JDOR - Java2D for ooRexx (and Other Programming Languages)

Rony G. Flatscher
Department of Information Systems and Operations Management
WU (Wirtschaftsuniversität)
Vienna Austria
Rony.Flatscher@wu.ac.at

ABSTRACT

JDOR (Java2D for ooRexx) is a Rexx command handler that allows using simple string commands to create Java2D graphics and animations. The tool is part of the open source BSF4ooRexx850 package, a bidirectional ooRexx-Java bridge, and allows in addition for recording JDOR commands and to replay them later one by one. This allows for creating Java2D graphics and running simple Java2D animations from plain text files such that programs in any programming language can take advantage of JDOR.

The design and implementation of JDOR simplifies the interface to the Java2D classes considerably such that even students without any professional graphics background (like business administration students) can successfully take advantage of the tool to create even complex Java2D graphics and Java2D based animations which would be otherwise impossible for them as they lack the necessary Java programming and graphical skills.

Video: https://zenodo.org/record/8003114

CCS CONCEPTS

• Computing methodologies~Computer graphics • Computing methodologies~Image manipulation • Computing methodologies~Procedural animation • Applied computing~Education • Software and its engineering~Scripting languages • Software and its engineering~Command and control languages • Software and its engineering~Object oriented languages • Information systems~Open source software

KEYWORDS

Java2D, commands, graphics, tool, JDOR, object-oriented programming, ooRexx, Java, BSF4ooRexx850, BSF4ooRexx

1 Introduction

Graphics can be created in Java since its first version using classes from the *java.awt* package, most notably *java.awt.Graphics*. In Java 1.2 an improved version, the subclass *java.awt.Graphics2D*, got introduced with related classes, like those implementing the *java.awt.Shape* interface, which are subsumed under the name "Java2D". Java2D got exploited in order to implement the *javax.swing* controls which were introduced with Java 1.2 as well. [1] gives an overview of Java2D with links that allow to get acquainted with Java2D, [2] introduces and demonstrates many important aspects of Java2D.

ooRexx [3] is the open-source version of IBM's Object Rexx, which was devised as the object-oriented successor to IBM's Rexx programming language. ooRexx is released and maintained by the non-profit "Rexx Language Association (RexxLA)" [4]. BSF4ooRexx850 [5] is an open-source package that realizes a bidirectional ooRexx-Java bridge, making it possible to use the Java runtime environment and take advantage of all Java classes in it directly from ooRexx programs. Using BSF4ooRexx850 it becomes possible to exploit Java2D from ooRexx by directly using all Java2D classes.

The release of ooRexx 5.0 at the end of 2022 added the ability to the interpreter to install Rexx command handlers at runtime for the first time. BSF4ooRexx850 takes advantage of the new ooRexx 5.0 native APIs and implemented the ability to create Rexx command handlers in Java rather than in C++. As a proof of concept a core version of JDOR (Java2D for ooRexx) got implemented and reported in the fall of 2022 at the International Rexx symposium [6]. This experiment showed that it is indeed possible to define Rexx commands (strings) that will allow for exploiting Java2D in a much simpler manner than is possible compared to directly interacting and employing the respective Java2D classes from ooRexx. In the months that followed JDOR was enhanced to cover all of the functionality of the "*java.awt.Graphics2D*" class, and the Java classes implementing the "*java.awt.Shape*" interface and named with a trailing "2D" like "*java.awt.geom.Rectangle2D*" and including "*java.awt.geom.Path2D*". JDOR includes the ability to log all JDOR commands such that one can save them in a plain text file that can then be used to recreate the graphics by rerunning these commands with the JDOR Rexx command handler.

In addition JDOR defines commands for displaying the Java2D graphic in a window, moving and hiding such windows, loading from and saving to files, as well as printing them. This way JDOR can be used to create and run Java2D macros where the commands can be created in any programming language. Given the ability to display the Java2D graphic in real time and the ability to delay changes in the displayed Java2D graphic allows for creating animation effects in a simple and easy to use manner. This way JDOR can be regarded as a tool that allows for creating and animating Java2D graphics with Rexx commands which are plain strings.

