JGDS2 - Java GDS Library

Rob rob4542@gmail.com

John Treichler treichler@cnf.cornell.edu

Cornell University, Ithaca, NY

The JGDS2 stream Java based library was written by John Treichler to facilitate scripting of shapes and features directly to GDSII. The library provides access to fully control vertices when dealing with curved geometries and is ideal for parametrizing CAD patterns. The library also contains a `Parts Library` composed various static methods that produce complex shapes. This eliminates the need to rewrite methods such as circles, ellipses, tori, etc. We encourage you to submit requests to John or Rob for additional complex shapes. We will continuously add shapes to further extend the `Parts Library` resources. Because the library was developed in Java, access to all the standard Java classes for generating graphics (paths, bezier curves, affine transforms, etc) as well as fonts (FontMetrics) are available. Consequently, custom patterns with arbitrary complexity along with incorporation of labels within CAD could be generated with ease. Contents of this document will describe, in great detail, various examples provided within the distributed Java Netbeans project.

The JGDS2 stream library is distributed in hope that it will be useful, but without any warranty, without even an implied warranty for any particular

purpose. Free distribution of the JGDS2 stream library is allowed. Currently, will not provide source code for the JGDS2 stream library (at some point in time the Java source code will be posted as open source).

If you have comments, encounter bugs or need help with using the JGDS2 library, email John or Rob at the above email address.

Contents

1 Overview **3**
- 1.1 Background - JGDS2 stream library . . . 3
- 1.2 Netbeans JGDStutorial project . . . 3

2 JGDS2 Example Template **6**

3 Simple shapes - Manhattan Geometry **7**
- 3.1 Simple Rectangle . . . 7
- 3.2 Arrays of Rectangles . . . 7
- 3.3 Diamond - Rotated Instance of a Square . . . 8

4 Curved Structures - Paths and GAreas **9**
- 4.1 Circles and Ellipses . . . 9
- 4.2 Torus - Subtraction of two Circles . . . 10
- 4.3 Torus - Two Path Loops . . . 11
- 4.4 Bezier Curves . . . 11

5 Text - Labels **14**

6 Complex Shapes **16**
- 6.1 Fractals - Sierpinski Triangle . . . 16
- 6.2 Fractals - Curved Tree . . . 17
- 6.3 Fractals - Another Tree . . . 20
- 6.4 Fractals - Yet Another Tree . . . 21

7 JGDS2 - Details **23**
- 7.1 Arrays - Nonorthogonal . . . 23
- 7.2 setRenderReso . . . 25
- 7.3 Circumvent Point Limits with GDS files . . . 27
- 7.4 Parts Lib . . . 27
- 7.5 setReso . . . 28

1 Overview

1.1 Background - JGDS2 stream library

Most CAD tools fall short when exporting curved features at the sub-micron dimension. It is virtually impossible to control vertices of individual features, drawn with the provided primitive tools, within a particular CAD file. Scripting with within these tools alleviates some of these strains, however, even for an experienced programmer, in certain instances snapping at the wrong vertex can occur during the stream out process. Table 1 shows the number of vertices generated using LEdit's primitive circle-drawing tool. In this case, reducing the snapping grid could diminish the number of vertices, however, if the pattern file contains small (sub-μm) features in conjunction with large ones, the large will appear fine whereas small will appear disfigured. Large number of vertices will affect the total write time with lithographic pattern generation tools. For instance, when utilizing the Heidelberg laser writer, even if the instanced file is small the conversion software has to flatten the data file. The process of flattening takes time and depends on the size of the flattened file. A CAD file with a cell instance of 1000 x 1000 array of circles could be a few kB, however, flattening this file with individual circular elements comprised of large number of vertices could result in a file size easily exceeding a GB (perhaps several GBs). Consequently, the laser writer pattern generator could either take hours to initiate conversion or crash. JGDS2 stream library allows ultimate control of vertices of generated shapes. With JGDS2, all Java graphics libraries are available, rendering mathematically defined shapes, text, affine transforms and a myriad of other functionality, easy to implement within patterns. Figure 1 illustrates interaction of the internal `GDS2` objects.

Table 1: *Vertices generated using the primitive circle element within LEdit 15.*

Circle element	
Radius (μm)	Vertices
10	592
100	1892
500	4100
1000	5884
10000	18800

1.2 Netbeans JGDStutorial project

The distribution tar file contains an instructional tutorial pdf document (this file), example NetBeans project (created using NetBeans 7.2) within a directory `JGDStutorial`,

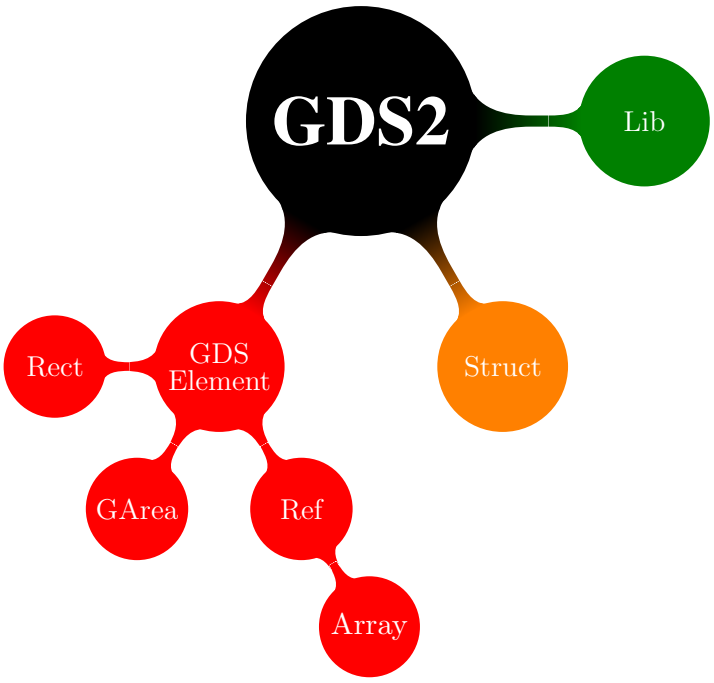

Figure 1: *GDS2 library*

`javadoc` directory containing all the auto generated java documentation via NetBeans (open the index.html file within a browser) and the JGDS2 stream library, stream2.07.jar. We recommend installing the latest NetBeans, then opening the enclosed tutorial project. Initially the project is not linked to the JGDS stream library and upon opening the project, a `Reference Problems` dialog box will pop-up. Click `Close` within the box. The project as well as various sections of the source folder will appear red, indicating errors. To resolve this dilemma, first, right click on the `JGDStutorial` library name within the projects tab, then choose `Properties`, within the `Project Properties - JGDStutorial` window choose the `Libraries` category. Within the `Compile` tab click the `Add JAR/Folder` button, then navigate to the directory where the stream jar file resides, click on the file and choose `Open`. The project is now linked to the JGDS2 stream library. Consequently errors stemming from the initially non-linked library will disappear. When creating projects in NetBeans, use the above method to link to the JGDS2 stream library.

The project `JGDStutorial` contains a package `jgdstutorial` with a variety of `.java` example files illustrating various ways of casting primitive shapes, mathematically defined paths, bezier curves, text labels and other objects directly to `GDSII` files. Following sections will explain details of the enclosed tutorial examples.

`NOTE:` methods within each of the example classes are `static`. This implies when

you are writing code you can use these methods without creating a new object. See the following examples:

```
Struct myCircleStruct = Ex04CircleEllipse.CircleEllipse(0.0, 0.0, 10.0, 10.0,
64, 44);
```

will create a 64 sided circle centered at $(0, 0)$ with a 10μm radius in a GDS layer number 44.

```
Struct myEllipseStruct = Ex04CircleEllipse.CircleEllipse(0.0, 0.0, 10.0, 4.0,
64, 44);
```

will create a 64 sided ellipse centered at $(0, 0)$ with a 10μm radius along the x-direction, a 4μm radius along the x-direction, in a GDS layer number 44.

```
Struct myTorusStruct = Ex06Torus.CircleEllipse(0.0, 0.0, 8.0, 10.0, 360.0,
64, 4);
```

will create a 64 sided torus centered at $(0, 0)$ with a 8μm inner radius , a 10μm outer radius, in a GDS layer number 4.

