QR Code Tactical Decision Aid

Instructions

Mike Bailey

Moves Institute

Naval Postgraduate School

6 July 2015

# Introduction

The QR Code Tactical Decision Aid, or QRTDA, is a software package to allow experimentation with QR images as a medium for optical data communication.

# Software Architecture

QRTDA performs two basic functions:

1. Capturing a QR image, typically from a camera attached to the running computer, decoding its content, which is typically in the form of a string of text, and displaying or saving that content.
2. Taking text input (from keyboard or file), creating a QR image which encodes the text, and displaying or saving that image.

(The software has been expanded and now has several uses which have nothing to do with QR images. For instance, it can be used to accept keyboard text entry, then display an encoding in either signal flags, semaphore or morse code.)

Surrounding and amid the two basic functions are special tasks to provide other optional services, such as text encryption/decryption, image insertion, intermediate file saving, etc. To accommodate such varying task lists in a flexible way, a QRTDA execution is built as one or more “data flows”, each one consisting of a directed graph of data moving between operations. In the case of QRTDA, the input data items are typically either captured camera images, existing QR images, or text strings. The output data items from a dataflow are typically one of the same types.

To provide the software framework for this design, QRTDA makes use of an open source library called “TinkerPop” (<https://github.com/tinkerpop/pipes/wiki>). This library provides the “plumbing” for constructing the data flows and moving data through them.

QR image encoding and decoding is done through a library from Google called “Zxing” (from “Zebra Crossing”) at <https://github.com/zxing/zxing/>. This library may also be used to handle other image types, such as UPC bar codes.

Other libraries:

* Webcam capture -- <https://github.com/zxing/zxing/>
* text en/decryption -- <http://jasypt.org/>.

# Required System Software

QRTDA is written in Java and is portable to systems which support a Java Standard Edition Runtime Environment, or JRE (SE), version 8 or better. The JRE enables the execution of QRTDA. To work with QRTDA as a developer, the development system must also support a Java Development Kit, or JDK (SE). Recent versions of MacOSX, Windows, Linux and other Unixes fulfill this requirement. QRTDA developers use an interactive development environment (IDE), Netbeans.

# Required Hardware

Any system supporting a JRE referenced above may be used. The most common usage of QRTDA will be capturing and decoding QR images. To do this, an attached camera will be required. The built-in, video chat cameras on current laptops may be used for testing. QRTDA queries the webcam capture library for a list of attached cameras. Questions regarding camera support should be directed to the above web page.

QRTDA may be used in a non-gui mode (“headless”), so it can be configured to support back-end server data flows without operator intervention.

# Getting the Software

# Using QRTDA

## Workspace location

The QRTDA application is deployed as an executable Java “jar” file. The jar file may exist anywhere on the system (logically in an “application” directory for a non-development system) and be run from that location. A per-user file area is required and is automatically created on first launch.

On a Mac, the default workspace location is ~/QrTdaWorkspace. (I.e., in the user’s home directory). On a Windows machine, the default workspace location is C:/Users/<username>/QrTdaWorkspace.

The workspace holds run scripts, configuration files, and is a default location for image and text file creation.

## Elements

QRTDA execution is based on a data flow model, where data moves from component to component and each component performs some operation or transformation on the object. For instance, a data flow which reads a QR image from an attached camera may consist of elements which 1) extract a frame from the camera, 2) analyze it to see if it contains a QR image, 3) decode the image into text, 4) display the text in a GUI, and 5) pass the text to a “chat” application component. In addition, intermingled in the data flow among those 5 elements may be elements whose function is solely to convert incoming data types to types require by subsequent elements.

Each element expects an input data object of a particular type and will emit an object of a particular type. When elements are “stacked” into a data flow, an element’s output type must match the input type of the following element. To help constructing data flows, the following table can help. It shows all the elements in the QRTDA software distribution organized by type and displaying input and output data types. “Sources” are typically at the beginning of a data flow and “Sinks” at the end.

### 

## JSON Configuration File

The file format for describing QRTDA dataflows is JSON. One json file can describe several data flows, and each flow will display a separate GUI window. For instance, in a “chat” application, which is a two-way communication setup, one data flow will accept text entry by the user, encode as a QR image, and project the image, while a second will capture a QR frame from a camera, decode it, and display the decoded text. Therefore, two data flows must be defined for this application. An example chat application configuation file follows.

