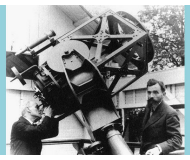
Instruments and Imaging Section
Technical Tips No. 3
20 February 2012



Director: Bob Marriott ram@hamal.demon.co.uk 01604 765190 http://britastro.org/iandi

Following the Nuclear Test Ban Treaty (1963) and the Treaty on Principles Governing the Activities of States in the Exploration and Use of Outer Space (1967), the US Vela satellites were designed to detect gamma-radiation pulses emitted by nuclear weapons tested in space. On 2 July 1967, Vela 3 and Vela 4 detected a flash of gamma-radiation unlike any known nuclear weapon's signature, and more flashes were detected later. Their cosmic origin was eventually recognised, and the discoveries – logged at Los Alamos Scientific Laboratory – were declassified and published in *Astrophysical Journal Letters* in 1973. In the previous issue of *Technical Tips*, the 'mystery object' is a ground-based automated instrument designed to 'stare' at space in order to detect explosions of nuclear weapons. Instead, it detected gamma-ray bursts.

This issue's identity challenge is, as usual, shown at top right. What is the instrument, and who are the men? The only clue I will provide is that the photograph was taken *c*.1927.

The article below relates to an aspect of observing which, like several other basic observing skills and requirements, seems to have fallen by the wayside, but nevertheless is still of importance. As is well known, dark adaption serves several purposes, and the education of the novice is not helped by television programmes showing 'observers' surrounded by lighting. Moreover, for many of us affected by light pollution, dark adaption produces the ability to see everything without a personal artificial light. I have not used a torch in my garden for several years.

The man who had no 'go to' facility

Dark-adapted software

Graham Relf

There are now many computer applications that are really useful at the telescope; but there is a problem with computer screen brightness, which cannot be reduced to anything like a level at which the dark adaption of the observer's eyes is unaffected. There are several possible approaches to solving the problem, including making the screen red. You might

- place a sheet of dark plastic – particularly a dark red one – over the screen;
- go into your operating system's settings (such as Windows Control Panel) to alter the colour scheme used for all displayed components;
- look for software that has been written to display only in dark shades (mostly red) on a black background; or
- write your own software that really makes everything dark and red.

I have noticed that the first of these is a requirement at star parties advertised on the Internet. (A further tip: stick black tape over that 'cool' but irritatingly bright LED logo on your laptop case; and probably also over those white or blue LEDs that show disc activity, caps lock status, and so on. My laptop is a patchwork of black tape.)

None of the approaches listed above is ideal, because

- it is hard to find a sufficiently large and affordable plastic sheet which is truly clear and darkens the screen by just the right amount; a red sheet is not likely to be spectrally pure red, but is quite likely to cause some colours on the screen to become indistinguishable from each other; some vital text may become invisible against its background;
- system settings tend not to allow you to change everything, and it can be hard to follow the nomenclature, to ensure that every type of component displayed by your application (and the desk-top, and so on) is darkened as required; you also risk spoiling the appearance of other applications;
- are there many applications written to truly solve this problem? (I do not know – please inform me); and
- how many of us are programmers?

On the Computing Section website we provide versions of all the asteroid and comet charts in pure red on black. If you display such a chart truly full-screen (without any window border) it should help.

As a Java programmer I write my own tools to use on my laptop at the telescope. I have worked out how to make everything red on black, including window borders. Being an unusual requirement, the details of how to do this are not well documented. It is not covered by Oracle's (formerly Sun's) Java Tutorial. Therefore, I thought I would share my findings.

One of the nice features of Java's swing graphics package is called Pluggable Look And Feel (PLAF). To make an application look right on whatever system it is running, the following is required as one of the first executable statements (unless otherwise stated, classes in my sample code are in the standard package javax.swing).

```
UIManager.setLookAndFeel
      UIManager.getSystemLookAndFeelClassName
                                                ));
```

In theory, PLAF enables completely different window styles to be used in an application. It is possible to create a new look and feel from scratch or, more practically, by subclassing

```
javax.swing.plaf.basic.BasicLookAndFeel
```

How to do this is described, up to a point, in some books. (For example, Matthew Robinson and Pavel Vorobiev: *Swing*, second edn., Manning, 2003. ISBN 1-930110-88-X. The relevant Chapter 21 of this book can be found online). However, I found this approach quite complicated and, as far as I went with it, it seemed to upset the behaviour of user interaction with tables (javax.swing.JTable). So I tried a different approach, with complete success.

The Metal Look & Feel is one of the standard PLAFs supplied with Java. It is also known as the Java Look & Feel, because it is what you get if you do not set any particular look and feel with the UIManager statement shown above. Metal has 'themability', and this proved to be an easier way to achieve my objective.

