Jogl and Java 3D - The State of Java Graphics Libraries

Douglas Lyon and Predrag Bokšić

ABSTRACT

We are faced with an aging API in Jogl and Java 3D graphics libraries that make use of the OpenGL and offer a range of 3D functions. Various Java classes have been significantly modified or are considered “deprecated” as of the JDK version 9, with consequences for these graphics libraries. [Oracle]

Older Java 3D installs abound (e.g. the version 1.5) and their prevalence makes the configuration and deployment a nightmare. Our system for the deployment of Jogl-based Java 3D libraries enables the transparent click and go applications that make use of the high-performance graphics that "just works".

Deployment Methodology

1. Problem Statement

We are given a desktop platform with Java 8 or Java 9 installed and seek to find a way to deploy high-performance Java applications that use Java 3D and/or Jogl without having to run an installer. We are subject to the constraint that the applications be signed and deployed so that they can be run in a trusted environment (i.e., outside of the sandbox). Further, we seek to do this in a way that does not depend on bundling a JRE with our applications, as this makes downloads and installations rather long.

The novel javapackager command (included in the JDK versions 8 and 9), produces the program installation packages in the range of sizes from 57 to 202 Megabytes approximately.

2. Motivation

We have a long history of making use of Java 3D applications in the software engineering community. Since the introduction of Java 3D in 1998, a number of open-source projects have

sprung up to help preserve it (including Jogl). These projects were needed because the Sun Corporation's development of Java 3D had discontinued in 2004. At that point, it was released as a community-driven open source project. By 2008, Java 3D was put on hold in favor of the JavaFX framework with 3D support. JavaFX contains many proprietary software components whose code has not been released to the public. Moreover, switching from Java 3D to JavaFX is generally accepted as non-trivial.

There are now approximately 10 million Java developers. ("The 2009 Global Developer Population and Demographics Survey, conducted by Evans Data Corporation, reports the Java-developer population equals 9,007,346.") If just 1% are using Java 3D and must rewrite their code as a result, that is 100,000 impacted developers with programs of unknown sizes.

I possess 493 java files in my 3D project with 42,936 lines of code for an average of just over 87 lines per file. If each of the Java 3D developer's code exhibits just 1% of my project's file size, this implies the 429 lines of code times 100,000 developers or almost 43 million lines of code. If it takes just $15 to write a line of code, (some estimates are as high as $40) the costs are between the $645 million and the $1.7 billion dollars [Koopman].

3. Approach

```
Our approach to the deployment of the new Java 3D libraries is to create
a single extension to the JNLP resources, called the "jogl.jnlp". This
deployment strategy makes a direct reference to the bundled native
libraries as well as the Java 3D jars. The listing of the "jogl.jnlp"
file follows:

<jnlp spec="1.0+"
codebase="http://show.docjava.com:80/book/cgij/code/jnlp/">
<information>
    <title>Java 3D</title>
    <vendor>DocJava Inc.</vendor>
    <homepage href="http://www.docjava.com"/>
    <description>Java 3D library</description>
</information>
<security>
    <all-permissions/>
</security>
<resources>
    <jar href="libs/jogl/j3dcore.jar" download="eager"/>
    <jar href="libs/jogl/j3dutils.jar" download="eager"/>
    <jar href="libs/jogl/vecmath.jar" download="eager"/>
    <jar href="libs/jogl/jogamp-fat.jar" download="eager"/>
</resources>
<component-desc/>
</jnlp>
```

The j3dcore.jar, j3dutils.jar, and vecmath.jar are pure Java code. They compose the Java 3D library. The required native methods (JNI) for the standard platforms (Windows 64, Windows

