IDL®

Excerpts From:

Working with Images in IDL

RESEARCH SYSTEMS
A Kodak Company

March, 2002 Edition

Acknowledgments

IDL® is a registered trademark of Research Systems Inc., registered in the United States Patent and Trademark Office, for the computer program described herein. Software ≡ Vision™ is a trademark of Research Systems, Inc.

Numerical Recipes™ is a trademark of Numerical Recipes Software. Numerical Recipes routines are used by permission.

GRG2™ is a trademark of Windward Technologies, Inc. The GRG2 software for nonlinear optimization is used by permission.

NCSA Hierarchical Data Format (HDF) Software Library and Utilities
Copyright © 1988-1998 The Board of Trustees of the University of Illinois
All rights reserved.

CDF Library
Copyright © 1999
National Space Science Data Center
NASA/Goddard Space Flight Center

NetCDF Library
Copyright © 1993-1996 University Corporation for Atmospheric Research/Unidata

HDF EOS Library
Copyright © 1996 Hughes and Applied Research Corporation

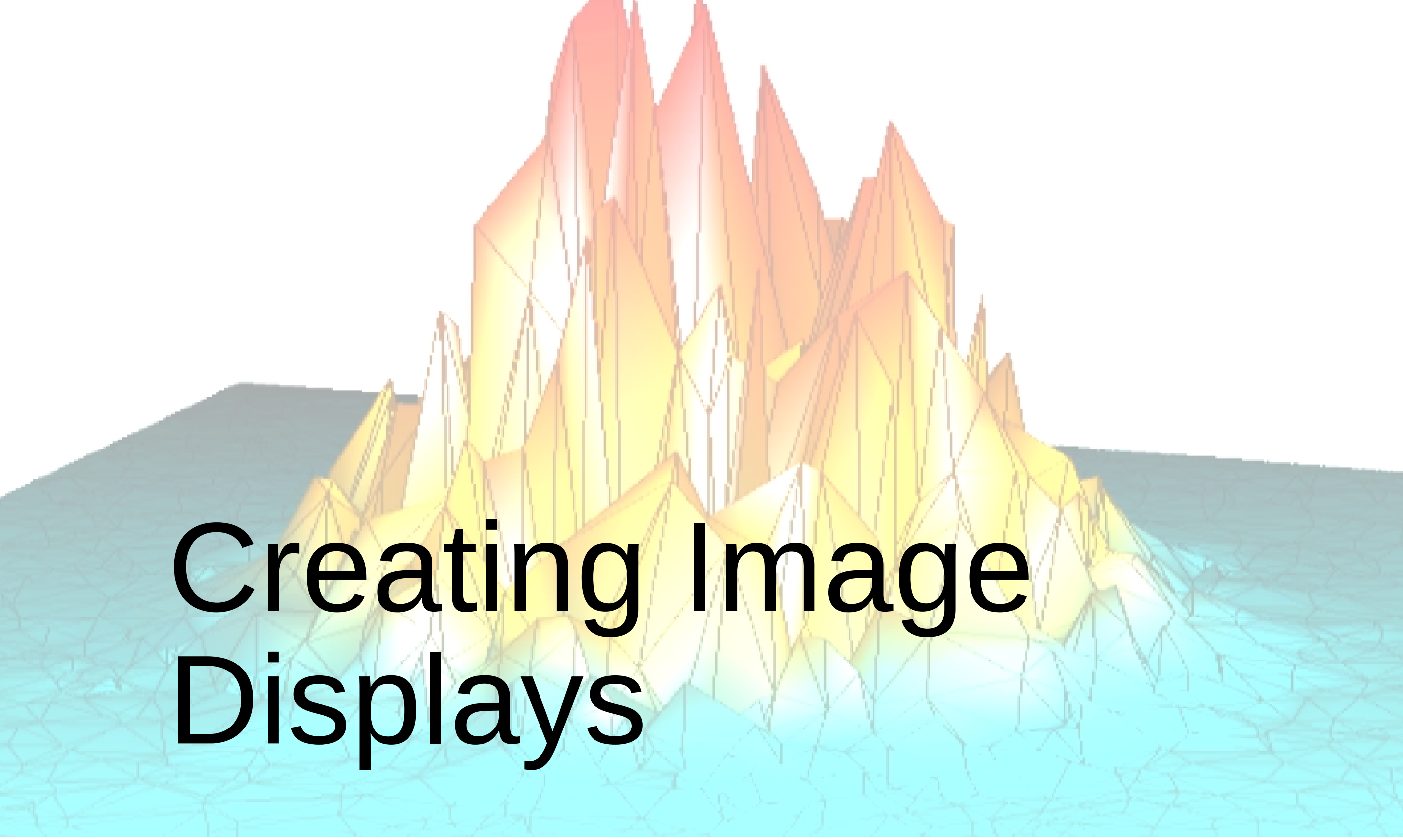
This software is based in part on the work of the Independent JPEG Group.

This product contains StoneTable™, by StoneTablet Publishing. All rights to StoneTable™ and its documentation are retained by StoneTablet Publishing, PO Box 12665, Portland OR 97212-0665. Copyright © 1992-1997 StoneTablet Publishing

WASTE text engine © 1993-1996 Marco Piovanelli

Portions of this software are copyrighted by INTERSOLV, Inc., 1991-1998.

Other trademarks and registered trademarks are the property of the respective trademark holders.



Creating Image Displays

This chapter describes the following topics:

Overview of Creating Image Displays

To understand how to display an image, you must understand IDL's graphics systems, window coordinate systems, and the types of images you can display. IDL has two types of graphics systems, Direct and Object. Direct Graphics displays directly to a window. Object Graphics uses object-oriented programming to display within instances of window objects. For more information, see "Differentiating Between Graphics Systems" on page 4

One of IDL's three window coordinate systems controls the placement of the image within Direct and Object Graphics displays. These window coordinate systems are data, device, and normalized. Data coordinates are the same as device coordinates for images. Device coordinates are based on the pixel locations of an image. Normalized coordinates range from zero and one and are based on the width and height of the image. See "Understanding Windows and Related Device Coordinates" on page 5 for more information.

IDL can display four types of images; binary, grayscale, indexed, and RGB. IDL treats binary and grayscale images as a subsets of indexed images. How the image is displayed depends on its type. Binary images have only two values, 0 and 1. Grayscale images represent intensities and use a normal grayscale color table. Indexed images use an associated color table. RGB images contain their own color information in layers known as bands or channels.Any of these images can be displayed with Direct Graphics, see "Creating Direct Graphics Image Displays" on page 7, or Object Graphics, see "Creating Object Graphics Image Displays" on page 21.

Multiple images can also be displayed in the same. The location of the images depends on the window coordinate system. For more information, see "Displaying Multiple Images in a Window" on page 37.

You can magnify specific areas of an image by changing the display show just that region. This magnification is known as zooming in on the image. By zooming in on an area, you can visualize each pixel that make up a feature or a boundary. See "Zooming in on an Image" on page 48 for more information.

When you zoom in on a feature within an image, you may want to move along the feature at that magnification. The movement is known as panning. For more information on panning, see "Panning within an Image" on page 54.

The following list introduces image display tasks and associated IDL image processing routines covered in this chapter.

Task	Routine(s)	Description
"Creating Direct Graphics Image Displays" on page 7.	TV TVSCL	Display binary, grayscale, indexed, and RGB images using the Direct Graphics system.
"Creating Object Graphics Image Displays" on page 21	IDLgrImage IDLgrPalette	Display binary, grayscale, indexed, and RGB images using the Object Graphics system.
"Displaying Multiple Images in a Window" on page 37.	TV TVSCL IDLgrImage	Display multiple images in a single Direct Graphics and Object Graphics window.
"Zooming in on an Image" on page 48.	ZOOM ZOOM_24 IDLgrImage IDLgrView	Magnify specific areas of an image using Direct and Object Graphics.
"Panning within an Image" on page 54.	SLIDE_IMAGE IDLgrImage IDLgrView	Zoom in on specific areas of an image and then move to another area within the image using Direct and Object Graphics.

Table 1: Image Display Tasks and Related Image Display Routines.

Note

This chapter uses data files from the `IDL x.x/examples/data` directory. Two files, `data.txt` and `index.txt`, contain descriptions of the files, including array sizes.

Differentiating Between Graphics Systems

IDL supports two distinct graphics modes: Direct Graphics and Object Graphics. Direct Graphics rely on the concept of a current graphics device; IDL commands like TV or PLOT create displays directly on the current graphics device. Object Graphics use an object-oriented programmer' interface to create graphic objects, which must then be drawn, explicitly, to a destination of the programmer's choosing.

Direct Graphics

The important aspects of Direct Graphics are:

- Direct Graphics use a graphics device (**X** for X-windows systems displays, **WIN** for Microsoft Windows displays, **MAC** for Macintosh displays, **PS** for PostScript files, etc.). You switch between graphics devices using the SET_PLOT command, and control the features of the current graphics device using the DEVICE command.
- Commands like TV, PLOT, XYOUTS, MAP_SET, etc. all draw their output directly on the current graphics device.
- Once a direct-mode graphic is drawn to the graphics device, it cannot be altered or re-used. This means that if you wish to recreate the graphic on a different device, you must reissue the IDL commands to create the graphic.
- When you add a new item to an existing direct-mode graphic (using a routine like OPLOT or XYOUTS), the new item is drawn in front of the existing items.

Object Graphics

The important aspects of Object Graphics are:

- Object graphics are device independent. There is no concept of a current graphics device when using object-mode graphics; any graphics object can be displayed on any physical device for which a destination object can be created.
- Object graphics are object-oriented. Graphic objects are meant to be created and reused; you may create a set of graphic objects, modify their attributes, draw them to a window on your computer screen, modify their attributes again, then draw them to a printer device without reissuing all of the IDL commands used to create the objects. Graphics objects also encapsulate functionality; this means that individual objects include method routines that provide functionality specific to the individual object.

- Object graphics are rendered in three dimensions. Rendering implies many operations not needed when drawing Direct Graphics, including calculation of normal vectors for lines and surfaces, lighting considerations, and general object overhead. As a result, the time needed to render a given object—a surface, say—will often be longer than the time taken to draw the analogous image in Direct Graphics.
- Object Graphics use a programmer's interface. Unlike Direct Graphics, which are well suited for both programming and interactive, ad hoc use, Object Graphics are designed to be used in programs that are compiled and run. While it is still possible to create and use graphics objects directly from the IDL command line, the syntax and naming conventions make it more convenient to build a program offline than to create graphics objects on the fly.
- Because Object Graphics persist in memory, there is a greater need for the programmer to be cognizant of memory issues and memory leakage. Efficient design—remembering to destroy unused object references and cleaning up—will avert most problems, but even the best designs can be memory-intensive if large numbers of graphic objects (or large datasets) are involved.

Understanding Windows and Related Device Coordinates

Images are displayed within a window (Direct Graphics) or within an instance of a window object (Object Graphics). In Direct Graphics, the WINDOW procedure is used to initialize the coordinates system for the image display. In Object Graphics, the IDLgrWindow, IDLgrView, and IDLgrModel objects are used to initialize the coordinate system for the image display.

A coordinate system determines how and where the image appears within the window. You can specify coordinates to IDL using one of the following coordinate systems:

- Data Coordinates - This system usually spans the window with a range identical to the range of the data. The system can have two or three dimensions and can be linear, logarithmic, or semi-logarithmic.
- Device Coordinates - This coordinate system is the physical coordinate system of the selected device. Device coordinates are integers, ranging from (0, 0) at the bottom-left corner to (V_x –1, V_y –1) at the upper-right corner of the display. V_x and V_y are the number of columns and rows of the device (a display window for example).

Note

For images, the data coordinates are the same as the device coordinates. The devices coordinates of an image are directly related to the pixel (data) locations within an image.

- Normal Coordinates - The normalized coordinate system ranges from zero to one over columns and rows of the device.

Creating Direct Graphics Image Displays

The procedure used to display an image in Direct Graphics depends upon the type of image to be displayed. Binary, grayscale, and indexed images are two-dimensional arrays. In Direct Graphics, these images are displayed with the TV or TVSCL procedures. The TV procedure displays the image in its original form. The TVSCL procedure displays the image scaled to range from 0 up to 255 depending on the colors available to IDL. Three dimensional RGB images are displayed with the TV procedure.

Examples of creating such displays are shown in the following sections:

- "Displaying Binary Images with Direct Graphics".
- "Displaying Grayscale Images with Direct Graphics" on page 10.
- "Displaying Indexed Images with Direct Graphics" on page 12.
- "Displaying RGB Images with Direct Graphics" on page 16.

Displaying Binary Images with Direct Graphics

Binary images are composed of pixels having one of two values, usually zero or one. With most color tables, pixels having values of zero or one are displayed with almost the same color, especially with a normal grayscale color table. Thus, a binary image is usually shown in a scaled display to show the zeros as black and the ones as white.

The following example imports a binary image of the world from the `continent_mask.dat` binary file. In this image, the oceans are zeros (black) and the continents are ones (white). This type of image can be used to mask out (omit) data over the oceans. The image contains byte data values and is 360 pixels by 360 pixels.

For code that you can copy and paste into an Editor window, see "Example Code: Displaying Binary Images with Direct Graphics" on page 9 or complete the following steps for a detailed description of the process.

1. Determine the path to the `continent_mask.dat` file:

```
file = FILEPATH('continent_mask.dat', $
   SUBDIRECTORY = ['examples', 'data'])
```

2. Initialize the image size parameter:

```
imageSize = [360, 360]
```

3. Use READ_BINARY to import the image from the file:

   ```
   image = READ_BINARY(file, DATA_DIMS = imageSize)
   ```

4. If you are running IDL on a TrueColor display, set the DECOMPOSED keyword to the DEVICE command to zero before your first color table related routine is used within an IDL session or program. See “How Colors are Associated with Indexed and RGB Images” on page 84 for more information.

   ```
   DEVICE, DECOMPOSED = 0
   ```

5. Load a grayscale color table:

   ```
   LOADCT, 0
   ```

6. Create a window and display the original image with the TV procedure:

   ```
   WINDOW, 0, XSIZE = imageSize[0], YSIZE = imageSize[1], $
      TITLE = 'A Binary Image, Not Scaled'
   TV, image
   ```

   The resulting window should be all black (blank). The binary image contains zeros and ones, which are almost the same color (black). Binary images should be displayed with the TVSCL procedure in order to scale the ones to white.

7. Create another window and display the scaled binary image:

   ```
   WINDOW, 1, XSIZE = imageSize[0], YSIZE = imageSize[1], $
      TITLE = 'A Binary Image, Scaled'
   TVSCL, image
   ```

The following figure shows the results of scaling this display.

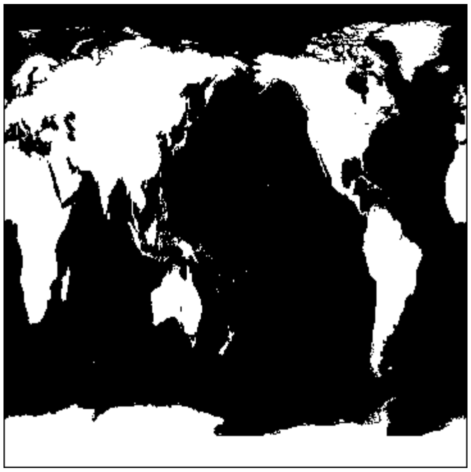


Figure 1: Binary Image in Direct Graphics

Example Code: Displaying Binary Images with Direct Graphics

Copy and paste the following text into the IDL Editor window. After saving the file as `DisplayBinaryImage_Direct.pro`, compile and run the program to reproduce the previous example.

```
PRO DisplayBinaryImage_Direct

; Determine the path to the file:
file = FILEPATH('continent_mask.dat', $
   SUBDIRECTORY = ['examples', 'data'])

; Initialize the image size parameter.
imageSize = [360, 360]

; Import in the image from the file.
image = READ_BINARY(file, DATA_DIMS = imageSize)

; Initialize the display,
DEVICE, DECOMPOSED = 0
LOADCT, 0

; Create a window and display the original image.
WINDOW, 0, XSIZE = imageSize[0], YSIZE = imageSize[1], $
```

```
   TITLE = 'A Binary Image, Not Scaled'
TV, image

; Create another window and display the image scaled
; to range from 0 up to 255.
WINDOW, 1, XSIZE = imageSize[0], YSIZE = imageSize[1], $
   TITLE = 'A Binary Image, Scaled'
TVSCL, image

END
```

Displaying Grayscale Images with Direct Graphics

Features within grayscale images are created by pixels that have varying intensities. Pixel values range from least intense (black) to the most instance (white). Since a grayscale image is composed of pixels of varying intensities, it is best displayed with a color table that progresses linearly from black to white. Although IDL has several such predefined color tables, the grayscale color table (B-W LINEAR), is the most fitting choice when displaying grayscale images.

The following example imports a grayscale image from the `convec.dat` binary file. This grayscale image shows the convection of the Earth’s mantle. The image contains byte data values and is 248 pixels by 248 pixels. Since the data type is byte, this image does not need to by scaled before display. If the data was of any type other than byte and the data values were not within the range of 0 up to 255, the display would need to scale the image in order to show its intensities.

For code that you can copy and paste into an Editor window, see “Example Code: Displaying Grayscale Images with Direct Graphics” on page 11 or complete the following steps for a detailed description of the process.

1. Determine the path to the `convec.dat` file:

   ```
   file = FILEPATH('convec.dat', $
      SUBDIRECTORY = ['examples', 'data'])
   ```

2. Initialize the image size parameter:

   ```
   imageSize = [248, 248]
   ```

3. Using READ_BINARY, import the image from the file:

   ```
   image = READ_BINARY(file, DATA_DIMS = imageSize)
   ```

4. If you are running IDL on a TrueColor display, set the DECOMPOSED keyword to the DEVICE command to zero before your first color table related routine is used within an IDL session or program. See “How Colors are Associated with Indexed and RGB Images” on page 84 for more information.

```
DEVICE, DECOMPOSED = 0
```

5. Load a grayscale color table:

```
LOADCT, 0
```

6. Create a window and display the original image with the TV procedure:

```
WINDOW, 0, XSIZE = imageSize[0], YSIZE = imageSize[1], $
   TITLE = 'A Grayscale Image'
TV, image
```

The following figure shows the resulting grayscale image display.

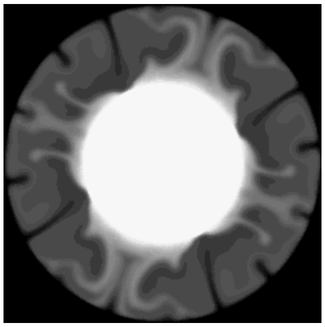


Figure 2: Grayscale Image in Direct Graphics

Example Code: Displaying Grayscale Images with Direct Graphics

Copy and paste the following text into the IDL Editor window. After saving the file as `DisplayGrayscaleImage_Direct.pro`, compile and run the program to reproduce the previous example.

```
PRO DisplayGrayscaleImage_Direct

; Determine the path to the file.
file = FILEPATH('convec.dat', $
   SUBDIRECTORY = ['examples', 'data'])

; Initialize the image size parameter.
```

```
imageSize = [248, 248]

; Import in the image from the file.
image = READ_BINARY(file, DATA_DIMS = imageSize)

; Initialize the display.
DEVICE, DECOMPOSED = 0
LOADCT, 0

; Create a window and display the image.
WINDOW, 0, XSIZE = imageSize[0], YSIZE = imageSize[1], $
   TITLE = 'A Grayscale Image'
TV, image

END
```

Displaying Indexed Images with Direct Graphics

An indexed image contains up to 256 colors, typically defined by a color table associated with the image. The value of each pixel relates to a color within the associated color table. Combinations of the primary colors (red, green, and blue) make up the colors within the color table. Indexed images are usually stored in image files, instead of binary files, since they typically contain related color information (an associated color table). You can query an image file to determine if it contains an indexed image.

The following example imports an indexed image from the `avhrr.png` image file. This indexed image is a satellite photograph of the world. Most indexed images range in value from 0 up to 255 because of the range of the associated color table. Therefore, most indexed images do not require scaled displays.

For code that you can copy and paste into an Editor window, see “Example Code: Displaying Indexed Images with Direct Graphics” on page 15 or complete the following steps for a detailed description of the process.

1. Determine the path to the `avhrr.png` file:

   ```
   file = FILEPATH('avhrr.png', $
      SUBDIRECTORY = ['examples', 'data'])
   ```

2. Use QUERY_IMAGE to query the file to determine image parameters:

   ```
   queryStatus = QUERY_IMAGE(file, imageInfo)
   ```

3. Output the results of the file query:

   ```
   PRINT, 'Query Status = ', queryStatus
   HELP, imageInfo, /STRUCTURE
   ```

   The following text appears in the Output Log:

   ```
   Query Status =            1
   ** Structure <141d0b0>, 7 tags, length=36, refs=1:
      CHANNELS      LONG     1
      DIMENSIONS    LONG     Array[2]
      HAS_PALETTE   INT      1
      IMAGE_INDEX   LONG     0
      NUM_IMAGES    LONG     1
      PIXEL_TYPE    INT      1
      TYPE          STRING   'PNG'
   ```

4. Set the image size parameter from the query information:

   ```
   imageSize = imageInfo.dimensions
   ```

   The HAS_PALETTE tag has a value of 1. Thus, the image has a palette (color table), which is also contained within the file. The color table is made up of its three primary components (the red component, the green component, and the blue component).

5. Use READ_IMAGE to import the image and its associated color table from the file:

   ```
   image = READ_IMAGE(file, red, green, blue)
   ```

6. If you are running IDL on a TrueColor display, set the DECOMPOSED keyword to the DEVICE command to zero before your first color table related routine is used within an IDL session or program. See “How Colors are Associated with Indexed and RGB Images” on page 84 for more information.

   ```
   DEVICE, DECOMPOSED = 0
   ```

7. Load the red, green, and blue components of the image’s associated color table:

   ```
   TVLCT, red, green, blue
   ```

8. Create a window and display the original image with the TV procedure:

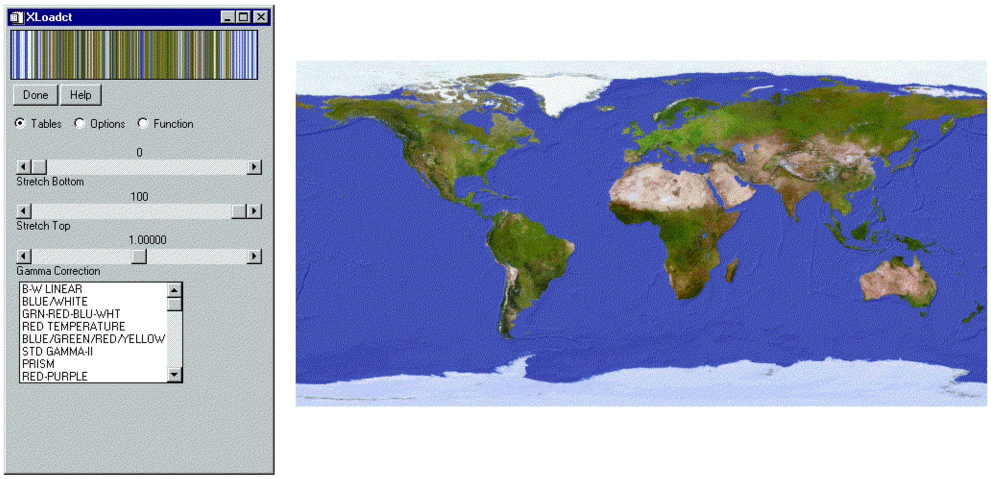
   ```
   WINDOW, 0, XSIZE = imageSize[0], YSIZE = imageSize[1], $
      TITLE = 'An Indexed Image'
   TV, image
   ```

9. Use the XLOADCT utility to display the associated color table:

   ```
   XLOADCT
   ```

   Click on the **Done** button of XLOADCT to exit out of the utility.

   The following figure shows the resulting indexed image and its color table.



(● Tables ○ Options ○ Function)

0
Stretch Bottom

100
Stretch Top

1.00000
Gamma Correction

B-W LINEAR
BLUE/WHITE
GRN-RED-BLU-WHT
RED TEMPERATURE
BLUE/GREEN/RED/YELLOW
STD GAMMA-II
PRISM
RED-PURPLE

Figure 3: Indexed Image and Associated Color Table in Direct Graphics

The data values within the image are indexed to specific colors within the table. You can change the color table associated with this image to show how an indexed images is dependent upon its related color table.

10. Change the current color table to the EOS B pre-defined color table:

    ```
    LOADCT, 27
    ```

11. Redisplay the image to show the color table change:

    ```
    TV, image
    ```

Note

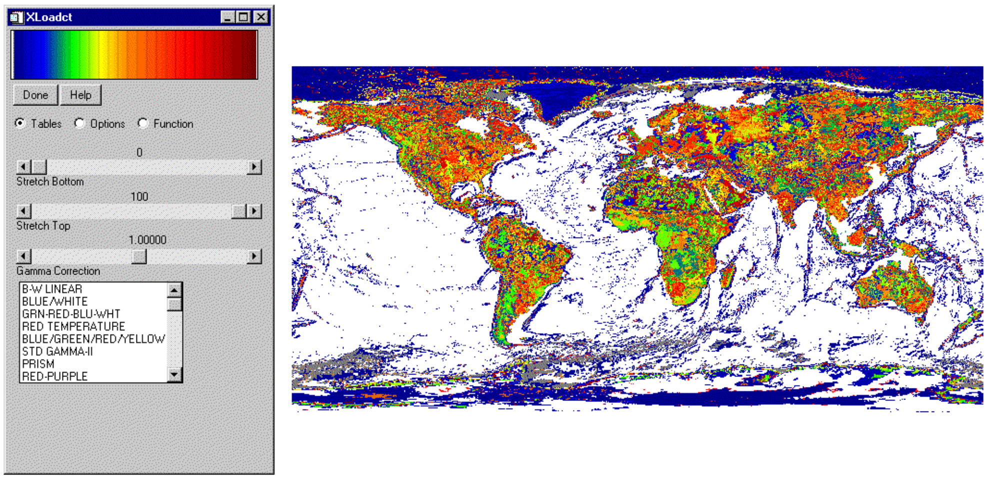
This step is not always necessary to redisplay the image. On some computers, the display will update automatically when the current color table is changed.

12. Use the XLOADCT utility to display the current color table:

    ```
    XLOADCT
    ```

    Click on the **Done** button of XLOADCT to exit out of the utility.

    The following figure shows the indexed image with the EOS B color table.



0
Stretch Bottom
100
Stretch Top
1.00000
Gamma Correction

B-W LINEAR
BLUE/WHITE
GRN-RED-BLU-WHT
RED TEMPERATURE
BLUE/GREEN/RED/YELLOW
STD GAMMA-II
PRISM
RED-PURPLE

Figure 4: Indexed Image and EOS B Color Table in Direct Graphics

Example Code: Displaying Indexed Images with Direct Graphics

Copy and paste the following text into the IDL Editor window. After saving the file as `DisplayIndexedImage_Direct.pro`, compile and run the program to reproduce the previous example. The BLOCK keyword is set when using the XLOADCT utility to force the example routine to wait until the **Done** button is pressed to continue.

```
PRO DisplayIndexedImage_Direct

; Determine the path to the file.
file = FILEPATH('avhrr.png', $
   SUBDIRECTORY = ['examples', 'data'])

; Query the file to determine image parameters.
queryStatus = QUERY_IMAGE(file, imageInfo)

; Output the results of the file query.
```

```
PRINT, 'Query Status = ', queryStatus
HELP, imageInfo, /STRUCTURE

; Set image size parameter.
imageSize = imageInfo.dimensions

; Import in the image and its associated color table
; from the file.
image = READ_IMAGE(file, red, green, blue)

; Initialize the display.
DEVICE, DECOMPOSED = 0
TVLCT, red, green, blue

; Create a window and display the image.
WINDOW, 0, XSIZE = imageSize[0], YSIZE = imageSize[1], $
   TITLE = 'An Indexed Image'
TV, image

; Use the XLOADCT utility to display the color table.
XLOADCT, /BLOCK

; Change the color table to the EOS B pre-defined color
; table.
LOADCT, 27

; Redisplay the image with the EOS B color table.
TV, image

; Use the XLOADCT utility to display the current color
; table.
XLOADCT, /BLOCK

END
```

Displaying RGB Images with Direct Graphics

RGB images are three-dimensional arrays made up of width, height, and three channels of color information. In Direct Graphics, these images are displayed with the TV procedure. The TRUE keyword to TV is set according to the interleaving of the RGB image. With RGB images, the interleaving, or arrangement of the channels within the image file, dictates the setting of the TRUE keyword. If the image is:

- pixel interleaved (3, w, h), TRUE is set to 1.
- line interleaved (w, 3, h), TRUE is set to 2.