It is necessary to subclass javax.swing.plaf.metal.MetalTheme: I wrote org.britastro.grelf.ObservingTheme in which I overrode a number of extremely simple methods that return colour and font resource objects to the system (JVM). The full code of ObservingTheme is given at the end of this article. Then the following statements at the start of the main program create the required look and feel.

```
UIManager.setLookAndFeel ("javax.swing.plaf.metal.MetalLookAndFeel");
MetalLookAndFeel.setCurrentTheme (new ObservingTheme ());
UIManager.setLookAndFeel (new MetalLookAndFeel ());
```

This somewhat illogical-looking code successfully changes all buttons, text fields, drop-downs, text labels, background panels, and so on, but *not* the borders and caption bars of windows and dialogues, because those are normally the responsibility of the operating system rather than the application. However, the Java Look & Feel does allow control of those too.

Window borders and caption bars comprise the 'decoration' of the window. There are two ways to prevent them spoiling our attempt at dark-adapted software. The first way is to declare windows and dialogues to be undecorated, just after construction but before they are displayed:

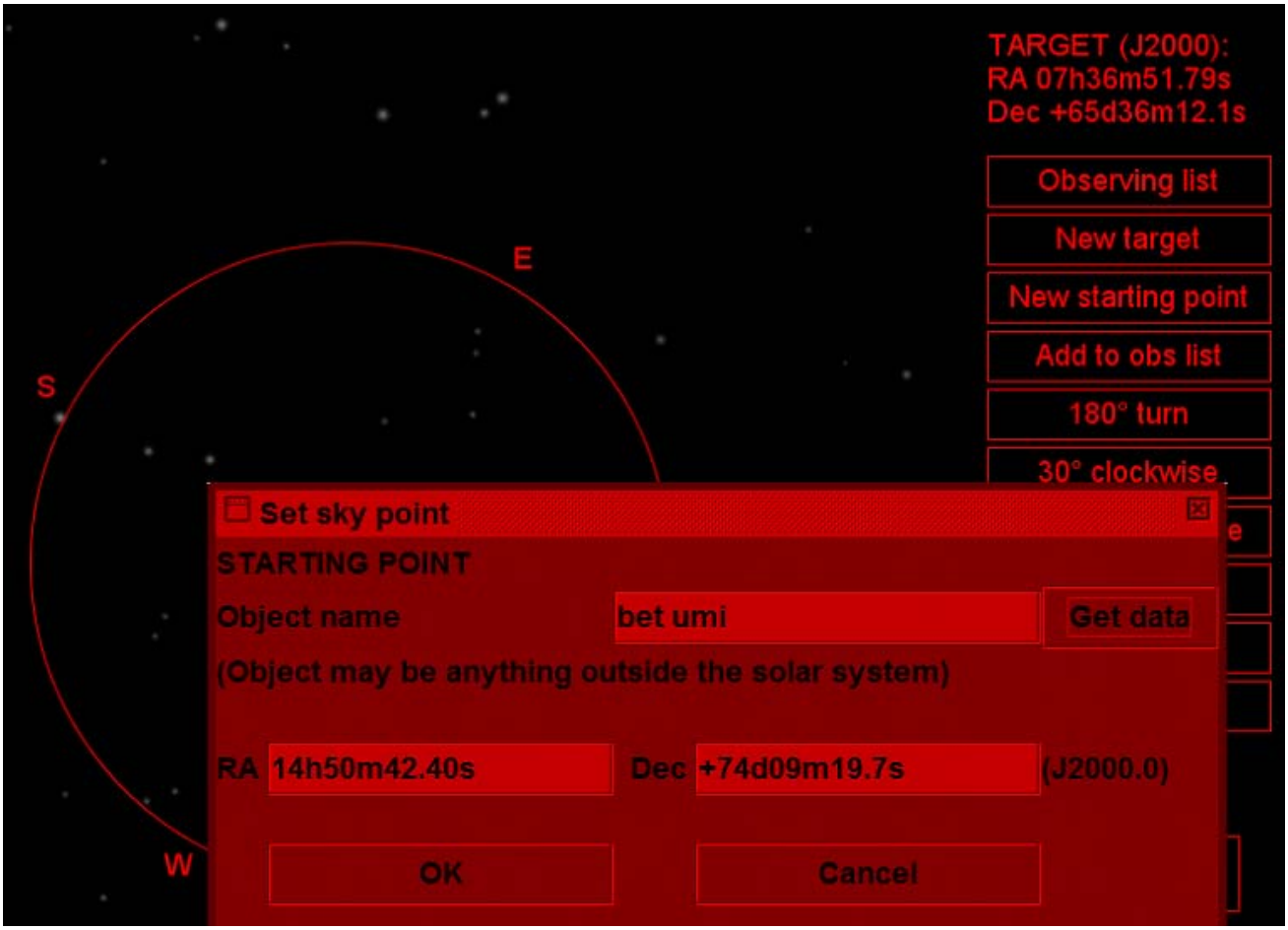
```
JFrame frame = new JFrame ();
frame.setUndecorated (true);
```

This works, but without borders or a caption bar the window can never be resized or dragged around by the user. That is generally not very clever, but it is appropriate for a background main window that deliberately fills the whole screen to blot out the light from the desk-top.

So a better way for sub-windows is to add a further statement:

```
JFrame frame = new JFrame ("Caption - will be seen now");
frame.setUndecorated (true);
frame.getRootPane().setWindowDecorationStyle (JRootPane.FRAME);
```

This magically gives the JFrame (or JDialog, or whatever) basic but functional decorations in the required red style. Here is an example from one of my observer-friendly applications.



