Technique to Rectify Displaced Vector Graphics Drawn Over Scalable Raster Drawing

Yew Kwang Hooi, Wan Fatimah Wan Ahmad and Leong Siew Yoong

***Abstract*—Demarcation circumscribes sections of interest of an image by drawing perimeters known as clouds. The clouds are vector graphics stored as an array of coordinate points drawn on the raster image at runtime. At design time, presence of two or more separate coordinate systems introduces disparity in coordinate scales and origin. Consequently, clouds drawn by are sometimes displaced. This paper proposed a drawing mechanism in Java Graphics2D that contains techniques to prevent graphics displacement problem in systems that combine vector graphics with scalable raster image. Calibration of scale sizes and coordinate origins are simple yet useful techniques that allow vector graphics to be drawn correctly over raster drawing regardless of the magnification ratio.**

***Index Terms*—Computer interfaces, human computer interaction, software design.**

I. INTRODUCTION

Demarcation used geometrical lines to draw shapes that highlight areas of interest on an engineering drawing. An area surrounded in the shape is called a cloud. This is a common feature of some image viewing tool such as Acrobat Reader. In this study, a software tool to view and create clouds on engineering drawing was developed. Engineering drawing is often saved in postscript for better portability and viewing [1].

In an electrical safety design risk assessment, [1]single-line diagrams (SLDs) are systematically inspected by safety engineers to identify possible design flaw that may lead to safety hazards. If any design component in an SLD is thought to compromise safety, then a cloud perimeter is drawn to mark it for further inspection and action. Clouds are depicted as geometries with dashed perimeters, as depicted in Fig. 1.

Manuscript received March 1st, 2011.
Yew Kwang Hooi is with Department of Computer and Information Sciences, Universiti Teknologi PETRONAS, Bandar Seri Iskandar, 31750 Tronoh, Perak, Malaysia.
(phone: +60166685887, fax: +6053656180, email: yewkwanghooi@petronas.com.my)
Dr. Wan Fatimah Wan Ahmad is with Department of Computer and Information Sciences, Universiti Teknologi PETRONAS, Bandar Seri Iskandar, 31750 Tronoh, Perak, Malaysia.
(phone: +60125151100, fax: +6053656180, email: fatimhd@petronas.com.my)
Leong Siew Yoong is with Department of Petrochemical Engineering, Universiti Tunku Abdul Rahman Jalan Universiti, Bandar Barat, 31900 Kampar, Perak, Malaysia.(phone: +60165518999, email: leongsy@utar.edu.my)

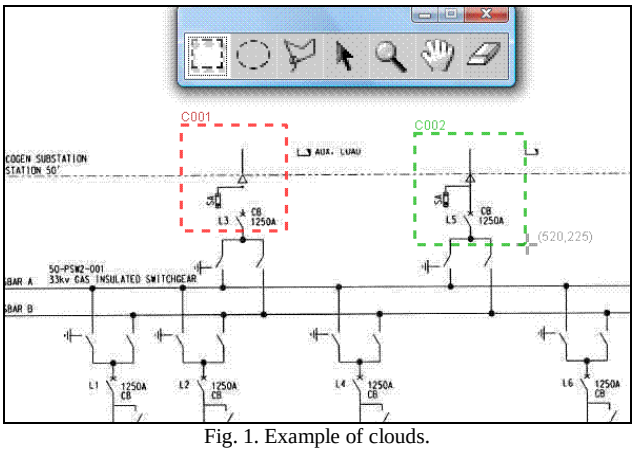

Fig. 1. Example of clouds.

The clouds should be in an editable format to accommodate modifications. Hence, cloud shape and position are stored in vector format whilst the background SLD is commonly in raster format. Instead of using vector format, a large amount of existing engineering drawing databases are still in raster format [1]. Storing vector-valued parameters for engineering drawing, on the other hand, provides the benefits of easy scaling, shape manipulation and undoing mistakes [2,3]. Hence, it may be often necessary to mix both types of graphics in a common environment for display or editing. Both formats should nevertheless be stored separately for later retrieval and changes if necessary, i.e. not to rasterize the vector graphics and merged with the background raster graphics at file saving.