- image interleaved (w, h, 3), TRUE is set to 3.

You can determine if an image file contains an RGB image by querying the file. The CHANNELS tag of the resulting query structure will equal 3 if the file’s image is RGB. The query does not determine which interleaving is used in the image, but the array returned in DIMENSIONS tag of the query structure can be used to determine the type of interleaving.

Unlike indexed images (two dimensional arrays with an associated color table), RGB images contain their own color information. However, if you are using a PseudoColor display, your RGB images must be converted to indexed images to be displayed within IDL. See “How Colors are Associated with Indexed and RGB Images” on page 84 for more information on RGB images and PseudoColor displays.

The following example queries and imports a pixel-interleaved RGB image from the `rose.jpg` image file. This RGB image is a close-up photograph of a red rose. It is pixel interleaved.

For code that you can copy and paste into an Editor window, see “Example Code: Displaying RGB Images with Direct Graphics” on page 19 or complete the following steps for a detailed description of the process.

1. Determine the path to the `rose.jpg` file:

```
file = FILEPATH('rose.jpg', $
   SUBDIRECTORY = ['examples', 'data'])
```

2. Use QUERY_IMAGE to query the file to determine image parameters:

```
queryStatus = QUERY_IMAGE(file, imageInfo)
```

3. Output the results of the file query:

```
PRINT, 'Query Status = ', queryStatus
HELP, imageInfo, /STRUCTURE
```

The following text appears in the Output Log:

```
Query Status =            1
** Structure <14055f0>, 7 tags, length=36, refs=1:
   CHANNELS       LONG      3
   DIMENSIONS     LONG      Array[2]
   HAS_PALETTE    INT       0
   IMAGE_INDEX    LONG      0
   NUM_IMAGES     LONG      1
   PIXEL_TYPE     INT       1
   TYPE           STRING    'JPEG'
```

The CHANNELS tag has a value of 3. Thus, the image is an RGB image.

4. Set the image size parameter from the query information:

   ```
   imageSize = imageInfo.dimensions
   ```

   The type of interleaving can be determined from the image size parameter and actual size of each dimension of the image. To determine the size of each dimension, you must first import the image.

5. Use READ_IMAGE to import the image from the file:

   ```
   image = READ_IMAGE(file)
   ```

6. Determine the size of each dimension within the image:

   ```
   imageDims = SIZE(image, /DIMENSIONS)
   ```

7. Determine the type of interleaving by comparing the dimension sizes to the image size parameter from the file query:

   ```
   interleaving = WHERE((imageDims NE imageSize[0]) AND $
      (imageDims NE imageSize[1])) + 1
   ```

8. Output the results of the interleaving computation:

   ```
   PRINT, 'Type of Interleaving = ', interleaving
   ```

   The following text appears in the Output Log:

   ```
   Type of Interleaving = 1
   ```

   The image is pixel interleaved. If the resulting value was 2, the image would have been line interleaved. If the resulting value was 3, the image would have been image interleaved.

9. If you are running IDL on a TrueColor display, set the DECOMPOSED keyword to the DEVICE command to one before your first RGB image is displayed within an IDL session or program. See “How Colors are Associated with Indexed and RGB Images” for more information:

   ```
   DEVICE, DECOMPOSED = 1
   ```

10. Create a window and display the image with the TV procedure:

   ```
   WINDOW, 0, XSIZE = imageSize[0], YSIZE = imageSize[1], $
      TITLE = 'An RGB Image'
   TV, image, TRUE = interleaving[0]
   ```

The following figure shows the resulting RGB image display.

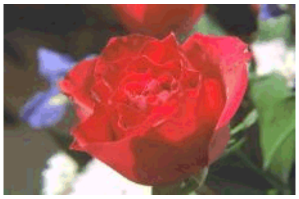


Figure 5: RGB Image in Direct Graphics

Example Code: Displaying RGB Images with Direct Graphics

Copy and paste the following text into the IDL Editor window. After saving the file as `DisplayRGBImage_Direct.pro`, compile and run the program to reproduce the previous example.

```
PRO DisplayRGBImage_Direct

; Determine the path to the file.
file = FILEPATH('rose.jpg', $
   SUBDIRECTORY = ['examples', 'data'])

; Query the file to determine image parameters.
queryStatus = QUERY_IMAGE(file, imageInfo)

; Output the results of the file query.
PRINT, 'Query Status = ', queryStatus
HELP, imageInfo, /STRUCTURE

; Set the image size parameter from the query
; information.
imageSize = imageInfo.dimensions

; Import the image.
image = READ_IMAGE(file)

; Determine the size of each dimension within the image.
imageDims = SIZE(image, /DIMENSIONS)

; Determine the type of interleaving by comparing the
; dimension sizes with the image size parameter from the
; file query.
interleaving = WHERE((imageDims NE imageSize[0]) AND $
   (imageDims NE imageSize[1])) + 1

; Output the results of the interleaving computation.
PRINT, 'Type of Interleaving = ', interleaving

; Initialize display.
DEVICE, DECOMPOSED = 1

; Create a window and display the image with the TV
; procedure and its TRUE keyword.
WINDOW, 0, XSIZE = imageSize[0], YSIZE = imageSize[1], $
   TITLE = 'An RGB Image'
TV, image, TRUE = interleaving[0]

END
```

Creating Object Graphics Image Displays

In Object Graphics, binary, grayscale, indexed, and RGB images are contained in image objects. The image object is contained within a model object, which is a part of a view object. The view object is displayed within a window object with the Draw method to that window object. For some types of images, their data values must be scaled with the BYTSCL function before displaying the image in a window object. When the scaling is required depends on the type of image to be displayed.

This section includes the following examples:

- "Displaying Binary Images with Object Graphics".
- "Displaying Grayscale Images with Object Graphics" on page 24.
- "Displaying Indexed Images with Object Graphics" on page 27.
- "Displaying RGB images with Object Graphics" on page 32.

Displaying Binary Images with Object Graphics

Binary images are composed of pixels having one of two values, usually 0 or 1. These values are usually zero and one. With most color tables, pixels having values of zero and one are displayed with almost the same color, especially with a normal grayscale color table. Thus, a binary image is usually scaled to display the zeros as black and the ones as white.

The following example imports a binary image of the world from the `continent_mask.dat` binary file. In this image, the oceans are zeros (black) and the continents are ones (white). This type of image can be used to mask out (omit) data over the oceans. The image contains byte data values and is 360 pixels by 360 pixels.

For code that you can copy and paste into an Editor window, see "Example Code: Displaying Binary Images with Object Graphics" on page 23 or complete the following steps for a detailed description of the process.

1. Determine the path to the `continent_mask.dat` file:

   ```
   file = FILEPATH('continent_mask.dat', $
      SUBDIRECTORY = ['examples', 'data'])
   ```

2. Initialize the image size parameter:

   ```
   imageSize = [360, 360]
   ```

3. Use READ_BINARY to import the image from the file:

   ```
   image = READ_BINARY(file, DATA_DIMS = imageSize)
   ```

4. Initialize the display objects:

   ```
   oWindow = OBJ_NEW('IDLgrWindow', RETAIN = 2, $
      DIMENSIONS = imageSize, $
      TITLE = 'A Binary Image, Not Scaled')
   oView = OBJ_NEW('IDLgrView', $
      VIEWPLANE_RECT = [0., 0., imageSize])
   oModel = OBJ_NEW('IDLgrModel')
   ```

5. Initialize the image object:

   ```
   oImage = OBJ_NEW('IDLgrImage', image)
   ```

6. Add the image object to the model, which is added to the view, then display the view in the window:

   ```
   oModel -> Add, oImage
   oView -> Add, oModel
   oWindow -> Draw, oView
   ```

   The resulting window should be all black (blank). The binary image contains zeros and ones, which are almost the same color (black). A binary image should be scaled prior to displaying in order to show the ones as white.

7. Initialize another window:

   ```
   oWindow = OBJ_NEW('IDLgrWindow', RETAIN = 2, $
      DIMENSIONS = imageSize, $
      TITLE = 'A Binary Image, Scaled')
   ```

8. Update the image object with a scaled version of the image:

   ```
   oImage -> SetProperty, DATA = BYTSCL(image)
   ```

9. Display the view in the window:

   ```
   oWindow -> Draw, oView
   ```

The following figure shows the results of scaling this display.

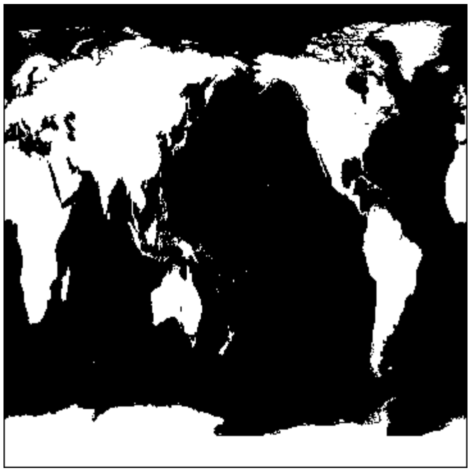


Figure 6: Binary Image in Object Graphics

10. Cleanup the object references. When working with objects always remember to cleanup any object references with the OBJ_DESTROY routine. Since the view contains all the other objects, except for the window (which is destroyed by the user), you only need to use OBJ_DESTROY on the view object.

    ```
    OBJ_DESTROY, [oView]
    ```

Example Code: Displaying Binary Images with Object Graphics

Copy and paste the following text into the IDL Editor window. After saving the file as `DisplayBinaryImage_Object.pro`, compile and run the program to reproduce the previous example.

```
PRO DisplayBinaryImage_Object

; Determine the path to the file.
file = FILEPATH('continent_mask.dat', $
   SUBDIRECTORY = ['examples', 'data'])

; Initialize the image size parameter.
imageSize = [360, 360]

; Import the image.
image = READ_BINARY(file, DATA_DIMS = imageSize)
```

```
; Initialize display objects.
oWindow = OBJ_NEW('IDLgrWindow', RETAIN = 2, $
   DIMENSIONS = imageSize, $
   TITLE = 'A Binary Image, Not Scaled')
oView = OBJ_NEW('IDLgrView', $
   VIEWPLANE_RECT = [0., 0., imageSize])
oModel = OBJ_NEW('IDLgrModel')

; Initialize image object.
oImage = OBJ_NEW('IDLgrImage', image)

; Add the image to the model, which is added to the
; view, and then display the view in the window.
oModel -> Add, oImage
oView -> Add, oModel
oWindow -> Draw, oView

; Initialize another window.
oWindow = OBJ_NEW('IDLgrWindow', RETAIN = 2, $
   DIMENSIONS = imageSize, $
   TITLE = 'A Binary Image, Scaled')

; Update the image object with a scaled version of the
; image.
oImage -> SetProperty, DATA = BYTSCL(image)

; Display the view in the window.
oWindow -> Draw, oView

; Cleanup object references.
OBJ_DESTROY, [oView]

END
```

Displaying Grayscale Images with Object Graphics

Features within grayscale images are created by pixels that have varying intensities. Pixel values range from least intense (black) to the most instance (white). Since a grayscale image is composed of pixels of varying intensities, it is best displayed with a color table that progresses linearly from black to white. IDL provides several pre-defined color tables that progress linearly from black to white, but the best representation of a grayscale image is still the normal grayscale color table, which is the default palette for IDL Object Graphics.

The following example imports a grayscale image from the `convec.dat` binary file. This grayscale image shows the convection of the Earth’s mantle. The image contains byte data values and is 248 pixels by 248 pixels. Since the data type is byte, this image does not need to by scaled before display. If the data was of any type other than byte and the data values were not within the range of 0 up to 255, the display would need to scale the image in order to show its intensities.

For code that you can copy and paste into an Editor window, see “Example Code: Displaying Grayscale Images with Object Graphics” on page 26 or complete the following steps for a detailed description of the process.

1. Determine the path to the `convec.dat` file:

```
file = FILEPATH('convec.dat', $
   SUBDIRECTORY = ['examples', 'data'])
```

2. Initialize the image size parameter:

```
imageSize = [248, 248]
```

3. Using READ_BINARY, import the image from the file:

```
image = READ_BINARY(file, DATA_DIMS = imageSize)
```

4. Initialize the display objects:

```
oWindow = OBJ_NEW('IDLgrWindow', RETAIN = 2, $
   DIMENSIONS = imageSize, $
   TITLE = 'A Grayscale Image')
oView = OBJ_NEW('IDLgrView', $
   VIEWPLANE_RECT = [0., 0., imageSize])
oModel = OBJ_NEW('IDLgrModel')
```

5. Initialize the image object:

```
oImage = OBJ_NEW('IDLgrImage', image, /GREYSCALE)
```

6. Add the image object to the model, which is added to the view, then display the view in the window:

```
oModel -> Add, oImage
oView -> Add, oModel
oWindow -> Draw, oView
```

The following figure shows the resulting grayscale image display

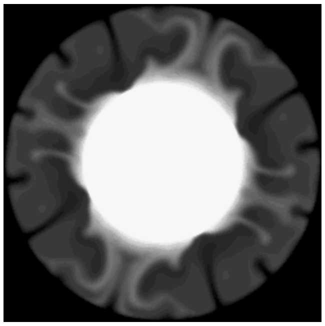


Figure 7: Grayscale Image in Object Graphics

7. Cleanup the object references. When working with objects always remember to cleanup any object references with the OBJ_DESTROY routine. Since the view contains all the other objects, except for the window (which is destroyed by the user), you only need to use OBJ_DESTROY on the view object.

   ```
   OBJ_DESTROY, [oView]
   ```

Example Code: Displaying Grayscale Images with Object Graphics

Copy and paste the following text into the IDL Editor window. After saving the file as `DisplayGrayscaleImage_Object.pro`, compile and run the program to reproduce the previous example.

```
PRO DisplayGrayscaleImage_Object

; Determine the path to the file.
file = FILEPATH('convec.dat', $
   SUBDIRECTORY = ['examples', 'data'])

; Initialize the image size parameters.
imageSize = [248, 248]

; Import the image.
image = READ_BINARY(file, DATA_DIMS = imageSize)

; Initialize display objects.
oWindow = OBJ_NEW('IDLgrWindow', RETAIN = 2, $
   DIMENSIONS = imageSize, $
```

```
   TITLE = 'A Grayscale Image')
oView = OBJ_NEW('IDLgrView', $
   VIEWPLANE_RECT = [0., 0., imageSize])
oModel = OBJ_NEW('IDLgrModel')

; Initialize image object.
oImage = OBJ_NEW('IDLgrImage', image, $
   /GREYSCALE)

; Add the image object to the model, which is added to
; the view, then display the view in the window.
oModel -> Add, oImage
oView -> Add, oModel
oWindow -> Draw, oView

; Cleanup object references.
OBJ_DESTROY, [oView]

END
```

Displaying Indexed Images with Object Graphics

An indexed image contains up to 256 colors, typically defined by a color table associated with the image. The value of each pixel relates to a color within the associated color table. Combinations of the primary colors (red, green, and blue) make up the colors within the color table. Indexed images are usually stored in image files, instead of binary files, since they typically contain related color information (an associated color table). You can query an image file to determine if it contains an indexed image.

The following example imports an indexed image from the `avhrr.png` image file. This indexed image is a satellite photograph of the world. Most indexed images range in value from 0 up to 255 because of the range of the associated color table. Therefore, most indexed images do not require scaled displays.

For code that you can copy and paste into an Editor window, see "Example Code: Displaying Indexed Images with Object Graphics" on page 31 or complete the following steps for a detailed description of the process.

1. Determine the path to the `avhrr.png` file:

   ```
   file = FILEPATH('avhrr.png', $
      SUBDIRECTORY = ['examples', 'data'])
   ```

2. Use QUERY_IMAGE to query the file to determine image parameters:

   ```
   queryStatus = QUERY_IMAGE(file, imageInfo)
   ```

3. Output the results of the file query:

```
PRINT, 'Query Status = ', queryStatus
HELP, imageInfo, /STRUCTURE
```

   The following text appears in the Output Log:

```
Query Status =            1
** Structure <141d0b0>, 7 tags, length=36, refs=1:
   CHANNELS       LONG     1
   DIMENSIONS     LONG     Array[2]
   HAS_PALETTE    INT      1
   IMAGE_INDEX    LONG     0
   NUM_IMAGES     LONG     1
   PIXEL_TYPE     INT      1
   TYPE           STRING   'PNG'
```

4. Set the image size parameter from the query information:

```
imageSize = imageInfo.dimensions
```

   The HAS_PALETTE tag has a value of 1. Thus, the image has a palette (color table), which is also contained within the file. The color table is made up of its three primary components (the red component, the green component, and the blue component).

5. Use READ_IMAGE to import the image and its associated color table from the file:

```
image = READ_IMAGE(file, red, green, blue)
```

6. Initialize the display objects:

```
oWindow = OBJ_NEW('IDLgrWindow', RETAIN = 2, $
   DIMENSIONS = imageSize, TITLE = 'An Indexed Image')
oView = OBJ_NEW('IDLgrView', $
   VIEWPLANE_RECT = [0., 0., imageSize])
oModel = OBJ_NEW('IDLgrModel')
```

7. Initialize the image's palette object:

```
oPalette = OBJ_NEW('IDLgrPalette', red, green, blue)
```

8. Initialize the image object with the resulting palette object:

```
oImage = OBJ_NEW('IDLgrImage', image, $
   PALETTE = oPalette)
```

9. Add the image object to the model, which is added to the view, then display the view in the window:

```
oModel -> Add, oImage
oView -> Add, oModel
oWindow -> Draw, oView
```

10. Use the colorbar object to display the associated color table in another window:

```
oCbWindow = OBJ_NEW('IDLgrWindow', RETAIN = 2, $
   DIMENSIONS = [256, 48], $
   TITLE = 'Original Color Table')
oCbView = OBJ_NEW('IDLgrView', $
   VIEWPLANE_RECT = [0., 0., 256., 48.])
oCbModel = OBJ_NEW('IDLgrModel')
oColorbar = OBJ_NEW('IDLgrColorbar', PALETTE = oPalette, $
   DIMENSIONS = [256, 16], SHOW_AXIS = 1)
oCbModel -> Add, oColorbar
oCbView -> Add, oCbModel
oCbWindow -> Draw, oCbView
```

The following figure shows the resulting indexed image and its color table.

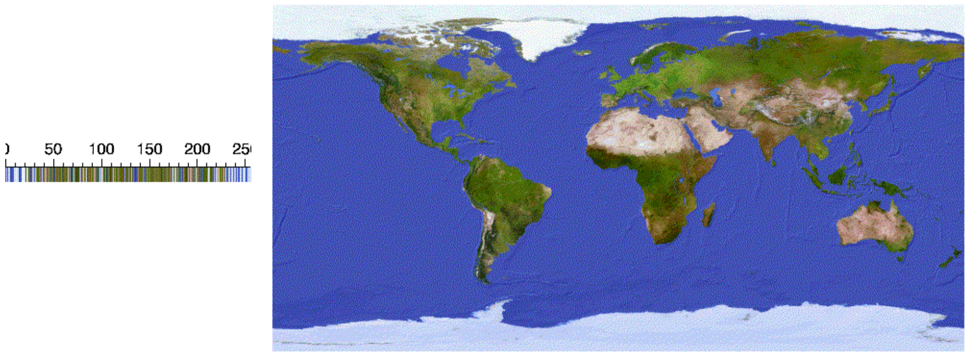


Figure 8: Indexed Image and Associated Color Table in Object Graphics

The data values within the image are indexed to specific colors within the table. You can change the color table associated with this image to show how an indexed image is dependent upon its related color tables.

11. Change the palette (color table) to the EOS B pre-defined color table:

```
oPalette -> LoadCT, 27
```

12. Redisplay the image in another window to show the palette change:

```
oWindow = OBJ_NEW('IDLgrWindow', RETAIN = 2, $
   DIMENSIONS = imageSize, TITLE = 'An Indexed Image')
oWindow -> Draw, oView
```

13. Redisplay the colorbar in another window to show the palette change:

```
oCbWindow = OBJ_NEW('IDLgrWindow', RETAIN = 2, $
   DIMENSIONS = [256, 48], $
   TITLE = 'EOS B Color Table')
oCbWindow -> Draw, oCbView
```

    The following figure shows the indexed image with the EOS B color table.

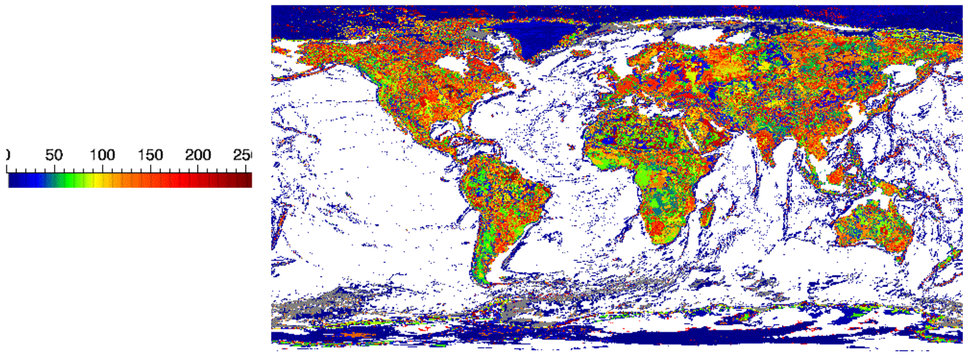


Figure 9: Indexed Image and EOS B Color Table in Object Graphics

14. Cleanup the object references. When working with objects always remember to cleanup any object references with the OBJ_DESTROY routine. Since the view contains all the other objects, except for the window (which is destroyed by the user), you only need to use OBJ_DESTROY on the view object.

```
OBJ_DESTROY, [oView, oCbVeiw, oPalette]
```

Example Code: Displaying Indexed Images with Object Graphics

Copy and paste the following text into the IDL Editor window. After saving the file as `DisplayIndexedImage_Object.pro`, compile and run the program to reproduce the previous example.

```
PRO DisplayIndexedImage_Object

; Determine the path to the file.
file = FILEPATH('avhrr.png', $
   SUBDIRECTORY = ['examples', 'data'])

; Query the file to determine image parameters.
queryStatus = QUERY_IMAGE(file, imageInfo)

; Output the results of the query.
PRINT, 'Query Status = ', queryStatus
HELP, imageInfo, /STRUCTURE

; Set the image size parameter.
imageSize = imageInfo.dimensions

; Import in the image.
image = READ_IMAGE(file, red, green, blue)

; Initialize the display objects.
oWindow = OBJ_NEW('IDLgrWindow', RETAIN = 2, $
   DIMENSIONS = imageSize, TITLE = 'An Indexed Image')
oView = OBJ_NEW('IDLgrView', $
   VIEWPLANE_RECT = [0., 0., imageSize])
oModel = OBJ_NEW('IDLgrModel')

; Initialize the image's palette object.
oPalette = OBJ_NEW('IDLgrPalette', red, green, blue)

; Initialize the image object with the resulting
; palette object.
oImage = OBJ_NEW('IDLgrImage', image, $
   PALETTE = oPalette)

; Add the image object to the model, which is added to
; the view, then display the view in the window.
oModel -> Add, oImage
oView -> Add, oModel
oWindow -> Draw, oView

; Use the colorbar object to display the associated
; color table in another window.
```

```
oCbWindow = OBJ_NEW('IDLgrWindow', RETAIN = 2, $
   DIMENSIONS = [256, 48], $
   TITLE = 'Original Color Table')
oCbView = OBJ_NEW('IDLgrView', $
   VIEWPLANE_RECT = [0., 0., 256., 48.])
oCbModel = OBJ_NEW('IDLgrModel')
oColorbar = OBJ_NEW('IDLgrColorbar', PALETTE = oPalette, $
   DIMENSIONS = [256, 16], SHOW_AXIS = 1)
oCbModel -> Add, oColorbar
oCbView -> Add, oCbModel
oCbWindow -> Draw, oCbView

; Change the palette (color table) to the EOS B
; pre-defined color table.
oPalette -> LoadCT, 27

; Redisplay the image with the other color table in
; another window.
oWindow = OBJ_NEW('IDLgrWindow', RETAIN = 2, $
   DIMENSIONS = imageSize, TITLE = 'An Indexed Image')
oWindow -> Draw, oView

; Redisplay the colorbar with the other color table
; in another window.
oCbWindow = OBJ_NEW('IDLgrWindow', RETAIN = 2, $
   DIMENSIONS = [256, 48], $
   TITLE = 'EOS B Color Table')
oCbWindow -> Draw, oCbView

; Cleanup object references.
OBJ_DESTROY, [oView, oCbView, oPalette]

END
```

Displaying RGB images with Object Graphics

RGB images are three-dimensional arrays. In Object Graphics, an RGB image is contained within an image object. An image object is contained within a model object, which is a part of a view object. The view object is displayed within a window object with the Draw method to that window object.

The INTERLEAVE keyword to the Init method of the image object is used when displaying an RGB image. With RGB images, the interleaving, or arrangement of the channels within the image file, dictates the setting of the INTERLEAVE keyword. If the image is:

- pixel interleaved (3, w, h), INTERLEAVE is set to 0.

- line interleaved (w, 3, h), INTERLEAVE is set to 1.
- image interleaved (w, h, 3), INTERLEAVE is set to 2.

You can determine if an image file contains an RGB image by querying the file. The CHANNELS tag of the resulting query structure will equal 3 if the file's image is RGB. The query does not determine which interleaving is used in the image, but the array returned in DIMENSIONS tag of the query structure can be used to determine the type of interleaving.

Unlike the previous two-dimensional array images, RGB images contain their own color information. RGB images do not require color tables (palettes). Image files containing RGB images usually do not contain the associated color table information.

The following example queries and imports a pixel-interleaved RGB image from the `rose.jpg` image file. This RGB image is a close-up photograph of a red rose. It is pixel interleaved.

For code that you can copy and paste into an Editor window, see "Example Code: Displaying RGB Images with Object Graphics" on page 35 or complete the following steps for a detailed description of the process.

1. Determine the path to the `rose.jpg` file:

```
file = FILEPATH('rose.jpg', $
   SUBDIRECTORY = ['examples', 'data'])
```

2. Use QUERY_IMAGE to query the file to determine image parameters:

```
queryStatus = QUERY_IMAGE(file, imageInfo)
```

3. Output the results of the file query:

```
PRINT, 'Query Status = ', queryStatus
HELP, imageInfo, /STRUCTURE
```

The following text appears in the Output Log:

```
Query Status =            1
** Structure <14055f0>, 7 tags, length=36, refs=1:
   CHANNELS       LONG      3
   DIMENSIONS     LONG      Array[2]
   HAS_PALETTE    INT       0
   IMAGE_INDEX    LONG      0
   NUM_IMAGES     LONG      1
   PIXEL_TYPE     INT       1
   TYPE           STRING    'JPEG'
```

The CHANNELS tag has a value of 3. Thus, the image is an RGB image.

