Seminararbeit

SBWL: Business Information Systems

im **SS 2023**

BSF4ooRexx850 JDOR: Java 2D Drawing for ooRexx

Elif Deger h11919438

PI-Leitung

ao.Univ.Prof. Dr. Rony G. Flatscher

Wien, 14.06.2023

Inhaltsverzeichnis

Abstract

This seminar paper showcases the application of the latest BSF4ooRexx850 extension called JDOR in the context of ooRexx programming to generate diverse images. The paper presents "Nutshell-Examples" to demonstrate the fundamental operations and their implementation using JDOR. ooRexx, is utilized along with the powerful BSF4ooRexx850 framework to leverage Java's extensive functionality while benefiting from the easy-to-understand syntax of ooRexx. With the help of JDOR, even programmers who have limited understanding of ooRexx and Java can create detailed and visually captivating drawings. This is particularly beneficial for those who are interested in simple graphic design.

1. Introduction

This paper explores Java 2D drawing in ooRexx, focusing on the powerful capabilities offered by the JDOR (Java 2D Drawing for ooRexx) software library. JDOR serves as a versatile tool for creating captivating 2D graphics in Java, and when combined with ooRexx and BSF4ooRexx850, it enables seamless integration of visually stunning graphics into Rexx scripts.

The paper begins by providing an introduction to Java 2D, laying the foundation for understanding the potential of JDOR in crafting engaging 2D graphics within the Java programming language. It explores the Java 2D API, which extends the Abstract Windowing Toolkit (AWT) and offers an extensive set of functionalities for graphic creation and manipulation. The integration of JDOR with the Java 2D API provides a user-friendly approach, allowing Rexx programmers to leverage the power of Java's graphics capabilities without the need for in-depth knowledge of Java syntax and structure.

Furthermore, the paper delves into the JDOR command handler, a Rexx command handler implemented in Java. This command handler simplifies the utilization of Java 2D for Rexx programmers, providing a set of commands that mirror the methods in the Java.awt.Graphics and Java.awt.Graphics2D classes. With JDOR, Rexx programmers can effortlessly perform tasks such as drawing shapes, lines, images, and text, setting colors, fonts, and strokes, and accessing the current state and data of JDOR for flexible graphic manipulation.

The paper concludes with practical programming examples that showcase the functionalities of JDOR in ooRexx. These examples serve as a foundation for developing more complex programs and unlocking the full potential of JDOR in creating captivating 2D graphics. By delving into Java 2D drawing in ooRexx through JDOR, users can unleash their artistic visions and bring them to life with ease and efficiency.

2. JDOR (Java 2D Drawing for ooRexx)

The introduction of BSF4ooRexx850 beta has simplified the implementation of Rexx command handlers in Java (Flatscher, 2023). An example of this is the JDOR (Java2D for ooRexx) Rexx command handler provided as part of the package (Flatscher, 2022). JDOR is a software library designed for creating 2D graphics in ooRexx.

JDOR provides a user-friendly and efficient approach to 2D graphic creation. With its intuitive interface and comprehensive tools, JDOR makes it easy to bring creativity to life without the usual complexities of graphic development. BSF4ooRexx850 enables the creation of sophisticated 2D graphics that respond to user input, adapt to changing conditions, and offers a captivating experience by smoothly working with Java objects.

The needed installations to start using JDOR can be found under Appendix A1 Installation Guide.

Before diving into creating drawings with JDOR, it is essential to establish a solid understanding of Java 2D. The following chapter serves as an introduction to Java 2D, laying the groundwork for the subsequent chapters. By familiarizing ourselves with Java 2D, we can fully grasp the capabilities and potential of JDOR for crafting captivating 2D graphics within the Java programming language.

3. Java Graphics Creation

Java offers a diverse range of tools and frameworks for programmers to develop graphics and graphical user interface (GUI) components. Many of these tools are encompassed within the Java Foundation Classes (JFC), which come pre-integrated with Java (Oracle, o.D -a).The graphics created in this seminar paper are created using the Java 2D API feature.

The Java 2D API serves as an extension of the Abstract Windowing Toolkit (AWT). It provides an extensive set of functionalities for graphic creation and manipulation. One notable aspect is the integration of the REXX command handler known as "JDOR" (Java Drawing for ooRexx). This enables programmers to harness the power of the Java 2D API's Graphics and Graphics2D classes within ooRexx, without the need for prior knowledge of Java's syntax and structure. This integration facilitates a seamless experience for developers, allowing them to leverage the capabilities of the Java 2D API within the ooRexx environment, thereby enhancing their ability to create visually appealing graphics. (Flatscher, 2022).

3.1 Abstract Windowing Toolkit (AWT)

In order to write a useful application, it is necessary to have a user interface (Holt, 1999). Abstract Windowing Toolkit (AWT) packages provide a set of classes to allow you to create a GUI interface using graphical components in Java programs (Cowell, 1999). Since the AWT has been a fixed part of the Java class hierarchy since the very first Java version 1.0, such graphical applications can run on all operating systems thanks to the portability of Java (Schäling, 2010).

Java AWT components are platform-dependent because components are displayed according to the view of the operating system. Java AWT calls Operating systems subroutine for creating components such as textbox, button, etc. An application built on AWT looks like a Windows application when it runs on Windows, but the same application would look like a Mac application when runs on Mac OS (https://dotnettutorials.net).

AWT features include; a set of native user interface components, a robust event-handling model, graphics and imaging tools, including shape, color, and font classes, layout managers, for flexible window layouts that do not depend on a particular window size or screen resolution, data transfer classes, for cut-and-paste through the native platform clipboard (Oracle, o.D -g).

3.2 Java 2D API

Java 2D API Enables developers to easily incorporate high-quality 2D graphics, text, and images in applications and applets. Java 2D includes extensive APIs for generating and sending high-quality output to printing devices (Oracle, o.D -g).

The Java 2D API provides two-dimensional graphics, text, and imaging capabilities for Java programs through extensions to the Abstract Windowing Toolkit (AWT) (Oracle, o.D.-f). Through the REXX command handler “JDOR” programmers have the capability to use elements from the Java 2D APIs “Graphics” and “Graphics2D” classes in ooRexx without prior knowledge of the syntax and structure of Java (Flatscher, 2022b).

The primary class in this API is the Graphics2D, which is a subclass of the Graphics class. Graphics2D provides uniform support and advanced control over 2D shapes, such as text, lines, and objects, compared to Graphics class (Oracle, o.D.-b).

The Java 2D API operates with two coordinate spaces: user space and device space. User space is a device-independent logical coordinate system used by your program to specify graphics primitives. All geometries provided to Java 2D rendering routines are defined in user-space coordinates. On the other hand, device space represents the coordinate system of the specific output device, such as a screen, window, or printer. The coordinate systems of different devices can vary significantly, but Java programs are shielded from these differences. The API automatically handles the necessary conversions between user space and device space during rendering, ensuring that graphics are accurately displayed regardless of the target device (Oracle, o.D.-c).

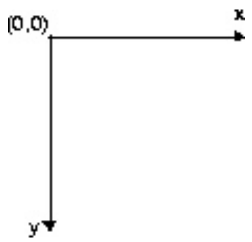

Figure 1 User Space Coordinate System

The Java 2D API offers three levels of configuration information to help convert from the device-independent user-space to the device-dependent device-space: GraphicsEnvironment, GraphicsDevice, and GraphicsConfiguration. GraphicsEnvironment provides a collection of all the rendering devices connected to the platform and a list of available fonts. GraphicsDevice describes a visible rendering device that can have multiple GraphicsConfigurations, which describe certain modes like 1920x1080 or 1280x720 (Blauensteiner, 2023).

The Java 2D API has a unified coordinate transformation model. All coordinate transformations, including transformations from user to device space, are represented by AffineTransform objects. AffineTransform defines the rules for manipulating coordinates using matrices (Sun-Microsystems, 1999).

You can add an AffineTransform to the graphics context to rotate, scale, translate, or shear a geometric shape, text, or image when it is rendered. The added transform is applied to any graphic object rendered in that context. The transform is performed when user space coordinates are converted to device space coordinates (Sun-Microsystems, 1999).