2 JGDS2 Example Template

Each example contains a main method (`public static void main(String[] args)`) within which `try` and `catch` routines define and upon completion export data to a `GDSII` file. This code is shown in Listing 1, an excerpt of lines 21 through 30 from the `Ex00Template.java` file.

Listing 1: Excerpt from Ex00Template.java

```
21          FileOutputStream fileOUT;
22          File f = new File("Ex00Template.gds");        // change gds Filename
23          fileOUT = new FileOutputStream(f);
24          DataOutputStream dO = new DataOutputStream(fileOUT);
25          GDSWriter g = new GDSWriter(dO);
26          Lib lib = new Lib();
27
28          // Insert Code Here
29
30          lib.GDSOut(g);
```

Lines $21-26$ set up the GDS file and library, with line 30 representing the writing and closing out of the GDS file. Objects that are written to the GDS file will be contained between lines 26 and 30, as illustrated by the comment in line 28.

3 Simple shapes - Manhattan Geometry

3.1 Simple Rectangle

`Ex01Rectangle.java` illustrates creation of `Struct` objects and casting of rectangles into a `Rect` object. In order to create a rectangle, first, a cell called `top` is created using

```
34        Struct top = new Struct("top");
```

then using a GDS layer 4, a 10μm $\times$ 20μm rectangle is placed into the `top` cell. Rectangle is defined by a lower left corner $(0.0, 0.0)$ and an upper right corner $(10.0, 20.0)$

```
37        top.add(new Rect(0.0, 0.0, 10.0, 20.0, 4));
```

`Rect` is defined by `Rect(`x_{ll}`,` y_{ll}`,` x_{ur}`,` y_{ur}`, GDS layer number)`, where lower left (ll) and upper right (ur) coordinates are `double` values and the `GDS layer number` is an `int` value spanning $0 - 255$. Structure `top` containing the rectangle is added to the library using line 44

```
44        lib.add(new Ref(top, 0, 0));
```

`Ref` is defined by `Ref(Struct s, double` x`, double` y`)`.

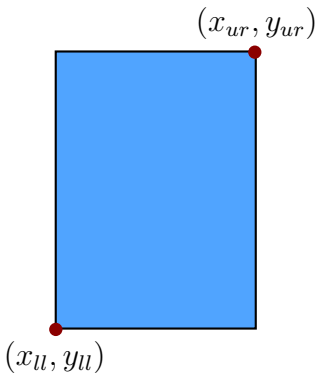

Figure 2: *Example of a rectangle extending from* (x_{ll}, y_{ll}) *to* (x_{ur}, y_{ur}).

3.2 Arrays of Rectangles

`Ex02RectangleArray.java` illustrates a procedure for creating a rectangle in a cell, then instancing an array of those cells within a `top` cell. First, as shown in line 36, structure called `top` is created

```
36        Struct top = new Struct("top");
```

then a structure `oneRectangle` with a string value of `singleRectangle` is created

```
37        Struct oneRectangle = new Struct("singleRectangle");
```

As defined in the previous section, a 10μm $\times$ 20μm rectangle is then placed into `oneRectangle`

```
39        oneRectangle.add(new Rect(0.0, 0.0, 10.0, 20.0, 4));
```

A 4×5 array of rectangles is created by instancing the `oneRectangle` structure within the `top` structure. The array starts at $(2, 2)$ and is spaced by 25.0μm in x and 20.0μm in y directions.

```
42        top.add(new Array(oneRectangle, 2.0, 2.0, 4, 5, (4.0 * 20.0), (5.0
              * 25.0)));
```

Here, `Array` is defined as `Array(Struct s, double x, double y, int Columns, int Rows, double (Columns * DX), double (Rows * DY))`, where `s` is the instanced structure, x and y are the starting coordinates of the array, `Columns` and `Rows` represent the number of columns and rows respectively, `DX` and `DY` are the spacing between array elements along the x and y directions respectively.

3.3 Diamond - Rotated Instance of a Square

`Ex03Diamond.java` example shows a procedure for rotating structure instances. Here a square is drawn and then instanced within the `top` structure at an angle of 45°. First the `top` and `square` structures are created in lines 34 and 35. Then in line 36, using a GDS layer number 4, a 10μm $\times$ 10μm square is then placed into `square` structure.

```
34        Struct top = new Struct("top");
35        Struct square = new Struct("10umSquare");
36        square.add(new Rect(0.0, 0.0, 10.0, 10.0, 4));
```

Line 38 shows the `square` structure instanced at a 45° angle within the `top` structure.

```
38        top.add(new Ref(square, 0.0, 0.0, 0, 45.0));
```

Constructors for `Ref` are the following:

```
Ref(Struct s, double x, double y)
Ref(Struct s, double x, double y, int mirror)
Ref(Struct s, double x, double y, int mirror, double angle)
Ref(Struct s, double x, double y, int mirror, double mag, double angle)
```

To use mirroring, first, the class must implement an interface `Const` (as shown in the `Fractals - Curved Tree` example), then set `int mirror` to `MIRROR`.

4 Curved Structures - Paths and GAreas

4.1 Circles and Ellipses

`Ex04CircleEllipse.java` example demonstrates paths using `Path2D.Double`. Also, this example has 2 methods, one returns a `Struct` and the other returns `GArea` element. `CircleEllipse` method returns a `Struct` and is defined in line 67 as

```
67  static Struct CircleEllipse(double x, double y, double radiusX, double
        radiusY, int numSides, int layer) {
```

where the shape is centered around (x, y), with two radii `radiusX` and `radiusY` (for a circle `radiusX` = `radiusY`), `numSides` represents the number of sides for the generated shapes and `layer` is the GDS layer number. Similarly, `CircleEllipseArea` method returns a `GArea` and is defined in line 84 as

```
84  static GArea CircleEllipseArea(double x, double y, double radiusX, double
        radiusY, int numSides, int layer) {
```

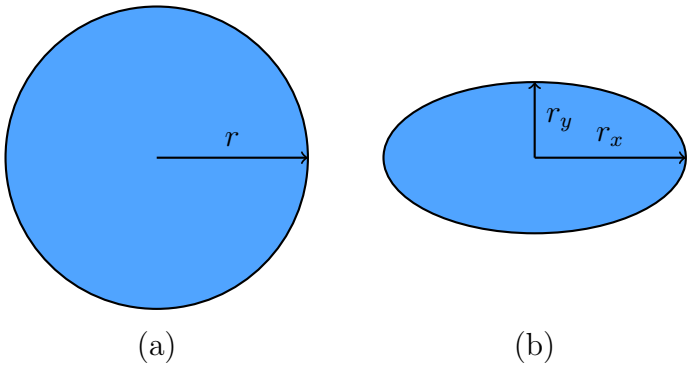

Figure 3: *Illustration of (a) a circle with radius $r = r_x = r_y$ and (b) an ellipse with radii r_x and r_y.*

Within Listing 2, lines $68 - 78$ show the contents of the `CircleEllipse` method. Line 68 sets up the `poly` object and in the following line places the object at the starting point using the `moveTo` method. The curve is generated using the `lineTo` method within the `for` loop in line 73. Upon completion of the path, the path is then closed using `closePath()` method in line 75. Line 76 casts the path into a `GArea` which is then placed within a `Struct` in line 77. The structure is then returned in line 78.

