Extending JDORFX: A JavaFX Graphics Framework for ooRexx

Improving JDOR Compatibility, Extending 3D Functionality, and Adding Animation Support

by

Alireza Ismaili

A thesis submitted in partial fulfillment
of the requirements for the degree of
Bachelor of Science (WU)
in the
Bachelor's Degree Program in Business, Economics and Social Sciences
at
Vienna University of Economics and Business

Supervisor: ao.Univ.Prof.Mag.Dr.rer.soc.oec Rony G. Flatscher
Student ID: 12035862
Department of Information Systems & Society
October 2026

Abstract

JDORFX[1] is a JavaFX-based graphics framework that exposes graphics functionality to Open Object Rexx (ooRexx) programs through the BSF4ooRexx command-handler infrastructure. It was developed as a JavaFX-based counterpart to JDOR, which uses Java2D. The inherited JDORFX implementation already provided substantial 2D drawing functionality and basic 3D support, but 31 canonical JDOR commands were still unimplemented and several behavioral differences remained because Java2D and JavaFX use fundamentally different graphics models.

This thesis extends JDORFX in three main areas. First, the previously missing JDOR functionality is implemented and compatibility behavior is revised where direct translation to JavaFX is insufficient. This includes image handling, printing, clipboard access, clipping, compositing, graphics-state management, and related operations. Architectural changes reconcile Canvas-based drawing with retained JavaFX Scene Graph nodes and improve JavaFX thread and window handling. Second, six additional 3D-control commands provide camera translation and rotation, camera clipping-plane control, PointLight translation, and Shape3D face culling. Third, 18 animation commands expose JavaFX transitions, timelines, key frames, key values, interpolators, playback control, and sequential and parallel animation composition to ooRexx.

The final comparison with JDOR covers 100 canonical commands: 92 are implemented without a known compatibility limitation, three are implemented with documented JavaFX platform limitations, five remain partially compatible, and none remains completely unimplemented. The documented JDORFX command interface contains 140 canonical commands. The resulting framework therefore covers the complete canonical JDOR command set considered in this thesis while also exposing JavaFX-specific 3D and animation functionality through the existing command-oriented abstraction.

[1] Source code and samples: https://github.com/aismaili10/jdorfx_pub [6].

Acknowledgements

I would like to thank Prof. Rony G. Flatscher for his guidance, valuable feedback, and continued support throughout the development of this thesis.

Contents

Abstract III

List of Abbreviations IX

List of Figures XI

List of Tables XIII

List of Listings XV

1 Introduction 1
- 1.1 Motivation . . . 1
- 1.2 Problem Statement . . . 2
- 1.3 Objectives of the Thesis . . . 3
- 1.4 Scope of the Thesis . . . 4
- 1.5 Structure of the Thesis . . . 4

2 Technical Background 7
- 2.1 Open Object Rexx . . . 7
- 2.2 Bean Scripting Framework for ooRexx . . . 8
- 2.3 Rexx Command Handlers . . . 8
- 2.4 JDOR . . . 9
- 2.5 JavaFX . . . 10
- 2.6 JDORFX . . . 10
- 2.7 Relevant JavaFX Graphics Concepts . . . 11
  - 2.7.1 Scene Graph and Nodes . . . 11
  - 2.7.2 Canvas and GraphicsContext . . . 11
  - 2.7.3 2D and 3D Shapes . . . 12
  - 2.7.4 Cameras and Lights . . . 12
  - 2.7.5 Materials and Images . . . 13
  - 2.7.6 Transformations and Animations . . . 13

3 Existing JDORFX Implementation 15
3.1 Original Project State . . . 15
3.2 Architecture and Command Processing . . . 16
3.3 Relationship Between JDOR and JDORFX . . . 18
3.4 Limitations and Areas for Extension . . . 19

4 Design and Extension of JDORFX 21
4.1 Design Goals . . . 21
4.1.1 Maintaining Compatibility with JDOR . . . 22
4.1.2 Command Design and Canonical Output . . . 22
4.1.3 Error Handling and Validation . . . 23
4.1.4 Compatibility Comparison . . . 23
4.2 Missing JDOR Functionality . . . 24
4.2.1 Image-Related Commands . . . 24
4.2.1.1 Loading and Saving Images . . . 24
4.2.1.2 Drawing Images . . . 26
4.2.1.3 Image Copying and Manipulation . . . 27
4.2.1.4 Image Query Commands . . . 27
4.2.2 Print-Related JDOR Commands . . . 28
4.2.3 Additional JDOR Commands . . . 28
4.3 Compatibility Corrections and Architectural Improvements . . . 31
4.3.1 Rendering and Geometry Compatibility . . . 31
4.3.2 Window Lifecycle and Update Behavior . . . 35
4.3.3 JavaFX Thread Handling and Multiple Handlers . . . 35
4.4 New 3D Functionality . . . 38
4.4.1 Camera Transformation Commands . . . 38
4.4.2 Camera Clipping Plane Commands . . . 40
4.4.3 Light Transformation Commands . . . 40
4.4.4 Cull Face Support . . . 40
4.5 Animation Support . . . 44
4.5.1 JavaFX Animation Model . . . 44
4.5.2 Animation Creation . . . 44
4.5.3 Animation Control . . . 45
4.5.4 Timeline-Based Animation and Camera Animation . . . 46
4.5.5 JavaFX Animation Basics and JDORFX . . . 47
4.5.6 Comparing JDOR and JDORFX Animation . . . 49

4.5.7 Complex JavaFX Animation Case Study . . . 51
4.6 Input and Output Redirection . . . 54
4.6.1 Canonical Command Output . . . 54
4.6.2 Recording JDORFX Macros . . . 55
4.6.3 Replaying Recorded Commands . . . 55
4.6.4 Textual Command Interchange . . . 57

5 Compatibility, Limitations, and Known Issues 59
5.1 Compatibility with JDOR . . . 59
5.2 Differences Between Java2D and JavaFX . . . 60
5.3 Remaining JDOR Compatibility Constraints . . . 61
5.4 Known JDOR Issues Affecting JDORFX . . . 62
5.5 JavaFX-Specific Limitations . . . 63
5.6 Final Compatibility Status . . . 64

6 Conclusion 65
6.1 Summary of the Work . . . 65
6.2 Main Contributions and Results . . . 66
6.3 Future Work . . . 67

References 69

A Complete JDORFX Command Interface 75
A.1 Command Syntax and Conventions . . . 75
A.2 Window and Scene Commands . . . 75
A.3 2D Drawing Commands . . . 76
A.4 2D Shape Commands . . . 78
A.5 Image, Clipboard, and Print Commands . . . 79
A.6 2D Transformation Commands . . . 81
A.7 3D Shape Commands . . . 81
A.8 Camera Commands . . . 81
A.9 Light Commands . . . 82
A.10 Material and Texture Commands . . . 82
A.11 3D Transformation Commands . . . 82
A.12 Cull Face Commands . . . 83
A.13 Animation Commands . . . 83

B Selected Complete Examples 85
B.1 Image Loading, Drawing, and Saving 85
B.2 CullFace 87
B.3 Extended Camera Controls 89
B.4 JDORFX Transition Comparison 91
B.5 Oracle Tree Animation Case Study 93
B.6 Canonical Recording and Replay 99

List of Tools and Aids Used 101

List of Abbreviations

2D	two-dimensional
3D	three-dimensional
API	application programming interface
AWT	Abstract Window Toolkit
BSF4ooRexx	Bean Scripting Framework for ooRexx
JDOR	Java Drawing for ooRexx
JDORFX	JavaFX-based counterpart to JDOR
LOC	lines of code
PNG	Portable Network Graphics
RC	Rexx return-code variable
XOR	exclusive OR

List of Figures

3.1 Inherited and revised JDORFX scene hierarchies 17
3.2 Simplified JDOR and JDORFX rendering architectures 18

4.1 Deterministic PNG output generated by the JDORFX image-command sample: scaled full image (left) and scaled source-region crop (right) 26
4.2 JDORFX Paint, alpha-compositing, and XOR demonstration: gradient and half-transparent `SRC_OVER` rendering (left), one XOR application that remains visible (center), and restoration after applying the same XOR geometry twice (right) 30
4.3 Committing retained 2D rendering to the Canvas 32
4.4 JDORFX clipping and raster-copy demonstration: intersected rectangular and Shape clipping in the labelled original region (left), and the displaced `copyArea` result with a white region subsequently produced by `clearRect` (right) 34
4.5 JDORFX command threads, shared JavaFX runtime, and handler-owned windows 36
4.6 The fixed box before and after translating and rotating the camera 39
4.7 JDORFX CullFace results for one internally lit box shell under five combinations of culling mode and camera viewpoint 43
4.8 Representative frames exported from the JDORFX counterpart of Oracle's basic Timeline example 48
4.9 Representative output states of the JDORFX `RotateTransition` used for the 2-130 JDOR/JDORFX comparison 50
4.10 Representative states exported from the JDORFX Oracle Tree Animation case study 53
4.11 Canonical JDORFX command recording, storage, and replay workflow 57

B.1 Three stages of the JDORFX 2-130 bitmap rotation over 3033 ms 91
B.2 Five stages of the JDORFX Oracle Tree case study, captured during playback of the submitted program 93

List of Tables

4.1 Overview of the main JDORFX extensions 21
4.2 JDORFX animation-control commands 45
4.3 Mapping Oracle's basic Timeline example to JDORFX 47
4.4 Static source measurements for the JDOR animation comparisons 51
4.5 Redirected canonical handler traffic 51

5.1 Summary of the current JDORFX compatibility status 64

List of Listings

4.1 Excerpt from the JDORFX image-command demonstration 25
4.2 Excerpt from the JDORFX Paint, Composite, and XOR demonstration 29
4.3 Excerpt from `JavaFXDrawingHandler`: committing retained 2D rendering to the Canvas 32
4.4 Excerpt from the JDORFX clipping and copy-area demonstration 33
4.5 Excerpt from `JavaFXDrawingHandler`: synchronous JavaFX-thread dispatch . . . 37
4.6 Excerpt from the JDORFX extended-camera demonstration 38
4.7 Excerpt from the JDORFX CullFace demonstration 41
4.8 Oracle's basic Timeline animation expressed through JDORFX 47
4.9 Replacing the 2-130 redraw loop with a named RotateTransition 49
4.10 Composition structure of the JDORFX Tree Animation case study 52
4.11 Excerpt from `JavaFXDrawingHandler`: redirected-input processing and `RC` propagation 55
4.12 Excerpt from the JDORFX canonical recording and replay demonstration . . . 56

B.1 Complete runnable JDORFX image-command demonstration 85
B.2 Complete runnable JDORFX CullFace demonstration 87
B.3 Complete runnable JDORFX extended-camera demonstration 89
B.4 Complete runnable JDORFX 2-130 RotateTransition comparison 91
B.5 Complete runnable JDORFX Oracle Tree Animation case study 93
B.6 Complete runnable JDORFX canonical recording and replay example 99

I Introduction

JDORFX is a JavaFX-based graphics framework that makes graphics functionality available to Open Object Rexx (ooRexx) programs through a command-oriented interface. It builds on JDOR, whose commands expose Java2D drawing functionality to ooRexx, while using JavaFX as the underlying graphics framework. This also allows JDORFX to provide JavaFX-specific functionality such as retained Scene Graph objects and three-dimensional graphics. The version inherited at the beginning of this thesis had already established this basic architecture and provided substantial 2D drawing functionality and basic 3D support [42].

This thesis continues the development of JDORFX with two main goals: improving compatibility with the established JDOR interface and extending the JavaFX-specific functionality of the framework. Achieving both requires accounting for the different graphics models of Java2D and JavaFX. JDOR renders primarily into a raster image through `Graphics2D`, whereas JDORFX combines immediate-mode drawing on a JavaFX `Canvas` with retained Scene Graph nodes. Some extensions therefore require changes to graphics state, rendering, window handling, and thread interaction rather than a direct translation of individual Java2D operations.

1.1 Motivation

The command-oriented model used by JDOR and JDORFX allows an ooRexx program to perform graphical operations without directly constructing and coordinating the corresponding Java objects. This is useful in the context of BSF4ooRexx, which provides the bridge between ooRexx and Java: the application can remain an ooRexx program while the graphics implementation is provided by Java classes. JDOR established such an interface on top of Java2D, and JDORFX transferred the same general idea to JavaFX [42].

JavaFX offers functionality that goes beyond replacing one rendering API with another. Its Scene Graph provides retained graphical objects, while its graphics API also includes 3D shapes, cameras, lights, materials, and a dedicated animation framework. These features allow JDORFX to support operations that are not available through the original two-dimensional JDOR model. At the same time, existing JDOR programs should require as few changes as practical when moved

to JDORFX. Compatibility therefore concerns not only command names, but also their behavior and visible results.

The inherited implementation did not yet achieve this completely. Schaller documented a number of canonical JDOR commands that were represented by the interface but not processed by the handler [38]. Some existing commands also behaved differently because concepts available in Java2D do not always have direct JavaFX equivalents. In addition, several useful JavaFX capabilities were not yet exposed through JDORFX, particularly additional control over cameras and lights and direct access to the JavaFX animation framework.

The aim of this thesis is not to expose the complete JavaFX API through JDORFX. Instead, it focuses on improving compatibility with JDOR and on adding selected JavaFX features that fit the existing command-oriented interface.

1.2 Problem Statement

The work in this thesis addresses three related problems in the inherited JDORFX implementation.

The first problem was incomplete compatibility with JDOR. Thirty-one canonical JDOR commands were still unimplemented, including substantial parts of image handling, printing, clipboard support, clipping, compositing, graphics-state management, and several drawing and geometry operations [38]. Adding support for these command names alone was not sufficient, since compatibility also depends on their arguments, return values, persistent state, and error behavior.

The second problem was architectural. JDOR renders primarily into a `BufferedImage` through a stateful `Graphics2D` context, whereas JDORFX combines `Canvas`/`GraphicsContext` drawing with retained JavaFX nodes. Once a Java2D drawing operation has been performed, its result forms part of the raster image. A retained JavaFX node, by contrast, continues to exist as an object that can later be modified.

This difference affects image retrieval, clipping, compositing, copying, destructive drawing, and the ordering of retained shapes relative to Canvas content. JavaFX also imposes lifecycle and threading requirements on `Stage` and Scene Graph operations. Some compatibility problems therefore required changes to the rendering and update architecture rather than direct translations of individual commands.

The third problem concerned JavaFX-specific functionality. The inherited implementation already supported primitive 3D shapes, cameras, lights, transformations, materials, and texture maps, but it did not expose several controls needed to manipulate an existing 3D scene. These included camera transformation, camera clipping distances, movement of point lights, and

face culling. It also did not provide a dedicated command interface to the JavaFX animation framework. Animation could be approximated through repeated commands and delays, but the ooRexx program itself had to perform each update instead of declaring an animation and leaving interpolation and timing to JavaFX.

The thesis therefore addresses both command-level compatibility and the architectural differences between Java2D and JavaFX, while also extending JDORFX with selected JavaFX-specific functionality.

1.3 Objectives of the Thesis

The first objective is to improve compatibility with JDOR. This includes implementing the 31 canonical JDOR commands that were still unprocessed by the inherited handler and correcting important behavioral differences in commands that already existed. Compatibility is considered at the behavioral level as well as at the level of command availability, including arguments, return values, graphics state, and error handling.

The second objective is to adapt JDORFX where differences between Java2D and JavaFX prevent a direct translation of JDOR semantics. This includes providing a consistent representation of the current 2D result despite the coexistence of Canvas pixels and retained Scene Graph nodes, maintaining additional compatibility state where necessary, and executing JavaFX operations on the appropriate thread.

The third objective is to extend the JavaFX-specific 3D command interface. The work adds commands for translating and rotating registered cameras, querying and changing their near and far clipping planes, translating registered point lights, and querying or changing the face-culling mode of 3D shapes.

The fourth objective is to introduce direct JavaFX animation support for ooRexx programs. The intended interface covers predefined transitions, playback control, interpolators, timelines, key frames, key values, timed pauses, and sequential and parallel composition. The animation model should work with retained 2D shapes as well as applicable 3D shapes, cameras, and lights.

The fifth objective is to preserve the textual command-processing model of JDORFX. Canonical command output and redirected input should remain usable with newly implemented functionality so that accepted command sequences can be recorded and replayed. This also provides a textual interchange boundary without exposing the internal JavaFX object model.

Finally, the resulting implementation is compared with JDOR to determine which differences remain. The JavaFX-specific extensions are demonstrated through runnable examples, visual results, and animation comparisons.

1.4 Scope of the Thesis

This thesis focuses on the further development of the JDORFX command handler together with the examples and documentation needed to demonstrate the resulting functionality. The work covers the previously missing JDOR functionality, architectural changes required to support it, selected extensions to the existing 3D interface, JavaFX animation support, canonical command recording and replay, and documentation of the resulting command set.

The thesis does not attempt to expose the complete JavaFX API through ooRexx. Facilities such as user-defined meshes through `TriangleMesh` and `MeshView`, general support for JavaFX effects, `SubScene`, charts, media playback, `WebView`, and event handling remain outside the implemented command set. The animation interface likewise exposes selected targets and properties rather than arbitrary JavaFX property binding, arbitrary event callbacks, or user-defined `Transition` subclasses.

Complete pixel identity between JDOR and JDORFX is also outside the scope of this work. Java2D and JavaFX use different rendering pipelines, object models, antialiasing behavior, image representations, and platform-dependent graphics implementations. The goal is to reproduce the public command semantics and observable behavior as closely as practical while documenting cases in which exact equivalence cannot be achieved. The thesis also makes no general performance claim, since it does not include comprehensive CPU, memory, or rendering-performance benchmarks.

1.5 Structure of the Thesis

Chapter 2 introduces the technical background required for the implementation. It covers ooRexx, BSF4ooRexx, Rexx command handlers, JDOR, JavaFX, and the JavaFX graphics concepts most relevant to JDORFX, including the Scene Graph, Canvas drawing, 2D and 3D shapes, cameras, lights, materials, transformations, and animation.

Chapter 3 describes the JDORFX implementation inherited at the beginning of this thesis. It summarizes the available functionality, the command-processing and rendering architecture, the relationship between JDOR and JDORFX, and the limitations that motivated the extensions developed in this work.

Chapter 4 presents the design and implementation of the extensions. It covers the previously missing JDOR functionality, compatibility corrections in rendering and window handling, JavaFX thread handling and support for multiple handlers, the new 3D controls, the animation interface, and canonical command recording and replay. Runnable examples, implementation excerpts, and visual results illustrate the most important additions.

Chapter 5 compares the resulting implementation with JDOR and documents the remaining limitations. It distinguishes between differences caused by Java2D and JavaFX, partially compatible commands, known defects in the JDOR reference implementation, and JavaFX functionality that remains outside the current JDORFX interface.

Chapter 6 summarizes the work and its principal results and identifies possible directions for future development. Appendix A provides a compact reference for the complete JDORFX command interface, while Appendix B contains selected complete runnable examples used in Chapter 4.

2 Technical Background

JDORFX brings together several technologies: the Open Object Rexx programming language, the BSF4ooRexx bridge between ooRexx and Java, the Rexx command-handler mechanism, and the JavaFX graphics framework. It also builds directly on JDOR, which exposes Java2D functionality through a Rexx command interface.

This chapter introduces these technologies and the JavaFX concepts needed to understand the implementation described in the following chapters. The focus is on the concepts that directly affect the architecture and functionality of JDORFX rather than on providing a general introduction to Java or graphical user-interface programming.

2.1 Open Object Rexx

Open Object Rexx (ooRexx) is an object-oriented implementation of the Rexx programming language. It retains the procedural programming model of traditional Rexx while adding classes, objects, methods, messaging, inheritance, and polymorphism. Procedural and object-oriented programming styles can therefore be used independently or combined within the same program [2, 3].

Rexx is an interpreted language and imposes comparatively few formatting requirements on source programs. Instructions can span several lines, multiple clauses can occur on one line, and keywords can be written using upper, lower, or mixed case. Variables do not require an explicitly declared data type and may refer to different kinds of objects [2, 3].

For JDORFX, an important feature of Rexx is its ability to issue commands to external command environments. A scriptable application can register such an environment with the Rexx language processor, after which commands can be directed to it. The `ADDRESS` instruction is used to select the environment that receives subsequent command clauses or to direct a single command to a particular environment [2, 3].

For example, `address jdorfx` selects `JDORFX` as the current command environment. Subsequent command clauses can then be handled by the implementation registered under that environment name rather than by the operating-system command processor.

After an external environment processes a command, it returns control to the Rexx interpreter and provides a return code. Rexx makes this return code available through the special variable `RC`. Command environments can also signal `ERROR` and `FAILURE` conditions, which Rexx programs can trap and handle [3, 2].

This command-oriented mechanism is fundamental to both JDOR and JDORFX. An ooRexx program normally interacts with the frameworks by issuing commands rather than by directly instantiating and manipulating the underlying Java graphics classes.

2.2 Bean Scripting Framework for ooRexx

The Bean Scripting Framework for ooRexx, commonly referred to as BSF4ooRexx, provides interoperability between ooRexx and Java. It acts as a bidirectional bridge through which Java classes and objects can be accessed from ooRexx and Java applications can in turn interact with ooRexx [4].

This interoperability is essential to JDORFX because the graphics framework is implemented in Java while its user-facing interface consists of ooRexx commands. BSF4ooRexx allows Java objects to be made available to ooRexx and provides the infrastructure needed to register Java implementations as Rexx command handlers.

In JDORFX, ooRexx provides the command-level interface, while Java implements the graphics operations. BSF4ooRexx connects these two sides, allowing an ooRexx program to use JavaFX functionality without requiring the programmer to write the corresponding Java code directly.

Schaller's original JDORFX implementation used this mechanism to register `JavaFXDrawingHandler` as the handler for the `JDORFX` environment [42]. An ooRexx program can therefore select the environment using `ADDRESS JDORFX` and send its graphics commands to the Java implementation.

2.3 Rexx Command Handlers

A Rexx command handler connects a named command environment to an implementation capable of processing commands sent to that environment. From the perspective of the Rexx program, a command is a character string. The selected environment receives this string, performs the corresponding operation, and then returns control to the Rexx interpreter [3].

The ooRexx program therefore does not need to know which Java methods implement an individual graphics operation. A program can contain commands such as

```
color red
moveTo 50 50
```

```
drawLine 100 100
```

without directly invoking the corresponding Java implementation. In JDORFX, the command handler is responsible for parsing the command, validating its arguments, maintaining the required state, and invoking the appropriate graphics operation.

The `ADDRESS` instruction controls which environment receives such commands. Since a Rexx program can address different environments during its execution, operating-system commands and commands belonging to a domain-specific environment can coexist in the same program [2, 3].

ooRexx also allows the standard input, standard output, and standard error of commands to be redirected using the `ADDRESS` instruction. Input can be supplied from Rexx objects, while output and error streams can be routed back to Rexx objects [3]. This mechanism is relevant to JDORFX because its command-handler interface also supports redirected command processing. Its use for canonical command output, recording, and replay is discussed later in this thesis.

2.4 JDOR

JDOR, short for *Java Drawing for ooRexx*, is a graphics command handler for ooRexx that exposes Java2D functionality through a command-oriented interface. It forms the conceptual and compatibility basis on which JDORFX was developed [42].

JDOR's central Java class is `JavaDrawingHandler`. Its rendering is based primarily on Java2D, using a `Graphics2D` object associated with a `BufferedImage`. The Java `Graphics2D` API maintains rendering attributes such as paint, stroke, font, clipping, compositing, and coordinate transformations and applies them when rendering shapes, text, or images [16]. A `BufferedImage` represents image data with an accessible image buffer and can itself provide a `Graphics2D` context for rendering into that image [14].

This model is well suited to a stateful command interface. Properties such as the current color, transformation, stroke, or drawing position can be maintained across successive commands and reused by later drawing operations. Once a drawing operation has been rendered into the target image, its result becomes part of that image.

For JDORFX, compatibility with JDOR involves more than matching command names. The JDOR interface also defines behavior concerning accepted arguments, query and return values, persistent graphics state, errors, and canonical command output. JDORFX therefore uses JDOR as a compatibility baseline at both the syntactic and behavioral levels.

The detailed relationship between JDOR and the inherited JDORFX implementation is discussed in Chapter 3.

2.5 JavaFX

JavaFX is a Java framework for graphical applications. This thesis targets JavaFX 8, whose APIs provide the application lifecycle, window and scene infrastructure, Scene Graph, shapes, transformations, Canvas drawing, images, effects, and animation APIs [22].

The `Application` class defines a JavaFX application's lifecycle. A `Stage` is a top-level window. Graphical content is placed in a `Scene`, which can be displayed in a Stage [11, 32, 30].

The contents of a Scene are organized as a hierarchy of graphical objects known as the Scene Graph. The individual elements of this hierarchy derive from `Node`. JavaFX nodes include, among others, containers, two- and three-dimensional shapes, cameras, lights, image views, and Canvas objects [21].