The background is the screen-filling main window of the application. A temporary window is over that, presenting a ‘Set sky point’ dialogue to the user.

(I use this application for finding faint objects very effectively from a naked-eye starting point. It shows me a view similar to what I see using low power on my 10-inch Newtonian, rotatable to match my view. It shows me in which direction to move to the target and how far I still have to go. I do not have a ‘go to’ mount, so this program saves me an enormous amount of time.)

Finally, here is the simple subclass that I had to write:

```
package org.britastro.grelf.plaf;

// Author: Graham Relf (www.grelf.net) 2012

import java.awt.Color;
import java.awt.Font;
import javax.swing.plaf.ColorUIResource;
import javax.swing.plaf.FontUIResource;
import javax.swing.plaf.metal.MetalTheme;

/** Theme that uses only red and black, for minimal effect on dark-adapted astronomers' eyes.
  * For use with Swing's MetalLookAndFeel. */
public class ObservingTheme extends MetalTheme
{
  private static final FontUIResource FONT = new FontUIResource ("Arial", Font.BOLD, 18);
  private static final ColorUIResource LIGHT_RED = new ColorUIResource (new Color (255, 0, 0));
  private static final ColorUIResource RED = new ColorUIResource (new Color (191, 0, 0));
  private static final ColorUIResource DARK_RED = new ColorUIResource (new Color (127, 0, 0));
  private static final ColorUIResource VERY_DARK_RED = new ColorUIResource (new Color (91, 0, 0));
  private static final ColorUIResource BLACK = new ColorUIResource (Color.BLACK);

  @Override public String getName () { return "ObservingTheme"; }

  @Override protected ColorUIResource getBlack () { return BLACK; }
  @Override public    ColorUIResource getControl () { return getSecondary3 (); }
  @Override public    ColorUIResource getControlDarkShadow () { return VERY_DARK_RED; }
  @Override public    ColorUIResource getControlDisabled () { return DARK_RED; }
  @Override public    ColorUIResource getControlHighlight () { return LIGHT_RED; }
  @Override public    ColorUIResource getControlInfo () { return BLACK; }
  @Override public    ColorUIResource getControlShadow () { return DARK_RED; }
  @Override public    ColorUIResource getControlTextColor () { return getControlInfo (); }
  @Override public    ColorUIResource getDesktopColor () { return VERY_DARK_RED; }
  @Override public    ColorUIResource getFocusColor () { return getPrimary2 (); }
  @Override public    ColorUIResource getHighlightedTextColor () { return BLACK; }
  @Override public    ColorUIResource getInactiveControlTextColor ()
                                                    { return getControlDisabled (); }
  @Override public    ColorUIResource getInactiveSystemTextColor () { return VERY_DARK_RED; }
  @Override public    ColorUIResource getPrimaryControl () { return RED; }
  @Override public    ColorUIResource getPrimaryControlShadow () { return DARK_RED; }
  @Override public    ColorUIResource getPrimaryControlDarkShadow () { return VERY_DARK_RED; }
  @Override public    ColorUIResource getPrimaryControlInfo () { return BLACK; }
  @Override public    ColorUIResource getPrimaryControlHighlight () { return LIGHT_RED; }
  @Override protected ColorUIResource getPrimary1 () { return RED; }
  @Override protected ColorUIResource getPrimary2 () { return RED; }
  @Override protected ColorUIResource getPrimary3 () { return RED; }
  @Override protected ColorUIResource getSecondary1 () { return DARK_RED; }
  @Override protected ColorUIResource getSecondary2 () { return DARK_RED; }
  @Override protected ColorUIResource getSecondary3 () { return DARK_RED; }
  @Override public    ColorUIResource getSystemTextColor () { return BLACK; }
  @Override public    ColorUIResource getTextHighlightColor () { return LIGHT_RED; }
  @Override public    ColorUIResource getUserTextColor () { return BLACK; }
  @Override protected ColorUIResource getWhite () { return LIGHT_RED; }
  @Override public    ColorUIResource getWindowBackground () { return RED; }
  @Override public    ColorUIResource getWindowTitleBackground () { return getPrimary3 (); }
  @Override public    ColorUIResource getWindowTitleForeground () { return getBlack (); }
  @Override public    ColorUIResource getWindowTitleInactiveBackground ()
                                      { return getSecondary3 (); }
  @Override public    ColorUIResource getWindowTitleInactiveForeground() { return VERY_DARK_RED; }

  @Override public FontUIResource getControlTextFont () { return FONT; }
  @Override public FontUIResource getMenuTextFont () { return FONT; }
  @Override public FontUIResource getSubTextFont () { return FONT; }
  @Override public FontUIResource getSystemTextFont () { return FONT; }
  @Override public FontUIResource getUserTextFont () { return FONT; }
  @Override public FontUIResource getWindowTitleFont () { return FONT; }

} // ObservingTheme
```

It is not clear in the code of MetalTheme what Primary1, and so on, are used for. To really check this it would be necessary to examine Oracle's source code for many other classes. It is readily available, but I found I could do what I needed without going that far.