4. Set the image size parameter from the query information:

   ```
   imageSize = imageInfo.dimensions
   ```

   The type of interleaving can be determined from the image size parameter and actual size of each dimension of the image. To determine the size of each dimension, you must first import the image.

5. Use READ_IMAGE to import the image from the file:

   ```
   image = READ_IMAGE(file)
   ```

6. Determine the size of each dimension within the image:

   ```
   imageDims = SIZE(image, /DIMENSIONS)
   ```

7. Determine the type of interleaving by comparing the dimension sizes to the image size parameter from the file query:

   ```
   interleaving = WHERE((imageDims NE imageSize[0]) AND $
      (imageDims NE imageSize[1]))
   ```

8. Output the results of the interleaving computation:

   ```
   PRINT, 'Type of Interleaving = ', interleaving
   ```

   The following text appears in the Output Log:

   ```
   Type of Interleaving = 0
   ```

   The image is pixel interleaved. If the resulting value was 1, the image would have been line interleaved. If the resulting value was 2, the image would have been image interleaved.

9. Initialize the display objects:

   ```
   oWindow = OBJ_NEW('IDLgrWindow', RETAIN = 2, $
      DIMENSIONS = imageSize, TITLE = 'An RGB Image')
   oView = OBJ_NEW('IDLgrView', $
      VIEWPLANE_RECT = [0., 0., imageSize])
   oModel = OBJ_NEW('IDLgrModel')
   ```

10. Initialize the image object:

    ```
    oImage = OBJ_NEW('IDLgrImage', image, $
       INTERLEAVE = interleaving[0])
    ```

11. Add the image object to the model, which is added to the view, then display the view in the window:

    ```
    oModel -> Add, oImage
    oView -> Add, oModel
    oWindow -> Draw, oView
    ```

The following figure shows the resulting RGB image display.

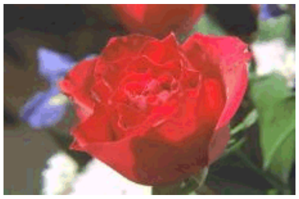


Figure 10: RGB Image in Object Graphics

12. Cleanup the object references. When working with objects always remember to cleanup any object references with the OBJ_DESTROY routine. Since the view contains all the other objects, except for the window (which is destroyed by the user), you only need to use OBJ_DESTROY on the view object.

    ```
    OBJ_DESTROY, [oView]
    ```

Example Code: Displaying RGB Images with Object Graphics

Copy and paste the following text into the IDL Editor window. After saving the file as `DisplayRGBImage_Object.pro`, compile and run the program to reproduce the previous example.

```
PRO DisplayRGBImage_Object

; Determine the path to the file.
file = FILEPATH('rose.jpg', $
   SUBDIRECTORY = ['examples', 'data'])

; Query the file to determine image parameters.
queryStatus = QUERY_IMAGE(file, imageInfo)

; Output the results of the query.
PRINT, 'Query Status = ', queryStatus
HELP, imageInfo, /STRUCTURE

; Set the image size parameter from the query
; information.
imageSize = imageInfo.dimensions

; Import in the image.
image = READ_IMAGE(file)
```

```
; Determine the size of each dimension within the image.
imageDims = SIZE(image, /DIMENSIONS)

; Determine the type of interleaving by comparing
; dimension size and the size of the image.
interleaving = WHERE((imageDims NE imageSize[0]) AND $
   (imageDims NE imageSize[1]))

; Output the results of the interleaving computation.
PRINT, 'Type of Interleaving = ', interleaving

; Initialize the display objects.
oWindow = OBJ_NEW('IDLgrWindow', RETAIN = 2, $
   DIMENSIONS = imageSize, TITLE = 'An RGB Image')
oView = OBJ_NEW('IDLgrView', $
   VIEWPLANE_RECT = [0., 0., imageSize])
oModel = OBJ_NEW('IDLgrModel')

; Initialize the image object.
oImage = OBJ_NEW('IDLgrImage', image, $
   INTERLEAVE = interleaving[0])

; Add the image object to the model, which is added to
; the view, then display the view in the window.
oModel -> Add, oImage
oView -> Add, oModel
oWindow -> Draw, oView

; Cleanup object references.
OBJ_DESTROY, [oView]

END
```

Displaying Multiple Images in a Window

How multiple images are displayed in a single window depends upon which graphics system is being used to display the images. Direct Graphics uses location input arguments for the TV procedure to position images in a window, see "Displaying Multiple Images in Direct Graphics" for more information. Object Graphics used the LOCATION keyword to the Init method of the image object to position images in a window, see "Displaying Multiple Images in Object Graphics" on page 42 for more information.

Displaying Multiple Images in Direct Graphics

An RGB image contains three channels (or bands) of color information, one for each primary color (red, green, and blue). A channel is actually a grayscale image (a two-dimensional intensity array with an associated grayscale color table). These intensity images show how much of each primary color makes up the RGB image. You can display all three channels by extracting them from the RGB image.

The following example imports an RGB image from the `rose.jpg` image file. This RGB image is a close-up photograph of a red rose and is pixel interleaved. This example extracts the three channels of this image, and displays them as grayscale images in various locations within the same window.

For code that you can copy and paste into an Editor window, see "Example Code: Displaying Multiple Image in Direct Graphics" on page 40 or complete the following steps for a detailed description of the process.

1. Determine the path to the `rose.jpg` file:

   ```
   file = FILEPATH('rose.jpg', $
      SUBDIRECTORY = ['examples', 'data'])
   ```

2. Use QUERY_IMAGE to query the file to determine image parameters:

   ```
   queryStatus = QUERY_IMAGE(file, imageInfo)
   ```

3. Set the image size parameter from the query information:

   ```
   imageSize = imageInfo.dimensions
   ```

4. Use READ_IMAGE to import the image from the file:

   ```
   image = READ_IMAGE(file)
   ```

5. Extract the channels (as images) from the pixel interleaved RGB image:

   ```
   redChannel = REFORM(image[0, *, *])
   greenChannel = REFORM(image[1, *, *])
   blueChannel = REFORM(image[2, *, *])
   ```

6. If you are running IDL on a TrueColor display, set the DECOMPOSED keyword to the DEVICE command to zero before your first color table related routine is used within an IDL session or program. See “How Colors are Associated with Indexed and RGB Images” for more information.

   ```
   DEVICE, DECOMPOSED = 0
   ```

7. Since the channels are grayscale images, load a grayscale color table:

   ```
   LOADCT, 0
   ```

   The TV procedure can be used to display the channels (grayscale images). The TV procedure has two different location input arguments. One argument is *position*. This argument arranges the image in a calculated location based on the size of the display and the dimension sizes of the image. See TV in the *IDL Reference Guide* for more information.

8. Create a window and horizontally display the three channels with the *position* argument:

   ```
   WINDOW, 0, XSIZE = 3*imageSize[0], YSIZE = imageSize[1], $
      TITLE = 'The Channels of an RGB Image'
   TV, redChannel, 0
   TV, greenChannel, 1
   TV, blueChannel, 2
   ```

   The following figure shows the resulting grayscale images.

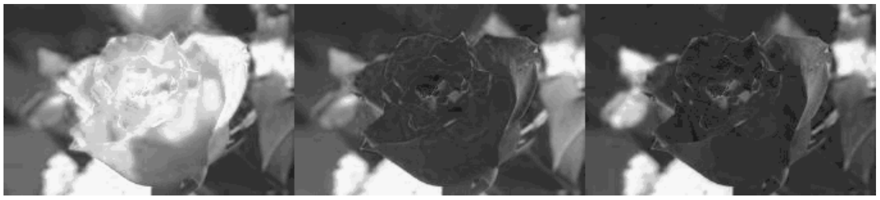


Figure 11: Horizontal Display of RGB Channels in Direct Graphics

The TV procedure can also be used with its *x* and *y* input arguments. These arguments define the location of the lower left corner of the image. The values of these arguments are in device coordinates by default. However, you can provide data or normalized coordinates when the DATA or NORMAL keyword is set. See TV in the *IDL Reference Guide* for more information.

9. Create a window and vertically display the three channels with the *x* and *y* arguments:

```
WINDOW, 0, XSIZE = imageSize[0], YSIZE = 3*imageSize[1], $
   TITLE = 'The Channels of an RGB Image'
TV, redChannel, 0, 0
TV, greenChannel, 0, imageSize[1]
TV, blueChannel, 0, 2*imageSize[1]
```

The following figure shows the resulting grayscale images.

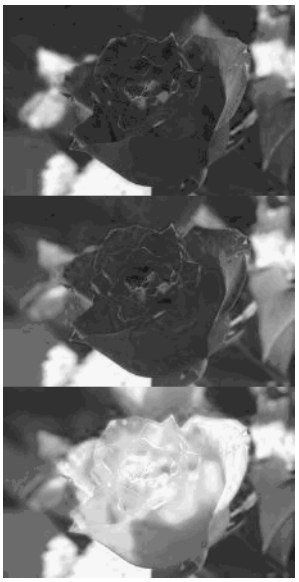


Figure 12: Vertical Display of RGB Channels in Direct Graphics

The *x* and *y* arguments can also be used to create a display of overlapping images. When overlapping images in Direct Graphics, you must remember the last image placed in the window will be in front of the previous images. So if you want to bring a display from the back of the window to the front, you must redisplay it after all the other displays.

10. Create another window:

```
WINDOW, 2, XSIZE = 2*imageSize[0], YSIZE = 2*imageSize[1], $
   TITLE = 'The Channels of an RGB Image'
```

11. Make a white background to distinguish the edges of the images:

```
ERASE, !P.COLOR
```

12. Diagonally display the three channels with the *x* and *y* arguments:

```
TV, redChannel, 0, 0
TV, greenChannel, imageSize[0]/2, imageSize[1]/2
TV, blueChannel, imageSize[0], imageSize[1]
```

The following figure shows the resulting grayscale images.

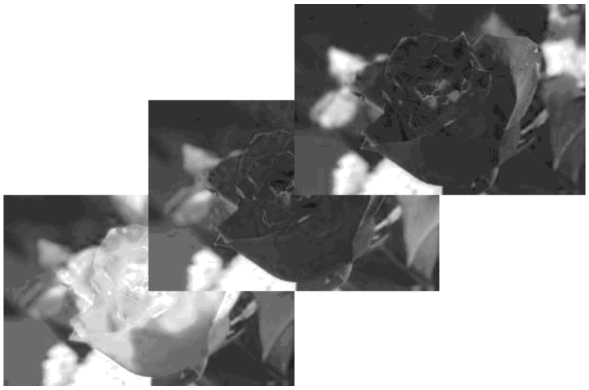


Figure 13: Diagonal Display of RGB Channels in Direct Graphics

Example Code: Displaying Multiple Image in Direct Graphics

Copy and paste the following text into the IDL Editor window. After saving the file as `DisplayMultiples_Direct.pro`, compile and run the program to reproduce the previous example.

```
PRO DisplayMultiples_Direct

; Determine the path to the file.
file = FILEPATH('rose.jpg', $
   SUBDIRECTORY = ['examples', 'data'])

; Query the file to determine image parameters.
```

```
queryStatus = QUERY_IMAGE(file, imageInfo)

; Set the image size parameter from the query
; information.
imageSize = imageInfo.dimensions

; Import the image.
image = READ_IMAGE(file)

; Extract the channels (as images) from the RGB image.
redChannel = REFORM(image[0, *, *])
greenChannel = REFORM(image[1, *, *])
blueChannel = REFORM(image[2, *, *])

; Initialize displays.
DEVICE, DECOMPOSED = 0
LOADCT, 0

; Create a window and horizontally display the channels.
WINDOW, 0, XSIZE = 3*imageSize[0], YSIZE = imageSize[1], $
   TITLE = 'The Channels of an RGB Image'
TV, redChannel, 0
TV, greenChannel, 1
TV, blueChannel, 2

; Create another window and vertically display the
; channels.
WINDOW, 1, XSIZE = imageSize[0], YSIZE = 3*imageSize[1], $
   TITLE = 'The Channels of an RGB Image'
TV, redChannel, 0, 0
TV, greenChannel, 0, imageSize[1]
TV, blueChannel, 0, 2*imageSize[1]

; Create another window.
WINDOW, 2, XSIZE = 2*imageSize[0], YSIZE = 2*imageSize[1], $
   TITLE = 'The Channels of an RGB Image'

; Make a white background.
ERASE, !P.COLOR

; Diagonally display the channels.
TV, redChannel, 0, 0
TV, greenChannel, imageSize[0]/2, imageSize[1]/2
TV, blueChannel, imageSize[0], imageSize[1]

END
```

Displaying Multiple Images in Object Graphics

An RGB image contains three channels (or bands) of color information, one for each primary color (red, green, and blue). A channel is actually a grayscale image (a two-dimensional intensity array with an associated grayscale color table). These intensity images show how much of each primary color makes up the RGB image. You can display all three channels by extracting them from the RGB image.

The following example imports an RGB image from the `rose.jpg` image file. This RGB image is a close-up photograph of a red rose and is pixel interleaved. This example extracts the three channels of this image, and displays them as grayscale images in various locations within the same window.

For code that you can copy and paste into an Editor window, see "Example Code: Displaying Multiple Image in Object Graphics" on page 46 or complete the following steps for a detailed description of the process.

1. Determine the path to the `rose.jpg` file:

```
file = FILEPATH('rose.jpg', $
   SUBDIRECTORY = ['examples', 'data'])
```

2. Use QUERY_IMAGE to query the file to determine image parameters:

```
queryStatus = QUERY_IMAGE(file, imageInfo)
```

3. Set the image size parameter from the query information:

```
imageSize = imageInfo.dimensions
```

4. Use READ_IMAGE to import the image from the file:

```
image = READ_IMAGE(file)
```

5. Extract the channels (as images) from the pixel interleaved RGB image:

```
redChannel = REFORM(image[0, *, *])
greenChannel = REFORM(image[1, *, *])
blueChannel = REFORM(image[2, *, *])
```

The LOCATION keyword to the Init method of the image object can be used to position an image within a window. The LOCATION keyword uses data coordinates, which are the same as device coordinates for images. Before initializing the image objects, you should initialize the display objects. The following steps display multiple images horizontally, vertically, and diagonally.

6. Initialize the display objects:

```
oWindow = OBJ_NEW('IDLgrWindow', RETAIN = 2, $
   DIMENSIONS = imageSize*[3, 1], $
   TITLE = 'The Channels of an RGB Image')
oView = OBJ_NEW('IDLgrView', $
   VIEWPLANE_RECT = [0., 0., imageSize]*[0, 0, 3, 1])
oModel = OBJ_NEW('IDLgrModel')
```

7. Now initialize the image objects and arrange them with the LOCATION keyword, see IDLgrImage for more information:

```
oRedChannel = OBJ_NEW('IDLgrImage', redChannel)
oGreenChannel = OBJ_NEW('IDLgrImage', greenChannel, $
   LOCATION = [imageSize[0], 0])
oBlueChannel = OBJ_NEW('IDLgrImage', blueChannel, $
   LOCATION = [2*imageSize[0], 0])
```

8. Add the image objects to the model, which is added to the view, then display the view in the window:

```
oModel -> Add, oRedChannel
oModel -> Add, oGreenChannel
oModel -> Add, oBlueChannel
oView -> Add, oModel
oWindow -> Draw, oView
```

   The following figure shows the resulting grayscale images.

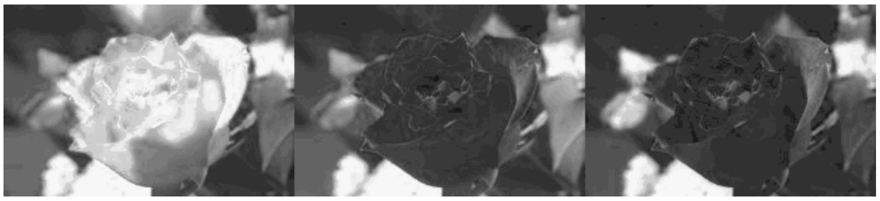


Figure 14: Horizontal Display of RGB Channels in Object Graphics

   These images can be displayed vertically in another window by first initializing another window and then updating the view and images with different location information.

9. Initialize another window object:

```
oWindow = OBJ_NEW('IDLgrWindow', RETAIN = 2, $
   DIMENSIONS = imageSize*[1, 3], $
   TITLE = 'The Channels of an RGB Image')
```

10. Change the view from horizontal to vertical:

```
oView -> SetProperty, $
   VIEWPLANE_RECT = [0., 0., imageSize]*[0, 0, 1, 3]
```

11. Change the locations of the channels:

```
oGreenChannel -> SetProperty, LOCATION = [0, imageSize[1]]
oBlueChannel -> SetProperty, LOCATION = [0, 2*imageSize[1]]
```

12. Display the updated view within the new window:

```
oWindow -> Draw, oView
```

The following figure shows the resulting grayscale images.

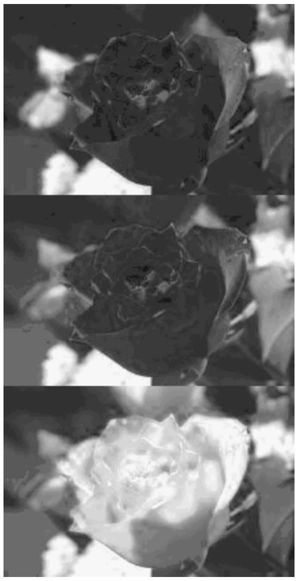


Figure 15: Vertical Display of RGB Channels in Object Graphics

These images can also be displayed diagonally in another window by first initializing the other window and then updating the view and images with different location information.The LOCATION can also be used to create a display overlapping images. When overlapping images in Object Graphics, you must remember the last image added to the model will be in front of the previous images.

13. Initialize another window object:

```
oWindow = OBJ_NEW('IDLgrWindow', RETAIN = 2, $
   DIMENSIONS = imageSize*[2, 2], $
   TITLE = 'The Channels of an RGB Image')
```

14. Change the view to prepare for a diagonal display:

```
oView -> SetProperty, $
   VIEWPLANE_RECT = [0., 0., imageSize]*[0, 0, 2, 2]
```

15. Change the locations of the channels:

```
oGreenChannel -> SetProperty, $
   LOCATION = [imageSize[0]/2, imageSize[1]/2]
oBlueChannel -> SetProperty, $
   LOCATION = [imageSize[0], imageSize[1]]
```

16. Display the updated view within the new window:

```
oWindow -> Draw, oView
```

The following figure shows the resulting grayscale images.

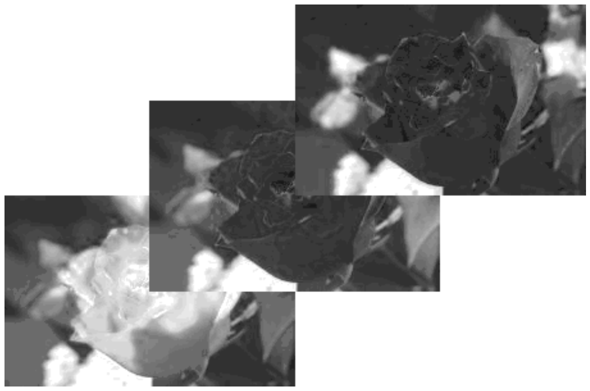


Figure 16: Diagonal Display of RGB Channels in Object Graphics

17. Cleanup the object references. When working with objects always remember to cleanup any object references with the OBJ_DESTROY routine. Since the view contains all the other objects, except for the window (which is destroyed by the user), you only need to use OBJ_DESTROY on the view object.

```
OBJ_DESTROY, [oView]
```

Example Code: Displaying Multiple Image in Object Graphics

Copy and paste the following text into the IDL Editor window. After saving the file as `DisplayMultiples_Object.pro`, compile and run the program to reproduce the previous example.

```
PRO DisplayMultiples_Object

; Determine the path to the file.
file = FILEPATH('rose.jpg', $
   SUBDIRECTORY = ['examples', 'data'])

; Query the file to determine image parameters.
queryStatus = QUERY_IMAGE(file, imageInfo)

; Set the image size parameter from the query
; information.
imageSize = imageInfo.dimensions

; Import the image.
image = READ_IMAGE(file)

; Extract the channels (as images) from the RGB image.
redChannel = REFORM(image[0, *, *])
greenChannel = REFORM(image[1, *, *])
blueChannel = REFORM(image[2, *, *])

; Horizontally display the channels.

; Initialize the display objects.
oWindow = OBJ_NEW('IDLgrWindow', RETAIN = 2, $
   DIMENSIONS = imageSize*[3, 1], $
   TITLE = 'The Channels of an RGB Image')
oView = OBJ_NEW('IDLgrView', $
   VIEWPLANE_RECT = [0., 0., imageSize]*[0, 0, 3, 1])
oModel = OBJ_NEW('IDLgrModel')

; Initialize the image objects.
oRedChannel = OBJ_NEW('IDLgrImage', redChannel)
oGreenChannel = OBJ_NEW('IDLgrImage', greenChannel, $
   LOCATION = [imageSize[0], 0])
oBlueChannel = OBJ_NEW('IDLgrImage', blueChannel, $
   LOCATION = [2*imageSize[0], 0])

; Add the image objects to the model, which is added to
; the view, then display the view in the window.
oModel -> Add, oRedChannel
oModel -> Add, oGreenChannel
oModel -> Add, oBlueChannel
oView -> Add, oModel
oWindow -> Draw, oView

; Vertically display the channels.

; Initialize another window object.
oWindow = OBJ_NEW('IDLgrWindow', RETAIN = 2, $
   DIMENSIONS = imageSize*[1, 3], $
   TITLE = 'The Channels of an RGB Image')

; Change the view from horizontal to vertical.
oView -> SetProperty, $
   VIEWPLANE_RECT = [0., 0., imageSize]*[0, 0, 1, 3]

; Change the locations of the channels.
oGreenChannel -> SetProperty, $
   LOCATION = [0, imageSize[1]]
oBlueChannel -> SetProperty, $
   LOCATION = [0, 2*imageSize[1]]

; Display the updated view in the new window.
oWindow -> Draw, oView

; Diagonally display the channels.

; Initialize another window object.
oWindow = OBJ_NEW('IDLgrWindow', RETAIN = 2, $
   DIMENSIONS = imageSize*[2, 2], $
   TITLE = 'The Channels of an RGB Image')

; Change the view from vertical to diagonal.
oView -> SetProperty, $
   VIEWPLANE_RECT = [0., 0., imageSize]*[0, 0, 2, 2]

; Change the locations of the channels.
oGreenChannel -> SetProperty, $
   LOCATION = [imageSize[0]/2, imageSize[1]/2]
oBlueChannel -> SetProperty, $
   LOCATION = [imageSize[0], imageSize[1]]

; Display the updated view in the new window.
oWindow -> Draw, oView

; Cleanup object references.
OBJ_DESTROY, [oView]

END
```

Zooming in on an Image

Enlarging on a specific section of an image is known as zooming in on an image. How zooming is performed within IDL depends on the graphics system. In Direct Graphics, you can use the ZOOM procedure to zoom in on a specific section of an image, see "Zooming in on a Direct Graphics Image Display" for more information. If you are working with RGB images, you can use the ZOOM_24 procedure.

In Object Graphics, the VIEWPLANE_RECT keyword is used to change the view object. The entire image is still contained within the image object while the view is changed to only show specific areas of the image object, see "Zooming in on a Object Graphics Image Display" on page 50 for more information.

Zooming in on a Direct Graphics Image Display

The following example imports a grayscale image from the `convec.dat` binary file. This grayscale image shows the convection of the Earth's mantle. The image contains byte data values and is 248 pixels by 248 pixels. The ZOOM procedure, which is a Direct Graphics routine, is used to zoom in on the lower left corner of the image.

For code that you can copy and paste into an Editor window, see "Example Code: Zooming in Direct Graphics" on page 50 or complete the following steps for a detailed description of the process.

1. Determine the path to the `convec.dat` file:

```
file = FILEPATH('convec.dat', $
   SUBDIRECTORY = ['examples', 'data'])
```

2. Initialize the image size parameter:

```
imageSize = [248, 248]
```

3. Import the image from the file:

```
image = READ_BINARY(file, DATA_DIMS = imageSize)
```

4. If you are running IDL on a TrueColor display, set the DECOMPOSED keyword to the DEVICE command to zero before your first color table related routine is used within an IDL session or program. See "How Colors are Associated with Indexed and RGB Images" for more information.

```
DEVICE, DECOMPOSED = 0
```

5. Load a grayscale color table:

```
LOADCT, 0
```

6. Create a window and display the original image with the TV procedure:

   ```
   WINDOW, 1, XSIZE = imageSize[0], YSIZE = imageSize[1], $
      TITLE = 'A Grayscale Image'
   TV, image
   ```

   The following figure shows the resulting grayscale image display.

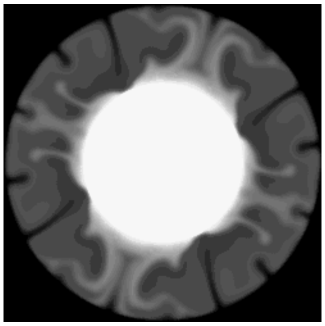


Figure 17: A Grayscale Image in Direct Graphics

7. Use ZOOM to enlarge the lower left quarter of the image:

   ```
   ZOOM, /NEW_WINDOW, FACT = 2, $
      XSIZE = imageSize[0], YSIZE = imageSize[1]
   ```

   Click in the lower left corner of the original image window.

   The following figure shows the resulting zoomed image.

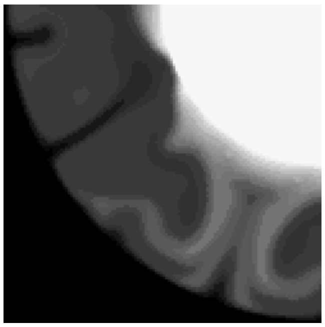


Figure 18: Enlarged Image Area in Direct Graphics

8. Right-click in the original image window to quit out of the ZOOM procedure.

Example Code: Zooming in Direct Graphics

Copy and paste the following text into the IDL Editor window. After saving the file as `Zooming_Direct.pro`, compile and run the program to reproduce the previous example.

```
PRO Zooming_Direct

; Determine the path to the file.
file = FILEPATH('convec.dat', $
   SUBDIRECTORY = ['examples', 'data'])

; Initialize the image size parameter.
imageSize = [248, 248]

; Import in the image from the file.
image = READ_BINARY(file, DATA_DIMS = imageSize)

; Initialize the display.
DEVICE, DECOMPOSED = 0
LOADCT, 0

; Create a window and display the image.
WINDOW, 1, XSIZE = imageSize[0], YSIZE = imageSize[1], $
   TITLE = 'A Grayscale Image'
TV, image

; Zoom into the lower left quarter of the image.
ZOOM, /NEW_WINDOW, FACT = 2, $
   XSIZE = imageSize[0], YSIZE = imageSize[1]

END
```

Zooming in on a Object Graphics Image Display

The following example imports a grayscale image from the `convec.dat` binary file. This grayscale image shows the convection of the Earth’s mantle. The image contains byte data values and is 248 pixels by 248 pixels. The VIEWPLANE_RECT keyword to the view object is updated to zoom in on the lower left corner of the image.

For code that you can copy and paste into an Editor window, see “Example Code: Zooming in Object Graphics” on page 52 or complete the following steps for a detailed description of the process.