2 ooRexx and Rexx Commands

ooRexx (open object Rexx) has been originally developed by IBM under the name "Object Rexx" as an object-oriented successor to IBM's Rexx language. After negotiations with the non-profit "Rexx Language Association", IBM handed over the source code for RexxLA to open source and maintain it. ooRexx is designed to run Rexx programs unchanged, yet adds the ability to define classes and implements the message paradigm which is used exclusively for interacting with Rexx objects.

ooRexx is a dynamic language which is caseless, dynamically typed and dispatches messages dynamically. There are no reserved keywords and the language supports four instruction types: assignment instructions, keyword instructions, command instructions and directive instructions. These properties of the language make it easy to apprehend for novices and has been successfully used to teach the foundations of object-oriented programming to business information system students but interestingly also to interested business administration students in a single semester [7].

Figure 1 displays an ooRexx program that demonstrates the four instruction types, its output is given in Figure 2. ooRexx will process programs in three phases: the loading and syntax check phase 1, the setup phase 2 in which directive instructions get carried out by the interpreter, and the execution phase 3 in which a program gets executed starting with the first instruction at the top of the program. The setup phase 2 will process the routine directive instruction and the routine named "*pp*" will be created which encloses the first argument in square brackets. The execution phase 3 causes the assignment of the value "*Hello world*" to a variable named "*a*", then repeats a loop three times in which each time a string gets created using the current values of the variables "*a*" and "*i*" and using the "*say*" keyword instruction to output the string value to the console (*stdout*). Finally a Rexx command instruction gets demonstrated: if an instruction is neither an assignment, a keyword or a directive, then it is a command instruction.

```
a="Hello world"          /* assignment   */
do i=1 to 3                /* keyword   */
   say "... #" i":" a      /* keyword   */
end                        /* keyword   */
cmd="echo" a "..."       /* assignment   */
say "cmd:" pp(cmd)         /* keyword   */
cmd      /* command (has return code)   */
say "return code:" pp(RC)  /* keyword   */

::routine pp             /* directive    */
  return "["arg(1)"]"      /* keyword   */
```

Figure 1: Four ooRexx instruction types

```
... # 1: Hello world
... # 2: Hello world
... # 3: Hello world
cmd: [echo Hello world ...]
Hello world ...
return code: [0]
```

Figure 2: Output of program in Figure 1

A Rexx command consists of a string that can be given as a literal, as an expression yielding a string or if a variable is supplied its value gets evaluated to a string and used as the command. The command then will by default be addressed to the operating system environment by the interpreter. Upon return of the command its return code will be assigned to a Rexx variable named "*RC*" by the interpreter and can be immediately used in the program. In Figure 1 the variable "*cmd*" gets the result of the expression assigned to it which refers to the string "*echo Hello world ...*". The command instruction "cmd" will yield its string value which gets sent to the system' shell for execution which will output "*Hello world ...*" to the console. Upon return from the "*echo*" command the Rexx "*say*" keyword instruction displays the value of the return code using the Rexx variable "*RC*" which in this case yields "*0*" indicating that the command was executed without any problems.

By default Rexx commands get used to address the operating system for execution. However, Rexx command handlers can be freely programmed and it is possible to define different command handlers at the same time to process command instructions. To address the appropriate command handler there is the Rexx keyword instruction "*address*" that names the command handler environment that is to execute the command.

ooRexx 5.0 implemented the optional ANSI Rexx [8] feature of the Rexx "*address*" keyword instruction that allows for redirecting input ("*stdin*"), output ("*stdout*") and error output ("*stderr*") while a command gets executed. If a command is written as a filter program that reads from "*stdin*" and writes to "*stdout*" or "*stderr*" then it would be possible with such a redirection to feed the command with "*stdin*" data supplied directly by ooRexx programs using, e.g., arrays, and output to ooRexx objects serving as the redirection targets for "*stdout*" and "*stderr*".