The API offers four basic transformation methods: “Translate”, “Rotate”, “Scale” and “Shear”. “Translate” moves the origin (x=0, y=0) of the graphics context to a new point, “Rotate” rotates a previously created object by a specified angle, “Scale” applies a multiplier to both axes for all the following commands, and “Shear” shifts or slants coordinates in one axis as a function of their second axis (Blauensteiner, 2023).

While the Java 2D API offers a number of complex methods for creating graphics, most programs only use a subset of the capabilities found in the Graphics class. Graphics methods can be divided into two groups: rendering basic shapes, texts, and images through the draw and fill methods and setting attributes to those basic drawings and fillings. These method groups can be combined to create a wide variety of graphics (Blauensteiner, 2023).

3.3 JDOR Command Handler

The JDOR is a Rexx command handler that serves the purpose of exploiting Java's awt 2D classes for graphics manipulation. This implementation, developed in Java using BSF4ooRexx850, provides various functionalities such as accessing, creating, and dropping Rexx variables within the caller's context. Its primary objective is to enable seamless integration with the Java awt graphics 2D subsystem. To ensure ease and simplicity for Rexx programmers, the JDOR adheres to the Rexx philosophy and offers well-thought-out commands and their corresponding arguments. Furthermore, it is essential to configure the Rexx interpreter to load and employ these specialized Rexx command handlers effectively. Through JDOR, Rexx programmers can effortlessly harness the power of Java's awt graphics 2D subsystem in their applications (Flatscher, 2022).

The Rexx command handler, implemented in Java, aims to simplify the utilization of Java2D for Rexx programmers without requiring direct usage of Java code. Its main purpose is to facilitate the exploitation of Java awt package's Graphics and Graphics2D drawing capabilities through a set of commands. These commands enable Rexx programmers to perform tasks such as drawing strings, lines, rectangles, ovals, images, and more, as well as setting colors, fonts, and strokes. Additionally, the command handler allows access to the current state and JDOR data, including the directories and HashMaps of loaded colors, fonts, and strokes. This provides the flexibility to define custom colors, fonts, and strokes from within the Rexx program and store them for future use. The command handler also provides features like temporary execution halt for animation purposes, easy saving and restoration of graphic configurations and image states at runtime, and effortless saving and loading of images. It further enables the recording and replaying of commands, effectively creating Rexx macros for Java 2D graphics, which can be stored even in plain text files (Flatscher, 2022).

The drawing area is a canvas with a specific width and height in pixels, where the origin (x=0, y=0) is positioned at the top left corner. The translate command allows for moving the canvas, and in this coordinate system, the x coordinate increases towards the right, while the y coordinate increases towards the bottom (Flatscher, 2022).

When using the Rexx command handler, the commands are structured based on the methods in the java.awt.Graphics and java.awt.Graphics2D Java classes. However, there is a crucial distinction in how the x and y coordinates are handled. In many Java methods, these coordinates are explicitly included as the first two arguments. In contrast, the Rexx programmer defines these coordinates using the moveTo x y command before executing other commands. Consequently, the Rexx commands, which mirror the Java method names, do not explicitly mention the x and y coordinates. Instead, they rely on the previously set positions for their values. This approach simplifies the Rexx commands and aligns them with the Java counterparts while offering flexibility and ease of use for Rexx programmers. (Blauensteiner, 2023).

Below is a table containing the JDOR commands used in this paper, along with their respective descriptions. The documentation of the JDOR Commands can be found in the BSF4ooRexx-folder with the following path:

/BSF4ooRexx850\information\jdor\jdor_doc.html

Command	**Description**
background	Sets the color of the background.
color *nickname*	Supplying only the *colorNickName* argument will load the color from the internal register or from a Rexx variable by that name referring to a color.
drawImage	Draws an image which got previously loaded from the filesystem with the command *loadImage* and stored internally with an *imageNickName* in the internal image registry.
drawLine *x y*	Draws a line from the current coordinates to the given coordinates.
drawOval *width height*	Draws an oval in an invisible rectangle from the current coordinates (upper- left) with the given *width* and *height.*
drawPolygon	Draws a polygon using *nPoints* coordinates from xPoints-array and yPoints-array .The polygon gets closed by drawing a line from the first and last point.
drawPolyline	Draws a polyline using *nPoints* coordinates from xPoints-array and yPoints-array.
drawRect *width height*	Draws a rectangle from the current coordinates (upper- left) with the given *width* and *height .*
drawString *text*	Draws a string (=text) at the current coordinates.

fillOval *width height*	Fills an oval in an invisible rectangle starting from the current coordinates (upper- left) with the given *width* and *height.*
fillPolygon	Fills a polygon using *nPoints* coordinates from xPoints-array and yPoints-array.
fillRect *width height*	Fills a rectangle starting from the current coordinates (upper-left) with the given *width* and *height.*
font *nickname*	Sets a previously saved font as the font for the following commands.
fontSize *size*	Sets the font size for the following commands.
fontStyle *style*	Sets the font style for the following commands. Style-attribute (0: Normal, 1: Bold, 2: Italic, 3: Bold+Italic).
goto *x y*	Sets the x1 and y1 coordinates for the following commands.
loadImage *nickname path*	Saves an image from the given path under the given nickname
Rotate *angle in degree*	Rotates the following drawing in the given theta (=angle in degree) around the origin of the coordinate system. „x "and „y"sets a new origin for the rotation.
saveImage	Saves the current image to a file.
Scale	Queries and optionally changes ("concatenates") the scale factor for the x and y axis.
Shear	Applies a factor that determines how much an object shifts in relation to its "x" and "y" coordinates.
Sleep	Sleeps (halts execution) for the given interval expressed in *seconds*.
Stroke *NickName width cap join miterlimit dashArray dashPhase*	Defines a new stroke of width in pixels, cap, join, miterlimit, dashArray, dashPhase, stores it in the internal registry with the uppercased strokeNickName and returns the previous stroke via the Rexx variable RC.
Transform	An *AffineTransform* defines a matrix that gets used to calculate the effective *x* and *y* values for the target device according to this formula: x' = translateX + scaleX*x + shearX*y y' = translateY + scaleY*y + shearY*x
Translate *x y*	Sets a new origin for the coordinate-system.
winShow	Shows the current window.
winSize *width height*	Sets the size (width and height) of a new window.

winTitle	Queries and optionally sets the title of the frame (window) that displays the current image.

Table 1: JDOR commands

4. JDOR Examples in ooRexx

In this chapter, practical programming examples to illustrate the functionalities of the JDOR package are provided. The examples build upon each other progressively, enhancing the capabilities introduced in the previous program. By following this approach, the chapter offers fundamental use-cases and practical guidance for effective command utilization. Programmers can use these examples as a foundation for developing more complex programs and unleash the full potential of the JDOR package.

To begin working with JDOR, the following code block should alyaws be executed:

```
call addjdorhandler    -- load and add the java rexx command handler, using default name: jdor
address jdor           -- set default environment to jdor
```

These instructions ensure that the JDOR package is properly loaded and set as the default environment for further operations.

4.1 Creating Text - JDOR-text.rxj

The Java 2D API has various text rendering capabilities including methods for rendering strings and entire classes for setting font attributes and performing text layout (Oracle, o.D - f).
The "Graphics" and "Graphics2D" classes provide a range of options for presenting text within a window. Along with the font selection, these classes allow for customization of the text's size and style to suit the specific context. In order to draw a static text string, the most direct way to render it directly through the Graphics class by using the drawString method (Oracle, o.D -e). In order to utilize a specific font in JDOR, it is necessary to define it beforehand. Within JDOR, there are typically two methods available for defining a new font. However, it is important to note that only fonts that are already installed on the system can be used with both of these approaches. In the example below, the used fonts have been obtained from the list of the fonts available on the system, which are saved under the program named "2-110_JDOR_listShowPrintFonts.rxj,". The program can be found in the “samples” folder of the installed BSF4ooRexx850 package.

The given code excerpt below demonstrates the usage of JDOR to create a graphical window and display text using various fonts and colors:

Initially, a new window is created with a width of 550 and a height of 300 using the "winSize" command, followed by displaying the window using "winShow", between line 13 and 16.

The program starts by setting the font size to 14 and the font to "14_Comic" (Comic Sans MS) in lines 18 and 20. In line 21, the "goto" command positions the drawing cursor at coordinates (70, 60), and in line 24 the "drawString" command is used to display the text " font:". The "stringBounds" command in line 25 retrieves the bounding box information of the text, and the "parse var" statement in line 26 extracts the width of the text, which is then output using "say".