1. Determine the path to the `convec.dat` file:

```
file = FILEPATH('convec.dat', $
   SUBDIRECTORY = ['examples', 'data'])
```

2. Initialize the image size parameter:

```
imageSize = [248, 248]
```

3. Import the image from the file:

```
image = READ_BINARY(file, DATA_DIMS = imageSize)
```

4. Initialize the display objects:

```
oWindow = OBJ_NEW('IDLgrWindow', RETAIN = 2, $
   DIMENSIONS = imageSize, $
   TITLE = 'A Grayscale Image')
oView = OBJ_NEW('IDLgrView', $
   VIEWPLANE_RECT = [0., 0., imageSize])
oModel = OBJ_NEW('IDLgrModel')
```

5. Initialize the image object:

```
oImage = OBJ_NEW('IDLgrImage', image, /GREYSCALE)
```

6. Add the image object to the model, which is added to the view, then display the view in the window:

```
oModel -> Add, oImage
oView -> Add, oModel
oWindow -> Draw, oView
```

The following figure shows the resulting grayscale image display.

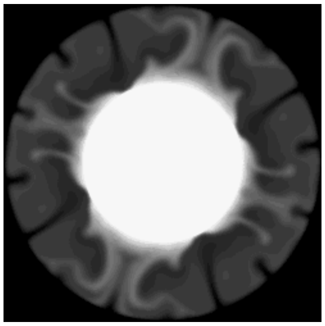


Figure 19: A Grayscale Image in Object Graphics

7. Initialize another window:

```
oWindow = OBJ_NEW('IDLgrWindow', RETAIN = 2, $
   DIMENSIONS = imageSize, TITLE = 'Zoomed Image')
```

8. Change the view to enlarge the lower left quarter of the image:

```
oView -> SetProperty, $
   VIEWPLANE_RECT = [0., 0., imageSize/2]
```

   The view object still contains the entire image object, but the region displayed by the view (the viewplane rectangle) is reduced in size by half in both directions. Since the window object remains the same size, the view region is enlarged to fit it to the window.

9. Display the updated view in the new window:

```
oWindow -> Draw, oView
```

   The following figure shows the resulting zoomed image.

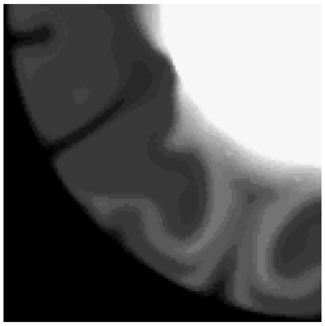


Figure 20: Enlarged Image Area in Object Graphics

10. Cleanup the object references. When working with objects always remember to cleanup any object references with the OBJ_DESTROY routine. Since the view contains all the other objects, except for the window (which is destroyed by the user), you only need to use OBJ_DESTROY on the view object.

```
OBJ_DESTROY, [oView]
```

Example Code: Zooming in Object Graphics

Copy and paste the following text into the IDL Editor window. After saving the file as `Zooming_Object.pro`, compile and run the program to reproduce the previous example.

```
PRO Zooming_Object

; Determine the path to the file.
file = FILEPATH('convec.dat', $
   SUBDIRECTORY = ['examples', 'data'])

; Initialize the image size parameter.
imageSize = [248, 248]

; Import in the image from the file.
image = READ_BINARY(file, DATA_DIMS = imageSize)

; Initialize display objects.
oWindow = OBJ_NEW('IDLgrWindow', RETAIN = 2, $
   DIMENSIONS = imageSize, $
   TITLE = 'A Grayscale Image')
oView = OBJ_NEW('IDLgrView', $
   VIEWPLANE_RECT = [0., 0., imageSize])
oModel = OBJ_NEW('IDLgrModel')

; Initialize image object.
oImage = OBJ_NEW('IDLgrImage', image, /GREYSCALE)

; Add the image object to the model, which is added to
; the view, then display the view in the window.
oModel -> Add, oImage
oView -> Add, oModel
oWindow -> Draw, oView

; Initialize another window.
oWindow = OBJ_NEW('IDLgrWindow', RETAIN = 2, $
   DIMENSIONS = imageSize, TITLE = 'Enlarged Area')

; Change view to zoom into the lower left quarter of
; the image.
oView -> SetProperty, $
   VIEWPLANE_RECT = [0., 0., imageSize/2]

; Display updated view in new window.
oWindow -> Draw, oView

; Cleanup object references.
OBJ_DESTROY, [oView]

END
```

Panning within an Image

Panning involves moving an area of focus from one section of an image to other sections. How panning is performed within IDL depends on the graphics system. In Direct Graphics, you can use the SLIDE_IMAGE procedure to pan with sliders in an application that contains the image, see "Panning in Direct Graphics" for more information.

In Object Graphics, the VIEWPLANE_RECT keyword is used to change the view object. The entire image is still contained within the image object, but the view is changed to pan over specific areas of the image object, see "Panning in Object Graphics" on page 56 for more information.

Panning in Direct Graphics

The following example imports a grayscale image from the `nyny.dat` binary file. This grayscale image is an aerial view of New York City. The image contains byte data values and is 768 pixels by 512 pixels. You can use the SLIDE_IMAGE procedure to zoom in on the image and pan over it.

For code that you can copy and paste into an Editor window, see "Example Code: Panning in Direct Graphics" on page 55 or complete the following steps for a detailed description of the process.

1. Determine the path to the `convec.dat` file:

   ```
   file = FILEPATH('nyny.dat', $
      SUBDIRECTORY = ['examples', 'data'])
   ```

2. Initialize the image size parameter:

   ```
   imageSize = [768, 512]
   ```

3. Import the image from the file:

   ```
   image = READ_BINARY(file, DATA_DIMS = imageSize)
   ```

4. If you are running IDL on a TrueColor display, set the DECOMPOSED keyword to the DEVICE command to zero before your first color table related routine is used within an IDL session or program. See "How Colors are Associated with Indexed and RGB Images" for more information.

   ```
   DEVICE, DECOMPOSED = 0
   ```

5. Load a grayscale color table:

   ```
   LOADCT, 0
   ```

6. Display the image with the SLIDE_IMAGE procedure:

   ```
   SLIDE_IMAGE, image
   ```

   Use the sliders in the display on the right side to pan over the image.

   The following figure shows a possible display within the SLIDE_IMAGE application.

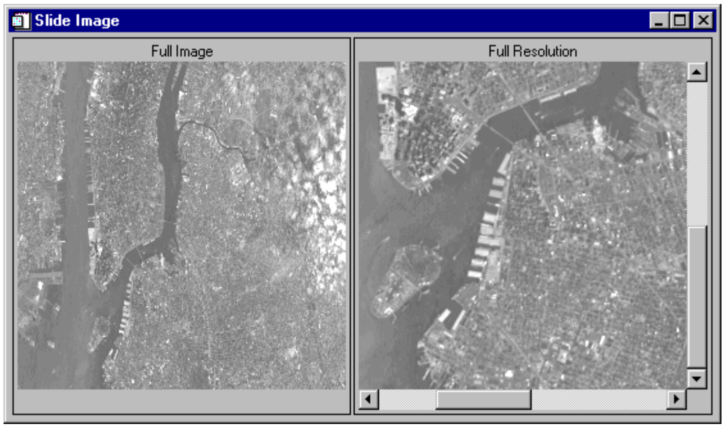


Figure 21: The SLIDE_IMAGE Application Displaying an Image of New York

Example Code: Panning in Direct Graphics

Copy and paste the following text into the IDL Editor window. After saving the file as `Panning_Direct.pro`, compile and run the program to reproduce the previous example.

```
PRO Panning_Direct

; Determine the path to the file.
file = FILEPATH('nyny.dat', $
   SUBDIRECTORY = ['examples', 'data'])

; Initialize the image size parameter.
imageSize = [768, 512]

; Import in the image from the file.
image = READ_BINARY(file, DATA_DIMS = imageSize)

; Initialize the display.
DEVICE, DECOMPOSED = 0
```

```
LOADCT, 0

; Display the image with the SLIDE_IMAGE procedure.
SLIDE_IMAGE, image

END
```

Panning in Object Graphics

The following example imports a grayscale image from the `nyny.dat` binary file. This grayscale image is an aerial view of New York City. The image contains byte data values and is 768 pixels by 512 pixels. The VIEWPLANE_RECT keyword to the view object is updated to zoom in on the lower left corner of the image. Then the VIEWPLANE_RECT keyword is used to pan over the bottom edge of the image.

For code that you can copy and paste into an Editor window, see "Example Code: Panning in Object Graphics" on page 59 or complete the following steps for a detailed description of the process.

1. Determine the path to the `convec.dat` file:

   ```
   file = FILEPATH('nyny.dat', $
      SUBDIRECTORY = ['examples', 'data'])
   ```

2. Initialize the image size parameter:

   ```
   imageSize = [768, 512]
   ```

3. Import the image from the file:

   ```
   image = READ_BINARY(file, DATA_DIMS = imageSize)
   ```

4. Resize this large image to entirely display it on the screen:

   ```
   imageSize = [256, 256]
   image = CONGRID(image, imageSize[0], imageSize[1])
   ```

5. Initialize the display objects:

   ```
   oWindow = OBJ_NEW('IDLgrWindow', RETAIN = 2, $
      DIMENSIONS = imageSize, $
      TITLE = 'A Grayscale Image')
   oView = OBJ_NEW('IDLgrView', $
      VIEWPLANE_RECT = [0., 0., imageSize])
   oModel = OBJ_NEW('IDLgrModel')
   ```

6. Initialize the image object:

   ```
   oImage = OBJ_NEW('IDLgrImage', image, /GREYSCALE)
   ```

7. Add the image object to the model, which is added to the view, then display the view in the window:

   ```
   oModel -> Add, oImage
   oView -> Add, oModel
   oWindow -> Draw, oView
   ```

   The following figure shows the resulting grayscale image display.

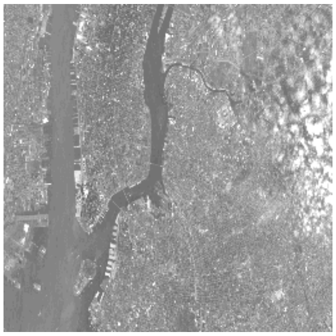


Figure 22: A Grayscale Image Of New York in Object Graphics

8. Initialize another window:

   ```
   oWindow = OBJ_NEW('IDLgrWindow', RETAIN = 2, $
      DIMENSIONS = imageSize, TITLE = 'Zoomed Image')
   ```

9. Change the view to zoom into the lower left quarter of the image:

   ```
   oView -> SetProperty, $
      VIEWPLANE_RECT = [0., 0., imageSize/2]
   ```

   The view object still contains the entire image object, but the region displayed by the view (the viewplane rectangle) is reduced in size by half in both directions. Since the window object remains the same size, the view region is enlarged to fit it to the window.

10. Display the updated view in the new window:

    ```
    oWindow -> Draw, oView
    ```

The following figure shows the resulting enlarged image area.

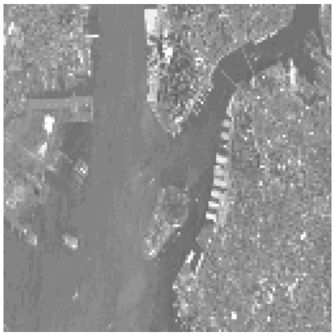


Figure 23: Enlarged Image Area of New York in Object Graphics

11. Pan the view from the left side of the image to the right side of the image:

```
FOR i = 0, ((imageSize[0]/2) - 1) DO BEGIN & $
   viewplane = viewplane + [1., 0., 0., 0.] & $
   oView -> SetProperty, VIEWPLANE_RECT = viewplane & $
   oWindow -> Draw, oView & $
ENDFOR
```

Note

The $ after BEGIN and the & allow you to use the FOR/DO loop at the IDL command line. These $ and & symbols are not required when the FOR/DO loop in placed in an IDL program as shown in “Example Code: Panning in Object Graphics” on page 59.

The following figure shows the resulting enlarged image area panned to the right side.

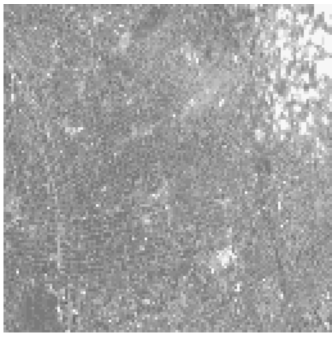


Figure 24: Enlarged New York Image Area Panned to the Right in Object Graphics

12. Cleanup the object references. When working with objects always remember to cleanup any object references with the OBJ_DESTROY routine. Since the view contains all the other objects, except for the window (which is destroyed by the user), you only need to use OBJ_DESTROY on the view object.

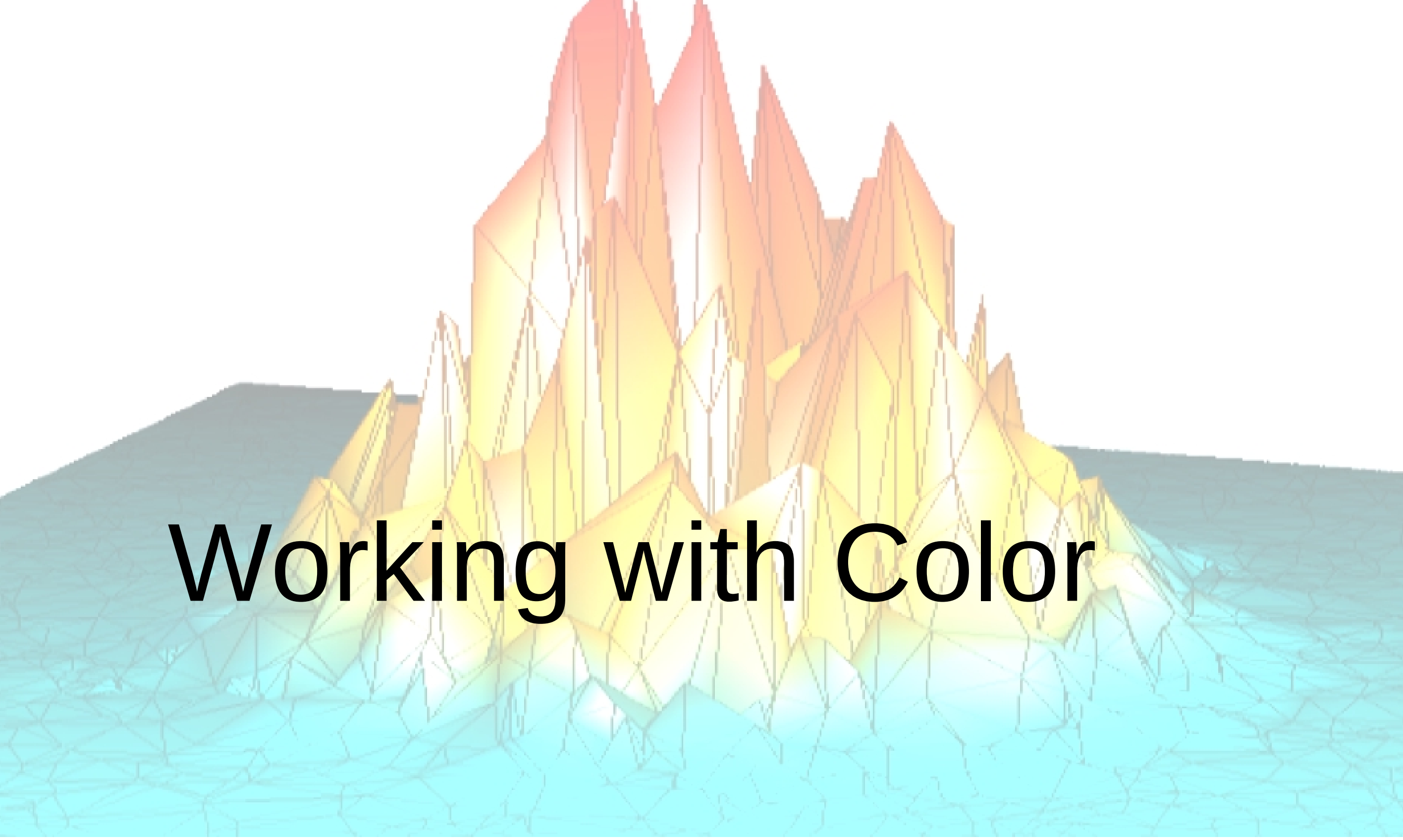
```
OBJ_DESTROY, [oView]
```

Example Code: Panning in Object Graphics

Copy and paste the following text into the IDL Editor window. After saving the file as `Panning_Object.pro`, compile and run the program to reproduce the previous example.

```
PRO Panning_Object

; Determine the path to the file.
file = FILEPATH('nyny.dat', $
   SUBDIRECTORY = ['examples', 'data'])

; Initialize the image size parameter.
imageSize = [768, 512]

; Import in the image from the file.
image = READ_BINARY(file, DATA_DIMS = imageSize)

; Resize the image.
imageSize = [256, 256]
```

```
image = CONGRID(image, imageSize[0], imageSize[1])

; Initialize display objects.
oWindow = OBJ_NEW('IDLgrWindow', RETAIN = 2, $
   DIMENSIONS = imageSize, $
   TITLE = 'A Grayscale Image')
oView = OBJ_NEW('IDLgrView', $
   VIEWPLANE_RECT = [0., 0., imageSize])
oModel = OBJ_NEW('IDLgrModel')

; Initialize image object.
oImage = OBJ_NEW('IDLgrImage', image, /GREYSCALE)

; Add the image object to the model, which is added to
; the view, then display the view in the window.
oModel -> Add, oImage
oView -> Add, oModel
oWindow -> Draw, oView

; Initialize another window.
oWindow = OBJ_NEW('IDLgrWindow', RETAIN = 2, $
   DIMENSIONS = imageSize, $
   TITLE = 'Panning Enlarged Image')

; Change view to zoom into the lower left quarter of
; the image.
viewplane = [0., 0., imageSize/2]
oView -> SetProperty, VIEWPLANE_RECT = viewplane

; Display updated view in new window.
oWindow -> Draw, oView

; Pan the view from the left side of the image to the
; right side of the image.
FOR i = 0, ((imageSize[0]/2) - 1) DO BEGIN
   viewplane = viewplane + [1., 0., 0., 0.]
   oView -> SetProperty, VIEWPLANE_RECT = viewplane
   oWindow -> Draw, oView
ENDFOR

; Cleanup object references.
OBJ_DESTROY, [oView]

END
```



Working with Color

This chapter describes the following topics:

Overview of Working with Color

An overview of color systems, display devices, image types, and IDL's interaction with these elements provide a basis for understanding how to effectively extract information from your images.

Color Systems

Although numerous ways of specifying and measuring color exist, most color systems are three-dimensional in nature, relying on three values to define each individual color. Common color systems include RGB (red, green, and blue), HSV (hue, saturation, and value), HLS (hue, lightness, and saturation), and CMY (cyan, magenta, and yellow).

Computer display devices typically rely on the RGB color system. In IDL, the RGB color system is implemented with a three-dimensional coordinate system with the red, green, and blue dimensions ranging from 0 to 255 (the range of a byte data type). Each individual color definition consists of three numbers; a red value, a green value, and a blue value.

The following figure shows that each displayable color corresponds to a location within a three-dimensional color cube. The origin, (0, 0, 0), where each color coordinate is 0, is black. The point at (255, 255, 255) is white and represents an additive mixture of the full intensity of each of the three colors. Points along the main diagonal—where the intensities of each of the three primary colors are equal—are shades of gray. The color yellow is represented by the coordinate (255, 255, 0), or a mixture of 100% red, plus 100% green, and no blue.

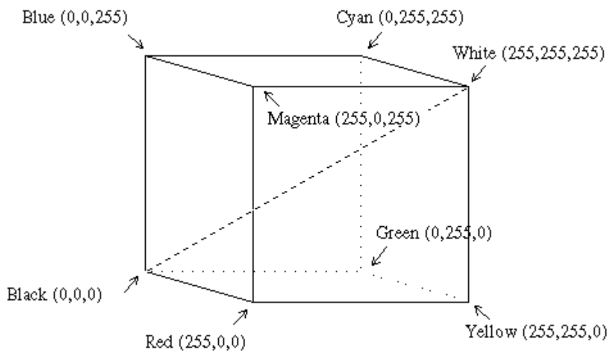


Figure 1: RGB Color Cube (Note: grays are on the main diagonal.)

Color Visuals: PseudoColor Versus TrueColor

Typically, digital display devices represent each component of an RGB color coordinate as an *n*-bit integer in the range of 0 to 2^n –1. Each displayable color is an RGB coordinate triple of *n*-bit numbers yielding a palette containing 2^n total colors. Colors displayed with 8-bit numbers (PseudoColor visuals) yield 2^8 or 256 colors. Colors displayed with 24-bit numbers (TrueColor visuals) yield 2^{24} or 16,777,216 colors. The term visual refers to the visual attribute setup.

A color is specified within a PseudoColor (8-bit) visual as an index into a Look-Up Table (LUT). The table contains RGB triples for 256 colors provided within a PseudoColor visual. A color is specified within a TrueColor (24-bit) visual as a single integer value derived from the following equation:

$$\mathrm{value} = \mathrm{red} + 256\mathrm{green} + 256^2\mathrm{blue}$$

where *red*, *green*, and *blue* are either scalars of values ranging from 0 to 255 that represent the amount of red, green, and blue in the RGB triple.

PseudoColor visuals have been available in the marketplace for a longer period of time than TrueColor visuals. PseudoColor visuals require less system memory to run, but TrueColor visuals support more colors. PseudoColor visuals automatically update the colors of a window. TrueColor visuals do not automatically update (the window must be re-drawn to update colors), but this static behavior allows you to apply different colors to different windows. With PseudoColor visuals, every IDL window must use the same colors.

The type of visual (PseudoColor or TrueColor) IDL uses by default depends more on your computer system than IDL. On X Windows systems, IDL tries to set a TrueColor visual before a PseudoColor visual. If a TrueColor visual is not provided by the computer system, then IDL will set a PseudoColor visual. Also on X Windows systems, the visual can be specified with the PSEUDO and TRUE keywords to the DEVICE command before the first window is opened within the IDL session. On Windows and Macintosh systems, the type of visual depends upon the system setting.

Note

This chapter assumes a TrueColor display is used. If a PseudoColor display will provide a different behavior, this change in behavior will be noted.

How Colors are Associated with Indexed and RGB Images

Common images are either indexed or RGB (red, green, and blue). An indexed image is a two-dimensional array of values ranging from 0 to 255. Indexed images are usually stored as byte data. A two-dimensional array of a different data type can be made into an indexed image by scaling it to range from 0 to 255 with the BYTSCL function. See BYTSCL in the *IDL Reference Guide* for more information.

An indexed image does not explicitly contain any color information. Its pixel values represent indices into a color Look-Up Table (LUT). Colors are applied by using these indices to look up the corresponding RGB triple in the LUT. In some cases, the pixel values of an indexed image reflect the relative intensity of each pixel. In other cases, each pixel value is simply an index, in which case the image is usually intended to be associated with a specific LUT.

An RGB (red, green, and blue) image is a three-dimensional byte array, which does maintain its own color information. In an RGB image, each pixel is represented by three values (a triple) to define its color. Computers commonly store scanned photographs as RGB images. The color information is stored in three sections of a third dimension of the image. These sections are known as color channels, color bands, or color layers. One channel represents the amount of red in the image (the red channel), one channel represents the amount of green in the image (the green channel), and one channel represents the amount of blue in the images (the blue channel).

Color interleaving is a term used to describe which of the dimensions of an RGB image contain the three color channel values. Three types of color interleaving are supported by IDL:

- Pixel interleaving - the color information is contained in the first dimension, (3, n, m).
- Line interleaving - the color information is contained in the second dimension, (n, 3, m).
- Image interleaving - the color information is contained in the third dimension, (n, m, 3).

For PseudoColor visuals and indexed images, the LUT (usually just known as a color table or color palette) is used to associate the value of a pixel with a color triple. Given 8-bit pixels (found in many PseudoColor visuals), a color table containing 2^8 = 256 elements is required. The color table element with an index of *i* specifies the color for pixels with a value of *i*. Since indexed images do not maintain their own color information, these images are usually associated with color tables to provide color information. Some indexed images are even saved to image files with an associated color table.

Within IDL, a color table is represented by either a 3 by 256 byte array or three byte vectors of 256 elements each. In other words, a color table contains 256 RGB triples. The first component (column) in the 3 by 256 array is the red values of each triple, the second component is the green values, and the third component is the blue values. This array or these vectors can be created or changed to provide different color tables within IDL. IDL provides 41 pre-defined color tables.

PseudoColor visuals are only associated with indexed images and color tables. You can display RGB images on PseudoColor visuals with IDL by using the COLOR_QUAN function. This function creates a color table for displaying the RGB image and then maps this image to the new palette. See "Converting RGB Images to Indexed Images" on page 93 and `COLOR_QUAN` in the *IDL Reference Guide* for more information.

Color tables were derived for PseudoColor visuals. TrueColor visuals (by default) do not typically utilize color tables. You can use color tables on TrueColor visuals of IDL by setting the DECOMPOSED keyword to the DEVICE routine to a value of zero (`DEVICE, DECOMPOSED = 0`). IDL must be instructed (by the user) to apply color tables on TrueColor visuals. If you do not want a color table applied to a window within a TrueColor visuals (when displaying an RGB image for example), set the DECOMPOSED keyword to one (`DEVICE, DECOMPOSED = 1`). You can use the following table to determine how to set up your screen device per the type of images you are displaying and your default visual.

Visual	Indexed Images	RGB Images
PseudoColor	N/A*	use COLOR_QUAN function
TrueColor	DEVICE, DECOMPOSED = 0	DEVICE, DECOMPOSED = 1

Table 1: Setting up the Display per Visual and Type of Image

* You can set DEVICE, DECOMPOSED = 0 in a PseudoColor visual, but the command will be ignored.

Also see “Keywords Accepted by the IDL Devices” in Appendix B in the *IDL Reference Guide* for more information.

Chapter Overview

The following list describes the color image display tasks and associated IDL image color display routines covered in this chapter.

Tasks	Routine(s)	Description
“Loading Pre-defined Color Tables” on page 68.	LOADCT XLOADCT	Load and view one of IDL’s pre-defined color tables.
“Modifying and Converting Color Tables” on page 71.	XLOADCT XPALETTE TVLCT MODIFYCT HLS HSV COLOR_CONVERT	Use the XLOADCT and XPALETTE utilities to modify a color table and apply it to an image. Save this new color table as one of IDL’s pre-defined tables.

Table 2: Color Image Display Tasks and Related Color Display Routines

Tasks	Routine(s)	Description
"Converting Between Image Types" on page 89.	TVLCT COLOR_QUAN	Change an indexed image with an associated color table to an RGB image, and vice versa.
"Highlighting Features with a Color Table" on page 97.	TVLCT IDLgrPalette IDLgrImage	Create an entire color table to highlight features within an image.
"Showing Variations in Uniform Areas" on page 108.	H_EQ_CT H_EQ_INT TVLCT	Modify a color table with histogram equalization to display minor variations in nearly uniform areas of an image.
"Applying Color Annotations to Images" on page 116.	TVLCT IDLgrPalette	Apply specific colors to annotations on indexed or RGB images to highlight certain features within these images.

Table 2: Color Image Display Tasks and Related Color Display Routines

Note

This chapter uses data files from the IDL `x.x/examples/data` directory. Two files, `data.txt` and `index.txt`, contain descriptions of these files, including array sizes.

Loading Pre-defined Color Tables

Although you can define your own color tables, IDL provides 41 pre-defined color tables. You can access these tables through the LOADCT routine. Each color table contained within this routine is specified through an index value ranging from 0 to 40.

Tip

If you are running IDL on a TrueColor display, set `DEVICE, DECOMPOSED = 0` before your first color table related routine is used within an IDL session or program. See "How Colors are Associated with Indexed and RGB Images" on page 64 for more information.