Lines 37-40 define a 64 sided circle with a radius of 10.0μm within a GDS layer number 4. The shape is cast to a `circle` structure.

```
37          Double radX = 10.00, radY = 10.00;
38          int numberOfSides = 64, gdsLayer = 4;
```

Listing 2: CircleEllipse method from Ex04CircleEllipse.java

```
78      Path2D.Double poly = new Path2D.Double();
79      poly.moveTo(x + radiusX, y);
80      int steps = numSides;
81      double c = 2 * Math.PI / numSides;
82      for (int i = 0; i < steps; i++) {
83          poly.lineTo(radiusX * Math.cos(i * c), radiusY * Math.sin(i * c));
84      }
85      poly.closePath();
86      GArea v = new GArea(poly, layer);
87      Struct lf = new Struct("CircleEllipse_" + (int) (1000 * radiusX) + "_"
            + (int) (1000 * radiusY), v);
88      return lf;
```

```
39          Struct circle = CircleEllipse(0, 0, radX, radY, numberOfSides,
                gdsLayer);
40          lib.add(new Ref(circle, 0, 0));
```

Lines 42-44 define a 64 sided ellipse with a radius of 10.0μm and 4.0μm in x and y respectively, within a GDS layer number 4. The shape is cast to a `circle` structure.

```
42          radY = 4.0;
43          Struct ellipse = CircleEllipse(0, 0, radX, radY, numberOfSides,
                gdsLayer);
44          lib.add(new Ref(ellipse, 0, 0));
```

Analogously, using the `CircleEllipseArea` method, lines 49-51 cast a 4μm radius circle into a `circle2 GArea`.

```
49          radX = 4.0;
50          radY = 4.0;
51          GArea circle2 = CircleEllipseArea(0, 0, radX, radY, numberOfSides,
                gdsLayer);
```

`circle2 GArea` is then cast into a `circleGArea` structure in the following line

```
53          Struct circleGArea = new Struct("circleGArea", circle2);
```

4.2 Torus - Subtraction of two Circles

`Ex05TorrusViaBoolean.java` is an example of generating a torus by subtracting two circular structures. As defined in the example above, here a circular objects is generated within a structure `cir` and another circular object within a `GArea cir2` in lines 40 and 42 respectively.

```
40          Struct cir = CircleEllipse(0, 0, radX, radY, numberOfSides,
                gdsLayer);
```

```
42          GArea cir2 = CircleEllipseArea(0, 0, 2 * radX, 2 * radY,
                numberOfSides, gdsLayer);
```

Subtraction is performed between the two circular objects and cast into a `torus` structure in lines 44 and 45 respectively.

```
44          cir2.subtract(cir);

45          Struct torus = new Struct("Torrus", cir2);
```

4.3 Torus - Two Path Loops

`Ex06Torus.java` is an example of a torus generated using 2 paths within a `Torus` method. Here, an inner and outer paths are used to define the two respective perimeters of the torus. `Torus` method returns a `Struct` method and is defined in line 53 as

```
53  static Struct Torus(double x, double y, double radInner, double radOuter,
54          double angle, int NumSides, int layer) {
```

where the shape is centered around (x, y), `radInner` and `radOuter` are the inner and outer radius respectively, `angle` is the sweep angle (from $0 - 360°$), `NumSides` represents the number of sides for the generated shapes and `layer` is the GDS layer number. Lines $36 - 39$ define a 64 sided torus with respective inner and outer radii of 5.0μm and 8.0μm, sweep angle of 360°, using GDS layer number 4 (Figure 4a).

```
36          double radiusInner = 5.0, radiusOuter = 8.0, sweepAngle = 360.0;
37          int numberOfSides = 64, gdsLayer = 4;
38          Struct torus = Torus(0, 0, radiusInner, radiusOuter, sweepAngle,
                numberOfSides, gdsLayer);
39          lib.add(new Ref(torus, 0, 0));
```

With similar dimensions, a torus with a sweep angle of 110° is defined within lines $41 - 43$ (Figure 4b)

```
41          sweepAngle = 110.0;
42          Struct torus2 = Torus(0, 0, radiusInner, radiusOuter, sweepAngle,
                numberOfSides, gdsLayer);
43          lib.add(new Ref(torus2, 0, 0));
```

4.4 Bezier Curves

`Ex07BezierCurve.java` employs the `curveTo` method to generate a Bezier curve. This method uses two control points for the resulting curve. Here a method `Bezier` returns a `Struct` as defined in lines $58 - 59$

```
58  static Struct Bezier(double x, double y, double cx1, double cy1, double cx2
        , double cy2,
59          double x2, double y2, double endW, int layer) {
```

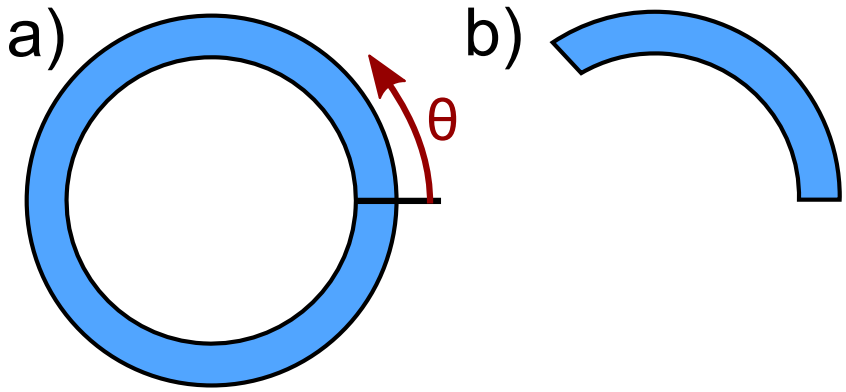

Figure 4: *Torus example at sweep angles of (a) $\theta = 360°$ and (b) $\theta = 110°$.*

where (x, y) and (x_2, y_2) are the respective starting and end points of the curve, with two control points (cx_1, cy_1) and (cx_2, cy_2), `endW` is the curve width and `layer` is the GDS layer number. Listing 4 contains lines 60-69 which are contents of the Bezier method. Line 60 sets up the `poly` object and in the following line places the object at the starting point (x, y) using the `moveTo` method. Bezier curve is generated using the `curveTo` method in line 62. Line 63 sets up the `BasicStroke` by providing the curve width (`endW`) value, defining end caps (`BasicStroke.CAP_BUTT`) and line joins (`BasicStroke.JOIN_ROUND`). `BasicStroke` allows `CAP_BUTT`, `CAP_ROUND`, and `CAP_SQUARE` for end cap declarations, and `JOIN_BEVEL`, `JOIN_MITER`, and `JOIN_ROUND` for line joins declarations. For more available options, consult the Java `BasicStroke` documentation. The path is then added to a `GArea` in line 64. The following line 65 defines the structure object `lf` with a concatenated string name composed of passed parameters. Line 69 returns the `lf` structure.

Listing 3: Setting up and passing parameters to Bezier method

```
60      Path2D.Double poly = new Path2D.Double();
61      poly.moveTo(x, y);
62      poly.curveTo(cx1, cy1, cx2, cy2, x2, y2);
63      Stroke bs = new BasicStroke((float) endW, BasicStroke.CAP_BUTT,
            BasicStroke.JOIN_ROUND);
64      GArea v = new GArea(bs.createStrokedShape(poly), layer);
65      Struct lf = new Struct("bezier_" + (int) (x * 1000) + "_" + (int) (y *
            1000) + "_"
66              + (int) (cx1 * 1000) + "_" + (int) (cy1 * 1000) + "_"
67              + (int) (cx2 * 1000) + "_" + (int) (cy2 * 1000) + "_"
68              + (int) (x2 * 1000) + "_" + (int) (y2 * 1000) + "_" + layer, v)
                    ;
69      return lf;
```

Lines $36-47$ create a 1μm wide S-shaped curve starting at $(0,0)$, ending at $(20,10)$ with 2 control points placed at $(10,0)$ and $(10,10)$.