Starting from line 39, similar steps are repeated for other fonts, including "20_Bradley" (Bradley Hand ITC), "18_Copper" (Copperplate Gothic Light), and "20_Colonna" (Colonna MT). Different texts are displayed using the respective fonts, and their bounding box information is obtained to extract the width of each text, which is again output using "say".

The code then defines several colors using the "color" command, each specified with their respective RGB values, which can be found in the fully code in Appendix.

Furthermore, the line 96 "sleep 40" introduces a script pause of 40 seconds, allowing for a controlled timing delay in the execution of the script. This feature can be useful for various purposes such as coordinating actions or providing time for user interaction.

Lastly, the inclusion of "::requires "jdor.cls"" in the code signifies the inclusion of the "jdor.cls" file, which grants access to the "addJdorHandler" routine. This import enables the utilization of specific functionalities or capabilities provided by the "jdor.cls" file within the script, expanding the range of tools and features available for use.

```
12  --Creating and showing a new window
13  win_width = 500
14  win_height = 180
15  winSize win_width win_height
16  winShow
17
18  fontSize 14
19  fontStyle 1 -- 1=BOLD
20  font 14_Comic "Comic Sans MS"
21  goto 70 60
22  color black
23  font 14_Comic
24  drawString "font:"
25  stringBounds "font:"
26  parse var rc x " " y " " width " " height
27  say width
28  color black
29  drawLine 70+width 60
30  goto 270 60
31  drawString "text:"
32  stringBounds "text:"
33  parse var rc x " " y " " width " " height
34  say width
35  color black
36  drawLine 270+width 60
37
38  --create the 1st
39  fontSize 20
40  fontStyle 3 -- 3=BOLD+ITALIC
```

```
font 20_Bradley "Bradley Hand ITC"
goto 270 90
color shallowSea
font 20_Bradley
drawString "Dream big, work hard"
stringBounds "Dream big, work hard"
parse var rc x " " y " " width " " height
say width
color black
drawLine 270+width 90
goto 70 90
color enchanting
font 20_Bradley
drawString "Bradley Hand ITC:"
stringBounds "Bradley Hand ITC:"
parse var rc x " " y " " width " " height
say width

--create the 2nd
fontSize 18
fontStyle 1
font 18_Copper "Copperplate Gothic Light"
goto 270 120
color lagoon
font 18_Copper
drawString "Stay curious"
stringBounds "Stay curious"
parse var rc x " " y " " width " " height
say width
goto 70 120
color warmSpring
drawString "Forte:"
stringBounds "Forte:"
parse var rc x " " y " " width " " height
say width

--create the 3rd
fontSize 20
FontStyle 3
font 20_Colonna "Colonna MT"
goto 270 150
color mosaicTile
font 20_Colonna
drawString "Embrace the challenge"
stringBounds "Embrace the challenge"
parse var rc x " " y " " width " " height
say width
goto 70 150
color cerulean
font 20_Colonna
drawString "Colonna MT:"
stringBounds "Colonna MT:"
parse var rc x " " y " " width " " height
say width

sleep 40
::requires "jdor.cls"
```

Figure 2: JDOR-text.rxj (extract- complete code in Appendix -A2.1 JDOR_text.rxj)

Figure 3: Output of JDOR-text.rxj

4.2 Drawing - JDOR-drawing.rxj

The Java 2D API provides a useful set of standard shapes such as points, lines, rectangles, arcs, ellipses, and curve from “Graphics” and “Graphics2D”.

The “draw”-command only draws the outlines of the respective shapes in the previously defined color. For example, the “drawRect” command draws an empty rectangle with the given color. In order to fill the rectangle, the “fillRect” command will be used.

The provided code excerpt below demonstrates the use of JDOR for creating and graphical elements. The first part of code draws a series of ovals at different positions on the window. To display these drawings in ooRexx, a new window or frame must be created. Before the first oval is drawn, the starting point (“x” and “y”) of the new drawing must first be selected with the “goto” command. The “goto” command moves the drawing cursor to a specific position, and the drawOval command is used to draw ovals with the specified dimensions (see lines 11-23).

In lines 26 and 27, the width and height of the rectangles to be drawn are defined by "rect_width = 30" and "rect_height = 30". These values determine the dimensions of the rectangles, ensuring consistency in their size. Additionally, in lines 30 and 31, the initial position of the first rectangle is set with "start_x = 200" and "start_y = 5". By specifying the coordinates (x and y), the position of the first rectangle is established within the graphical context, providing a starting point for subsequent drawings.

The second part of the code, starting in line 34, uses a loop (do i = 1 to 10) to draw a pattern of rectangles. It calculates the position of each rectangle based on the loop index (i) and the defined width and height. The “goto” command moves the drawing cursor to the current position, and the “fillrect” command fills the rectangles with the specified dimensions.

```
8   -- Set the color
9   color mulberry 192 69 161
10  -- Draw the ovals
11  goto 50 50
12  drawOval 40 40
```

```
goto 53 53
drawOval 60 60
goto 56 56
drawOval 80 80
goto 59 59
drawOval 100 100
goto 62 62
drawOval 120 120
goto 65 65
drawOval 140 140
goto 68 68
drawOval 160 160
-- Define the size of the rectangles
rect_width = 30
rect_height = 30

-- Set the initial position for the first rectangle
start_x = 200
start_y = 5

-- Draw the pattern of Sapphire colored rectangles
do i = 1 to 10
 -- Calculate the position of the current rectangle
 rect_x = start_x + (i - 1) * rect_width
 rect_y = start_y + (i - 1) * rect_height

-- Fill the rectangle at the current position with the random color
 goto rect_x rect_y
color sapphire 79 118 231
 fillrect rect_width rect_height
end
-- Define the size of the rectangles
rect_width = 30
rect_height = 30
-- Set the initial position for the first rectangle
start_x = 230
start_y = 5
-- Draw the pattern of orange rectangles
do i = 1 to 10
-- Calculate the position of the current rectangle
rect_x = start_x + (i - 1) * rect_width
rect_y = start_y + (i - 1) * rect_height
 -- Fill the rectangle at the current position with the random color
goto rect_x rect_y
color orange
fillrect rect_width rect_height
end
```

Figure 4: JDOR-drawing.rxj (excerpt- complete code in Appendix -A2.2 JDOR-drawing.rxj)

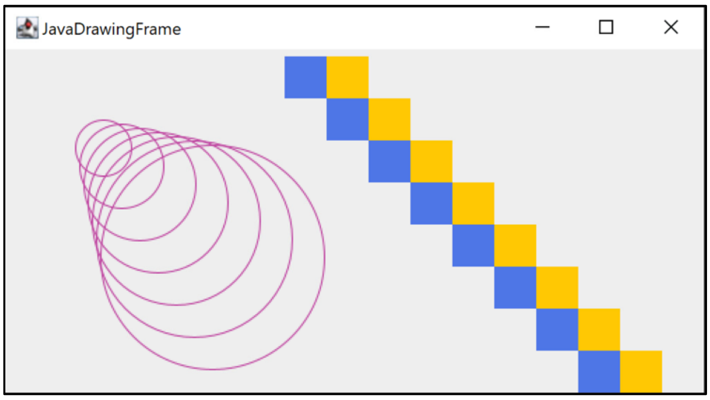

Figure 5: Output of JDOR-drawing.rxj

4.3 Visualizing with Images - JDOR-images.rxj

Images are an important component of many modern graphical user interfaces, and leveraging the capabilities of the "Graphics" and "Graphics2D" libraries is crucial for image manipulation. JDOR enables developers to utilize these essential functions within the ooRexx environment.

The application can draw on to image by using Java 2D API graphics calls. So, images are not limited to displaying photographic type images. Different objects such as line art, text, and other graphics and even other images can be drawn onto an image (Oracle, o.D.-d). The resulting image can then be drawn to a screen, sent to a printer, or saved in a graphics format such as PNG, GIF etc (Oracle, o.D.-d).

In the given example below, the emphasis is placed on three key commands: "loadImage", "drawImage" and "saveImage". These commands hold significant importance when it comes to handling and storing images. The " loadImage" command is utilized to import an image into the JDOR registry. To position the image in the center of the frame, the image's dimensions are required, which can be obtained using the "imageSize nickname" command. The width and height of the image are then stored in the "rc" variable. By combining the window size and image dimensions, the starting point for the image can be calculated and specified using the "goto" command. Finally, the "drawImage *nickname*" command is used to draw the image at the current location in the JDOR window. With the “saveImage *nickname* the resulting image is saved under the name “*nickname.png*” in the current path.