This article shows how the Java drawing mechanism can be exploited to mix vector and raster graphics and to magnify both types of graphics. The paper pointed out the lack of facilities in standard Java's graphics library to address the displacement issue, hypothesize the cause and suggested an algorithm to address the problem is proposed. The final section discusses the result prospect and some limitations of the algorithm.

II. DRAWING MECHANISM IN JAVA

The drawing mechanism assumed by the author is common in most graphics system regardless of its implementation.

The application was developed using Java Swing and Java Graphics2D. Java is chosen for its portability and better performance of speed compensated by faster hardware nowadays. The standard Java libraries, such as Abstract Window Toolkit (AWT) and Swing, provide rich object-oriented facilities to develop Graphical User Interface (GUI) windows, components and graphics. The scope of this work is 2-Dimensional (2D) graphics, hence mechanism using Java's Graphics2D object is described.

As summarized in Fig. 2 block diagram, image of engineering drawing is first loaded into BufferedImage object, a temporary container before the drawing is displayed on the screen. While in the buffer, drawing can be refreshed, edited, magnified or mixed with new graphics on top without causing flickers [6]. This is done by Graphics2D object acquired from BufferedImage object.

Graphics is an object used to draw to screen surface in Java, .Net and most graphics systems. All graphics on the screen, both vector and raster, are redrawn by the Graphics object from stored instructions or binary values at every screen refresh during run-time.

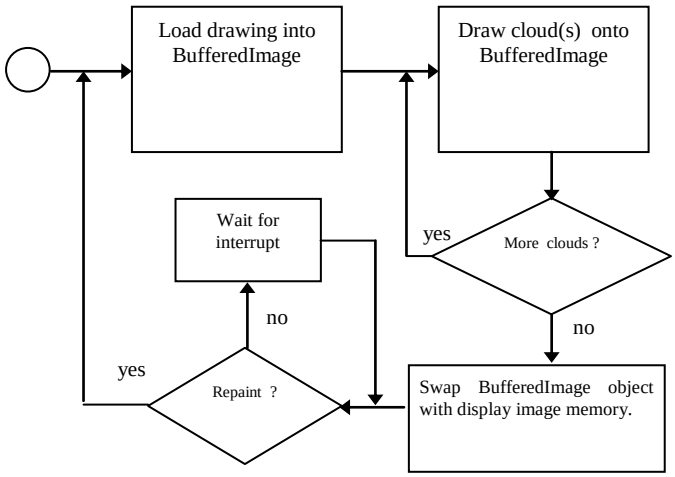

Fig. 2. Cloud drawing mechanism using Java Graphics2D.

Next, Graphics2D object checks the arrays for information of cloud objects. Each cloud object contains x and y coordinate pairs that are used by Graphics2D to determine the position and shape of the cloud(s) on buffered image.

Finally, the completed drawing is copied to display. The buffer cleared for next refresh cycle.

In the case of adding a new cloud, activating the cloud tool, as depicted in Fig. 2 and clicking a space on the engineering drawing will initiate the drawing. The consecutive clicks and mouse movement will mark the boundary of the cloud. The events generated will trigger refresh of display, invoking the following steps:

- Step 1: Load engineering drawing (raster format) into buffer and retrieve Graphics2D object.
- Step 2: Graphics2D object paints existing clouds onto the buffer.
- Step 3.1: If there is any unfinished cloud from previous refresh, Graphics2D will repaint the cloud to the last point inserted and wait for further consecutive mouse-clicks to complete the shape of the cloud. A double-click mouse event signals end of new cloud construction.
- Step 3.2: Else, wait for a new mouse click to initiate drawing of a new cloud.
- Step 4: If refresh event is generated before a new cloud is finished, save the coordinates into a temporary array for next refresh cycle.
- Step 5: Copy all buffer contents (raster format engineering drawing and vector format clouds) to screen for display.
- Step 6: Clear the buffer. Repeat all steps for next cycle.