Figure 3 defines three command instructions, Figure 4 shows the output of the program on the console: the first "*echo*" command gets implicitly sent to the operating system shell, the second "*echo*" command uses explicitly the "*address*" keyword instruction to direct the command to the "*system*" shell. The "*sort*" command will be sent to the system shell with input ("*stdin*") redirected from the ooRexx array named "*ua*" of unsorted names and the output (*stdout*) redirected to the ooRexx array named "*sa*". Finally, both array objects get displayed as comma-blank separated strings such that one can see the unsorted names fed to the system's shell "*sort*" command as input via "*stdin*" and the resulting sorted names fetched into the "*sa*" array from "*stdout*".

```
"echo hello world 1"          -- by default addresses system
say "--> echo's return code:" rc
address system "echo hello world  2"   -- explicit addresss
say "--> echo's return code:" rc
ua="John", "Hans", "Alicia", "Xaver", "Josep"   -- unsorted
sa=.array~new  -- array to receive the sorted names
ADDRESS system "sort" WITH INPUT USING (ua) OUTPUT USING (sa)
say "--> sort's return code:" rc
say "unsorted names:" ua~makeString("L",", ")
say "sorted names:  " sa~makeString("L",", ")
```

Figure 3: Using sort command for sorting Rexx arrays

```
hello world 1
--> echo's return code: 0
hello world  2
--> echo's return code: 0
--> sort's return code: 0
unsorted names: John, Hans, Alicia, Xaver, Josep
sorted names:   Alicia, Hans, John, Josep, Xaver
```

Figure 4: Output of program in Figure 3

3 BSF4ooRexx850 and Rexx Command Handlers

BSF4ooRexx850 defines external Rexx functions that are implemented in native code, i.e. C++ for ooRexx' and Java's native interface, and defines utility routines and utility classes in ooRexx and in Java. The accompanying ooRexx program "*BSF.CLS*" is an ooRexx package of routines and classes that among other features supplies "ooRexx camouflaging support" in the form of an ooRexx proxy class for Java classes named "*BSF*" which can be used to reference any Java object as if it was an ooRexx object. Consequently this allows for sending ooRexx messages to Java objects.

Figure 5 uses the requires directive instruction which will be carried out in the setup phase 2 to make all public routines and public classes of the ooRexx package "*BSF.CLS*" available in the execution phase 3 to the program.

```
d=.bsf~new("java.awt.Dimension",111,222)
say "d:          " d  -- displays object's name
say "d~toString:" d~toString  -- send message to Java
say                -- new line
say "... d~setSize(333,444):"
d~setSize(333,444)            -- send message to Java
say "d~toString:" d~toString  -- send message to Java
say                -- new line
say "... Java fields as if they were Rexx attributes:"
d~width =555       -- camouflaged as Rexx attribute
d~height=666       -- camouflaged as Rexx attribute
say "d~toString:" d~toString  -- send message to Java
say "            d~width:" d~width "d~height:" d~height

::requires "BSF.CLS"     -- get ooRexx-Java bridge
```

Figure 5: Communicating with Java objects

```
d:          java.awt.Dimension@3d012ddd
d~toString: java.awt.Dimension[width=111,height=222]

... d~setSize(333,444):
d~toString: java.awt.Dimension[width=333,height=444]

... Java fields as if they were Rexx attributes:
d~toString: java.awt.Dimension[width=555,height=666]
            d~width: 555 d~height: 666
```

Figure 6: Output of program in Figure 5

The program uses the ooRexx proxy class "*BSF*" from the package "*BSF.CLS*" to create an instance of the Java class named "*java.awt.Dimension*" supplying a width of *111* and a height of *222* pixels. The ooRexx proxy object will get the Java object's default string value assigned as its object name which as a result gets displayed to the console (cf. "*java.awt.Dimension@ 3d012ddd*" in the first line in Figure 6). The "*toString*" message will return the Java string rendering of the Java object, the "*setSize*" message and its arguments will change the *Dimension* object fields accordingly as can be seen from the result of the next "*toString*" message.