Listing 4: Setting up and passing parameters to Bezier method

```
36          // start coordinates
37          double x = 0.0, y = 0.0;
38          // Control Points for bezier curve
39          double cx1 = 10.0, cy1 = 0, cx2 = 10, cy2 = 10;
40          // end coordinates
41          double x2 = 20.0, y2 = 10.0;
42          //line width and gds Layer number
43          double width = 1.0;
44          int gdsLayer = 4;
45          // create bezier Struct - S-Shaped curve starting at (0,0) and
                ending at (20, 10)
46          Struct bezierShape = Bezier(x, y, cx1, cy1, cx2, cy2, x2, y2, width
                , gdsLayer);
47          lib.add(new Ref(bezierShape, 0, 0));
```

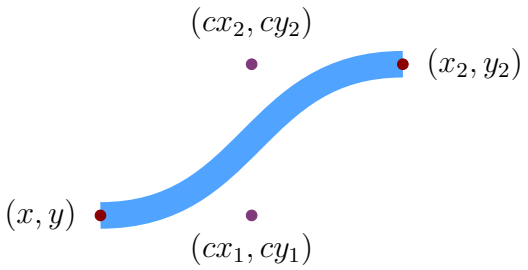

Figure 5: *Illustration of an S-shaped Bezier curve with the following parameters* $(x = 0, y = 0)$, $(x_2 = 20, y_2 = 10)$, $(cx_1 = 10, cy_1 = 0)$, $(cx_2 = 10, cy_2 = 10)$.

5 Text - Labels

`Ex08Text.java` is an example showing usage of `Text` objects and affine transforms. Listing 5 (lines $36-43$) shows a procedure for placing text of a particular font info GDS. Line 37 casts a string (defined by line 36) of size 50μm (see Java docs for Font definitions) into layer 1. Line 40 sets the font to `Serif`. Chosen fonts must be vector fonts, otherwise they will not render properly. For instance `Wingdings` font will result in squares. Line 41 sets up the rendering resolution to 100nm. Units of `setRenderReso` method are in μm with a value of a `double`. Line 42 adds the text label object `lb` to a structure object `labelString`. Left most output of Figure 7 shows the text string rendered within the resulting GDSII file. Figure 7 shows various fonts used to create a string `JAVA GDS2`.

Listing 5: Text to GDS from Ex08Text.java

```
36        // Placing a text string within a cell -> GDS
37        String label = "Bam!";
38        // setup Text
39        Text lb = new Text(label, 1, 50.0);  // (string,layer,size in um)
40        lb.setFont("Serif");          // vector font
41        lb.setRenderReso(0.100);      // rendering resolution in um
42        Struct labelString = new Struct("lb", lb);
43        lib.add(new Ref(labelString, 0, 0));
```

Figure 6: *Text example. (left) String defined by lines* $36-43$ *and (right) affine transform* ($-90°$ *rotation) operation on the string.*

Java's affine transforms are used with `Text` objects and shown in Listing 6, lines $45-52$. Here, extents of the string are determined, then the text is rotated by $-90°$ (`-Math.PI`/2.0) around the centroid (see Figure 7). Line 51 shows a structure composed of an `GArea` built from the rotated `Area` element. Java class `Area` is in `java.awt.geom.Area`. A myriad of operations and transformations are available for the `Area` class. Consult Java docs for `Area` constructor and method summaries and details.

Figure 7: *Various fonts used to create a string of characters.*

Listing 6: Text to GDS from Ex08Text.java

```
45          // Using affine transforms to find center and rotate objects
46          // Rotating the above string by -90 degrees around the centroid
47          Area a = new Area(lb.getArea());
48          Rectangle2D rec = a.getBounds2D();
49          a.transform(AffineTransform.getTranslateInstance(-rec.getCenterX(),
                -rec.getCenterY()));
50          a.transform(AffineTransform.getRotateInstance(-Math.PI / 2.0));
51          Struct lbl = new Struct("lblRotated", new GArea(a, 1));
52          lib.add(new Ref(lbl, 0, 0));
```

6 Complex Shapes

6.1 Fractals - Sierpinski Triangle

`Ex09FractalSierpinski` examples creates a Sierpinski triangle fractal structure. The structure is self similar and is composed of various levels of repeating structures. Figure 8 shows the various GDS structures generated from the output of `Ex09FractalSierpinski.java`. The figure illustrates the method used to create the self similar Sierpinski triangle fractal structure. First a triangle is formed (Figure 8a) and stored into a structure named `v0`. Then structure `v0` is instanced 3 times to create structure `v1` (Figure 8a)), then structure `v1` is instanced 3 times to create structure `v2` (Figure 8c)), and so on. This process is repeated a number of times, as defined by the variable `iterations` on line 35. The process starts with by creating a triangle using a path in conjunction with `moveTo` (moving to a first point), then various `lineTo` segments to create the triangle, and finally a `closePath` statement to close the path. This procedure is captured in lines $37 - 42$.

```
34          //number of Sierpinski levels
35          int iterations = 10;
36          //creating a triangle structure
37          Path2D.Double poly = new Path2D.Double();
38          poly.moveTo(0, 0);
39          poly.lineTo(1, 2);
40          poly.lineTo(2, 0);
41          poly.lineTo(0, 0);
42          poly.closePath();
43          GArea va = new GArea(poly, 2);
```

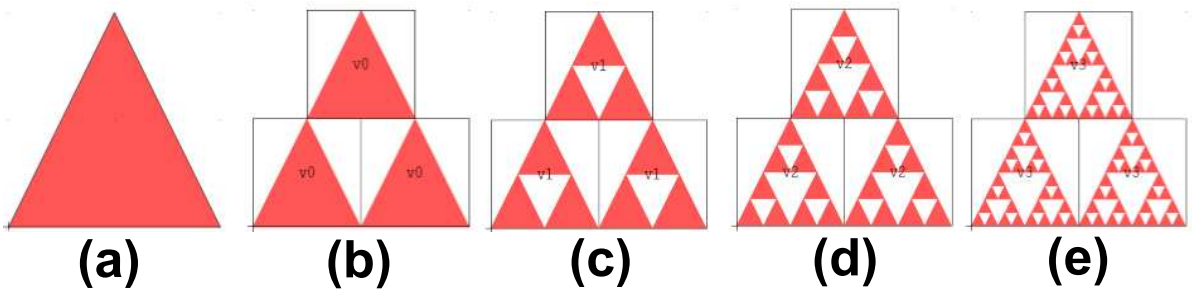

Figure 8: *Sierpinski triangle formation from various structures. (a) single triangle v0, (b) structure v1 formed from 3 instances of v0, (c) structure v2 formed from 3 instances of v1, (d) structure v3 formed from 3 instances of v2 and (e) structure v4 formed from 3 instances of v3.*

Lines 45 – 64 define the Sierpinski triangle formation. As before, a structure called `top` is defined in line 46. Line 46 defines an `ArrayList` of structures and the first triangle (one formed in lines 37 – 43) is added to the `ArrayList` as a structure `v0`. The `for` loop, defined by lines 51 – 63, scales and instances the previously defined structure three times into a new element of the `ArrayList`. To further elaborate, first time through the loop, the simple triangle structure `v0` created in lines 37 – 43, is scaled and instanced three times and placed into a structure `v1`. This placement occurs in line 62 where structure `temp` is placed into the `ArrayList`. First time through the loop, counter `i = 1`, hence the `temp` structure is named `v1`, as defined in line 52. This procedure is repeated up until the counter reaches a number less than the iteration number. With `iterations = 10`, we generate 10 structures used (`v0` to `v9`) to create the Sierpinski triangle.

```
45          Struct top = new Struct("top");
46          ArrayList<Struct> v = new ArrayList();
47          v.add(new Struct("v0", va));
48          Struct temp;
49          Ref r;
50          int i;
51          for (i = 1; i < iterations; i++) {
52              temp = new Struct("v" + i);
53              r = new Ref(v.get(i - 1), 0, 0);
54              r.setMag(.5);
55              temp.add(r);
56              r = new Ref(v.get(i - 1), .5, 1);
57              r.setMag(.5);
58              temp.add(r);
59              r = new Ref(v.get(i - 1), 1, 0);
60              r.setMag(.5);
61              temp.add(r);
62              v.add(temp);
63          }
64          top.add(new Ref(v.get(i - 1), 0, 0));
```

