Java Splash Screen

Roy Ratcliffe

20th January 2004

Abstract

Designing a simple but effective splash screen for Java is the primary goal. Learning about Java is a secondary objective. This mini-project successfully develops a Java splash screen implementation while meeting essential design goals of maximum run-time efficiency along with minimal developer effort. Minimum run-time overhead comes from using Java's ImageObserver interface for fully asynchronous image loading.

Contents

List of Figures

1 Introduction

Applications frequently display a "splash screen" at start-up. Splash screens have become a user-interface standard. Java applications are no exception. System start-up is a good time to *advertise*. Many applications unavoidably delay the user for a second or longer while starting up. Giving the user something to look at and read during that inevitable delay inarguably enhances feedback, an important user interface design quality. Of course, zero delay is best, but delay with splash beats delay with nothing!

2 Requirements analysis

Splash screen functional requirements:

- Create a general-purpose splash screen for application start-up.
- The splash screen includes an image and optionally a status bar for progress messages during load-time.

Operational constraints and non-functional requirements:

- Minimise any delay caused by the splash screen itself. Negligible delay preferably if possible. If the application loads faster than the image, skip the splash screen.
- Implement using Java AWT only, not Swing. Executes faster because Java has fewer classes to load before execution begins.

Figure 1's Use Case encapsulates the fundamental requirement.

Figure 2 sketches the user interface. Note absence of standard window decorations: no title bar, sizing borders.

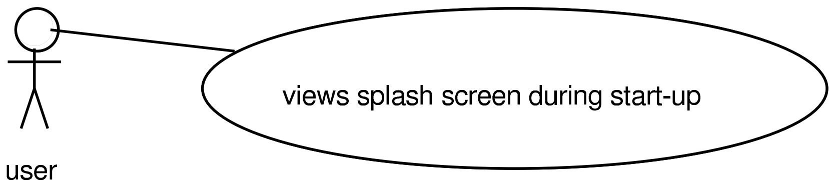

Figure 1: User views splash screen during application start-up

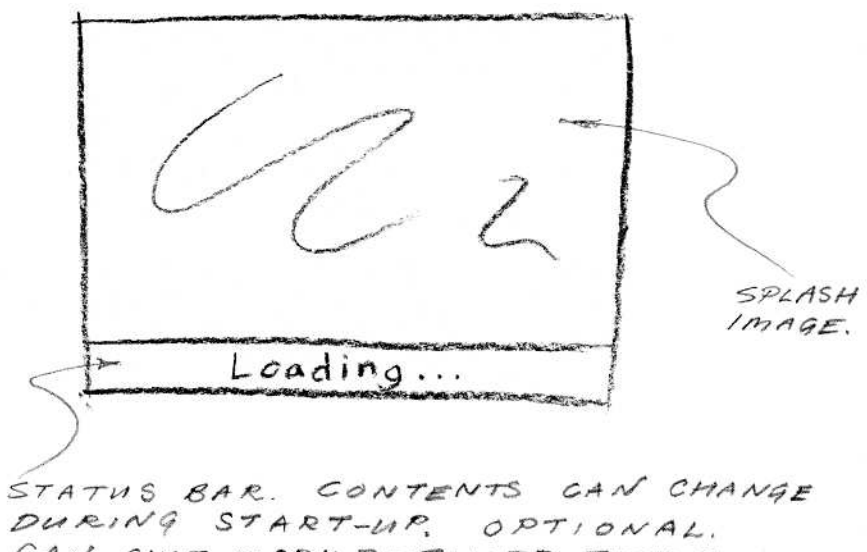
Loading...
SPLASH IMAGE.
STATUS BAR. CONTENTS CAN CHANGE DURING START-UP. OPTIONAL. CAN GIVE MORE DETAILED FEEDBACK.

Figure 2: Splash screen user interface

2.1 Comparisons

Other authors and engineers have also published splash screen software in Java language. See Colston, Randelshofer, O'Hanley, Berthou, Gagnon. However, all these implementations share one or more of the following disadvantages.

- Uses Java's MediaTracker to load the image and thereby introduces delay in loading the application as a whole. The thread waits for complete loading before proceeding with application start-up. This defeats the minimal overhead requirement listed above, Section 2.
- Uses Swing classes which take longer to load compared to simpler AWT. Randelshofer argues this point very well.
- Does not include a status bar below the splash image. Hence there is no way to report progress, assuming something to report!

This implementation overcomes all these disadvantages, as well as achieving simpler implementation with less overhead.

2.2 Optional requirements

There are many splash screen implementations for Java. Those cited in the previous section are only a handful. The various designs reflect subtle differences in requirements. Those differences step beyond the very basic requirements outlined in Section 2. For this reason, they might be called 'optional' requirements. They boil down to two behavioural features:

- displaying the splash screen for a minimum period of time;
- always displaying the splash screen regardless of start-up speed.

Both of these mitigate the requirement to minimise start-up delays. They introduce extra delay unnecessarily. Arguably, this can be viewed as poor user interface design. Derek Clarkson makes this point. News article reproduced with permission, see Appendix D. The balance of conflicting requirements should fall on the side of feedback, a fundamental tenant of user interface design. At the point of intersection between start-up completion and splash screen, the splash role changes from useful feedback to cruft[1]. This is a personal view.

Assuming continuous increase of computer power, an application loading in three seconds today might load in 300 milliseconds next year, then perhaps 3 milliseconds in three years. This not wishful thinking. 64-bit architectures, dual cores, hyperthreading and solid-state storage make this possibility far less than far-fetched. In reality however, what hardware gives, software might take away, as Randelshofer rightly points out. Notwithstanding, even today you can experience this phenomenon using

[1]`http://www.hyperdictionary.com/computing/cruft`

older applications, those predating contemporary machines. The hard drive light goes out long before the splash screen disappears. The artificial delay is obvious. This refers to application loading not operating system booting which has increased in load time along with its complexity. So do you really want to torment the user again and again? He enjoyed the splash for the first 100 times, but now it irritates. Especially after the upgrade; his new top-of-the-line workstation loads the application in exactly the same time as the old one. How exasperating! Again, speaking from personal experience.

Nevertheless, the analysis adds these two design requirements as optional at developer's discretion. They represent a pragmatic compromise based on others' differing design requirements. Certainly, a human-interfacing case exists for the first of these, i.e. displaying for minimum time, for the purpose of avoiding splash screen flash. Though not regardless of start-up speed. If the application is ready to roll before the splash starts, this requirement does *not* imply waiting! The second requirement to 'always display the splash' implies waiting. This fully compromises the original mandatory design requirement.

3 Object-oriented design and implementation

3.1 Classes

See Figure 3. This diagram depicts the basic object classes.

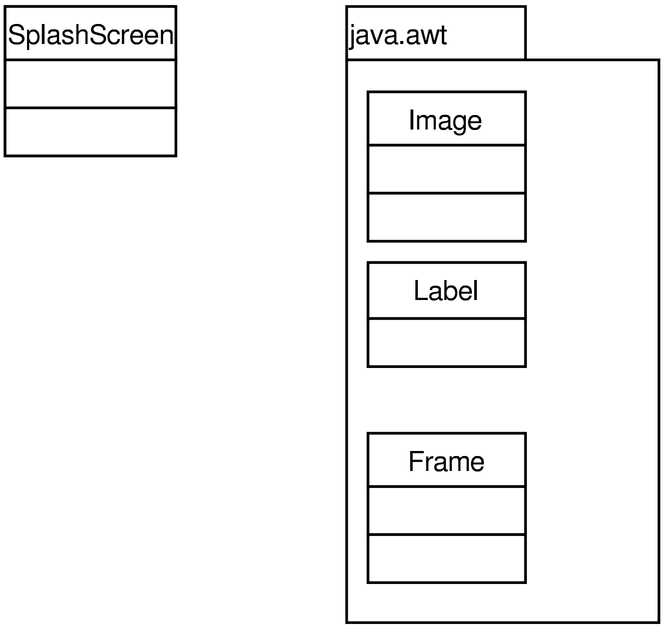

Figure 3: Basic class diagram, AWT user interface components on the right

Notice that SplashScreen does *not* inherit from Frame. This is a design decision.

Although programmers may think of it as a Frame, or Window, these are merely elements of the splash screen's composition and implementation. SplashScreen is a separate and distinct concept. Inheriting from widgets also fails the 'is a kind of' test, e.g. relative to Frame. For example, you can add more contents to a Frame. You can add a menu bar, turn the whole thing into an icon, and so forth. These are not appropriate SplashScreen behaviours, therefore SplashScreen is not a Frame or Window or any such thing.

3.2 Associations

SplashScreen does have association with the Graphical User Interface classes, but not inheritance. Figure 4 depicts *compositional* associations.

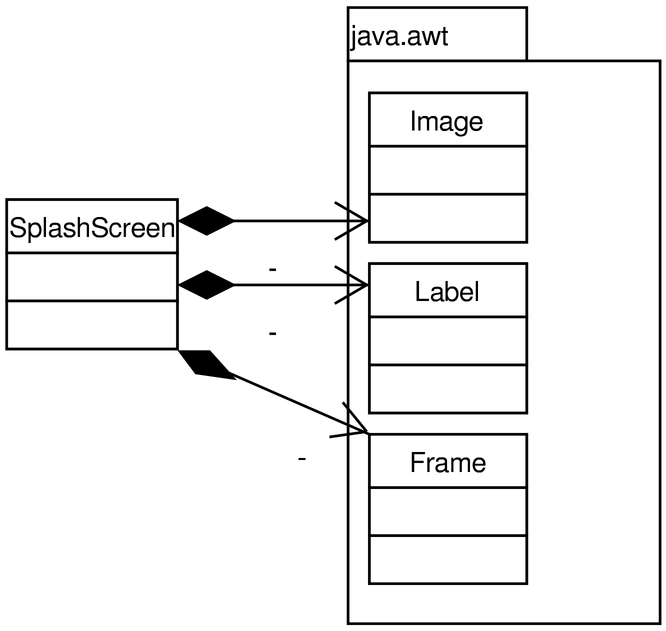

Figure 4: Class diagram depicting composition associations

Therefore, using Java AWT, SplashScreen comprises one Image, one Label and one Frame. The image and label appear together inside the frame. There could be other implementations using different components. This is just one example using AWT.

3.3 Behaviours

Figure 5 draws the SplashScreen class with its basic methods added. Notice that SplashScreen here assumes a default image, `splash.gif` for example. It does not need telling which image unless it differs from the default. Nor have we given it a

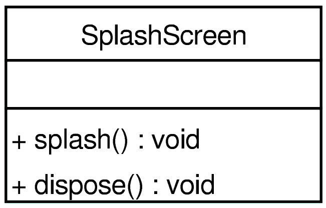

Figure 5: SplashScreen class with methods for splash and dispose

status bar message. Again it assumes a default, `Loading...` for example. Therefore these two methods without arguments represent the simplest mode of operation. You simply invoke SplashScreen.splash() at the start of main() and invoke SplashScreen.dispose() at the end.

Here is an example.

```
public static void main(String[] args) {
    SplashScreen splashScreen = new SplashScreen();
    splashScreen.splash();
    //
    //
    //
    splashScreen.dispose();
}
```

Importantly, SplashScreen.splash() returns almost immediately. There is no delay while waiting for the image to load; it all happens asynchronously. Therefore, normal application start-up proceeds unabated.

3.3.1 Optional behaviours

See Section 2.2 concerning optional requirements. The design incorporates two modifications to the basic behaviour. The first meets optional requirement for minimum splash duration. So if the splash screen is already up, it lasts for some minimum time before disappearing. See outline below.

```
public static void main(String[] args) {
    SplashScreen splashScreen = new SplashScreen();
    splashScreen.splash();
    //
    //
    //
    splashScreen.splashFor(1000); // 1000ms=1s
    splashScreen.dispose();
}
```

This example adds splashFor(1000). If the splash screen is already displayed, it waits for at most 1,000 milliseconds before returning. Note this means 1,000 milliseconds of total splash! So, if already displayed for 1,000 milliseconds or more, the delay is 0. In other words, it avoids flickering the splash on then off if the splash timing coincides with application start-up timing. This is a compromise between user feedback and cruft. If the splash takes longer than start-up, it does not appear at all. If already display for a fraction of the given time, the delay is *only* the remainder.