III. SCALING MECHANISM

From the perspective of SLD, scaling allows for detailed inspection of components contained within cloud perimeter. Magnify tool is used to scale graphics, as depicted in Fig. 3.

Scaling is a useful feature to provide close-up view of an image or a document when necessary. New scaling metaphor and ratio automated by scrolling rate were tested [4,5] with promising results and reasonable practicality.

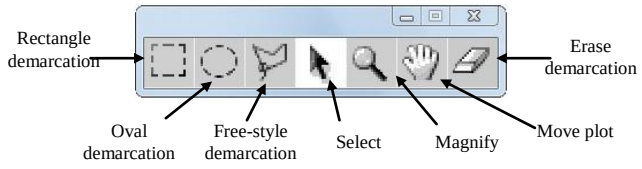

Fig. 3. Demarcation tools.

Scaling mechanism of earlier works was done on either entirely raster or entirely vector graphics but not both. This work presents scaling mechanism of a composition of both raster and vector graphics. Magnification should enlarge both clouds (vector graphics) and rasterized engineering drawing in the correct proportion simultaneously whenever a user clicks on anywhere of the image using the magnify tool.

Programmatically, at every screen refresh, the loaded image is multiplied with a double value stored in a variable. A value of 1.0 retains the size of the image at its default. The variable stores the magnification ratio, i.e. $size_{new}$: $size_{old}$, whereby size is either width or height of the display. The variable stores the scaling factor or zoom factor that multiplies the original size of the image. The significance of value stored in the variable is depicted in Table 1.

TABLE 1
MEANING OF ZOOM_FACTOR VARIABLE VALUES

ZOOM_FACTOR	Image size and cloud size
1.0	Default sizes.
< 1.0	Shrunken sizes.
> 1.0	Magnified sizes.

In code implementation, the variable is declared with default value 1.0, as depicted in Fig. 4.

```
double ZOOM_FACTOR = 1.0;
```

Fig. 4. Zoom factor declaration.

Resizing graphics is an iterative multiplication of the variable, i.e. zoom factor, with itself and triggered by mouse-click event. To magnify, the zoom factor is greater than 1.0. For example, to magnify by 10%, code in Fig. 5 is executed.

```
ZOOM_FACTOR  *=1.1;
```

Fig. 5. Code to magnify graphics sizes.

On the other hand, to shrink, the zoom factor is multiplied with a value less than 1.0. For example, to shrink by 10%, code in Fig. 6 is used.

```
ZOOM_FACTOR  *=0.9;
```

Fig. 6. Code to scale down graphics sizes.

ZOOM_FACTOR is passed as arguments to the scale method of Graphics2D object. The arguments are important to increase or decrease the original width and height of the drawing graphics. The new width and height are derived from multiplying the initial width and height with the argument respectively, as depicted in Fig. 7.

```
Graphics2D graphicsPen
= (Graphics2D)bufferedImage.getGraphics();

graphicsPen.scale(ZOOM_FACTOR,ZOOM_FACTOR);

//graphicsPen draws background image
BufferedImage originalImg =
javax.imageio.ImageIO.read(imageFile);
graphicsPen.drawImage(originalImg, null, 0, 0);

//graphicsPen draws every cloud
for (Cloud c : cloudList) {
 graphicsPen.drawPolygon(c.getXPoints(),
 c.getYPoints(), c.getPointCount());
}
```

Fig. 7. Scaling raster and vector graphics simultaneously using Java Graphics2D object.

IV. VIEWPORT SHIFT

One unexpected behavior during magnification is that the scaled image seems to drift away from the position clicked by the mouse pointer. Fig. 8. (a) depicts the initial cursor and mouse pointer positions, which are overlapping. After magnification, a gap is noticed between the cursor and the pointer due to new placement of the viewport, as depicted in Fig. 8. (b).