6.2 Fractals - Curved Tree

`Ex10CurvedTree` examples creates a curved tree fractal structure. In a similar fashion to the Sierpinski example, here a structure is created using a bezier curve (Figure 9a), the structure is then mirrored (Figure 9b), the mirrored structure is then scaled and repeated to create the fractal curved tree structure (Figure 9c-e). In a similar manner to the Bezier curve example, the curved shape from Figure 9a is constructed using a `moveTo` method to define the starting point (line 40), then a `curveTo` method to generate the bezier curve using two control points and an end point (line 41). The path is then

stroked (line 42) and cast into a `GArea` (line 43). In line 44, the resulting `GArea` is cast into a structure called `s`. The structure `s` is then instanced into a structure called `d0` (line 45). Then a mirrored and 180° rotated instance of `s` is placed into structure `d0` to create the base zero level curved tree structure seen in Figure 9b. Line 47 creates an `ArrayList` object and in line 48 stores `d0` into the first element (or element zero) of the `ArrayList`.

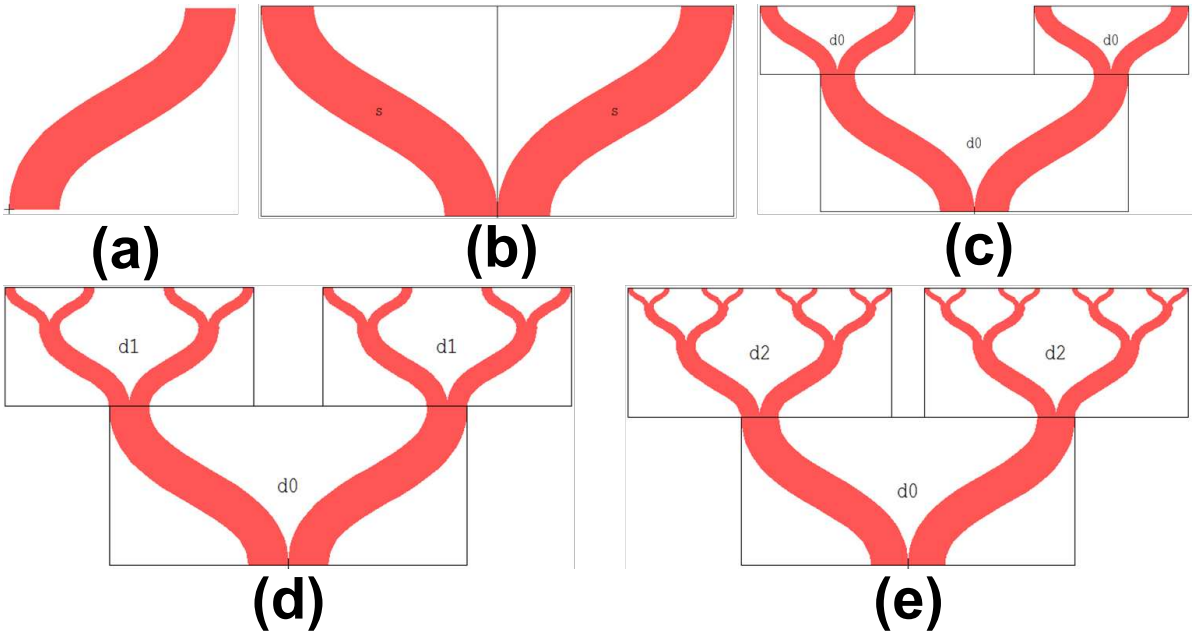

Figure 9: *Curved tree fractal structure. (a) bezier curve is formed and stored into strucure* `s`*, (b) structure is then mirrored and stored into a structure* `d0`*, (c) structure* `d1` *is formed from three scaled instances of* `d0`*, (d) structure* `d2` *formed from* `d0` *and two instances of* `d1`*, (d) structure* `d3` *formed from* `d0` *and two instances of* `d2`*.*

```
37          //number of levels
38          int iterations = 12;
39          Path2D.Double poly3 = new Path2D.Double();
40          poly3.moveTo(0.0625, 0);
41          poly3.curveTo(.0625, .25, .5, .25, .5, .5);
42          Stroke bs = new BasicStroke((float) .125, BasicStroke.CAP_BUTT,
                BasicStroke.JOIN_MITER, (float) 2.5);
43          GArea fa = new GArea(bs.createStrokedShape(poly3), 2);
44          Struct s = new Struct("s", fa);
45          Struct d = new Struct("d0", new Ref(s, 0, 0));
46          d.add(new Ref(s, 0, 0, MIRROR, 180));
47          ArrayList<Struct> n = new ArrayList();
48          n.add(d);
```

NOTE: The Ex10CurvedTree class implements an interface called `Const` (line 26). The implementation allows for mirroring of structures, as performed in line 46 where the structure `s` is mirrored and instanced into structure `d0`.

```
26 public class Ex10CurvedTree implements Const {
```

In a similar fashion to the Sierpinski example, the curved tree fractal is formed using a `for` loop, defined by lines $54 - 65$. The loop first creates a new structure with each iteration (`d1`, `d2`, `d3`, `d4` ...) in line 55. The previously defined structure is then instanced (line 56), then scaled (line 57), and added to the created structure (line 58). The structure defines the upper left branch of the new structure (e.g. upper left structure in Figure 9c). This operation (lines $57 - 59$) is repeated in lines $59 - 61$ to create the upper right branch of the new structure (e.g. upper right structure in Figure 9c). The zero level structure is then instanced and added to the new structure (lines $62 - 63$). This represents the main structure connected to the upper and lower branches in Figure 9c. The next iteration forms a similar structure with more complexity Figure 9d, and so on. After the loop terminates on the final level (defined by variable `iterations`, in line 38) the last structure is then instanced within the `top` cell.

```
50          Struct top = new Struct("top");
51          Struct temp;
52          Ref r;
53          int i;
54          for (i = 1; i < iterations; i++) {
55              temp = new Struct("d" + i);
56              r = new Ref(n.get(i - 1), -.5, .5);
57              r.setMag(.5);
58              temp.add(r);
59              r = new Ref(n.get(i - 1), .5, .5);
60              r.setMag(.5);
61              temp.add(r);
62              r = new Ref(n.get(0), 0, 0);
63              temp.add(r);
64              n.add(temp);
65          }
66          top.add(new Ref(n.get(i - 1), 4, 4));
```

6.3 Fractals - Another Tree

In a similar fashion as described by the previous two examples, the `Ex11AnotherTree` example creates a base structure using a path with `lineTo` segments and stores the path into a `GArea` (lines 35-42). The tree fractal is formed using a `for` loop in lines $50-67$ using instancing of scaled and rotated shapes. The resulting GDS file is a tree with 12 levels of complexity, defined by `iterations = 12` in line 34, and is shown in Figure 10.

```
33          //number of levels
34          int iterations = 12;
35          Path2D.Double poly2 = new Path2D.Double();
36          poly2.moveTo(0, 0);
37          poly2.lineTo(.25, 2);
38          poly2.lineTo(.5, 2);
39          poly2.lineTo(.35, 0);
40          poly2.lineTo(0, 0);
41          poly2.closePath();
42          GArea tr = new GArea(poly2, 2);
43
44          Struct top = new Struct("top");
45          Struct temp;
46          Ref r;
47          int i;
48          ArrayList<Struct> t = new ArrayList();
49          t.add(new Struct("t0", tr));
50          for (i = 1; i < 12; i++) {
51              temp = new Struct("t" + i);
52              r = new Ref(t.get(i - 1), .25, 2);
53              r.setMag(.75);
54              r.setAngle(355);
55              temp.add(r);
56              r = new Ref(t.get(i - 1), .2, 1.7);
57              r.setMag(.44);
58              r.setAngle(60);
59              temp.add(r);
60              r = new Ref(t.get(i - 1), .4, 1);
61              r.setMag(.37);
62              r.setAngle(300);
63              temp.add(r);
64              r = new Ref(t.get(0), 0, 0);
65              temp.add(r);
66              t.add(temp);
67          }
68          top.add(new Ref(t.get(i - 1), 0, 3));
```