The second optional behaviour requires waiting for the splash before proceeding.

```
 public static void main(String[] args) {
     SplashScreen splashScreen = new SplashScreen();
     splashScreen.splash();
     //
     //
     //
     splashScreen.waitForSplash();
     splashScreen.splashFor(1000);
     splashScreen.dispose();
 }
```

The example adds waitForSplash. As a guide, invoke this method at the *end* of start-up, not the beginning. Waiting for the image to load does not make the image load faster, necessarily. Image loading is an 'input bound' process, reading from filesystem or network. Remaining start-up steps are typically 'compute bound' and likely compute resource is available for consumption. Most likely, start-up mixes input and compute resource demands, and possibly even output.

This guideline applies to uniprocessor as well as multiprocessor platforms. Waiting only wastes any available compute cycles. If you need to delay unnecessarily, do this at the end when there is nothing left to do. Even so, this practice can be viewed as user interface cruft! Please use with care.

3.4 Implementation

By realising (i.e. implementing) Java's java.awt.image.ImageObserver interface, SplashScreen meets two simultaneous goals:

1. operates almost entirely asynchronously, so no appreciable application delay while starting up and start-up speed remains unaffected by splash image size or location;

2. if the application loads *before* the splash, the splash screen does *not* flash before the eyes for a small fraction of a second, instead it sensibly skips the splash screen.

See Figure 6. New class diagram also includes +showStatus(s: String): void method used for updating status bar contents.

Appendix A lists the complete Java implementation.

4 Tests

Tests use Java's System.currentTimeMillis() method for timing. See Appendix B listing a simple class for marking points on the millisecond time axis and reporting elapsed

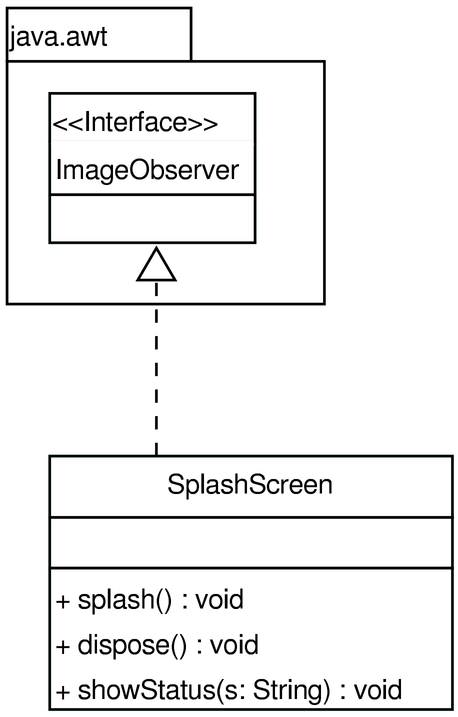

Figure 6: SplashScreen realises ImageObserver interface

intervals.

All test results use Java 2 Platform, Standard Edition SDK, version 1.4.1 running on Linux.

4.1 Null test

```
class test1 {
    public static void main(String[] args) {
        Time.mark(); // t0
        System.out.println(Time.mark("t1 after {0}ms"));
        System.out.println(Time.mark("t2 after {0}ms"));
    }
}
```

Very simple test without splash screen, without anything except timing points. This is the baseline. The timeline marks three points: t_0 when entering main(), t_2 at the end just before leaving main() and t_1 falls somewhere in the middle. After adding the splash screen, t_1 represents the transition point where splash screen start-up ends and application start-up begins.

Compile and run gives:

```
t1 after 0ms
t2 after 29ms
```

4.2 Loop delay

```
class test2 {
    public static void main(String[] args) {
        Time.mark(); // t0
        System.out.println(Time.mark("t1 after {0}ms"));

        for (int i = 5; i > 0; i--) { // 5, 4, 3, 2, 1
            try {
                Thread.sleep(1000);
            }
            catch (InterruptedException ie) {}
        }

        System.out.println(Time.mark("t2 after {0}ms"));
    }
}
```

Loop with sleep delay emulates start-up activity. Five seconds of activity runs in-between the two time marks, t_1 and t_2. Still no splash screen.

Compile and run gives:

```
t1 after 0ms
t2 after 5,078ms
```

4.3 Simple splash

```
class test3 {
    public static void main(String[] args) {
        Time.mark(); // t0

        SplashScreen splashScreen = new SplashScreen();
        splashScreen.splash();

        System.out.println(Time.mark("t1 after {0}ms"));

        for (int i = 5; i > 0; i--) { // 5, 4, 3, 2, 1
            splashScreen.showStatus(Integer.toString(i) + "...");
            try {
                Thread.sleep(1000);
            }
            catch (InterruptedException ie) {}
        }

        splashScreen.dispose();

        System.out.println(Time.mark("t2 after {0}ms"));
    }
}
```

This test finally adds the splash screen. SplashScreen.splash() method runs between t_0 and t_1. SplashScreen.dispose() runs just before t_2. Results as follows. Notice the change.

```
t1 after 443ms
t2 after 5,077ms
```

Because it loads `splash.gif` from the filesystem, it loads quickly and the splash screen appears before the first of the five count-down seconds elapses. Loading images across a network usually takes longer. Subsequent tests will attempt this.

4.3.1 Conclusions

- Java AWT consumes about half a second just to create a new Frame instance.
- Disposing of the splash screen Frame takes no significant time.

4.4 Frame instantiation

```
class test4 {
    public static void main(String[] args) {
        Time.mark(); // t0

        new java.awt.Frame();

        System.out.println(Time.mark("t1 after {0}ms"));
    }
}
```

Tests the previous conclusion from simple splash test, Section 4.3. Does simple Frame object instantiation really take about 500 milliseconds? Test gives:

```
t1 after 424ms
```

It turns out that running the tests through a 100-megabit X Windows connection adds 100 milliseconds to this delay. Running the tests on a direct terminal brings the delay down to around 350ms.

4.4.1 Conclusions

- Previous conclusion confirmed, i.e. Java AWT consumes nearly half a second just to create a new Frame instance.
- SplashScreen is not consuming time unnecessarily. This delay is the minimum. It needs to construct a frame, although perhaps parallelising this step may be possible as a future project.

4.5 Google splash

```
import java.net.*;

class test5 {
    public static void main(String[] args) {
        Time.mark(); // t0

        SplashScreen splashScreen = new SplashScreen();
        URL url = null;
        try {
            url = new URL("http://www.google.com/logos/newyear04.gif");
        }
        catch (MalformedURLException mue) {}
        splashScreen.setImage(url);
        splashScreen.splash();

        System.out.println(Time.mark("t1 after {0}ms"));

        for (int i = 5; i > 0; i--) { // 5, 4, 3, 2, 1
            splashScreen.showStatus(Integer.toString(i) + "...");
            try {
                Thread.sleep(1000);
            }
            catch (InterruptedException ie) {}
        }

        splashScreen.dispose();

        System.out.println(Time.mark("t2 after {0}ms"));
    }
}
```

This test now loads the splash image from `http://www.google.com/logos/newyear04.gif` through a 576-kilobit per second Internet connection. Naturally, this takes longer to load, but will this affect t_1? See below.

```
t1 after 453ms
t2 after 5,093ms
```

No significant difference. However, it does take a second longer for the splash screen to appear. In fact, the status bar reads `4...` by the time it appears.

4.5.1 Conclusions

- Fully asynchronous image loading works.
- Application start-up can freely update the splash screen's status bar before it actually appears.

4.6 Randelshofer comparison

Adding time markers to Randelshofer ScreenWindow gives the following measurements. Appendix C lists timing additions to `MyAppSplash.java` test source.

```
t1 after 1,536ms
t2 after 296ms
```

The delay between t_0 and t_1 becomes one-and-a-half seconds. This is the time it takes to *start* the splash screen, or about three times as slow compared to SplashScreen. Waiting for completed image load makes this long delay. It is also highly variable, depending on connection bandwidth availability. Randelshofer waits for image loading before displaying the slash screen *and* before proceeding with application start-up. As a consequence, the splash quickly flashes on the screen before disappearing again and the application appears.

4.7 Slow splash

Discover what happens when things fail to go as expected!

```
import java.net.*;

class test6 {
    public static void main(String[] args) {
        Time.mark(); // t0

        SplashScreen splashScreen = new SplashScreen();
        URL url = null;
        try {
            url = new URL("http://www.google.com/logos/newyear04.gif");
        }
        catch (MalformedURLException mue) {}
        splashScreen.setImage(url);
        splashScreen.splash();

        System.out.println(Time.mark("t1␣after␣{0}ms"));

        ;

        splashScreen.dispose();

        System.out.println(Time.mark("t2␣after␣{0}ms"));
    }
}
```

The Google test from Section 4.5 is the basis of this first pathology test. However, by removing the delay loop, the five-second delay drops to zero seconds! This simulates fast application start-up versus long splash screen start-up.

```
t1 after 457ms
t2 after 25ms
```

No application start-up delay, and the splash screen does not appear.

4.8 Multiple splashes

Not a serious idea. *Flashing* the splash screen by recreating it is not a requirement, but it does validate the class clean-up logic, demonstrating its repeatability and therefore to some extent its reliability. Think of it as a destruction test, like hitting it with a hammer.

```java
import java.net.*;

class test7 {
    public static void main(String[] args) {
        Time.mark(); // t0

        SplashScreen splashScreen = new SplashScreen();
        URL url = null;
        try {
            url = new URL("http://www.google.com/logos/newyear04.gif");
        }
        catch (MalformedURLException mue) {}
        splashScreen.setImage(url);
        for (int repeats = 3; repeats > 0; repeats--) {
            splashScreen.splash();

            System.out.println(Time.mark("t1 after {0}ms"));

            for (int i = 5; i > 0; i--) { // 5, 4, 3, 2, 1
                splashScreen.showStatus(Integer.toString(i) + "...");
                try {
                    Thread.sleep(1000);
                }
                catch (InterruptedException ie) {}
            }

            splashScreen.dispose();
        }
        System.out.println(Time.mark("t2 after {0}ms"));
    }
}
```

This is the Google splash from Section 4.5 but repeated three times.

```
t1 after 450ms
t1 after 5,271ms
t1 after 5,243ms
t2 after 5,046ms
```

It passes point t_1 three times, once for each splash iteration. Interestingly, note the reduced splash start-up delay on second and third iterations. It nearly halves to around 250ms! The reason? Java has already loaded the AWT classes.

4.9 Splash duration

```
import java.net.*;

class test8 { // based on test6
    public static void main(String[] args) {
        Time.mark(); // t0

        SplashScreen splashScreen = new SplashScreen();
        URL url = null;
        try {
            url = new URL("http://www.google.com/logos/newyear04.gif");
        }
        catch (MalformedURLException mue) {}
        splashScreen.setImage(url);
        splashScreen.splash();

        System.out.println(Time.mark("t1 after {0}ms"));

        ;

        splashScreen.splashFor(1000); // finish if started
        splashScreen.dispose();

        System.out.println(Time.mark("t2 after {0}ms"));
    }
}
```

Based on Section 4.7, this example has no appreciable start-up delay. However, before disposing the splash, it invokes splashFor(1000) on the SplashScreen instance. If the splash screen is up, it will keep it up for one second in total by sleeping the caller's thread. But if not yet up, splashFor returns immediately and the test disposes the splash screen. It gives the following output.

```
t1 after 476ms
t2 after 26ms
```

No splash screen appears, and it reaches t_1 in only 26ms. Method splashFor returned immediately because the splash screen was not yet displayed. This is correct behaviour for this part of the optional requirements.