32, Linux 64, Linux 32, and the Mac) are all found inside the jogamp-fat.jar library. A JNLP file that launches a Jogl application follows:

```
    <jnlp spec="1.0+"
    codebase="http://show.docjava.com:80/book/cgij/code/jnlp/">
    <information>
        <title>j3d.JOGLQuad</title>
        <vendor>DocJava, Inc.</vendor>
        <homepage href="http://www.docjava.com"/>
        <icon href="http://show.docjava.com:80/consulti/docjava.jpe"/>
        <offline-allowed/>
    </information>
    <security>
        <all-permissions/>
    </security>
    <resources>
    <j2se version="1.8+" initial-heap-size="64m" max-heap-size="384m" java-
    vm-args="
            --add-exports=java.base/java.lang=ALL-UNNAMED
            --add-exports=java.desktop/sun.awt=ALL-UNNAMED
            --add-exports=java.desktop/sun.java2d=ALL-UNNAMED"/>
        <extension name="Java3D" href="jogl.jnlp"></extension>
        <jar href="joglquad.jar"/>
    </resources>
    <application-desc main-class="j3d.JOGLQuad"/>
</jnlp>
```

Here we present the common extension for Jogl support, <extension name="Java3D" href="jogl.jnlp"></extension>. This extension enables the common deployment strategy to all Java 3D and Jogl based programs. Furthermore, critical to the proper operation is the j2se tag.

<j2se version="1.8+" initial-heap-size="64m" max-heap-size="384m" java-vm-args="--add-exports=java.base/java.lang=ALL-UNNAMED

```
      --add-exports=java.desktop/sun.awt=ALL-UNNAMED
      --add-exports=java.desktop/sun.java2d=ALL-UNNAMED"/>

    The "--add-exports" is a part of the new-style module arguments. For
    example:

        --add-exports=java.base/java.lang=ALL-UNNAMED
```

follows the pattern:

```
    --add-exports <source-module>/<package>=<target-module>
```

where <source-module> is "java.base", <package> is "java.lang" and <target-module> is "ALL-UNNAMED". The "ALL-UNNAMED" enables all modules to access all classes in java.lang. In the case of legacy code, all the modules will be unnamed.

The extension loading may be perceived as a separate program that the user needs to allow to run

immediately after the first part of the program has been deployed to the user's platform.

As WebStart applications must be signed before the modern Java will run them, I have authored a collection of articles that guide the reader through the signing process [Lyon, 2004a, b, 2008].

The signed jar files are all available from tinyurl.com/signedJars.

A deprecation warning has been announced for the JNLP deployment in the future release of JDK. “Java Applet and WebStart functionality, including the Applet API, The Java plug-in, the Java Applet Viewer, JNLP and Java Web Start including the javaws tool are all deprecated in JDK 9 and will be removed in a future release.” [http://www.oracle.com/technetwork/java/javase/9-deprecated-features-3745636.html]

The open source WebStart replacement IcedTea offers further options to run the JNLP files and the application itself in the sandbox even if the application is signed with a trusted certificate. IcedTea offers the optional safety limitations of the application’s functionality. The following two pop-up windows captured in the OpenSuSE Linux demonstrate this aspect of the user’s experience.