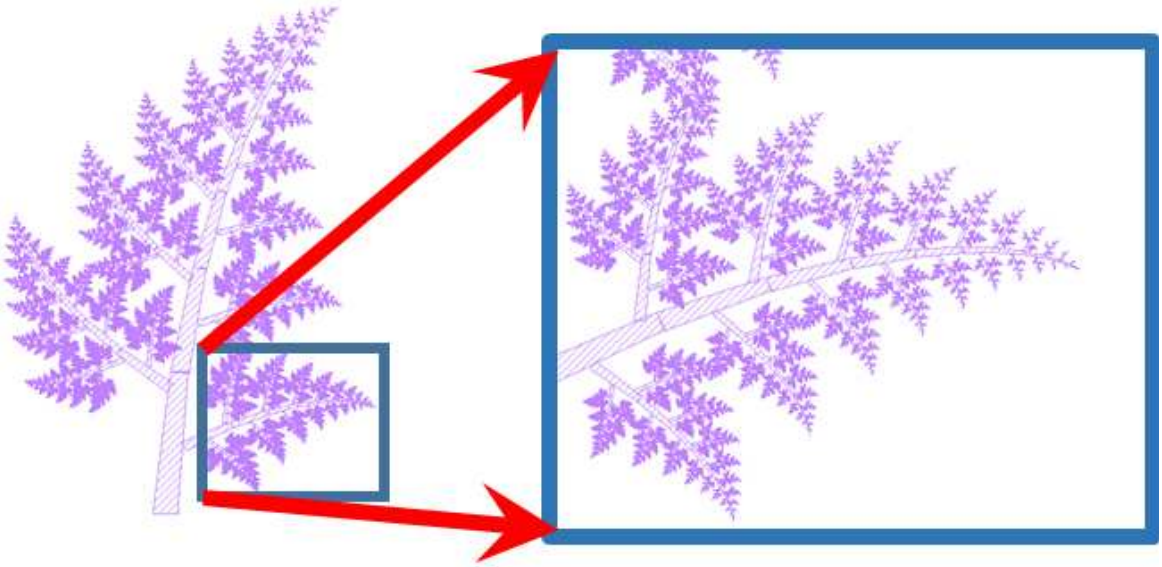

Figure 10: *Tree with branches fractal structure with* 12 *levels (variable* `iterations = 12` *in line* 34*). The inset shows a zoomed-in image of the tree.*

6.4 Fractals - Yet Another Tree

`Ex12YetAnotherTree` class implements the `Const` interface in order to use mirroring (the `MIRROR` constant). The example has a `tree` method used to create fractals. Here we create a `top` structure and create 3 tree fractals using lines 40, 42 and 44.

```
38        Struct top = new Struct("top");
39        //cell tree11_1843197
40        top.add(new Ref(tree(.025, 2.44, 1.5, 12), 6, 4, 0, 4, 0));
41        //cell tree11_-458895
42        top.add(new Ref(tree(.125, -.675, 2.5, 12), 6, 8, 0, 4, 0));
43        //cell tree39_3962446
44        top.add(new Ref(tree(.02, 5, 14, 40), 6, 16, 0, 4, 0));
```

Tree method creates fractals in a similar fashion as described in the previous examples and illustrates virtually endless possibilities for creating shapes of most extreme complexity. Figure 11 shows 3 trees created using the `tree` method.

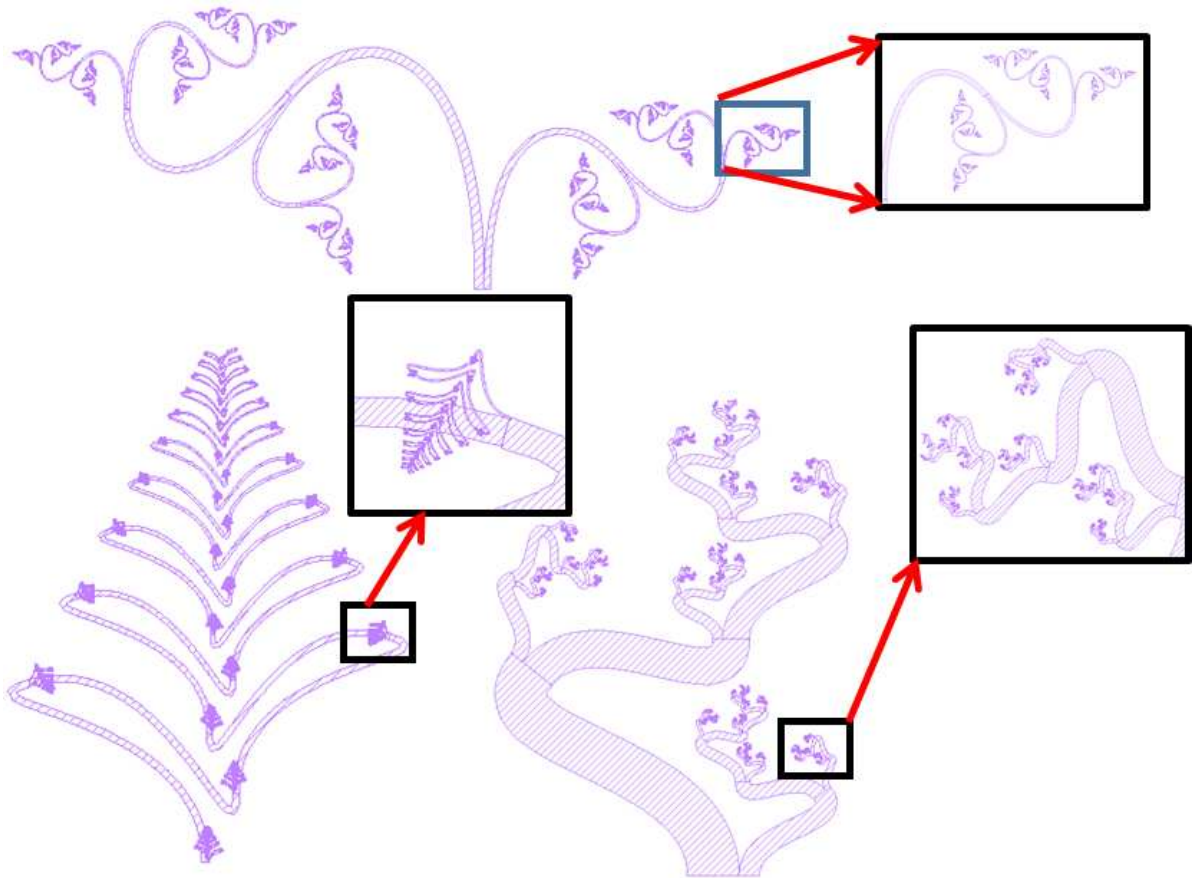

Figure 11: *Lower left GDS shape is structure* `tree39_3962446` *created using line 44. Lower right GDS shape is structure* `tree11_-458895` *created using line 42. Top GDS shape is structure* `tree11_1843197` *created using line 40. The insets show zoomed-in regions of various trees.*