1. View a list of IDL's tables and their related indices by calling LOADCT without an argument:

   ```
   LOADCT
   ```

   The following list is displayed in the Output Log:

   ```
   % Compiled module: LOADCT.
   % Compiled module: FILEPATH.
   0-B-W LIMEAR           14-STEPS                 28-Hardcandy
   1-BLUE/WHITE           15-STERN SPECIAL         29-Nature
   2-GRN-RED-BLU-WHT      16-Haze                  30-Ocean
   3-RED TEMPERATURE      17-Blue-Pastel-Red       31-Peppermint
   4-BLU/GRN/RED/YEL      18-Pastels               32-Plasma
   5-STD GAMMA-II         19-Hue Sat Lightness 1   33-Blue-Red
   6-PRISM                20-Hue Sat Lightness 2   34-Rainbow
   7-RED-PURPLE           21-Hue Sat Value 1       35-Blue Waves
   8-GREEN/WHITE LINEAR   22-Hue Sat Value 2       36-Volcano
   9-GRN/WHT EXPOMENTIAL  23-Purple-Red+Stripes    37-Waves
   10-GREEN-PINK          24-Beach                 38-Rainbow18
   11-BLUE_RED            25-Mac Style             39-Rainbow+white
   12-16 LEVEL            26-Eos A                 40-Rainbow+black
   13-RAINBOW             27-Eos B
   ```

   When running LOADCT without an argument, it will prompt you to enter the number of one of the above color tables at the IDL command line.

2. Enter in the number 5 at the `Enter table number:` prompt:

   ```
   Enter table number: 5
   ```

   The following text is displayed in the Output Log:

   ```
   % LOADCT: Loading table STD GAMMA-II
   ```

   If you already know the number of the pre-defined color table you want, you can load a color table by providing that number as the first input argument to LOADCT.

3. Load in color table number 13 (RAINBOW):

   ```
   LOADCT, 13
   ```

   The following text is displayed in the Output Log:

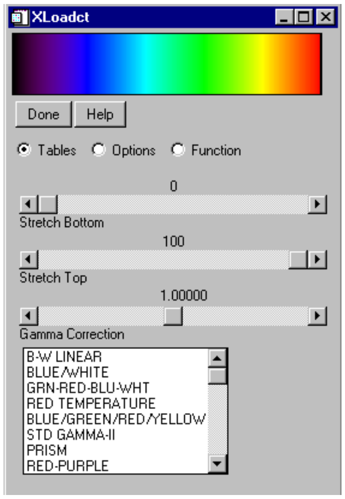
   ```
   % LOADCT: Loading table RAINBOW
   ```

   You can view the current color table with the XLOADCT utility.

4. View color table with XLOADCT utility:

   ```
   XLOADCT
   ```

   The following figure shows the resulting XLOADCT display.



[✓ Tables] Options Function

0
Stretch Bottom
100
Stretch Top
1.00000
Gamma Correction

B-W LINEAR
BLUE/WHITE
GRN-RED-BLU-WHT
RED TEMPERATURE
BLUE/GREEN/RED/YELLOW
STD GAMMA-II
PRISM
RED-PURPLE

Figure 2: The XLOADCT Utility

This utility is designed to individually display each pre-defined color table. When the **Done** button is pressed, the selected color table automatically becomes IDL’s current color table. IDL maintains a color table on PseudoColor displays or when the DECOMPOSED keyword to the DEVICE command is set to zero (`DEVICE, DECOMPOSED = 0`) on TrueColor displays. XLOADCT also allows you to make adjustments to the current color table. Among other options, you can stretch the bottom, stretch the top, or apply a gamma correction factor. See the next section, “Modifying and Converting Color Tables” on page 71, for more information.

Modifying and Converting Color Tables

IDL contains two graphical user interface (GUI) utilities for modifying a color table, XLOADCT and XPALETTE. "Using the XLOADCT Utility" (below) and "Using the XPALETTE Utility" on page 81 describe how to use these utilities to modify color tables. See "Highlighting Features with a Color Table" on page 97 for more information on how to programmatically (without a GUI utility) modify and design a color table. Then the "Using the MODIFYCT Routine" on page 87 section shows how to add the changed color table from XLOADCT and XPALETTE to IDL's list of pre-defined color tables.

The following examples are based on the default RGB (red, green, and blue) color system. IDL also contains routines that allow you to use other color systems including hue, saturation, and value (HSV) and hue, lightness, and saturation (HLS). These routines and color systems are explained in "Converting to Other Color Systems" on page 88.

Using the XLOADCT Utility

The XLOADCT utility allows you to load one of IDL's 41 pre-defined color tables and change that color table if necessary. The following example shows how to use XLOADCT to load a color table and then change that table to highlight specific features of an image. The indexed image used in this example is a computerized tomography (CT) scan of a human thoracic cavity and is contained (without a default color table) within the `ctscan.dat` file in IDL's `examples/data` directory.

For code that you can copy and paste into an IDL Editor window, see "Example Code: Using the XLOADCT Utility" on page 79 or complete the following steps for a detailed description of the process.

1. Determine the path to the `ctscan.dat` binary file:

```
ctscanFile = FILEPATH('ctscan.dat', $
   SUBDIRECTORY = ['examples', 'data'])
```

2. Initialize the image size parameter:

```
ctscanSize = [256, 256]
```

3. Import in the image from the file:

```
ctscanImage = READ_BINARY(ctscanFile, $
   DATA_DIMS = ctscanSize)
```

4. If you are running IDL on a TrueColor display, set the DECOMPOSED keyword to the DEVICE command to zero before your first color table related routine is used within an IDL session or program. See "How Colors are Associated with Indexed and RGB Images" on page 64 for more information.

   ```
   DEVICE, DECOMPOSED = 0
   ```

   Since the imported image does not have an associated color table, you need to apply a pre-defined color table to display the image.

5. Initialize the display by applying the B-W LINEAR color table (index number 0):

   ```
   LOADCT, 0
   WINDOW, 0, TITLE = 'ctscan.dat', $
      XSIZE = ctscanSize[0], YSIZE = ctscanSize[1]
   ```

6. Display the image using this color table:

   ```
   TV, ctscanImage
   ```

   As the following figure shows, the B-W LINEAR color table does not highlight all of the aspects of this image. The XLOADCT utility can be used to change the color table to highlight more features.

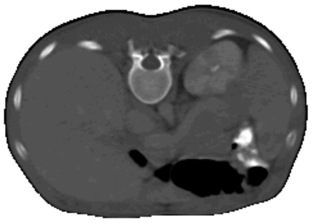


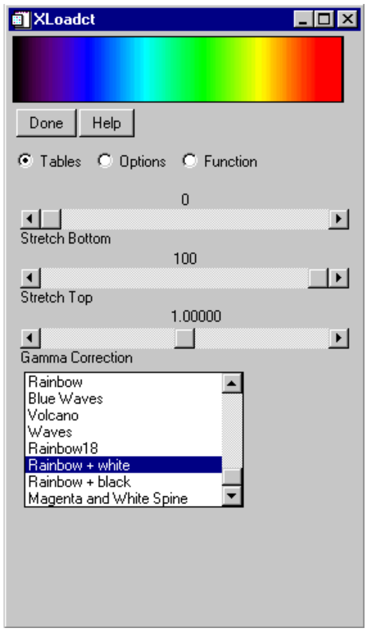
Figure 3: CT Scan Image with Grayscale Color Table

7. Open the XLOADCT utility:

   ```
   XLOADCT
   ```

   Select Rainbow + white and click **Done** to apply the color table.

   The following figure shows the resulting XLOADCT display



[✓ Tables] Options Function

0
Stretch Bottom
100
Stretch Top
1.00000
Gamma Correction

Rainbow
Blue Waves
Volcano
Waves
Rainbow18
[✓ Rainbow + white]
Rainbow + black
Magenta and White Spine

Figure 4: Selecting Rainbow + white Color Table in XLOADCT Utility

After applying the new color table, you can now see the spine, liver, and kidney within the image, as shown in the following figure. However, the separations between the skin, the organs, and the cartilage and bone within the spine are hard to distinguish.

8. Now re-display the image to show it on the Rainbow + white color table:

```
TV, ctscanImage
```

Note

You do not have to perform the previous step on a PseudoColor display. Changes to the current color table automatically show in the current image window within a PseudoColor display.

The following figure shows the CT scan image with the Ranbow+white color table.

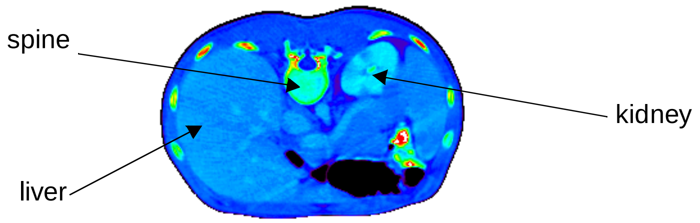


Figure 5: CT Scan Image with the Rainbow + white Color Table

9. Redisplay the color table with the XLOADCT utility:

```
XLOADCT
```

Comparing the image to the color table, you can see that most image pixels are not within the black to purple range. Therefore the black to purple pixels in the image can be replaced by black. The black range can be stretched to move the purple range to help highlight more features.

The **Stretch Bottom** slider in the XLOADCT utility increases the range of the lowest color index. For example, if black was the color of the lowest index and you increased the bottom stretch by 50 percent, the lower half of the color table would become all black. The remaining part of the color table will contain a scaled version of all the previous color ranges.

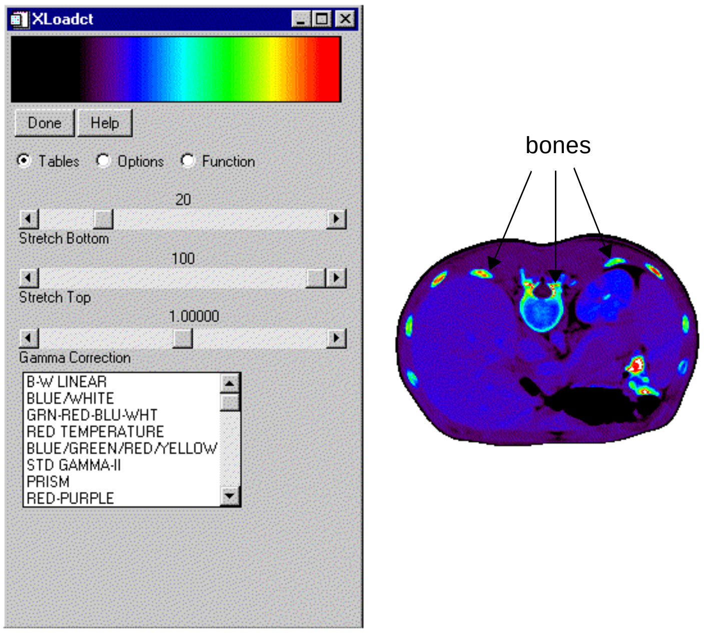
10. Within XLOADCT, stretch the bottom part of the color table by 20 percent by moving the slider as shown in the following figure:

```
TV, ctscanImage
```

Tip

Remember to click on the **Done** button after changing the **Stretch Bottom** slider, then use TV to re-display the image to include the last changing made in the XLOADCT utility.

In the following figure, you can now see the difference between skin and organs. You can also see where cartilage and bone is located within the spine, but now organs are hard to see. Most of the values in the top (the yellow to red to white ranges) of the color table show just the bones. You can use less of these ranges to show bones by stretching the top of the color table.



[✓ Tables] Options Function

Stretch Bottom 20

Stretch Top 100

Gamma Correction 1.00000

Figure 6: CT Scan Image with Bottom Stretched by 20%

The **Stretch Top** slider in the XLOADCT utility allows you increase the range of the highest color index. For example, if white was the color of the highest index and you increased the top stretch by 50 percent, the higher half of the color table would become all white. The remaining part of the color table will contain a scaled version of all the previous color ranges.

11. Open XLOADCT:

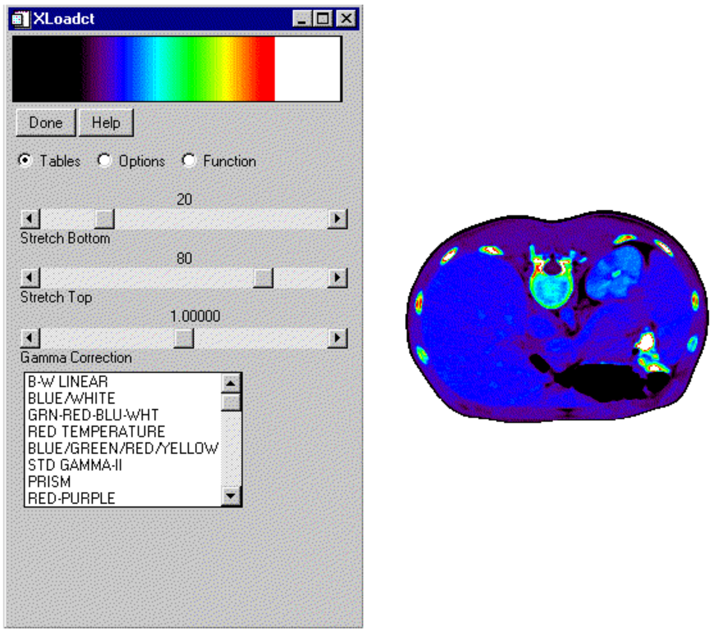
    ```
    XLOADCT
    ```

    Stretch the bottom part of the color table by 20 percent and stretch the top part of the color table by 20 percent (changing it from 100 to 80 percent).

    Click **Done** and redisplay the image:

    ```
    TV, ctscanImage
    ```

    The following figure shows that the organs are more distinctive, but now the liver and kidneys are not clearly distinguished. These features occur in the blue range. You can shift the green range more toward the values of these organs with a gamma correction.



[✓ Tables] Options Function

20
Stretch Bottom

80
Stretch Top

1.00000
Gamma Correction

Figure 7: CT Scan Image with Bottom and Top Stretched by 20%

With the **Gamma Correction** slider in the XLOADCT utility you can change the contrast within the color table. A value of 1.0 indicates a linear ramp (no gamma correction). Values other than 1.0 indicate a logarithmic ramp. Higher values of Gamma give more contrast. Values less than 1.0 yield lower contrast.

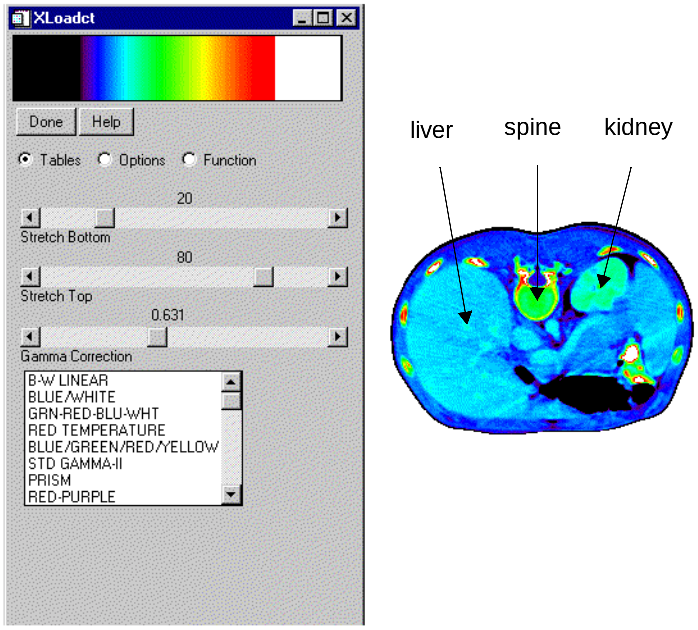
12. Within XLOADCT, stretch the bottom part of the color table by 20 percent, stretch the top part of the color table by 20 percent (change it from 100 percent to 80 percent), and decrease the Gamma Correction factor to 0.631:

```
XLOADCT
```

Redisplay the image:

```
TV, ctscanImage
```

All the features are now highlighted within the image as shown in the following figure:



[✓ Tables] Options Function

20
Stretch Bottom
80
Stretch Top
0.631
Gamma Correction

B-W LINEAR
BLUE/WHITE
GRN-RED-BLU-WHT
RED TEMPERATURE
BLUE/GREEN/RED/YELLOW
STD GAMMA-II
PRISM
RED-PURPLE

Figure 8: CT Scan Image with Bottom and Top Stretched by 20% and Gamma Correction at 0.631

The previous steps showed how to use the **Tables** section of the XLOADCT utility. XLOADCT also contains two other sections: **Options** and **Function**. The **Options** section allows you to change what the sliders represent and how they are used. The sliders can be dependent upon each other when the **Gang** option is selected. When either the **Stretch Bottom** or **Stretch Top** sliders are moved, the other ones reset to their default values (0 or 100, respectively). The top of the color table can be chopped off (the range of the **Stretch Top** is now black instead of the color at the original highest index) with the **Chop** option. The slider can also be changed to control intensity instead of index location with the **Intensity** option. The **Stretch Bottom** slider will darken the color table and the **Stretch Top** slider will brighten the color table.

The **Function** section allows you to place control points which you can use to change the color table with respect to the other colors in that table. The color table function is shown as a straight line increasing from the lowest index (0) to the highest index (255). The x-axis ranges from 0 to 255 and the y-axis ranges from 0 to 255. Moving a control point in the x-direction has the same effects as the previous sliders. Moving a control point in the y-direction changes the color of that index to another color within the color table. For example, if a control point is red at an index of 128 and the color table is green at an index of 92, when the control point is moved in the y-direction to an index of 92, the color at that x-location will become green. To understand how the **Function** section work, you can use it to highlight just the bones with the CT scan image.

13. Open XLOADCT:

    ```
    XLOADCT
    ```

    Select the Rainbow + white color table.

    Switch to the **Function** section by selecting that option.

    Select the **Add Control Point** button, and drag this new center control point one half of the way to the right and one quarter of the way down as shown in the following figure.

    Click **Done** and redisplay the image:

    ```
    TV, ctscanImage
    ```

The bones in the image are now highlighted.

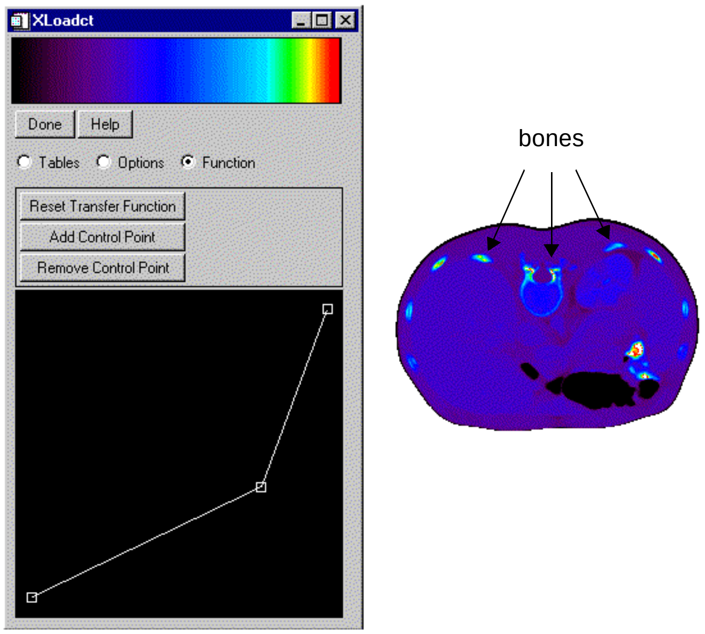


Figure 9: CT Scan Image with Central Control Point Moved One Half to the Right and One Quarter Down

Example Code: Using the XLOADCT Utility

Copy and paste the following text into the IDL Editor window. After saving the file as `UsingXLOADCT.pro`, compile and run the program to reproduce the previous example. The BLOCK keyword is set when using XLOADCT to force the example routine to wait until the **Done** button is pressed to continue. If the BLOCK keyword was not set, the example routine would produce all of the displays at once and then end.

```
PRO UsingXLOADCT

; Determine the path to the file.
ctscanFile = FILEPATH('ctscan.dat', $
   SUBDIRECTORY = ['examples', 'data'])

; Initialize image size parameter.
```

```
ctscanSize = [256, 256]

; Import in the image from the file.
ctscanImage = READ_BINARY(ctscanFile, $
   DATA_DIMS = ctscanSize)

; Initialize display.
DEVICE, DECOMPOSED = 0
LOADCT, 0
WINDOW, 0, TITLE = 'ctscan.dat', $
   XSIZE = ctscanSize[0], YSIZE = ctscanSize[1]

; Display image.
TV, ctscanImage

; Select and display the "Rainbow + white" color
; table
XLOADCT, /BLOCK
TV, ctscanImage

; Increase "Stretch Bottom" by 20%.
XLOADCT, /BLOCK
TV, ctscanImage

; Increase "Stretch Bottom" by 20% and decrease
; "Stretch Top" by 20% (to 80%).
XLOADCT, /BLOCK
TV, ctscanImage

; Increase "Stretch Bottom" by 20%, decrease "Stretch
; Top" by 20% (to 80%), and decrease "Gamma Correction"
; to 0.631.
XLOADCT, /BLOCK
TV, ctscanImage

; Switch to "Function" section, select "Add Control
; Point" and drag this center control point one quarter
; of the way up and one quarter of the way left.
XLOADCT, /BLOCK
TV, ctscanImage

END
```

Using the XPALETTE Utility

Another utility, XPALETTE, can be used to change a specific color table entry or range of entries. This example uses a single color (orange) to highlight pixels within the spine of the CT scan image. Then, starting with the entry that was changed to orange, a range of entries is selected and replaced with a ramp from orange to white to highlight the bones within this image.

For code that you can copy and paste into an IDL Editor window, see "Example Code: Using the XPALETTE Utility" on page 85 or complete the following steps for a detailed description of the process.

1. Determine the path to the `ctscan.dat` binary file:

```
ctscanFile = FILEPATH('ctscan.dat', $
   SUBDIRECTORY = ['examples', 'data'])
```

2. Initialize the image size parameter:

```
ctscanSize = [256, 256]
```

3. Import in the image from the file:

```
ctscanImage = READ_BINARY(ctscanFile, $
   DATA_DIMS = ctscanSize)
```

4. If you are running IDL on a TrueColor display, set the DECOMPOSED keyword to the DEVICE command to zero before your first color table related routine is used within an IDL session or program. See "How Colors are Associated with Indexed and RGB Images" on page 64 for more information.

```
DEVICE, DECOMPOSED = 0
```

5. Display the image from the `ctscan.dat` file with the B-W LINEAR color table:

```
LOADCT, 0
WINDOW, 0, TITLE = 'ctscan.dat', $
   XSIZE = ctscanSize[0], YSIZE = ctscanSize[1]
TV, ctscanImage
```

As shown in the following figure, the B-W LINEAR color table does not distinguish all of the aspects of this image. The XPALETTE utility can be used to change the color table.

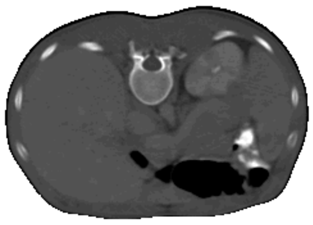


Figure 10: CT Scan Image with Grayscale Color Table

6. Open the XPALETTE utility:

   ```
   XPALETTE
   ```

   Select the **Predefined** button in the XPALETTE utility to change the color table to Rainbow + white.

   Click on the **Done** button after you select the Rainbow + white color table in XLOADCT and then click on the **Done** button in XPALETTE.

The following figure shows the resulting XPALETTE and XLOADCT displays.

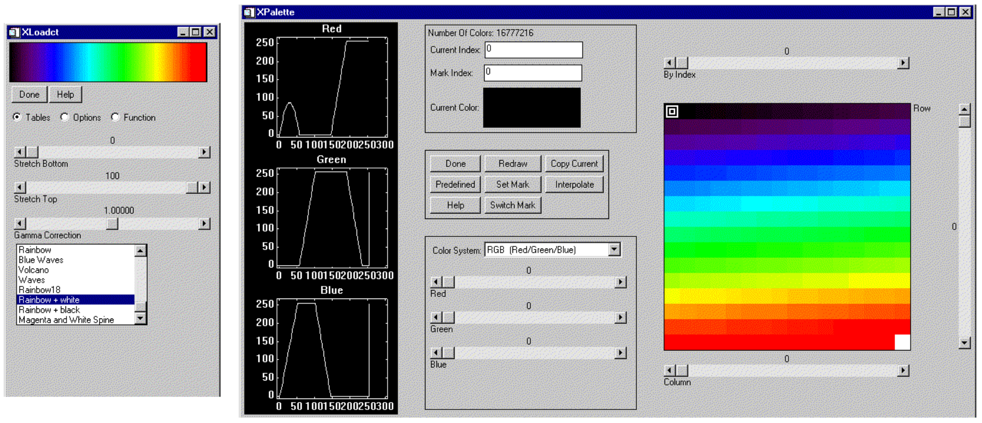


Figure 11: Selecting Rainbow + white Color Table in XPALETTE Utility

7. Now redisplay the image to show it with the Rainbow + white color table:

```
TV, ctscanImage
```

Your display should be similar to the following figure.

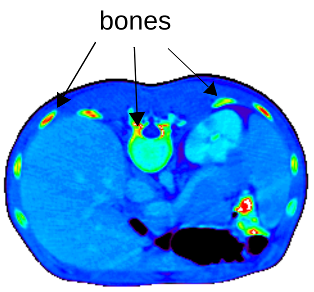


Figure 12: CT Scan Image with the Rainbow + white Color Table

You can use XPALETTE to change a single color within the current color table. For example, you can change the color at index number 115 to orange.

8. Open XPALETTE and click on the 115th index (in column 3 and row 7):

```
XPALETTE
```

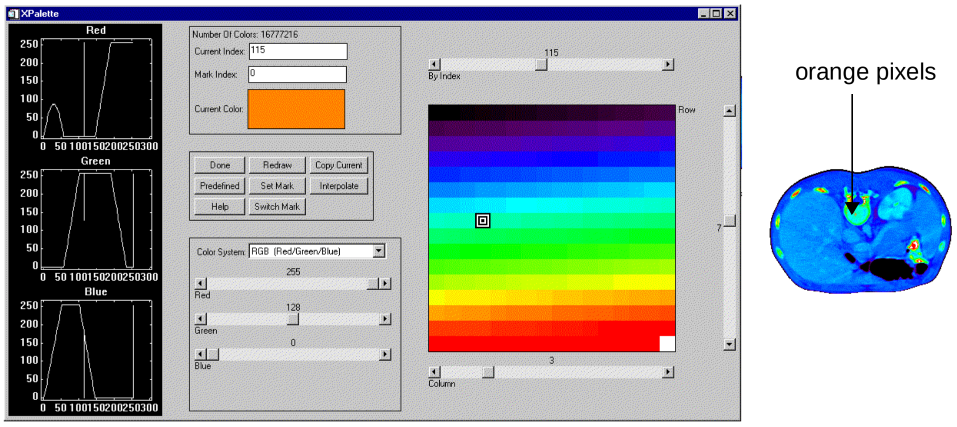
Change its color to orange by moving the RGB (red, green, and blue) sliders (Orange is made up of 255 red, 128 green, and 0 blue)

Click on the **Done** button after changing the **Red**, **Green**, and **Blue** sliders.

Use TV to redisplay the image to include the last changes made in the XPALETTE utility:

```
TV, ctscanImage
```

The orange values now highlight some areas of the spine, kidney, and bones as shown in the following figure.



Number Of Colors: 16777216
Current Index: 115
Mark Index: 0
Current Color:
Color System: RGB (Red/Green/Blue)
Red: 255
Green: 128
Blue: 0
By Index: 115
Row: 7
Column: 3

Figure 13: CT Scan Image with Orange Added to the Color Table

You can highlight the bones even further by interpolating a new range in between the orange and white indices.

9. Open XPALETTE:

   Click on the 115th index and select the Set Mark button.