The following example will visualize the load of an image of the Pyramids of Giza into the ooRexx frame and adding the names of the pyramids in various colors and fonts. Additionally, some drawings will be added on the screen.

In the code excerpt given below, in line 12 an image file of the Giza Pyramids named "*py.png*" is imported and assigned the nickname "*Pyramids_of_Giza*" using the “loadImage” command. The image is then drawn on the window using the “drawImage” command in line 13.

Next, rectangles and circles are drawn and filled, starting from line 16. The cursor is moved to specific coordinates using “goto”, and the “drawRect” and “fillRect” commands are used to draw and fill rectangles, while the “drawOval” and “fillOval” commands are employed for circles. The desired colors are applied to the shapes (see lines 16, 23, 29)

Text drawing follows, starting with the drawing of the first pyramid's name in line 33. The font size is set to 16 using “fontSize 16”, and the "Berlin Sans FB" font is selected with the font command. The color "silkribbon" is applied (respective RGB values of the colors are defined at the beginning of the code, which can be found in the full code in Appendix A 2.3), and the text "MENKAURE" is drawn at coordinates (50, 100) using “drawstring”. The string's bounding box is determined using “stringBounds”, and the width of the box is extracted and displayed using parse var. A line is drawn from the starting point to the end of the text using “drawLine” with the calculated width.

Starting in lines 45 and 52, the second and third pyramids' names are drawn similarly, but with different font sizes, fonts, and colors. The text "KHUFU" is drawn at (250, 50) using a font size of 32 and the "Forte" font. The text "KHAFRE" is drawn at (400, 110) with a font size of 24 and the "Arabic Typesetting" font.

Lastly, the resulting image is saved in the same directory as "*Names_of_Giza_Pyramids.png*" using “saveImage”.

```
11    -- import the image
12  loadImage Pyramids_of_Giza "py.png" -- nickname and path
13  drawImage Pyramids_of_Giza
14
15      -- draw and fill rectangle
16  color powderblue
17  goto 140 200
18  drawRect 60 40
19  fillRect 60 40
20
21    -- draw and fill circle
22  goto 170 210
23  color thistle
24  drawOval 70 70
25  fillOval 70 70
26
27    -- draw rectangle
28  goto 260 220
29  color tropicaldream
30  drawRect 70 70
31
32    -- 1st Pyramid
33  fontSize16
34  fontStyle 1 -- 1=BOLD
35  font 16_Berlin_S "Berlin Sans FB"
36  color silkribbon
37  goto 50 100
38  drawString "MENKAURE"
39  stringBounds"MENKAURE"
40  parse var rc x " " y " " width " " height
41  say width
```

```
color citron
drawLine50 + width 100

  --2nd Pyramid
fontSize 32
font 32_Forte "Forte"
color blazeorange
goto 250 50
drawString "KHUFU"

  -- 3rd Pyramid
fontSize 24
font 24_Arabic_T "Arabic Typesetting"
color jamaicansea
goto 400 110
drawString "KHAFRE"

  --Saving the created image in the same directory
saveImage "Names_of_Giza_Pyramids.png"
sleep 40
::requires "jdor.cls"
```

Figure 6: JDOR-images.rxj (excerpt- complete code in Appendix -A2.3 JDOR-images.rxj)

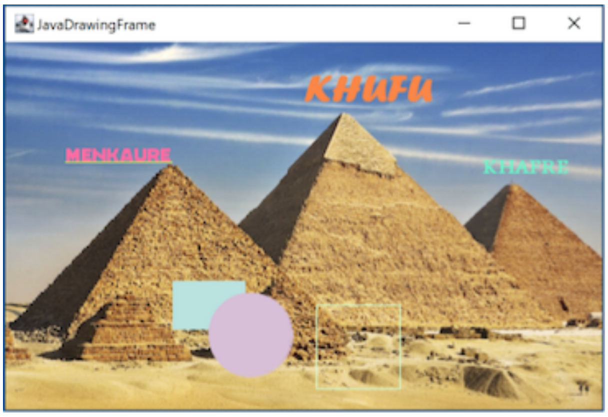

Figure 7: Output of JDOR-images.rxj

4.4 Rotate, Scale, Translate and Shear – JDOR-manipulate.rxj

The transform attribute in the Graphics2D context can be modified to move, rotate, scale, and shear graphics primitives when they are rendered. The transform attribute is defined by an instance of the Affine Transform class. An affine transform is a transformation such as translate, rotate, scale, or shear in which parallel lines remain parallel even after being transformed (Oracle, o.D.-h).

The Graphics2D provides transformation methods that allow you to modify the existing transform. An angle of rotation in radians can be specified, allowing for rotation. Scaling can be achieved by specifying factors for both the x and y directions. Shearing can be performed by specifying shearing factors for both the x and y directions. Translation can be accomplished by specifying offsets for both the x and y directions (Oracle, o.D.-h).

The given example below starts by drawing a coordinate system by looping through the width and height of the window, which can be found between lines 13 and 20. The “goto” statement moves the drawing cursor to the specified coordinates, and “drawLine” draws lines to connect the points. The coordinate system lines are drawn with the coordinate_system color.

After that, various methods and transformations are applied. The program starts drawing two lines forming an X shape in line 28. The “moveTo” command sets the starting point of the lines, and the color command defines a color named "pantone" with RGB values (0, 206, 209) and an alpha value of 127 (50% transparency). The “fillRect”, “drawRect”, and “drawOval” methods are used to fill and draw rectangles and ovals with the specified colors.

A translation is performed using the “translate” command in line 37, shifting subsequent drawings to a new position (260, 250). The “rotate” command rotates subsequent shapes by 45 degrees counterclockwise around the origin (0, 0). The “fillRect” and “fillOval” methods are applied with the “pantone” color, and “drawRect” and “drawOval” are used with the color blue.

The “goto” statement in line 48 moves the drawing cursor to (150, 15), and “drawPolygon” is used to draw a polygon with a size of 50x50 pixels. The “rotate” command in line 50 rotates the subsequent polygon by 45 degrees. Another “drawPolygon” with the same size is drawn after the rotation.

Starting in line 54, moving to (70, 70) using “goto”, an orange color is set using “color”, and “fillOval” fills an oval with dimensions of 40x40 pixels in line 56. The "shear -1 0" command applies a shearing transformation with a horizontal shear factor of -1. Then, a pink color is set using “color”, and another “fillOval” fills an oval with the same dimensions in pink color.

```
11    --drawing the system
12  color coordinate_system
13    do i=0 to win_width by 25
14  goto i 0
15  drawline i win_height
16    end
17    do i=0 to win_height by 25
18  goto 0 i
19  drawline win_width i
20    end
21  color middle
22  goto win_width/2 0
23  drawline win_width/2 win_height
24  goto 0 win_height/2
25  drawline win_width win_height/2
26  -- Applying methods
27  -- draw two lines forming a big X
28  moveTo 70 80        -- currX=70, currY=80
29    -- define and set color, register it with the name "pantone"
```

```
color pantone 0 206 209 127   -- R,G,B,alpha=127 (50 % transparency)
fillRect 50 50
color blue
drawRect 50 50
color blue
drawOval 50 50

translate 260 250
moveTo 0 0
rotate 45
color pantone
fillRect 50 50
fillOval 50 50
color blue
drawRect 50 50
color blue
drawOval 50 50

"goto 150 15"
drawPolygon 50 50
rotate 45
drawPolygon 50 50
rotate 45

goto 70 70
color orange
fillOval 40 40
"shear -1 0"
color pink
fillOval 40 40
```

Figure 8: JDOR-manipulate.rxj (excerpt- complete code in Appendix- A 2.4 JDOR-manipulate.rxj)

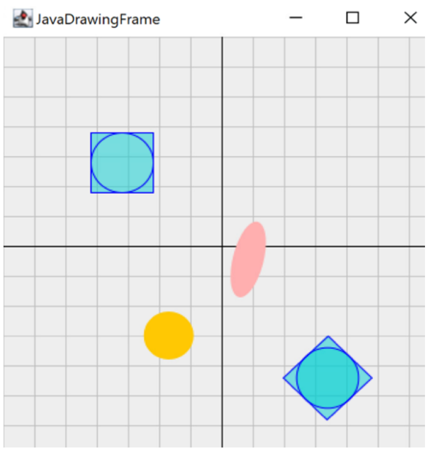

Figure 9: Output of JDOR-manipulate.rxj

4.5 Moving Objects - JDOR-move.rxj

To “animate” objects in ooRexx through JDOR, you can create the illusion of movement by repeatedly drawing an object at different positions within short time intervals. While the object remains stationary in reality, the rapid succession of these drawings makes it appear as if the object is actually moving from one place to another (Blauensteiner, 2023). This technique utilizes the capabilities of Java 2D, allowing one to create dynamic and visually engaging animations within your ooRexx scripts.