JavaFX also provides the `Canvas` API for direct drawing. Drawing commands are issued through a `GraphicsContext`, which maintains rendering state such as transforms, clipping, fill and stroke properties, fonts, and effects [17].

The coexistence of retained Scene Graph objects and Canvas-based drawing is especially relevant to JDORFX. Canvas operations provide a drawing model that is comparatively close to the stateful Java2D interface of JDOR, whereas JavaFX features such as 3D shapes, cameras, lights, materials, and node-based animation rely on retained Scene Graph objects.

The following subsections introduce the JavaFX graphics concepts that are most relevant to JDORFX in greater detail.

2.6 JDORFX

JDORFX is the JavaFX-based counterpart to JDOR and applies its command-oriented concept to JavaFX. The original implementation was developed by Philipp Schaller and forms the technical starting point of the present thesis [42].

The objective of the original project was to retain JDOR's ooRexx-oriented command interface while replacing the Java2D-based rendering implementation with JavaFX and introducing JavaFX 3D functionality. The inherited implementation already provided a substantial part of JDOR's 2D drawing interface as well as JavaFX-specific support for boxes, cylinders, spheres, parallel and perspective cameras, ambient and point lights, transformations, Phong materials, and image maps [42].

JDORFX is accessed through a Rexx command environment in the same general manner as JDOR. Its Java backend interprets textual commands and translates them into corresponding JavaFX operations. Its rendering architecture nevertheless differs from JDOR because JDORFX uses both Canvas drawing and retained Scene Graph objects.

The state of JDORFX before the work of this thesis is examined in detail in Chapter 3. The extensions introduced in this thesis are described in Chapter 4.

2.7 Relevant JavaFX Graphics Concepts

The following JavaFX concepts are needed to understand both the inherited JDORFX implementation and the extensions described later in this thesis.

2.7.1 Scene Graph and Nodes

The JavaFX Scene Graph is a hierarchical representation of graphical content. Every element in the Scene Graph is a `Node`. A node has zero or one parent and is either a leaf node or a branch within the tree. Branch nodes derive from `Parent` and may contain child nodes [21].

This is a retained-mode graphics model. When a `Shape` or another node is inserted into the Scene Graph, the object continues to exist after it has been displayed. Its properties can subsequently be queried or modified, and changes to those properties can affect the rendered result without replacing the object itself.

This differs from drawing a primitive into an image or Canvas. In an immediate drawing operation, the rendered pixels remain, but there is no automatically retained graphical object representing the original primitive.

Retained nodes are important for JavaFX 3D graphics and animation. Cameras, lights, 3D shapes, and ordinary Shape nodes remain addressable objects and can therefore be transformed or have their properties changed over time.

2.7.2 Canvas and GraphicsContext

A JavaFX `Canvas` is itself a Scene Graph node, but its visible contents are produced through an associated `GraphicsContext`. The `GraphicsContext` issues drawing operations to a buffer that is later rendered onto the Canvas [17].

The context maintains rendering attributes including the current transformation, clipping state, fill and stroke paints, line properties, font settings, global alpha, blend mode, and effects [17]. Drawing methods such as `fillRect()`, `strokeLine()`, and `drawImage()` use the currently configured attributes when issuing their rendering operations.

A shape drawn through a `GraphicsContext` does not become an independent JavaFX `Shape` node. For example, after `fillRect()` has been executed, the rectangle is part of the rendered Canvas contents rather than a separately addressable `Rectangle` object. By contrast, a `Rectangle` inserted into the Scene Graph remains a retained node.

The stateful nature of `GraphicsContext` makes it conceptually closer to Java2D's `Graphics2D` interface than the retained Scene Graph is. Canvas drawing is therefore suitable for translating many JDOR drawing operations, while retained nodes remain necessary for functionality such as 3D graphics and node-based animations.

JavaFX also imposes threading requirements on Canvas modification. Once a Canvas belongs to a Scene, modifications to the Canvas, including calls on its `GraphicsContext`, must occur on the JavaFX Application Thread [17]. The architectural consequences of this requirement are addressed in Chapter 3.

2.7.3 2D and 3D Shapes

JavaFX provides retained object representations for both two-dimensional and three-dimensional geometry. Retained two-dimensional geometric shapes derive from `Shape` and include classes such as `Line`, `Rectangle`, `Ellipse`, `Polygon`, curves, and `Path`. As Scene Graph nodes, they retain their properties and can be transformed or otherwise modified after being inserted into a Scene.

Three-dimensional shapes derive from `Shape3D`. JavaFX provides the primitive classes `Box`, `Cylinder`, and `Sphere`, as well as `MeshView` for displaying user-defined mesh data [31].

`Shape3D` defines properties shared by three-dimensional objects, including the material applied to the surface, the draw mode used for rendering the geometry, and face culling [31]. Draw mode determines whether the object is rendered as filled geometry or as lines. Face culling determines which oriented faces of the geometry are omitted from rendering.

These capabilities go beyond JDOR's two-dimensional image-oriented rendering model and form part of the JavaFX-specific functionality that can be exposed through JDORFX.

2.7.4 Cameras and Lights

A JavaFX `Camera` defines how Scene coordinates are mapped onto the displayed view. The abstract `Camera` class has two concrete subclasses, `ParallelCamera` and `PerspectiveCamera` [15].

A parallel camera renders the scene without perspective correction, while a perspective camera uses perspective projection. Perspective rendering therefore makes the apparent size of objects dependent on their position relative to the camera. Cameras also define near and far clipping distances, which delimit the visible depth range. Perspective cameras additionally provide a field-of-view setting [23].

Cameras derive from `Node`, which makes node transformations relevant to camera positioning and orientation. Translation and rotation can therefore be used to change the effective viewpoint.

Lighting is represented by subclasses of `LightBase`, the base class for JavaFX light sources [20]. An `AmbientLight` illuminates affected objects independently of a specific position or direction [9].

A `PointLight`, in contrast, acts as a point source whose position affects the illumination of 3D objects [28].

Light objects also participate in the Scene Graph. Relevant transformations, particularly translation of positional lights, can therefore change how 3D objects are illuminated.

2.7.5 Materials and Images

JavaFX represents the appearance of a three-dimensional surface through `Material`. The principal material implementation used for JavaFX 3D shapes is `PhongMaterial`, which is based on the Phong lighting model [24].

A `PhongMaterial` provides diffuse and specular color properties and can additionally use images as maps. These include diffuse, specular, bump, and self-illumination maps. A diffuse map contributes the visible surface color, a specular map influences reflection properties, a bump map describes surface-normal variations, and a self-illumination map contributes self-illuminated appearance [24].

JavaFX image data is represented primarily through `Image` and `WritableImage`. A `WritableImage` provides writable pixel storage and can receive image data from a `PixelReader`. `PixelReader` provides access to pixel data, while `PixelWriter` provides operations for modifying writable pixel surfaces [18, 34, 25, 26].

An important distinction exists between a JavaFX image object and the current contents of a Canvas. A Canvas is a live drawing surface and Scene Graph node, whereas an `Image` represents image data independently of that Canvas. This distinction becomes important when translating image-oriented JDOR functionality to JavaFX, particularly for operations involving image copying, querying, retrieval, or saving.

2.7.6 Transformations and Animations

JavaFX provides transformations for changing the coordinate mapping of graphical objects. The principal transformation types include translation, rotation, scaling, and shearing. JavaFX represents these operations through classes derived from `Transform`, including `Translate`, `Rotate`, `Scale`, `Shear`, and `Affine`.

The `Affine` class represents a general two- or three-dimensional affine transformation. It can combine sequences of rotations, translations, scales, and shears while preserving straight lines and parallelism [7]. Nodes also expose transformation-related properties such as translation, rotation, and scale values.

Many of these properties can be animated directly. The abstract `Animation` class provides the common lifecycle for JavaFX animations. Its controls include playing, pausing, stopping, playback rate, cycle count, and automatic reversal [10].

JavaFX provides transition classes for common predefined animations and `Timeline` for more general property-based animation. A `Timeline` consists of one or more `KeyFrame` objects. Individual key frames contain target values represented through `KeyValue` objects. JavaFX interpolates animated properties between their values over the course of the timeline [33].

For example, instead of repeatedly changing the position of a node in an application-controlled loop, a timeline can specify a target translation value at a particular time and allow JavaFX to interpolate the intermediate values.

This animation model works naturally with retained Scene Graph objects because their properties remain addressable while an animation is running. These JavaFX mechanisms provide the technical basis for the animation functionality added to JDORFX and discussed in Chapter 4.

3 Existing JDORFX Implementation

Before the work presented in this thesis began, JDORFX already existed as a functional JavaFX-based counterpart to JDOR. Philipp Schaller developed the original implementation as part of his bachelor thesis, and this version formed the technical starting point for the present work. Its objective was to retain the command-oriented interface of JDOR while replacing its Java2D-based graphics implementation with JavaFX and extending the framework with basic 3D functionality [42].

This chapter describes the state of JDORFX before the extensions developed in this thesis. It focuses on the inherited functionality, the architecture of the command handler, its relationship to JDOR, and the limitations that motivated further development.

3.1 Original Project State

The inherited version of JDORFX already operated as a Rexx command handler. It supported window management, basic 2D drawing, colors and backgrounds, fonts and strokes, 2D transformations, JavaFX Shapes, Paths, Area operations, and several state and utility operations. The inherited documentation distinguished between JDOR commands adopted directly and commands translated to corresponding JavaFX functionality [37, 39].

The main functionality beyond JDOR was JavaFX 3D support. JDORFX could create boxes, cylinders, and spheres and display them either as filled objects or as wireframe models. It also supported parallel and perspective cameras, ambient and point lights, transformations of 3D shapes, Phong materials, and image maps used by materials. These capabilities were made available through commands including `shape3d`, `camera`, `light`, `rotate3DShape`, `translate3DShape`, `material`, `materialMap`, and `setMaterial`. Schaller presented 3D graphics as the principal new capability of JDORFX in comparison with JDOR [42, 39].

The inherited implementation did not provide a dedicated command interface to the JavaFX animation classes. Commands such as `sleep` could be combined with repeated drawing or transformation operations to create command-driven sequences, but JavaFX animation objects themselves were not exposed through JDORFX [37, 39].

Compatibility with JDOR was also incomplete. Schaller documented 31 JDOR commands that were not implemented in JDORFX. These included printing and clipboard operations, most general image operations, gradient and paint functionality, composite and XOR modes, additional clipping operations, `copyArea`, graphics-context and image stacks, and retrieval of a Shape's `PathIterator` [38].

Commands that were already implemented did not always behave in the same way as their JDOR counterparts. For example, the inherited implementation documented limitations concerning changes to `winFrame` after a window had already been shown. Differences between the Java2D and JavaFX representations of Shapes and Paths could likewise lead to behavior that differed from the corresponding JDOR implementation [40, 42].

3.2 Architecture and Command Processing

`JavaFXDrawingHandler` is the central class of the inherited implementation. It extends JavaFX's `Application` class and implements `RexxRedirectingCommandHandler`, combining the JavaFX application lifecycle with the Rexx command-handler interface [37, 11].

Commands received from ooRexx enter the handler through the `handleCommand` callback. From there they are passed to `processCommand`, which separates the command name from its arguments. The command name is resolved to an internal enumeration value and processed by a large `switch` statement. The individual cases validate their arguments and perform the corresponding JavaFX operation. The handler also supports redirected input, through which several commands can be processed sequentially, and redirected output, through which the canonical representation of processed commands can be written [37].

The handler maintains graphics state across successive commands. This includes the current drawing coordinates, font and stroke properties, visibility and window settings, and the transformation stored by the Canvas `GraphicsContext`. Java objects that need to be referenced by later commands are stored in `HashMap` registries and associated with textual nicknames. The inherited implementation contained registries for colors, fonts, strokes, 2D Shapes, 3D Shapes, transformations, cameras, lights, materials, and texture maps. It did not yet contain a corresponding registry for animation objects [37].

The graphical contents were represented by `JavaFXDrawingFrame`, a subclass of `Scene`. Its root node was a JavaFX `Pane`. During construction, the Canvas and the three retained-object groups shown on the left of Figure 3.1 were inserted in that order [37]. For comparison, the right side shows the revised grouping introduced later in Section 4.3.1.

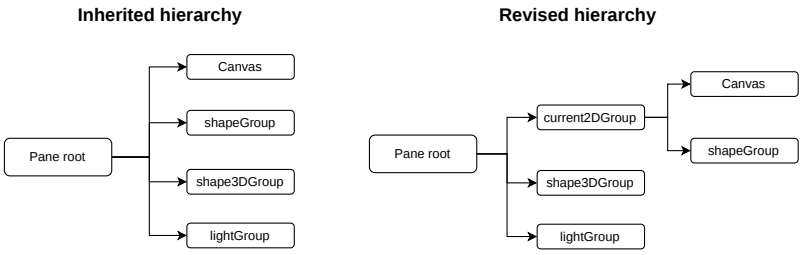


Figure 3.1: Inherited and revised JDORFX scene hierarchies

The Canvas was used primarily for immediate-mode 2D drawing through its `GraphicsContext`. The remaining groups stored retained Scene Graph objects such as JavaFX Shapes, 3D Shapes, and lights [17, 19].

The handling of JavaFX Shapes illustrates the difference between the two rendering approaches used in JDORFX. Ordinary drawing commands such as lines and rectangles were rendered into the Canvas through its `GraphicsContext`. Named JavaFX Shape objects, however, were stored separately and inserted into the Scene Graph when commands such as `drawShape` or `fillShape` were executed. Drawing properties and transformations maintained by the current graphics state therefore had to be transferred to the retained Shape object when it was displayed [37, 17, 19].

Retained Shape handling was not completely uniform in the inherited implementation. For example, `drawShape` and `fillShape` did not use precisely the same insertion mechanism, while `clipShape` used a Shape as a clip of the root node. These differences became important when the framework was extended because Scene Graph objects and Canvas content do not share a single rendering state [37].

Updating the graphical window required a separate mechanism. On the first command, the handler started the JavaFX application through a separate thread. After command processing, the current `JavaFXDrawingFrame` reference was placed in a `ConcurrentLinkedDeque`, and a flag indicated that a Scene update was available. The deque stored references to the Scene rather than independent copies of the Scene for every processed command [37].

Within `start(Stage)`, another thread repeatedly scheduled an updater on the JavaFX Application Thread by calling `Platform.runLater()`. The loop waited approximately 10 milliseconds between iterations. When a Scene update was available and window updating was enabled, the updater assigned the Scene to the Stage and applied pending Stage-related settings such as the title, decoration, location, resizability, visibility, and always-on-top state [37, 27].

This design separated Rexx command processing from Stage updates. It did not, however, provide a general mechanism that synchronously routed every modification of retained Scene Graph

objects through the JavaFX Application Thread. This became important for later extensions that modified retained JavaFX objects and introduced JavaFX animation functionality [37, 11].

3.3 Relationship Between JDOR and JDORFX

JDORFX retains JDOR's command-oriented interface. In both frameworks, an ooRexx program sends textual commands to a Rexx command environment rather than interacting directly with the underlying Java graphics API. Commands can therefore modify and reuse graphics state across several operations [42, 37].

The main difference is the underlying rendering technology. JDOR is based on Java2D and uses a `Graphics2D` object to draw into a `BufferedImage`. JDORFX instead uses JavaFX. For ordinary 2D drawing, the closest equivalent is the `GraphicsContext` associated with a JavaFX Canvas. Schaller's comparison of the two implementations already describes the replacement of `Graphics2D` and `BufferedImage` by `GraphicsContext` and Canvas as a central architectural difference [42, 16, 14, 17].

The distinction is particularly important for Shapes. When a Java2D Shape is drawn through `Graphics2D`, the result becomes part of the target image. In the inherited JDORFX implementation, a JavaFX Shape instead remained a retained Scene Graph object. Commands such as `drawShape` and `fillShape` therefore could not simply use equivalent Canvas drawing operations. The Shape had to be configured using the current graphics state and inserted into the Scene Graph [42, 37, 19].

Figure 3.2 summarizes the two rendering architectures and the additional branch introduced by retained JavaFX nodes.

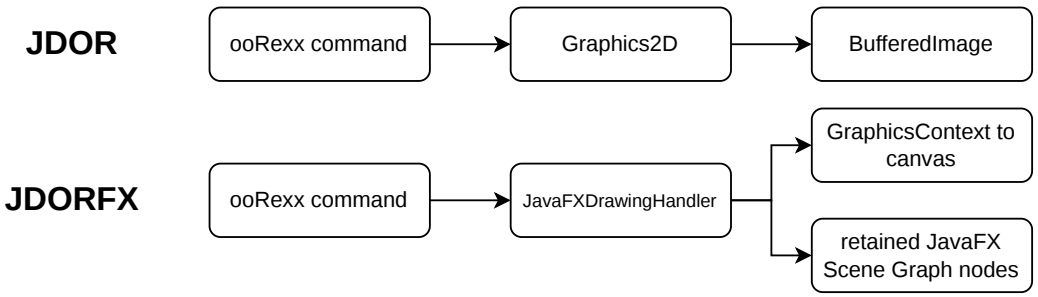


Figure 3.2: Simplified JDOR and JDORFX rendering architectures

JDOR consequently follows a predominantly raster-oriented drawing model, whereas JDORFX combines immediate-mode Canvas rendering with retained-mode Scene Graph rendering.

This distinction has consequences for compatibility. Content rendered into the Canvas becomes part of its raster content, whereas retained Shapes remain independent JavaFX nodes.

Their visual order is influenced by the Scene Graph hierarchy in addition to the sequence of commands. Operations that assume one authoritative image or graphics context, such as copying, saving, clipping, compositing, or retrieving image content, may therefore require additional handling rather than a direct one-to-one translation from Java2D to JavaFX [42, 17, 19].

Retained nodes are also the basis for JavaFX features such as 3D graphics and animation. JDORFX therefore cannot simply replace retained rendering with Canvas drawing in order to reproduce JDOR's raster-oriented model [19].

3.4 Limitations and Areas for Extension

The inherited implementation required further development in two main areas: compatibility with JDOR and JavaFX-specific functionality.

The missing JDOR functionality was concentrated in several areas, particularly image handling. Commands such as `loadImage`, `drawImage`, `saveImage`, `imageCopy`, `imageSize`, and `imageType` were not implemented. Printing and clipboard operations, composite and XOR modes, gradient and general Paint support, graphics-context stacks, image stacks, and several clipping operations were also absent [38].

Existing commands also had to match JDOR closely enough in their accepted arguments, return values, canonical output, error handling, graphics-state effects, object handling, and resulting rendering behavior. Command presence alone therefore did not establish behavioral compatibility.

The inherited project documentation also proposed several JavaFX-specific extensions, including enabling the Scene depth buffer, transformations of Camera objects, camera near and far clipping planes, transformations of PointLights, and access to the `cullFace` property of 3D Shapes. Dedicated JavaFX animation support was another possible extension that had not been part of the inherited implementation [41].

Other proposed areas included `TriangleMesh`, effects, `SubScene`, charts, media, `WebView`, and event handling. These were possible directions for continued development rather than requirements that all had to be implemented within the scope of this thesis [41].

The architecture itself also imposed constraints. Schaller noted that further optimization could improve performance, particularly for 3D graphics, and that long-running programs could block the user interface and make it unresponsive. The interaction between Canvas rendering, retained Scene Graph objects, command processing, and separately scheduled Stage updates therefore had to be considered when extending the framework [42].

These limitations provide the starting point for the extensions described in Chapter 4. The remaining differences between JDOR and the final JDORFX implementation are discussed in Chapter 5.

4 Design and Extension of JDORFX

Building on the inherited JDORFX implementation described in Chapter 3, this chapter presents the extensions developed in this thesis. The work comprises three main areas. First, the 31 JDOR commands that were declared but not processed by the inherited handler were implemented [38]. Second, existing rendering, window-management, and threading behavior was revised where differences between Java2D and JavaFX prevented the inherited implementation from reproducing JDOR semantics consistently. Third, 24 new JDORFX command entries were introduced: six commands for additional 3D control and 18 commands for JavaFX animation functionality.

The following sections focus on the design decisions and implementation techniques behind these extensions rather than providing a complete command-by-command manual. The full JDORFX command interface is documented in Appendix A. Table 4.1 summarizes the main extension areas covered in this chapter.

Table 4.1: Overview of the main JDORFX extensions

Area	Contribution
JDOR compatibility	Implementation of the 31 JDOR commands that were still unimplemented in the inherited JDORFX handler.
Rendering and architecture	Compatibility corrections for the mixed Canvas/Scene Graph model, graphics state, clipping, compositing, window behavior, and JavaFX thread handling.
3D functionality	Six commands for camera transformations, camera clipping planes, PointLight translation, and face culling.
Animation	Eighteen commands exposing JavaFX transitions, timelines, key frames, key values, interpolators, composition, and playback control.

4.1 Design Goals

The implementation follows two main design principles: JDOR commands should retain their established semantics as closely as JavaFX permits, while JavaFX-specific functionality should remain available where it does not conflict with those semantics.

This requires more than translating individual API calls. JDOR is based primarily on Java2D and a `Graphics2D` context associated with a `BufferedImage`, whereas JDORFX combines immediate-mode drawing on a JavaFX `Canvas` with retained Scene Graph nodes [16, 14, 17, 19]. As discussed in Chapters 2 and 3, some JDOR operations therefore require additional state management, conversion, or synchronization in JDORFX.

The implementation preserves retained graphical objects, 3D scene elements, and animation where these JavaFX features remain compatible with the intended command behavior.

4.1.1 Maintaining Compatibility with JDOR

Compatibility with JDOR was treated as behavioral rather than merely syntactic. Implementing a command with the same name is insufficient if it accepts different arguments, returns a different value, modifies different graphics state, or handles invalid input differently.

For commands originating from JDOR, the existing JDOR interface therefore served as the reference. The implementation preserves command syntax and aliases, query/set behavior, return values, nickname handling, canonical command output, and error conditions where the JavaFX APIs permit the same semantics.

Several areas require explicit compatibility state because JavaFX does not provide direct equivalents for all Java2D concepts. For example, JavaFX `Image` objects do not expose the storage types of `BufferedImage`, while JavaFX Canvas clipping cannot be queried and managed in exactly the same manner as a Java2D user clip [14, 18, 17]. These differences are handled by maintaining additional metadata or compatibility state rather than by changing the JDOR command interface.

Where exact equivalence cannot be achieved because of differences between Java2D and JavaFX, the implementation preserves the closest practical behavior and documents the remaining difference as a limitation. These cases are discussed in Chapter 5.

4.1.2 Command Design and Canonical Output

New commands follow the command-oriented structure inherited from JDOR and JDORFX. Each command is represented internally by an enumeration value and is resolved case-insensitively from its command name or one of its registered aliases. The corresponding branch of `processCommand` validates the arguments and performs the requested operation.

JavaFX objects that need to be referenced by later commands are generally associated with textual nicknames. This design is used for Shapes, 3D Shapes, transforms, cameras, lights, materials, images, and animations. The ooRexx program can therefore work with named graphical objects without directly managing Java references.

Canonical output is treated as part of the command interface. When command output is redirected, successfully processed commands are written in a normalized representation. Depending on the command, this can include a canonical command name, normalized booleans, integer values, symbolic constant names, or Rexx array expressions.

This representation is particularly useful for command recording and replay, which are discussed in Section 4.6. The implementation therefore constructs canonical output not merely as diagnostic information but as a reproducible form of an accepted command.

4.1.3 Error Handling and Validation

Command processing distinguishes between unknown commands, invalid arguments, missing registered objects, and failures occurring while a valid operation is being performed. These conditions are returned through the Rexx command-handler mechanism.

Where possible, validation is completed before persistent graphics state is modified. This is especially relevant for commands that create or install objects such as Strokes, transforms, image types, clipping distances, or Composite settings. Invalid input should not leave the handler in a partially updated state.

Nickname-based commands additionally validate the expected object type. For example, Camera commands require a registered Camera, and the command for translating a PointLight rejects other Light types.

The same principle applies where JavaFX and JDOR representations differ. The implementation does not invent Java2D-specific information when it is not available. An arbitrary JavaFX Image, for instance, does not intrinsically have a JDOR-compatible `BufferedImage` storage type. Such cases are reported explicitly rather than returning artificial metadata.

4.1.4 Compatibility Comparison

JDOR served as the reference for commands originating from the original interface. Schaller's documentation listed 31 commands that were represented by the JDORFX interface but not processed by the inherited handler [38].

The comparison considered command availability together with accepted arguments, return values, canonical output, graphics-state changes, object handling, and documented behavior. Existing commands were also compared where the JavaFX implementation differed from the corresponding JDOR semantics.

The final compatibility results are presented in Chapter 5.

4.2 Missing JDOR Functionality

One of the central objectives of the extension was to close the explicit command gap between JDOR and the inherited JDORFX implementation. Schaller documented 31 JDOR commands that were already represented by the command interface but were not processed by the handler [38].

The missing functionality covered image management, printing, clipboard access, Paint and Composite functionality, clipping, graphics-state management, image stacks, and several additional drawing and geometry operations.