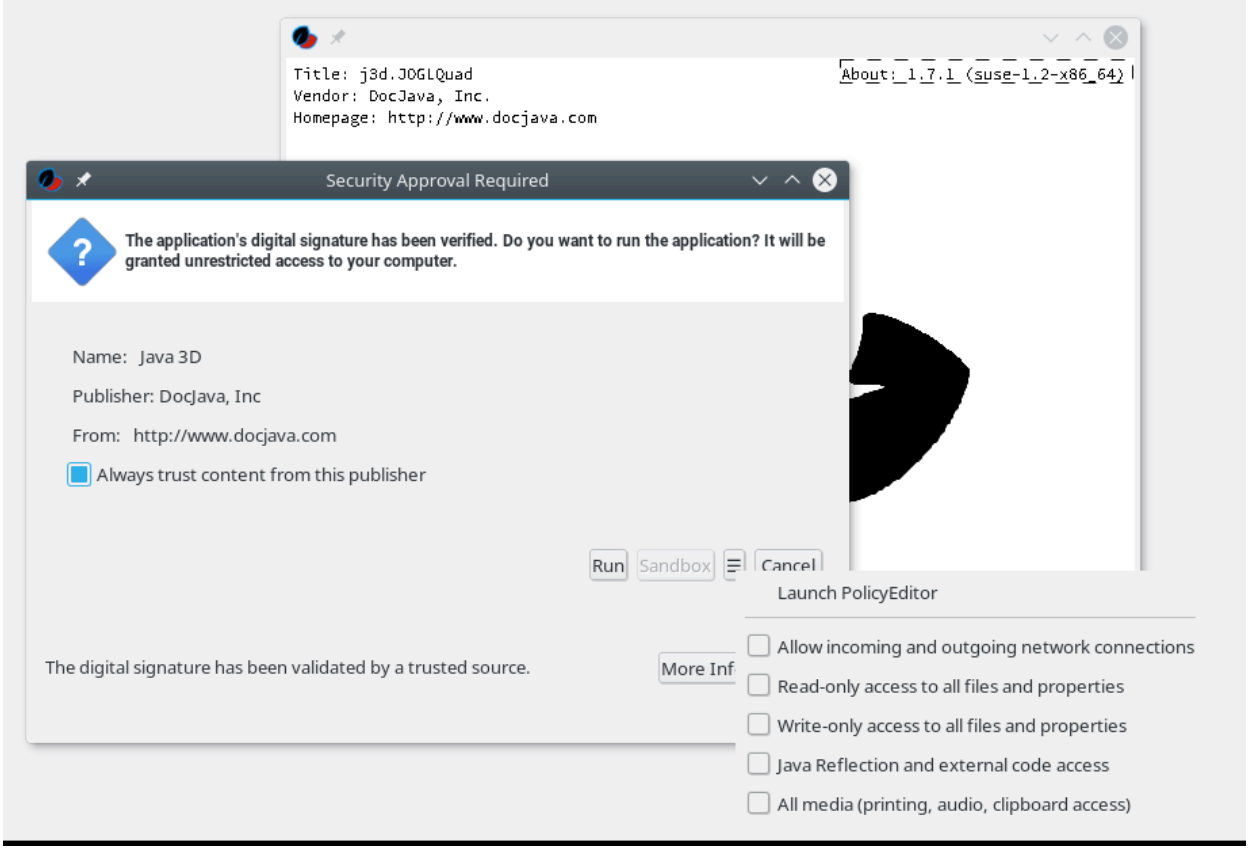

Figure 1. Security Approval Dialog

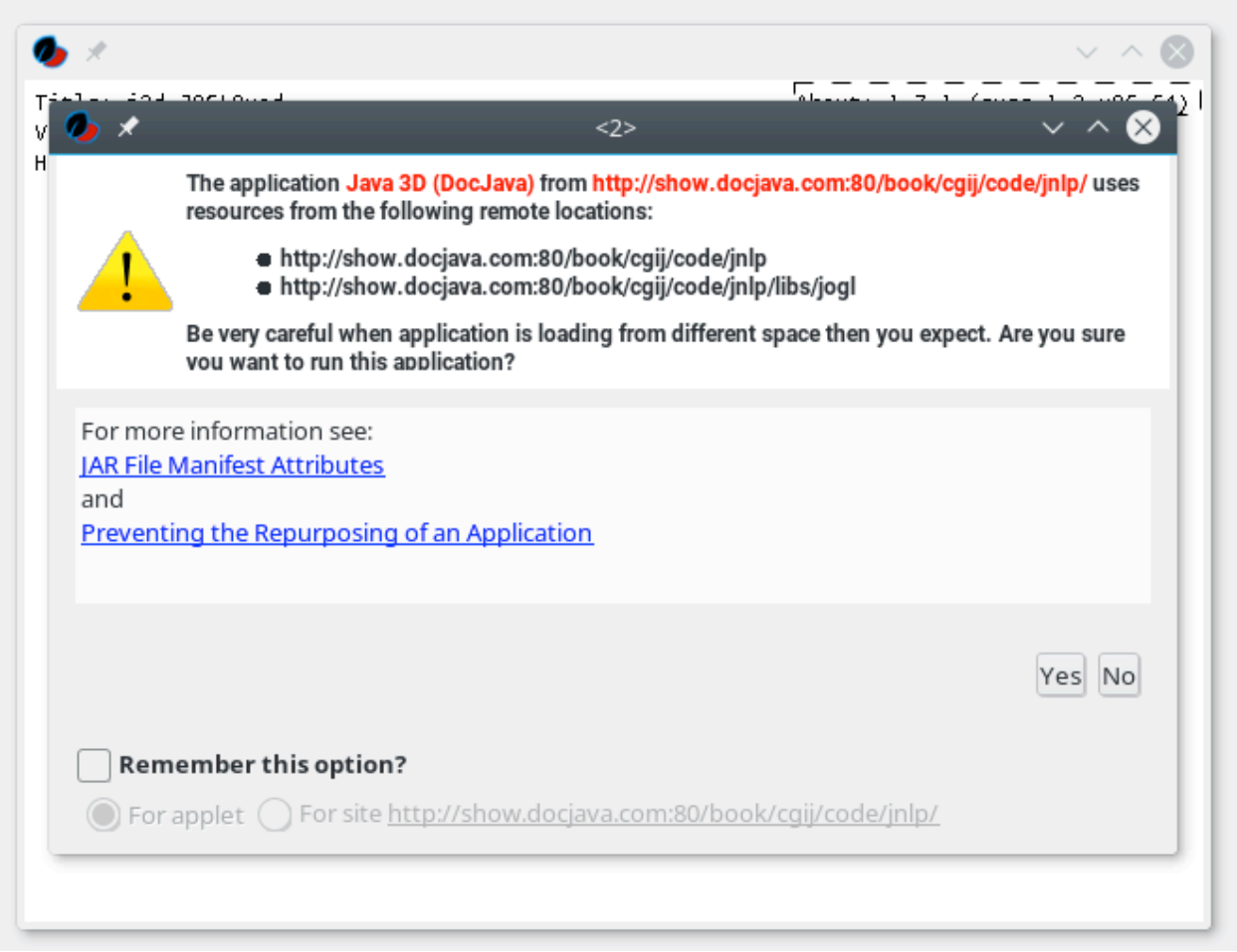

Figure 2 - IcedTea Warnings, JNLP file execution

A developer working in Windows with Oracle JDK 9 and the standard WebStart, has the option to decrease the local security by adding an exception in the Exception Site List, part of the Configure Java panel. The local security exception for the "file:/" protocol should be entered, which allows the execution of any local self-signed jar files during the development phase – the least security level allowed by the WebStart in JDK 9.

The series of commands required for self-signing follows.

```
keytool -genkey -keystore myKeystore -alias myself
keytool -selfcert -alias myself -keystore myKeystore
keytool -list -keystore myKeystore
jarsigner -keystore myKeystore test.jar myself
```

A JDK 8 keytool requires an upgrade by issuing a single-line command:

```
keytool -importkeystore -srckeystore myKeystore -destkeystore
 myKeystore -deststoretype pkcs12
```

The Mac and Linux users may encounter a bug – the Oracle Java control panel window may refuse to show itself and disallow adding any security exception in this manner.

The JNLP deployment is a delicate matter that takes into account the Manifest attributes, deployment rulesets, and signing. Developers may benefit from writing the JNLP files manually as shown because the various integrated development environments do not produce the JNLP files accurately enough.