4.10 Crufted splash

```
import java.net.*;

class test9 { // based on test8
    public static void main(String[] args) {
        Time.mark(); // t0

        SplashScreen splashScreen = new SplashScreen();
        URL url = null;
        try {
            url = new URL("http://www.google.com/logos/newyear04.gif");
        }
        catch (MalformedURLException mue) {}
        splashScreen.setImage(url);
        splashScreen.splash();

        System.out.println(Time.mark("t1␣after␣{0}ms"));

        ;

        splashScreen.waitForSplash(); // cruft?
        System.out.println(Time.mark("t1a␣after␣{0}ms"));
        splashScreen.splashFor(1000);
        splashScreen.dispose();

        System.out.println(Time.mark("t2␣after␣{0}ms"));
    }
}
```

This test adds waitForSplash() before the splashFor(1000). It waits for the splash to load and display first. Then it displays for one second. The test incorporates a new timing test point, t_{1a}. It marks the point of transition between waiting for the splash to display and waiting for one second. Output as follows.

```
t1 after 480ms
t1a after 669ms
t2 after 1,021ms
```

These figures show that the splash image loads 669 milliseconds after starting. It then waits for a further second before completing. Compared to previous test therefore, the user waits about two seconds compared to half a second. In this worst case example, crufted splash quadruples the application start-up time.

References

R. Berthou. Rbl java tips: A splash screen in java. `http://www.javaside.com/asp/mus.asp?page=/us/tips/j_9.shtml`.

Tony Colston. Java tip 104: Make a splash with swing. `http://www.javaworld.com/javaworld/javatips/jw-javatip104.html`.

Waterwerks Pty Ltd Derek Clarkson. Re: A splash screen. `news://comp.lang.java.gui`.

Real Gagnon. Display a splash screen. `http://www.rgagnon.com/javadetails/java-0267.html`.

John O'Hanley. Collected java practices: Splash screen. `http://www.javapractices.com/Topic149.cjp`.

Werner Randelshofer. How to do a fast splash screen in java. `http://www.randelshofer.ch/oop/javasplash/javasplash.html`.

Acknowledgements

Thanks to Werner Randelshofer `mailto:werner.randelshofer@bluewin.ch` for overseeing Apple Mac OS X platform debugging, including much constructive analysis and warm encouragement.

A SplashScreen.java

```java
/*
 * @(#)SplashScreen.java 1.5 20-Jan-04
 *
 * Copyright (C) 2004, Roy Ratcliffe, Lancaster, United Kingdom.
 * All rights reserved.
 *
 * This software is provided ``as is'' without warranty of any kind,
 * either expressed or implied. Use at your own risk. Permission to
 * use or copy this software is hereby granted without fee provided
 * you always retain this copyright notice.
 */

import java.awt.*;
import java.awt.image.ImageObserver;
import java.net.URL;

/**
 * SplashScreen is a general-purpose splash screen for application
 * start-up. Usage is straightforward: simply construct a
 * SplashScreen at the start of main() and call its splash() method.
 * Proceed with start-up as normal. Use showStatus(String) for
 * reporting progress during start-up. Finally, at the end of main()
 * call SplashScreen's dispose() method. By default, the splash
 * loads image splash.gif but you can change this if necessary.
 *
 * <h3>Example 1</h3>
 * <pre>
 * class splasher1 {
 *     public static void main(String[] args) {
 *         SplashScreen splashScreen = new SplashScreen();
 *         splashScreen.splash();
 *         for (int i = 10; i > 0; i--) {
 *             splashScreen.showStatus(Integer.toString(i) + "...");
 *             try {
 *                 Thread.sleep(1000);
 *             }
 *             catch (InterruptedException ie) {}
 *         }
 *         ; // frame.show() <-- here
 *         splashScreen.dispose();
 *         ; // frame.show() <-- or here
 *     }
 * }
 * </pre>
 *
 * <h3>Example 2</h3>
 * <pre>
 * class splasher2 {
 *     public static void main(String[] args) {
 *         SplashScreen splashScreen = new SplashScreen();
 *          splashScreen.splash();
 *          try {
 *              Thread.sleep(500);
 *          }
 *          catch (InterruptedException ie) {}
 *          splashScreen.splashFor(1000); // discretion
 *          splashScreen.dispose();
 *      }
 * }
 * </pre>
 * Note following comments quoted from design documentation by R.R.
 * <blockquote>
 * This example adds splashFor(1000).  If the splash screen is
 * already displayed, it waits for at most 1000 milliseconds before
 * returning.  Note this means 1000 milliseconds of total splash!  So, if
 * already displayed for 1000 milliseconds or more, the delay is 0.
 * In other words, it avoids flickering the splash on then off if the
 * splash timing coincides with application start-up timing.  This is
 * a compromise between user feedback and cruft.  If the splash takes
 * longer than start-up, it does not appear at all.  If already
 * display for a fraction of the given time, the delay is
 * <em>only</em> the remainder.
 * </blockquote>
 *
 * <h3>Example 3</h3>
 * <pre>
 * class splasher3 {
 *      public static void main(String[] args) {
 *          SplashScreen splashScreen = new SplashScreen();
 *          splashScreen.splash();
 *          try {
 *              Thread.sleep(100);
 *          }
 *          catch (InterruptedException ie) {}
 *          splashScreen.waitForSplash(); // cruft zone?
 *          splashScreen.splashFor(1000);
 *          splashScreen.dispose();
 *      }
 * }
 * </pre>
 * This example adds waitForSplash.  It waits for the splash screen
 * even though the application loads faster.  Not recommended as some
 * users consider this bad practice.
 *
 * <h3>Example 4</h3>
 * <pre>
 * class splasher4 {
 *      public static void main(String[] args) {
 *          SplashScreen.instance().splash();
 *          SplashScreen.instance().delayForSplash();
 *          try {
 *              Thread.sleep(100);
 *          }
 *          catch (InterruptedException ie) {}
 *          SplashScreen.instance().splashFor(1000);
 *          SplashScreen.instance().dispose();
 *      }
 * }
 * </pre>
 * This example demonstrates two new features of version 1.5
 * SplashScreen.  Firstly, the Singleton pattern.  Class-scoped method
 * instance() accesses the single SplashScreen instance.  You can
 * therefore access this instance from anywhere.
 * <p>
 * Secondly, method delayForSplash() appears just after splash().
 * This <em>possibly</em> delays the main thread, allowing the splash
 * screen to load and display.  Tests on some uniprocessor platforms
 * show poor multi-threading performance.  See Appendix F of design
 * documentation by R.R.  The new method bases the extent of delay if
 * any on number of available computing resources.
 *
 * <h3>Modelling</h3>
 * In U.M.L. modelling terms, SplashScreen fulfils the following
 * requirement depicted as a Use Case.
 * <p>
 * <img src="UseCaseDiagram1.gif">
 * <p>
 * The sketch below outlines the user interface design.
 * <p>
 * <img src="hci.gif">
 * <p>
 * To meet this requirement, the implementation uses the following
 * class design.
 * <p>
 * <img src="ClassDiagram2.gif">
 * <p>
 * Or in full detail as follows.
 * <p>
 * <img src="ClassDiagram2a.gif">
 * <p>
 *
 * @todo Add method or methods for adjusting background colours.
 * @author Roy Ratcliffe
 * @version 1.5
 */
public class SplashScreen implements ImageObserver {

    // Design decision.
    // Choose delegation over inheritance.  SplashScreen is not a
    // Frame or a Window, or an Image for that matter; it is a
    // concept.  Frame and Image are its components.
    //
    // Conceptually, the splash screen is an image with text
    // underneath: an image and a label in Java terms.  The Frame
    // is a somewhat more abstract software-engineering
    // entity.
    //
    // Instantiate the label now and give default contents.  Use
    // method showStatus(s: String) to override the default.  You can
    // call this before splash().  Design feature.
    private Image image;
    private Label label = new Label("Loading...", Label.CENTER);

    private Frame frame;
    private long splashTime = 0;

    /**
     * Constructs SplashScreen using a given filename for the splash image.
     * @param filename name of an image file
     */
    public SplashScreen(String filename) {
        setImage(filename);
    }
    /**
     * Constructs SplashScreen using a given URL for the splash image.
     * @param url the URL of an image
     */
    public SplashScreen(URL url) {
        setImage(url);
    }
    /**
     * Constructs SplashScreen using filename "splash.gif" for the image
     * unless you change the default using setImage or call splash with an
     * argument specifying a different image.
     */
    public SplashScreen() {
    }

    /**
     * Uses the given filename for the splash image.  This method
     * calls Toolkit.getImage which resolves multiple requests for
     * the same filename to the same Image, unlike createImage which
     * creates a non-shared instance of Image.  In other words,
     * getImage caches Images, createImage does not.  Use
     * <code>splash(createImage(</code>... if you want Image privacy.
     * @param filename name of an image file
     */
    public void setImage(String filename) {
        image = Toolkit.getDefaultToolkit().getImage(filename);
    }
    /**
     * Uses the given URL for the splash image.
     * @param url the URL of an image
     */
    public void setImage(URL url) {
        image = Toolkit.getDefaultToolkit().getImage(url);
    }

    /**
     * Starts the asynchronous splash screen using the given filename
     * for the image.
     * @param filename name of an image file
     */
    public void splash(String filename) {
        splash(Toolkit.getDefaultToolkit().getImage(filename));
    }
    /**
     * Starts the asynchronous splash screen using the given URL
     * for the image.
     * @param url the URL of an image
     */
    public void splash(URL url) {
        splash(Toolkit.getDefaultToolkit().getImage(url));
    }
    /**
     * Starts the asynchronous splash screen using the previously
     * specified image, or using filename "splash.gif" by default if
     * no image yet specified.
     */
    public void splash() {
        if (image != null) splash(image);
        else splash(Toolkit.getDefaultToolkit().getImage("splash.gif")); // async
    }

    /**
     * <em>Splash</em> the screen!  Or, in other words, make a splash!
     *
     * Actually, this method merely starts the process of
     * splashing.  The splash screen will appear sometime later when
     * the splash image is ready for display.
     *
     * Note that this splash screen implementation uses fully asynchronous
     * image loading.  The splash() method itself returns to the
     * caller as soon as possible.  The screen appears later as soon
     * as image loading completes.
     *
     * @pre Do not double-splash!  It creates waves!  That is, do not
     * invoke splash() twice, not without calling dispose() in-between.
     *
     * @param img the image used for splashing
     */
    public void splash(Image img) {
        image = img;
        frame = new Frame();
        frame.setUndecorated(true);

        if (!Toolkit.getDefaultToolkit().prepareImage(image, -1, -1, this)) return;
        // Arriving here means the image is already fully loaded.
```

```java
        // The splash screen can proceed without delay.
        splashScreen();
    }

    /**
     * Runs during image loading. Informs this ImageObserver about progress.
     * This method does <b>not</b> run in the main thread, i.e. the thread which
     * calls splash() and dispose(). That is why methods
     * splashScreen() and dispose() are <code>synchronized</code>.
     */
    public boolean imageUpdate(Image img, int infoflags,
                               int x, int y, int width, int height) {
        // debug...
        //      System.err.println("img=" + img +
        //                         ",infoflags=" + infoflags +
        //                         ",x=" + x +
        //                         ",y=" + y +
        //                         ",width=" + width +
        //                         ",height=" + height);

        // Return false if infoflags indicate that the image is
        // completely loaded; true otherwise.
        boolean allbits = infoflags == ImageObserver.ALLBITS;
        if (allbits) splashScreen();
        return !allbits;
    }

    /**
     * Runs when the splash image is fully loaded, all its bits and
     * pieces.
     *
     * @todo Animated splash screens! Is there a requirement? If
     * so, implement animation support.
     */
    private synchronized void splashScreen() {
        // Which thread runs this method? One of two: either the
        // main thread if the image has already loaded, or Java's
        // image loader thread.
        if (frame == null) return;
        final int width = image.getWidth(null);
        final int height = image.getHeight(null);

        // Why use a Canvas? It allows packing. This way, AWT's
        // normal packing mechanism handles the sizing and layout.
        Canvas canvas = new Canvas() {
                // Fix thanks to Werner Randelshofer as follows.
                // Canvas class' update(g) method first clears then
                // invokes paint(g). Its superclass, Component, only
                // calls paint(g). Just paint, not clear! Clearing
                // first unhappily creates flicker. The following
                // override reverts to the super-superclass behaviour
                // of Component.
                public void update(Graphics g) {
                    paint(g);
                }
                public void paint(Graphics g) {
                    g.drawImage(image, 0, 0, this);
                }
                // Fixed for Mac OS X, also thanks to Werner.
                // Werner's kind testing on Apple's JVM reveals an
                // important and subtle difference versus Sun's Java for
                // Linux.  Under Linux, Component.setSize() alters the
                // Canvas ``preferred size'' and therefore affects
                // layout.  Under Mac OS X, it does not.  Actually,
                // reviewing Component.java source reveals that
                // preferredSize() dependency is complex, depending on
                // prefSize attribute if set and Component isValid, or
                // otherwise depends on Component peer, or finally the
                // ``minimum size'' if no peer yet.  Werner's solution
                // seems advisable therefore: override
                // getPreferredSize() method.
                public Dimension getPreferredSize() {
                    return new Dimension(width, height);
                }
            };

        frame.add(canvas, BorderLayout.CENTER);
        frame.add(label, BorderLayout.SOUTH);
        frame.pack();

        Dimension screenSize = Toolkit.getDefaultToolkit().getScreenSize();
        Dimension frameSize = frame.getSize();
        frame.setLocation((screenSize.width - frameSize.width) >> 1, // /2
                          (screenSize.height - frameSize.height) >> 1); // /2
        frame.show();
        splashTime = System.currentTimeMillis();
    }

    /**
     * Changes the status message just below the image.  Use it to
     * display start-up progress.
     */
    public void showStatus(String s) {
        label.setText(s);
    }

    /**
     * Waits for the splash screen to load, returns when the splash
     * starts.  The wait is indefinite if necessary.  The operation
     * returns immediately if the splash image has already loaded.
     * <p>
     * Please note following discussion taken from design
     * documentation by R.R.
     * <blockquote>
     * As a guide, invoke this method at the <em>end</em> of start-up, not the
     * beginning.  Waiting for the image to load does not make the image load
     * faster, necessarily.  Image loading is an 'input bound' process,
     * reading from filesystem or network.  Remaining start-up steps are
     * typically 'compute bound' and likely compute resource is available for
     * consumption.  Most likely, start-up mixes input and compute resource
     * demands, and possibly even output.
     * <p>
     * This guideline applies to uniprocessor as well as multiprocessor
     * platforms.  Waiting only wastes any available compute cycles.  If you
     * need to delay unnecessarily, do this at the end when there is nothing
     * left to do.  Even so, this practice can be viewed as user interface
     * cruft!  Please use with care.
     * </blockquote>
     */
    public void waitForSplash() {
        MediaTracker mt = new MediaTracker(frame);
        mt.addImage(image, 0);
        try {
            mt.waitForID(0);
        }
        catch (InterruptedException ie) {}
        // assert splashTime!=0
    }
    /**
     * Waits for the splash screen to load for a limited amount of
     * time.  Method returns when the splash has loaded, or when the
     * given time limit expires.
     * @param ms milliseconds to wait for
     */
    public void waitForSplash(long ms) {
        MediaTracker mt = new MediaTracker(frame);
        mt.addImage(image, 0);
        try {
            mt.waitForID(0, ms);
        }
        catch (InterruptedException ie) {}
        // assert splashTime!=0
    }
    /**
     * Optimise splash latency by delaying the calling thread
     * according to number of processors available.  Multiprocessor
     * platforms successfully load the splash image in parallel with
     * low overhead.  Uniprocessors struggle however!  This method
     * offers compromise.  It delays <em>indefinitely</em> with one
     * processor, same as waitForSplash(); however, it returns
     * immediately with four our more processors thereby maximising
     * parallel execution; or waits for 500 milliseconds at most with
     * dual processors.  Call delayForSplash() in place of
     * waitForSplash().
     */
    public void delayForSplash() {
        int cpus = Runtime.getRuntime().availableProcessors();
        switch (cpus) {
        case 0: // pathology!
        case 1:
            waitForSplash();
            break;
        case 2:
        case 3: // ?
            waitForSplash(1000 / cpus);
        }
    }
    /**
     * Splashes the screen for at least the given number of
     * milliseconds if, and only if, the splash screen has already
     * loaded.  If not already splashed, the method returns
     * immediately.  Invoke this method before disposing if you want
     * to force a minimum splash period.
     * <p>
     * Why is this method <code>synchronized</code>?  In order to
     * avoid a race condition.  It accesses the splashTime attribute
     * which updates in another thread.
     * @param ms milliseconds of minimum splash
     * @pre The argument is greater than zero.
     * You already called splash.
     */
    public synchronized void splashFor(int ms) {
        if (splashTime == 0) return;
        long splashDuration = System.currentTimeMillis() - splashTime;
        // What time does System.currentTimeMillis measure?  Real
        // time, process time, or perhaps thread time?  If real time,
        // the following sleep duration inaccurately represents the
        // remaining delay.  This process could switch out at any
        // point in-between sampling the time and sleeping, and
        // thereby add to the existing delay!  Ignored for now.
        if (splashDuration < ms)
            try {
                Thread.sleep(ms - splashDuration);
            }
            catch (InterruptedException ie) {}
    }

    /**
     * Closes the splash screen if open, or abandons splash screen if
     * not already open.  Relatively long image loading delays the
     * opening.  Call this method at the end of program start-up,
     * i.e. typically at the end of main().
     * <p>
     * Implementation note.
     * If you dispose too fast, this method could coincide with
     * splashScreen().  We cannot now preempt the other thread.  It
     * needs synchronisation.  This situation and its requirement
     * proves inevitable when two threads access the same thing.  For
     * this reason, methods dispose() and splashScreen() share the
     * <code>synchronized</code> attribute.
     *
     * @pre Assumes previous invocation of splash().  Do not dispose
     * before the splash!
     */
    public synchronized void dispose() {
        // Of course, it is conceivable though unlikely that the
        // frame may not really exist before disposing.  For example,
        // if the splash phase of intialisation runs very quickly.
        // Two splash cycles in a row?  Remove the label ready for
        // the next iteration.  Not a requirement but pathologically safe.
        frame.remove(label);
        frame.dispose();
        frame = null;
        splashTime = 0;
    }

    private static SplashScreen singleton = null;
    /**
     * Ensures the class has only one instance and gives a global
     * point of access to it.  The Singleton pattern avoids using
     * global variables.  Instead, the class itself references the
     * single instance using a class-scoped variable, <em>static</em>
     * in Java terms.
     * <p>
     * The implementation actually mixes singleton and non-singleton
     * patterns.  (Tempting to call it Multiton but that refers to a
     * variation of Singleton where the instance has <em>many</em>
     * multiplicity instead of unity.)  Correctly applying the
     * Singleton pattern requires closing access to constructor
     * methods.  However, SplashScreen retains public constructors, so
     * compromises the pattern.  You can follow Singleton usage or
     * not at your own discretion.
     *
     * @return singleton SplashScreen instance
     */
    // See Double-checked locking and the Singleton pattern
    // http://www-106.ibm.com/developerworks/java/library/j-dcl.html?dwzone=java
    public static synchronized SplashScreen instance() {
        if (null == singleton) singleton = new SplashScreen();
        return singleton;
    }

}
```