The BSF4ooRexx850 Java classes allow for implementing Rexx command handlers in Java quite easily and get demonstrated with the programs in the subdirectories of "*BSF4ooRexx850*" installation directory "*BSF4ooRexx850/samples/Java/handlers/ commandHandlers*". The subdirectories with names ending in "850" contain the samples that demonstrate how to take advantage of this new feature. It is possible for Java programs to implement and define different Rexx command handlers and preconfigure the Java Rexx engine such that Rexx scripts that get executed can address these different Rexx command handlers directly. It would even be possible to exploit this infrastructure and implement Rexx command handlers directly in ooRexx exploiting BSF4ooRexx850 for this purpose which opens up new options and possibilities.

Once a Rexx command handler has been implemented and can be loaded using BSF4ooRexx850 the Rexx programmer needs to create an instance of such a Rexx command handler class and register it with the ooRexx runtime system supplying a name that needs to be used with the "address" keyword instruction in order to have the command directed to that particular Rexx command handler for execution.

Figure 7 demonstrates the necessary statements at the top: first an instance of the JDOR Rexx command handler gets created and assigned to the variable "*jdh*" and then the external Rexx function named "*BsfCommandHandler*" gets invoked to add this Java command handler "*jdh*" under the address name "*JDOR*" to the ooRexx environment. The program in Figure 7 will then use the "*address*" keyword statement to define the default environment for commands to be "*JDOR*" such that all commands by default will get directed at JDOR instead of the system shell. As in this case the output redirection gets used, the JDOR handler will output each executed JDOR command in a canonical form.

```
jdh=.bsf~new("org.oorexx.handlers.jdor.JavaDrawingHandler")
call BsfCommandHandler "add", "JDOR", jdh -- add handler
address JDOR with output using (.output)  -- set JDOR handler

newImage 200 200          -- create image width=200, height=200
winShow                   -- show window
fn1='rexxla.png'          -- filename
loadImage img1 fn1        -- load image, RC gets its dimension
say "--" fn1": RC=["RC"]"    -- show file name and RC
parse var RC cbsW cbsH    -- parse RC to get width and height
moveTo 16 25              -- set location to x=16 y=25
drawImage img1 (cbsW/7) (cbsH/7)    -- resize image to 1/7
fn2='bsf4oorexx_256.png' -- filename (used also as nickname)
loadImage img2 fn2        -- load image, RC gets its dimension
say "--" fn2": RC=["RC"]"    -- show file name, RC
parse var RC ptW ptH      -- parse RC to get width and height
moveTo 75 145             -- set location to x=75 y=145
drawImage img2 (ptW/5) (ptH/5)  -- resize image to 1/5
saveImage "20_images.png"    -- save image to file
sleep 3                   -- sleep a bit
parse pull .              -- user needs to press <enter>

::requires "BSF.CLS"      -- get ooRexx-Java bridge
```

Figure 7: Addressing JDOR command handler

4 JDOR - Java2D for ooRexx

"Java2D" is a term that was introduced with Java 1.2 in the context of the new Java class "*java.awt.Graphics2D*" which extends the original "*java.awt.Graphics*" class. Together with it classes that implement "*java.awt.Shape*" like "*java.awt.geom. Rectangle2D*" became part of the Java runtime environment and existing classes like "*java.awt.Polygon*" added the Shape interface for Java 1.2. Controls in the "*javax.swing*" package introduced with Java 1.2 use Java2D to draw the controls which among other things allowed for adding skinnability ("pluggable look and feel (PLAF)"), which allows for custom drawing all the swing controls [9] if desired. In Java 6 Java2D got enhanced most notably with the addition of "*java.awt.geom.Path2D*" allowing for creating free form shapes.

In general Java2D allows for creating 2D graphics of any complexity in Java which can be exploited for different application needs including creating Java games [10]. In order to take advantage of Java2D one needs to be aware of all the classes that are spread over different Java packages and how these classes play together. If color or fonts get used one needs to become aware of the "*java.awt.Color*" and "*java.awt.Font*" classes in order to take advantage of them, if one wishes to display the graphics on screen and control display properties or print the Java2D images one needs to get acquainted with all the appropriate methods of the appropriate Java classes.