4. Results

We have been able to deploy to the Windows 7, Windows 10 and the Mac platforms. The WebStart applications that utilize the Java 3D library are available free to run at the author's web page (http://www.docjava.com). The binaries for deployment are the easiest way to reproduce the results and are available from the author's website. To view the results of the work without running any code, there is a YouTube video available at tinyurl.com/j3dAndJogl.

5. Literature Cited

Full texts of the below articles cited are available from the author's website at http://www.docjava.com.

[Koopman] Better Embedded System SW by Phil Koopman. https://betterembsw.blogspot.com/2010/10/embedded-software-costs-15-40-per-line.html. Last accessed 1/12/18.

[Lyon 2004a] Project Initium: Programmatic Deployment by Douglas A. Lyon, Journal of Object Technology, vol. 3, no. 8, September-October 2004, pp. 55-69.

[Lyon 2004b] The Initium X.509 Certificate Wizard by Douglas A. Lyon, Journal of Object Technology, vol. 3, no. 10, November-December 2004, pp. 75-88.

[Lyon 2008] I Resign! Resigning Jar Files with Initium, by Douglas A. Lyon, Journal of Object Technology, vol. 7, no. 4, April-May 2008, and pp. 9-27.

[Oracle] http://www.oracle.com/technetwork/java/javase/9-deprecated-features-3745636.html. Last accessed 1/12/18.