As discussed in Section 3.3, implementing these commands required more than a direct translation between Java APIs. The differences between JDOR's raster-oriented rendering model and JDORFX's combination of Canvas drawing and retained Scene Graph objects required additional compatibility state and conversion mechanisms.

4.2.1 Image-Related Commands

Image handling represented one of the largest remaining gaps in the inherited implementation. The missing commands included `loadImage`, `saveImage`, `drawImage`, `image`, `imageCopy`, `imageSize`, `imageType`, `preferredImageType`, `pushImage`, and `popImage` [38].

The complete image-command syntax is summarized in Section A.5 of Appendix A.

A central difficulty is the difference between JDOR and JavaFX image representations. JDOR operates on `BufferedImage` objects and can therefore expose their underlying storage type. JavaFX `Image` objects do not provide an equivalent set of selectable `BufferedImage.TYPE_*` layouts [14, 18].

The extended handler therefore maintains JavaFX images in a nickname registry while separately tracking JDOR-compatible image-type metadata when such information is known. Conversion between JavaFX Images and `BufferedImage` instances is performed when a JDOR command requires a Java2D image representation.

The current drawing surface requires additional handling because the visible 2D result is not necessarily contained exclusively in the Canvas. Retained 2D Shape renderings may exist alongside Canvas pixels. Operations that require the current image therefore use a snapshot of the complete JDOR-compatible 2D presentation rather than a snapshot of the Canvas alone. JavaFX-specific 3D objects, lights, cameras, and Stage presentation are not part of this image representation.

4.2.1.1 Loading and Saving Images

The `loadImage` command loads an external image and associates it with a nickname. The JavaFX `Image` is stored in the image registry for later drawing and image-management operations.

Where possible, the same source file is additionally examined through Java image I/O to determine its `BufferedImage` type. This value is kept as compatibility metadata. The command returns the width and height of the loaded image, corresponding to JDOR behavior.

The `saveImage` command operates on the current complete 2D rendering. The combined 2D content is first snapshotted into a `WritableImage`, converted to a `BufferedImage`, and written through `ImageIO`. The requested file extension determines the output format; PNG is used as the default format when no format can be obtained from the filename.

This approach preserves the image-oriented JDOR interface while accounting for the fact that JDORFX can keep some visible 2D content as retained Scene Graph nodes.

Listing 4.1 demonstrates the image commands; license text, handler setup, result queries, window lifetime, and shutdown are omitted. The complete program is `samples/12_jdorfx_images.rxj` and is reproduced as Listing B.1.

```
1  -- newImage [width height [type [depthBuffer]]]
2  -- newImage width height PREFERRED depthBuffer
3  newImage 520 300
4  -- winTitle [title]
5  winTitle "JDORFX images"
6  -- background [colorName]
7  background WHITE
8  -- loadImage imageName fileName
9  loadImage LOGO "assets/oorexx_256.png"
10 -- imageSize [imageName]
11 imageSize LOGO
12 say "-- Source dimensions:" rc
13 -- imageType [imageName]
14 imageType LOGO
15 say "-- Source image type:" rc
16 -- preferredImageType [type]
17 preferredImageType
18 say "-- Preferred snapshot type:" rc
19
20 -- moveTo [x [y]]
21 moveTo 15 20
22 -- drawImage imageName [width height [sourceX1 sourceY1 sourceX2 sourceY2]]
       [backgroundColor]
23 drawImage LOGO 220 220
24 moveTo 265 20
25 drawImage LOGO 220 220 48 48 208 208 WHITE
```

```
26 -- The extension determines the output format.
27 -- saveImage fileName
28 saveImage "12_jdorfx_images.png"
```

Listing 4.1: Excerpt from the JDORFX image-command demonstration

Figure 4.1 shows the PNG written by the `saveImage` command in the listing. The left image is the complete logo scaled to 220 × 220 pixels. The right image uses the source-region form of `drawImage`, so its different crop and scale demonstrate that the new implementation does more than load and redisplay an unchanged file.



Figure 4.1: Deterministic PNG output generated by the JDORFX image-command sample: scaled full image (left) and scaled source-region crop (right)

4.2.1.2 Drawing Images

The `drawImage` implementation supports the argument forms required by JDOR. An image can be drawn using its original size, scaled to a requested destination size, or rendered from a selected source region. An optional background color is also supported.

Images are first resolved by nickname from the JDORFX image registry. If no managed image exists, a Rexx context variable with the supplied name is examined. JavaFX `Image` and Java `BufferedImage` objects can both be accepted, with a `BufferedImage` converted to a JavaFX Image before rendering.

The actual drawing operation is performed through the Canvas `GraphicsContext`. For source-region drawing, the two supplied source corners are translated into the width and height expected by the JavaFX `drawImage` overload [17].

Where a background color is supplied, the destination region is covered by that color before the image is rendered. This reproduces the corresponding JDOR command form without requiring the caller to manage a separate background operation.

4.2.1.3 Image Copying and Manipulation

The commands `imageCopy`, `pushImage`, and `popImage` provide image duplication and temporary image storage.

`imageCopy` creates an independent copy of either the current 2D rendering or a named image. Pixel data is copied to a `WritableImage`. If a background color is supplied, source pixels containing transparency are composited with that background. The resulting image is returned as a `BufferedImage`, preserving known type metadata where practical.

The image stack used by `pushImage` and `popImage` is maintained separately from the JavaFX Scene Graph. `pushImage` captures the current complete 2D rendering, converts it to a `BufferedImage`, and places it on the stack. If a nickname is supplied, the image is additionally registered for later lookup.

`popImage` removes the most recently pushed image and renders it back onto the current Canvas. An attempt to pop from an empty image stack results in an error condition.

This design preserves the image-stack abstraction of JDOR while hiding the internal distinction between retained 2D nodes and Canvas content from the ooRexx program.

4.2.1.4 Image Query Commands

The remaining image commands expose information about the current or named images.

`imageSize` returns the dimensions of the current drawing surface or of a named image. Compatible image objects supplied through Rexx variables can also be resolved.

`image` returns either the current complete 2D rendering or a named managed image. Since the JDOR interface exposes a Java2D image object, the result is returned as a `BufferedImage`.

The `preferredImageType` command maintains the default `BufferedImage` type associated with newly created JDOR-compatible images. Numeric values and symbolic `BufferedImage.TYPE_*` names are accepted.

The `imageType` command exposes type information for the current drawing surface or a named image. Because an arbitrary JavaFX `Image` does not possess a native `BufferedImage` storage layout, a type is returned only where compatible metadata is actually known.

This behavior illustrates the general compatibility strategy: JDOR semantics are reproduced where the corresponding information exists, while differences in the JavaFX object model are exposed rather than hidden through invented values.

4.2.2 Print-Related JDOR Commands

The inherited JDORFX implementation did not process the four printing commands `printScaleToPage`, `printScale`, `printPos`, and `printImage` [38].

The extended implementation keeps the printing configuration as persistent handler state. `printScaleToPage` determines whether an image should be scaled automatically to the printable area. `printScale` stores horizontal and vertical scale factors for manual scaling, while `printPos` stores the requested print position.

`printImage` can print either the current 2D rendering or a named image. Printing uses the JavaFX printing API. A temporary Canvas is created with the printable dimensions of the selected page layout, and the image is rendered onto this Canvas using the current print configuration.

When scale-to-page mode is enabled, a uniform scale factor is calculated so that the image fits inside the printable width and height. Otherwise, the stored horizontal and vertical scale factors and print position are used. The resulting Canvas is submitted through a JavaFX `PrinterJob` [29].

As with saving and retrieving images, printing the current image uses the JDOR-compatible 2D snapshot rather than the entire JavaFX Scene.

4.2.3 Additional JDOR Commands

The remaining previously unimplemented commands cover several areas of the JDOR interface.

Clipboard support was added through `clipboardGet`, `clipboardSet`, and `clipboardSetWithoutAlpha`. `clipboardGet` retrieves an image from the system clipboard and makes it available through the JDORFX image registry. `clipboardSet` places either the current image or a named image on the clipboard. The `clipboardSetWithoutAlpha` variant performs the corresponding JDOR-style alpha-removal conversion for image types for which that conversion is applicable.

Paint functionality was extended through `gradientPaint` and `paint`. Linear gradients are represented by JavaFX `LinearGradient` objects and stored under nicknames. The `paint` command can then select either a registered gradient or Color as the current Paint.

The commands `composite`, `setPaintMode`, and `setXorMode` required additional compatibility logic because JavaFX does not directly expose all Java2D drawing modes. Java2D provides compositing through `AlphaComposite`, whereas JavaFX offers its own blending operations [8, 13]. The handler therefore maintains an authoritative logical `AlphaComposite` state. Composite rules that correspond directly to suitable JavaFX blending can use native Canvas state, while other Porter–Duff rules are handled through a software raster path. XOR mode similarly uses explicit compatibility state.

The final static result in Figure 4.2 makes these modes directly comparable. The left region selects a JavaFX gradient through the JDOR-compatible Paint interface and overlaps it with a blue

oval using `SRC_OVER` at an alpha value of 0.5. In the other two regions, the same XOR geometry is applied once and twice respectively: one application leaves a visible changed region, whereas the second application restores the pixels that preceded the first. Listing 4.2 shows the relevant commands from `samples/15_jdorfx_paint_composite_xor.rxj`. Panel construction, labels, handler setup, and window lifetime are omitted.

```
-- A half-opaque blue source overlaps the gradient.
-- composite [rule [alpha]]
composite SRC_OVER 0.5
-- paint [colorName|gradientName]
paint BLUE
moveTo 150 100
-- fillOval width height
fillOval 240 150
composite SRC_OVER
-- The left mark is XOR-drawn once and remains visible.
-- setXorMode colorName
setXorMode WHITE
-- color [colorName [red green blue [alpha]]]
color BLACK
moveTo 475 135
fillRect 100 80

-- Equivalent geometry in the right panel is XOR-drawn twice. The second
-- application restores exactly the pixels that were present before the first.
moveTo 685 135
fillRect 100 80
moveTo 685 135
fillRect 100 80
-- setPaintMode
setPaintMode
```

Listing 4.2: Excerpt from the JDORFX Paint, Composite, and XOR demonstration

The `render` command represents another case where the interface can be preserved more completely than the underlying rendering control. Java2D provides rendering hints for geometric and text antialiasing that do not have one-to-one equivalents in JavaFX Canvas. JDORFX therefore preserves the command's query/set state and canonical behavior even where a corresponding JavaFX rendering switch is unavailable [17]. The resulting limitation is described in Chapter 5.

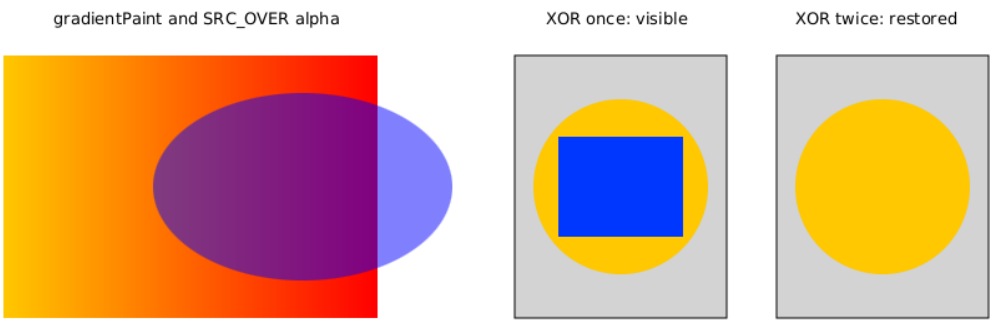


Figure 4.2: JDORFX Paint, alpha-compositing, and XOR demonstration: gradient and half-transparent `SRC_OVER` rendering (left), one XOR application that remains visible (center), and restoration after applying the same XOR geometry twice (right)

The clipping commands `clip` and `clipRemove` require separate state because the JavaFX `GraphicsContext` does not expose the same clip-query semantics as Java2D [17]. The handler maintains a private Java2D `Graphics2D` context as the authoritative model of the JDOR user clip. Changes performed through `clip`, `clipShape`, and `clipRemove` update this model, after which corresponding Canvas clip coverage is rebuilt.

The graphics-state commands `pushGC` and `popGC` use an explicit state object rather than relying exclusively on `GraphicsContext.save()` and `restore()`. In addition to native JavaFX state, this object contains JDOR compatibility state such as the logical Paint, Stroke, Composite, XOR settings, backgrounds, antialiasing settings, and clipping model.

The `copyArea` implementation first commits retained 2D content where necessary and obtains an immutable snapshot of the source pixels. The source region is then rendered at the displaced destination. Using a snapshot avoids feedback if source and destination overlap. The implementation also reproduces the transformation restrictions observed in the Java2D implementation used by JDOR, rather than silently accepting more general transformations with different semantics.

The commands `draw3DRect` and `fill3DRect` are JDOR 2D drawing operations and are unrelated to JDORFX's JavaFX 3D Shapes. Their raised or lowered appearance is reproduced on the Canvas by drawing highlighted and shadowed edges around the rectangular region.

Finally, `pathIterator` completes the inherited Shape and Path command set. A compatible Shape is converted to a Java2D geometry representation and a Java `PathIterator` is returned. An optional stored transformation and flatness value can be applied according to the command form.

Together, these implementations close the command-presence gap documented for the inherited JDORFX implementation. More difficult behavioral differences remain where the semantics

depend on Java2D raster state or drawing order. The mechanisms introduced to reduce these differences are described in the following section.

4.3 Compatibility Corrections and Architectural Improvements

As described in Section 3.3, the inherited JDORFX architecture combines Canvas-based drawing with retained Scene Graph nodes. This difference affects more than individual command cases. Operations such as image retrieval, compositing, clipping, copying, clearing, and graphics-state restoration require a consistent notion of the complete current 2D result.

The extended implementation therefore introduces additional mechanisms around the inherited architecture. Canvas content and retained 2D Shape renderings are grouped into one logical 2D presentation, retained content can be committed to the Canvas when raster semantics are required, and explicit compatibility state is maintained for Java2D concepts that do not map directly to JavaFX.

Window and threading behavior were revised for similar reasons. JavaFX Stage operations and Scene Graph changes are subject to JavaFX lifecycle and application-thread requirements [27]. The revised architecture also allows multiple JDORFX handler instances to share one JavaFX application runtime.

4.3.1 Rendering and Geometry Compatibility

The revised `JavaFXDrawingFrame` groups the Canvas and retained 2D Shape renderings below a common `current2DGroup`. The 3D Shape group and light group remain separate.

The revised hierarchy is shown on the right of Figure 3.1; the inherited arrangement is shown beside it for comparison.

This hierarchy creates a well-defined boundary for operations that require the complete current JDOR-compatible 2D result. Saving, retrieving, printing, copying, or pushing the current image can snapshot `current2DGroup` and therefore include both Canvas pixels and retained 2D Shape renderings.

The retained Shape handling was also made more uniform. Both `drawShape` and `fillShape` create a rendering of the named Shape and place it in the 2D Shape group. The logical source Shape remains registered separately.

This distinction becomes important for animation. A named source Shape can remain in the registry while the visible Scene contains a particular rendered node. The handler therefore records the most recent retained rendering associated with a registered Shape so that animation commands can address the node that is actually displayed.

Retained rendering cannot be used for all JDOR operations. Java2D applies commands sequentially to one raster destination, while separate Scene Graph nodes continue to exist independently. Composite modes, XOR rendering, destructive image modifications, arbitrary Java `Stroke` implementations, and area-copying operations may therefore require the existing retained history to become raster content.

For such operations, the handler can commit retained 2D rendering to the Canvas. The current combined 2D presentation is snapshotted, the Canvas is replaced with that visual result, the corresponding retained nodes are removed, and animation references to nodes that no longer exist are invalidated. Subsequent drawing then operates on one raster destination.

Figure 4.3 shows the order of the retained-to-raster operation.

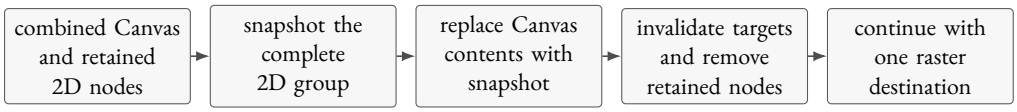


Figure 4.3: Committing retained 2D rendering to the Canvas

Listing 4.3 shows the core of this transaction. The JavaFX-thread dispatch and exception propagation that follow the shown range are omitted. The ordering is significant: the combined presentation is captured before the retained nodes are detached, and animations targeting those nodes are invalidated before the retained-rendering registries are cleared.

```
10418     private void commitCurrent2DRenderingToCanvas()
10419     {
10420         Runnable action=() -> {
10421             if (fxframe==null || fxframe.shapeGroup.getChildren().isEmpty()) return;
10422 
10423             List<Node> removedRoots=new ArrayList<>(fxframe.shapeGroup.getChildren());
10424             WritableImage flattened=snapshotCurrent2DRenderingOnFxThread();
10425             replaceCanvasPixelsIgnoringUserClip(flattened,
10426                 (int)canvas.getWidth(),(int)canvas.getHeight());
10427             invalidateAnimationsForRemoved2DNodes(removedRoots);
10428             fxframe.shapeGroup.getChildren().clear();
10429             latestRenderedShapes.clear();
10430         };
```

Listing 4.3: Excerpt from `JavaFXDrawingHandler`: committing retained 2D rendering to the Canvas

Flattening is performed only where required. Shapes remain retained where possible so that JavaFX transformations and animations remain available, and are converted to raster content only when a JDOR operation requires raster semantics.

Compositing uses the same general strategy. The handler maintains the Java2D `AlphaComposite` as authoritative logical state. Native JavaFX blend state is used for directly representable cases. Other Porter–Duff rules are evaluated through a software path that combines source, destination, alpha, and coverage information at pixel level.

XOR mode similarly uses a software compatibility path because JavaFX does not provide an equivalent of `Graphics2D.setXORMode()` [17].

Geometry conversions are also required in several areas. Named Shapes are primarily represented through JavaFX geometry because retained nodes are needed for rendering and animation. Commands whose semantics depend on Java2D geometry convert these objects to compatible implementations of `java.awt.Shape` or `Path2D`. This is used for Area operations, Path iteration, clipping, and arbitrary Stroke processing.

A Java `BasicStroke` can largely be represented using Canvas line properties such as width, cap, join, miter limit, and dash settings. An arbitrary implementation of the Java `Stroke` interface cannot. For such Strokes, the handler obtains the stroked geometry through `createStrokedShape()` and renders the resulting outline.

Clipping required its own compatibility model. A private Java2D `Graphics2D` object represents the authoritative JDOR user clip. The model is kept synchronized with the active affine transformation and is translated to Canvas clipping coverage after modifications. This enables the handler to preserve queryable clip bounds and to integrate clipping with `pushGC` and `popGC`.

Listing 4.4 and Figure 4.4 show these raster-oriented operations together. The sample first establishes the narrow rectangle (80, 20, 100, 210) and intersects it with the registered ellipse before painting the orange region. The resulting orange vertical band has straight sides from the rectangular clip and curved ends from the ellipse, making the intersection visible. After `clipRemove`, `copyArea` copies the 220 × 210 source region at (30, 20) by an offset of (270, 0), and `clearRect` removes a 40 × 40 area at (360, 100) from the copy. The result illustrates the raster-style sequence while JDORFX coordinates Canvas pixels with retained-node state; no exact Java2D pixel identity is assumed. Listing 4.4 shows the relevant part of `samples/16_jdorfx_clipping_copyarea.rxj`; image setup, borders, labels, handler setup, and window lifetime are omitted.

```
-- clip [x y width height]
clip 80 20 100 210
-- clipShape shapeName
clipShape OVAL
-- color [colorName [red green blue [alpha]]]
color ORANGE
-- moveTo [x [y]]
moveTo 10 10
```

```
-- fillRect width height
fillRect 250 230              -- only the oval intersection is painted
-- clipRemove
clipRemove
color BLACK
moveTo 80 20
drawRect 100 210              -- outline the rectangular clipping boundary
moveTo 30 20
-- drawRect width height
drawRect 220 210              -- outline the region copied below
moveTo 30 20
-- copyArea width height deltaX deltaY
copyArea 220 210 270 0      -- copy the finished pixels to the right
moveTo 360 100
-- clearRect width height
clearRect 40 40             -- replace one part of the copy with background
```

Listing 4.4: Excerpt from the JDORFX clipping and copy-area demonstration

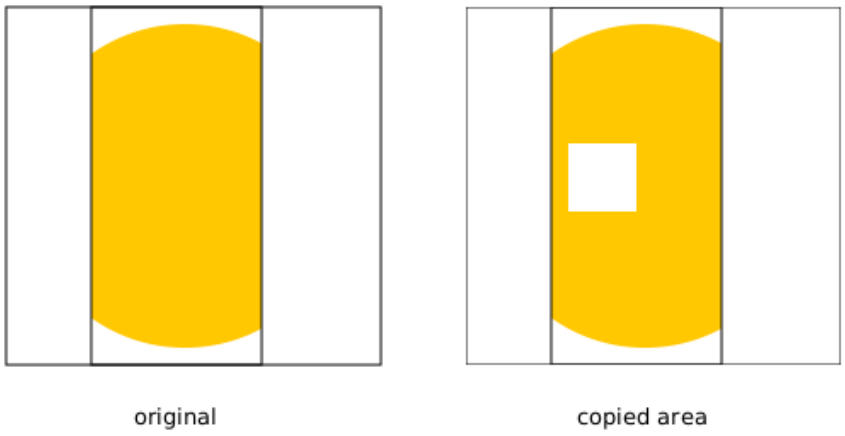


Figure 4.4: JDORFX clipping and raster-copy demonstration: intersected rectangular and Shape clipping in the labelled original region (left), and the displaced `copyArea` result with a white region subsequently produced by `clearRect` (right)

These mechanisms substantially reduce the difference between Java2D and JavaFX behavior, but they cannot make the two rendering models identical in all cases. Remaining differences, particularly those involving retained-node ordering, are discussed in Chapter 5.

4.3.2 Window Lifecycle and Update Behavior

Window handling also required changes because the drawing surface and the top-level window have different lifecycles in JDOR and JavaFX. In JDOR, the underlying image can remain valid independently of the `JFrame` used to display it. In JavaFX, visible content is held by a `Scene`, which is displayed through a `Stage` [30, 32].

The revised implementation therefore treats the Canvas, Scene, and Stage as related but distinct objects.

One correction concerns `winSize`. The inherited `resizeSurface` implementation resized both the Scene content and the Canvas [37]. This coupled the window dimensions with the dimensions of the drawing image. In the revised implementation, `winSize` changes the preferred content size without resizing the Canvas. Image dimensions remain controlled by `newImage`.

The `winClose` behavior was similarly separated from the lifetime of the drawing state. Closing the window removes the top-level Stage while preserving the Scene, Canvas, registered objects, and drawing content. A later window operation can create a new Stage and attach the preserved Scene.

Changing the decoration of an already visible Stage requires special handling because JavaFX Stage style is normally established before the Stage is shown [32]. Where necessary, the revised implementation rebuilds the Stage with the requested decoration and transfers the existing Scene and relevant window properties.

The behavior of `winUpdate` also needed adaptation. Retained Scene Graph nodes remain live even if ordinary Stage-update logic is suspended. Simply setting an update flag is therefore insufficient to freeze the display.

When window updates are disabled, the implementation creates an independent copy of the currently visible 2D presentation and places that copy in the displayed Scene. The authoritative `current2DGroup` continues to be modified by later commands. When updates are re-enabled, the frozen presentation is replaced by the current live group.

This separates the displayed state from the working drawing state and allows command processing to continue while the visible 2D presentation remains unchanged.

4.3.3 JavaFX Thread Handling and Multiple Handlers

JavaFX requires Stage and Scene Graph operations to be performed on the JavaFX Application Thread, while JDORFX commands normally enter the handler through the thread executing the ooRexx program [27].

The inherited implementation already separated Rexx command processing from Stage updates. It started JavaFX separately and used a `ConcurrentLinkedDeque` together with an updater scheduled through `Platform.runLater()` [37].

The revised implementation removes this shared Scene-deque path. JavaFX startup remains shared across all handler instances: the static startup state, `fxReady`, and `fxStartupFailure` ensure that the JavaFX runtime is launched once and that command processing observes the result of that launch. The primary Stage supplied to `Application.start()` is kept hidden and serves only to bootstrap the runtime; it is not used as a handler window.

Active handler instances are recorded in the thread-safe `activeHandlers` list. Each handler owns its Canvas, Scene, Stage, registries, graphics state, and window state. The periodic updater runs on the JavaFX Application Thread, iterates over this list, and invokes `updateStageOnFxThread()` for each handler. Actual windows are therefore separate Stages created and managed by their respective handlers.

The helper `runOnFxThreadAndWait` executes an operation immediately when already running on the JavaFX Application Thread. Otherwise, it submits the operation using `Platform.runLater()` through a `FutureTask` and waits for completion.

Figure 4.5 summarizes the shared runtime, the per-handler state, and the synchronous transfer of JavaFX operations.