With BSF4ooRexx850 it becomes possible to take advantage of all of the Java2D classes from ooRexx and employ them for creating any image of any complexity in ooRexx that can also be created with Java. However, it may be the case that one wants to take advantage of Java2D but wishes to use a simpler interface to it than the Java2D classes with their methods.

This is where the idea to define a set of commands comes into play, that would simplify the creation of Java2D graphics as much as possible. Ideally ooRexx programmers would become able to take advantage of it to create images but would have no need to learn the programming language Java to do so. These commands should be realized therefore by implementing a Rexx command handler in Java which would be able to process the received commands and carry out the appropriate Java2D operations using the relevant Java2D classes and methods.

A first prototype got developed and named "JDOR", "Java2D for ooRexx", and demonstrated at the International Rexx symposium in the fall of 2022 [6] exploiting the new support for direct Rexx command handlers in BSF4ooRexx850. The prototype has been extended since to become a full implementation of Java2D. One interesting design goal was to also allow for ooRexx programmers to get access to the Java2D objects in case they wished to directly interact with them. This has been made possible in the JDOR Rexx command handler implementation by returning the Java proxies via the "*RC*" variable that gets set upon return of executing commands. One notable addition in this endeavor to complete the Java2D implementation in form of JDOR Rexx commands has been creating new commands that supply support for "*java.awt.Shape*" classes and the "*java.awt.geom.Path2D*" class explicitly.

Taking advantage of the redirection ability of Rexx command handlers the JDOR command handler will take advantage of redirection as well and if at runtime it realizes that the command's output got redirected it will write the processed JDOR command with all arguments in a canoncial format to it. This makes it possible to display and to record the JDOR commands for a specific image and later use these very same commands to recreate the image at a later time by sending these commands to a new JDOR command handling program. This way JDOR commands can be stored in text files, created manually or by programs in any programming language, and then used as a macro to (re-)create the desired image with Java2D.

As any program can create strings it becomes possible that any program can produce JDOR commands which then get merely executed by any "JDOR tool" which is available whenever ooRexx 5.0 with BSF4ooRexx850 is available.

The JDOR commands are documented in the BSF4ooRexx850 installation directory "*BSF4ooRexx850/information/jdor/jdor_doc.html*" and due to their naming should be apprehensible for anyone with a basic education in geometrics and more so for Java programmers who know Java2D. The documentation contains links that will also refer to the JavaDocs for the respective Java2D classes such that interested readers can get directly at the classes that get used for implementing the respective JDOR command.

The following sections will explain and demonstrate how JDOR commands can be used to load, draw and save images with a few commands and without a need to know the documentation of the Java2D classes that get used to execute the commands. The examples should be understandable even if there is no prior knowledge of ooRexx, BSF4ooRexx and of JDOR commands, as they appear almost as if they were formulated in pseudo code to communicate the structure and sequence of JDOR commands and have line comments.

4.1 Executing JDOR Commands

Figure 7 above depicts an ooRexx program that creates and registers a JDOR command handler using "*JDOR*" as its environment name. Then the "*address*" keyword instruction gets used to set the JDOR command environment as the default environment such that any commands get directed at it. The program loads two images from the current directory, fetches their dimensions by parsing the returned value using the "*RC*" variable, reduces their width and height to 1/10th, respectively to 1/7th of their original size and then draws them on the Java2D image at different locations and finally saves the new image under the name "*20_images.png*" in the current directory.