B Time.java

```java
/*
 * @(#)Time.java 1.2 18-Jan-04
 *
 * Copyright (C) 2004, Roy Ratcliffe, Lancaster, United Kingdom.
 * All rights reserved.
 *
 * This software is provided ''as is'' without warranty of any kind,
 * either expressed or implied.  Use at your own risk.  Permission to
 * use or copy this software is hereby granted without fee provided
 * you always retain this copyright notice.
 */

import java.text.MessageFormat;

class Time {
    static long t = 0;
    static long mark() {
        long now = System.currentTimeMillis();
        long ms = now - t;
        t = now;
        return ms;
    }
    static String mark(String pattern) {
        long ms = mark();
        Object[] args = {new Long(ms), new Double(ms / 1000.0)};
        return new MessageFormat(pattern).format(args);
    }
    static long at() {
        long now = System.currentTimeMillis();
        long ms = now - t;
        return ms;
    }
    static String at(String pattern) {
        long ms = at();
        Object[] args = {new Long(ms), new Double(ms / 1000.0)};
        return new MessageFormat(pattern).format(args);
    }
}
```

C MyAppSplash.java differences

```
***************
*** 24,29 ****
--- 24,30 ----
   */
  public class MyAppSplash extends Object {
      public static void main(String[] args) {
+       Time.mark();
          // NOTE: The splash window should appear as early as possible.
          //       The code provided here uses Reflection to avoid time
          //       consuming class loading before the splash window is
***************
*** 35,41 ****

          // TO DO: Replace 'splash.gif' with the file name of your splash image.
          Frame splashFrame = null;
!         URL imageURL = MyAppSplash.class.getResource("splash.gif");
          if (imageURL != null) {
              splashFrame = SplashWindow.splash(
                  Toolkit.getDefaultToolkit().createImage(imageURL)
--- 36,46 ----

          // TO DO: Replace 'splash.gif' with the file name of your splash image.
          Frame splashFrame = null;
!       URL imageURL = null;
!       try {
!          imageURL = new URL("http://www.google.com/logos/newyear04.gif");
!       }
!       catch (java.net.MalformedURLException mue) {}
          if (imageURL != null) {
              splashFrame = SplashWindow.splash(
                  Toolkit.getDefaultToolkit().createImage(imageURL)
***************
*** 56,61 ****
--- 61,67 ----

          // TO DO: Replace 'MyApp' with the fully qualified class
          // name of your application.
+       System.out.println(Time.mark("t1 after {0}ms"));
          try {
              Class.forName("MyApp")
              .getMethod("main", new Class[] {String[].class})
***************
*** 69,73 ****
--- 75,80 ----
          // Dispose the splash window by disposing its parent frame
          // --------------------------------------------------
          if (splashFrame != null) splashFrame.dispose();
+       System.out.println(Time.mark("t2 after {0}ms"));
      }
  }
```

D Derek Clarkson's comp.lang.java.gui article

```
Path:
        news.dial.pipex.com!bnewshfeed00.bru.ops.eu.uu.net!master.news.eu.uu.net!bn
        ewsspool00.bru.ops.eu.uu.net!bnewsinpeer01.bru.ops.eu.uu.net!emea.uu.net!fe
        ed.news.tiscali.de!newsfeed.vmunix.org!news1.optus.net.au!optus!news.mel.co
        nnect.com.au!news.syd.connect.com.au!mail.netspeed.com.au!not-for-mail
From: Derek Clarkson <nonsupplied@nospam.com.au>
Newsgroups:
        comp.lang.java.gui,comp.lang.java.help,comp.lang.java.programmer
Subject: Re: A splash screen
Followup-To: comp.lang.java.gui
Date: Tue, 16 Dec 2003 10:08:11 +1100
Organization: Waterwerks Pty Ltd
Lines: 34
Message-ID: <3fde3f66@mail.netspeed.com.au>
References: <3fdcfa1a@rpc1284.daytonoh.ncr.com>
NNTP-Posting-Host: 203.31.48.12
X-Trace: merki.connect.com.au 1071529844 5307 203.31.48.12 (15 Dec 2003
        23:10:44 GMT)
X-Complaints-To: abuse@connect.com.au
NNTP-Posting-Date: 15 Dec 2003 23:10:44 GMT
User-Agent: KNode/0.7.2
X-Original-NNTP-Posting-Host: 210.9.231.227
X-Original-Trace: 16 Dec 2003 10:10:30 +1100, 210.9.231.227
Xref: news.dial.pipex.com comp.lang.java.gui:123864
        comp.lang.java.help:225275 comp.lang.java.programmer:624749
MIME-Version: 1.0
Content-Type: text/plain; charset=us-ascii
Content-Transfer-Encoding: 7Bit

Hi,
I would suggest that having a timed splash screen is not a good idea. I know
that there are a number of products which do this, but I consider it to be
generally not good. The reasons is that quite often a user is starting an
application in order to do something specific. I.e. it might be started in
response to an association on a file or the user might want to quickly
start it, modify a piece of information and then exit. These sorts of
situations are where users get frustrated with applications which take a
long time to start.

Having a splash screen which is sitting in front of them and not assisting
in what they are doing is only going to increase any such annoyance.
Especially if they can see that the application is up and ready behind it
and they can't get to it.

You can add a "close" button to the splash screen but then you are asking
users to perform an additional task in order to use your application. Again
they won't like it.

cruft like splash screens is a great way to make an application look
"pretty", but if they get in the way of the user in any way shape or form,
then you run the risk of the user going somewhere else. Remember, the main
job of any application is to perform a task for the user. Cruft should
never get in the way of this.

If your not convinced by these thoughts, I would suggest you doing some
ready on programs and user interaction. An especially good one that came
out recently was Eric S Raymonds "the Art of Unix Programming" which
highlights a lot of issues in regard to well behaved programs.


--
cio
Derek
```

E Asynchronous load timing experiment

This small experiment aims to discover the practical benefit of asynchronously loading and displaying splash screens without first waiting, assuming benefit exists at all. The design of SplashScreen assumes that freeing the main thread for normal application start-up takes advantage of input-bound image loading. It separates loading from the more compute-bound start-up. Both run in separate threads of execution. Hypothetically, not waiting for complete image load before proceeding does not significantly affect image loading speed and therefore splash timing. Such is the design's theory, but how about the practice?