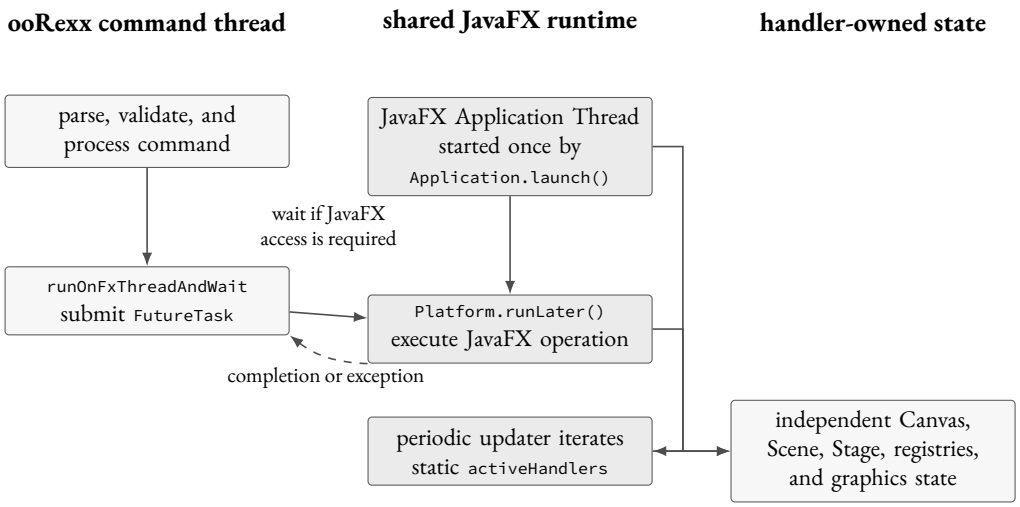


Figure 4.5: JDORFX command threads, shared JavaFX runtime, and handler-owned windows

This is important for operations such as Scene construction, retained Shape insertion, snapshots, Stage lifecycle changes, and animation control. The command-processing thread does not report successful completion before the corresponding JavaFX operation has completed, and exceptions from that operation can propagate back to the command path.

Listing 4.5 contains the complete helper method as an excerpt from the handler; adjacent diagnostics and other threading helpers are omitted. The early branch prevents a self-deadlock when the caller is already on the JavaFX Application Thread, while `FutureTask.get()` gives other callers synchronous completion and exception propagation.

```
872     private void runOnFxThreadAndWait(Runnable action)
873     {
874         if (Platform.isFxApplicationThread())
875         {
876             action.run();
877             return;
878         }
879 
880         java.util.concurrent.FutureTask<Void> task=
881             new java.util.concurrent.FutureTask<>(() -> {
882                 action.run();
883                 return null;
884             });
885         Platform.runLater(task);
886 
887         try
888         {
889             task.get();
890         }
891         catch (java.util.concurrent.ExecutionException e)
892         {
893             Throwable cause=e.getCause();
894             if (cause instanceof RuntimeException) throw (RuntimeException)cause;
895             if (cause instanceof Error) throw (Error)cause;
896             throw new IllegalStateException("JavaFX update failed",cause);
897         }
898         catch (InterruptedException e)
899         {
900             Thread.currentThread().interrupt();
901             throw new IllegalStateException("Interrupted while waiting for JavaFX update", e);
902         }
903     }
```

Listing 4.5: Excerpt from `JavaFXDrawingHandler`: synchronous JavaFX-thread dispatch

This separation is necessary because JavaFX's `Application.launch()` initializes one JavaFX application runtime and cannot be invoked independently for every handler [11]. Multiple handlers can consequently maintain independent windows within one shared JavaFX runtime, and closing an individual handler Stage does not terminate that runtime.

The revised architecture does not move all command processing to the JavaFX Application Thread. Parsing, validation, Rexx context interaction, and other non-graphical work continue on the calling thread. Only operations requiring JavaFX thread access are transferred when necessary.

4.4 New 3D Functionality

The inherited JDORFX implementation already provided primitive 3D Shapes, parallel and perspective cameras, ambient and point lights, transformations of 3D Shapes, Phong materials, and texture maps [42]. The work of this thesis extends this interface with additional controls for existing 3D scene elements.

Six new command entries were added: camera translation, camera rotation, near- and far-clipping-plane control, PointLight translation, and face culling.

The complete syntax for the 3D interface is summarized in Sections A.7–A.12 of Appendix A.

4.4.1 Camera Transformation Commands

The inherited implementation allowed Cameras to be created at a specified position and assigned to the Scene using `setCamera`, but did not provide commands for transforming an existing Camera after creation [37].

The commands `translateCamera` and `rotateCamera` provide this functionality.

`translateCamera` accepts a registered Camera nickname and translation values for the x-, y-, and z-axes. The handler creates a JavaFX `Affine`, appends the requested translation, and adds the transform to the Camera's transformation list [7, 15].

`rotateCamera` similarly accepts an angle, a three-dimensional pivot, and a three-dimensional rotation axis. The resulting Affine rotation is added to the Camera.

The syntax intentionally mirrors the transformation commands already available for 3D Shapes. Successive transformations are appended and can therefore accumulate.

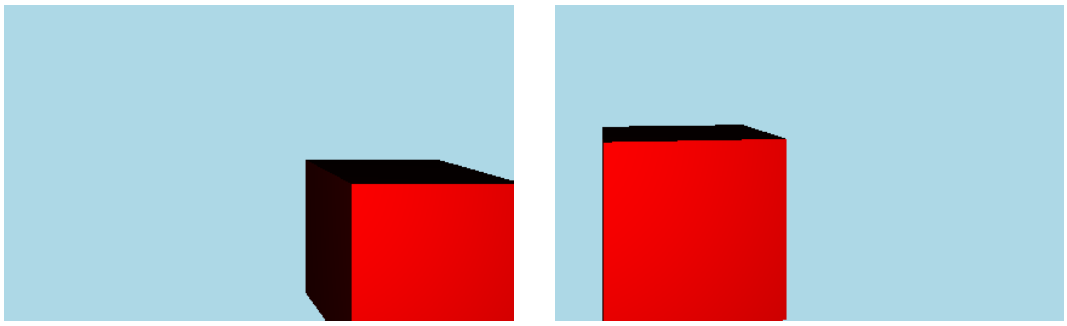
Both commands validate that the supplied nickname refers to a registered Camera before modifying the Scene Graph.

Listing 4.6 shows the relevant part of the camera demonstration reproduced in Listing B.3. Comments, handler setup, value queries, the viewing wait, and shutdown are omitted. In Figure 4.6, the box, material, light, clipping interval, and camera type remain unchanged. The second panel is captured after `translateCamera VIEW 200 50 0` and `rotateCamera VIEW 10 300 200 0 0 1 0`. The box changes position and viewing angle in the image because the camera is transformed; its geometry stays fixed.

```
1 -- shape3d shapeName box x y z width height depth
```

```
shape3d BOX box 300 200 700 180 180 180
-- material materialName [color]
material RED_MAT RED
-- setMaterial shapeName materialName
setMaterial BOX RED_MAT
-- fill3DShape shapeName
fill3DShape BOX
-- Keep the scene lighting fixed so only the camera changes between views.
light LAMP point 150 100 200 WHITE
setLight LAMP

-- camera cameraName perspective x y z [fieldOfView [fixedEyeAtCameraZero]]
camera VIEW perspective 0 0 0 45 .true
-- cameraNearClip cameraName [distance]
cameraNearClip VIEW 1
-- cameraFarClip cameraName [distance]
cameraFarClip VIEW 2000
-- setCamera [cameraName]
setCamera VIEW
-- The camera, rather than the box, moves and turns.
-- translateCamera cameraName deltaX deltaY deltaZ
translateCamera VIEW 200 50 0
-- rotateCamera cameraName angle pivotX pivotY pivotZ axisX axisY axisZ
rotateCamera VIEW 10 300 200 0 0 1 0
```

Listing 4.6: Excerpt from the JDORFX extended-camera demonstration



(a) Initial camera view (b) After camera translation and rotation

Figure 4.6: The fixed box before and after translating and rotating the camera

4.4.2 Camera Clipping Plane Commands

JavaFX Cameras expose near and far clipping distances. Geometry outside the visible depth interval is excluded from rendering [15, 23].

JDORFX exposes these properties through `cameraNearClip cameraName [distance]` and `cameraFarClip cameraName [distance]`.

Both commands use query/set semantics. When only the nickname is supplied, the current value is returned. When a new distance is supplied, the previous value is returned and the new value is installed.

The implementation validates the value before changing the Camera. The distance must be finite and greater than zero. In addition, the near clipping distance must remain smaller than the far clipping distance and the far distance must remain greater than the near distance.

These commands work with the Camera abstraction and therefore apply to registered parallel and perspective Cameras.

4.4.3 Light Transformation Commands

The inherited implementation provided both `AmbientLight` and `PointLight`, and a PointLight could be given an initial position [42]. It could not, however, be moved afterwards through a JDORFX command.

The new command `translatePointLight lightName tx ty tz`

resolves the selected light and explicitly requires it to be a `PointLight`. An Affine translation is then appended to the light's transformation list.

The distinction between PointLight and AmbientLight is intentional because the position of a PointLight has a direct effect on scene illumination, whereas AmbientLight is not used as a positional light source in the same manner [28, 9].

4.4.4 Cull Face Support

JavaFX `Shape3D` objects expose the `cullFace` property, whose values are `NONE`, `BACK`, and `FRONT` [31].

The new command `cullFace shapeName [NONE|BACK|FRONT]`

provides query and set access to this property.

Without a second argument, the current culling mode is returned. When a new mode is supplied, the previous value is returned and the selected mode is installed. Unsupported values are rejected explicitly.

Face culling should be distinguished from depth buffering. Culling determines whether faces are rendered based on their orientation, while depth buffering resolves which surfaces are visible

based on their depth [31, 30]. The extended `newImage` command can create a Scene with depth buffering enabled, while `cullFace` independently configures an individual 3D Shape.

The demonstration treats a JavaFX `Box` as a hollow surface shell. A white `PointLight` is placed inside it, and a small blue marker sphere is positioned nearby. The marker and its highlight make the interior easier to identify when culling exposes it. A dim ambient light keeps the exterior legible without changing the placement of the internal PointLight.

Listing 4.7 shows the relevant commands from the complete demonstration in Listing B.2. Handler registration, window lifetime, waits, and shutdown are omitted. The box, marker, materials, and lights remain unchanged; only the culling mode and the outside or inside camera change between the five states in Figure 4.7.

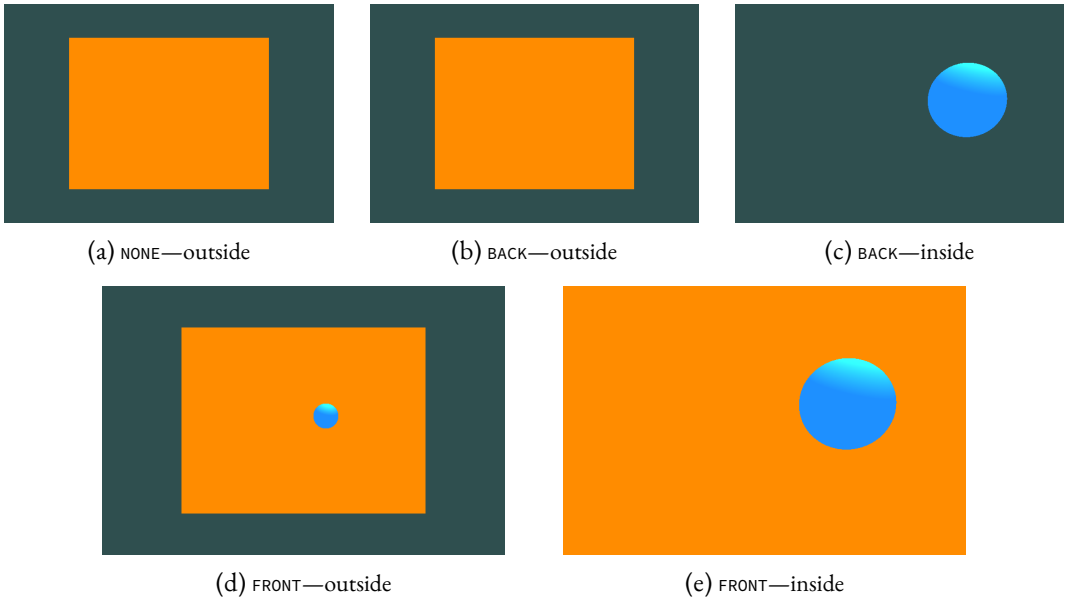
Panel (a) provides the `NONE` outside baseline. From the same outside viewpoint, panel (b) with `BACK` remains visually similar because the outward-facing surface is retained. Moving the camera inside the shell in panel (c) changes the orientation of the visible surfaces relative to the camera, causing `BACK` culling to remove them. Conversely, `FRONT` removes the outward-facing near wall in panel (d), exposing the illuminated interior and marker from outside. Panel (e) shows the same mode from inside the box. Together, the five states show how the result of face culling depends on both face orientation and camera viewpoint.

```
-- shape3d shapeName box x y z width height depth
shape3d ROOM box 360 240 700 420 320 360
-- material materialName [color]
material ROOM_MAT DARKORANGE
-- setMaterial shapeName materialName
setMaterial ROOM ROOM_MAT
-- fill3DShape shapeName
fill3DShape ROOM

-- A small blue sphere identifies the interior and the nearby PointLight.
shape3d MARKER sphere 410 230 675 28
material MARKER_MAT DODGERBLUE
setMaterial MARKER MARKER_MAT
fill3DShape MARKER

-- The PointLight is inside the box; a dim ambient light reveals the exterior.
-- light lightName point x y z [color]
light ROOM_LIGHT point 420 170 650 WHITE
setLight ROOM_LIGHT
light FILL_LIGHT ambient DIMGRAY
setLight FILL_LIGHT
```

```
-- camera cameraName perspective x y z [fieldOfView [fixedEyeAtCameraZero]]
camera OUTSIDE perspective 360 240 0 48 .true
cameraNearClip OUTSIDE 1
cameraFarClip OUTSIDE 2000
camera INSIDE perspective 360 240 540 64 .true
cameraNearClip INSIDE 1
cameraFarClip INSIDE 2000
-- setCamera [cameraName]
setCamera OUTSIDE
-- cullFace shapeName [NONE|BACK|FRONT]
cullFace ROOM NONE
winTitle "CullFace NONE - outside baseline"
cullFace ROOM BACK
setCamera OUTSIDE
winTitle "CullFace BACK - outside"
setCamera INSIDE
winTitle "CullFace BACK - inside"
cullFace ROOM FRONT
setCamera OUTSIDE
winTitle "CullFace FRONT - outside"
setCamera INSIDE
winTitle "CullFace FRONT - inside"
```

Listing 4.7: Excerpt from the JDORFX CullFace demonstration

These additions extend the inherited 3D interface from object creation toward continued manipulation of a running 3D scene.



(a) NONE—outside (b) BACK—outside (c) BACK—inside

(d) FRONT—outside (e) FRONT—inside

Figure 4.7: JDORFX CullFace results for one internally lit box shell under five combinations of culling mode and camera viewpoint

4.5 Animation Support

Animation support is a major JavaFX-specific extension of JDORFX. The inherited handler did not expose the JavaFX animation framework directly [37]. Although repeated commands and `sleep` could produce simple command-driven sequences, they required the ooRexx program to perform each update itself.

JavaFX instead animates writable properties of retained Scene Graph objects over time [10, 33]. The extended handler exposes this model through 18 animation command entries covering predefined transitions, animation composition, interpolators, timelines, key frames, key values, timed pauses, status queries, and playback control.

The complete animation command interface is summarized in Section A.13 of Appendix A.

Three registries support the animation interface: `hmAnimations`, `hmTimelines`, and `hmKeyValues`. Timelines are additionally entered in the general Animation registry so that the same control commands can be used for transitions and timelines.

4.5.1 JavaFX Animation Model

As described in Section 2.7.6, JavaFX provides predefined Transition classes for common animation types and Timeline-based animation for more general property changes [10, 33].

JDORFX exposes both approaches. Transition commands provide concise access to common effects, while `timeline`, `keyValue`, and `keyframe` allow individual writable properties to be animated.

Animation durations are expressed in milliseconds. Where supported, animations can additionally use a cycle count and automatic reversal. A cycle count of `-1` is translated to JavaFX's indefinite cycle mode.

The supported interpolators include `DISCRETE`, `LINEAR`, `EASE_IN`, `EASE_OUT`, and `EASE_BOTH`. Cubic spline interpolation can additionally be specified through four control values.

The target of a node-based animation is resolved from the JDORFX object registries. Supported targets include retained 2D Shape renderings, 3D Shapes, Cameras, and Lights.

For a 2D Shape, the logical object stored by `shape` is not itself necessarily the visible node. The handler therefore resolves the most recent retained rendering produced by `drawShape` or `fillShape`. If that rendering has subsequently been flattened into the Canvas, it is no longer available as an animation target.

4.5.2 Animation Creation

JDORFX provides predefined transition commands for fading, rotation, scaling, translation, path movement, fill-color changes, and stroke-color changes.

`animationFade` creates a JavaFX `FadeTransition`. `animationRotate` creates a `RotateTransition`. `animationScale` and `animationTranslate` expose scale and translation changes, including z-axis values where the target supports them.

`animationPath` creates a `PathTransition`. A named Shape provides the path geometry while another resolved node is moved along it. The orientation can remain unchanged or be set to follow the tangent of the path.

`animationFill` and `animationStroke` operate on retained 2D Shapes. Because these transitions modify JavaFX Shape properties, the named Shape must have a retained visible rendering. Creating the source Shape alone is insufficient.

Animations can also be combined. `animationSequential` creates a `SequentialTransition`, while `animationParallel` creates a `ParallelTransition`. Child animations are resolved by nickname from the general animation registry.

Both composition commands can optionally receive a common Node target. If the first argument after the new animation nickname resolves to a JDORFX animation target, it is used as the parent transition's node; otherwise that token is interpreted as the first child-animation nickname.

The resulting composed animation is itself registered and can therefore be controlled normally or nested inside another composed animation.

The command `animationPauseTransition` creates a JavaFX `PauseTransition` with a specified duration. It must not be confused with `animationPause`, which pauses an already running animation. A PauseTransition is itself an Animation and is particularly useful inside a SequentialTransition to introduce a timed delay between other animations.

4.5.3 Animation Control

The general Animation registry allows transitions, composed animations, PauseTransitions, and Timelines to use the same playback interface.

Table 4.2 summarizes the four commands.

Table 4.2: JDORFX animation-control commands

Syntax	Operation
`animationPlay animationName`	Start or resume an animation.
`animationPause animationName`	Pause an animation at its current time.
`animationStop animationName`	Stop an animation and reset its play head.
`animationStatus animationName`	Query the current status without changing it.

The first three commands obtain the animation's current JavaFX status before performing the requested state change and return that previous state. `animationStatus` returns the current status without modifying the animation.

The JavaFX status values are `RUNNING`, `PAUSED`, and `STOPPED` [10].

Playback operations are executed through the synchronous JavaFX-thread mechanism described in Section 4.3.3. The command therefore does not return before the corresponding JavaFX state change has been executed.

The `setInterpolator` command changes the interpolation strategy of supported Transition objects. It is not applied to Timelines because Timeline interpolation is defined individually by each `KeyValue`. Sequential and parallel transitions likewise do not use one interpolator in the same manner as an ordinary Transition.

4.5.4 Timeline-Based Animation and Camera Animation

Timelines provide a more general animation mechanism for properties that are not conveniently represented by one predefined transition.

The `timeline` command creates or retrieves a named `Timeline`. Optional arguments configure the cycle count and automatic reversal. The Timeline is stored both in the Timeline registry and in the general Animation registry.

The `keyValue` command connects a writable property of a named target with a target value. Supported properties include opacity, rotation, translation, scaling, and selected Shape-specific properties such as fill and stroke color.

An optional interpolator can be assigned when the KeyValue is created. This allows different properties in the same Timeline to use different interpolation behavior.

The `keyframe` command associates one or more KeyValues with a time in a named Timeline. A single registered KeyValue can be referenced by nickname. Alternatively, a Rexx variable containing a JavaFX `KeyValue[]` can be supplied so that multiple properties receive target values at the same point in time.

This generic node-property model also enables Camera animation. Cameras are included in the animation target resolution because JavaFX Camera derives from `Node` [15, 21]. Translation, rotation, and other applicable Node properties can therefore be animated using ordinary transitions or Timeline KeyValues.

No separate Camera-animation class is required. The immediate Camera transformation commands in Section 4.4.1 alter a Camera directly, while the animation commands change applicable properties gradually over time.

Lights are resolved through the same node-target mechanism and can therefore participate in suitable generic transitions and Timeline animations as well.

The result is a two-level animation interface: common effects can be created with predefined transitions, while more complex animation behavior can be constructed from KeyValues, KeyFrames, Timelines, and composed Animations.

4.5.5 JavaFX Animation Basics and JDORFX

Oracle's JavaFX 8 Animation Basics examples provide small reference cases in which the animated property, endpoint, duration, repetition, and composition are all explicit [35]. Six examples were recreated as paired Java and JDORFX programs: fade, path, parallel, and sequential transitions, the basic Timeline, and the built-in interpolator example. The JDORFX versions retain the published geometry, durations, and animation parameters where the command interface provides corresponding functionality.

The basic Timeline is a particularly direct mapping. Oracle's Java example constructs a Rectangle at $x = 100$, creates a KeyValue whose target is the Rectangle's `xProperty()` at $x = 300$, places that value in a 500-ms KeyFrame, and plays an indefinitely repeating, auto-reversing Timeline. The JDORFX version in Listing 4.8 targets the geometric `x` property as well; it does not replace the operation with a translation. Table 4.3 summarizes the correspondence between the JavaFX objects and the JDORFX commands.

Table 4.3: Mapping Oracle's basic Timeline example to JDORFX

JavaFX structure	JDORFX command abstraction
Rectangle construction and insertion in the Scene Graph	`shape` followed by `fillShape`
`Timeline`, indefinite cycles, and auto-reverse	`timeline basicTimeline "-1" .true`
`KeyValue(rect.xProperty(), 300)`	`keyValue timelineKV1 rectBasicTimeline x 300`
500-ms `KeyFrame` containing the KeyValue	`keyframe basicTimeline 500 timelineKV1`
`timeline.play()`	`animationPlay basicTimeline`

```
-- Create and fill the rectangle animated by the timeline.
-- shape parameters: name, type, x, y, width, height
color red
shape rectBasicTimeline rect 100 50 100 50
fillshape rectBasicTimeline

-- Create a timeline that repeats indefinitely and reverses after each cycle.
```

```
-- Parameters: timelineName, cycleCount (-1 means indefinite), autoReverse
timeline basicTimeline "-1" .true
-- Add a keyframe reached after 500 ms.
-- First create a named KeyValue, then add it to the timeline.
keyValue timelineKV1 rectBasicTimeline x 300
keyframe basicTimeline 500 timelineKV1
-- Timelines use the same animationPlay/animationStop controls as transitions.
animationPlay basicTimeline
```

Listing 4.8: Oracle's basic Timeline animation expressed through JDORFX

Listing 4.8 shows the relevant part of `samples/comparisons/oracle/Example_05_BasicTimelineAnimation.rxj`; handler setup, window configuration, the viewing wait, and shutdown are omitted. Figure 4.8 shows three representative states of the JDORFX Timeline. They correspond to selected instants of a time-dependent, auto-reversing animation: the rectangle begins at $x = 100$, passes through the intermediate position, and reaches the declared $x = 300$ KeyValue at 500 ms.



(a) 0 ms: initial (b) 250 ms: intermediate (c) 500 ms: KeyValue reached

Figure 4.8: Representative frames exported from the JDORFX counterpart of Oracle's basic Timeline example

The mapping is not equally literal for every predefined Transition. Oracle's parallel and sequential examples set an initial `translateX` and `translateY`, then use explicit `fromX`/`toX` endpoints. JDORFX's translation command exposes relative distances instead. The comparison programs therefore incorporate the initial translation into the Rectangle geometry and use the equivalent displacement. Visible start and end positions, durations, cycle counts, auto-reverse behavior, and child composition are preserved, but geometry and translation are not represented as separate initial properties. By contrast, path orientation maps directly to `ORTHOGONAL_TO_TANGENT`, and the built-in Timeline example maps `EASE_BOTH` directly to the KeyValue interpolator.

Two Oracle examples are not reproduced because they depend on functionality that JDORFX does not expose. Oracle's Timeline Events example attaches callbacks and uses an `AnimationTimer`, while its custom-interpolator example defines a Java subclass with a non-monotonic interpolation function [35]. JDORFX exposes KeyValues and the built-in interpolators, including cubic splines, but does not expose arbitrary event handlers, AnimationTimer callbacks, or user-defined Java Interpolator subclasses.