In the following example, inside the animation window, a square object will appear to move in a circular path. The object's size is determined by the “square_size” variable. The animation will continue indefinitely as the square object moves from one position to another.

First, the code imports the Java Math class from the java.lang package using the bsf.import function, assigning it the name "calc" in ooRexx. This allows access to various mathematical functions provided by the Math class.

Next, several variables are defined: “win_width” and “win_height” represent the dimensions of the animation window, “square_size” determines the size of the square object to be drawn, speed sets the rate at which the object moves, and “desertSunrise” defines a custom color using RGB values.

The script proceeds with defining variables in lines 19-22: centerX and centerY represent the coordinates of the window's center, radius determines the distance from the center at which the object will move, and angle holds the initial angle for the object.

In line 25, within an infinite loop, the script calls “getState”, to retrieve the current state of the animation.

Next, the code calculates the current X and Y positions of the object based on the centerX, centerY, angle, and radius variables in lines 26 and 28. The currX and currY variables represent the top-left coordinates of the square object to be drawn. The *“cos”* function is used in the code to calculate the X-coordinate of the current position on a circular path. It helps determine the horizontal position based on the angle and radius. Similarly, the *“sin”* function is used to calculate the Y-coordinate, representing the vertical position on the circular path. Together, these calculations enable the object to move smoothly along the circular trajectory in the animation.

The “goto” statement moves the drawing cursor to the specified currX and currY coordinates, and “fillRect” fills a square of size “square_size “at the current cursor position.

In line 32, the angle is then incremented by the speed value to control the object's movement.

```
1   call addjdorhandler
2   address jdor
3   call bsf.import "java.lang.math", "calc" -- allows access to various mathematical functions
4   -- Create a new window
5   win_width = 500
6   win_height = 500
7   square_size = 50
8
9   speed = 2 -- speed of the animation is set to 2
10  color desertSunrise 255 167 146
```

```

winsize win_width win_height
new win_width win_height
background white
clearRect win_width win_height
winshow
color desertsunrise

centerX = win_width / 2
centerY = win_height / 2
radius = win_width / 4
angle = 0
  -- Start the loop
do forever
getstate
currx = centerx + .calc~cos(.calc~toradians(angle)) * radius - square_size / 2 -- Calculates the X-
coordinate of the current position based on the angle and radius.
curry = centery + .calc~sin(.calc~toradians(angle)) * radius - square_size / 2

goto currx curry
fillRect square_size square_size
angle = angle + speed -- update the angle for the next iteration
sleep 0.01
end  --end the infinite loop
::requires "jdor.cls"
```

Figure 10: JDOR-move.rxj

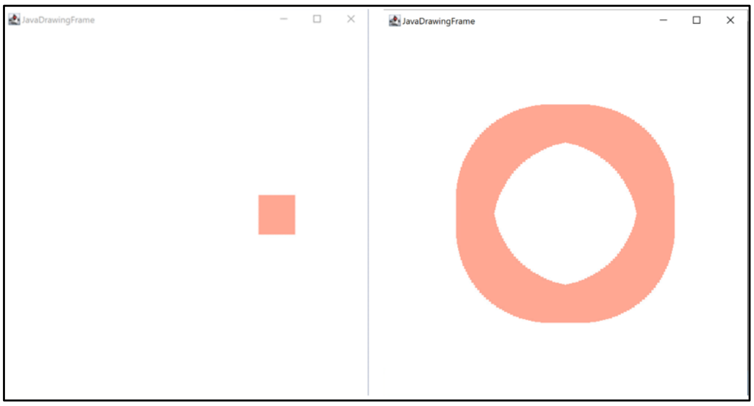

Figure 11: Output of JDOR-move.rxj

5. Additional Examples

This chapter expands on JDOR by showcasing additional examples for better understanding.

5.1 Example 1 - JDOR-PurpleStar.rxj

The following is an example of an animation, in which a “star shape” is created from a single circle. The star will be drawn as filled ovals with a specific size and color. The circles will move in a circular pattern starting from the given position of the window and gradually increasing their distance from the center. As the circles move, they will leave a trail behind, creating an animated effect.

```
call addjdorhandler
address jdor
call bsf.import "java.lang.Math", "calc"

win_width = 500
win_height = 500
star_size = 50
speed = 1

color daylightlilac 158 124 243
winsize win_width win_height
new win_width win_height
background white
clearoval win_width win_height
winshow

color daylightLilac
centerX = win_width / 2
centerY = win_height / 2
radius = win_width / 4
angle = 0
delta_angle = .calc~toRadians(72)  -- 360 degrees divided by 5 sides of the star
distance = 0

 do forever
  getState
  currX = centerX + .calc~cos(angle) * distance
  curry = centery + .calc~sin(angle) * distance
  goto currx curry
  fillOval star_size star_size
  angle = angle + delta_angle
  if angle > 2 * .calc~pi then angle = angle - 2 * .calc~pi
  distance = distance + speed
  if distance > radius then distance = 0

::requires "jdor.cls"
```

Figure 12: JDOR-PurpleStar.rxj

Initially, the code calls the "addJdorHandler" command to load the Java Rexx command handler, and the "address jdor" command sets the default environment to JDOR. Then, the "bsf.import" command is used to import the "java.lang.Math" class and its "calc" method.

The code proceeds to define variables such as "win_width" (window width), "win_height" (window height), "star_size" (size of the star), "speed" (movement speed), and "daylightLilac" (a specific color defined using RGB values).

The "winSize" command sets the size of the window based on "win_width" and "win_height", followed by creating a new window using the "new" command with the same dimensions. The "background white" command sets the background color of the window to white, and the "clearOval win_width win_height" command clears any existing ovals from the window. Finally, the "winShow" command displays the window.

Next, the code sets the current color to "daylightLilac" using the "color" command. The variables "centerX" and "centerY" are calculated as the center coordinates of the window, and "radius" is set to one-fourth of the window width (see lines 18-20). In lines 21 and 22, the "angle" variable is initialized to 0, representing the starting angle of the star, and "delta_angle" is calculated as the equivalent of 72 degrees in radians, which will be used to increment the angle in each iteration. In line 23, the "distance" variable is set to 0, representing the initial distance from the center.

Inside the infinite loop created by "do forever", the code calls the "getState" command to retrieve the current state of the graphical window, which can be found starting in line 25. The coordinates of the current star position are calculated based on the center coordinates, the angle, and the distance from the center using the trigonometric functions provided by the "calc" method (see lines 27 and 28). The "goto" command moves the drawing cursor to the calculated position, and the "fillOval" command draws a filled oval with the specified star size.

In line 31, the angle is incremented by "delta_angle," and if it exceeds 2π (full circle), it is adjusted to keep it within the valid range. In line 33, the "distance" variable is increased by the "speed" value, representing the distance traveled from the center. If the distance exceeds the radius of the star pattern, it is reset to 0.