Appendix A. Problems and Workarounds

The Impending Java 3D End-of-Service?

With the arrival of JDK 9, most developers have noticed that their programs do not function anymore. If your program utilizes the Java 3D library (j3dcore.jar, j3dutils.jar, and vecmath.jar), the program depends on the Jogl library as well. You can find the Jogl library in the jogamp-fat.jar file. Please obtain the standard versions of the library files from:
https://jogamp.org/deployment/jogamp-current/fat/ https://jogamp.org/deployment/java3d/1.6.0-final/

Both libraries were made with older versions of JDK and can be compiled with JDK 8 (v1.8.152) nowadays.

If you initially succeed running your program under the JRE 9, you may notice a series of warnings such as these:

```
WARNING: An illegal reflective access operation has occurred
WARNING: Illegal reflective access by javax.media.j3d.JoglPipeline (file:/
     j3d/j3d-core/build/jars/j3dcore.jar) to method
     sun.awt.AppContext.getAppContext()
WARNING: Please consider reporting this to the maintainers of
     javax.media.j3d.JoglPipeline
WARNING: Use --illegal-access=warn to enable warnings of further illegal
     reflective access operations
WARNING: All illegal access operations will be denied in a future release
```

These warnings cannot be removed, but they will not prevent you from running your program either. In the case of JNLP deployment, the particular VM arguments are necessary:

```
--add-exports=java.base/java.lang=ALL-UNNAMED
--add-exports=java.desktop/sun.awt=ALL-UNNAMED
--add-exports=java.desktop/sun.java2d=ALL-UNNAMED
```

These arguments do not change the way the program executes otherwise, as in the case of a program running as an executable jar file.

Reportedly, you may encounter such warnings when you refer to or otherwise use a field, method, or a class in the Java runtime library that has been marked as private in some sense or that is disappearing due to the JDK redesign or code deprecation. Let us consider a sketch. You may wish to list the loaded native libraries by using the command

```
LIBRARIES = ClassLoader.class.getDeclaredField("loadedLibraryNames");
```

Seemingly, there is no private field access involved here and accessing one in a trivial manner would be impossible by design. However, as you follow the line of code execution, you will discover that you are trying to get the list of loaded libraries from the private static Vector<String> loadedLibraryNames in the ClassLoader class. This methodology of listing the loaded native libraries functions well, but it will not function in the future.

If you want to redesign the Java 3D library to compile under the JDK 9, you may do so quickly by erasing a few functionalities that raise critical errors. For example, you would need to remove the references to the package sun.applet. If you wish to solve the next 100 warnings as well, the amount of rewriting required is much higher. It is faster to compile it with the JDK 8 and include the library as such in your application. Still, the problems lurk ahead.

Your Java 3D program will require a few tweaks to get going.

```
System.setProperty("sun.java2d.noddraw", "false");
     System.setProperty("sun.awt.noerasebackground", "true");
```

You may use the Swing framework and display the graphics contents in a JFrame, but you cannot use the JavaFX.

If you want to display the progress of the native libraries loading, you should tweak the Jogl library.

```
System.setProperty("jogamp.debug.JNILibLoader", "true");
```

The native libraries are unpacked and loaded from the temporary directories, so embrace yourself for the lengthy output.

The chances are that the current versions of Java 3D 1.6 and Jogl 2.3.2 (Oct. 2015) won't allow you to use them on the Macintosh computers with the JDK version 9, and on certain Linux computers, notably on the VMware virtual machines (depending from the specific hardware details) due to the Mesa graphics driver-related bug. The libraries can run on the Windows 10 computers with JDK 9, on the Ubuntu 16 distributions with the Oracle JDK 9, and so forth.

By extensive testing, we came to notice the possible workaround for the bug that affects the Macintosh computers so that now they should be able to run the Java 3D and Jogl libraries in the

Java 9 runtime environment. The Jogamp community (that maintains the Jogl library) has suggested the Mesa driver-related bug fix for the next release of the Jogl library.