4.5.6 Comparing JDOR and JDORFX Animation

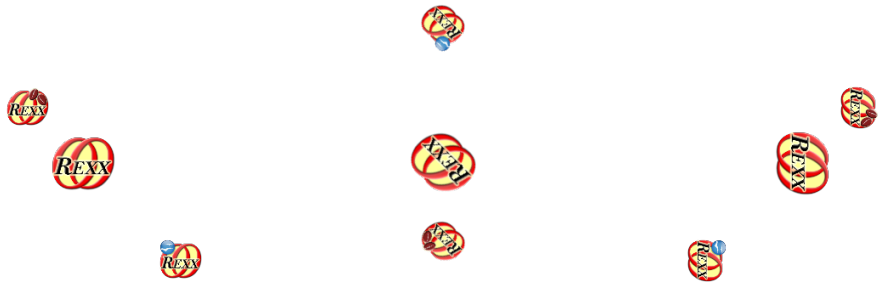
The supervisor-provided JDOR programs `2-130_JDOR_animate_two_frames.rxj` and `3-110_JDOR_animate_composite.rxj` were retained as the reference versions. Each has a manually driven JDORFX port with the same drawing-loop structure and a separate JDORFX version built from JavaFX animation objects. This produces three conceptually comparable implementations without replacing the useful one-to-one ports.

In the first example, JDOR and the manual JDORFX port repeatedly install a rotation, draw the image, and sleep for 1/30 second. Ninety-one iterations advance from 0 to 450 degrees in five-degree steps. The Transition version instead declares one 450-degree `RotateTransition` lasting 3033 ms, uses linear interpolation, and starts it once. Listing 4.9 shows the application-level declaration; JavaFX calculates the intermediate angles after `animationPlay` returns.

```
newImage 400 400
winUpdate .false
shape PICTURE Rectangle 0 0 400 400
fillShape PICTURE
getState
call bitmapFill rc~renderedShapes~get("PICTURE"),sourceBitmap
winUpdate .true
winFrame .false
winShow
winLocation (screenWidth-400) 0

-- The manual loop performs 91 five-degree steps and sleeps 1/30 s per step:
-- 450 degrees in approximately 3033 ms.
animationRotate TURN PICTURE 3033 450 1 .false
setInterpolator TURN LINEAR
say "-- Left: source bitmap. Right: a single JavaFX rotation animation."
animationPlay TURN
```

Listing 4.9: Replacing the 2-130 redraw loop with a named RotateTransition

This listing is an excerpt from the complete Transition program in Listing B.4; bitmap construction, window placement, the viewing wait, and shutdown are omitted. Figure 4.9 exports the initial, halfway, and final states of its retained node. The original JDOR program and the manual JDORFX port serve the same purpose by drawing these three logos and advancing their rotation in five-degree application-loop steps. The Transition version preserves that visible purpose while delegating the intermediate states to JavaFX. Because 450 degrees is one full turn plus 90 degrees, the final panel is perpendicular to the initial orientation rather than identical to it.



(a) 0 ms: 0° (b) 1516 ms: approximately 225° (c) 3033 ms: 450°

Figure 4.9: Representative output states of the JDORFX `RotateTransition` used for the 2-130 JDOR/JDORFX comparison

The second example fades in three logos, moves them out in different directions, moves a message into the window, and flashes a changing countdown. JDOR and the manual JDORFX port implement these effects through four explicit application loops that repeatedly alter Composite state, redraw pixels, update the window, and sleep. The Transition version first creates retained bitmap nodes, prepares the countdown states, and then composes FadeTransitions, TranslateTransitions, and a discrete Timeline into one named SequentialTransition. Its exact interpolated trajectories are not the same as the original pixel-step loops, but the visible stages and their order are preserved.

Tables 4.4 and 4.5 compare the source structure and command traffic of the manual and JavaFX-animation versions. The "All lines" column includes every physical line in the sample, including comments and blank lines. The non-comment line count excludes blank lines, the shebang, and Rexx line and block comments; it is not a count of commands. A handler-command line is a source line containing a JDOR or JDORFX graphics or animation command. Thus, a line is counted once regardless of how often a loop executes it, while the same command written in alternate redirection branches counts as two source lines. The sleep-command column counts all source lines containing a sleep command, including sleeps outside animation loops. Framework and handler source is not counted.

For the dynamic count, each program was run with its optional redirection argument, and non-empty, non-comment canonical output lines were counted. This counts commands actually delivered during the complete timed run, including setup and shutdown. Both JDOR and JDORFX could therefore be measured. The final column gives the signed change in redirected stream lines relative to the corresponding JDOR program; a positive value means more traffic and a negative value means less. The JDOR row itself is the baseline. The Transition variants use a small ooRexx/Java bridge during setup to paint a bitmap into a retained Rectangle; that bridge

Table 4.4: Static source measurements for the JDOR animation comparisons

Program	All lines	Non-comment lines	Command lines	Sleep lines
2-130 JDOR	185	96	36	4
2-130 JDORFX manual	197	104	39	4
2-130 JDORFX Transition	64	53	31	1
3-110 JDOR	209	126	45	6
3-110 JDORFX manual	207	125	44	5
3-110 JDORFX Transition	124	101	57	1

call is not a handler command and consequently does not appear in the canonical stream. The stream sizes are thus comparable as handler-command traffic, but the Transition recordings are not self-contained replays of the bitmap setup.

Table 4.5: Redirected canonical handler traffic

Program	Stream lines	Change vs. JDOR
2-130 JDOR	298	baseline
2-130 JDORFX manual	300	+2 (+0.7%)
2-130 JDORFX Transition	31	−267 (−89.6%)
3-110 JDOR	10,915	baseline
3-110 JDORFX manual	10,874	−41 (−0.4%)
3-110 JDORFX Transition	169	−10,746 (−98.5%)

The 3-110 Transition program has more command lines in its source: 57, compared with 45 in JDOR and 44 in the manual JDORFX port. It uses these extra lines to set up nodes and countdown KeyFrames. During execution, though, JavaFX updates the animation without a Rexx command for each frame. The Transition program therefore sends only 169 handler commands, compared with 10,915 for JDOR. This shows a reduction in commands sent during execution, even though the Transition program has more command lines in its source. No CPU, memory, or rendering-performance conclusion is drawn because no performance benchmark was performed.

4.5.7 Complex JavaFX Animation Case Study

Oracle's Tree Animation Example was chosen as a larger case study because it combines several animation mechanisms in one application [36]. It uses ordinary and custom Transitions, Timelines, KeyFrames and KeyValues, sequential and parallel composition, property bindings, event handlers, randomized Scene Graph generation, and repeating motion. This makes the example useful for examining both the capabilities and the limits of the JDORFX animation interface.

The program `samples/25_jdorfx_oracle_tree_animation.rxj` is a conceptual counterpart rather than a pixel-perfect port. It creates a smaller tree scene with nine leaves, three flowers, and 15 grass blades. Timelines reveal branch generations sequentially; leaf FadeTransitions run in parallel; repeating RotateTransitions provide wind-like branch and grass motion; FillTransitions alternate leaf colors; and a delayed parallel group translates, rotates, and fades a falling leaf. Listing 4.10 shows how named animations express the top-level structure.

```
-- Spring appearance is parallel; the complete growth is sequential.
do leafSpec over leafData
    parse var leafSpec leafName .
    animationFade ("APPEAR_"leafName) (leafName) 1100 0 1
end
animationParallel SPRING APPEAR_LEAF_1 APPEAR_LEAF_2 APPEAR_LEAF_3 ,
    APPEAR_LEAF_4 APPEAR_LEAF_5 APPEAR_LEAF_6 APPEAR_LEAF_7 ,
    APPEAR_LEAF_8 APPEAR_LEAF_9
animationSequential GROW_TREE GROW_TRUNK GROW_LOW GROW_HIGH SPRING
animationParallel WIND TREE_WIND GRASS_WIND

-- Fill transitions provide a repeatable green/autumn/green season cycle.
do leafSpec over leafData
    parse var leafSpec leafName .
    animationFill ("SEASON_"leafName) (leafName) 4500 ,
        FORESTGREEN DARKORANGE 2 .true
end
animationParallel SEASONS SEASON_LEAF_1 SEASON_LEAF_2 SEASON_LEAF_3 ,
    SEASON_LEAF_4 SEASON_LEAF_5 SEASON_LEAF_6 SEASON_LEAF_7 ,
    SEASON_LEAF_8 SEASON_LEAF_9 "-1" .false

-- A delayed leaf fall combines translation and fading in parallel.
animationPauseTransition FALL_DELAY 5200
animationTranslate LEAF_DESCENT FALLING_LEAF 3600 90 280 0
animationFade LEAF_DISAPPEAR FALLING_LEAF 3600 1 0
animationRotate LEAF_TURN FALLING_LEAF 1800 360 2 .false
animationParallel FALL_MOTION LEAF_DESCENT LEAF_DISAPPEAR LEAF_TURN
animationSequential FALL_SEQUENCE FALL_DELAY FALL_MOTION

-- Oracle likewise combines wind with a sequential growth/season program.
animationParallel ALL GROW_TREE WIND SEASONS FALL_SEQUENCE
```

Listing 4.10: Composition structure of the JDORFX Tree Animation case study

The listing is an excerpt from the complete runnable case study in Listing B.5. Scene construction, per-node declarations, playback lifetime, and shutdown are omitted. Figure 4.10 complements the composition code with three exported states. The first captures the initial growth stage; the second shows the revealed crown and appearing leaves; the third shows the later orange season, displaced wind state, and descending leaf. The directly reproduced behavior is the composition of sequential growth, parallel appearance, repeating color changes, and concurrent fall operations. Branch and grass wind and the straight falling-leaf path are approximations of Oracle's property bindings and custom trajectories; event-driven replacement of fallen elements remains outside the JDORFX abstraction. No pixel-perfect equivalence with Oracle is claimed.



(a) Initial growth state (b) Growth and appearance (c) Later season and leaf fall

Figure 4.10: Representative states exported from the JDORFX Oracle Tree Animation case study

The example demonstrates several benefits of the command abstraction. It has no JavaFX `Application` lifecycle boilerplate, does not explicitly marshal work to the JavaFX Application Thread, and uses nicknames rather than Java object references to connect Shapes, KeyValues, Timelines, and composed animations. The ooRexx program can be executed directly without a Java compilation cycle. Within the functionality exposed by JDORFX, this makes JavaFX animation concepts accessible through compact command declarations.

The case study also exposes the abstraction boundary. JDORFX cannot define a custom Transition subclass or arbitrary property binding. Oracle's grass uses a custom phase property and trigonometric bindings, whereas the counterpart rotates individual retained lines. Oracle grows branch endpoint properties and uses a hierarchy of branch-local transforms; JDORFX instead reveals flat named branch nodes generation by generation and rotates each around its own center. The command interface exposes opacity as an animation target, but it has no synchronous command for setting a retained Node's opacity or visibility before playback. Consequently, Shapes belonging to later sequential phases initially retain JavaFX's default opacity and can be visible before their FadeTransition begins; direct JavaFX avoids this by calling `Node.setOpacity(0.0)` before showing the Stage. This is a limitation of the present JDORFX abstraction, not of JavaFX animation. Oracle's falling elements follow sine-derived bound trajectories and use event callbacks to replace or hide nodes; JDORFX approximates this with a pre-created leaf, a straight Translate-

Transition, rotation, and fading. Arbitrary event handlers, bindings, specialized algorithms, and the example's media and performance-tracker code are outside the present command interface.

Direct JavaFX remains more flexible: Java code can use the complete API, define new properties and subclasses, install bindings and event handlers, and implement specialized algorithms. JDORFX intentionally exposes a smaller command-oriented subset. The case study therefore demonstrates both the supported animation composition and the functionality that remains outside the current command interface.

4.6 Input and Output Redirection

JDORFX implements the BSF4ooRexx `RexxRedirectingCommandHandler` interface and therefore participates in the input and output redirection mechanisms of ooRexx command environments [37]. ooRexx allows the `ADDRESS` instruction to redirect command input and output to supported Rexx objects and streams [3].

In JDORFX, redirected output provides a normalized textual representation of successfully processed commands. Redirected input allows command lines to be fed back into the same command-processing path. Together, these mechanisms allow command sequences to be recorded and replayed without introducing a separate JDORFX macro language.

4.6.1 Canonical Command Output

When output redirection is active, the handler writes a canonical representation of a recognized command rather than necessarily reproducing the original input string verbatim.

The canonical form begins with the command's defined mixed-case name. Aliases, capitalization differences, and alternative forms can therefore be normalized.

Two configuration fields influence further normalization. `bUseNames4canonical` controls whether symbolic constant names are used instead of integer values, while `bUseInt4numbers` controls whether integer-valued numerical arguments are emitted in normalized integer form. These controls are inherited from JDOR's canonical-output design [5].

Depending on the command, booleans can therefore be represented as `.true` or `.false`, enumeration values can use their symbolic names, and numeric or array arguments can be written in a normalized form.

Canonical output is intended to describe an accepted operation in a reproducible form. It is therefore distinct from diagnostic logging.

Comment and empty lines are handled differently. Recognized comment lines are copied unchanged to redirected output. When processing fails, the handler can write a comment-style `ERROR` or `FAILURE` diagnostic and create the corresponding Rexx condition rather than recording the

invalid command as a successful operation. The distinction between normal return codes and ERROR/FAILURE conditions follows the ooRexx command-handler contract [1].

4.6.2 Recording JDORFX Macros

Redirected canonical output can be used to record JDORFX command sequences.

An ooRexx program can redirect output from the JDORFX environment to a supported destination using the ordinary ADDRESS redirection mechanism [3]. Each successfully processed command contributes its canonical textual representation. The resulting recording represents the sequence of command-level operations.

Canonicalization makes the recording less dependent on the exact spelling of the original program. Aliases, capitalization, boolean forms, and symbolic constants can be normalized before being stored.

Return values are not automatically embedded in a canonical command line. ooRexx assigns a command handler's return value to RC when the handler returns [1]. For sequences that need to retain the current Rexx RC, the assignRC command can store it in a named Rexx context variable. Because assignRC itself has canonical output, it can participate in a recorded command stream.

4.6.3 Replaying Recorded Commands

Redirected input provides the complementary mechanism.

When handleCommand detects redirected input, it first processes the command that invoked the handler. It then repeatedly reads additional command lines from the redirected source and passes each of them to the same processCommand method used by ordinary commands.

Replayed commands therefore use the normal command lookup, parsing, validation, object resolution, return-value handling, and graphics implementation.

Processing continues until the redirected source contains no more commands. After each command read from redirected input, the handler checks whether a Rexx condition has been established. If so, it stops processing further redirected commands.

Listing 4.11 shows the redirected-input branch of the command handler. The ordinary single-command branch, update signaling, and diagnostics outside this range are omitted. Because the loop executes inside one handler invocation, ooRexx cannot update RC between its lines [1]; the explicit context assignment therefore preserves every non-null result for a following assignRC command in the replay stream.

```
2072        if (isInputRedirected(slot))    // take commands via redirected input
2073        {
```

```
2074            res=processCommand(slot, address, command, nrCommand, null);    // process kick-off command
        first
2075
2076            String currCommand=null;
2077            while ( (currCommand=readInput(slot)) != null)   // feed commands from redirected input
2078            {
2079                res=processCommand(slot, address, command, nrCommand, currCommand);
2080                if (checkCondition(slot))   // a Rexx condition got set: stop processing, return
        immediately
2081                {
2082                    return null;
2083                }
2084                // A redirected command stream is processed inside one handler
2085                // invocation, so ooRexx cannot update RC between its lines as it
2086                // does for individually addressed commands.  Preserve each
2087                // non-null result explicitly so a following assignRC command can
2088                // make the stream self-contained for macro replay.
2089                if (res!=null)
2090                {
2091                    setContextVariable(slot,"RC",res);
2092                }
2093            }
2094        }
```

Listing 4.11: Excerpt from `JavaFXDrawingHandler`: redirected-input processing and `RC` propagation

Since comment lines are recognized by the command processor, comments can also be included in a recorded command sequence.

The combination of canonical output and redirected input consequently forms a simple macro mechanism without requiring a second parser or execution engine.

Listing 4.12 shows both directions in one small example. It is explicitly an excerpt: handler registration, stream construction, cleanup, and program termination are omitted. The complete runnable `samples/26_jdorfx_macro_record_replay.rxj` appears as Listing B.6. Output redirection records accepted commands in canonical form, then input redirection feeds the saved stream to a second handler. The comment command triggers redirected-input processing; it is not a special replay language construct.

```
1 recording=.stream~new(macroFile)~~open("write replace")
2 address record with output using (recording) error using (.stderr)
3 newImage 320 180
4 background WHITE
5 color STEELBLUE
```

```
shape PANEL rectangle 40 35 240 110
fillShape PANEL
winShow
winClose
recording~close

commands=.stream~new(macroFile)~~open("read")
address replay with input using (commands) error using (.stderr)
"-- trigger redirected-input processing"
```

Listing 4.12: Excerpt from the JDORFX canonical recording and replay demonstration

Figure 4.11 summarizes the architectural path. The same textual boundary also permits a producer written in another language, for example `python producer.py | rexx jdorfx.rxj ?`; `?` is used as the comment-style kick-off command for redirected input. Rendering still occurs through the ooRexx command environment and JDORFX handler.

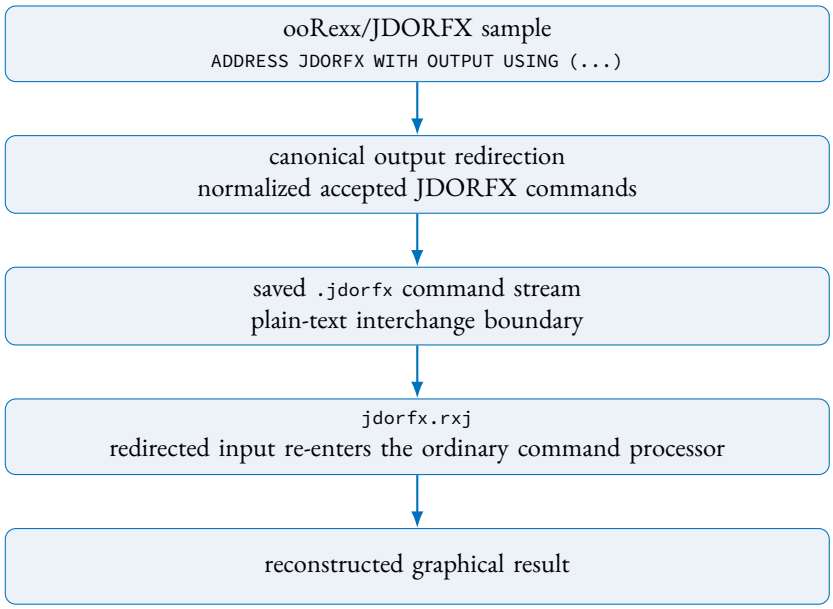


Figure 4.11: Canonical JDORFX command recording, storage, and replay workflow

4.6.4 Textual Command Interchange

Canonical command recordings are plain textual JDORFX commands rather than serialized JavaFX objects or language-specific executable representations.

This textual form can therefore also be generated, inspected, stored, or modified by software outside the immediate ooRexx source program. A resulting command stream can subsequently

be supplied to JDORFX through the supported ooRexx command-environment and redirection mechanisms.

This does not imply that JDORFX exposes a native graphics API for arbitrary programming languages. Execution still takes place through the JDORFX Rexx command handler and the BSF4ooRexx integration layer.

The interoperability lies in the textual boundary: an external producer only needs to construct valid JDORFX commands, while the handler remains responsible for argument validation, Java object lookup, JavaFX-thread handling, rendering, and error reporting.

Canonical output therefore supports three related purposes: reproducible command recording, replay of previously recorded operations, and a textual interchange representation that does not expose JDORFX's internal JavaFX object model.

5 Compatibility, Limitations, and Known Issues

The extensions described in Chapter 4 close the previously documented gaps in the JDOR command interface and reduce several behavioral differences between JDOR and JDORFX. The 31 JDOR commands that were listed as unimplemented in the inherited version are now represented by corresponding implementations, while several existing commands were revised to reproduce JDOR behavior more closely [38].

Complete command coverage does not imply exact behavioral or visual identity. JDOR is implemented on top of Java2D and primarily draws into a `BufferedImage` through `Graphics2D`, whereas JDORFX combines JavaFX Canvas drawing with retained Scene Graph objects. Some Java2D concepts have no direct JavaFX equivalent, and some operations cannot preserve both JDOR raster semantics and retained JavaFX behavior at the same time [16, 14, 17, 19].

This chapter summarizes the resulting compatibility status. It distinguishes between differences caused by the underlying graphics platforms, constraints of the JDORFX compatibility architecture, known behavior of JDOR itself, and JavaFX functionality that remains outside the current scope of JDORFX.

5.1 Compatibility with JDOR

The comparison with JDOR covers 100 canonical commands. In the final JDORFX implementation, 92 are implemented without a known compatibility limitation, three are implemented with a documented JavaFX platform limitation, five are partially compatible, and none remains completely unimplemented. This 100-command compatibility set is distinct from the complete JDORFX interface, which contains 140 canonical commands including JavaFX-specific extensions and is summarized in Appendix A. Internal enumeration sentinels are outside both counts.

The comparison considered command availability as well as arguments, return values, state changes, error handling, and other externally observable behavior. Commands that already existed in the inherited JDORFX implementation were also compared with JDOR where their JavaFX implementation differed from the original Java2D behavior.

For a large part of the interface, JDORFX can reproduce the JDOR behavior closely. This applies particularly to ordinary Canvas-based drawing, coordinate handling, colors, many font and Stroke settings, transformations, many image-management operations, command redirection, and the general nickname-based object model.

Compatibility is nevertheless not binary. A command can preserve its public syntax and state behavior while still differing visually or internally because the corresponding JavaFX API does not expose the same functionality as Java2D. The remaining differences are therefore treated as platform limitations or partial compatibility rather than as missing commands.

5.2 Differences Between Java2D and JavaFX

The main source of remaining differences is the graphics architecture itself.

JDOR uses a predominantly raster-oriented model. Drawing operations executed through `Graphics2D` immediately affect a `BufferedImage`. A Shape that is drawn or filled therefore becomes part of the same pixel destination as lines, images, text, and earlier Shapes [16, 14].

JDORFX uses JavaFX. Ordinary drawing commands operate on a `GraphicsContext` associated with a `Canvas`, while named 2D Shapes can remain JavaFX Scene Graph nodes. JavaFX-specific objects such as 3D Shapes, Cameras, Lights, and animations are also retained objects [17, 19, 21].

As Figure 3.2 shows, JDORFX combines an immediate Canvas path with a retained Scene Graph path. Retained nodes enable transformations, property changes, 3D rendering, and animation, but they also complicate operations whose semantics depend on one authoritative sequence of raster drawing operations.

The current implementation reduces this difference by grouping the Canvas and retained 2D renderings into one logical 2D presentation. Retained content can also be flattened into the Canvas when an operation requires raster semantics, as described in Section 4.3.1.

Flattening cannot completely remove the architectural distinction. Before flattening, Canvas pixels and retained Shapes remain separate JavaFX objects. Their visual order is therefore influenced by the Scene Graph structure as well as by command execution order. This differs from a `Graphics2D` destination in which every drawing operation immediately becomes part of the same raster.

Images present another difference. Java2D's `BufferedImage` exposes a specific storage type such as `TYPE_INT_ARGB` or `TYPE_3BYTE_BGR`. A JavaFX `Image` does not expose an equivalent `BufferedImage` storage-layout property. JDORFX therefore maintains type information as compatibility metadata where it is known. An arbitrary JavaFX Image supplied by the caller may consequently have no meaningful value that can be returned by the JDOR `imageType` interface [14, 18].

Graphics state also differs. JavaFX `GraphicsContext` provides many of the properties required for JDOR-style drawing, but not every state element has a direct Java2D equivalent or vice versa. The additional compatibility state described in Chapter 4 is therefore necessary for clipping, Composite operations, XOR mode, arbitrary Java Strokes, and related commands.

Finally, JavaFX introduces lifecycle and threading constraints that are not present in the same form in Java2D. Stage and Scene Graph modifications must respect the JavaFX Application Thread, and some Stage properties can only be configured at particular points in the Stage lifecycle. JDORFX handles these requirements through its update and synchronization mechanisms, but they remain part of the platform distinction [27, 32].

5.3 Remaining JDOR Compatibility Constraints

The remaining compatibility differences fall into two groups: platform limitations and partial compatibility. The first group contains commands whose public contract is implemented although JavaFX cannot reproduce one underlying Java2D feature exactly.

The three platform-limited commands are `newImage`, `render`, and `stroke`. JavaFX Canvas has no selectable `BufferedImage.TYPE_*` backing raster. `newImage` therefore preserves the JDOR type syntax, state, and metadata and applies constructible types when pixels are converted to a real `BufferedImage`; it cannot give the Canvas that storage layout [14, 17].

JDOR uses Java2D rendering hints to control geometric and text antialiasing. JavaFX Canvas does not provide an equivalent geometric antialiasing switch or the same ON/OFF/DEFAULT text-antialiasing model. JDORFX preserves the `render` command's values, query result, validation, canonical output, and graphics-state-stack behavior without claiming a corresponding visual switch [16, 17].