Due to the characteristics of the ooRexx language the code in Figure 7 looks almost like pseudo code and it is hoped that the name and arguments to the JDOR commands therein are self-explanatory such that no detailed explanations beyond the sup-

plied line comments are needed to understand what the program does. If this is the case it should serve as a proof that indeed one can understand Java2D programs without knowing any details of the Java2D classes that get employed in order to come up with the resulting image in Figure 8.

```
newImage 200 200
winShow
loadImage IMG1 rexxla.png
-- rexxla.png: RC=[1200 927]
moveTo 16 25
drawImage IMG1 171 132
loadImage IMG2 bsf4oorexx_256.png
-- bsf4oorexx_256.png: RC=[256 256]
moveTo 75 145
drawImage IMG2 51 51
saveImage 20_images.png
sleep 3.0
```

Figure 8: Output and image by program in Figure 7

Figure 8 displays the output of running the program in Figure 7 together with the produced image and if redirecting this output into a file one can use the JDOR commands as input to a JDOR filter program like that shown in Figure 9 ("*jdor.rex*"). If the commands depicted in the background of Figure 8 were saved into a text file named "*commands.txt*", then one can recreate the image by running the filter with "*rexx jdor.rex < commands.txt*". Alternatively, one could use the filter in a pipe, e.g., "*cat commands.txt | rexx jdor.rex*" which allows JDOR to become usable by any program and application.

```
jdh=.bsf~new("org.oorexx.handlers.jdor.JavaDrawingHandler")
call BsfCommandHandler "add", "JDOR", jdh -- add handler
address jdor with input using (.stdin)     -- from stdin
"-- comment"           -- kick off handler

::requires "BSF.CLS"     -- get ooRexx-Java bridge
```

Figure 9: Filter program "*jdor.rex*"

4.2 Affine Transform JDOR Demo

In the Java based tutorial on programming Java games with Java2D [10] there is a section that demonstrates Java2D's affine transformations employing a polygon shape [11]. This is done by creating a "*java.awt.Polygon*" which gets drawn in original size at the default location "*0,0*" (left upperhand corner) by filling it with a green color. According to the polygon's coordinates part of it does not get shown. Then the scaling (an affine transform) gets increased by 20% affecting all Java2D operations that follow. Using the "*translate*" operation the origin gets moved right and down by 50 pixels, the polygon gets drawn again in green and then in a loop the origin gets translated 50 pixels to the right and 5 pixels to the bottom, the polygon gets drawn again by filling it in blue, rotated by 15 degrees (an affine transform) and drawn over it once more by filling it in red.

Figure 10 depicts a program that creates the same image using JDOR commands, the resulting image is shown in Figure 12. It is equivalent to the Java program in [11] shown in Figure 11 which allows for comparing it with the Java solution and also comparing the involved complexity in creating such an image exploiting Java2D with Java and with JDOR commands.

```
jdh=.bsf~new("org.oorexx.handlers.jdor.JavaDrawingHandler")
call BsfCommandHandler "add", "JDOR", jdh -- add  handler
address JDOR       -- set JDOR as default environment
newImage 640 480   -- JDOR command to create new image
winShow            -- show image in a window
winTitle "Affine Transform Demo (ooRexx)" -- set title
polygonXs="(-20,0,+20,0)"   -- define four x coordinates
polygonYs="(20,10,20,-20)" -- define four y coordinates
shape myP polygon polygonXs polygonYs 4    -- create polygon
color green        -- set color to green
fillShape myP      -- fill (and show) the polygon shape
translate 50 50    -- move origin (x=x+50, y=y+50)
scale 1.2 1.2      -- increase scale symmetrically by 20%
fillShape myP      -- fill (and show) the polygon shape
do 5               -- repeat five times
   translate 50 5 -- move origin (x=x+50, y=y+5)
   color blue      -- set color to blue
   fillShape myP   -- fill (and show) the polygon shape
   rotate 15       -- rotate by 15°
   color red       -- set color to red
   fillShape myP   -- fill (and show) the polygon shape
end
sleep 3            -- sleep a bit

::requires "BSF.CLS"     -- get ooRexx-Java bridge
```