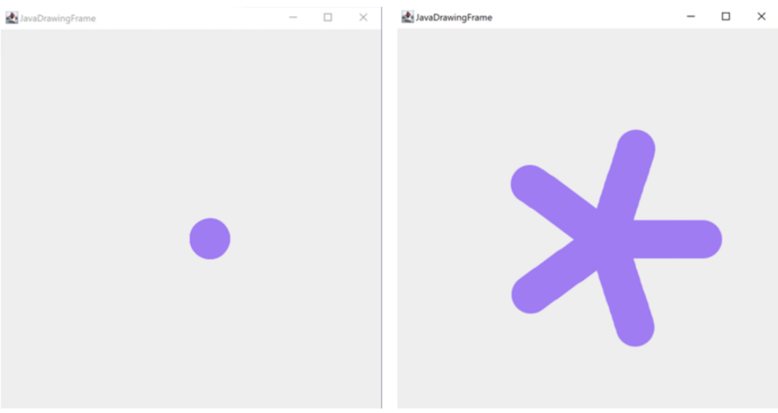

Figure 13: Output of JDOR-PurpleStar.rxj

5.2 Example 2 - JDOR-AffineTransformation.rxj

The following is an example of Affine Transformation. A red triangle shape will be created and then applied transformations to make it rotate and scale.

```
1  jdh=.bsf~new("org.oorexx.handlers.jdor.JavaDrawingHandler")
2  say "JDOR version:" jdh~version -- show version
3  call BsfCommandHandler "add", "jdor", jdh
4  address jdor
5
6  newImage 300 300 -- create new image
7  winShow -- show image in a window
8  winTitle "Affine Transform Demo (ooRexx)" -- set window's title
9
10 polygonXs="(20,0,40)" -- define three x coordinates for the triangle
11 polygonYs="(40,20,40)" -- define three y coordinates for the triangle
12 shape myP polygon polygonXs polygonYs 3 -- create triangle shape
13
14 translate 200 200 -- move origin (x=200, y=200)
15 scale 1.1 1.1 -- increase the triangle shape size 10%
16 rotate 20 -- rotate by 20 degrees
17 color red -- set color to red
18 do 20
19   fillShape myP -- fill (and show) the triangle shape
20   rotate 20
21 end
22 say 'Hit <enter> to proceed (end) ...'
23 parse pull data -- wait until user presses <enter> on the keyboard
24 ::requires "bsf.cls"
```

Figure 14: JDOR-AffineTransformation.rxj

Let's break down the code given above:

A new image is created using the “newImage” command, specifying its dimensions as 300x300 pixels, which can be found in line 6. In line 7, the “winShow” command displays the image in a window, making it visible to the user. In line 8 the “winTitle” command sets the title of the window to "Affine Transform Demo (ooRexx)".

In lines 10 and 11, using the polygonXs and polygonYs variables, a triangle shape is defined by providing three sets of x and y coordinates for its vertices. In line 12, the shape command is used to create a shape object named myP using the defined triangle shape. Starting line 14, transformation commands are then applied to the shape. The translate command moves the origin of the shape to the specified coordinates (200, 200). The scale command increases the size of the shape by 10% in both the x and y directions. The rotate command rotates the shape by 20 degrees.

In line 18, to create a rotating effect, a loop is initiated using the “do” command, which repeats the enclosed commands a specified number of times (in this case, 20 times). Within the loop, the “fillShape” command fills and displays the transformed triangle shape, and the “rotate” command is used to rotate the shape by 20 degrees each time.

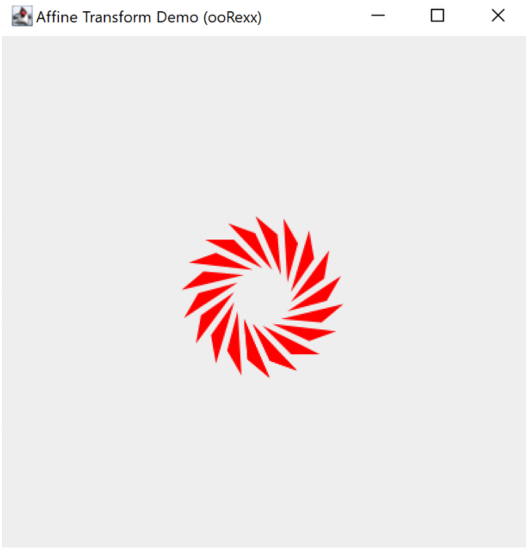

Figure 15: Output of JDOR-AffineTransformation.rxj

5.3 Example 3 - JDOR-CubePyramid.rxj

In the following example, a cube will be drawn, using the color "LemonLime." The rectangles sides will be connected by drawing lines between corresponding vertices. Additionally, a pyramid will be drawn with a front, back and side faces, using the color "peonypink," and its sides will also be connected.

```
-- Draw the Cube
--Creating / Saving stroke
dashphase_stroke1=bsf.createJavaArrayOf("float.class", 15, 8, 15,8)
stroke stroke1 3 2 0 10 "dashphase_stroke1" 0
-- Draw the front face of the cube
color LemonLime
goto 50 50
stroke strokeA
drawLine 150 50
goto 150 50
drawLine 150 150
goto 150 150
drawLine 50 150
goto 50 150
drawLine 50 50
-- Draw the back face of the cube
goto 70 70
drawLine 170 70
goto 170 70
drawLine 170 170
goto 170 170
drawLine 70 170
```

```
34 goto 70 170
35 drawLine 70 70
36 -- Connect the corresponding vertices of the front and back faces
37 goto 50 50
38 drawLine 70 70
39 goto 150 50
40 drawLine 170 70
41 goto 150 150
42 drawLine 170 170
43 goto 50 150
44 drawLine 70 170
45
46 -- Draw the Triangle
47 -- Draw the front face of the triangle
48 color peonypink
49 goto 190 190
50 drawLine 290 190
51 goto 240 290
52 drawLine 190 190
53 goto 240 290
54 drawLine 290 190
55 -- Draw the back face of the triangle
56 goto 210 210
57 drawLine 310 210
58 goto 260 310
59 drawLine 210 210
60 goto 260 310
61 drawLine 310 210
62 -- Connect the corresponding vertices of the front and back faces
63 goto 190 190
64 drawLine 210 210
65 goto 290 190
66 drawLine 310 210
67 goto 240 290
68 drawLine 260 310
```

Figure 16: JDOR-CubePyramid.rxj (extract - complete code in Appendix -A2.5 JDOR-CubePyramid.rxj)

The code proceeds to draw a cube by creating a stroke pattern in line 14, represented by the "dashphase_stroke1" array, which is then assigned to the "stroke1" stroke. Starting in line 17, the front face of the cube is drawn by setting the color to "LemonLime" (which was pre-assigned and can be found in the fully code in Appendix A 2.5 JDOR- CubePyramid.rxj) and using the "goto" and "drawLine" commands to connect the specified points. Similarly, the back face of the cube is drawn starting line 27. The corresponding vertices of the front and back faces are connected by drawing lines between them, in lines 37 - 44.

Following the cube, a pyramid is drawn using the color "peonypink", starting from line 48. Between lines 56 and 61 the front and back faces of the triangle are created using the "goto" and "drawLine" commands, connecting the specified points. The corresponding vertices of the front and back faces are then connected with lines using the "drawLine" command.

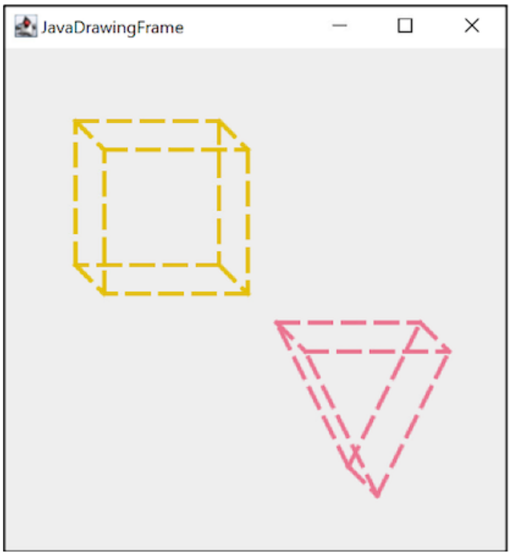

Figure 17: Output of JDOR-CubePyramid.rxj

5.4 Example 4 - JDOR-RotatingSquare.rxj

In the following example, a black square will be created at the bottom of the window. As the loop iterates, the square will be drawn at different positions and orientations, creating a visual effect of a rotating square ascending through the window.

```
call addJdorHandler
address jdor
-- creating and showing a new window
new 500 500
winShow
-- define the initial size of the square
square_size = 50
-- set the rotation angle
angle = 5
-- calculate the center coordinates of the window
center_x = 500 / 2
center_y = 500 / 2
-- calculate the starting position of the square at the bottom of the window
square_x = center_x - square_size / 2
square_y = 500 - square_size
-- draw and rotate the square
do while square_y > 0
-- draw the square at the current position
 goto square_x square_y
 drawRect square_size square_size
-- rotate the square
 rotate center_x center_y angle
-- update the position of the square
 square_y = square_y - 1
 -- pause to observe the rotation
```

```
26  sleep 0.005
27 end
28 sleep 60
29 ::requires "jdor.cls"
```

Figure 18: JDOR-RotatingSquare.rxj

The script defines the initial size of a square, setting the square_size variable to 50, in line 7. It also assigns an angle of 5 to the angle variable in line 9, which will be used for rotation calculations.