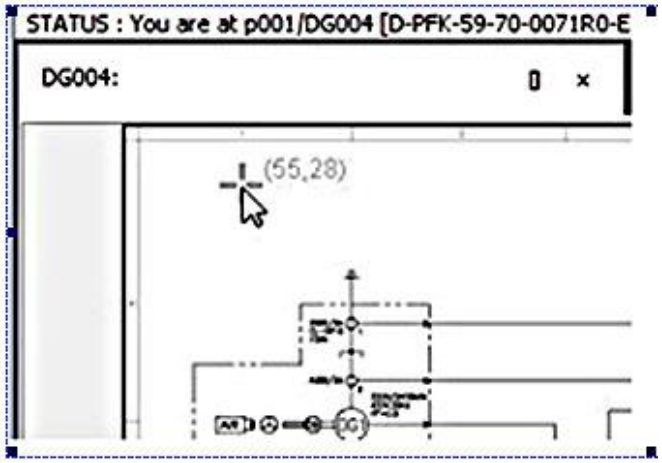

Fig. 8. (a) Initial mouse pointer and cursor positions.

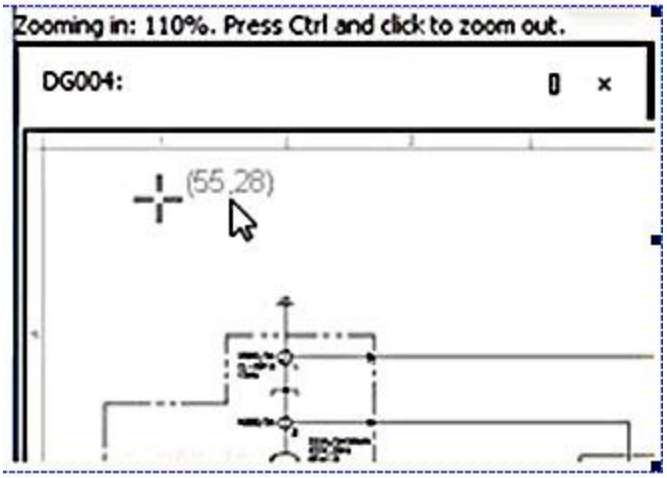

Fig. 8. (b) Gap between mouse pointer and cursor.

The cause of the drift is explainable by the position of the scaling point of reference in the display. The origin, i.e. coordinate (0,0) is located at the top left position of the screen. To rectify, the origin should be mapped to the position of the mouse-pointer. The code in Fig. 9 depicts the corrective measure.

In brief, the coordinate of the mouse pointer is retrieved and adjusted so that it is based on the image coordinate instead of

JScrollPane coordinates. The expected coordinate of the pointer after magnification can be determined by dividing the x or y coordinate value with zoom factor. Then, the difference between coordinate before or after is computed. The viewport is adjusted accordingly based on the difference computed.

```
javax.swing.JViewport viewport =
JScrollPane.getViewport();

int pointerX =
getXFromImageOrigin(e.getX());

int pointerY =
getYFromImageOrigin(e.getY());

int adjustViewportX = viewport.getX() +
(int) (pointerX - (pointerX / ZOOM_FACTOR));

int adjustViewportY = viewport.getY() +
(int) (pointerY - (pointerY / ZOOM_FACTOR));

viewport.setViewPosition(new
Point(adjustViewportX, adjustViewportY));
```

Fig. 9. Code to adjust viewport to the mouse-clicked coordinate on image.

V. CLOUD- IMAGE DISPLACEMENT.

Demarcation can be done when the image is at its default size or after it has been scaled. At default size, demarcation task is accomplished as desired. However, clouds drawn after any scaling become unexpectedly. By right, the cloud should be drawn within perimeter of mouse-clicks, as indicated by the grey rectangle in Fig. 10.