Figure 10: JDOR solution of Java program in Figure 11

```
import java.awt.*;
import java.awt.geom.AffineTransform;
import javax.swing.*;

/** Test applying affine transform on vector graphics */
@SuppressWarnings("serial")
public class AffineTransformDemo extends JPanel {
   // Named-constants for dimensions
   public static final int CANVAS_WIDTH = 640;
   public static final int CANVAS_HEIGHT = 480;
   public static final String TITLE = "Affine Transform Demo";
   // Define an arrow shape using a polygon centered at (0, 0)
   int[] polygonXs = { -20, 0, +20, 0};
   int[] polygonYs = { 20, 10, 20, -20};
   Shape shape = new Polygon(polygonXs, polygonYs, polygonXs.length);
   double x = 50.0, y = 50.0;  // (x, y) position of this Shape
   /** Constructor to set up the GUI components */
   public AffineTransformDemo() {
      setPreferredSize(new Dimension(CANVAS_WIDTH, CANVAS_HEIGHT));
   }
   /** Custom painting codes on this JPanel */
   @Override
   public void paintComponent(Graphics g) {
      super.paintComponent(g);    // paint background
      setBackground(Color.WHITE);
      Graphics2D g2d = (Graphics2D)g;
      // Save the current transform of the graphics contexts.
      AffineTransform saveTransform = g2d.getTransform();
      // Create a identity affine transform, and apply to the Graphics2D context
      AffineTransform identity = new AffineTransform();
      g2d.setTransform(identity);
      // Paint Shape (with identity transform), centered at (0, 0) as defined.
      g2d.setColor(Color.GREEN);
      g2d.fill(shape);
      // Translate to the initial (x, y) position, scale, and paint
      g2d.translate(x, y);
      g2d.scale(1.2, 1.2);
      g2d.fill(shape);
      // Try more transforms
      for (int i = 0; i < 5; ++i) {
         g2d.translate(50.0, 5.0);  // translates by (50, 5)
         g2d.setColor(Color.BLUE);
         g2d.fill(shape);
         g2d.rotate(Math.toRadians(15.0)); // rotates about transformed origin
         g2d.setColor(Color.RED);
         g2d.fill(shape);
      }
      // Restore original transform before returning
      g2d.setTransform(saveTransform);
   }
   /** The Entry main method */
   public static void main(String[] args) {
      // Run the GUI codes on the Event-Dispatching thread for thread safety
      SwingUtilities.invokeLater(new Runnable() {
         @Override
         public void run() {
            JFrame frame = new JFrame(TITLE);
            frame.setContentPane(new AffineTransformDemo());
            frame.pack();
            frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
            frame.setLocationRelativeTo(null); // center the application window
            frame.setVisible(true);
         }
      });
   }
}
```

Figure 11: AffineTransform Java program from [11]

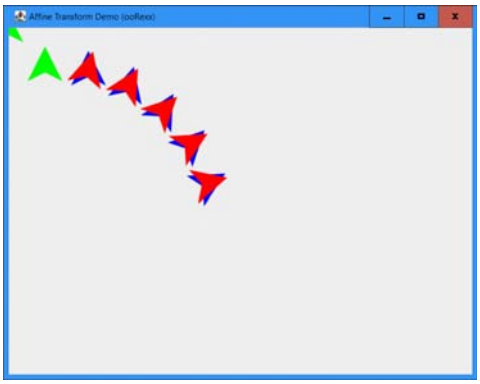

Figure 12: Image by program in Figure 10

5 Roundup

This article introduced the JDOR Rexx command handler which has been implemented in BSF4ooRexx850 and which can be exploited by Rexx programs by using JDOR commands. In order to understand the tool the constituting software components, the ooRexx programming language and the BSF4ooRexx850 ooRexx-Java bridge got briefly characterized such that it should have become possible to relate to Rexx commands and Rexx command handlers implemented in Java.