7 JGDS2 - Details

7.1 Arrays - Nonorthogonal

GDS format allows for a wide range of parameters to control instanced arrays of structures. Constructors within the `GDS2` library for the `Array` are the following:

`Array(Struct s, double` $posX$`, double` $posY$`, int` $cols$`, int` $rows$`, double` rxx`, double` ryy`)`
`Array(Struct s, double` $posX$`, double` $posY$`, int` $cols$`, int` $rows$`, double` rxx`, double` ryy`, double` ang`)`
`Array(Struct s, double` $posX$`, double` $posY$`, int` $cols$`, int` $rows$`, double` rxx`, double` rxy`, double` ryx`, double` ryy`)`
`Array(Struct s, double` $posX$`, double` $posY$`, int` $cols$`, int` $rows$`, double` rxx`, double` rxy`, double` ryx`, double` ryy`, double` ang`)`

In a previous example, `Ex02RectangleArray` illustrated the first of the four `Array` constructors. In `Ex13Arrays` we focus on the last of the constructors, where the following parameters are being user: `s, posX, posY, cols, rows, rxx, rxy, ryx, ryy, ang`. Line $32-43$ show construction of two arrays, each with 4 columns and 5 rows instancing a 2μm $\times 4\mu$m rectangle, with differing range (r_{ij}) and angle parameters.

```
32        Struct top = new Struct("top");
33        Struct oneRectangle = new Struct("singleRectangle");
34        oneRectangle.add(new Rect(0.0, 0.0, 2.0, 4.0, 44));
35        Struct arrayV1 = new Struct("arrayV1");
36        arrayV1.add(new Array(oneRectangle, 0.0, 0.0, 4, 5,
37                (10.0 * 4), (1.0 * 4), (4.0 * 5), (8.0 * 5), 20));
38        Struct arrayV2 = new Struct("arrayV2");
39        arrayV2.add(new Array(oneRectangle, 0.0, 0.0, 4, 5,
40                (8.0 * 4), (2.0 * 4), (5.0 * 5), (10.0 * 5), 0));
41        top.add(new Ref(arrayV1, 0, 0));
42        top.add(new Ref(arrayV2, 60, 0));
43        lib.add(new Ref(top, 0, 0));
```

Lines $36-37$ define an `Array` where each element is rotated by `ang`$= 20°$. The second array is defined by lines $39-40$ without any rotation. The two arrays are placed side by side in Figure 12.
The range, r_{ij}, parameters are defined in the following manner:
$r_{xx} = C_x * cols, \quad r_{xx} = C_y * cols, \quad r_{yx} = R_x * rows, \quad r_{yy} = R_y * rows$
where $cols$ and $rows$ are number of rows and columns respectively, C_x and C_y are the corresponding column steps in x and y, and R_x and R_y are the corresponding row steps in x and y. To further exemplify these parameters, consider the right most generated array in Figure 12, where the corresponding number of columns and rows are 4 and 5. Here

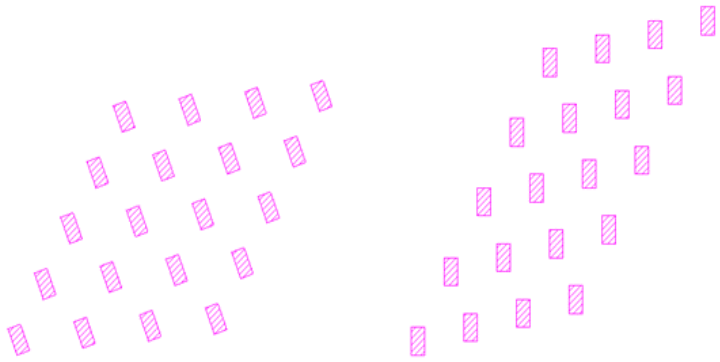

Figure 12: *Two arrays defined by lines* $36-37$ *(left image) with each instance rotated by* $20°$ *and lines* $39-40$ *(right image) constructing an instance without rotation. The two arrays have differing range parameters.*

$C_x = 8.0\mu$m, $C_y = 2.0\mu$m, $R_x = 5.0\mu$m, $R_y = 10.0\mu$m, without `ang`$= 0°$. Illustration in figure 13 shows how radial parameters are defined. Again, in this example, we are using the following `Array` constructor:
`Array(Struct s, double` *posX*`, double` *posY*`, int` *cols*`, int` *rows*`, double` *rxx*`, double` *rxy*`, double` *ryx*`, double` *ryy*`, double` *ang*`)`

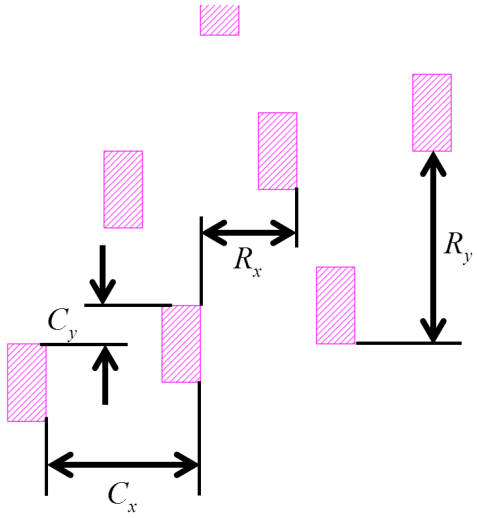

Figure 13: *Zoomed-in illustration showing column and row steps in* x *and* y *directions.*

7.2 setRenderReso

`setRenderReso` is a public method that sets the resolution of the rendered Bezier curves. As seen in the `JGDS2` java docs, method details are the following:
`public void setRenderReso(double r)`
parameter r is the value for rendering resolution of Bezier curves. This method is used, for instance, in the creation of `Text` objects, where vectorized fonts are defined by `lineTo` and `curveTo` segments, the latter being a Bezier curve with two control points (see Bezier curve subsection for further explanation). Lines $32-48$ (in `Ex14TextRenderReso.java` example file) generate three text structures with a character `f` rendered at $r = 0.010$, $r = 0.100$ and $r = 1.000$, generating respectively 201, 67 and 34 points for the 50μm character shape. Figure 14 shows the three examples of character `f` rendered at $r = 0.010$, $r = 0.100$ and $r = 1.000$.

As before, line 32 creates a structure called `top`. Line 33 then sets up a `Text` object `tf1` with a character `f` cast to GDS layer 1 of height 50μm. Line 34 sets the font for the text object to `Serif`. Line 35 then sets the render resolution of the Bezier portions of the font to $r = 0.010\mu$m. Line 36 casts `tf1` to a structure `f1`. Similar procedure is followed to create text objects `tf2` and `tf3` cast respectively to structures `f2` and `f3` with corresponding rendering resolution values of $r = 0.100\mu$m and $r = 1.000\mu$m. The three structures are then instanced, 25μm apart in the $x-$direction, within the `top` cell. In line 48, `top` cell is added to the GDS library.

