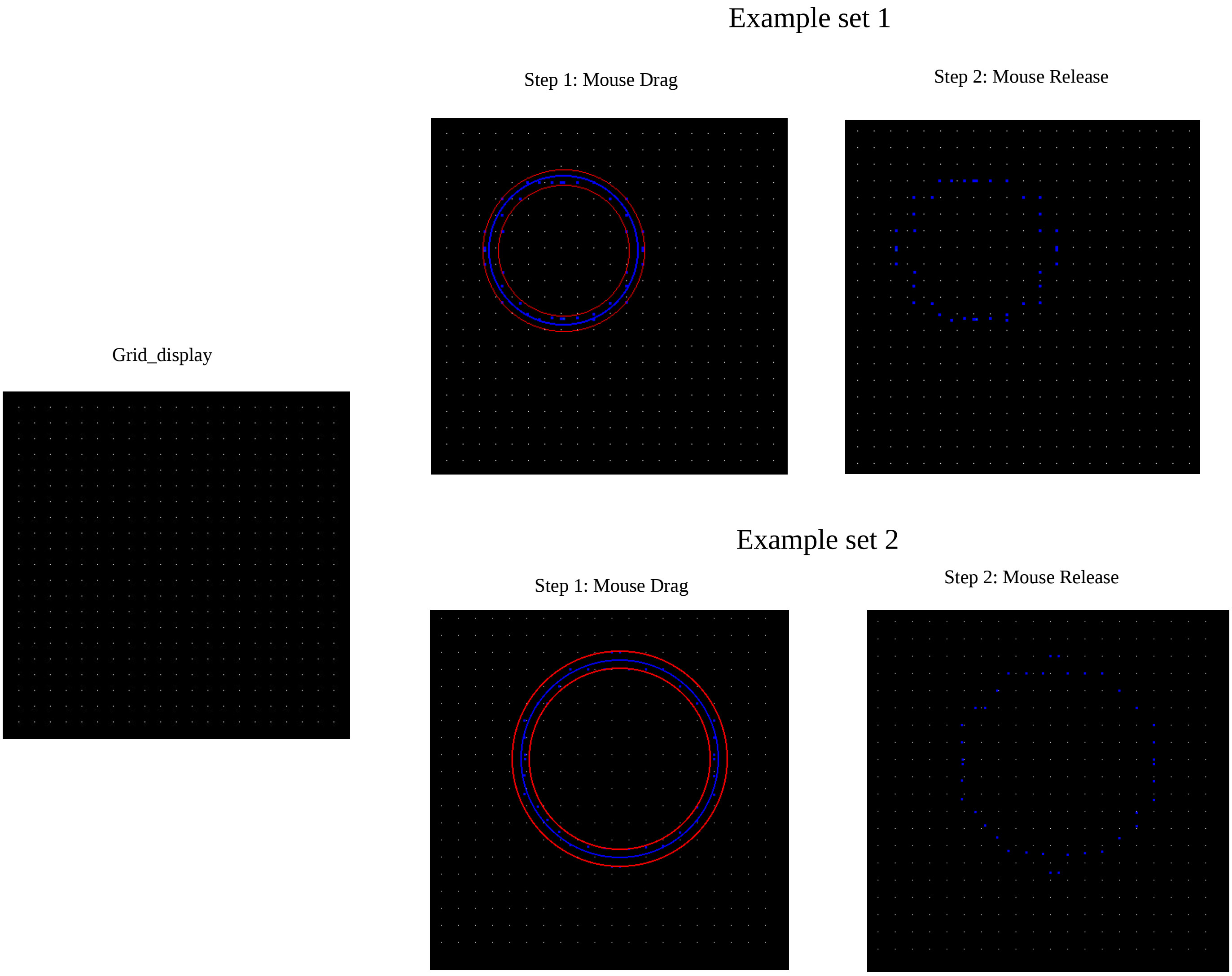

Figure 2. Results of circle randering algorithm

Conclusion:

Using Pigeonhole principle the given problem can be solved in linear time. There are certain methods to improve the results considering the angle range from [0-360]. One can also solve the problem in logarithmic time using binary search technique in the first step of Algorithm 1_1.