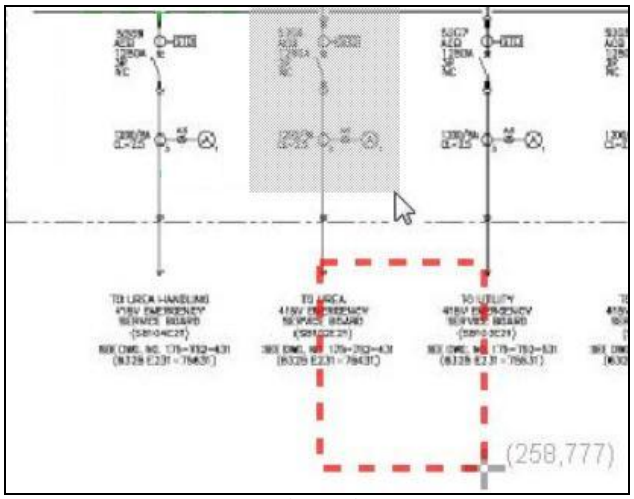

Fig. 10. Displacement of cloud from intended positions.

The displacement is caused by lack of calibration of coordinate systems in Java graphics system. In this work's implementation, two coordinate systems are used. JFrame container (container) and BufferedImage (image) each contains its own coordinate system [7,8]. Referencing two coordinate systems pose a grave problem because both do not share the same origin. The mouse coordinate is based on coordinate system of JFrame whilst Graphics2D object (which draws graphics) is based on coordinate system of image canvas, as depicted in Fig. 11.

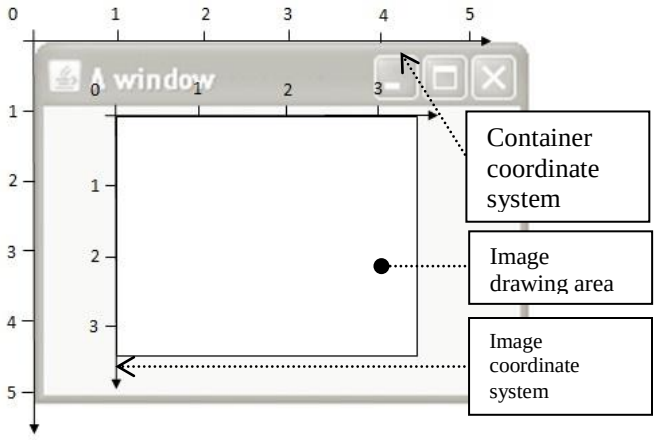

Fig. 11. Container coordinate system and image coordinate system.

Characteristics of displacement are reported in Table 2.

TABLE 2
TASK OBSERVATION

Task	Observation
Clouding at default size.	Cloud is drawn correctly within perimeter coordinates pointed by the mouse pointer.
Zoom 1.5x, then redraw similar cloud.	Instead of appearing within perimeter clicked using mouse pointer, cloud is displaced, and then enlarged disproportionately. The displacement distance from actual point is consistent for all new clouds drawn after a single magnification.
Zoom 3x, then redraw similar cloud.	Displacement distance increases. The displacement distance that occurs after 3x zoom is twice longer than displacement that occurs after 1.5x zoom.
Repeat with different zoom factors.	Subsequent tests reveal a pattern that displacement lengthens coherently with magnification ratio.

To address differences of both coordinate systems, the container coordinates registered by mouse pointer are mapped to the image coordinates. This can be done by calculating the offsets, i.e. the distance between both origins using equations (1) and (2).

$$x_{offset} = (Width_{container} - Width_{image}) \div 2 \quad (1)$$

$$y_{offset} = (Height_{container} - Height_{image}) \div 2 \quad (2)$$

Adjusting the offsets alone does not prevent the displacement. Another factor is the scale size of the coordinate systems, which becomes different after scaling.