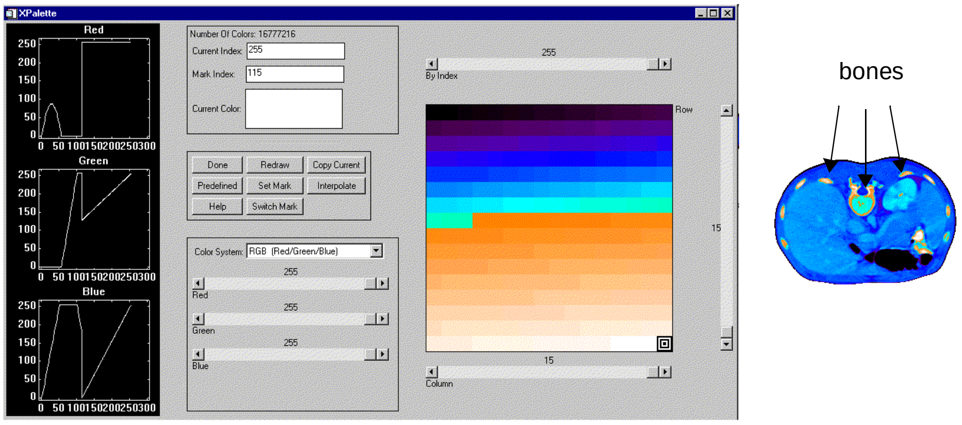
   Click on the highest index (which is usually 255 but it could be less) and then select the **Interpolate** button.

   To see the result of this interpolation within XPALETTE, click on the **Redraw** button.

   Click **Done** and redisplay the image:

   ```
   TV, ctscanImage
   ```

   The following figure displays the image using the modified color table.


XPalette

Number Of Colors: 16777216
Current Index: 255
Mark Index: 115
Color System: RGB (Red/Green/Blue)

Figure 14: CT Scan Image with Orange to White Range Added

Example Code: Using the XPALETTE Utility

Copy and paste the following text into the IDL Editor window. After saving the file as `UsingXPALETTE.pro`, compile and run the program to reproduce the previous example. The BLOCK keyword is set when using XPALETTE to force the example routine to wait until the **Done** button is pressed to continue. If the BLOCK keyword was not set, the example routine would produce all of the displays at once and then end.

```
PRO UsingXPALETTE

; Determine the path to the file.
```

```
ctscanFile = FILEPATH('ctscan.dat', $
   SUBDIRECTORY = ['examples', 'data'])

; Initialize image size parameter.
ctscanSize = [256, 256]

; Import in the image from the file.
ctscanImage = READ_BINARY(ctscanFile, $
   DATA_DIMS = ctscanSize)

; Initialize display.
DEVICE, DECOMPOSED = 0
LOADCT, 0
WINDOW, 0, TITLE = 'ctscan.dat', $
   XSIZE = ctscanSize[0], YSIZE = ctscanSize[1]

; Display image.
TV, ctscanImage

; Click on the "Predefined" button and select the
; "Rainbow + white" color table.
XPALETTE, /BLOCK
TV, ctscanImage

; Click on the 115th index, which is in column 3 and row
; 7, and then change its color to orange with the RGB
; (red, green, and blue) sliders.  Orange is made up of
; 255 red, 128 green, and 0 blue.
XPALETTE, /BLOCK
TV, ctscanImage

; Click on the 115th index, click on the "Set Mark"
; button, click on the 255th index, and click on the
; "Interpolate" button.  The colors within the 115 to
; 255 range are now changed to go between orange and
; white.  To see this change within the XPALETTE
; utility, click on the "Redraw" button.
XPALETTE, /BLOCK
TV, ctscanImage

; Obtain the red, green, and blue vectors of this
; current color table.
TVLCT, red, green, blue, /GET

; Add this modified color table to IDL's list of
; pre-defined color tables and display results.
MODIFYCT, 41, 'Orange to White Bones', $
   red, green, blue
XLOADCT, /BLOCK
```

```
TV, ctscanImage

END
```

Using the MODIFYCT Routine

The previously derived color table created in "Using the XPALETTE Utility" on page 81 can be added to IDL's list of pre-defined color tables with the TVLCT and MODIFYCT routines. For code that you can copy and paste into a text editor (for example the IDL Editor), see "Example Code: Using the XPALETTE Utility" on page 85.

By default, TVLCT allows you to load in red, green, and blue vectors (either derived by you or imported in from an image file) to load a different current color table. TVLCT also has a GET keyword. When the GET keyword is set, TVLCT returns the red, green, and blue vectors of the current color table back to you. Using this you can obtain the red, green, and blue vectors of the previously derived color table.

1. Obtain the red, green, and blue vectors of the current color table after performing the steps in "Using the XPALETTE Utility" on page 81:

   ```
   TVLCT, red, green, blue, /GET
   ```

   The MODIFYCT routine uses these vectors as arguments. Now you can use MODIFYCT to add this new color table to IDL's list of pre-defined color tables.

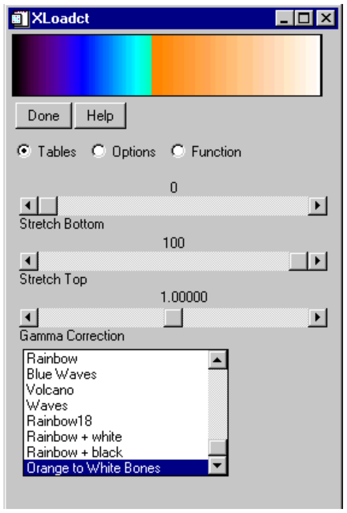
2. Add this modified color table to IDL's list of pre-defined color tables and display results:

   ```
   MODIFYCT, 41, 'Orange to White Bones', $
      red, green, blue
   ```

3. Display the results with XLOADCT:

   ```
   XLOADCT
   ```

The modified color table has been added to IDL's list of pre-defined color tables as shown in the following figure.



● Tables ○ Options ○ Function

Stretch Bottom: 0
Stretch Top: 100
Gamma Correction: 1.00000

Rainbow
Blue Waves
Volcano
Waves
Rainbow18
Rainbow + white
Rainbow + black
Orange to White Bones

Figure 15: XLOADCT Showing Results of MODIFYCT

The MODIFYCT routine can also be used to save changes to one of the existing pre-defined color tables. See MODIFYCT *in the IDL Reference Guide* for more information.

Converting to Other Color Systems

IDL defaults to the RGB color system, but if you are more accustomed to other color systems, IDL is not restricted to working with only the RGB color system. You can also use either the HSV (hue, saturation, and value) system or the HLS (hue, lightness, and saturation) system. The HSV or HLS system can be specified by setting the appropriate keyword (for example /HSV or /HLS) when using IDL color routines.

IDL also contains routines to create color tables based on these color systems. The HSV routine creates a color table based on the Hue, Saturation, and Value (HSV) color system. The HLS routine creates a color table based on the Hue, Lightness, Saturation (HLS) color system. You can also convert values of a color from any of these systems to another with the COLOR_CONVERT routine. See `COLOR_CONVERT` in the *IDL Reference Guide* for more information.

Converting Between Image Types

Sometimes an image must be converted from indexed to RGB or RGB to indexed. An image may be imported into IDL as an indexed image (from a PNG file for example) and you may need to export it as an RGB image (to a JPEG file for example). Of course, the opposite may also occur: importing an RGB image and exporting it as an indexed image. See "How Colors are Associated with Indexed and RGB Images" on page 64 for more information on indexed and RGB images.

Converting Indexed Images to RGB Images

The `convec.dat` file is a binary file containing an indexed image. The file contains an indexed image of the convection of the earth's mantle. This file does not contain a related color table. The following example applies a color table to this image and then converts the image and table to an RGB image (which contains its own color information).

For code that you can copy and paste into an IDL Editor window, see "Example Code: Converting Indexed Images to RGB Images" on page 91 or complete the following steps for a detailed description of the process.

1. Determine the path to the `convec.dat` binary file:

   ```
   convecFile = FILEPATH('convec.dat', $
      SUBDIRECTORY = ['examples', 'data'])
   ```

2. Initialize the image size parameter:

   ```
   convecSize = [248, 248]
   ```

3. Import in the image from the file:

   ```
   convecImage = READ_BINARY(convecFile, $
      DATA_DIMS = convecSize)
   ```

4. If you are running IDL on a TrueColor display, set the DECOMPOSED keyword to the DEVICE command to zero before your first color table related routine is used within an IDL session or program. See "How Colors are Associated with Indexed and RGB Images" on page 64 for more information.

   ```
   DEVICE, DECOMPOSED = 0
   ```

   The EOS B color table is applied to highlight the features of this image.

5. Load the EOS B color table (index number 27) to highlight the image's features and initialize the display:

```
LOADCT, 27
WINDOW, 0, TITLE = 'convec.dat', $
   XSIZE = convecSize[0], YSIZE = convecSize[1]
```

6. Now display the image with this color table:

```
TV, convecImage
```

Your display should be similar to the following figure.

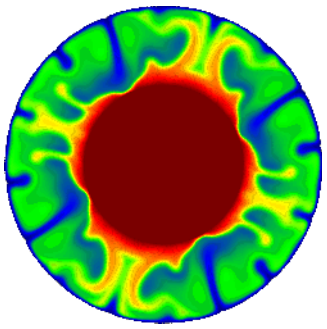


Figure 16: Example of an Indexed Image With Associated Color Table

A color table is formed from three vectors (the red vector, the green vector, and the blue vector). The same element of each vector form an RGB triple to create a color. For example, the *ith* element of the red vector may be 255, the *ith* element of the green vector may be 255, and the *ith* element of the blue vector maybe 0. The RGB triplet of the *ith* element would then be (255, 255, 0), which is the color of yellow. Since a color table contains 256 indices, its three vectors have 256 elements each. You can access these vectors with the TVLCT routine using the GET keyword.

Note

On some PseudoColor displays, fewer than 256 entries will be available.

7. Access the values of the applied color table by setting the GET keyword to the TVLCT routine.

   ```
   TVLCT, red, green, blue, /GET
   ```

   This color table (color information) can be stored within the image by converting it to an RGB image. For this example, the RGB image will be pixel interleaved to be written to a JPEG file.

Tip

If the original indexed image contains values of a data type other than byte, you should byte-scale the image (with the BYTSCL routine) before using the following method.

Before converting the indexed image into an RGB image, the resulting three-dimensional array must be initialized.

8. Initialize the data type and the dimensions of the resulting RGB image:

   ```
   imageRGB = BYTARR(3, convecSize[0], convecSize[1])
   ```

   Each channel of the resulting RGB image can be derived from the red, green, and blue vectors of the color table and the original indexed image.

9. Use the red, green, and blue vectors of the color table and the original indexed image to form a single three-channelled image:

   ```
   imageRGB[0, *, *] = red[convecImage]
   imageRGB[1, *, *] = green[convecImage]
   imageRGB[2, *, *] = blue[convecImage]
   ```

10. Write the resulting RGB image out to a JPEG file.

    ```
    WRITE_JPEG, 'convecImage.jpg', imageRGB, TRUE = 1, $
       QUALITY = 100.
    ```

    The TRUE keyword is set to 1 because the resulting RGB image is pixel interleaved. See WRITE_JPEG for more information.

Example Code: Converting Indexed Images to RGB Images

Copy and paste the following text into the IDL Editor window. After saving the file as `IndexedToRGB.pro`, compile and run the program to reproduce the previous example.

```
PRO IndexedToRGB
```

```
; Determine the path to the file.
convecFile = FILEPATH('convec.dat', $
   SUBDIRECTORY = ['examples', 'data'])

; Initialize the image size parameter.
convecSize = [248, 248]

; Import in the image from the file.
convecImage = READ_BINARY(convecFile, $
   DATA_DIMS = convecSize)

; Initialize display.
DEVICE, DECOMPOSED = 0
LOADCT, 27
WINDOW, 0, TITLE = 'convec.dat', $
   XSIZE = convecSize[0], YSIZE = convecSize[1]

; Display image.
TV, convecImage

; Obtain the red, green, and blue vectors that form the
; current color table.
TVLCT, red, green, blue, /GET

; Initialize the resulting RGB image.
imageRGB = BYTARR(3, convecSize[0], convecSize[1])

; Derive each color image from the vectors of the
; current color table.
imageRGB[0, *, *] = red[convecImage]
imageRGB[1, *, *] = green[convecImage]
imageRGB[2, *, *] = blue[convecImage]

; Write the resulting RGB image out to a JPEG file.
WRITE_JPEG, 'convec.jpg', imageRGB, TRUE = 1, $
   QUALITY = 100.

END
```

Converting RGB Images to Indexed Images

The `elev_t.jpg` file contains a pixel interleaved RGB image, which has its own color information. This example converts the image to an indexed image with an associated color table.

For code that you can copy and paste into an IDL Editor window, see "Example Code: Converting RGB Images to Indexed Images" on page 95 or complete the following steps for a detailed description of the process.

1. Determine the path to the `elev_t.jpg` file:

   ```
   elev_tFile = FILEPATH('elev_t.jpg', $
      SUBDIRECTORY = ['examples', 'data'])
   ```

2. Import in the image from the `elev_t.jpg` file into IDL:

   ```
   READ_JPEG, elev_tFile, elev_tImage
   ```

3. Determine the size of the imported image:

   ```
   elev_tSize = SIZE(elev_tImage, /DIMENSIONS)
   ```

4. If you are running IDL on a TrueColor display, set the DECOMPOSED keyword to the DEVICE command to one before your first RGB image is displayed within an IDL session or program. See "How Colors are Associated with Indexed and RGB Images" on page 64 for more information:

   ```
   DEVICE, DECOMPOSED = 1
   ```

5. Initialize the display:

   ```
   WINDOW, 0, TITLE = 'elev_t.jpg', $
      XSIZE = elev_tSize[1], YSIZE = elev_tSize[2]
   ```

6. Display the imported image:

   ```
   TV, elev_tImage, TRUE = 1
   ```

Your display should be similar to the following figur.e.

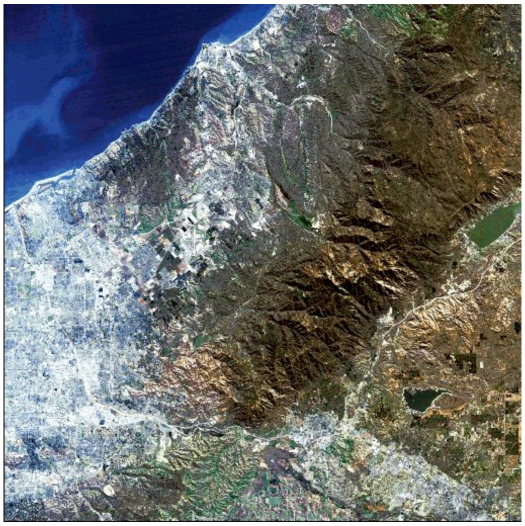


Figure 17: Example of an RGB Image

Note

If you are running IDL on a PseudoColor display, the RGB image will not be displayed correctly. A PseudoColor display only allows the display of indexed images. You can change the RGB image to an indexed image with the COLOR_QUAN routine. An example of this method is shown in this section.

The RGB image is converted to an indexed image with the COLOR_QUAN routine, but the DECOMPOSED keyword to the DEVICE command must be set to zero (for TrueColor displays) before using the COLOR_QUAN because it is a color table related routine. See COLOR_QUAN in the *IDL Reference Guide* for more information.

Note

COLOR_QUAN may result in some loss of color fidelity since it quantizes the image to a fixed number of colors (stored in the color table)

7. If you are running IDL on a TrueColor display, set the DECOMPOSED keyword to the DEVICE command to zero before your first color table related routine is used within an IDL session or program. See "How Colors are Associated with Indexed and RGB Images" on page 64 for more information.

   ```
   DEVICE, DECOMPOSED = 0
   ```

8. Convert the RGB image to an indexed image with an associated color table:

   ```
   imageIndexed = COLOR_QUAN(elev_tImage, 1, red, green, $
      blue)
   ```

9. Write the resulting indexed image and its associated color table out to a PNG file:

   ```
   WRITE_PNG, 'elev_t.png', imageIndexed, red, green, blue
   ```

Example Code: Converting RGB Images to Indexed Images

Copy and paste the following text into the IDL Editor window. After saving the file as `RGBToIndexed.pro`, compile and run the program to reproduce the previous example.

```
PRO RGBToIndexed

; Determine path to the "elev_t.jpg" file.
elev_tFile = FILEPATH('elev_t.jpg', $
   SUBDIRECTORY = ['examples', 'data'])

; Import image from file into IDL.
READ_JPEG, elev_tFile, elev_tImage

; Determine the size of the imported image.
elev_tSize = SIZE(elev_tImage, /DIMENSIONS)

; Initialize display.
DEVICE, DECOMPOSED = 1
WINDOW, 0, TITLE = 'elev_t.jpg', $
   XSIZE = elev_tSize[1], YSIZE = elev_tSize[2]

; Display image.
TV, elev_tImage, TRUE = 1

; Convert RGB image to indexed image with associated
; color table.
DEVICE, DECOMPOSED = 0
imageIndexed = COLOR_QUAN(elev_tImage, 1, red, green, $
   blue)
```

```
; Write resulting image and its color table to a PNG
; file.
WRITE_PNG, 'elev_t.png', imageIndexed, red, green, blue

END
```

Highlighting Features with a Color Table

For indexed images, entire color tables can be derived to highlight specific features. Color tables are usually designed to vary within certain ranges to show dramatic changes within an image. Some color tables are designed to highlight features with drastic color change in adjacent ranges (for example setting 0 through 20 to black and setting 21 through 40 to white). Regardless, some features within an image may not be significantly highlighted by IDL's 41 pre-defined color tables.

Note

Color tables are associated with indexed images. RGB images already contain their own color information. If you want to derive a color table for an RGB image, you should convert it to an indexed image with the COLOR_QUAN routine. You should also set COLOR_QUAN's CUBE keyword to 6 to insure the resulting indexed image is an intensity representation of the original RGB image. See `COLOR_QUAN` in the *IDL Reference Guide* for more information

Highlighting Features with Color in Direct Graphics

The data in the `mineral.png` file in the `examples/data` directory comes with its own color table. The following example will apply this related color table, then a pre-defined color table, and finally derive a new color table to highlight specific features.

For code that you can copy and paste into an IDL Editor window, see "Example Code: Highlighting Features with Color in Direct Graphics" on page 101 or complete the following steps for a detailed description of the process.

1. Determine the path to the `mineral.png` file:

```
mineralFile = FILEPATH(‘mineral.png', $
   SUBDIRECTORY = [‘examples', ‘data'])
```

2. Import the image from the `minernal.png` file into IDL:

```
mineralImage = READ_PNG(mineralFile, red, green, blue)
```

The image's associated color table is contained within the resulting red, green, and blue vectors.

3. Determine the size of the imported image:

```
mineralSize = SIZE(mineralImage, /DIMENSIONS)
```

4. If you are running IDL on a TrueColor display, set the DECOMPOSED keyword to the DEVICE command to zero before your first color table related routine is used within an IDL session or program. See "How Colors are Associated with Indexed and RGB Images" on page 64 for more information.

   ```
   DEVICE, DECOMPOSED = 0
   ```

5. Load the image's associated color table with the TVLCT routine:

   ```
   TVLCT, red, green, blue
   ```

6. Initialize the display:

   ```
   WINDOW, 0, XSIZE = mineralSize[0], YSIZE = mineralSize[1], $
      TITLE = 'mineral.png'
   ```

7. Display the imported image:

   ```
   TV, mineralImage
   ```

   This scanning electron microscope image shows mineral deposits in a sample of polished granite and gneiss. The associated color table is a reverse grayscale.

   The following figure shows that the associated color table highlights the gneiss very well, but the other features are not very clear. The other features can be defined with IDL's pre-defined color table, RAINBOW.

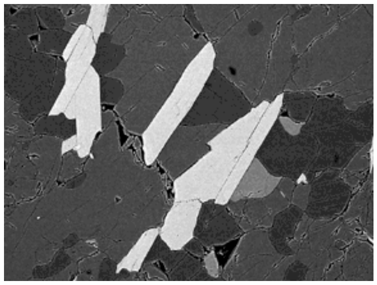


Figure 18: Mineral Image and Default Color Table (Direct Graphics)

8. Load the RAINBOW color table and redisplay the image in another window:

```
LOADCT, 13
WINDOW, 1, XSIZE = mineralSize[0], YSIZE = mineralSize[1], $
   TITLE = ‘RAINBOW Color'
TV, mineralImage
```

The following figure shows that the yellow, cyan, and red sections are now apparent, but the cracks are no longer visible. Details within the yellow areas and the green background are also difficult to distinguish. These features can be highlighted by designing your own color table.

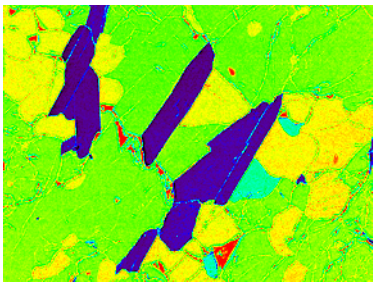


Figure 19: Mineral Image and RAINBOW Color Table (Direct Graphics)

The features within the image are at specific ranges in between 0 and 255. Instead of a progressive color table, specific colors can be defined to be constant over these ranges. Any contrasting colors can be used, but it is easiest to derive the additive and subtractive primary colors used in the previous section.

9. Define the colors for a new color table:

```
colorLevel = [[0, 0, 0], $ ; black
   [255, 0, 0], $ ; red
   [255, 255, 0], $ ; yellow
   [0, 255, 0], $ ; green
   [0, 255, 255], $ ; cyan
   [0, 0, 255], $ ; blue
   [255, 0, 255], $ ; magenta
   [255, 255, 255]] ; white
```

10. Create a new color table that contains eight levels, including the highest end boundary by first deriving levels for each color in the new color table:

    ```
    numberOfLevels = CEIL(!D.TABLE_SIZE/8.)
    level = INDGEN(!D.TABLE_SIZE)/numberOfLevels
    ```

11. Place each color level into its appropriate range.

    ```
    newRed = colorLevel[0, level]
    newGreen = colorLevel[1, level]
    newBlue = colorLevel[2, level]
    ```

12. Include the last color in the last level:

    ```
    newRed[!D.TABLE_SIZE - 1] = 255
    newGreen[!D.TABLE_SIZE - 1] = 255
    newBlue[!D.TABLE_SIZE - 1] = 255
    ```

13. Make the new color table current:

    ```
    TVLCT, newRed, newGreen, newBlue
    ```

14. Display the image with this new color table in another window:

    ```
    WINDOW, 2, XSIZE = mineralSize[0], $
       YSIZE = mineralSize[1], TITLE = 'Cube Corner Colors'
    TV, mineralImage
    ```

    The following figure shows that each feature is now highlighted including the cracks. The color table also highlights at least three different types of cracks.

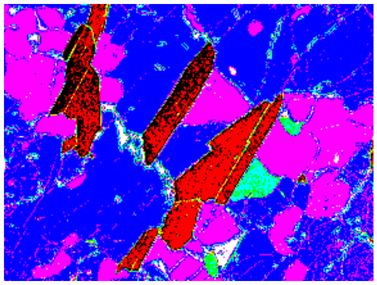


Figure 20: Mineral Image and Derived Color Table (Direct Graphics)

Example Code: Highlighting Features with Color in Direct Graphics

Copy and paste the following text into the IDL Editor window. After saving the file as `HighlightFeatures_Direct.pro`, compile and run the program to reproduce the previous example.

```
PRO HighlightFeatures_Direct

; Determine path to "mineral.png" file.
mineralFile = FILEPATH('mineral.png', $
   SUBDIRECTORY = ['examples', 'data'])

; Import image from file into IDL.
mineralImage = READ_PNG(mineralFile, $
   red, green, blue)

; Determine size of imported image.
mineralSize = SIZE(mineralImage, /DIMENSIONS)

; Apply imported color vectors to current color table.
DEVICE, DECOMPOSED = 0
TVLCT, red, green, blue

; Initialize display.
WINDOW, 0, XSIZE = mineralSize[0], YSIZE = mineralSize[1], $
   TITLE = 'mineral.png'

; Display image.
TV, mineralImage

; Load "RAINBOW" color table and display image in
; another window.
LOADCT, 13
WINDOW, 1, XSIZE = mineralSize[0], YSIZE = mineralSize[1], $
   TITLE = 'RAINBOW Color'
TV, mineralImage

; Define colors for a new color table.
colorLevel = [[0, 0, 0], $ ; black
   [255, 0, 0], $ ; red
   [255, 255, 0], $ ; yellow
   [0, 255, 0], $ ; green
   [0, 255, 255], $ ; cyan
   [0, 0, 255], $ ; blue
   [255, 0, 255], $ ; magenta
   [255, 255, 255]] ; white
```

```
; Derive levels for each color in the new color table.
; NOTE: some displays may have less than 256 colors.
numberOfLevels = CEIL(!D.TABLE_SIZE/8.)
level = INDGEN(!D.TABLE_SIZE)/numberOfLevels

; Place each color level into its appropriate range.
newRed = colorLevel[0, level]
newGreen = colorLevel[1, level]
newBlue = colorLevel[2, level]

; Include the last color in the last level.
newRed[!D.TABLE_SIZE - 1] = 255
newGreen[!D.TABLE_SIZE - 1] = 255
newBlue[!D.TABLE_SIZE - 1] = 255

; Make the new color table current.
TVLCT, newRed, newGreen, newBlue

; Display image in another window.
WINDOW, 2, XSIZE = mineralSize[0], $
   YSIZE = mineralSize[1], TITLE = 'Cube Corner Colors'
TV, mineralImage

END
```

Highlighting Features with Color in Object Graphics

The previous example could have also been done with Object Graphics. The color table is derived in the same matter.This example shows how to create a color table to highlight image features using Object Graphics.

For code that you can copy and paste into an IDL Editor window, see "Example Code: Highlighting Features with Color in Object Graphics" on page 106 or complete the following steps for a detailed description of the process.

1. Determine the path to the `mineral.png` file:

```
mineralFile = FILEPATH(‘mineral.png', $
   SUBDIRECTORY = [‘examples', ‘data'])
```

2. Import the image and its associated color table into IDL:

```
mineralImage = READ_PNG(mineralFile, red, green, blue)
```

3. Determine the size of the imported image:

```
mineralSize = SIZE(mineralImage, /DIMENSIONS)
```

4. Initialize objects necessary for a graphics display:

```
oWindow = OBJ_NEW('IDLgrWindow', RETAIN = 2, $
   DIMENSIONS = [mineralSize[0], mineralSize[1]], $
   TITLE = 'mineral.png')
oView = OBJ_NEW('IDLgrView', VIEWPLANE_RECT = [0., 0., $
   mineralSize[0], mineralSize[1]])
oModel = OBJ_NEW('IDLgrModel')
```

5. Initialize a palette object containing the image's associated color table and apply the palette to the image objects:

```
oPalette = OBJ_NEW('IDLgrPalette', red, green, blue)
oImage = OBJ_NEW('IDLgrImage', mineralImage, $
   PALETTE = oPalette)
```

The objects are then added to the view, which is displayed in the window.

6. Add the image to the model, then add the model to the view:

```
oModel -> Add, oImage
oView -> Add, oModel
```

   Draw the view in the window:

```
oWindow -> Draw, oView
```

   This scanning electron microscope image shows mineral deposits in a sample of polished granite and gneiss. The associated color table is a reverse grayscale.

   The following figure shows that the associated color table highlights the gneiss very well, but the other features are not very clear. The other features can be defined with IDL's pre-defined color table, RAINBOW.