Avoiding the GLCanvas-related JVM Crash

```
# A fatal error has been detected by the Java Runtime Environment:
#  SIGSEGV (0xb) at pc=0x00000001190a80cc, pid=49353, tid=775
# JRE version: Java(TM) SE Runtime Environment (9.0+11) (build 9.0.1+11)
# Java VM: Java HotSpot(TM) 64-Bit Server VM (9.0.1+11, mixed mode, tiered,
    compressed oops, g1 gc, bsd-amd64)
# Problematic frame:
# C  [libosxapp.dylib+0x20cc]  -[NSApplicationAWT sendEvent:]+0x179
```

The error shown above occurs on the Macintosh computers featuring the latest version of the MacOS X as well as the earlier editions, given that these computers have the Java 9 installed and your application uses the Java 3D and/or Jogl libraries. One might assume that the Java AWT (Abstract Window Toolkit) contains a bug, but the explanation is largely incomplete. You could setup your environment (i.e. execute this command in the terminal) to print the AWT debugging info:

```
export JAVA_AWT_VERBOSE=1
```

Nonetheless, you will not be able to properly run and debug the program. We cannot claim who is responsible for the bug ultimately, but it is possible to point at the Jogl library usage.

Let us consider a Java class JoglTest that implements the GLEventListener from the package com.jogamp.opengl. Any such class needs to implement the necessary methods that draw graphics visuals to a GLCanvas. We shall ignore the details and focus on the Frame or JFrame to which we add the GLCanvas in order to display the graphics animation.

```
GLCanvas canvas = new GLCanvas();
JFrame jFrame = new JFrame("Jogl Drawing");
Animator animator = new Animator(canvas);
canvas.addGLEventListener(new JoglTest ());
jFrame.add(canvas);
```

The problem occurs at the very moment when you close the Frame or JFrame. The default on-close operation is hide-on-close, which does nothing particularly useful anyways. You need at least to set the exit-on-close operation.

```
jFrame.setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE);
```

If you close the jFrame, you crash the JVM 9. Further, the typical usage scenario is to attach a window adapter to the Frame or JFrame that stops the animator and disposes the window.

```
        jFrame.addWindowListener(new WindowAdapter() {
            @Override
            public void windowClosing(WindowEvent e) {
                animator.stop();
                jFrame.dispose(); // JVM 9 CRASH
            }
        });
```

The code above makes the previously mentioned exit-on-close operation irrelevant and shows where the JVM crash occurs specifically. Therefore, in order to avoid the crash, you need to eliminate the command jFrame.dispose() and replace it with the command to exit the program altogether, System.exit(0).

Seemingly, without any negative effect, we allow the user to use the application safely and exit. It is safe to assume that there is a memory leak associated with the step in which we avoid disposing the jFrame. Disposing the window can still be performed on the computers other than the Mac.

```
if (!System.getProperty("os.name").toLowerCase().contains("mac os"))
     jFrame.dispose();
```

Next, the developers should avoid any similar situations in which the frame may be disposed of.

Minimal Corrections of Java 3D Library

When you use the Java 3D library, the presence of the Jogl bug is much more cryptic to read from the error output. In order to initiate a Java 3D application, developers needs to create a GraphicsConfiguration by using the template known as the GraphicsConfigTemplate3D.

```
GraphicsConfigTemplate3D tmp = new GraphicsConfigTemplate3D();
GraphicsConfiguration graphicsConfiguration = getLocalGraphicsEnvironment()
                    .getDefaultScreenDevice().getBestConfiguration(tmp);
```

The JVM 9 crash occurs when trying to obtain the graphicsConfiguration, in the second line of the code shown above. The bug can be further traced to the GraphicsConfigTemplate3D class in the package javax.media.j3d. There, a single line causes the crash.

```
GraphicsConfiguration graphicsConfiguration = (GraphicsConfiguration)
     testCfg;
```

Next, we can trace the bug in the JoglPipeline class in the package javax.media.j3d, in the method:

```
GraphicsConfiguration getBestConfiguration(GraphicsConfigTemplate3D gct,
     GraphicsConfiguration[] gc)
```

in the line canvas.doQuery(). Finally, in the doQuery() method we can see the two lines of code that exhibit an immediate effect on the JVM 9 crash.

```
context.destroy();
nativeWindow.destroy();
```

Removing these lines of code annuls the crash. In addition, the JoglPipeline class contains another potential source of crash, the line glContext.destroy(), which we can remove as well.