E.1 Method

The experiment method incorporates uniprocessor and multiprocessor systems. We can expect difference certainly, but because image loading depends on input resource not computing resource (meaning C.P.U.) the difference should not be significant, theoretically.

The test comprises two small Java programs, listings follow. Both measure elapsed time between entering main() and showing the splash. 'Showing splash' refers to entering the paint(Graphics) method. However, the second test waits for image loading, the first does not.

Both tests use SplashScreen version 1.3 of 6th January 2004. It differs only by a println statement on entering paint. The new Canvas.paint(Graphics) method override reads:

```
public void paint(Graphics g) {
    System.out.println(Time.mark("t1a after {0}ms"));
    g.drawImage(image, 0, 0, this);
}
```

The experiment plots three test points:

- t_1 marks the starting point of a simulated 500-millisecond start-up activity;
- t_{1a} marks the image paint entry-point, as above;
- t_2 marks the point of splash dispose.

E.1.1 Test program 1

```
class async1 {
    public static void main(String[] args) {
        Time.mark(); // t0
        SplashScreen splashScreen = new SplashScreen();
        splashScreen.splash();
        System.out.println(Time.mark("t1 after {0}ms"));
        try {
            Thread.sleep(500);
        }
        catch (InterruptedException ie) {}
        splashScreen.splashFor(1000);
        System.out.println(Time.mark("t2 after {0}ms"));
        splashScreen.dispose();
    }
}
```

Calling Thread.sleep(500) simulates 500-milliseconds of start-up activity.

E.1.2 Test program 2

```
class async2 {
    public static void main(String[] args) {
        Time.mark(); // t0
        SplashScreen splashScreen = new SplashScreen();
        splashScreen.splash();
        splashScreen.waitForSplash(); // does it speed up?
        System.out.println(Time.mark("t1 after {0}ms"));
        try {
            Thread.sleep(500);
        }
        catch (InterruptedException ie) {}
        splashScreen.splashFor(1000);
        System.out.println(Time.mark("t2 after {0}ms"));
        splashScreen.dispose();
    }
}
```

Adds one new statement: wait for image loading using a MediaTracker before resuming normal start-up activity. Does this speed up the splash screen?

E.2 Results

Two platforms both running Java 2 SDK version 1.4.1:

- uniprocessor running Windows NT 4.0 (Service Pack 6a);
- dual SMP multiprocessor running Linux x86.

Both platforms load the splash image from their local filesystem, NTFS and XFS respectively.

The two tests run ten times each for each platform. Results compare the average timings.

E.2.1 Uniprocessor NT

First test *without* waitForSplash() gives output as:

```
t1 after 901ms
t1a after 411ms
t2 after 981ms
t1 after 891ms
t1a after 411ms
```

```
t2 after 981ms
t1 after 891ms
t1a after 411ms
t2 after 981ms
t1 after 901ms
t1a after 411ms
t2 after 981ms
t1 after 891ms
t1a after 411ms
t2 after 981ms
t1 after 912ms
t1a after 410ms
t2 after 972ms
t1 after 892ms
t1a after 410ms
t2 after 982ms
t1 after 891ms
t1a after 420ms
t2 after 982ms
t1 after 891ms
t1a after 411ms
t2 after 981ms
t1 after 891ms
t1a after 421ms
t2 after 981ms
```

Average time points, millisecond units:

t_1	t_{1a}	t_2
895.2	412.7	980.3

In summary, 895.2ms elapses before application start-up can begin. 412.7ms later the splash screen paints itself *asynchronously*. That is, *during* normal start-up activity but in another thread of execution. It disappears nearly a second later.

Second test *with* waitForSplash() gives:

```
t1a after 40ms
t1 after 2,053ms
t1a after 631ms
t2 after 330ms
t1a after 1,322ms
t1 after 60ms
t2 after 921ms
t1a after 1,252ms
t1 after 60ms
```

```
t2 after 921ms
t1a after 1,242ms
t1 after 60ms
t2 after 931ms
t1a after 1,262ms
t1 after 60ms
t2 after 922ms
t1a after 1,252ms
t1 after 60ms
t2 after 922ms
t1a after 1,262ms
t1 after 60ms
t2 after 912ms
t1a after 20ms
t1 after 1,252ms
t2 after 962ms
t1a after 20ms
t1 after 1,251ms
t2 after 962ms
t1a after 20ms
t1 after 1,242ms
t2 after 961ms
```

Because of the wait for splash, the paint event t_{1a} occurs *before* the start-up activity begins t_1. Sometimes an initial pre-paint makes two t_{1a} points, as in the first case above. The typical sequence of events goes t_{1a} paint, t_1 start-up begins, t_2 ready to dispose. The waitForSplash() forces the paint to the start. Average times, excluding double-paint figures:

t_{1a}	t_1	t_2
1265.33	60	934.889

Compare with the previous results. Time to splash painting of 1265.33ms versus $895.2 + 412.7 = 1307.9$ms, a small difference of only 42.57ms faster when waiting on a uniprocessor.

Also note that because of the wait, the second test takes 370.13ms longer before start-up can begin. That adds 370ms of wasted C.P.U. cycles.

E.2.2 Dual-processor Linux

The first test program gives:

```
t1 after 455ms
t1a after 364ms
t1a after 11ms
```

```
t2 after 1,008ms
t1 after 456ms
t1a after 354ms
t1a after 17ms
t2 after 1,011ms
t1 after 455ms
t1a after 373ms
t1a after 8ms
t2 after 1,012ms
t1 after 446ms
t1a after 336ms
t2 after 1,296ms
t1 after 461ms
t1a after 379ms
t1a after 14ms
t2 after 1,013ms
```

Firstly, again notice the multiple t_{1a} test points. For reasons unexplained at present, Sun's Linux JVM calls paint(Graphics) twice in quick succession. It does not affect the experiment. We take the final paint as point t_{1a}. Averages excluding outliers:

t_1	t_{1a}	t_2
454.6	361.2	1011

So splash occurs $454.6 + 361.2 = 815.8$ms after entering main. The second half of this period of 361.2ms runs asynchronously with normal start-up.

Second test program gives:

```
t1 after 16ms
t1a after 814ms
t1a after 12ms
t2 after 1,007ms
t1a after 842ms
t1 after 19ms
t1a after 6ms
t2 after 1,010ms
t1a after 827ms
t1a after 20ms
t1 after 5ms
t2 after 995ms
t1a after 816ms
t1 after 7ms
t1a after 20ms
t2 after 995ms
t1a after 790ms
```

```
t1 after 4ms
t1a after 23ms
t2 after 991ms
t1a after 791ms
t1 after 4ms
t1a after 23ms
t2 after 990ms
t1 after 4ms
t1a after 827ms
t1a after 22ms
t2 after 996ms
t1a after 767ms
t1 after 27ms
t1a after 4ms
t2 after 1,010ms
t1a after 805ms
t1 after 14ms
t1a after 12ms
t2 after 1,007ms
t1a after 816ms
t1 after 15ms
t1a after 10ms
t2 after 1,006ms
```

Therefore on average ignoring outliers:

t_{1a}	t_1	t_2
809.5	18.2	1008

Compare image loading delay with previous result. Now 809.5ms, or $815.8 - 809.5 = 6.3$ms faster when waiting.

E.3 Conclusion

In conclusion, waiting for the splash does not significantly speed up the time before it appears. Loading and displaying asynchronously is therefore advantageous because it frees the main thread for normal start-up activity.

Of course, these tests do not use highly compute-bound start-up simulation. If they did, this might subtract cycles from the asynchronous splash activity and delay it somewhat, especially on uniprocessor platforms. However, start-up might typically mix compute and input-output bound activities. Therefore we can reasonably assume the SplashScreen implementation successfully meets its requirement for minimum start-up delay even without waiting using waitForSplash() at the start.

Range of platforms is a limitation of these tests. It includes Windows and Linux, single and dual processors, but excludes non-Intel platforms.

F Follow-on load timing experiment

The first load timing experiment had shortcomings. First, it uses thread sleeping to simulate start-up activity. Clearly, this is unrealistic. Application start-up invariably comprises frenetic activity, busily consuming computing resources. Second, it does not adequately explore ranges of activity. How does splash versus start-up performance vary according to differing requirements and usage? Start-up can vary in complexity. How does performance react? The performance profile could tell much about overall performance.

Important questions remain unresolved. For example, should application software wait for the splash screen before proceeding? Does asynchronous splashing give an advantage? Possibly yes but only when assuming splash is mainly an input-bound activity. It involves locating and loading the image. Whether from network or filesystem, location and load depends on input bottlenecks mainly. Typically, computing resource far exceeds it, and therefore a large chunk of processing bandwidth lies waiting. Splashing asynchronously releases this available bandwidth. Well, such is the hypothesis! At some point in time, the image appears in main memory. When completely loaded, the graphics subsystem can render the splash screen. Meanwhile, the C.P.U. can carry some computing load. Any advantage assumes a computing load requirement exists. If the entire start-up depends on input- or output-bound steps, no substantial advantage exists unless an earlier request results in earlier delivery. On the other hand, no substantial disadvantage must arise from parallel execution.

F.1 Method

The experiment simulates compute-bound start-up using a 'simple' loop, see example below. The loop iterates an integer $i = 0...n - 1$.

```
int n;
long y = 0;
for (int i = 0; i < n; i++)
    y += Integer.parseInt("1000000");
```

The loop parses a string and sums the result to make the loop more substantial. Processors normally handle integer loops very quickly. In fact, optimisers can cleverly skip null-body loops. Parsing and summing circumvents optimisation and increases the duration of each iteration. Parsing the same number makes computational demand increase with n linearly. The result y equals $1{,}000{,}000 \times n$ but hopefully optimisations if any will not spot that!

This loop stands for the *entire* start-up activity. The experiment analyses the start and end of this activity, called t_1 and t_2. Application 'start-up latency' corresponds to t_2. This is the earliest point at which the application can actually begin. A frame.show() operation would normally appear at this point, and the user sees the final result of his application start-up request for the first time. Time t_{1a} marks the point in time when the splash screen appears for the first time. This point measures 'splash latency.' Figure 7 plots two time-lines, one each for the two threads of activity. Point t_0 marks

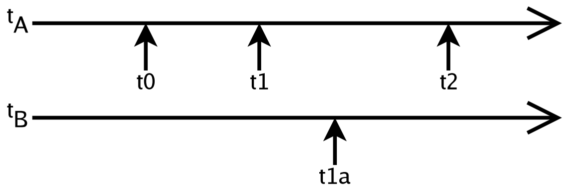

Figure 7: Main t_A and image loader t_B thread time-lines

the time origin: the point just before requesting a splash. It takes some time before Java reaches this point, however it is common to all Java software. The operating system must load the virtual machine, load the necessary initial classes and execute the initial threads. Therefore, assuming classes load on demand, measuring from previous points in time only offsets the results. The experiment can ignore this initial loading time constant.