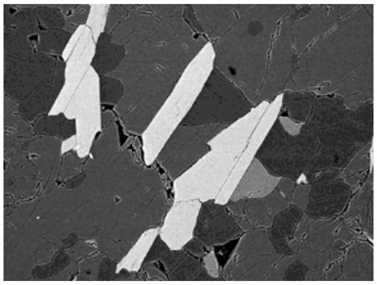


Figure 21: Mineral Image and Default Color Table (Object Graphics)

The palette can easily be modified to show the RAINBOW pre-defined color table in another instance of the window object.

7. Update palette with RAINBOW color table and then display the image with this color table in another instance window of the window object:

```
oPalette -> LoadCT, 13
oWindow = OBJ_NEW('IDLgrWindow', RETAIN = 2, $
   DIMENSIONS = [mineralSize[0], mineralSize[1]], $
   TITLE = 'RAINBOW Color')
oWindow -> Draw, oView
```

The following figure shows that the yellow, cyan, and red sections are now apparent, but the cracks are no longer visible. Details within the yellow areas and the green background are also difficult to distinguish. These features can be highlighted by designing your own color table.

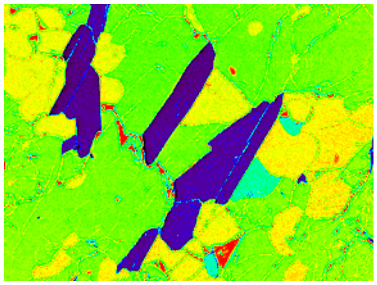


Figure 22: Mineral Image and RAINBOW Color Table (Object Graphics)

The features within the image are at specific ranges in between 0 and 255. Instead of a progressive color table, specific colors can be defined to be constant over these ranges. Any contrasting colors can be used, but the easiest to derive are the additive and subtractive primary colors used in the previous section.

8. Define colors for a new color table:

```
colorLevel = [[0, 0, 0], $ ; black
   [255, 0, 0], $ ; red
   [255, 255, 0], $ ; yellow
   [0, 255, 0], $ ; green
   [0, 255, 255], $ ; cyan
   [0, 0, 255], $ ; blue
   [255, 0, 255], $ ; magenta
   [255, 255, 255]] ; white
```

9. Create a new color table that contains eight levels, including the highest end boundary by first deriving levels for each color in the new color table:

```
numberOfLevels = CEIL(!D.TABLE_SIZE/8.)
level = INDGEN(!D.TABLE_SIZE)/numberOfLevels
```

10. Place each color level into its appropriate range.

```
newRed = colorLevel[0, level]
newGreen = colorLevel[1, level]
newBlue = colorLevel[2, level]
```

11. Include the last color in the last level:

```
newRed[!D.TABLE_SIZE - 1] = 255
newGreen[!D.TABLE_SIZE - 1] = 255
newBlue[!D.TABLE_SIZE - 1] = 255
```

Apply the new color table to the palette object:

12. Display the image with this color table in another window:

```
oPalette -> SetProperty, RED_VALUES = newRed, $
   GREEN_VALUES = newGreen, BLUE_VALUES = newBlue
oWindow = OBJ_NEW('IDLgrWindow', RETAIN = 2, $
   DIMENSIONS = [mineralSize[0], mineralSize[1]], $
   TITLE = 'Cube Corner Colors')
oWindow -> Draw, oView
```

The following figure shows that each image feature is readily distinguishable.

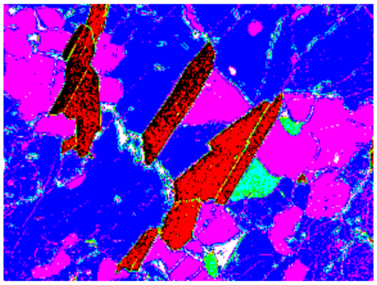


Figure 23: Mineral Image and Derived Color Table (Object Graphics)

13. Clean-up object references. When working with objects always remember to clean-up any object references with the OBJ_DESTROY routine. Since the view contains all the other objects, except for the window (which is destroyed by the user), you only need to use OBJ_DESTROY on the view and the palette object:

    ```
    OBJ_DESTROY, [oView, oPalette]
    ```

Example Code: Highlighting Features with Color in Object Graphics

Copy and paste the following text into the IDL Editor window. After saving the file as `HighlightFeatures_Object.pro`, compile and run the program to reproduce the previous example.

```
PRO HighlightFeatures_Object

; Determine path to "mineral.png" file.
mineralFile = FILEPATH('mineral.png', $
   SUBDIRECTORY = ['examples', 'data'])

; Import image from file into IDL.
mineralImage = READ_PNG(mineralFile, $
   red, green, blue)

; Determine size of imported image.
mineralSize = SIZE(mineralImage, /DIMENSIONS)

; Initialize objects.
; Initialize display.
oWindow = OBJ_NEW('IDLgrWindow', RETAIN = 2, $
   DIMENSIONS = [mineralSize[0], mineralSize[1]], $
   TITLE = 'mineral.png')
oView = OBJ_NEW('IDLgrView', VIEWPLANE_RECT = [0., 0., $
   mineralSize[0], mineralSize[1]])
oModel = OBJ_NEW('IDLgrModel')
; Initialize palette and image.
oPalette = OBJ_NEW('IDLgrPalette', red, green, blue)
oImage = OBJ_NEW('IDLgrImage', mineralImage, $
   PALETTE = oPalette)

; Add image to model, then model to view, and draw final
; view in window.
oModel -> Add, oImage
oView -> Add, oModel
oWindow -> Draw, oView
```

```
; Update palette with RAINBOW color table and then
; display image in another instance of the window object.
oPalette -> LoadCT, 13
oWindow = OBJ_NEW('IDLgrWindow', RETAIN = 2, $
   DIMENSIONS = [mineralSize[0], mineralSize[1]], $
   TITLE = 'RAINBOW Color')
oWindow -> Draw, oView

; Define colors for a new color table.
colorLevel = [[0, 0, 0], $ ; black
   [255, 0, 0], $ ; red
   [255, 255, 0], $ ; yellow
   [0, 255, 0], $ ; green
   [0, 255, 255], $ ; cyan
   [0, 0, 255], $ ; blue
   [255, 0, 255], $ ; magenta
   [255, 255, 255]] ; white

; Derive levels for each color in the new color table.
; NOTE: some displays may have less than 256 colors.
numberOfLevels = CEIL(!D.TABLE_SIZE/8.)
level = INDGEN(!D.TABLE_SIZE)/numberOfLevels

; Place each color level into its appropriate range.
newRed = colorLevel[0, level]
newGreen = colorLevel[1, level]
newBlue = colorLevel[2, level]

; Include the last color in the last level.
newRed[!D.TABLE_SIZE - 1] = 255
newGreen[!D.TABLE_SIZE - 1] = 255
newBlue[!D.TABLE_SIZE - 1] = 255

; Update palette with new color table and then
; display image in another instance of the window object.
oPalette -> SetProperty, RED_VALUES = newRed, $
   GREEN_VALUES = newGreen, BLUE_VALUES = newBlue
oWindow = OBJ_NEW('IDLgrWindow', RETAIN = 2, $
   DIMENSIONS = [mineralSize[0], mineralSize[1]], $
   TITLE = 'Cube Corner Colors')
oWindow -> Draw, oView

; Clean-up object references.
OBJ_DESTROY, [oView, oPalette]

END
```

Showing Variations in Uniform Areas

Histogram-equalization is used to change either an image or its associated color table to display minor variations within nearly uniform areas of the image. The histogram of the image is used to determine where the image or color table should be equalized to highlight these minor variations. Since this chapter pertains to color and color tables, this section only discusses histogram-equalization of color tables. See section *xxx* in chapter *xxx* for more information on how histogram-equalization effects images.

The histogram of an image shows the number of pixels for each color value within the range of the image. If the minimum value of the image is 0 and the maximum value of the image is 255, the histogram of the image shows the number of pixels for each value ranging between and including 0 and 255. Peaks in the histogram represent more common values within the image which usually consist of nearly uniform regions. Valleys in the histogram represent less common values. Empty regions within the histogram indicate that no regions within the image contain those values.

The following figure shows an example of a histogram and its related image. The most common value in this image is 180, which appears to be the background of the image. Although the background appears nearly uniform, it contains many variations (cracks).

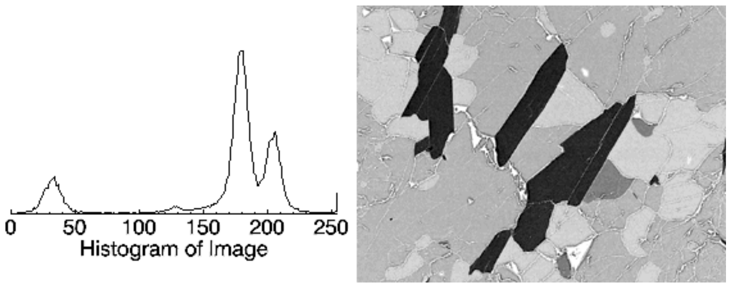


Figure 24: Example of a Histogram (left) and Its Related Image (right)

The empty regions within the image still correspond to color values or indices within the color table even though they do not appear in the image display. During histogram-equalization, the values occurring in the empty regions of the histogram are redistributed equally among the peaks and valleys. This process creates intensity gradients within these regions (replacing nearly uniform values), thus highlighting minor variations.

The following section provides a histogram-equalization example in Direct Graphics, which uses routines that directly work with the current color table. Since the current color table concept does not apply to Object Graphics, you must use histogram-equalization routines that directly effect the image, see section *xxx* in chapter *xxx* for more information on histogram-equalization with Object Graphics.

Showing Variations with Direct Graphics

The following example will apply histogram-equalization to a color table associated with an image of mineral deposits to reveal previously indistinguishable features.

For code that you can copy and paste into an IDL Editor window, see "Example Code: Showing Variations with Direct Graphics" on page 113 or complete the following steps for a detailed description of the process.

1. Determine the path to the `mineral.png` file:

   ```
   file = FILEPATH(‘mineral.png', $
      SUBDIRECTORY = [‘examples', ‘data'])
   ```

2. Import the image from the `mineral.png` file into IDL:

   ```
   image = READ_PNG(file)
   ```

3. Determine the size of the imported image:

   ```
   imageSize = SIZE(image, /DIMENSIONS)
   ```

4. If you are running IDL on a TrueColor display, set the DECOMPOSED keyword to the DEVICE command to zero before your first color table related routine is used within an IDL session or program:

   ```
   DEVICE, DECOMPOSED = 0
   ```

5. Initialize the image display:

   ```
   LOADCT, 0
   WINDOW, 0, XSIZE = 2*imageSize[0], YSIZE = imageSize[1], $
      TITLE = 'Histogram/Image'
   ```

6. Compute and display the histogram of the image. This step is not required to perform histogram-equalization on a color table within IDL. It is done here to show how the histogram-equalization affects the color table:

```
brightnessHistogram = BYTSCL(HISTOGRAM(image))
PLOT, brightnessHistogram, XSTYLE = 9, YSTYLE = 5, $
   POSITION = [0.05, 0.2, 0.45, 0.9], $
   XTITLE = 'Histogram of Image'
```

7. Display the image within the same window.

```
TV, image, 1
```

The following figure shows the resulting histogram and its related image.

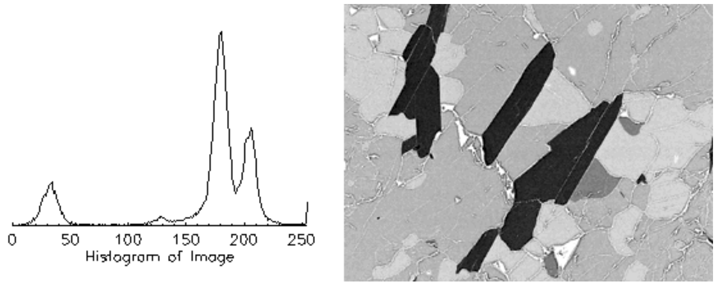


Figure 25: Histogram (left) of the Mineral Image (right) in Direct Graphics

8. Use the H_EQ_CT procedure to perform histogram-equalization on the current color table automatically updated it:

```
H_EQ_CT, image
```

9. Display the original image in another window with the updated color table:

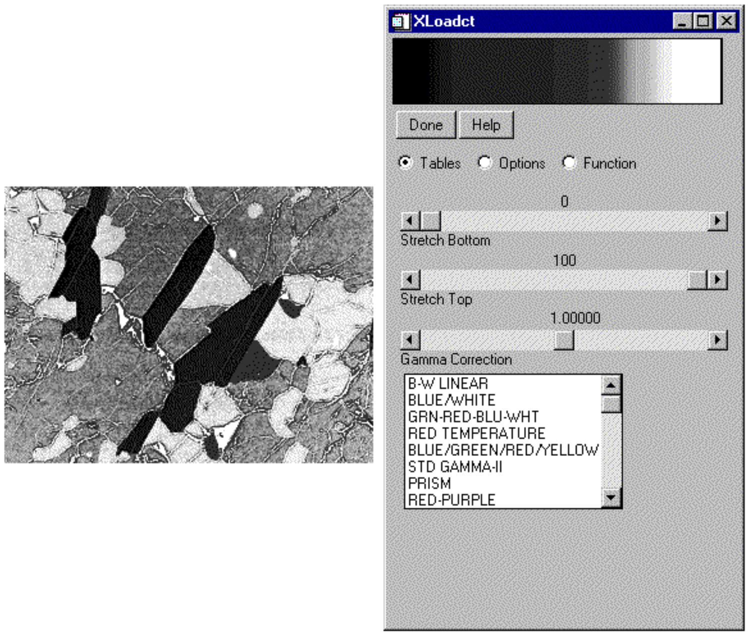
```
WINDOW, 1, XSIZE = imageSize[0], YSIZE = imageSize[1], $
   TITLE = 'Histogram-Equalized Color Table'
TV, image
```

Display the updated color table with the XLOADCT utility:

```
XLOADCT
```

Click on the **Done** button close the XLOADCT utility.

The following figure contains the results of the equalization on the image and its color table. After introducing intensity gradients within previously uniform regions of the image, the cracks are now more visible. However, some of the original features are not as clear. These regions can be clarified by interactively applying the amount of equalization to the color table.



(● Tables) ○ Options ○ Function

0
Stretch Bottom

100
Stretch Top

1.00000
Gamma Correction

B-W LINEAR
BLUE/WHITE
GRN-RED-BLU-WHT
RED TEMPERATURE
BLUE/GREEN/RED/YELLOW
STD GAMMA-II
PRISM
RED-PURPLE

Figure 26: Resulting Image (left) and Color Table (right) of the Histogram-Equalization in Direct Graphics

The histogram-equalizing process can also be interactively applied to a color table with the H_EQ_INT procedure. The H_EQ_INT procedure provides an interactive display, allowing you to use the cursor to control the amount of equalization. The equalization applied to the color table is scaled by a fraction, which is controlled by the movement of the cursor in the x-direction. If the cursor is all the way to the left side of the interactive display, the fraction equalized is close to zero, and the equalization has little effect on the color table. If the cursor is all the way to the right side of the interactive display, the fraction equalized is close to one, and the equalization is fully applied to the color table (which is similar to the results from the H_EQ_CT procedure). You can click on the right mouse button to set the amount of equalization and exit out of the interactive display.

10. Use the H_EQ_INT procedure to interactively perform histogram-equalization on the current color table:

```
H_EQ_INT, image
```

Place the cursor at about 130 in the x-direction, which is about 0.501 equalized (about 50% of the equalization applied by the H_EQ_CT procedure). You do not have to be exact for this example. The y-direction location is arbitrary.

Click on the right mouse button.

The interactive display is similar to the following figure.

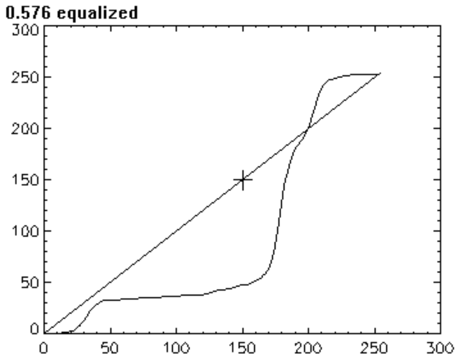


Figure 27: Interactive Display for Histogram-Equalization

11. Display the image using the updated color table in another window:

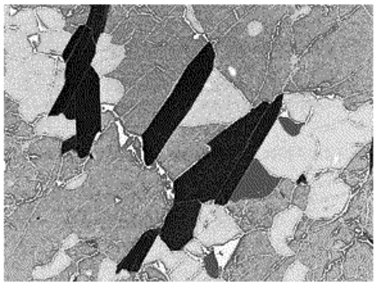
```
WINDOW, 2, XSIZE = imageSize[0], YSIZE = imageSize[1], $
   TITLE = 'Interactively Equalized Color Table'
TV, image
```

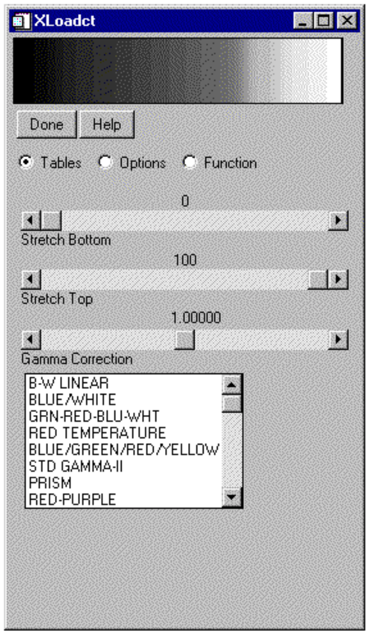
Display the updated color table with the XLOADCT utility:

```
XLOADCT
```

Click on the **Done** button close the XLOADCT utility.

The following figure contains the results of the equalization on the image and its color table. The original details have returned and the cracks are still visible.





XLoadct

Done Help

[✓ Tables] Options Function

0
Stretch Bottom

100
Stretch Top

1.00000
Gamma Correction

B-W LINEAR
BLUE/WHITE
GRN-RED-BLU-WHT
RED TEMPERATURE
BLUE/GREEN/RED/YELLOW
STD GAMMA-II
PRISM
RED-PURPLE

Figure 28: Resulting Image (left) and Color Table (right) of the Interactive Histogram-Equalization in Direct Graphics

Example Code: Showing Variations with Direct Graphics

Copy and paste the following text into the IDL Editor window. After saving the file as `HistogramEqualizing_Direct.pro`, compile and run the program to reproduce the previous example. The BLOCK keyword is set when using XLOADCT to force the example routine to wait until the **Done** button is pressed to continue. If the BLOCK keyword was not set, the example routine would produce all of the displays at once and then end.

```
PRO HistogramEqualizing_Direct

; Determine path to file.
file = FILEPATH('mineral.png', $
   SUBDIRECTORY = ['examples', 'data'])
```

```
; Import image from file into IDL.
image = READ_PNG(file)

; Determine size of imported image.
imageSize = SIZE(image, /DIMENSIONS)

;Initialize IDL on a TrueColor display to use
; color-related routines.
DEVICE, DECOMPOSED = 0

; Initialize the image display.
LOADCT, 0
WINDOW, 0, XSIZE = 2*imageSize[0], YSIZE = imageSize[1], $
   TITLE = 'Histogram/Image'

; Compute and scale histogram of image.
brightnessHistogram = BYTSCL(HISTOGRAM(image))

; Display histogram plot.
PLOT, brightnessHistogram, XSTYLE = 9, YSTYLE = 5, $
   POSITION = [0.05, 0.2, 0.45, 0.9], $
   XTITLE = 'Histogram of Image'

; Display image.
TV, image, 1

; Histogram-equalize the color table.
H_EQ_CT, image

; Display image and updated color table in another
; window.
WINDOW, 1, XSIZE = imageSize[0], YSIZE = imageSize[1], $
   TITLE = 'Histogram-Equalized Color Table'
TV, image

; Display the updated color table with the XLOADCT
; utility.
XLOADCT, /BLOCK

; Interactively histogram-equalize the color table. The
; H_EQ_INT routine provides an interactive display to
; allow you to select the amount of equalization. Place
; the cursor at about 130 in the x-direction, which is
; about 0.501 equalized. The y-direction is arbitrary.
; Click on the right mouse button.
; NOTE: you do not have to be exact for this example.
H_EQ_INT, image
```

```
; Display image and updated color table in another
; window.
WINDOW, 2, XSIZE = imageSize[0], YSIZE = imageSize[1], $
   TITLE = 'Interactively Equalized Color Table'
TV, image

; Display the updated color table with the XLOADCT
; utility.
XLOADCT, /BLOCK

END
```

Applying Color Annotations to Images

Many images are annotated to explain certain features or highlight specific details. Color annotations are more noticeable than plain black or white annotations. In Direct Graphics, how color annotations are applied depends on the type of image (indexed or RGB) displayed. With indexed images, annotation colors are derived from the image's associated color table. With RGB images, annotation colors are independent of the RGB image in Direct Graphics. Annotation colors and images are separated within Object Graphics regardless of the image type.

Applying Color Annotations to Indexed Images in Direct Graphics

Indexed images are usually associated with color tables. With Direct Graphics, these related color tables are used for all the colors shown within a display. Color tables are made up of up to 256 color triples (red, green, and blue values of each color within the table). If you want to apply a specific color to data or to an annotation, you must change the red, green, and blue values at a specific index within the color table.

Color annotations are usually applied to label each color level within the image or to allow color comparisons. This section shows how to label each color level on an indexed image in Direct Graphics. As an example, an image of average world temperature is imported from the `worldtmp.png` file. This file does not contain a color table associated with this image, so a pre-defined color table will be applied. This table provides the colors for the polygons and text used to make a colorbar for this image. Each polygon uses the color of each level in the table. The text represents the average temperature (in Celsius) of each level.

For code that you can copy and paste into an IDL Editor window, see "Example Code: Applying Color Annotations to Indexed Images in Direct Graphics" on page 120 or complete the following steps for a detailed description of the process.

1. Determine the path to the `worldtmp.png` file:

```
worldtmpFile = FILEPATH('worldtmp.png', $
   SUBDIRECTORY = ['examples', 'demo', 'demodata'])
```

2. Import the image from the `worldtmp.png` file into IDL:

```
worldtmpImage = READ_PNG(worldtmpFile)
```

3. Determine the size of the imported image:

```
worldtmpSize = SIZE(worldtmpImage, /DIMENSIONS)
```

4. If you are running IDL on a TrueColor display, set the DECOMPOSED keyword to the DEVICE command to zero before your first color table related routine is used within an IDL session or program. See "How Colors are Associated with Indexed and RGB Images" on page 64 for more information.

   ```
   DEVICE, DECOMPOSED = 0
   ```

   Since the imported image does not have an associated color table, the Rainbow18 color table (index number 38) is applied to the display.

5. Initialize display:

   ```
   LOADCT, 38
   WINDOW, 0, XSIZE = worldtmpSize[0], YSIZE = worldtmpSize[1], $
      TITLE = 'Average World Temperature (in Celsius)'
   ```

6. Now display the image with this color table:

   ```
   TV, worldtmpImage
   ```

   The following figure is displayed.

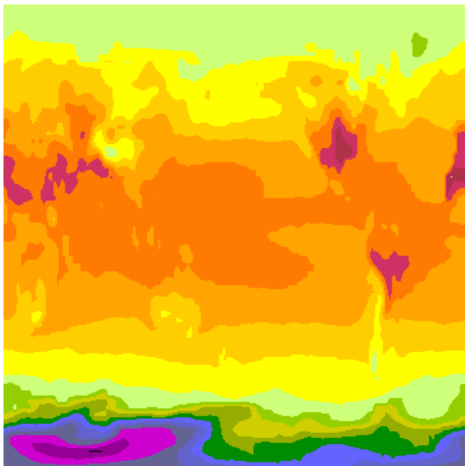


Figure 29: Temperature Image and Rainbow18 Color Table (Direct Graphics)

Before applying the color polygons and text of each level, you must first initialize their color values and their locations. The Rainbow18 color table has only 18 different color levels, but still has 256 elements. You can use the INDGEN routine to make an array of 18 elements ranging from 0 to 17 in value, where each element contains the index of that element. Then you can use the BYTSCL routine to scale these values to range from 0 to 255. The resulting array contains the initial color value (from 0 to 255) of the associated range (from 0 to 17, equalling 18 elements).

7. Initialize the color level parameter:

```
fillColor = BYTSCL(INDGEN(18))
```

8. Initialize the average temperature of each level, which directly depends on the initial color value of each range. Temperature is 0.20833 ($^5/_{24}$th) of the color value. You can use this relationship to calculate the temperature and then convert it to a string variable to be used as text:

```
temperature = STRTRIM(FIX((5.*fillColor)/42.), 2)
```

Note

When the *fillColor* variable in the previous statement is multiplied by the floating-point value of 5 (denoted by the decimal after the number), the elements of the array are converted from byte values to floating-point values. These elements are then converted to integer values with the FIX routine so the decimal part will not be displayed. The STRTRIM routine converts the integer values to string values to be displayed as text. The second argument to STRTRIM is set to 2 to note the leading and trailing blank characters should be trimmed away when the integer values are converted to string values.

With the polygon color and text now defined, you can determine their locations. You can use the POLYFILL routine to draw each polygon and the XYOUTS routine to display each element of text. The process is repetitive from level to level, so a FOR/DO loop is used to display the entire colorbar. Since each polygon and text is drawn individually within the loop, you only need to determine the location of a single polygon and an array of offsets for each step in the loop. The following two steps describe this process.

9. Initialize the polygon and the text location parameters. Each polygon is 25 pixels in width and 18 pixels in height. The offset will move the y-location 18 pixels every time a new polygon is displayed:

```
x = [5., 30., 30., 5., 5.]
y = [5., 5., 23., 23., 5.] + 5.
offset = 18.*FINDGEN(19) + 5.
```

10. Apply the polygons and text:

```
FOR i = 0, (N_ELEMENTS(fillColor) - 1) DO BEGIN & $
   POLYFILL, x, y + offset[i], COLOR = fillColor[i], $
   /DEVICE & $
   XYOUTS, x[0] + 5., y[0] + offset[i] + 5., $
   temperature[i], COLOR = 255*(fillColor[i] LT 255), $
   /DEVICE & $
ENDFOR
```

Note

The $ after BEGIN and the & allow you to use the FOR/DO loop at the IDL command line. These $ and & symbols are not required when the FOR/DO loop in placed in an IDL program as shown in "Example Code: Applying Color Annotations to Indexed Images in Direct Graphics" on page 120.

The following figure displays the colorbar annotation applied to the image.