The second factor is not obvious because prior to scaling, default scale sizes of both coordinate systems are the same, as illustrated in Fig. 12. Image scaling increases or decreases the scale size of image coordinate system only. The unchanging scale size of the container's coordinate system eventually leads to discrepancy.

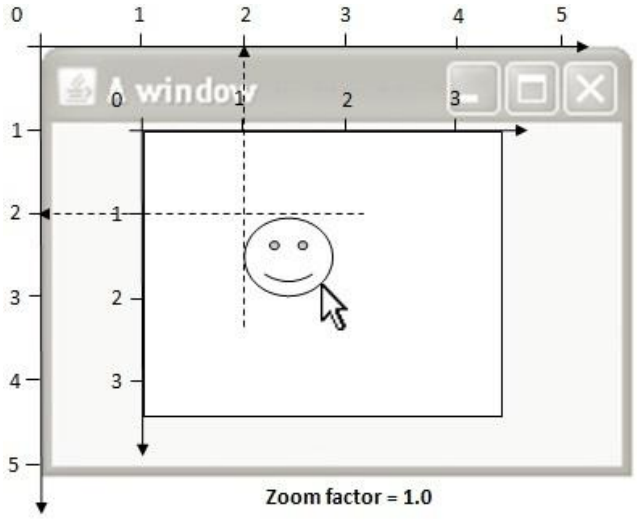

Fig. 12. Coordinate systems with uniform scale sizes.

An example of erroneous output is depicted in Fig. 13. A cloud, represented by a smiley shape, should have been drawn at coordinate (0.9, 0.9) but instead appears at position (2,2). Mouse-clicks by default register vertices of the drawing to coordinate system of container instead of image. On the other hand, Graphics2D object which renders graphics in Java refers to coordinates of the image, which by now already has its scale size enlarged.

Hence, calibration should be done by transforming any coordinate registered by mouse click event to coordinate system of the image. Calibration is done on arrays of x and y coordinates registered by mouse-clicks. The coordinates give the vertices of the cloud shape. Formulae to calibrate the coordinate points are given by equations (3) and (4).

$$x_{image} = (x_{container} - x_{offset}) \div zf \quad (3)$$

$$y_{image} = (y_{container} - y_{offset}) \div zf \quad (4)$$

where zf is zoom factor

(3) subtracts x-coordinate offset value from x coordinate returned by mouse event. The result is divided by zoom factor in order to factor out the scaling of the image coordinate. Similar steps are repeated for the y-coordinate in (4).

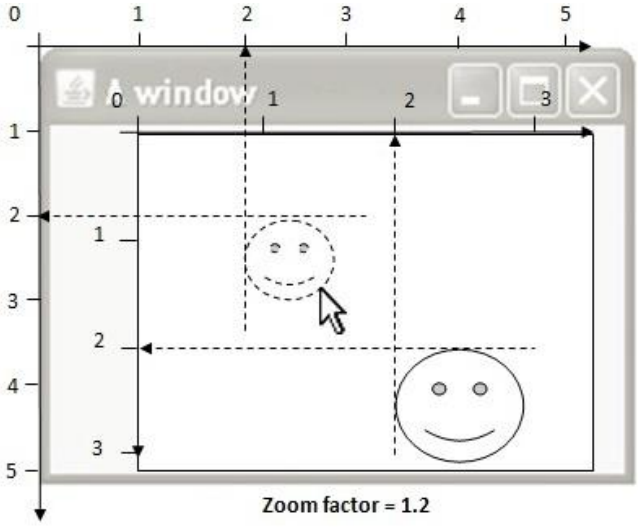

Fig. 13. Enlarged scale size of image coordinate.