Small addition to version 1.4 of `SplashScreen.java` signals the t_{1a} point. Difference as follows:

```
*** 285,290 ****
--- 285,291 ----
                        paint(g);
                }
                public void paint(Graphics g) {
+                       TimeSac.instance().t("t1a");
                        g.drawImage(image, 0, 0, this);
                }
                // Fixed for Mac OS X, also thanks to Werner.
```

F.1.1 Platforms

Two platforms hopefully representing different ends of the contemporary 32-bit Intel-based workstation spectrum.

Linux workstation Dual hyper-threading (four logical processors) Intel Xeon 3GHz (Intel family 15, model 2, stepping 7). SMP motherboard equipping 1GB DDR dual-channel memory. Gentoo Linux operating system running JDK version 1.4.1, Blackdown's port.

NT workstation Intel Pentium II (family 6, model 5, stepping 1) uniprocessor at 300MHz, equipped with 64MB RAM. Microsoft Windows NT 4.0 workstation operating system with Service Pack 6a, build 1381, running Java 2 runtime environment, standard edition, version 1.4.1 from Sun Microsystems. This system stretches the definition of 'contemporary.' Perhaps 'legacy' would better describe it.

F.1.2 Software

The Java test program, `async3.java`:

```
// Parse command-line arguments using gnu.getopt.Getopt, downloaded from
// ftp://ftp.urbanophile.com/pub/arenn/software/sources/java-getopt-1.0.9.jar
// Compile and execute using class path of -cp java-getopt-1.0.9.jar:.
import gnu.getopt.Getopt;

class async3 {
    public static void main(String[] args) {
        Getopt g = new Getopt("async3", args, "t:l:p:wx");
        String time_sac = "time.sac"; // filename of persistent TimeSac
        String ln_start = ""; // line start string, see -x
        boolean wait_for_splash = false; // wait for splash
        boolean xtract_stats1st = false; // extract stats 1st
        int c;
        while ((c = g.getopt()) != -1)
            switch (c) {
            case 't':
                time_sac = g.getOptarg();
                break;
            case 'l':
                ln_start = g.getOptarg();
                break;
            case 'p':
                Thread.currentThread().setPriority(Integer.parseInt(g.getOptarg()));
                break;
            case 'w':
                wait_for_splash = true;
                break;
            case 'x':
                xtract_stats1st = true;
            }
        TimeSac.instance().load(time_sac);
        if (xtract_stats1st) {
            java.util.Iterator it = TimeSac.instance().iterator();
            while (it.hasNext()) {
                java.util.Map.Entry me = (java.util.Map.Entry)it.next();
                StatisticalAccumulator sac = (StatisticalAccumulator)me.getValue();
                if (ln_start.length() != 0) System.out.print(ln_start);
                System.out.println(me.getKey() +
                          "," + sac.n() +
                          ',' + sac.min() +
                          ',' + sac.max() +
                          ',' + sac.average() +
                          ',' + sac.stDev());
            }
        }
        if (g.getOptind() == args.length) System.exit(0);
        int n = Integer.parseInt(args[g.getOptind()]);

        Time.mark(); // <-- t0 now
        SplashScreen.instance().splash();
        if (wait_for_splash)
            SplashScreen.instance().waitForSplash();

        TimeSac.instance().t("t1");
        long y = 0;
        for (int i = 0; i < n; i++)
            y += Integer.parseInt("1000000");
        TimeSac.instance().t("t2");

        TimeSac.instance().save(time_sac);
        SplashScreen.instance().dispose();
    }
}
```

The program's command-line argument specifies the number of required start-up loop iterations. For example

```
java async3 1000000
```

requests a million loop iterations. Meanwhile the splash screen loads asynchronously via the filesystem. Adding the `-w` option makes the test program wait for the image load, i.e. uses a MediaTracker before entering the start-up loop. For example

```
java async3 -w 1000000
```

waits for the splash, then runs one million loop iterations.

The experiment makes use of classes StatisticalAccumulator and TimeSac. See listings in Appendices G and H. Used together, they accumulate timing statistics.

The Linux box uses the following `bash` shell script. It accumulates timing statistics for iterations between 0 and $10{,}000{,}000$ in steps of $1{,}000{,}000$.

```
x=0
while [ $x -lt 10000000 ]
do
  rm -f timesac$x
  i=0
  while [ $i -lt 10 ]
    do
    java -cp java-getopt-1.0.9.jar:. async3 -t timesac$x $@ $x
    i=‘expr $i + 1‘
  done
  java -cp java-getopt-1.0.9.jar:. async3 -t timesac$x -l $x, -x
  x=‘expr $x + 1000000‘
done
```

Note: for the Windows NT workstation, the class path must become

```
    -cp "java-getopt-1.0.9.jar;."
```

because Windows uses semicolon to separate paths.

The following `bash` command runs the asynchronous experiment.

```
sh async3.sh | tee async3.out
```

This one runs the synchronous experiment. Option `-w` introduces the wait for splash.

```
sh async3.sh -w | tee async3.out-w
```

F.2 Results

F.2.1 Linux workstation

Linux results see Figures 8 through 10. Raw output looks like this for asynchronous splash:

```
0,t1,10,153.0,159.0,155.2,1.9321835661585918
0,t2,10,153.0,159.0,155.3,2.057506581601462
1000000,t1a,10,312.0,367.0,331.1,16.9865960228777
1000000,t1,10,152.0,185.0,157.8,9.795690662508466
1000000,t2,10,440.0,489.0,458.3,14.952145888355513
2000000,t1a,10,309.0,355.0,331.0,16.478942792411033
2000000,t1,10,152.0,160.0,154.6,3.1340424729448424
2000000,t2,10,693.0,1044.0,741.7,106.8082913968345
3000000,t1a,10,309.0,367.0,324.7,18.720458209017096
3000000,t1,10,152.0,168.0,154.5,4.904646323187388
3000000,t2,10,947.0,1011.0,965.6,19.466210040306596
4000000,t1a,10,316.0,347.0,330.4,12.851286144022923
4000000,t1,10,154.0,157.0,155.8,1.1352924243950935
4000000,t2,10,1212.0,1755.0,1278.9,167.58841779125962
5000000,t1a,10,306.0,344.0,320.9,14.563653387800741
5000000,t1,10,152.0,157.0,154.7,1.636391694484477
5000000,t2,10,1456.0,1493.0,1470.9,14.216187955988763
6000000,t1a,10,311.0,386.0,325.8,22.59695161348588
6000000,t1,10,152.0,193.0,157.8,12.497110777206776
6000000,t2,10,1719.0,1782.0,1733.9,18.44782432218559
7000000,t1a,10,307.0,356.0,329.2,15.739546795677864
7000000,t1,10,152.0,179.0,156.1,8.171087238958268
7000000,t2,10,1975.0,2017.0,1992.5,14.983324063771697
8000000,t1a,10,307.0,335.0,322.5,11.007573150638912
8000000,t1,10,152.0,156.0,153.8,1.6193277068654826
8000000,t2,10,2226.0,2247.0,2238.4,8.934328302800509
9000000,t1a,10,309.0,371.0,328.1,18.33303030052588
9000000,t1,10,154.0,186.0,159.2,9.600925881277169
9000000,t2,10,2487.0,2542.0,2504.2,16.55831446065021
```

The columns contain loop iterations, measurement, samples accumulated, minimum elapsed time from t_0 in millisecond units, maximum time, average and standard deviation. For synchronous splash, raw output is:

```
0,t1a,1,308.0,308.0,308.0,NaN
0,t1,10,287.0,360.0,303.6,29.549205667082756
0,t2,10,287.0,360.0,303.8,29.430898351524675
1000000,t1a,10,304.0,346.0,313.9,15.132378824523554
1000000,t1,10,286.0,329.0,296.6,15.042162965034871
1000000,t2,10,564.0,607.0,575.5,14.615440845595836
2000000,t1a,10,304.0,311.0,307.8,2.250925735484551
2000000,t1,10,285.0,294.0,290.0,2.211083193570267
2000000,t2,10,821.0,836.0,826.2,4.4671641514002545
3000000,t1a,10,301.0,393.0,315.3,27.48353042177159
3000000,t1,10,285.0,374.0,297.7,27.031052102679574
3000000,t2,10,1075.0,1169.0,1090.4,27.969030492075813
4000000,t1a,10,303.0,344.0,313.5,14.946199814296907
4000000,t1,10,286.0,324.0,295.5,14.713938969562161
4000000,t2,10,1330.0,1379.0,1345.6,18.745073426844126
5000000,t1a,10,305.0,368.0,314.1,19.069754994638906
5000000,t1,10,287.0,350.0,296.1,19.017243637171912
5000000,t2,10,1589.0,1653.0,1599.6,19.409047145883054
6000000,t1a,10,304.0,336.0,312.5,10.40566085252531
6000000,t1,10,284.0,321.0,294.6,11.137524161340547
6000000,t2,10,1847.0,1879.0,1856.1,10.577229210798912
7000000,t1a,10,302.0,352.0,310.8,14.718091663738958
7000000,t1,10,286.0,335.0,293.4,14.773850773128402
7000000,t2,10,2099.0,2145.0,2109.9,13.510900948657882
8000000,t1a,10,302.0,361.0,315.0,17.84501175243223
8000000,t1,10,285.0,343.0,296.4,17.939404176901256
8000000,t2,10,2352.0,2418.0,2367.8,19.848313670323623
9000000,t1a,10,303.0,363.0,311.7,18.16009055288241
9000000,t1,10,284.0,345.0,293.9,18.16865432551349
9000000,t2,10,2611.0,2671.0,2622.7,17.619749020787882
```

In all Figures, the continuous line represents asynchronous timings, those *without* waiting for splash before start-up using a MediaTracker. The dashed lines represent waiting for splash, or synchronous runs.

First note that when not waiting, start-up t_1 begins much sooner compared to waiting, as you might expect. See Figure 8, 150ms versus 300ms on average. Figure 9 compares splash latency, while Figure 10 compares start-up latency. Synchronous splashing time t_{1a}, the dashed line of Figure 9, hovers just around 310ms. Asynchronous comes just a little later around 325ms. No significant difference. The small overhead may relate to thread switching and synchronisation. This confirms the previous experiment conclusions. Also note $t_{1a} = 0$ at loop iterations $n = 0$. So event t_{1a} did not