You can download the Java 3D library with these minimal changes included, from the http://www.docjava.com web site. You can search for the comment lines that contain the warning: "Potential JVM 9 CRASH occurs here on MacOS" to guide yourself to the locations of interest.

The package jogamp.opengl in the Jogl library contains the class GLContextImpl that implements the method destroy(). The Java 3D library calls this method when the crash occurs, so if you omit the method, you avoid the crash.

However, the details remain fuzzy. The method can run in its entirety without causing any crash in a test program that does not use the Java 3D library. At startup, the method runs once on the main thread. When the window is closing and specifically the method dispose() is invoked, the JVM crashes. Prior to the crash, the method destroy() prints a message claiming that it ran again on the AWT-EventQueue-0 thread. Further testing will be required.

Notes on Building Jogl and Java 3D

The instructions for building Jogl can be found at the following address.

http://jogamp.org/gluegen/doc/HowToBuild.html. Compiling the project on Windows may prove to be flakey. Use the MinGW-W64 compiler version 4.9.x as the starting point. The Gluegen project may require you to run the ant process twice to get through the dll test error. The Jogl project may require you to have a newer compiler such as the gcc x86_64-win32-sjlj-rev1 7.2.0 due to the missing include files. A Cygwin's version of the same compiler just might be of use, if you edit the file gluegen\src\native\windows\WindowsDynamicLinkerImpl_JNI.c and replace

```
typedef int intptr_t;
```

with

```
typedef __intptr_t intptr_t;
```

Other platforms should give you less trouble compiling the project by using the latest gcc compiler and JDK 8. However, you must repeat the build procedure on different platforms to build all the library jars.

The Java 3D is easier to build. First, clone the repositories.

```
git clone git://github.com/hharrison/vecmath
git clone git://github.com/hharrison/java3d-core j3d-core
git clone git://github.com/hharrison/java3d-utils j3d-utils
mkdir jogl-v2.3.2
wget http://jogamp.org/deployment/v2.3.2/jar/gluegen-rt.jar
wget http://jogamp.org/deployment/v2.3.2/jar/jogl-all.jar
mv gluegen-rt.jar jogl-v2.3.2
mv jogl-all.jar jogl-v2.3.2
```

If you possess a custom-built Jogl and Gluegen library jars, include them in these two lines, found in the build.xml file, in the j3d-core directory.

```
<property name="jogl.lib" location="../jogl-v2.3.2/jogl-all.jar"/>
<property name="gluegen.lib" location="../jogl-v2.3.2/gluegen-rt.jar"/>
```

Next, run the ant in the vecmath and j3d-core directories (or, ant clean, if needed).

Your application will require the freshly produced libraries (j3dcore.jar, j3dutils.jar, and vecmath.jar) and the original jogamp-fat.jar. In the case that you do build your own custom Jogl and Gluegen projects, you can include the produced jars such as gluegen-rt-natives-windows-amd64.jar, gluegen-rt.jar, jogl-all-natives-windows-amd64.jar, and jogl-all.jar, whose exact names depend from the platform you are working on.

Acknowledgement

The authors are very thankful for the help of Julien Gouesse for his work in supporting JOGL and its hardware accelerated OpenGL rendering.

Biography

Douglas A. Lyon is a Professor in the Electrical and Computer Systems Engineering department at Fairfield University, in Fairfield Connecticut, USA, a licensed professional engineer, a senior member of the IEEE, President of DocJava, Inc., and President of the Inventors Association of Connecticut. Dr. Lyon teaches Engineering Entrepreneurship. He brought one successful Kickstarter project to market. He received the Ph.D., M.S., and B.S. degrees in computer and systems engineering from Rensselaer Polytechnic Institute (1991, 1985 and 1983). Dr. Lyon has worked at AT&T Bell Laboratories and the Jet Propulsion Laboratory. Dr. Lyon has authored or co-authored three books (Java Digital Signal Processing, Image Processing in Java and Java for Programmers). He has authored over 49 journal publications.

Predrag L. Bokšić graduated with a B.S. in physics from the University of Novi Sad, Serbia. He is a Java programming enthusiast with interests in computer graphics, chaos theory, and AI. He works in the localization and web services testing areas.