In lines 14 and 15, to determine the starting position of the square at the bottom of the window, the script calculates the center coordinates of the window by dividing its width and height (both set to 500) by 2. These coordinates are assigned to center_x and center_y. The square_x coordinate is derived by subtracting half of the square's size from center_x, while square_y is set to 500 minus the square's size.

Starting in line 17, the subsequent section of the code enters a loop that continues until the square_y coordinate becomes less than or equal to 0. Within this loop, the script draws the square at the current position using “goto” and “drawRect” commands. It then rotates the square around the center of the window, specified by center_x and center_y, using the “rotate” command. The position of the square is updated by decreasing the square_y coordinate by 1, in line 24. To observe the rotation, the script introduces a brief pause using the sleep 0.005 command in line 26. This loop iterates until the square reaches the top of the window.

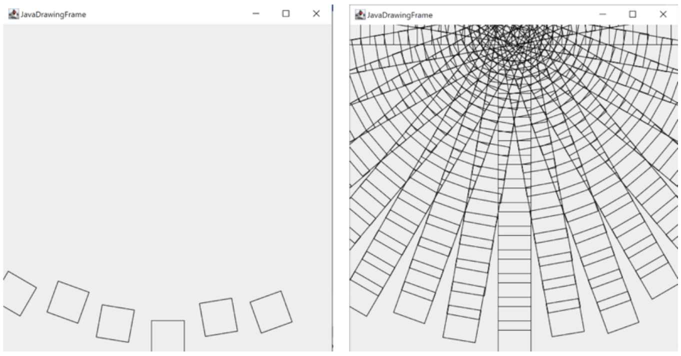

Figure 19: Output of JDOR-RotatingSquare.rxj

6. Conclusion

ooRexx is a programming language that offers a wide range of functionalities, making it particularly beginner-friendly and easy to grasp. One of its strengths lies in its ability to seamlessly integrate with other programs, opening up limitless opportunities for users. Such as Java.

This seminar paper explored the application of JDOR, a BSF4ooRexx 850 extension, in ooRexx programming to generate diverse 2D drawing programs. The combination of ooRexx and JDOR with the BSF4ooRexx850 framework enables programmers, even those with limited knowledge of ooRexx and Java, to create intricate and visually appealing drawings.

JDOR offers a user-friendly and efficient approach to 2D graphic creation by simplifying the development process and allowing programmers to focus on their creative ideas. Integration with the Java 2D API increases JDOR's capabilities, enabling the creation of 2D graphics that respond to user input and provide an immersive experience. Java 2D API, when integrated with JDOR, provides programmers with a comprehensive set of functionalities for graphic creation and manipulation. The introduction of the JDOR command handler simplifies the utilization of Java's awt 2D classes, allowing Rexx programmers to leverage the Java 2D API without directly using Java code. Practical examples showcased in the paper demonstrate the incremental growth and expansion of JDOR's capabilities.

Appendix

A 1. Installation Guide

To utilize JDOR with BSF4ooRexx850, firstly installing ooRexx is needed, which is essential for working with BSF4ooRexx850 JDOR. The installation process is straightforward and user-friendly, and an installer can be easily accessed on the ooRexx website: https://sourceforge.net/projects/oorexx/files/oorexx/5.0.0beta/

Secondly, to enable Rexx to interact with Java, it is necessary to install Java as well. The Java installation can be completed using a readily available installer found on the Java website. This installation process is similar to that of ooRexx and should pose no significant challenges. Java can be downloaded from the following URLs: http://www.java.com , http://www.adoptOpenJDK.org

Once ooRexx and Java installations are completed, proceed with the installation of BSF4ooRexx, a crucial component for utilizing the JDOR library. The BSF4ooRexx installer can be downloaded from the project website: https://sourceforge.net/projects/bsf4oorexx/files/beta/

After successfully installing ooRexx, Java, and BSF4ooRexx, you can verify the correctness of the installations by executing the ooRexxTry.rxj program file. This file, included in the BSF4ooRexx installation, tests the functionality of the setup. In case of any errors or issues, you may need to troubleshoot the installation process or consult the project's documentation and community for assistance.

Once the required files are confirmed to be in place, you can start using the Java2D Drawing library.

A 2. Codes

A 2.1 JDOR-text.rxj

```
call addJdorHandler -- load
address jdor -- set default environment to JDOR

-- setting the colors
color enchanting 41 128 185
color warmSpring 60 154 242
color cerulean 24 117 227
color shallowSea 40 180 99
color lagoon 62 181 161
color mosaicTile 29 130 118

--Creating and showing a new window
win_width = 500
win_height = 180
winSize win_width win_height
winShow
```

```

fontSize 14
fontStyle 1 -- 1=BOLD
font 14_Comic "Comic Sans MS"
goto 70 60
color black
font 14_Comic
drawString "font:"
stringBounds "font:"
parse var rc x " " y " " width " " height
say width
color black
drawLine 70+width 60
goto 270 60
drawString "text:"
stringBounds "text:"
parse var rc x " " y " " width " " height
say width
color black
drawLine 270+width 60

--create the 1st
fontSize 20
fontStyle 3 -- 3=BOLD+ITALIC
font 20_Bradley "Bradley Hand ITC"
goto 270 90
color shallowSea
font 20_Bradley
drawString "Dream big, work hard"
stringBounds "Dream big, work hard"
parse var rc x " " y " " width " " height
say width
color black
drawLine 270+width 90
goto 70 90
color enchanting
font 20_Bradley
drawString "Bradley Hand ITC:"
stringBounds "Bradley Hand ITC:"
parse var rc x " " y " " width " " height
say width

--create a 2nd
fontSize 18
fontStyle 1
font 18_Copper "Copperplate Gothic Light"
goto 270 120
color lagoon
font 18_Copper
drawString "Stay curious"
stringBounds "Stay curious"
parse var rc x " " y " " width " " height
say width
goto 70 120
color warmSpring
drawString "Forte:"
stringBounds "Forte:"
parse var rc x " " y " " width " " height
```

```
say width

--create a 3rd
fontSize 20
FontStyle 3
font 20_Colonna "Colonna MT"
goto 270 150
color mosaicTile
font 20_Colonna
drawString "Embrace the challenge"
stringBounds "Embrace the challenge"
parse var rc x " " y " " width " " height
say width
goto 70 150
color cerulean
font 20_Colonna
drawString "Colonna MT:"
stringBounds "Colonna MT:"
parse var rc x " " y " " width " " height
say width

SLEEP 40
::REQUIRES "jdor.cls"
```

A 2.2 JDOR-drawing.rxj

```
call addjdorhandler
address jdor
--Creating and showing a new window
win_width = 500
win_height = 245
new win_width win_height
winshow
-- Set the color
color mulberry 192 69 161
-- Draw the ovals
goto 50 50
drawOval 40 40
goto 53 53
drawOval 60 60
goto 56 56
drawOval 80 80
goto 59 59
drawOval 100 100
goto 62 62
drawOval 120 120
goto 65 65
drawOval 140 140
goto 68 68
drawOval 160 160
-- Define the size of the rectangles
rect_width = 30
rect_height = 30

-- Set the initial position for the first rectangle
```

```
start_x = 200
start_y = 5

-- Draw the pattern of Sapphire colored rectangles
do i = 1 to 10
 -- Calculate the position of the current rectangle
 rect_x = start_x + (i - 1) * rect_width
 rect_y = start_y + (i - 1) * rect_height

-- Fill the rectangle at the current position with the random color
 goto rect_x rect_y
color sapphire 79 118 231
 fillrect rect_width rect_height
end

-- Define the size of the rectangles
rect_width = 30
rect_height = 30

-- Set the initial position for the first rectangle
start_x = 230
start_y = 5
-- Draw the pattern of orange rectangles
do i = 1 to 10
-- Calculate the position of the current rectangle
rect_x = start_x + (i - 1) * rect_width
rect_y = start_y + (i - 1) * rect_height

-- Fill the rectangle at the current position with the random color
goto rect_x rect_y
color orange
fillrect rect_width rect_height
end
sleep 60
::requires "jdor.cls"
```