Algorithm in Fig.14 applies equations (1), (2), (3) and (4) to resolve the displacement problem. In any mouse-click event, the algorithm is invoked to map mouse pointer coordinates (which refers to container's coordinate) to its corresponding image coordinates. The converted coordinates are stored in the memory of each cloud and used by Graphics2D object to draw clouds (as depicted in Fig. 4).

```
Input: container x-coordinate CX, container
y-coordinate CY, zoom factor ZF, container
width CW, container height CH, image width
IW and image height IH.

Output: an array A containing image x-
coordinate and image y-coordinate results.

RESOLVE-DISPLACEMENT(CX,CY)
1. Let A be an new 2-dimensional array.
2. Offset coordinate CX and CY
2.1   OFFSETX=CW/2 - IW/2
2.2   OFFSETY=CH/2 - IH/2
2.3   CX CX - OFFSETX
2.4   CY CY - OFFSETY
3. Calibrate unit scales of coordinates
3.1   A[0] CX/ZF
3.2   A[1] CY/ZF
4. Return A
```

Fig. 14. Algorithm used to correct displacement

6. RESULT AND DISCUSSION

Unit testing was conducted to evaluate the algorithm. A gridded image was loaded. A control experiment was carried out by leaving the image at its default size.

The demarcation was done by enclosing the letter "T" in both applications with or without the corrective algorithm as depicted in Fig. 15a and Fig. 15b. As expected, in both, the demarcation appeared correctly and exactly at where it was intended.

The second demarcation was done after zooming of the image, this time on letter "E". Fig. 15. (a) shows unwanted displacement of the bounding rectangle. Similar attempts were repeated on application with the corrective algorithm, as depicted in Fig. 15. (b). The result shows absence of displacement, suggesting that the algorithm effectively removes the displacement-due-to-zooming problem.

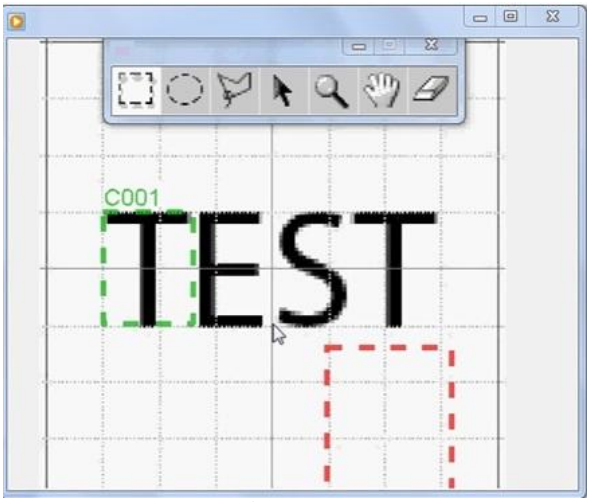

Fig. 15. (a) Images before corrective algorithm applied.

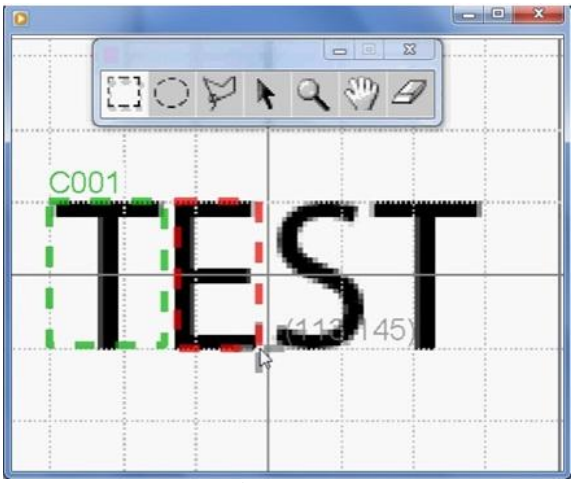

Fig. 15. (b) Images after corrective algorithm applied.

To confirm, similar testing was repeated with various images of various zooming degrees using all demarcation shapes available. All results were positive. Regardless of number of times the underlying raster image is scaled, the coordinate position of cloud relative to image is fixed.