{

"name": "QR Chat",

"description": "Shows 2 windows, receive and send, to implement chat with similar remote configuration",

"version": 1.0,

"headless": false,

"pipeLines": [{

"name": "QR Chat Image Decoder",

"description": "Reads camera frame, displays in gui, decodes QR, displays text, if not duplicate, display text in chat panel",

"version": 1.0,

"pipeElements": [{

"className": "edu.nps.moves.qrtda.elements.**QRCameraFrameGrabber**",

"option": [{

"key": "frameSleep",

"value": 1000

}]

}, {

"className": "edu.nps.moves.qrtda.elements.**QRImagePanelDisplayer**",

"option": []

}, {

"className": "edu.nps.moves.qrtda.elements.**QRImageDecoder**",

"option": []

}, {

"className": "edu.nps.moves.qrtda.elements.**QRDecodedTextDisplayer**",

"option": []

}, {

"className": "edu.nps.moves.qrtda.elements.**DuplicateRejector**",

"option": []

}, {

"className": "edu.nps.moves.qrtda.elements.**ChatReceive**",

"option": []

}]

}, {

"name": "QR Chat Text Encoder",

"description": "Reads chat panel text input, generates QR, optionally overlays an image, shows locally, optionally saves to file, sends to ImageProjector",

"version": 1.0,

"pipeElements": [{

"className": "edu.nps.moves.qrtda.elements.**ChatSend**",

"option": []

}, {

"className": "edu.nps.moves.qrtda.elements.**QRInputBuilder**",

"option": []

}, {

"className": "edu.nps.moves.qrtda.elements.**QRGenerator**",

"option": [{

"key": "errorCorrectionLevel",

"value": "H"

}, {

"key": "pixelSize",

"value": 550

}, {

"key": "pixelMargin",

"value": 4

}]

}, {

"className": "edu.nps.moves.qrtda.elements.**QRBufferedImageGenerator**",

"option": []

}, {

"className": "edu.nps.moves.qrtda.elements.**QRImageOverlayer**",

"option": [{

"key": "imageOverlayActive",

"value": false

}, {

"key": "imageOverlayFilePath",

"value": ""

}, {

"key": "imageOverlaySizeRatio",

"value": 0.25

}, {

"key": "imageOverlayTransparency",

"value": 1.0

}]

}, {

"className": "edu.nps.moves.qrtda.elements.**QRImagePanelDisplayer**",

"option": []

}, {

"className": "edu.nps.moves.qrtda.elements.**QRImageSaveToFile**",

"option": [{

"key": "imageFormat",

"value": "png"

}, {

"key": "directoryPath",

"value": "imageDirectory"

}, {

"key": "fileNamePrefix",

"value": "QRImageExport"

}]

}, {

"className": "edu.nps.moves.qrtda.elements.**ImageProjector**",

"option": []

}]

}]

}

Element names are shown in bold type, and, more correctly, are Java class names, corresponding to separate Java code files.

### Options

Each element may support one or more controlling parameters, which are very often exposed through the element’s GUI. The configuration file above lists element options as key value pairs, where the key is the option name, and the value is its default value. For example, the QRImageSaveToFile element specified above supports three options: imageFormat, directoryPath and fileNamePrefix.

QRTDA options, when changed through the GUI, are persistent, in that their values remain fixed between executions of QRTDA from one user account on a computer. Options that are specified on the command line (see below) are not persistent.

#### Run scripts

On some systems, QRTDA may be launched by a double-click of the jar file. However, most often it is desired to launch a particular configuration, as defined by a json file. Simple run scripts can be written to accomplish this.

A java program is always run by first executing the Java virtual machine, using the **java** command. The main argument passed to the java command will be the name of the jar containing the application’s code. Further arguments are specified by the application code itself.

Here’s an example run script for MacOSX or Unix to launch a “bounce” application, which receives a QR images, decodes it, recodes it and displays the resulting QR image.

#!/bin/bash

java -jar /Applications/QRTDA/QRTda.jar qrBounce.json

The first line defines the enclosing file as a Bash shell script. (Bash scripts must normally be marked with the “executable” attribute in a file system.) The second line runs the Java virtual machine, which is always on the user’s path when Java is installed on a system. “java” runs the jvm. “-jar /Applications/QRTDA/QRTda.jar” tells the jvm that the application code exists in the jar file specified. The final parameter, “qrBounce.json”, tells the QRTDA code which configuration file to used.

If no absolute location is given for the configuration file, as is the case here, QRTDA will look into the “configurations” subdirectory of the workspace directory to find the file.