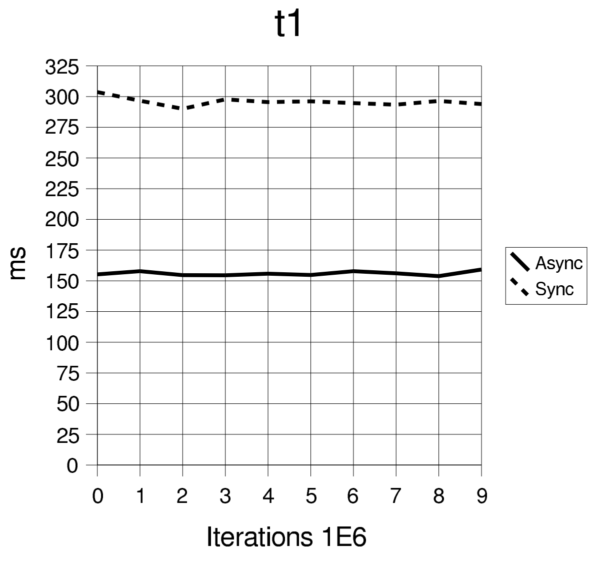

Figure 8: t_1 comparison, Linux workstation

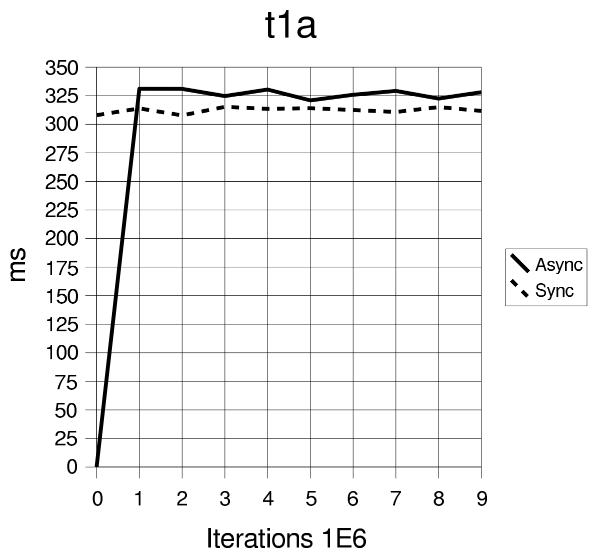

Figure 9: t_{1a} comparison, Linux workstation

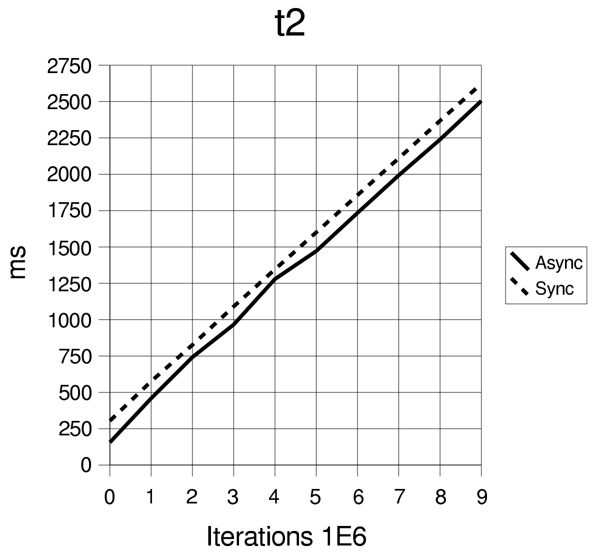

Figure 10: t_2 comparison, Linux workstation

occur with zero iterations. The test run terminated *before* the splash appeared. This is correct behaviour for asynchronous splashing.

Finally, for the Linux runs, note t_2 occurs consistently sooner when not waiting. Thus, asynchronous splashing gives better start-up latency.

F.2.2 NT workstation

Asynchronous results:

```
0,t1,10,671.0,721.0,680.0,15.238839267549947
0,t2,10,671.0,721.0,680.0,15.238839267549947
1000000,t1,10,671.0,701.0,680.0,11.972189997378647
1000000,t2,10,2744.0,2784.0,2759.0,13.5400640077266
2000000,t1,10,671.0,701.0,678.0,10.593499054713803
2000000,t2,10,4747.0,4897.0,4854.0,40.013886478460336
3000000,t1,10,671.0,701.0,676.0,9.7182531580755
3000000,t2,10,6849.0,6880.0,6860.6,10.002221975363497
4000000,t1,10,671.0,701.0,679.0,10.327955589886445
4000000,t2,10,8822.0,8923.0,8875.9,29.28196259360587
5000000,t1,10,671.0,701.0,679.0,10.327955589886445
5000000,t2,10,10875.0,10925.0,10888.2,15.70421033424547
6000000,t1,10,671.0,711.0,679.0,13.165611772087667
6000000,t2,10,12878.0,12949.0,12899.7,23.800326795142027
7000000,t1,10,670.0,701.0,677.9,9.573690801125528
7000000,t2,10,14881.0,14941.0,14897.3,16.350671070156245
8000000,t1,10,671.0,701.0,678.0,10.593499054713803
```

```
8000000,t2,10,16874.0,16965.0,16904.4,28.170906978654415
9000000,t1,10,671.0,711.0,678.0,13.374935098492585
9000000,t2,10,18897.0,18967.0,18916.2,20.208908926510603
```

Notice absence of t_{1a} events, no splashes! Synchronous results:

```
0,t1,10,961.0,992.0,967.5,9.902300518341965
0,t1a,10,951.0,992.0,960.4,14.60745623911904
0,t2,10,961.0,992.0,967.5,9.902300518341965
1000000,t1,10,961.0,1001.0,968.3,12.428014948315582
1000000,t1a,10,951.0,981.0,957.3,9.580651798749859
1000000,t2,10,3024.0,3064.0,3029.3,12.570247058475466
2000000,t1,10,961.0,1011.0,972.4,17.20594212603438
2000000,t1a,10,951.0,991.0,961.4,14.841383583300672
2000000,t2,10,5018.0,5067.0,5033.2,15.838069467092396
3000000,t1,10,961.0,992.0,970.5,10.124228365658293
3000000,t1a,10,951.0,982.0,962.5,10.013879257199868
3000000,t2,10,7020.0,7081.0,7033.2,17.319225027568514
4000000,t1a,10,951.0,992.0,961.3,13.589620221984784
4000000,t1,10,961.0,982.0,969.3,9.393259994982218
4000000,t2,10,9013.0,9073.0,9036.0,15.670212364724211
5000000,t1,10,961.0,991.0,969.2,10.3794669098819
5000000,t1a,10,951.0,981.0,961.2,10.549354903921325
5000000,t2,10,11026.0,11065.0,11034.9,11.685223337379755
6000000,t1,10,961.0,1001.0,971.3,12.211561006776416
6000000,t1a,10,951.0,981.0,961.3,10.231215850414738
6000000,t2,10,13018.0,13079.0,13033.6,19.068298298484844
7000000,t1,10,961.0,1001.0,970.3,12.815355372885035
7000000,t1a,10,951.0,981.0,959.3,10.29616973010406
7000000,t2,10,15021.0,15071.0,15034.5,14.766704288891125
8000000,t1a,10,951.0,992.0,961.4,14.229859060752812
8000000,t1,10,961.0,991.0,969.4,10.319345371140987
8000000,t2,10,17024.0,17065.0,17034.4,13.426342266852378
9000000,t1,10,961.0,1001.0,967.4,12.447221912271562
9000000,t1a,10,951.0,991.0,959.4,13.074147518417151
9000000,t2,10,19017.0,19077.0,19032.4,16.439789130845526
```

Figures 11 through 13 plot these data graphically.

Now the picture differs. Asynchronous splashing still gives faster t_1 and t_2, but no t_{1a}! The splash screen does not appear at all, see Figure 12. Even though sufficient *time* has elapsed for the splash, it does not appear. Reasons may include not having sufficient resource to progress multiple threads simultaneously. Uniprocessor bottleneck may be starving the image loader thread. Alternatively, or on top of that, coarse-grained thread switching implementation may starve the loader.

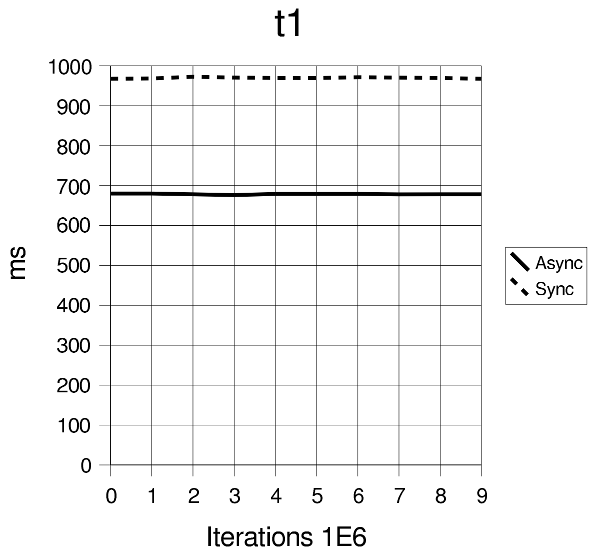

Figure 11: t_1 comparison, NT workstation

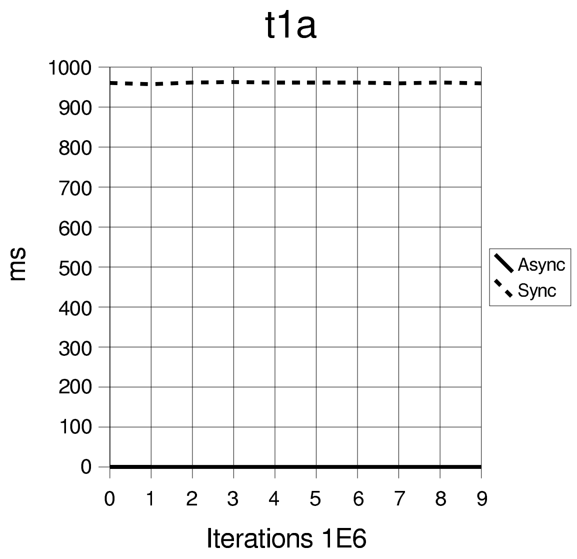

Figure 12: t_{1a} comparison, NT workstation

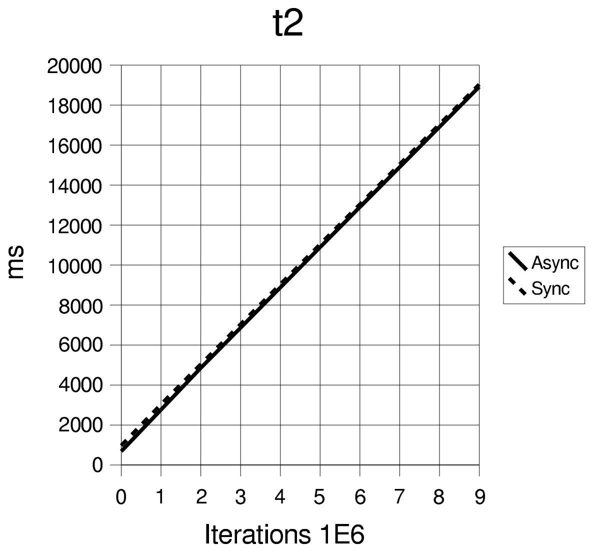

Figure 13: t_2 comparison, NT workstation

Figures 14 through 16 re-run the experiment but alter priority of the main thread to 4, one step lower. The image loader thread remains at priority 5. Java code in Appendix I dumps thread priorities.

Point t_1 shows no change. Start-up latency t_2 is now approximately identical. However, the splash screen now appears, but about 6 seconds later when asynchronously splashing. It points to multi-threading issues. Perhaps thread switching is *not* pre-emptive for threads having equal priority!

F.3 Conclusions

Based on these results:

- Current implementations of the Java virtual machine poorly handle asynchronous image loading on uniprocessor platforms.
- Waiting for the splash improves splash latency without adversely affecting start-up latency on uniprocessors.
- Multiprocessor platforms run Java threads in parallel without significant overhead. It does improve start-up latency without adversely affecting splash latency.