For `stroke`, Java2D defines a zero-width `BasicStroke` as a device-space hairline. JavaFX Canvas has no exact equivalent. The handler therefore retains zero as the logical width while native Canvas rendering uses width one [12, 17].

The five partially compatible commands are `image`, `clearRect`, `copyArea`, `drawShape`, and `fillShape`. The `image` command returns a visually complete `BufferedImage`, but it is detached. Modifying that returned object does not modify the current Canvas or a registered JavaFX Image as modifying JDOR's live backing image would.

The other four commands share one architectural limitation. Java2D records every 2D operation immediately in one raster, whereas retained JavaFX Shapes remain above Canvas pixels. JDORFX can commit the current retained 2D layer to the Canvas before destructive operations such as `clearRect` and `copyArea`, and it uses raster fallbacks when `drawShape` or `fillShape` requires non-retained rendering. These mechanisms preserve the current JavaFX layer order, but they can-

not reconstruct arbitrary earlier JDOR command chronology that has already been lost. A raster fallback also creates no retained animation target.

Composite and XOR commands themselves are implemented through native or software composition paths. They are relevant here because they can require a Shape rendering to use the raster fallback, but they are not classified as additional partially compatible commands.

Comparison with the Java2D implementation used by JDOR showed the transformation boundary of `copyArea`: translation and positive axis scaling transform the source and destination, while rotation, shear, negative scaling, and general affine transformations are rejected. JDORFX follows the same boundary, so these rejected transformation forms are not treated as additional incompatibilities.

These eight commands are therefore compatibility classifications rather than missing functionality. The three platform-limited commands implement their command contracts subject to the stated JavaFX constraints, while the five partially compatible commands retain the specific differences described above.

5.4 Known JDOR Issues Affecting JDORFX

Using JDOR as a compatibility reference also raises the question of whether JDORFX should reproduce the intended semantics of a command or the behavior of an apparent defect in a particular JDOR implementation.

The development history embedded in the Java drawing handler documents a number of corrections made to JDOR before JDORFX was created. These include fixes to coordinate values, `imageSize`, `drawString`, `winLocation`, `loadImage`, query/set return values, Composite handling, Shape operations, redirected command processing, and other parts of the interface. The reference behavior of JDOR has therefore itself changed over time [37].

Comparison with the JDOR handler source also showed an issue in the seven- and eight-argument forms of `drawImage`. These forms specify a destination size together with a source rectangle and an optional background color. The corresponding Java2D `Graphics2D.drawImage` overload expects the coordinates of two destination corners, but the JDOR implementation passes `width` and `height` directly as the second corner instead of using `currX + width` and `currY + height`. The destination is therefore positioned or scaled incorrectly whenever the current position is not the origin. The Java2D overload defines the destination through two corner pairs [5, 16].

JDORFX does not intentionally reproduce this behavior. Its corresponding JavaFX operation interprets the destination arguments as x-position, y-position, width, and height and therefore follows the intended command semantics.

A separate issue occurs in the no-argument `printPos` query in the JDOR handler. Although its argument-count check accepts the query form, the implementation evaluates `arrCommand[1]` before checking whether an argument exists. Issuing `printPos` without an argument therefore raises an array-index error instead of returning the current print position [5]. JDORFX guards the setter logic with an argument-count test, so its no-argument form returns the current x- and y-values and leaves them unchanged.

These cases show why compatibility cannot always mean reproducing every observed result of one implementation literally. Where JDOR behavior conflicts with the intended command meaning or with the contract of the underlying Java API, JDORFX follows the intended semantics rather than deliberately reproducing the same defect.

Known JDOR defects are therefore documented separately from genuine Java2D–JavaFX compatibility differences. Otherwise, an error in the reference implementation could incorrectly be classified as a JDORFX incompatibility.

5.5 JavaFX-Specific Limitations

JDORFX also has limitations that are independent of JDOR compatibility.

The current framework exposes selected parts of JavaFX rather than the entire JavaFX graphics and user-interface API. Its 3D Shape interface focuses on `Box`, `Cylinder`, and `Sphere`, together with Cameras, Lights, Phong materials, texture maps, transformations, culling, and animation.

Other JavaFX facilities proposed as possible extensions of the original project remain outside the scope of the current implementation. These include user-defined mesh geometry through `TriangleMesh` and `MeshView`, dedicated commands for creating and managing graphical effects, `SubScene`, charts, media display, `WebView`, and event handling. Schaller's extension notes identify these areas as possible future functionality rather than part of the original JDORFX interface [41].

Animation is likewise limited to properties and JavaFX Animation classes explicitly exposed by the command handler. The general JavaFX property system contains many more writable properties than those currently addressable through `keyValue`. Supporting arbitrary JavaFX properties would require a more general property-resolution or reflection mechanism.

Retained 2D Shape animation also depends on the continued existence of the rendered node. If a later JDOR-compatible raster operation causes the retained 2D presentation to be flattened into the Canvas, the corresponding node ceases to exist as an independent animation target. This is a direct consequence of supporting both JDOR raster operations and JavaFX retained animation in the same framework.

JavaFX rendering can additionally vary with the JavaFX implementation, graphics pipeline, operating system, font availability, and hardware. Minor pixel-level or font-rendering differences should therefore not automatically be interpreted as command-interface incompatibilities.

The framework remains centered on a command-oriented interface rather than exposing the complete JavaFX object model directly to ooRexx. This restriction preserves the usage model inherited from JDOR while allowing selected JavaFX-specific functionality to be added through named commands and objects.

5.6 Final Compatibility Status

The 31 commands documented as missing in the inherited JDORFX implementation are now implemented [38]. No canonical JDOR command in the 100-command compatibility set remains completely unsupported.

This does not mean that every command is behaviorally identical to JDOR. The platform limitations and partial compatibility cases described in Sections 5.2 and 5.3 remain part of the final implementation.

JavaFX functionality that was never part of the JDOR command interface is not counted as an unsupported JDOR command. Features such as mesh construction, effects, SubScenes, charts, media, WebView integration, and event handling are possible future extensions of JDORFX rather than remaining JDOR compatibility gaps.

Table 5.1 summarizes the 100-command JDOR compatibility set.

Table 5.1: Summary of the current JDORFX compatibility status

Category	Status
Implemented	92 canonical JDOR commands.
Implemented with documented platform limitation	3 commands: `newImage`, `render`, and `stroke`.
Partially compatible	5 commands: `image`, `clearRect`, `copyArea`, `drawShape`, and `fillShape`.
Missing	0 canonical JDOR commands.

The documented JDORFX command interface contains 140 canonical commands because it also includes JavaFX-specific functionality such as 3D graphics and animation. The principal remaining limitations are therefore no longer missing JDOR commands, but differences imposed by the underlying graphics technologies and the deliberate scope of the JavaFX-specific interface.

6 Conclusion

This thesis continued the development of JDORFX, a JavaFX-based graphics framework that makes selected JavaFX graphics functionality available to ooRexx programs through the BSF4ooRexx command-handler infrastructure. The inherited implementation already provided the foundation of the framework, including Canvas-based 2D drawing, retained JavaFX Shapes, basic 3D graphics, Cameras, Lights, materials, and transformations. At the same time, a substantial part of the JDOR command interface was still missing and several behavioral differences remained between the Java2D-based JDOR model and the JavaFX implementation.

The work presented in this thesis therefore concentrated on two broad goals. The first was to improve compatibility with JDOR by implementing the remaining commands and correcting behavior where a direct translation to JavaFX did not reproduce the intended JDOR semantics. The second was to extend JDORFX with functionality that takes advantage of JavaFX itself, particularly additional 3D controls and animation support.

The resulting implementation closes the previously documented command gap while retaining the JavaFX-specific capabilities that distinguish JDORFX from JDOR. Complete command coverage does not, however, imply exact rendering equivalence. The different graphics models of Java2D and JavaFX leave several limitations that must be handled explicitly.

6.1 Summary of the Work

The first major area of work was JDOR compatibility. The inherited JDORFX implementation contained 31 canonical JDOR commands that were represented by the command interface but not processed by the handler. These commands covered image handling, printing, clipboard operations, Paint and Composite functionality, graphics-state management, clipping, image stacks, and several additional drawing and geometry operations.

All of these previously missing commands were implemented. Image operations required particular attention because JavaFX `Image` objects and Java2D `BufferedImage` objects expose different functionality. Additional metadata and conversion mechanisms were therefore introduced where JDOR semantics depend on Java2D image types or raster access.

Several other commands required additional compatibility state or fallback mechanisms for Java2D concepts that JavaFX does not represent directly. These include clipping, compositing, XOR drawing, graphics-state stacks, arbitrary Strokes, and related operations.

A second major area of work concerned the interaction between immediate-mode Canvas drawing and retained Scene Graph objects. In the inherited architecture, these two rendering approaches could produce observable differences from the single-raster model used by JDOR.

The revised implementation groups the Canvas and retained 2D Shape renderings into one logical 2D presentation. Operations that require the complete current image can therefore include both kinds of content. Where an operation requires raster semantics, retained 2D content can be committed to the Canvas before the operation proceeds. This allows JDORFX to preserve retained JavaFX Shapes where possible while switching to raster processing when required by JDOR semantics.

Window and thread handling were also revised to support multiple JDORFX handler instances within the shared JavaFX runtime. Each handler maintains its own drawing state and window, while operations whose completion forms part of the command semantics are executed synchronously on the JavaFX Application Thread. Parsing, validation, and Rexx-side processing remain on the calling thread.

The third area of work was the extension of JDORFX-specific functionality. Six additional 3D-control commands were introduced. These allow existing Cameras to be translated and rotated, expose the near and far clipping planes of Cameras, allow PointLights to be translated, and provide access to JavaFX face culling for 3D Shapes.

Animation support represents the largest new JavaFX-specific extension. Eighteen animation-related commands were added. They expose predefined JavaFX transitions, playback control, interpolation, sequential and parallel composition, timed pauses, Timelines, KeyFrames, and KeyValues. Retained 2D Shapes, 3D Shapes, Cameras, and Lights can participate in this animation infrastructure through the existing nickname-based object model.

Finally, the existing input and output redirection mechanism was retained and extended to the newly implemented commands. Canonical command output can be recorded and later supplied again as redirected input, providing a textual mechanism for command recording and replay.

6.2 Main Contributions and Results

The final comparison with JDOR covers 100 canonical commands:

- 92 are implemented without a known compatibility limitation,
- 3 are implemented with documented JavaFX platform limitations,

- 5 remain partially compatible, and
- 0 remain completely unimplemented.

The three platform-limited commands are `newImage`, `render`, and `stroke`. In these cases, the command interface is implemented, but JavaFX does not expose an exact equivalent of the relevant underlying Java2D feature or property.

The five partially compatible commands are `image`, `clearRect`, `copyArea`, `drawShape`, and `fillShape`. Their remaining differences primarily result from the interaction between JavaFX retained nodes and the raster-oriented behavior of JDOR.

Implementing the missing commands required more than adding new command cases. Several operations depended on architectural changes that reconcile JDOR's raster model with JavaFX retained nodes. This is particularly visible in the treatment of retained Shapes. Keeping a Shape as a JavaFX node enables transformations and animation, but some JDOR operations require the complete current raster result. The implemented flattening mechanism provides a controlled transition between these two representations.

The same principle applies to Composite rules, XOR drawing, clipping, arbitrary Strokes, and image operations. Where JavaFX does not provide a direct equivalent of the required Java2D behavior, JDORFX maintains additional compatibility state or uses software processing.

Beyond compatibility, the six additional 3D commands make Cameras, Lights, and Shape culling more configurable after object creation. The 18 animation commands add a retained-property animation layer and make it possible to animate Shapes, Cameras, and Lights through the same command-oriented interface.

The resulting JDORFX implementation covers all 100 canonical JDOR commands considered in the comparison while retaining JavaFX-specific extensions for 3D graphics and animation. The documented JDORFX command interface contains 140 canonical commands.

6.3 Future Work

Although the canonical JDOR command gap has been closed, several areas remain for further development.

Future work could introduce a systematic cross-framework test suite that executes equivalent command sequences in JDOR and JDORFX. Such a suite could compare return values, error conditions, graphics-state changes, and generated raster output. It would be particularly useful for the five commands classified as partially compatible and for operations involving transformations, clipping, Composite rules, and retained Shapes.

A second area is further refinement of the mixed retained/raster architecture. The current solution deliberately prioritizes JDOR raster semantics when a command requires them. As a consequence, a retained 2D Shape can lose its independent Scene Graph representation when the 2D presentation is flattened.

A more detailed ordering model could be investigated to determine whether additional Scene Graph information can be retained while reproducing historical drawing order more closely. This would be especially useful for applications that combine JDOR-compatible destructive drawing operations with long-lived JavaFX animations.

There is also scope for extending the JavaFX-specific command interface. Possible additions include user-defined 3D geometry through `TriangleMesh` and `MeshView`, broader support for JavaFX effects, `SubScene`, charting, media playback, `WebView`, and event handling.

Event handling would be a particularly useful extension because the current framework is primarily command-driven. Exposing JavaFX events to ooRexx would allow JDORFX applications to respond directly to keyboard, mouse, and other user-interface events instead of relying primarily on externally controlled command sequences.

Animation can also be extended. The current `keyValue` interface supports a selected set of writable properties. A more general property-resolution mechanism could expose a larger part of the JavaFX property system without requiring one explicit mapping for every property. Additional animation types and event callbacks could further expand the kinds of interactive JavaFX applications that can be expressed through JDORFX.

The work presented in this thesis provides a basis for these extensions by completing the canonical JDOR command coverage, revising the compatibility architecture, and extending JDORFX with additional 3D controls and JavaFX animation support.

References

[1] W. David Ashley, Rony G. Flatscher, Mark Hessling, Rick McGuire, Lee Peedin, Oliver Sims, Erich Steinböck, and Jon Wolfers. *Open Object Rexx: Application Programming Interfaces*. Rexx Language Association, 2022.12.22 edition, 2022. ooRexx Documentation 5.0.0, accessed 11 October 2026. URL: https://www.oorexx.org/docs/pdf/rexxapi.pdf.

[2] W. David Ashley, Rony G. Flatscher, Mark Hessling, Rick McGuire, Lee Peedin, Oliver Sims, Erich Steinböck, and Jon Wolfers. *Open Object Rexx: Programmer Guide*. Rexx Language Association, 2022.12.22 edition, 2022. ooRexx Documentation 5.0.0, accessed 16 September 2026. URL: https://www.oorexx.org/docs/pdf/rexxpg.pdf.

[3] W. David Ashley, Rony G. Flatscher, Mark Hessling, Rick McGuire, Lee Peedin, Oliver Sims, Erich Steinböck, and Jon Wolfers. *Open Object Rexx: Reference*. Rexx Language Association, 2022.12.22 edition, 2022. ooRexx Documentation 5.0.0, accessed 16 September 2026. URL: https://www.oorexx.org/docs/pdf/rexxref.pdf.

[4] BSF4ooRexx Project. BSF4ooRexx850: A bidirectional ooRexx–Java bridge. Official project page, n.d. Accessed 16 September 2026. URL: https://sourceforge.net/projects/bsf4oorexx/.

[5] Rony G. Flatscher. JavaDrawingHandler.java. JDOR handler source file, version 100.20231114, 2023. Unpublished project source file.

[6] Alireza Ismaili. JDORFX. GitHub repository, 2026. Accessed 22 October 2026. URL: https://github.com/aismaili10/jdorfx_pub.

[7] Oracle. Affine (JavaFX 8). JavaFX API documentation, n.d. Accessed 29 September 2026. URL: https://docs.oracle.com/javase/8/javafx/api/javafx/scene/transform/Affine.html.

[8] Oracle. Alphacomposite (Java se 8). Java API documentation, n.d. Accessed 10 October 2026. URL: https://docs.oracle.com/javase/8/docs/api/java/awt/AlphaComposite.html.

[9] Oracle. AmbientLight (JavaFX 8). JavaFX API documentation, n.d. Accessed 16 September 2026. URL: https://docs.oracle.com/javase/8/javafx/api/javafx/scene/AmbientLight.html.

[10] Oracle. Animation (JavaFX 8). JavaFX API documentation, n.d. Accessed 29 September 2026. URL: https://docs.oracle.com/javase/8/javafx/api/javafx/animation/Animation.html.

[11] Oracle. Application (JavaFX 8). JavaFX API documentation, n.d. Accessed 14 September 2026. URL: https://docs.oracle.com/javase/8/javafx/api/javafx/application/Application.html.

[12] Oracle. BasicStroke (Java platform SE 8). Java SE API documentation, n.d. Accessed 19 September 2026. URL: https://docs.oracle.com/javase/8/docs/api/java/awt/BasicStroke.html.

[13] Oracle. Blendmode (JavaFX 8). JavaFX API documentation, n.d. Accessed 10 October 2026. URL: https://docs.oracle.com/javase/8/javafx/api/javafx/scene/effect/BlendMode.html.

[14] Oracle. BufferedImage (Java platform SE 8). Java SE API documentation, n.d. Accessed 14 September 2026. URL: https://docs.oracle.com/javase/8/docs/api/java/awt/image/BufferedImage.html.

[15] Oracle. Camera (JavaFX 8). JavaFX API documentation, n.d. Accessed 29 September 2026. URL: https://docs.oracle.com/javase/8/javafx/api/javafx/scene/Camera.html.

[16] Oracle. Graphics2D (Java platform SE 8). Java SE API documentation, n.d. Accessed 14 September 2026. URL: https://docs.oracle.com/javase/8/docs/api/java/awt/Graphics2D.html.

[17] Oracle. GraphicsContext (JavaFX 8). JavaFX API documentation, n.d. Accessed 14 September 2026. URL: https://docs.oracle.com/javase/8/javafx/api/javafx/scene/canvas/GraphicsContext.html.

[18] Oracle. Image (JavaFX 8). JavaFX API documentation, n.d. Accessed 29 September 2026. URL: https://docs.oracle.com/javase/8/javafx/api/javafx/scene/image/Image.html.

[19] Oracle. JavaFX architecture and the scene graph. JavaFX documentation, n.d. Accessed 14 September 2026. URL: https://docs.oracle.com/javase/8/javafx/get-started-tutorial/jfx-architecture.htm.

[20] Oracle. LightBase (JavaFX 8). JavaFX API documentation, n.d. Accessed 29 September 2026. URL: https://docs.oracle.com/javase/8/javafx/api/javafx/scene/LightBase.html.

[21] Oracle. Node (JavaFX 8). JavaFX API documentation, n.d. Accessed 29 September 2026. URL: https://docs.oracle.com/javase/8/javafx/api/javafx/scene/Node.html.

[22] Oracle. Overview (JavaFX 8). JavaFX API documentation, n.d. Accessed 29 September 2026. URL: https://docs.oracle.com/javase/8/javafx/api/overview-summary.html.

[23] Oracle. PerspectiveCamera (JavaFX 8). JavaFX API documentation, n.d. Accessed 16 September 2026. URL: https://docs.oracle.com/javase/8/javafx/api/javafx/scene/PerspectiveCamera.html.

[24] Oracle. PhongMaterial (JavaFX 8). JavaFX API documentation, n.d. Accessed 29 September 2026. URL: https://docs.oracle.com/javase/8/javafx/api/javafx/scene/paint/PhongMaterial.html.

[25] Oracle. PixelReader (JavaFX 8). JavaFX API documentation, n.d. Accessed 16 September 2026. URL: https://docs.oracle.com/javase/8/javafx/api/javafx/scene/image/PixelReader.html.

[26] Oracle. PixelWriter (JavaFX 8). JavaFX API documentation, n.d. Accessed 16 September 2026. URL: https://docs.oracle.com/javase/8/javafx/api/javafx/scene/image/PixelWriter.html.

[27] Oracle. Platform (JavaFX 8). JavaFX API documentation, n.d. Accessed 14 September 2026. URL: https://docs.oracle.com/javase/8/javafx/api/javafx/application/Platform.html.

[28] Oracle. PointLight (JavaFX 8). JavaFX API documentation, n.d. Accessed 16 September 2026. URL: https://docs.oracle.com/javase/8/javafx/api/javafx/scene/PointLight.html.

[29] Oracle. PrinterJob (JavaFX 8). JavaFX API documentation, n.d. Accessed 19 September 2026. URL: https://docs.oracle.com/javase/8/javafx/api/javafx/print/PrinterJob.html.

[30] Oracle. Scene (JavaFX 8). JavaFX API documentation, n.d. Accessed 16 September 2026. URL: https://docs.oracle.com/javase/8/javafx/api/javafx/scene/Scene.html.

[31] Oracle. Shape3D (JavaFX 8). JavaFX API documentation, n.d. Accessed 29 September 2026. URL: https://docs.oracle.com/javase/8/javafx/api/javafx/scene/shape/Shape3D.html.

[32] Oracle. Stage (JavaFX 8). JavaFX API documentation, n.d. Accessed 16 September 2026. URL: https://docs.oracle.com/javase/8/javafx/api/javafx/stage/Stage.html.

[33] Oracle. Timeline (JavaFX 8). JavaFX API documentation, n.d. Accessed 29 September 2026. URL: https://docs.oracle.com/javase/8/javafx/api/javafx/animation/Timeline.html.

[34] Oracle. WritableImage (JavaFX 8). JavaFX API documentation, n.d. Accessed 16 September 2026. URL: https://docs.oracle.com/javase/8/javafx/api/javafx/scene/image/WritableImage.html.

[35] Oracle Corporation. Animation basics (JavaFX 8). JavaFX tutorial, 2014. Accessed 19 September 2026. URL: https://docs.oracle.com/javase/8/javafx/visual-effects-tutorial/basics.htm.

[36] Oracle Corporation. Tree animation example (JavaFX 8). JavaFX tutorial and accompanying source archive, 2014. Accessed 19 September 2026. URL: https://docs.oracle.com/javase/8/javafx/visual-effects-tutorial/tree-animation.htm.

[37] Philipp Schaller. JavaFXDrawingHandler.java. Source file in the WU-hosted JDORFX code archive, 2024. Original handler source submitted with the bachelor's thesis. URL: https://wi.wu.ac.at/rgf/diplomarbeiten/BakkStuff/2024/20240527-Schaller_JDORFX_code.zip.

[38] Philipp Schaller. JDOR commands not implemented in JDORFX. Project documentation in the WU-hosted JDORFX code archive, 2024. File `jdorfx_commands_not_implemented.txt`, dated 26 May 2024. URL: https://wi.wu.ac.at/rgf/diplomarbeiten/BakkStuff/2024/20240527-Schaller_JDORFX_code.zip.

[39] Philipp Schaller. JDORFX command inventory. Project documentation in the WU-hosted JDORFX code archive, 2024. File `jdorfx_commands.txt`, dated 26 May 2024. URL: https://wi.wu.ac.at/rgf/diplomarbeiten/BakkStuff/2024/20240527-Schaller_JDORFX_code.zip.

[40] Philipp Schaller. Notes for existing JDORFX commands. Project documentation in the WU-hosted JDORFX code archive, 2024. File `Notes for existing JDORFX commands.txt`,

dated 7 June 2024. URL: https://wi.wu.ac.at/rgf/diplomarbeiten/BakkStuff/2024/20240527-Schaller_JDORFX_code.zip.

[41] Philipp Schaller. Possible extensions of JDORFX. Project documentation in the WU-hosted JDORFX code archive, 2024. File `Possible extensions of JDORFX.txt`, dated 7 June 2024. URL: https://wi.wu.ac.at/rgf/diplomarbeiten/BakkStuff/2024/20240527-Schaller_JDORFX_code.zip.

[42] Philipp Schaller. Transforming JDOR into JDORFX: Providing 3D graphics to ooRexx. Bachelor's thesis, Vienna University of Economics and Business (WU), May 2024. Department of Information Systems & Society, submitted 26 May 2024. URL: https://wi.wu.ac.at/rgf/diplomarbeiten/BakkStuff/2024/20240527-Schaller_JDORFX_BachelorThesis.pdf.

A Complete JDORFX Command Interface

A.1 Command Syntax and Conventions

JDORFX commands are case-insensitive. Nicknames identify objects retained in the handler's registries. In the syntax below, square brackets mark optional arguments and a vertical bar separates alternatives. Many query/set commands query when the optional value is absent and set when it is present; returned values are made available through the Rexx variable `RC`. The handler accepts several aliases, but this appendix lists only the canonical names used by canonical output. That output normalizes command names and, where applicable, argument forms. An ellipsis indicates that additional arguments of the same kind may follow.

This appendix lists all 140 canonical JDORFX commands. The first 100 entries, through Section A.6, correspond to the canonical JDOR compatibility set discussed in Chapter 5. The remaining 40 entries provide JavaFX-specific 3D and animation functionality. The HTML synopsis remains the detailed source for parameter constraints and error conditions.

A.2 Window and Scene Commands

`assignRC` *Syntax:* `assignRC rexxVariableName`. *Description:* Copies the current Rexx `RC` value to a named Rexx variable. *Return:* The unchanged value of `RC`. *See:* Section 4.6.3.

`GC` *Syntax:* `GC`. *Description:* Returns the current JavaFX `GraphicsContext`. *Return:* The active `GraphicsContext`.