Then Java2D has been briefly characterized and links got supplied that allow for a quick introduction and study of its architecture as well as the related Java2D classes that allow for creating 2-dimensional graphics in Java. BSF4ooRexx850 includes a Rexx command handler implemented in Java for the purpose of making the Java2D functionality available as simple Rexx commands named "*JDOR*" ("Java2D for ooRexx"). The JDOR documentation comes with BSF4ooRexx850 and can be studied at [12]. Two JDOR examples demonstrate how Java2D can be used with JDOR examples, one (Figure 7) loading two images that get rescaled and drawn on a JDOR image and saved into a file, one example (Figure 10) realizing the JDOR counterpart of a Java2D program that attempts to demonstrate applying affinity transforms.

The JDOR command handler is implemented as a redirectable Rexx command handler such that *stdin* and *stdout* can be redirected. Together with a simple JDOR filter program (Figure 9) it becomes possible to use JDOR commands to create Java2D graphics using redirections (redirecting the JDOR's filter *stdin* from a file, or via pipes that feed the JDOR filter's *stdin*). As a consequence it is possible to use JDOR commands in combination with a JDOR filter program to create Java2D graphics of any complexity from any program and application.

The implementation of the JDOR command handler is regarded to be feature complete. If Java2D features are sought but not made available via JDOR commands it would still be possible to supply them in ooRexx programs by interacting with the JDOR command handler's infrastructure, e.g., fetching the *java.awt.Graphics2D* Java object with the JDOR command "*gc*" and then interact directly with it from ooRexx.

ACKNOWLEDGMENTS

The author wishes to thank DI Walter Pachl and Dr. Till Winkler for their valuable feedback and help.

REFERENCES

[1] Java 2D Graphics and Imaging. Retrieved January 31, 2024 from https://docs.oracle.com/javase/6/docs/technotes/guides/2d/
[2] The Java Tutorials: Trail: 2D Graphics. Retrieved January 31, 2024 from https://docs.oracle.com/javase/tutorial/2d/TOC.html
[3] ooRexx. 2024. ooRexx (Open Object Rexx). Retrieved January 31, 2024 from https://sourceforge.net/projects/oorexx/
[4] RexxLA. 2023. The Rexx Language Association. Retrieved January 31, 2024 from https://www.RexxLA.org
[5] BSF4ooRexx850. 2024. Bean scripting framework for ooRexx. Retrieved January 31, 2024 from https://sourceforge.net/projects/bsf4oorexx/files/beta/20240109/
[6] BSF4ooRexx: Introducing the JDOR Rexx Command Handler for Easy Creation of Bitmaps and Bitmap Manipulations on Windows, Mac and Linux. 2022. Retrieved January 31, 2024 from https://www.rexxla.org/presentations/2022/202209_JDOR_command_handler.pdf
[7] Flatscher G. Rony and Müller Günter. 2021. "Business Programming" – Critical Factors from Zero to Portable GUI Programming in Four Hours. In 6th Business and Entrepreneurial Economics 2021 - Conference Proceedings. 76-82. Retrieved January 31, 2024 from https://research.wu.ac.at/files/32933846/2021_Flatscher_Mueller_BusinessProgramming_from_proceedings.pdf
[8] ANSI Rexx standard. Retrieved January 31, 2024 from https://www.rexxla.org/rexxlang/standards/j18pub.pdf
[9] The Java Tutorials: Modifying the Look and Feel. Retrieved January 31, 2024 from https://docs.oracle.com/javase/tutorial/uiswing/lookandfeel/index.html
[10] Chua Hock-Chuan. 2012. Java Game Programming 2D Graphics, Java2D and Images. Retrieved January 31, 2024 from https://www3.ntu.edu.sg/home/ehchua/programming/java/J8b_Game_2DGraphics.html
[11] Chua Hock-Chuan. 2012. Java Game Programming 2D Graphics, Java2D and Images, 2.2 Affine Transform (java.awt.geom.AffineTransform). Retrieved January 31, 2024 from https://www3.ntu.edu.sg/home/ehchua/programming/java/J8b_Game_2DGraphics.html#zz-2.2
[12] Documentation of the JDOR Commands. 2023. Retrieved January 31, 2024 from https://wi.wu.ac.at/rgf/rexx/misc/jdor_doc.tmp/jdor_doc.html