#### Command Line Operation

If you run QRTDA and specify “-help” as the single parameter, you get the following summary:

usage: java -jar QRTda.jar [dataflowconfigurationfile] [-ACTION] [-options...]

------------------------------------------------------------------------------

ACTIONS:

-cameradecodeloop display decoded text from QR, loop

-camerasaveqrimage <file> save captured image

-camerasaveqrimageloop save captured images continually into images directory

-cameratextdisplay open a text-receive window

-cameratextdisplayecho open a text-receive window and echo received text

-cameratextfileappend append received text to file

-cameratextfileappendecho append received text to file and echo received text

-chat open a chat window

-createqrimage <file> create a single named QR into the specified file

-last run last configuration script

OPTIONS:

-delay <msec> pipeline delay

-encryptionpassword <string> encryption password

-externaldisplayactive <true/false> external display active

-externaldisplayid <externdisplayid> external display ID

-framegrabinterval <interval> frame capture interval (ms)

-help display program command-line options

-imageoverlayactive <true/false> do overlaying

-imageoverlayfilepath <file> overlay image

-imageoverlaysizeratio <size> overlay image proportional size (0-1.0)

-imageoverlaytransparency <transparency> overlay image transparency (0-1.0)

-imagewatchdirectory <directory> directory to watch for images

-legendenabled <true/false> append legend

-legendsizepercentage <size> legend proportional size (0-1.0)

-qrdirectorypath <directory> image save directory

-qrerrorcorrectionLevel <level> error correctionlevel (H, Q, M or L)

-qrimagefilenameprefix <prefix> image save name prefix

-qrimageformat <format> image file format

-qrimagesize <size> image size in pixels

-qrmargin <size> image margin in squares

-qroverwriteimage <overwrite> overwrite existing file

-qrsaveenabled <true/false> save images

-textinput <text> text to encode

-textwatchdirectory <directory> directory to watch for text files

-webcamname <name> webcam name

------------------------------------------------------------------------------

Running with command line options instead of a configuration file runs QRTDA

# Developing QRTDA

## Netbeans Setup

The first step to begin QRTDA development is to acquire the QRTDA source tree from the source code repository and open its root directory as a project in Netbeans.

…

## New Element Development

Development of a new dataflow element (which typically accepts a data item, performs some operation using it, then emits it or another item) consists of writing a single class and an optional GUI class, if GUI control of the element operation is appropriate.

An element is identified in a JSON configuration file by class name. It is written to extend the abstract class, edu.nps.moves.qrtda.elements.QRFlowLink. The default package for QRTDA elements is edu.nps.moves.qrtda.elements , although a configuration file may refer to elements in any Java package. The default package for element GUI classes is edu.nps.moves.qrtda.elements.gui.

The QRFlowLink class is a “generic type”, and each class extending it must do so by defining two parameter types. These types refer to the data type which is input to the element and the data type which is output from the element.

### Element Constructor

A snippet from a sample QRTD element is shown below. Notice that the class statement includes the two generic types which correspond to input data and output data types.



### Element Annotations

Elements must provide 4 pieces of identifying data: handle, category, long description and short description. They do so through annotating corresponding fields with the appropriate annotation. The handle annotation is also passed to the super class in the constructor. The other three fields must also be returned through appropriate getters – see the last several lines. The category field must be one of the statically-defined fields in the PipeCategory class.

### Element Data Handling

The data processing work in an element is initiated through the handleNextData() method. The input data is read from the protected dataQueue() method in the super class, and the output is returned from this method. Note that a compile error will be generated if 1) the data type from the dataQueue() method is not handled correctly (as a String in this example); 2) the handleNextData() method does not return the correct type (BufferedImage in this example).

QRTDA provides a configuration utility function (in development). One convenient feature is a listing of all elements which are found in the default element package in the application jar, sorted by type, and displaying the input and output datatypes of the element. The current build of QRTDA contains the elements listed in the following screen capture.

Source elements, which exist at the start of a data flow are listed under the blue “TEXT\_SOURCE” and “IMAGE\_SOURCE” headings. Sink elements, which are placed at the end of a data flow are listed under the red “TEXT\_SINK” and “IMAGE\_SINK. Source and sink elements are still written to specify output data types, and normally output the same object received. They are classified as sinks because of their normal use. Similarly, source elements are written as accepting an input object, even though they are at the beginning of the chain. This is an implementation curiousity, and the source element itself “feeds” its data queue as if receiving the data object from a previous element.