`getState` *Syntax:* `getState [rexxVariableName]`. *Description:* Obtains a table describing current drawing state and registered objects, optionally assigning it to a Rexx variable. *Return:* The state table.

`newImage` *Syntax:* `newImage [width height [type [depthBuffer]]] | newImage width height PREFERRED depthBuffer`. *Description:* Creates a new Canvas-backed drawing surface. In the four-argument form, `PREFERRED` retains the preferred image type while specifying the JavaFX Scene depth buffer.

`reset` *Syntax:* `reset`. *Description:* Clears registries and stacks and resets position, transform, font metadata, and print settings. The current Canvas, its contents, and several active drawing settings are preserved.

sleep *Syntax:* `sleep seconds`. *Description:* Suspends command processing for the specified number of seconds.

winAlwaysOnTop *Syntax:* `winAlwaysOnTop [booleanValue]`. *Description:* Queries or changes whether the Stage stays above other windows. *Return:* Query: current Boolean state; set: previous state.

winAlwaysOnTopSupported *Syntax:* `winAlwaysOnTopSupported`. *Description:* Tests whether the platform supports always-on-top windows. *Return:* `1` when supported, otherwise `0`.

winClose *Syntax:* `winClose`. *Description:* Closes the Stage while preserving the Scene and drawing state.

winFrame *Syntax:* `winFrame [booleanValue]`. *Description:* Queries or changes whether the Stage is decorated. *Return:* Query: current Boolean state; set: previous state.

winHide *Syntax:* `winHide`. *Description:* Hides the current Stage without discarding it.

winLocation *Syntax:* `winLocation [x y]`. *Description:* Queries or changes the Stage position on screen. *Return:* Query: `x y`; set: previous position.

winResizable *Syntax:* `winResizable [booleanValue]`. *Description:* Queries or changes whether the Stage can be resized. *Return:* Query: current Boolean state; set: previous state.

winScreenSize *Syntax:* `winScreenSize`. *Description:* Queries the dimensions of the current screen. *Return:* `width height`.

winShow *Syntax:* `winShow`. *Description:* Shows the current Stage, creating one when necessary.

winSize *Syntax:* `winSize [width height]`. *Description:* Queries or changes the Stage's content size without resizing the Canvas. *Return:* Query: `width height`; set: previous size.

winTitle *Syntax:* `winTitle [title]`. *Description:* Queries or changes the Stage title, treating the remaining command text as the title. *Return:* Query: current title; set: previous title.

winToBack *Syntax:* `winToBack`. *Description:* Requests that the Stage move behind other windows.

winToFront *Syntax:* `winToFront`. *Description:* Requests that the Stage move in front of other windows.

winUpdate *Syntax:* `winUpdate [booleanValue]`. *Description:* Queries or changes automatic presentation of Canvas and retained 2D updates. *Return:* Query: current Boolean state; set: previous state.

winVisible *Syntax:* `winVisible [booleanValue]`. *Description:* Queries or changes Stage visibility. *Return:* Query: current Boolean state; set: previous state.

A.3 2D Drawing Commands

background *Syntax:* `background [colorNickName]`. *Description:* Queries or changes the clearing and Scene background color. *Return:* Query: current color; set: previous color.

clearRect *Syntax:* `clearRect width height`. *Description:* Clears a rectangle at the current position using the background color.

clip *Syntax:* `clip [x y width height]`. *Description:* Queries the current clip bounds or intersects the clip with a rectangle. *Return:* The previous bounds as `x y width height`, or `.nil`.

clipRemove *Syntax:* `clipRemove`. *Description:* Removes the current drawing clip.

`color` *Syntax:* `color [colorNickName [r g b [alpha]]]`. *Description:* Queries, selects, or defines the common fill and stroke color. *Return:* Query: current color; select/define: previous color.

`composite` *Syntax:* `composite [rule [alpha]]`. *Description:* Queries or changes the JDOR-compatible alpha-compositing rule. *Return:* Query: current composite; set: previous composite.

`copyArea` *Syntax:* `copyArea width height dX dY`. *Description:* Copies a rectangular region of the current 2D result from the current position by the supplied offset.

`draw3DRect` *Syntax:* `draw3DRect width height raised`. *Description:* Strokes a raised or sunken two-dimensional rectangle.

`drawArc` *Syntax:* `drawArc width height startAngle arcAngle`. *Description:* Strokes an open arc in the current-position bounding box.

`drawLine` *Syntax:* `drawLine toX toY`. *Description:* Strokes a line from the current position to the target point.

`drawOval` *Syntax:* `drawOval width height`. *Description:* Strokes an oval in the current-position bounding box.

`drawPolygon` *Syntax:* `drawPolygon xPointsArray yPointsArray nPoints`. *Description:* Strokes a closed polygon from the first `nPoints` coordinate pairs.

`drawPolyline` *Syntax:* `drawPolyline xPointsArray yPointsArray nPoints`. *Description:* Strokes an open polyline from the first `nPoints` coordinate pairs.

`drawRect` *Syntax:* `drawRect width height`. *Description:* Strokes a rectangle at the current position.

`drawRoundRect` *Syntax:* `drawRoundRect width height arcWidth arcHeight`. *Description:* Strokes a rounded rectangle at the current position.

`drawString` *Syntax:* `drawString text`. *Description:* Draws the complete remaining command text at the current position.

`fill3DRect` *Syntax:* `fill3DRect width height raised`. *Description:* Fills a raised or sunken two-dimensional rectangle.

`fillArc` *Syntax:* `fillArc width height startAngle arcAngle`. *Description:* Fills a round arc in the current-position bounding box.

`fillOval` *Syntax:* `fillOval width height`. *Description:* Fills an oval in the current-position bounding box.

`fillPolygon` *Syntax:* `fillPolygon xPointsArray yPointsArray nPoints`. *Description:* Fills a closed polygon from the first `nPoints` coordinate pairs.

`fillRect` *Syntax:* `fillRect width height`. *Description:* Fills a rectangle at the current position.

`fillRoundRect` *Syntax:* `fillRoundRect width height arcWidth arcHeight`. *Description:* Fills a rounded rectangle at the current position.

`font` *Syntax:* `font [fontNickName [fontDescription]]`. *Description:* Queries, selects, or defines the active JavaFX font. *Return:* Query: current font; select/define: previous font.

`fontSize` *Syntax:* `fontSize [size]`. *Description:* Queries or changes the size metadata used when defining fonts. *Return:* Query: current size; set: previous size.

fontStyle *Syntax:* `fontStyle [PLAIN|BOLD|ITALIC|BOLDITALIC|value]`. *Description:* Queries or changes the style metadata used when defining fonts. *Return:* Query: current numeric style; set: previous style.

gradientPaint *Syntax:* `gradientPaint [name [x1 y1 color1 x2 y2 color2 [cyclic]]]`. *Description:* Queries, selects, or defines a linear gradient paint. *Return:* Query: current paint; select/define: previous paint.

paint *Syntax:* `paint [paintNickName]`. *Description:* Queries or selects a registered color or gradient as the current paint. *Return:* Query: current paint; set: previous paint.

popGC *Syntax:* `popGC`. *Description:* Restores the most recently pushed graphics state. *Return:* The restored `GraphicsContext`.

pushGC *Syntax:* `pushGC`. *Description:* Pushes a copy of the current graphics state onto the graphics-state stack. *Return:* The current `GraphicsContext`.

render *Syntax:* `render [setting1 [setting2]]`. *Description:* Queries or changes the canonical JDOR rendering-hint settings. *Return:* Query: current settings; set: previous settings.

setPaintMode *Syntax:* `setPaintMode`. *Description:* Restores ordinary source-over drawing after XOR mode.

setXorMode *Syntax:* `setXorMode colorNickName`. *Description:* Enables XOR drawing using the supplied color.

stringBounds *Syntax:* `stringBounds text`. *Description:* Measures the remaining text using the active font. *Return:* Logical bounds as `x y width height`.

stroke *Syntax:* `stroke [name [width [cap join [miterLimit [dashArray dashPhase]]]]]`. *Description:* Queries, selects, or defines the active AWT-compatible stroke. *Return:* Query: current stroke; select/define: previous stroke.

A.4 2D Shape Commands

areaAdd *Syntax:* `areaAdd areaName shapeName`. *Description:* Adds a Shape's area to a registered Area.

areaExclusiveOr *Syntax:* `areaExclusiveOr areaName shapeName`. *Description:* Applies an exclusive-or area operation to a registered Area.

areaIntersect *Syntax:* `areaIntersect areaName shapeName`. *Description:* Intersects a registered Area with another Shape.

areaSubtract *Syntax:* `areaSubtract areaName shapeName`. *Description:* Subtracts a Shape's area from a registered Area.

areaTransform *Syntax:* `areaTransform areaName transformName`. *Description:* Applies a registered affine transform to an Area.

clipShape *Syntax:* `clipShape shapeName`. *Description:* Intersects the current drawing clip with a Shape's geometry.

drawShape *Syntax:* `drawShape shapeName`. *Description:* Draws a registered or Rexx-variable Shape, using retained rendering where compatible with the current graphics state.

`fillShape` *Syntax:* `fillShape shapeName`. *Description:* Fills a registered or Rexx-variable Shape, using retained rendering where compatible with the current graphics state.

`pathAppend` *Syntax:* `pathAppend pathName sourceName [connect]`. *Description:* Appends Shape or path-iterator geometry to a registered JavaFX Path.

`pathClone` *Syntax:* `pathClone pathName [cloneName]`. *Description:* Creates an independent geometry clone of a registered Path. *Return:* The cloned Path.

`pathClose` *Syntax:* `pathClose pathName`. *Description:* Closes the current subpath of a registered Path.

`pathCurrentPoint` *Syntax:* `pathCurrentPoint pathName`. *Description:* Queries the current endpoint of a registered Path. *Return:* `x y`, or `.nil` for an empty Path.

`pathCurveTo` *Syntax:* `pathCurveTo pathName cx1 cy1 cx2 cy2 x y`. *Description:* Adds a cubic Bézier segment to a registered Path.

`pathIterator` *Syntax:* `pathIterator shapeName [transformName [flatness]]`. *Description:* Creates an AWT path iterator over supported JavaFX or AWT Shape geometry. *Return:* The `PathIterator`.

`pathLineTo` *Syntax:* `pathLineTo pathName x y`. *Description:* Adds a line segment to a registered Path.

`pathMoveTo` *Syntax:* `pathMoveTo pathName x y`. *Description:* Starts a new subpath at the supplied point.

`pathQuadTo` *Syntax:* `pathQuadTo pathName cx cy x y`. *Description:* Adds a quadratic Bézier segment to a registered Path.

`pathReset` *Syntax:* `pathReset pathName`. *Description:* Removes all elements from a registered Path.

`pathTransform` *Syntax:* `pathTransform pathName transformName`. *Description:* Changes a Path's coordinates using an affine transform.

`pathWindingRule` *Syntax:* `pathWindingRule pathName [WIND_EVEN_ODD|WIND_NON_ZERO|value]`. *Description:* Queries or changes a Path's fill rule. *Return:* Query: current numeric rule; set: previous rule.

`shape` *Syntax:* `shape name [type typeArguments]`. *Description:* Queries or creates a named Area, Arc, curve, ellipse, line, Path, Polygon, rectangle, or rounded rectangle. *Return:* Query/creation: the registered Shape.

`shapeBounds` *Syntax:* `shapeBounds shapeName`. *Description:* Queries a Shape's untransformed geometric bounds. *Return:* `x y width height`.

A.5 Image, Clipboard, and Print Commands

`clipboardGet` *Syntax:* `clipboardGet [imageName]`. *Description:* Reads the system clipboard image into the image registry. Absence of a clipboard image raises an error. *Return:* The imported image on success. *See:* Section 4.2.3.

`clipboardSet` *Syntax:* `clipboardSet [imageName]`. *Description:* Copies the current or named image to the system clipboard with alpha. *Return:* The copied `BufferedImage`. *See:* Section 4.2.3.

clipboardSetWithoutAlpha *Syntax:* `clipboardSetWithoutAlpha [imageName]`. *Description:* Copies the current or named image to the clipboard after removing alpha. *Return:* The alpha-free `BufferedImage`. *See:* Section 4.2.3.

drawImage *Syntax:* `drawImage imageName [backgroundColor] | drawImage imageName width height [backgroundColor] | drawImage imageName width height srcX1 srcY1 srcX2 srcY2 [backgroundColor]`. *Description:* Draws a registered or Rexx-variable image at the current position, optionally scaled and cropped. *See:* Section 4.2.1.

image *Syntax:* `image [imageName]`. *Description:* Returns a detached `BufferedImage` representation of the current complete 2D rendering or a named image. *Return:* The requested `BufferedImage`. *See:* Section 4.2.1.

imageCopy *Syntax:* `imageCopy [imageName [backgroundColor]]`. *Description:* Creates a detached copy of the current or named image, optionally composited over a color. *Return:* The copied `BufferedImage`. *See:* Section 4.2.1.

imageSize *Syntax:* `imageSize [imageName]`. *Description:* Queries the current or named image dimensions. *Return:* `width height`. *See:* Section 4.2.1.

imageType *Syntax:* `imageType [imageName]`. *Description:* Queries remembered JDOR image-type metadata. *Return:* The numeric image type. *See:* Section 4.2.1.

loadImage *Syntax:* `loadImage imageName fileName`. *Description:* Loads and registers an image from a file. *Return:* `width height`. *See:* Section 4.2.1.

popImage *Syntax:* `popImage`. *Description:* Draws the most recently pushed image back onto the current Canvas. *Return:* The popped `BufferedImage`.

preferredImageType *Syntax:* `preferredImageType [type]`. *Description:* Queries or changes the preferred JDOR image type for subsequently created surfaces. *Return:* Query: current type; set: previous type.

printImage *Syntax:* `printImage [imageName]`. *Description:* Opens a JavaFX print job for the current or named image.

printPos *Syntax:* `printPos [x [y]]`. *Description:* Queries or changes the image's upper-left print position. *Return:* Query: `x y`; set: previous position.

printScale *Syntax:* `printScale [scaleX [scaleY]]`. *Description:* Queries or changes explicit image-print scaling. *Return:* Query: `scaleX scaleY`; set: previous scale.

printScaleToPage *Syntax:* `printScaleToPage [booleanValue]`. *Description:* Queries or changes automatic scaling to the printable page. *Return:* Query: current Boolean state; set: previous state.

pushImage *Syntax:* `pushImage [imageName]`. *Description:* Pushes a snapshot of the current complete 2D rendering onto the image stack, optionally registering the snapshot. *Return:* The pushed image.

saveImage *Syntax:* `saveImage fileName`. *Description:* Writes a snapshot of the current complete 2D rendering to an image file. *Return:* `1` after a successful write. *See:* Section 4.2.1.

A.6 2D Transformation Commands

moveTo *Syntax:* `moveTo [x [y]]`. *Description:* Queries or changes the current drawing position. *Return:* Query: `x y`; set: previous position.

rotate *Syntax:* `rotate angle [pivotX pivotY]`. *Description:* Appends a rotation to the Canvas transform, optionally around a pivot.

scale *Syntax:* `scale [scaleX [scaleY]]`. *Description:* Queries the current scale or appends a scale transform. *Return:* Query: `scaleX scaleY`; set: previous scale.

shear *Syntax:* `shear [shearX [shearY]]`. *Description:* Queries the current shear or appends a shear transform. *Return:* Query: `shearX shearY`; set: previous shear.

transform *Syntax:* `transform [RESET|name|tx ty sx sy shx shy|name tx ty sx sy shx shy]`. *Description:* Queries, resets, assigns, retrieves, or defines a JavaFX affine transform. *Return:* Current coefficients, previous coefficients, or a named `Affine`, according to the form.

translate *Syntax:* `translate [x [y]]`. *Description:* Queries the current translation or appends a translation. *Return:* Query: `x y`; set: previous translation.

A.7 3D Shape Commands

draw3DShape *Syntax:* `draw3DShape shapeName`. *Description:* Selects wireframe mode, adds a Shape3D to the Scene, and displays it. *Return:* The modified Shape3D.

fill3DShape *Syntax:* `fill3DShape shapeName`. *Description:* Selects filled mode, adds a Shape3D to the Scene, and displays it. *Return:* The modified Shape3D.

shape3d *Syntax:* `shape3d name [BOX x y z width height depth|CYLINDER x y z radius height|SPHERE x y z radius]`. *Description:* Queries or creates a registered JavaFX Shape3D without automatically displaying it. *Return:* The queried or created Shape3D.

A.8 Camera Commands

camera *Syntax:* `camera [name [PARALLEL x y z|PERSPECTIVE x y z [fieldOfView [fixedEyeAtCameraZero]]]]`. *Description:* Queries the active or named Camera, or creates a registered Camera. *Return:* The requested or created Camera; `.nil` for the implicit default.

cameraFarClip *Syntax:* `cameraFarClip cameraName [distance]`. *Description:* Queries or changes the Camera's far clipping distance. *Return:* Query: current distance; set: previous distance. *See:* Section 4.4.2.

cameraNearClip *Syntax:* `cameraNearClip cameraName [distance]`. *Description:* Queries or changes the Camera's near clipping distance. *Return:* Query: current distance; set: previous distance. *See:* Section 4.4.2.

rotateCamera *Syntax:* `rotateCamera cameraName angle pivotX pivotY pivotZ axisX axisY axisZ`. *Description:* Appends an axis-angle rotation to a registered Camera. *Return:* The modified Camera. *See:* Section 4.4.1.

setCamera *Syntax:* `setCamera [cameraName]`. *Description:* Selects a Camera, or restores JDORFX's default parallel Camera when no name is supplied. *Return:* The previously explicit Camera, or `.nil`.

translateCamera *Syntax:* `translateCamera cameraName tx ty tz`. *Description:* Appends a translation to a registered Camera. *Return:* The modified Camera. *See:* Section 4.4.1.

A.9 Light Commands

light *Syntax:* `light name [AMBIENT [color]|POINT x y z [color]]`. *Description:* Queries or creates a registered, initially disabled AmbientLight or PointLight. *Return:* The queried or created Light.

setLight *Syntax:* `setLight lightName [TURNON|TURNOFF]`. *Description:* Enables or disables a registered or Rexx-variable Light. *Return:* Its previous enabled state as `1` or `0`.

translatePointLight *Syntax:* `translatePointLight lightName tx ty tz`. *Description:* Appends a translation to a registered PointLight. *Return:* The modified PointLight. *See:* Section 4.4.3.

A.10 Material and Texture Commands

map *Syntax:* `map name imagePath [addWidth addHeight angle [color]]`. *Description:* Loads and registers a texture image, optionally enlarging and rotating it. *Return:* `width height`.

material *Syntax:* `material name [color]`. *Description:* Creates and registers a new PhongMaterial with an optional diffuse color. *Return:* The new PhongMaterial.

materialColor *Syntax:* `materialColor name DIFFUSE|DIFFUSECOLOR color | materialColor name SPECULAR|SPECULARCOLOR color [specularPower]`. *Description:* Changes a material's diffuse or specular color and, for the latter, its optional highlight concentration. *Return:* The previous color, or `.nil` when no specular color was set.

materialMap *Syntax:* `materialMap materialName BUMP|DIFFUSE|SELFILLUMINATION|SPECULAR map`. *Description:* Assigns an image to one PhongMaterial texture channel; the corresponding `*MAP` spellings are also accepted. *Return:* The previous map, or `.nil`.

setMaterial *Syntax:* `setMaterial shapeName materialName`. *Description:* Assigns a registered material to a Shape3D. *Return:* The Shape3D's previous material, or `.nil`.

A.11 3D Transformation Commands

rotate3DShape *Syntax:* `rotate3DShape name angle pivotX pivotY pivotZ axisX axisY axisZ`. *Description:* Appends an axis-angle rotation to a registered Shape3D. *Return:* The modified Shape3D.

`scale3DShape` *Syntax:* `scale3DShape name scaleX scaleY scaleZ pivotX pivotY pivotZ`. *Description:* Appends a three-dimensional scale around a pivot. *Return:* The modified Shape3D.

`shear3DShape` *Syntax:* `shear3DShape name shearX shearY pivotX pivotY`. *Description:* Appends an x/y-plane shear around a pivot. *Return:* The modified Shape3D.

`translate3DShape` *Syntax:* `translate3DShape name tx ty tz`. *Description:* Appends a translation to a registered Shape3D. *Return:* The modified Shape3D.

A.12 Cull Face Commands

`cullFace` *Syntax:* `cullFace shapeName [NONE|BACK|FRONT]`. *Description:* Queries or changes which oriented faces JavaFX omits when rendering a Shape3D. *Return:* Query: current mode; set: previous mode. *See:* Section 4.4.4.

A.13 Animation Commands

`animationFade` *Syntax:* `animationFade anim node duration [from [to [cycles [autoReverse]]]]`. *Description:* Creates and registers an opacity transition for a Node. *Return:* The new FadeTransition. *See:* Section 4.5.2.

`animationFill` *Syntax:* `animationFill anim shape duration [fromColor [toColor [cycles [autoReverse]]]]`. *Description:* Creates and registers a fill-color transition for a retained 2D Shape. *Return:* The new FillTransition.

`animationParallel` *Syntax:* `animationParallel anim [node] child1 child2 ... [cycles] [autoReverse]`. *Description:* Creates a group whose registered child animations run together. *Return:* The new ParallelTransition.

`animationPath` *Syntax:* `animationPath anim node duration path [cycles [autoReverse [orientation]]]`. *Description:* Creates a transition that moves a Node along registered 2D Shape geometry. *Return:* The new PathTransition.

`animationPause` *Syntax:* `animationPause anim`. *Description:* Pauses a registered animation at its current time. *Return:* Its previous status.

`animationPauseTransition` *Syntax:* `animationPauseTransition anim duration [cycles [autoReverse]]`. *Description:* Creates a timed pause for use alone or in a composed animation. *Return:* The new PauseTransition.

`animationPlay` *Syntax:* `animationPlay anim`. *Description:* Starts or resumes a registered animation. *Return:* Its previous status.

`animationRotate` *Syntax:* `animationRotate anim node duration [angle [cycles [autoReverse [axisX [axisY [axisZ]]]]]]`. *Description:* Creates a transition that rotates a Node during each cycle. *Return:* The new RotateTransition.

animationScale *Syntax:* `animationScale anim node duration [byX [byY [byZ [cycles [autoReverse]]]]]`. *Description:* Creates a transition that changes a Node's scale components. *Return:* The new ScaleTransition.

animationSequential *Syntax:* `animationSequential anim [node] child1 child2 ... [cycles] [autoReverse]`. *Description:* Creates a group whose registered child animations run in sequence. *Return:* The new SequentialTransition.

animationStatus *Syntax:* `animationStatus anim`. *Description:* Queries a registered animation without changing it. *Return:* `RUNNING`, `PAUSED`, or `STOPPED`.

animationStop *Syntax:* `animationStop anim`. *Description:* Stops a registered animation and resets it to the beginning. *Return:* Its previous status.

animationStroke *Syntax:* `animationStroke anim shape duration [fromColor [toColor [cycles [autoReverse]]]]`. *Description:* Creates and registers a stroke-color transition for a retained 2D Shape. *Return:* The new StrokeTransition.

animationTranslate *Syntax:* `animationTranslate anim node duration [byX [byY [byZ [cycles [autoReverse]]]]]`. *Description:* Creates a transition that changes a Node's translation components. *Return:* The new TranslateTransition.

keyframe *Syntax:* `keyframe timeline time keyValueNameOrArray`. *Description:* Adds a KeyFrame at a millisecond time to a registered Timeline. *Return:* The new KeyFrame.

keyValue *Syntax:* `keyValue name node property target [interpolator]`. *Description:* Creates a reusable target-property value for Timeline animation. *Return:* The new KeyValue.

setInterpolator *Syntax:* `setInterpolator anim LINEAR|DISCRETE|EASE_IN|EASE_OUT|EASE_BOTH|SPLINE(x1,y1,x2,y2)`. *Description:* Changes the progression rule of a supported Transition. *Return:* The previous Interpolator.

timeline *Syntax:* `timeline name [cycles [autoReverse]]`. *Description:* Creates or retrieves a Timeline and configures its cycle behavior. *Return:* The Timeline.

B Selected Complete Examples

This appendix contains six complete, runnable programs discussed in Chapter 4. They are selected from the submitted samples/ collection. The listings are generated directly from the corresponding source files using the RexxParser highlighter, so their printed line numbers match the complete programs and can be referenced from Chapter 4.

Apache 2.0 license headers are omitted from the printed listings for brevity; they remain present in the submitted source files.