A 2.3 JDOR-images.rxj

```
 call addJdorHandler -- load and add the Java Rexx command handler,
address jdor -- set default environment to JDOR

  --Creating and showing a new window
win_width = 500
win_height = 308
winsize win_width win_height
winshow

 -- import the image
loadImage Pyramids_of_Giza "py.png" -- nickname and path
drawImage Pyramids_of_Giza

   -- draw and fill rectangle
color powderblue
goto 140 200
drawRect 60 40
fillRect 60 40
```

```

  -- draw and fill circle
goto 170 210
color thistle
drawOval 70 70
fillOval 70 70

  -- draw rectangle
goto 260 220
color tropicaldream
drawRect 70 70

  -- 1st Pyramid
fontSize16
fontStyle 1 -- 1=BOLD
font 16_Berlin_S "Berlin Sans FB"
color silkribbon
goto 50 100
drawString "MENKAURE"
stringBounds"MENKAURE"
parse var rc x " " y " " width " " height
say width
color citron
drawLine50 + width 100

  --2nd Pyramid
fontSize 32
font 32_Forte "Forte"
color blazeorange
goto 250 50
drawString "KHUFU"

  -- 3rd Pyramid
fontSize 24
font 24_Arabic_T "Arabic Typesetting"
color jamaicansea
goto 400 110
drawString "KHAFRE"

  --Saving the created image in the same directory
saveImage "Names_of_Giza_Pyramids.png"
sleep 40
::requires "jdor.cls"

```

A 2.4 JDOR-manipulate.rxj

```
 call addJdorHandler
address jdor
--Creating and showing a new window
win_width = 350
win_height = 350
```

```
winSize win_width win_height
winShow

 --setting the colors
color coordinate_system 190 190 190 200
color middle 0 0 0 255
   --drawing the system
color coordinate_system
   do i=0 to win_width by 25
goto i 0
drawline i win_height
   end
   do i=0 to win_height by 25
goto 0 i
drawline win_width i
   end
color middle
goto win_width/2 0
drawline win_width/2 win_height
goto 0 win_height/2
drawline win_width win_height/2
-- Applying methods
-- draw two lines forming a big X
moveTo 70 80        -- currX=70, currY=80
   -- define and set color, register it with the name "pantone"
color pantone 0 206 209 127   -- R,G,B,alpha=127 (50 % transparency)
fillRect 50 50
color blue
drawRect 50 50
color blue
drawOval 50 50

translate 260 250
moveTo 0 0
rotate 45
color pantone
fillRect 50 50
fillOval 50 50
color blue
drawRect 50 50
color blue
drawOval 50 50

"goto 150 15"
drawPolygon 50 50
rotate 45
drawPolygon 50 50
rotate 45

goto 70 70
color orange
fillOval 40 40
"shear -1 0"
color pink
fillOval 40 40

say "press enter to end."; parse pull
sleep 400
```

```
::requires "jdor.cls"
```

A 2.5 JDOR-CubePyramid.rxj

```
call addJdorHandler
address jdor
-- Creating and showing a new window
win_width = 350
win_height = 350
NEW win_width win_height
WINSHOW
-- Setting the colors
color LemonLime 228 192 0
color peonypink 235 117 145
-- Draw the Cube
--Creating / Saving stroke
dashphase_stroke1=bsf.createJavaArrayOf("float.class", 15, 8, 15,8)
STROKE strokeA 3 2 0 10 "dashphase_stroke1" 0
-- Draw the front face of the cube
color LemonLime
goto 50 50
STROKE strokeA
drawLine 150 50
goto 150 50
drawLine 150 150
goto 150 150
drawLine 50 150
goto 50 150
drawLine 50 50
-- Draw the back face of the cube
goto 70 70
drawLine 170 70
goto 170 70
drawLine 170 170
goto 170 170
drawLine 70 170
goto 70 170
drawLine 70 70
-- Connect the corresponding vertices of the front and back faces
goto 50 50
drawLine 70 70
goto 150 50
drawLine 170 70
goto 150 150
drawLine 170 170
goto 50 150
drawLine 70 170
-- Draw the Triangle
-- Draw the front face of the triangle
color peonypink
```

```
goto 190 190
drawLine 290 190
goto 240 290
drawLine 190 190
goto 240 290
drawLine 290 190

-- Draw the back face of the triangle
goto 210 210
drawLine 310 210
goto 260 310
drawLine 210 210
goto 260 310
drawLine 310 210

-- Connect the corresponding vertices of the front and back faces
goto 190 190
drawLine 210 210
goto 290 190
drawLine 310 210
goto 240 290
drawLine 260 310

sleep 60
::requires "jdor.cls"
```

A 3. List of Figures

A 4. List of Tables

References

1. Blauensteiner, F. (2023). JDOR – An introduction to Java 2D's drawing classes with ooRexx and BSF4ooRexx,https://wi.wu.ac.at/rgf/diplomarbeiten/BakkStuff/2023/202302_Blauensteiner_JDOR.pdf

2. Cowell, J. (1999). The Abstract Windowing Toolkit. In: Essential Visual J++ 6.0 fast . Essential Series. Springer, London. https://doi.org/10.1007/978-1-4471-0565-7_11

3. Flatscher, R. G. (2022b). *BSF4ooRexx: Introducing the JDOR Rexx Command Handler for Easy Creation of Bitmaps and Bitmap Manipulations on Windows, Mac and Linux* International RexxLA Symposium, 2022-09, https://www.rexxla.org/presentations/2022/202209_JDOR_command_handler.pdf

4. Flatscher, R. G. (2023). Proposing ooRexx and BSF4ooRexx for Teaching Programming and Fundamental Programming Concepts. ISECON23-Conference

5. Holt, W. (1999). *THE EMBEDDED WINDOW TOOLKIT* (Doctoral dissertation, UNIVERSITY OF CALIFORNIA SANTA CRUZ). http://alumni.soe.ucsc.edu/~wholt/thesis.pdf

6. https://dotnettutorials.net/lesson/abstract-windows-toolkit-awt-in-java/), 2022. Abstract Windows Toolkit (AWT) in Java. Retrieved 10.04.2023 from https://dotnettutorials.net/lesson/abstract-windows-toolkit-awt-in-java/

7. Oracle (o. D. -a). About the JFC and Swing. Oracle. Retrieved 15.05.2023 from https://docs.oracle.com/javase/tutorial/uiswing/start/about.html

8. Oracle. (o.D.-b). *Class Graphics2D*. Oracle. Retrieved 10.04.2023 from https://docs.oracle.com/javase/8/docs/api/java/awt/Graphics2D.html

9. Oracle. (o.D.-c). *Coordinates*. Oracle. Retrieved 05.05.2023 from https://docs.oracle.com/javase/tutorial/2d/overview/coordinate.html

10. Oracle. (o.D.-d). Images. Oracle. Retrieved 05.05.2023 from https://docs.oracle.com/javase/tutorial/2d/overview/images.html

11. Oracle. (o.D.-e). *Lesson: Getting Started with Graphics*. Oracle. Retrieved 01.05.2023 from https://docs.oracle.com/javase/tutorial/2d/basic2d/index.html

12. Oracle. (o.D.-f). *Lesson: Overview of the Java 2D API Concepts*. www.docs.oracle.com. Retrieved 01.05.2023 from https://docs.oracle.com/javase/tutorial/2d/overview/index.html

13. Oracle. (o.D.-g). Trail: 2D Graphics.Oracle Retrieved 20.05.2023 from https://docs.oracle.com/javase/tutorial/2d/index.html

14. Oracle. (o.D.-h). Transforming Shapes, Text and Images. Retrieved 13.05.2023 from https://docs.oracle.com/javase/tutorial/2d/advanced/transforming.html

15. Schäling, B. (2010). *Programmieren in Java: Aufbau*. www.highscore.de.

16. Sun-Microsystems. (1999). *1.2 Rendering Model*. Nickerson Group at University of Washington. Retrieved 15.05.2023 from https://nick-lab.gs.washington.edu/java/jdk1.3.1/guide/2d/spec/j2d-intro.fm2.html