In this study, an application which contains the algorithm was tested with 5 actual users to ascertain the correctness of the algorithm. We first demonstrates the use of all three variations (oval, rectangle and polygonal) of lasso tools for demarcation and magnification tool to resize the background. The users were then requested to repeat the attempts on sample engineering drawing, and reminded to maximize the usage of magnification tools. After several attempts, they were queried about what they felt about the demarcation tool: - correctness, ease of use, precision and sensitivity.

The algorithm does not exhibit significant appreciable delays during interactive execution and is not affected by shape or number of clouds. This is because the coordinates of the clouds are calibrated only once, that is only when drawing a new cloud. Then, the Graphics2D object automatically retrieves the already calibrated coordinates of each cloud to refresh the display.

The technique is useful if the coordinate system of mouse event is incongruent with the coordinate system of draw components. Since this algorithm is developed and tested for graphics environment in Java, it may not work without modification for graphics system with dissimilar architecture.

Although the technique is developed to solve a problem of a specific application, the solution is applicable to generic problems with similar attributes, i.e. coding that mixes both vector and raster graphics with scaling features. The technique has very low coupling with the rest of the code, hence can easily be implemented as a class method for reuse by other applications through a static class method invocation.

7. CONCLUSION

This work provides a simple yet useful technique to calibrate positioning of vector shapes on raster background due to disproportionate scale sizes of two or more coordinate systems and different coordinate origins. The problem can be resolved by first resolving origin offset and then multiplication with inverted zoom factor to convert the container coordinate to the image coordinate. The converted values are stored and to be used repeatedly by Graphics2D to draw clouds of any vector shapes correctly on any zoom factor. The algorithm prevents displacement and preserves relative sizes of the clouds.

Future work may investigate reusability of similar technique for implementation with similar vector overlaying raster drawings but occurring in a different graphics environment.

ACKNOWLEDGMENT

Authors thank Ir. Salmey Hassan of Petronas for sharing information about occupational safety and design safety; Mr. Fakhizan Romlie of Electrical and Electronics Engineering for assistance in Single-Line Diagram; Professor Dr. Jubair Al-Mujawar and Dr. Etienne Schneider for constructive comments.

REFERENCES

[1] Boose, et. al., "A scalable solution for integrating illustrated parts drawings into a Class IV Interactive Electronic Technical Manual," in *Document Analysis and Recognition*, 2003.

[2] Sonmez, A.I.,"Interactive computer aided drawing, manipulating, storing, retrieving and analyses of overall facility layout," in *Factory 2001- Integrating Information and Material Flow*, Second International Conference, Cambridge, 1990.

[3] Y. C. Kuo, "An Interactive Design System for Engineering Drawings," in *IEEE Computer Software and Applications Conference*, Proceedings. COMPSAC 79. The IEEE Computer Society's Third International, p.738-743, 1979.

[4] R. St. Amant, T. E. Horton, "A tool-based interactive drawing environment," in *ACM International Conference Proceeding Series*; Vol. 24 Proceedings of the 2nd International Symposium on Smart Graphics, p. 86 - 93, Hawthorne, New York, 2002.

[5] A. Cockburn., J. Savage and A. Wallace, "Tuning and Testing Scrolling Interfaces that Automatically Zoom," in *ACM CHI 2005*, Portland, Oregon, USA, 2005.

[6] A. Denault, J. Kienzle, "Avoid Common Pitfalls when Programming 2D Graphics in Java – Lessons Learnt from Implementing the Minueto Toolkit. ", in *ACM Crossroads* 13.3: Computer Graphics, p.100, 2007.

[7] Oracle. (2010, April 12). JavaTM 2 Platform Std. Ed. v1.4.2. [Online]. Available: http://download.oracle.com/javase/1.4.2/docs/api/java/awt/event/MouseEvent.html

[8] Oracle. (2010, April 12). JavaTM 2 Platform Std. Ed. v1.4.2. [Online]. Available: http://download.oracle.com/javase/1.5.0/docs/api/java/awt/image/BufferedImage.html