B.1 Image Loading, Drawing, and Saving

Listing B.1 is generated from `samples/12_jdorfx_images.rxj`. Its saved PNG is shown in Figure 4.1.

```
#!/usr/bin/env rexx
/* Loading, drawing, querying, copying, and saving an image. */
call addJdorFXHandler
useRedirection? = arg()
if useRedirection? then
    address jdorfx with output using (.stdout) error using (.stderr)
else
    address jdorfx

-- newImage [width height [type [depthBuffer]]]
-- newImage width height PREFERRED depthBuffer
newImage 520 300
-- winTitle [title]
winTitle "JDORFX images"
-- background [colorName]
background WHITE
-- loadImage imageName fileName
loadImage LOGO "assets/oorexx_256.png"
-- imageSize [imageName]
imageSize LOGO
say "-- Source dimensions:" rc
```

```
-- imageType [imageName]
imageType LOGO
say "-- Source image type:" rc
-- preferredImageType [type]
preferredImageType
say "-- Preferred snapshot type:" rc

-- moveTo [x [y]]
moveTo 15 20
-- drawImage imageName [width height [sourceX1 sourceY1 sourceX2 sourceY2]]
    [backgroundColor]
drawImage LOGO 220 220
moveTo 265 20
drawImage LOGO 220 220 48 48 208 208 WHITE

-- assignRC records the returned copy for redirected command playback.
-- imageCopy [imageName [backgroundColor]]
imageCopy LOGO WHITE
-- assignRC variableName
assignRC logoCopy
imageSize "logoCopy"
say "-- Opaque copy dimensions:" rc
-- image [imageName]
image LOGO
say "-- Registered image:" rc

-- The extension determines the output format.
-- saveImage fileName
saveImage "12_jdorfx_images.png"
say "-- Saved current drawing to 12_jdorfx_images.png"
-- winShow
winShow
if useRedirection? then
    -- sleep seconds
    sleep 3
else
    pull .
-- winClose
winClose
exit 0

::requires "jdorfx.cls"
```

Listing B.1: Complete runnable JDORFX image-command demonstration

B.2 CullFace

Listing B.2 is generated from `samples/21_jdorfx_cullface.rxj`. The culling changes relevant to Figure 4.7 are on lines 72–93; the shared box, marker, lighting, and camera setup is on lines 36–68.

```
#!/usr/bin/env rexx
/* Observe the outside and inside of one box-like shell under face culling. */
call addJdorFXHandler
useRedirection? = arg()
if useRedirection? then
    address jdorfx with output using (.stdout) error using (.stderr)
else
    address jdorfx

-- newImage [width height [type [depthBuffer]]]
-- newImage width height PREFERRED depthBuffer
newImage 720 480 PREFERRED .true
-- background [colorName]
background DARKSLATEGRAY
-- shape3d shapeName box x y z width height depth
shape3d ROOM box 360 240 700 420 320 360
-- material materialName [color]
material ROOM_MAT DARKORANGE
-- setMaterial shapeName materialName
setMaterial ROOM ROOM_MAT
-- fill3DShape shapeName
fill3DShape ROOM

-- A small blue sphere identifies the interior and the nearby PointLight.
shape3d MARKER sphere 410 230 675 28
material MARKER_MAT DODGERBLUE
setMaterial MARKER MARKER_MAT
fill3DShape MARKER

-- The PointLight is inside the box; a dim ambient light reveals the exterior.
-- light lightName point x y z [color]
light ROOM_LIGHT point 420 170 650 WHITE
setLight ROOM_LIGHT
light FILL_LIGHT ambient DIMGRAY
setLight FILL_LIGHT

```

```
-- The outside camera faces the front wall.  The inside camera is just beyond
-- that wall and looks toward the illuminated marker and rear interior wall.
-- camera cameraName perspective x y z [fieldOfView [fixedEyeAtCameraZero]]
camera OUTSIDE perspective 360 240 0 48 .true
cameraNearClip OUTSIDE 1
cameraFarClip OUTSIDE 2000
camera INSIDE perspective 360 240 540 64 .true
cameraNearClip INSIDE 1
cameraFarClip INSIDE 2000
-- setCamera [cameraName]
setCamera OUTSIDE
-- winShow
winShow

-- cullFace shapeName [NONE|BACK|FRONT]
cullFace ROOM NONE
winTitle "CullFace NONE - outside baseline"
call waitForView useRedirection?

cullFace ROOM BACK
setCamera OUTSIDE
winTitle "CullFace BACK - outside"
call waitForView useRedirection?

setCamera INSIDE
winTitle "CullFace BACK - inside"
call waitForView useRedirection?

cullFace ROOM FRONT
setCamera OUTSIDE
winTitle "CullFace FRONT - outside"
call waitForView useRedirection?

setCamera INSIDE
winTitle "CullFace FRONT - inside"
call waitForView useRedirection?

-- winClose
winClose
exit 0

```

```
::routine waitForView
    use arg redirected
    if redirected then address jdorfx "sleep 2"
    else pull .
return

::requires "jdorfx.cls"
```

Listing B.2: Complete runnable JDORFX CullFace demonstration

B.3 Extended Camera Controls

Listing B.3 is generated from `samples/19_jdorfx_camera_extended.rxj`. Lines 50–65 configure and query the clipping interval. Line 73 translates the camera for Figure 4.6; line 75 then rotates it.

```
#!/usr/bin/env rexx
/* Camera translation, rotation, and clipping distances. */
call addJdorFXHandler
useRedirection? = arg()
if useRedirection? then
    address jdorfx with output using (.stdout) error using (.stderr)
else
    address jdorfx

-- newImage [width height [type [depthBuffer]]]
-- newImage width height PREFERRED depthBuffer
newImage 600 400 PREFERRED .true
-- winTitle [title]
winTitle "Extended camera controls"
-- background [colorName]
background LIGHTBLUE
-- shape3d shapeName box x y z width height depth
shape3d BOX box 300 200 700 180 180 180
-- material materialName [color]
material RED_MAT RED
-- setMaterial shapeName materialName
setMaterial BOX RED_MAT
-- fill3DShape shapeName
fill3DShape BOX
```

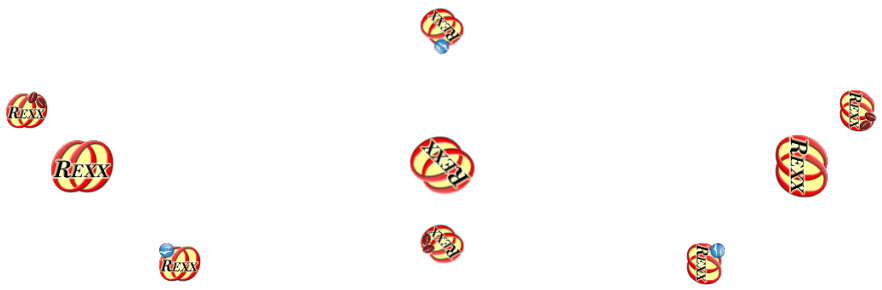
```
-- Keep the scene lighting fixed so only the camera changes between views.
light LAMP point 150 100 200 WHITE
setLight LAMP

-- camera cameraName perspective x y z [fieldOfView [fixedEyeAtCameraZero]]
camera VIEW perspective 0 0 0 45 .true
-- cameraNearClip cameraName [distance]
cameraNearClip VIEW 1
-- cameraFarClip cameraName [distance]
cameraFarClip VIEW 2000
-- setCamera [cameraName]
setCamera VIEW
-- The clipping distances are demonstrated through their set/query forms;
-- the two visible states below demonstrate camera transformation.
cameraNearClip VIEW
nearClip=rc~doubleValue
say "-- Near clipping distance:" nearClip
cameraFarClip VIEW
farClip=rc~doubleValue
say "-- Far clipping distance:" farClip
-- winShow
winTitle "Extended camera controls - initial view"
winShow
call waitForView useRedirection?

-- The camera, rather than the box, moves and turns.
-- translateCamera cameraName deltaX deltaY deltaZ
translateCamera VIEW 200 50 0
-- rotateCamera cameraName angle pivotX pivotY pivotZ axisX axisY axisZ
rotateCamera VIEW 10 300 200 0 0 1 0
winTitle "Extended camera controls - translated and rotated"
call waitForView useRedirection?
-- winClose
winClose
exit 0

::routine waitForView
    use arg redirected
    if redirected then address jdorfx "sleep 3"
    else pull .
return
```

```
87
88 ::requires "jdorfx.cls"
```

Listing B.3: Complete runnable JDORFX extended-camera demonstration

B.4 JDORFX Transition Comparison

Listing B.4 is generated from the file `2-130_JDORFX_animate_two_frames_transitions.rxj` in `samples/comparisons/jdor_vs_jdorfx_animation/`. Lines 47–52 replace the application-managed redraw loop with one named transition. Figure B.1 shows how the right-hand bitmap turns through one full revolution and a further quarter turn; the source bitmap in the left-hand window stays fixed.



(a) Start: 0° (b) Halfway: about 225° (c) End: 450°

Figure B.1: Three stages of the JDORFX 2-130 bitmap rotation over 3033 ms

```
1  #!/usr/bin/env rexx
6  call addJdorFXHandler "SOURCE"
7  call addJdorFXHandler "JDORFX"
8  useRedirection?=arg()
9  if useRedirection? then
10     address source with output using (.stdout) error using (.stderr)
11 else
12     address source
13
14 newImage 400 400
15 loadImage B4R "../../assets/bsf4oorexx_256.png"
16 moveTo 110 110
17 drawImage B4R 50 50
18 loadImage OOO "../../assets/oorexx4ooo_256.png"
```

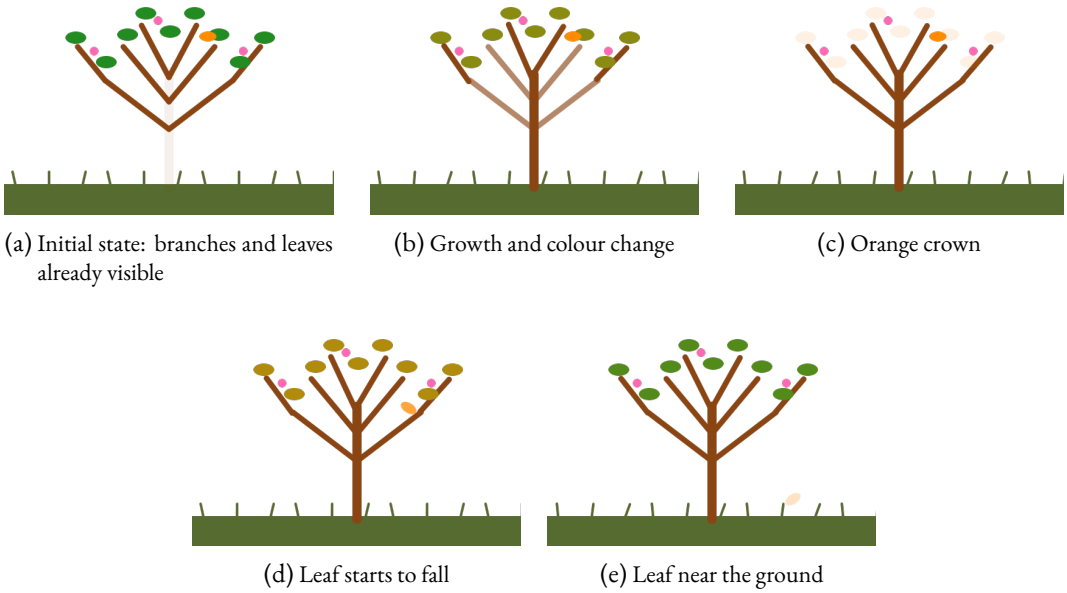
```
moveTo 290 290
drawImage OOO 50 50
loadImage ORX "../../assets/oorexx_256.png"
moveTo 162.5 162.5
drawImage ORX 75 75
image
sourceBitmap=rc
winTitle "2-130: original bitmap"
winShow
winLocation 0 0
winScreenSize
parse var rc screenWidth screenHeight

if useRedirection? then
    address jdorfx with output using (.stdout) error using (.stderr)
else
    address jdorfx
newImage 400 400
winUpdate .false
shape PICTURE Rectangle 0 0 400 400
fillShape PICTURE
getState
call bitmapFill rc~renderedShapes~get("PICTURE"),sourceBitmap
winUpdate .true
winFrame .false
winShow
winLocation (screenWidth-400) 0

-- The manual loop performs 91 five-degree steps and sleeps 1/30 s per step:
-- 450 degrees in approximately 3033 ms.
animationRotate TURN PICTURE 3033 450 1 .false
setInterpolator TURN LINEAR
say "-- Left: source bitmap. Right: a single JavaFX rotation animation."
animationPlay TURN
say "-- Rexx is waiting; JavaFX calculates and presents the intermediate angles."
if useRedirection? then
    sleep 5
else do
    say "-- Watch the rotation finish, then press Enter to close both windows."
    pull .
end
```

```
60 animationStop TURN
61 winClose
62 address source winClose
63
64 ::requires "bitmap_animation.cls"
```

Listing B.4: Complete runnable JDORFX 2-130 RotateTransition comparison

B.5 Oracle Tree Animation Case Study

Listing B.5 is generated from `samples/25_jdorfx_oracle_tree_animation.rxj`. The top-level composition discussed in Chapter 4 is on lines 138–187. Figure B.2 follows one playback: tree growth, repeating leaf-colour changes, and the delayed fall of a separate leaf occur alongside wind motion. Some branches and leaves are already visible in the first frame because the current commands cannot hide retained nodes before playback.



(a) Initial state: branches and leaves already visible

(b) Growth and colour change

(c) Orange crown

(d) Leaf starts to fall

(e) Leaf near the ground

Figure B.2: Five stages of the JDORFX Oracle Tree case study, captured during playback of the submitted program

```
1  #!/usr/bin/env rexx
23 /*
24   JDORFX case study based on Oracle JavaFX 8's Tree Animation Example.
25
```

```
   This is a conceptual reproduction, not a pixel-for-pixel port. It uses only
   public JDORFX commands: retained shapes, Timeline/KeyFrame/KeyValue,
   predefined transitions, and sequential/parallel composition.

   Run without arguments for an interactive viewing interval. Any argument
   enables canonical output and uses a fixed viewing interval.
*/
call addJdorFXHandler
useRedirection?=arg()
if useRedirection? then
    address jdorfx with output using (.stdout) error using (.stderr)
else
    address jdorfx

say "-- Oracle Tree Animation concepts reproduced with JDORFX"
newImage 900 600
winTitle "JDORFX - Oracle Tree Animation Case Study"
background LIGHTBLUE
winUpdate .false

-- Intentional JDORFX approximation: the current command interface cannot set
-- retained-node opacity or visibility synchronously before playback. Shapes
-- used in later phases can therefore be visible initially, unlike in the Java
-- counterpart. The leaf-only SPRING group does not delay the flowers, and the
-- falling leaf remains visible during FALL_DELAY before its movement and fade.
-- No low-level Java access is used to conceal these differences.

-- Ground and a compact set of individual grass blades.
color DARKOLIVEGREEN
shape GROUND rectangle 0 510 900 90
fillShape GROUND
stroke GRASS_STROKE 4 CAP_ROUND JOIN_ROUND
do blade=1 to 15
    x=45+blade*52
    tip=x+((blade//3)-1)*8
    grassName="GRASS_"blade
    shape (grassName) line (x) 525 (tip) 490
    drawShape (grassName)
end

-- A recognizable tree assembled from retained branch nodes.
```

```
color SADDLEBROWN
stroke TRUNK_STROKE 15 CAP_ROUND JOIN_ROUND
shape TRUNK line 450 515 450 330
drawShape TRUNK

stroke BRANCH_STROKE 9 CAP_ROUND JOIN_ROUND
branchData=.array~of(,
    "LOW_LEFT 450 420 345 340", "LOW_RIGHT 450 420 555 340",,
    "MID_LEFT 450 375 375 285", "MID_RIGHT 450 375 525 285",,
    "TOP_LEFT 450 335 405 250", "TOP_RIGHT 450 335 495 250",,
    "CROWN_LEFT 345 340 300 285", "CROWN_RIGHT 555 340 600 285")
do branchSpec over branchData
    parse var branchSpec branchName x1 y1 x2 y2
    shape (branchName) line (x1) (y1) (x2) (y2)
    drawShape (branchName)
end

-- Leaves and flowers use named retained Shapes so they can be animated.
color FORESTGREEN
leafData=.array~of(,
    "LEAF_1 280 260", "LEAF_2 330 300", "LEAF_3 365 255",,
    "LEAF_4 395 220", "LEAF_5 435 250", "LEAF_6 475 220",,
    "LEAF_7 515 255", "LEAF_8 550 300", "LEAF_9 590 260")
do leafSpec over leafData
    parse var leafSpec leafName lx ly
    shape (leafName) ellipse (lx) (ly) 34 20
    fillShape (leafName)
end

color HOTPINK
flowerData=.array~of("FLOWER_1 320 285", "FLOWER_2 425 235", "FLOWER_3 565 285")
do flowerSpec over flowerData
    parse var flowerSpec flowerName fx fy
    shape (flowerName) ellipse (fx) (fy) 14 14
    fillShape (flowerName)
end

-- A separate leaf represents Oracle's falling copies of leaves and petals.
color DARKORANGE
shape FALLING_LEAF ellipse 500 260 28 16
fillShape FALLING_LEAF
```

```
108
109 -- Each generation is a Timeline. Explicit 1 ms keyframes hide the generation;
110 -- later keyframes reveal it. The sequence expresses generation-by-generation
111 -- growth without an ooRexx frame-update loop.
112 timeline GROW_TRUNK 1 .false
113 keyValue TRUNK_HIDDEN TRUNK opacity 0 DISCRETE
114 keyframe GROW_TRUNK 1 TRUNK_HIDDEN
115 keyValue TRUNK_VISIBLE TRUNK opacity 1 EASE_OUT
116 keyframe GROW_TRUNK 1400 TRUNK_VISIBLE
117
118 timeline GROW_LOW 1 .false
119 do name over .array~of("LOW_LEFT", "LOW_RIGHT", "MID_LEFT", "MID_RIGHT")
120     hidden=name"_HIDDEN"
121     visible=name"_VISIBLE"
122     keyValue (hidden) (name) opacity 0 DISCRETE
123     keyframe GROW_LOW 1 (hidden)
124     keyValue (visible) (name) opacity 1 EASE_OUT
125     keyframe GROW_LOW 1300 (visible)
126 end
127
128 timeline GROW_HIGH 1 .false
129 do name over .array~of("TOP_LEFT", "TOP_RIGHT", "CROWN_LEFT", "CROWN_RIGHT")
130     hidden=name"_HIDDEN"
131     visible=name"_VISIBLE"
132     keyValue (hidden) (name) opacity 0 DISCRETE
133     keyframe GROW_HIGH 1 (hidden)
134     keyValue (visible) (name) opacity 1 EASE_OUT
135     keyframe GROW_HIGH 1300 (visible)
136 end
137
138 -- Spring appearance is parallel; the complete growth is sequential.
139 do leafSpec over leafData
140     parse var leafSpec leafName .
141     animationFade ("APPEAR_"leafName) (leafName) 1100 0 1
142 end
143 animationParallel SPRING APPEAR_LEAF_1 APPEAR_LEAF_2 APPEAR_LEAF_3 ,
144     APPEAR_LEAF_4 APPEAR_LEAF_5 APPEAR_LEAF_6 APPEAR_LEAF_7 ,
145     APPEAR_LEAF_8 APPEAR_LEAF_9
146 animationSequential GROW_TREE GROW_TRUNK GROW_LOW GROW_HIGH SPRING
147
148 -- Repeating crown and grass motion approximates wind. JDORFX rotates each
```

```
-- retained node around its own centre; Oracle binds branch-local transforms.
do name over .array~of("TOP_LEFT", "TOP_RIGHT", "CROWN_LEFT", "CROWN_RIGHT")
    animationRotate ("WIND_"name) (name) 1800 5 "-1" .true
    setInterpolator ("WIND_"name) EASE_BOTH
end
do blade=1 to 15
    grassName="GRASS_"blade
    animationRotate ("WIND_"grassName) (grassName) 1500 8 "-1" .true
end
animationParallel TREE_WIND WIND_TOP_LEFT WIND_TOP_RIGHT ,
    WIND_CROWN_LEFT WIND_CROWN_RIGHT
animationParallel GRASS_WIND_A WIND_GRASS_1 WIND_GRASS_2 ,
    WIND_GRASS_3 WIND_GRASS_4 ,
    WIND_GRASS_5 WIND_GRASS_6 WIND_GRASS_7 WIND_GRASS_8
animationParallel GRASS_WIND_B WIND_GRASS_9 WIND_GRASS_10 WIND_GRASS_11 ,
    WIND_GRASS_12 WIND_GRASS_13 WIND_GRASS_14 WIND_GRASS_15
animationParallel GRASS_WIND GRASS_WIND_A GRASS_WIND_B
animationParallel WIND TREE_WIND GRASS_WIND

-- Fill transitions provide a repeatable green/autumn/green season cycle.
do leafSpec over leafData
    parse var leafSpec leafName .
    animationFill ("SEASON_"leafName) (leafName) 4500 ,
        FORESTGREEN DARKORANGE 2 .true
end
animationParallel SEASONS SEASON_LEAF_1 SEASON_LEAF_2 SEASON_LEAF_3 ,
    SEASON_LEAF_4 SEASON_LEAF_5 SEASON_LEAF_6 SEASON_LEAF_7 ,
    SEASON_LEAF_8 SEASON_LEAF_9 "-1" .false

-- A delayed leaf fall combines translation and fading in parallel.
animationPauseTransition FALL_DELAY 5200
animationTranslate LEAF_DESCENT FALLING_LEAF 3600 90 280 0
animationFade LEAF_DISAPPEAR FALLING_LEAF 3600 1 0
animationRotate LEAF_TURN FALLING_LEAF 1800 360 2 .false
animationParallel FALL_MOTION LEAF_DESCENT LEAF_DISAPPEAR LEAF_TURN
animationSequential FALL_SEQUENCE FALL_DELAY FALL_MOTION

-- Oracle likewise combines wind with a sequential growth/season program.
animationParallel ALL GROW_TREE WIND SEASONS FALL_SEQUENCE
animationPlay ALL
winUpdate .true
```

```
winShow

say "-- Growth, parallel wind, season colour, and one falling leaf are active."
if useRedirection? then
    sleep 14
else do
    say "-- Press Enter to stop the case study."
    pull .
end
animationStop ALL
winClose
exit 0

::requires "jdorfx.cls"
```

Listing B.5: Complete runnable JDORFX Oracle Tree Animation case study

B.6 Canonical Recording and Replay

Listing B.6 is generated from `samples/26_jdorfx_macro_record_replay.rxj`. It is a compact complete example of both sides of the workflow in Figure 4.11.

```
#!/usr/bin/env rexx
/* Record canonical JDORFX output in a file, then replay that file as input. */
call addJdorFXHandler "RECORD"
call addJdorFXHandler "REPLAY"

macroFile="26_recorded_commands.jdorfx"
recording=.stream~new(macroFile)~~open("write replace")
address record with output using (recording) error using (.stderr)
newImage 320 180
background WHITE
color STEELBLUE
shape PANEL rectangle 40 35 240 110
fillShape PANEL
winShow
winClose
recording~close

commands=.stream~new(macroFile)~~open("read")
address replay with input using (commands) error using (.stderr)
"-- trigger redirected-input processing"
commands~close
call SysSleep 1
address replay winClose

say "Recorded and replayed" macroFile
exit 0

::requires "jdorfx.cls"
```

Listing B.6: Complete runnable JDORFX canonical recording and replay example

List of Tools and Aids Used

Title of the thesis:
Extending JDORFX: A JavaFX Graphics Framework for ooRexx
Improving JDOR Compatibility, Extending 3D Functionality, and Adding Animation Support

Author:
Alireza Ismaili, 12035862

Tool / Aid Used	**Type of Use**	**Affected Sections / Chapters**	**Documentation**
ChatGPT (OpenAI)	Support with understanding Java, JavaFX, ooRexx, and JDORFX concepts; discussion of implementation approaches; support with structuring, reviewing, and language revision of the thesis text.	Throughout the thesis and project work	Chat histories
OpenAI Codex	Support with implementation, revision, and review of Java and ooRexx code, sample programs, documentation, and thesis source files.	Across the thesis and project work	Codex session history and Git diffs
diagrams.net (drawio)	Creation and editing of diagrams and figures.	Sections containing diagrams created with the tool	Source `.drawio` files
Visual Studio Code (VS Code)	Editing and reviewing Java, ooRexx, documentation, and LaTeX source files.	Thesis and project source files	Source files
Git	Version control and review of changes to code, samples, and thesis sources.	Throughout the project	Git history and diffs
LuaLaTeX, BibTeX, and latexmk	Typesetting the thesis, generating the bibliography, and managing repeat builds.	Entire thesis	LaTeX sources and build configuration

Tool / Aid Used	**Type of Use**	**Affected Sections / Chapters**	**Documentation**
RexxParser	Syntax highlighting of ooRexx source listings included in the thesis.	Chapter 4 and Appendix B	Generated listings and listing-generation script
Java Development Kit (JDK), JavaFX, ooRexx, and BSF4ooRexx	Building, running, and testing the JDORFX handler and its sample programs.	Implementation and evaluation chapters	Java and ooRexx sources, build and test scripts

I hereby confirm that I have listed all tools and aids used in this work in the above table. If no tools or aids were used, this must be stated by entering “none” under “Tool / Aid Used”.

11 October 2026



Date

Signature