Listing 7: setRenderReso example Ex14TextRenderReso.java

```
32        Struct top = new Struct("top");
33        Text tf1 = new Text("f", 1, 50.0);  // (string,layer,size in um)
34        tf1.setFont("Serif");
35        tf1.setRenderReso(0.010);
36        Struct f1 = new Struct("f1", tf1);
37        Text tf2 = new Text("f", 1, 50.0);  // (string,layer,size in um)
38        tf2.setFont("Serif");
39        tf2.setRenderReso(0.100);
40        Struct f2 = new Struct("f2", tf2);
41        Text tf3 = new Text("f", 1, 50.0);  // (string,layer,size in um)
42        tf3.setFont("Serif");
43        tf3.setRenderReso(1.000);
44        Struct f3 = new Struct("f3", tf3);
45        top.add(new Ref(f1, 0, 0));
46        top.add(new Ref(f2, 25, 0));
47        top.add(new Ref(f3, 50, 0));
48        lib.add(new Ref(top, 0, 0));
```

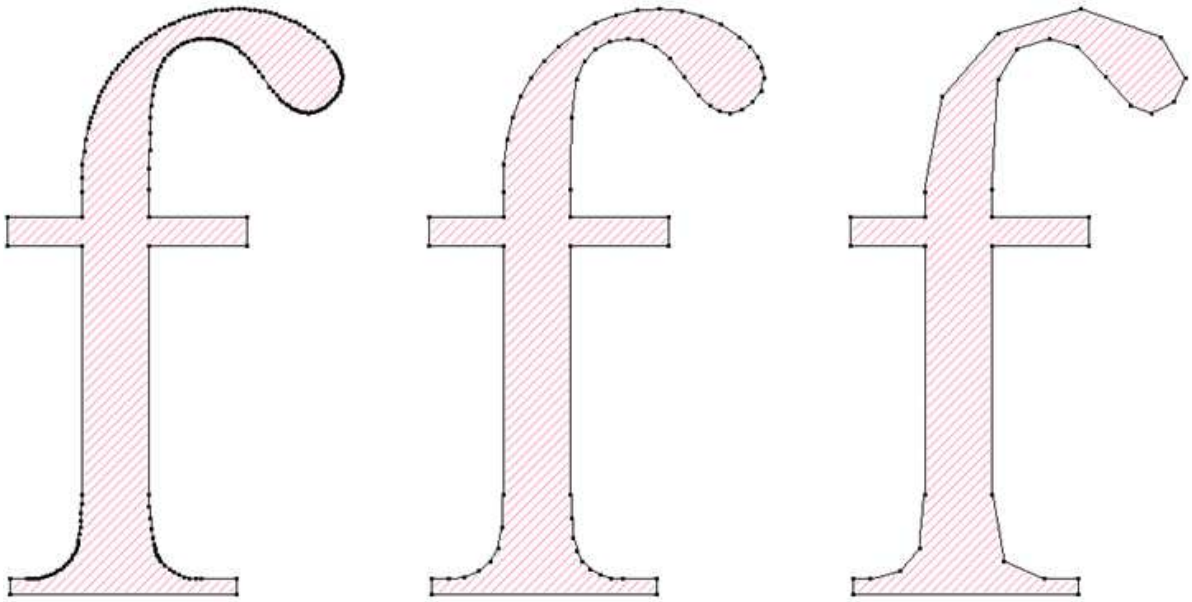

Figure 14: *setRenderReso(r) example with a text string f, where* $r = 0.010$ *(left) generated* 201 *points,* $r = 0.100$ *(center) generated* 67 *points, and* $r = 1.000$ *(right) generated* 34 *points. KLayout was used to view the generated GDS file. Displayed dots along the curve represent vertices of the shape.*

7.3 Circumvent Point Limits with GDS files

The JGDS2 library will export shapes with an arbitrary number of points. Most software such as LEdit, Genisys Layout BEAMER will complain if the point count is greater than 8000 per shape. To circumvent this dilemma, these files could be opened in `KLayout` and re-exported. For instance, creating a `Text` object with a character `f`, that's 500μm tall, with a `setRenderReso(0.0001)`, produces a shape with 12730 points. `KLayout` version 0.22.3 reads the content without errors.

7.4 Parts Lib

The `PartLib` package has a `PartsLib.class` file which contains various patters that could be reused. For instance, currently the parts lib has `circle, torus, Vern, and frame`. We will continuously update and add more to the parts library. These are static methods and could be called without creating new objects. For instance, lines $34-37$ create various shapes, that are then cast to the `top` cell and added to the library (lines $38-42$).

Circle: Line 34 creates a `GArea` circle of radius 5.0μm, with 64 sides in a GDS layer number 44.

Torus: Line 35 creates a `GArea` torus with an inner radius 8.0μm, an outer radius 10.0μm, with 64 sides in a GDS layer number 44.

Verniers: Line 36 creates a vernier structure between two layers, first layer being 4 and the second 44, the resolution is set to 0.050μm, with 20 ticks, the two layer string labels are set to `L4` and `L44`.

Frame: Line 37 creates a reticle frame for the `ASML` stepper, with a barcode set to a value of 'BarCode', the reticle label value set to 'Label' using a `Serif` font. This method automatically places reticle alignment marks within the frame.

Listing 8: Parts Lib example Ex15PartsLib.java

```
42        Struct top = new Struct("topp");
43        GArea cir = PartsLib.circle(5.0, 64, 44);
44        GArea tor = PartsLib.torus(8.0, 10.0, 64, 44);
45        Struct vrn = PartsLib.Vern(4, 44, 0.050, 20, "L4", "L44");
46        Struct frm = PartsLib.frame(ASML, "BarCode", "Label", "Serif");
47        top.add(cir);
48        top.add(tor);
49        top.add(new Ref(vrn, 100,0));
50        top.add(new Ref(frm, 0,0));
51        lib.add(new Ref(top, 0, 0));
```

7.5 setReso

`setReso` defines the snapping grid and is set to 0.001μm (1nm). Current generation of high resolution electron beam lithography systems have pixel values $< 1nm$, for instance the `JEOL9500` has a 0.5nm accessible grid within a 1mm field and in the fifth lens mode, the `JEOL6300` has a 0.125nm accessible grid within a 62.5μm field. `setReso (double r)` allows changing of the grid, where r is in units of μm. Within the `Ex16setReso` example, lines $31 - 34$ create a fractal tree by calling a static method from `Ex12YetAnotherTree` example and set the resolution to $1 \times 10^{-6}\mu$m, i.e. femtometers. Left side of figure 15 shows the tree fractal with default `setReso` units (i.e. without line 33), and the right side of figure 15 shows the tree fractal GDS file with `setReso` set to a femtometer (10^{-12}m).

```
31          Struct top = new Struct("top");
32          top.add(new Ref(Ex12YetAnotherTree.tree(.02, 5, 14, 10), 6, 16, 0,
                4, 0));
33          GDS2.setReso(1e-6); //1e-6 microns, i.e. femtometer
34          lib.add(new Ref(top, 0, 0));
```

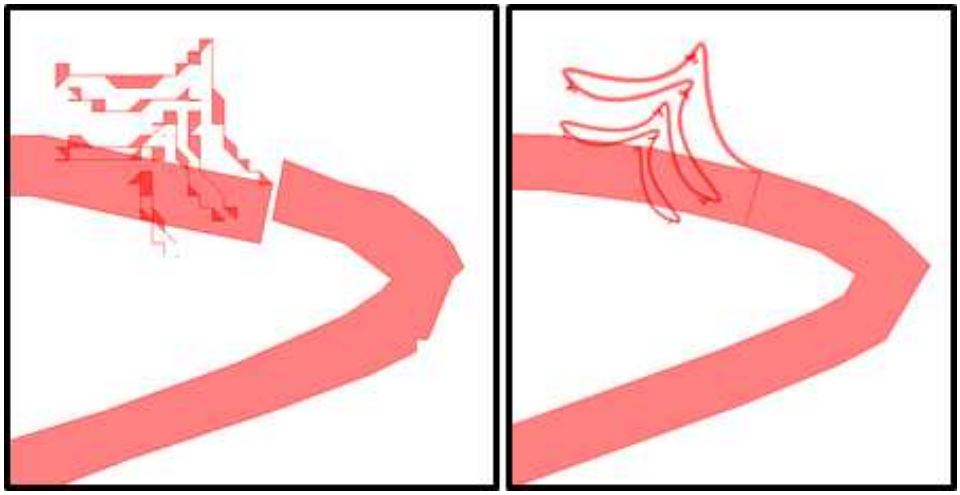

Figure 15: *`setReso(r)` example with a tree fractal with default `setReso` value (left) and with `setReso`*$(1e-6)$*, i.e. `setReso` set to a femtometer, (right).*