F.4 Solutions

Lowering main thread priority is one solution, or otherwise adjusting relative image loader versus main thread priorities. Shuffling priorities gives only partial improvement though, see Figure 15. It still takes 6 seconds longer. Curiously, start-up latency

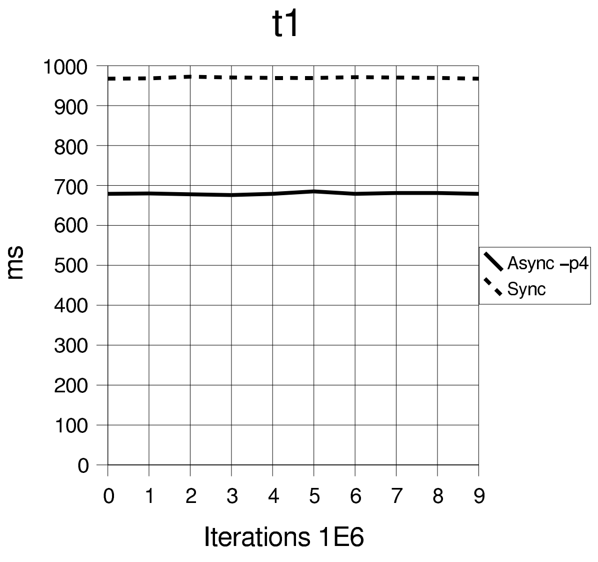

Figure 14: t_1 comparison, NT workstation, main thread priority 4

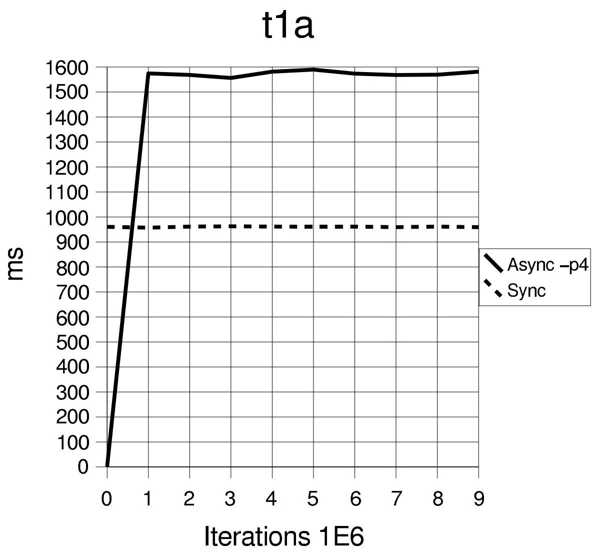

Figure 15: t_{1a} comparison, NT workstation, main thread priority 4

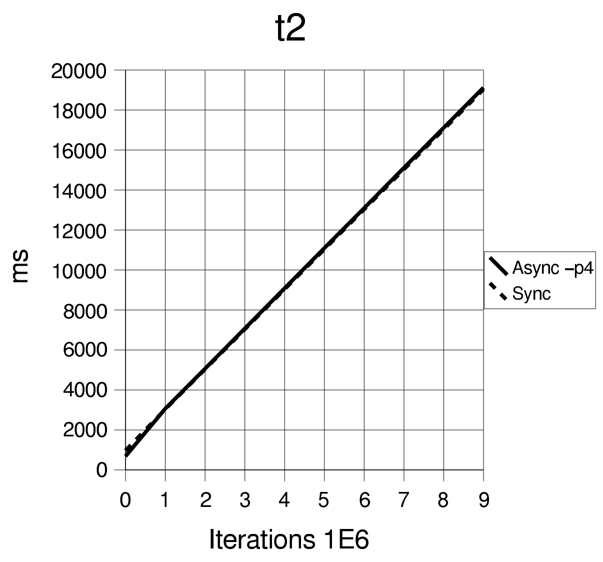

Figure 16: t_2 comparison, NT workstation, main thread priority 4

t_2 goes unaltered. Could this indicate that image loading has heavier computing load than expected? If so, it might help explain the disparities.

Pragmatic approach requires a *possible* delay just after starting the splash. Possibility depends on platform, more computing resources, less delay. Four or more processors, no delay. One processor, full delay. Java's Runtime class returns number of available processor. For example, the Linux box running these experiments gives Runtime.getRuntime().availableProcessors() equal to 4. This return value can decide how to delay, either full delay, no delay, or limited delay. Below gives one conceivable implementation.

```
public void delayForSplash() {
    int cpus = Runtime.getRuntime().availableProcessors();
    switch (cpus) {
    case 0: // pathology!
    case 1:
        waitForSplash();
        break;
    case 2:
    case 3: // ?
        waitForSplash(1000 / cpus);
    }
}
```

Extract below gives example usage. It uses the Singleton interface.

```
public static void main(String[] args) {
    SplashScreen.instance().splash();
    SplashScreen.instance().delayForSplash();
    //
    //
    //
    SplashScreen.instance().splashFor(1000);
```

```
        SplashScreen.instance().dispose();
    }
```

F.5 Further work

- Run the experiment on dual processor platforms. Also, platforms *not* based on Intel x86 32-bit architecture, uni- and multi-processors.
- Is ‘processor count’ the only factor? Try to discover other sources of bottlenecks.

G StatisticalAccumulator.java

```java
/*
 * @(#)StatisticalAccumulator.java 1.1 19-Jan-04
 *
 * Copyright (C) 2004, Roy Ratcliffe, Lancaster, United Kingdom.
 * All rights reserved.
 *
 * This software is provided ``as is'' without warranty of any kind,
 * either expressed or implied. Use at your own risk. Permission to
 * use or copy this software is hereby granted without fee provided
 * you always retain this copyright notice.
 */

import java.io.Serializable;

/**
 * Accumulates statistics.
 * Objects of the StatisticalAccumulator class emulate a
 * scientific calculator in statistical mode. You can use it to
 * compute the following statistics:
 * <ul>
 * <li> arithmetic mean (average)
 * <li> standard deviation, sample-based
 * <li> standard deviation, population-based
 * <li> minimum
 * <li> maximum
 * </ul>
 * Use the add(x) method to add data to the accumulator. This is
 * equivalent to the M+ button on a calculator. The statistics are
 * updated to reflect the new data value.
 * <p>
 * This implementation uses longs and doubles, not optimum for all
 * cases. Some usage may require narrower ranges for speed or other
 * reasons. The class has scope for expansion, a FloatSac subclass
 * for example.
 */
public class StatisticalAccumulator implements Serializable {

    private long nX;
    private double sumX;
    private double sumX2;
    private double minX;
    private double maxX;

    public void clear() {
        nX = 0;
        sumX = sumX2 = 0;
        minX = maxX = Double.NaN;
    }
    public StatisticalAccumulator add(double x) {
        nX++;
```

```
          sumX += x;
          sumX2 += x * x;
          if (nX == 1) {
              minX = x;
              maxX = x;
          }
          else {
              if (x < minX) minX = x;
              if (x > maxX) maxX = x;
          }
          return this;
      }

      public long n() {
          return nX;
      }
      public double sum() {
          if (nX == 0) return Double.NaN;
          return sumX;
      }
      public double sum2() {
          if (nX == 0) return Double.NaN;
          return sumX2;
      }
      public double min() {
          if (nX == 0) return Double.NaN;
          return minX;
      }
      public double max() {
          if (nX == 0) return Double.NaN;
          return maxX;
      }
      public double average() {
          if (nX == 0) return Double.NaN;
          return sumX / nX;
      }
      public double stDev() {
          if (nX < 2) return Double.NaN;
          return Math.sqrt((nX * sumX2 - sumX * sumX) / (nX * (nX - 1)));
      }
      public double stDevP() {
          if (nX < 1) return Double.NaN;
          return Math.sqrt((nX * sumX2 - sumX * sumX) / (nX * nX));
      }
      public double var() {
          if (nX < 1) return Double.NaN;
          return (nX * sumX2 - sumX * sumX) / (nX * nX);
      }

      public String toString() {
          return "[n=" + nX
              + ",sum=" + sumX
```

```
                + ",sum2=" + sumX2
                + ",min=" + minX
                + ",max=" + maxX
                + "]";
    }

}
```

H TimeSac.java

```java
/*
 * @(#)TimeSac.java 1.1 19-Jan-04
 *
 * Copyright (C) 2004, Roy Ratcliffe, Lancaster, United Kingdom.
 * All rights reserved.
 *
 * This software is provided ``as is'' without warranty of any kind,
 * either expressed or implied.  Use at your own risk.  Permission to
 * use or copy this software is hereby granted without fee provided
 * you always retain this copyright notice.
 */

import java.util.Map;
import java.util.HashMap;
import java.util.Iterator;
import java.io.Serializable;
import java.io.FileInputStream;
import java.io.ObjectInputStream;
import java.io.FileOutputStream;
import java.io.ObjectOutputStream;

/**
 * Statistically accumulates time samples.  Useful for performance
 * measurement.  Supports persistence between runs.
 * <pre>
 * TimeSac.instance().load("timesac1");
 * TimeSac.instance().t("t1");
 * TimeSac.instance().save("timesac1");
 * </pre>
 */
public class TimeSac implements Serializable {
    private Map x_sac_t = new HashMap();
    public void t(Object x) {
        long t = Time.at();
        StatisticalAccumulator sac;
        //      if (!x_sac_t.containsKey(x))
        //          x_sac_t.put(x, sac = new StatisticalAccumulator());
        //      else
        //          sac = (StatisticalAccumulator)x_sac_t.get(x);
        sac = (StatisticalAccumulator)x_sac_t.get(x);
        if (sac == null)
            x_sac_t.put(x, sac = new StatisticalAccumulator());
        sac.add(t);
    }
    public void load(String filename) {
        try {
            FileInputStream fis = new FileInputStream(filename);
            ObjectInputStream ois = new ObjectInputStream(fis);
            x_sac_t = (Map)ois.readObject();
        }
```

```
            catch (java.io.FileNotFoundException fnfe) {}
            catch (java.io.IOException ioe) {}
            catch (ClassNotFoundException cnfe) {}
        }
        public void save(String filename) {
            try {
                FileOutputStream fos = new FileOutputStream(filename);
                ObjectOutputStream oos = new ObjectOutputStream(fos);
                oos.writeObject(x_sac_t);
            }
            catch (java.io.IOException ioe) {}
        }
        public Iterator iterator() {
            return x_sac_t.entrySet().iterator();
        }
        public String toString() {
            StringBuffer sb = new StringBuffer();
            sb.append('{');
            Iterator it = iterator();
            boolean hasNext = it.hasNext();
            while (hasNext) {
                Map.Entry me = (Map.Entry)it.next();
                Object x = me.getKey();
                StatisticalAccumulator sac = (StatisticalAccumulator)me.getValue();
                sb.append(x +
                          ":n=" + sac.n() +
                          ",sum=" + sac.sum() +
                          ",sum2=" + sac.sum2() +
                          ",min=" + sac.min() +
                          ",max=" + sac.max() +
                          ",average=" + sac.average() +
                          ",st-dev=" + sac.stDev() +
                          ",st-dev-p=" + sac.stDevP() +
                          ",var=" + sac.var());
                hasNext = it.hasNext();
                if (hasNext) sb.append(' ');
            }
            sb.append('}');
            return sb.toString();
        }

        private static TimeSac singleton = null;
        public static synchronized TimeSac instance() {
            if (null == singleton) singleton = new TimeSac();
            return singleton;
        }
}
```

I threadtree.java

```java
/*
 * @(#)threadtree.java 1.1 20-Jan-04
 *
 * Copyright (C) 2004, Roy Ratcliffe, Lancaster, United Kingdom.
 * All rights reserved.
 *
 * This software is provided ''as is'' without warranty of any kind,
 * either expressed or implied.  Use at your own risk.  Permission to
 * use or copy this software is hereby granted without fee provided
 * you always retain this copyright notice.
 */

public class threadtree {
    public static String threadtree() {
        StringBuffer sb = new StringBuffer();

        // Java thread groups have a hierarchy.  Every thread group
        // has a parent, except the root group.
        // First, walk the thread group tree from group to group until
        // the parentless root group appears.
        ThreadGroup root = Thread.currentThread().getThreadGroup(), tg;
        while ((tg = root.getParent()) != null)
            root = tg;

        // Now, step through the tree of thread groups.  Start at the
        // root and, using a stack, push thread subgroups until the stack empties.
        java.util.Stack s = new java.util.Stack();
        s.push(root);
        while (!s.empty()) {
            tg = (ThreadGroup)s.pop();

            StringBuffer indentsb = new StringBuffer();
            ThreadGroup parent = tg;
            while ((parent = parent.getParent()) != null)
                indentsb.append('\t');
            String indent = indentsb.toString();
            sb.append(indent).append(tg).append('\n');

            // How many groups nested within this one?  Make an
            // initial estimate then enumerate each one.  Threads and
            // thread groups exist asynchronously, so every answer
            // gives a snapshot at /one/ point in time.
            int gc = tg.activeGroupCount();
            ThreadGroup[] tgs = new ThreadGroup[gc * 2];
            gc = tg.enumerate(tgs);
            for (int i = 0; i < gc; i++)
                s.push(tgs[i]);

            // Snapshot the threads currently active in this group.
            int c = tg.activeCount();
```

```
                Thread[] ts = new Thread[c * 2];
                c = tg.enumerate(ts);
                for (int i = 0; i < c; i++)
                    sb.append(indent).append(ts[i]).append('\n');

            }
            return sb.toString();
        }

        public static void main(String[] args) {
            System.out.print(threadtree());
        }
}
```