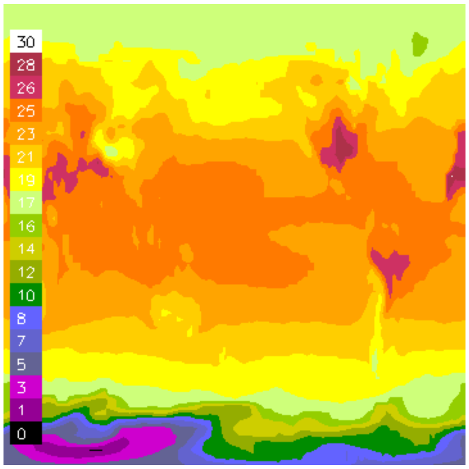


Figure 30: Temperature Image and Colorbar (Direct Graphics)

Example Code: Applying Color Annotations to Indexed Images in Direct Graphics

Copy and paste the following text into the IDL Editor window. After saving the file as `ApplyColorbar_Indexed_Direct.pro`, compile and run the program to reproduce the previous example.

```
PRO ApplyColorbar_Indexed_Direct

; Determine path to "worldtmp.png" file.
worldtmpFile = FILEPATH('worldtmp.png', $
   SUBDIRECTORY = ['examples', 'demo', 'demodata'])

; Import image from file into IDL.
worldtmpImage = READ_PNG(worldtmpFile)

; Determine size of imported image.
worldtmpSize = SIZE(worldtmpImage, /DIMENSIONS)

; Initialize display.
DEVICE, DECOMPOSED = 0
LOADCT, 38
WINDOW, 0, XSIZE = worldtmpSize[0], $
   YSIZE = worldtmpSize[1], $
   TITLE = 'Average World Temperature (in Celsius)'

; Display image.
TV, worldtmpImage

; Initialize color level parameter.
fillColor = BYTSCL(INDGEN(18))

; Initialize text variable.
temperature = STRTRIM(FIX((5.*fillColor)/42.), 2)

; Initialize polygon and text location parameters.
x = [5., 30., 30., 5., 5.]
y = [5., 5., 23., 23., 5.] + 5.
offset = 18.*FINDGEN(19) + 5.

; Apply polygons and text.
FOR i = 0, (N_ELEMENTS(fillColor) - 1) DO BEGIN
   POLYFILL, x, y + offset[i], COLOR = fillColor[i], $
   /DEVICE
   XYOUTS, x[0] + 5., y[0] + offset[i] + 5., $
   temperature[i], COLOR = 255*(fillColor[i] LT 255), $
   /DEVICE
```

```
ENDFOR

END
```

Applying Color Annotations to Indexed Images in Object Graphics

When using Object Graphics, the original color table does not need to be modified. The color table (palette) pertains only to the image object not the window, view, model, polygon, or text objects. Color annotations are usually applied to label each color level within the image or to allow color comparisons. This section shows how to label each color level on an indexed image in Object Graphics. As an example, an image of average world temperature is imported from the `worldtmp.png` file. This file does not contain a color table associated with this image, so a pre-defined color table will be applied. This table provides the colors for the polygons and text used to make a colorbar for this image. Each polygon uses the color of each level in the table. The text represents the average temperature (in Celsius) of each level.

For code that you can copy and paste into an IDL Editor window, see "Example Code: Applying Color Annotations to Indexed Images in Object Graphics" on page 125 or complete the following steps for a detailed description of the process.

1. Determine the path to the `worldtmp.png` file:

   ```
   worldtmpFile = FILEPATH('worldtmp.png', $
      SUBDIRECTORY = ['examples', 'demo', 'demodata'])
   ```

2. Import the image from the `worldtmp.png` file into IDL:

   ```
   worldtmpImage = READ_PNG(worldtmpFile)
   ```

3. Determine the size of the imported image:

   ```
   worldtmpSize = SIZE(worldtmpImage, /DIMENSIONS)
   ```

4. Initialize the display objects necessary for an Object Graphics display:

   ```
   oWindow = OBJ_NEW('IDLgrWindow', RETAIN = 2, $
      DIMENSIONS = [worldtmpSize[0], worldtmpSize[1]], $
      TITLE = 'Average World Temperature (in Celsius)')
   oView = OBJ_NEW('IDLgrView', $
      VIEWPLANE_RECT = [0, 0, worldtmpSize[0], $
      worldtmpSize[1]])
   oModel = OBJ_NEW('IDLgrModel')
   ```

5. Initialize palette object, load the Rainbow18 color table into the palette, and then apply the palette to an image object:

```
oPalette = OBJ_NEW('IDLgrPalette')
oPalette -> LoadCT, 38
oImage = OBJ_NEW('IDLgrImage', worldtmpImage, $
   PALETTE = oPalette)
```

6. Add the image to the model, then add the model to the view, and finally draw the view in the window:

```
oModel -> Add, oImage
oView -> Add, oModel
oWindow -> Draw, oView
```

The following figure is displayed.

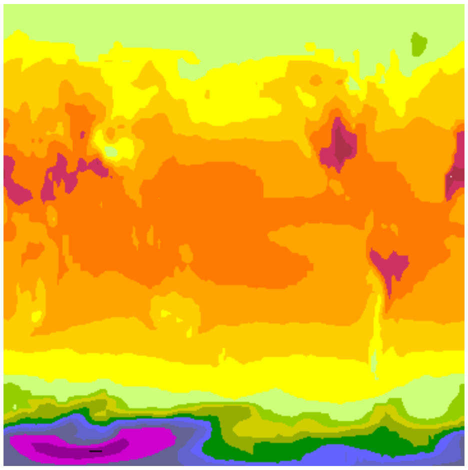


Figure 31: Temperature Image and Rainbow18 Color Table (Object Graphics)

Before applying the color polygons and text of each level, you must first initialize their color values and their locations. The Rainbow18 color table has only 18 different color levels, but still has 256 elements. You can use the INDGEN routine to make an array of 18 elements ranging from 0 to 17 in value, where each element contains the index of that element. Then you can use the BYTSCL routine to scale these values to range from 0 to 255. The resulting array contains the initial color value (from 0 to 255) of the associated range (from 0 to 17, equalling 18 elements).

7. Initialize the color level parameter:

```
fillColor = BYTSCL(INDGEN(18))
```

8. Initialize average temperature of each level, which directly depends on the initial color value of each range. Temperature is 0.20833 ($^5/_{24}$th) of the color value. You can use this relationship to calculate the temperature and then convert it to a string variable to be used as text:

```
temperature = STRTRIM(FIX((5.*fillColor)/42.), 2)
```

Note

When the *fillColor* variable in the previous statement is multiplied by the floating-point value of 5 (denoted by the decimal after the number), the elements of the array are converted from byte values to floating-point values. These elements are then converted to integer values with the FIX routine so the decimal part will not be displayed. The STRTRIM routine converts the integer values to string values to be displayed as text. The second argument to STRTRIM is set to 2 to note the leading and trailing black values should be trimmed away when the integer values are converted to string values.

With the polygon color and text now defined, you can determine their locations. You can use a polygon object to draw each polygon and text objects to display each element of text. The process is repetitive from level to level, so a FOR/DO loop is used to display the entire colorbar. Since each polygon and text is drawn individually within the loop, you only need to determine the location of a single polygon and an array of offsets for each step in the loop. The following two steps describe this process.

9. Initialize the polygon and the text location parameters. Each polygon is 25 pixels in width and 18 pixels in height. The offset will move the y-location 18 pixels every time a new polygon is displayed:

```
x = [5., 30., 30., 5., 5.]
y = [5., 5., 23., 23., 5.] + 5.
offset = 18.*FINDGEN(19) + 5.
```

10. Initialize the polygon and text objects:

```
oPolygon = OBJARR(18)
oText = OBJARR(18)
FOR i = 0, (N_ELEMENTS(oPolygon) - 1) DO BEGIN & $
   oPolygon[i] = OBJ_NEW('IDLgrPolygon', x, $
   y + offset[i], COLOR = fillColor[i], $
   PALETTE = oPalette) & $
   oText[i] = OBJ_NEW('IDLgrText', temperature[i], $
   LOCATIONS = [x[0] + 3., y[0] + offset[i] + 3.], $
   COLOR = 255*(fillColor[i] LT 255), $
   PALETTE = oPalette) & $
ENDFOR
```

Note

The $ after BEGIN and the & allow you to use the FOR/DO loop at the IDL command line. These $ and & symbols are not required when the FOR/DO loop in placed in an IDL program as shown in "Example Code: Applying Color Annotations to Indexed Images in Object Graphics" on page 125.

11. Add the polygons and text to the model, then add the model to the view, and finally redraw the view in the window:

```
oModel -> Add, oPolygon
oModel -> Add, oText
oWindow -> Draw, oView
```

The following figure displays the colorbar annotation applied to the image.

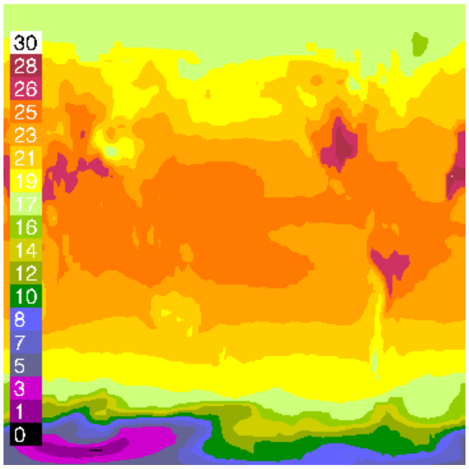


Figure 32: Temperature Image and Colorbar (Object Graphics

12. Clean-up object references. When working with objects always remember to clean-up any object references with the OBJ_DESTROY routine. Since the view contains all the other objects, except for the window (which is destroyed by the user), you only need to use OBJ_DESTROY on the view and the palette objects:

```
OBJ_DESTROY, [oView, oPalette]
```

Example Code: Applying Color Annotations to Indexed Images in Object Graphics

Copy and paste the following text into the IDL Editor window. After saving the file as `ApplyColorbar_Indexed_Object.pro`, compile and run the program to reproduce the previous example.

```
PRO ApplyColorbar_Indexed_Object

; Determine path to "worldtmp.png" file.
worldtmpFile = FILEPATH('worldtmp.png', $
   SUBDIRECTORY = ['examples', 'demo', 'demodata'])

; Import image from file into IDL.
worldtmpImage = READ_PNG(worldtmpFile)
```

```
; Determine size of imported image.
worldtmpSize = SIZE(worldtmpImage, /DIMENSIONS)

; Initialize objects.
; Initialize display.
oWindow = OBJ_NEW('IDLgrWindow', RETAIN = 2, $
   DIMENSIONS = [worldtmpSize[0], worldtmpSize[1]], $
   TITLE = 'Average World Temperature (in Celsius)')
oView = OBJ_NEW('IDLgrView', $
   VIEWPLANE_RECT = [0, 0, worldtmpSize[0], $
   worldtmpSize[1]])
oModel = OBJ_NEW('IDLgrModel')
; Initialize palette and image.
oPalette = OBJ_NEW('IDLgrPalette')
oPalette -> LoadCT, 38
oImage = OBJ_NEW('IDLgrImage', worldtmpImage, $
   PALETTE = oPalette)

; Add image to model, which is added to view, and then
; display view in window.
oModel -> Add, oImage
oView -> Add, oModel
oWindow -> Draw, oView

; Initialize color level parameter.
fillColor = BYTSCL(INDGEN(18))

; Initialize text variable.
temperature = STRTRIM(FIX((5*fillColor)/42), 2)

; Initialize polygon and text location parameters.
x = [5., 30., 30., 5., 5.]
y = [5., 5., 23., 23., 5.] + 5.
offset = 18.*FINDGEN(19) + 5.

; Initialize polygon and text objects.
oPolygon = OBJARR(18)
oText = OBJARR(18)
FOR i = 0, (N_ELEMENTS(oPolygon) - 1) DO BEGIN
   oPolygon[i] = OBJ_NEW('IDLgrPolygon', x, $
   y + offset[i], COLOR = fillColor[i], $
   PALETTE = oPalette)
   oText[i] = OBJ_NEW('IDLgrText', temperature[i], $
   LOCATIONS = [x[0] + 3., y[0] + offset[i] + 3.], $
   COLOR = 255*(fillColor[i] LT 255), $
   PALETTE = oPalette)
ENDFOR

; Add polygons and text to model and then re-display
; view in window.
oModel -> Add, oPolygon
oModel -> Add, oText
oWindow -> Draw, oView

; Clean-up object references.
OBJ_DESTROY, [oView, oPalette]

END
```

Applying Color Annotations to RGB Images in Direct Graphics

RGB images contain their own color information. Color tables do not apply to RGB images. With Direct Graphics the color of the annotations on an RGB image do not depend on a color table.

Tip

If you are running IDL on a PseudoColor display, use the COLOR_QUAN routine to convert the RGB image to an indexed image with an associated color table to display the image and see the previous section, "Applying Color Annotations to Indexed Images in Direct Graphics" on page 116.

If you want to apply a specific color to data or an annotation, you must provide the TrueColor index for that color. The TrueColor index ranges from 0 to 16,777,216. You can derive a TrueColor index from its red, green, and blue values:

```
red = 255
green = 128
blue = 0
trueColorIndex = red + (256L*green) + ((256L^2)*blue)
PRINT, trueColorIndex
  33023
```

where *red*, *green*, and *blue* are either scalars or vectors of values ranging from 0 to 255 and representing the amount of red, green, and blue in the resulting color. The L after the numbers defines that number as a longword integer data type. The above red, green, and blue combination creates the color of orange, which has a TrueColor index of 33,023.

In this example, a color spectrum of additive and subtractive primary colors will be drawn on an RGB image for comparison with the colors in an image. The `glowing_gas.jpg` file contains an RGB image of an expanding shell of glowing gas surrounding a hot, massive star in our Milky Way Galaxy. This image contains all the colors of this spectrum.

For code that you can copy and paste into an IDL Editor window, see "Example Code: Applying Color Annotations to Indexed Images in Direct Graphics" on page 120 or complete the following steps for a detailed description of the process.

1. Determine the path to the `glowing_gas.jpg` file:

   ```
   cosmicFile = FILEPATH('glowing_gas.jpg', $
      SUBDIRECTORY = ['examples', 'data'])
   ```

2. Import the image from the `glowing_gas.jpg` file into IDL:

   ```
   READ_JPEG, cosmicFile, cosmicImage
   ```

3. Determine the size of the imported image. The image contained within this file is pixel-interleaved (the color information is contained within the first dimension). You can use the SIZE routine to determine the other dimensions of this image:

   ```
   cosmicSize = SIZE(cosmicImage, /DIMENSIONS)
   ```

4. If you are running IDL on a TrueColor display, set the DECOMPOSED keyword to the DEVICE command to one before your first RGB image is displayed within an IDL session or program. See "How Colors are Associated with Indexed and RGB Images" on page 64 for more information:

   ```
   DEVICE, DECOMPOSED = 1
   ```

5. Use the dimensions determined in the previous step to initialize the display:

   ```
   WINDOW, 0, XSIZE = cosmicSize[1], YSIZE = cosmicSize[2], $
      TITLE = ‘glowing_gas.jpg'
   ```

6. Now display the image with the TRUE keyword set to 1 since the image is pixel interleaved:

   ```
   TV, cosmicImage, TRUE = 1
   ```

   The following figure shows that the image contains all of the colors of the additive and subtractive primary spectrum. In the following steps, a colorbar annotation will be added to allow you to compare the colors of that spectrum and the colors within the image.

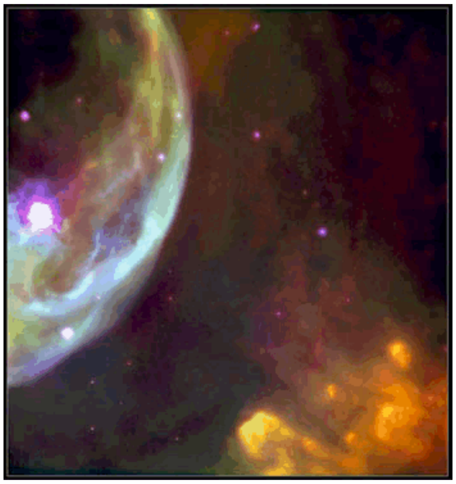


Figure 33: Cosmic RGB Image (Direct Graphics)

You can use the following to determine the color and location parameters for each polygon.

7. Initialize the color parameters:

```
red = BYTARR(8) & green = BYTARR(8) & blue = BYTARR(8)
red[0] = 0 & green[0] = 0 & blue[0] = 0 ; black
red[1] = 255 & green[1] = 0 & blue[1] = 0 ; red
red[2] = 255 & green[2] = 255 & blue[2] = 0 ; yellow
red[3] = 0 & green[3] = 255 & blue[3] = 0 ; green
red[4] = 0 & green[4] = 255 & blue[4] = 255 ; cyan
red[5] = 0 & green[5] = 0 & blue[5] = 255 ; blue
red[6] = 255 & green[6] = 0 & blue[6] = 255 ; magenta
red[7] = 255 & green[7] = 255 & blue[7] = 255 ; white
fillColor = red + (256L*green) + ((256L^2)*blue)
```

8. After defining the polygon colors, you can determine their locations. Initialize polygon location parameters:

```
x = [5., 25., 25., 5., 5.]
y = [5., 5., 25., 25., 5.] + 5.
offset = 20.*FINDGEN(9) + 5.
```

The *x* and *y* variables pertain to the x and y locations (in pixel units) of each box of color. The *offset* maintains the spacing (in pixel units) of each box. Since the image is made up of mostly a black background, the x border of the colorbar is also determined to draw a white border around the polygons.

9. Initialize location of colorbar border:

```
x_border = [x[0] + offset[0], x[1] + offset[7], $
   x[2] + offset[7], x[3] + offset[0], x[4] + offset[0]]
```

The y border is already defined by the y variable.

These parameters are used with POLYFILL and PLOTS to draw the boxes of the color spectrum and the colorbar border. Each polygon is 20 pixels wide and 20 pixels high. The offset will move the y-location 20 pixels every time a new polygon is displayed.

10. Apply the polygons and border. You can use the POLYFILL routine to draw each polygon. The process is repetitive from level to level, so a FOR/DO loop is used to display the entire colorbar. Since each polygon is drawn individually within the loop, you only need to determine the location of a single polygon and an array of offsets for each step in the loop:

```
FOR i = 0, (N_ELEMENTS(fillColor) - 1) DO POLYFILL, $
   x + offset[i], y, COLOR = fillColor[i], /DEVICE
PLOTS, x_border, y, COLOR = fillColor[7], /DEVICE
```

The POLYFILL and PLOTS routines result in the following display.

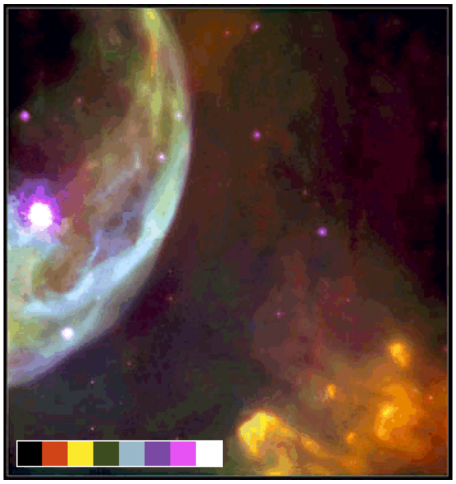


Figure 34: Specified Colors on an RGB Image (Direct Graphics)

Example Code: Applying Color Annotations to RGB Images in Direct Graphics

Copy and paste the following text into the IDL Editor window. After saving the file as `ApplyColorbar_RGB_Direct.pro`, compile and run the program to reproduce the previous example.

```
PRO ApplyingColorbar_RGB_Direct

; Determine path to "glowing_gas.jpg" file.
cosmicFile = FILEPATH('glowing_gas.jpg', $
   SUBDIRECTORY = ['examples', 'data'])

; Import image from file into IDL.
READ_JPEG, cosmicFile, cosmicImage

; Determine size of image.
cosmicSize = SIZE(cosmicImage, /DIMENSIONS)

; Initialize display.
DEVICE, DECOMPOSED = 1
WINDOW, 0, TITLE = 'glowing_gas.jpg', $
   XSIZE = cosmicSize[1], YSIZE = cosmicSize[2]
```

```
; Diplay image.
TV, cosmicImage, TRUE = 1

; Initialize color parameters.
red = BYTARR(8) & green = BYTARR(8) & blue = BYTARR(8)
red[0] = 0 & green[0] = 0 & blue[0] = 0 ; black
red[1] = 255 & green[1] = 0 & blue[1] = 0 ; red
red[2] = 255 & green[2] = 255 & blue[2] = 0 ; yellow
red[3] = 0 & green[3] = 255 & blue[3] = 0 ; green
red[4] = 0 & green[4] = 255 & blue[4] = 255 ; cyan
red[5] = 0 & green[5] = 0 & blue[5] = 255 ; blue
red[6] = 255 & green[6] = 0 & blue[6] = 255 ; magenta
red[7] = 255 & green[7] = 255 & blue[7] = 255 ; white
fillColor = red + (256L*green) + ((256L^2)*blue)

; Initialize polygon location parameters.
x = [5., 25., 25., 5., 5.]
y = [5., 5., 25., 25., 5.] + 5.
offset = 20.*FINDGEN(9) + 5.

; Initialize location of colorbar border.
x_border = [x[0] + offset[0], x[1] + offset[7], $
   x[2] + offset[7], x[3] + offset[0], x[4] + offset[0]]

; Apply polygons and border.
FOR i = 0, (N_ELEMENTS(fillColor) - 1) DO POLYFILL, $
   x + offset[i], y, COLOR = fillColor[i], /DEVICE
PLOTS, x_border, y, /DEVICE, COLOR = fillColor[7]

END
```

Applying Color Annotations to RGB Images in Object Graphics

When using Object Graphics, colors can be defined just by their red, green, and blue values. The TrueColor index conversion equation is not required for Object Graphics. In this example, a color spectrum of additive and subtractive primary colors will be drawn on an RGB image for comparison with the colors in that image. The `glowing_gas.jpg` file contains an RGB image of an expanding shell of glowing gas surrounding a hot, massive star in our Milky Way Galaxy. This image contains all the colors of this spectrum.

For code that you can copy and paste into an IDL Editor window, see “Example Code: Applying Color Annotations to RGB Images in Object Graphics” on page 136 or complete the following steps for a detailed description of the process.

1. Determine the path to the `glowing_gas.jpg` file:

   ```
   cosmicFile = FILEPATH('glowing_gas.jpg', $
      SUBDIRECTORY = ['examples', 'data'])
   ```

2. Import the image from the `glowing_gas.jpg` file into IDL:

   ```
   READ_JPEG, cosmicFile, cosmicImage
   ```

3. Determine the size of the imported image. The image contained within this file is pixel-interleaved (the color information is contained within the first dimension). You can use the SIZE routine to determine the other dimensions of this image:

   ```
   cosmicSize = SIZE(cosmicImage, /DIMENSIONS)
   ```

4. Initialize the display objects required for an Object Graphics display:

   ```
   oWindow = OBJ_NEW('IDLgrWindow', RETAIN = 2, $
      DIMENSIONS = [cosmicSize[1], cosmicSize[2]], $
      TITLE = 'glowing_gas.jpeg')
   oView = OBJ_NEW('IDLgrView', $
      VIEWPLANE_RECT = [0., 0., cosmicSize[1], $
      cosmicSize[2]])
   oModel = OBJ_NEW('IDLgrModel')
   ```

5. Initialize the image object. The INTERLEAVE keyword is set to 0 because the RGB image is pixel-interleaved:

   ```
   oImage = OBJ_NEW('IDLgrImage', cosmicImage, $
      INTERLEAVE = 0, DIMENSIONS = [cosmicSize[1], $
      cosmicSize[2]])
   ```

6. Add the image to the model, then add the model to the view, and finally draw the view in the window:

   ```
   oModel -> Add, oImage
   oView -> Add, oModel
   oWindow -> Draw, oView
   ```

   The following image contains all of the colors of the additive and subtractive primary spectrum. A colorbar annotation can be added to compare the colors of that spectrum and the colors within the image. The color of each box is defined in the following array.

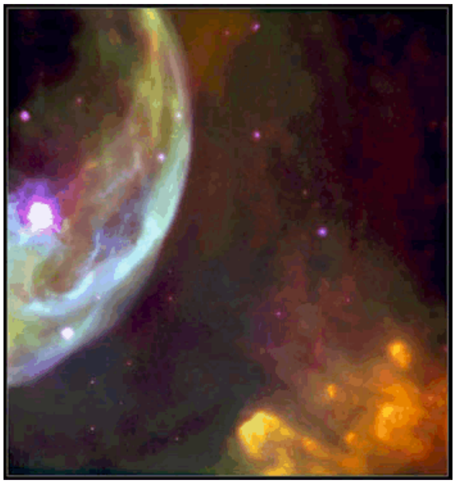


Figure 35: Cosmic RGB Image (Object Graphics)

You can use the following to determine the color and location parameters for each polygon.

7. Initialize the color parameters:

```
fillColor = [[0, 0, 0], $ ; black
   [255, 0, 0], $ ; red
   [255, 255, 0], $ ; yellow
   [0, 255, 0], $ ; green
   [0, 255, 255], $ ; cyan
   [0, 0, 255], $ ; blue
   [255, 0, 255], $ ; magenta
   [255, 255, 255]] ; white
```

8. After defining the polygon colors, you can determine their locations. Initialize polygon location parameters:

```
x = [5., 25., 25., 5., 5.]
y = [5., 5., 25., 25., 5.] + 5.
offset = 20.*FINDGEN(9) + 5.
```

The *x* and *y* variables pertain to the x and y locations (in pixel units) of each box of color. The *offset* maintains the spacing (in pixel units) of each box.

Since the image is made up of mostly a black background, the x border of the colorbar is also determined to draw a white border around the polygons.

9. Initialize location of colorbar border:

```
x_border = [x[0] + offset[0], x[1] + offset[7], $
   x[2] + offset[7], x[3] + offset[0], x[4] + offset[0]]
```

The y border is already defined by the y variable.

These parameters are used when initializing the polygon and polyline objects These objects will be used draw the boxes of the color spectrum and the colorbar border. Each polygon is 20 pixels wide and 20 pixels high. The offset will move the y-location 20 pixels every time a new polygon is displayed.

10. Initialize the polygon objects. The process is repetitive from level to level, so a FOR/DO loop will be used to display the entire colorbar. Since each polygon is drawn individually within the loop, you only need to determine the location of a single polygon and an array of offsets for each step in the loop:

```
oPolygon = OBJARR(8)
FOR i = 0, (N_ELEMENTS(oPolygon) - 1) DO oPolygon[i] = $
   OBJ_NEW('IDLgrPolygon', x + offset[i], y, $
   COLOR = fillColor[*, i])
```

11. The colorbar border is produced with a polyline object. This polyline object requires a *z* variable to define it slightly above the polygons and image. The z variable is required to place the polyline in front of the polygons. Initialize polyline (border) object:

```
z = [0.001, 0.001, 0.001, 0.001, 0.001]
oPolyline = OBJ_NEW('IDLgrPolyline', x_border, y, z, $
   COLOR = [255, 255, 255])
```

12. The polygon and polyline objects can now be added to the model and then displayed (re-drawn) in the window. Add the polygons and polyline to the model, then add the model to the view, and finally redraw the view in the window:

```
oModel -> Add, oPolygon
oModel -> Add, oPolyline
oWindow -> Draw, oView
```

The following figure shows the colorbar annotation applied to the image.

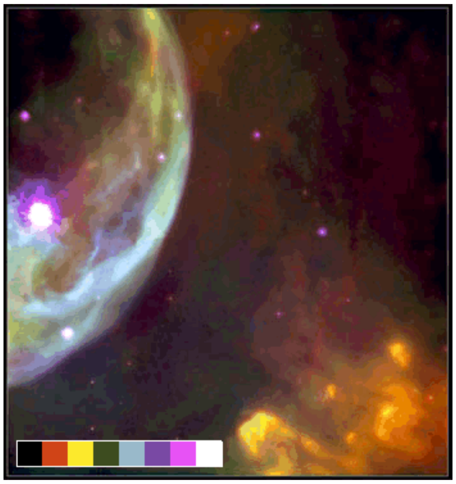


Figure 36: Specified Colors on an RGB Image (Object Graphics)

13. Cleanup object references. When working with objects always remember to clean-up any object references with the OBJ_DESTROY routine. Since the view contains all the other objects, except for the window (which is destroyed by the user), you only need to use OBJ_DESTROY on the view object:

    ```
    OBJ_DESTROY, [oView]
    ```

Example Code: Applying Color Annotations to RGB Images in Object Graphics

Copy and paste the following text into the IDL Editor window. After saving the file as `ApplyColorbar_RGB_Object.pro`, compile and run the program to reproduce the previous example.

```
PRO ApplyColorbar_RGB_Object

; Determine path to "glowing_gas.jpg" file.
cosmicFile = FILEPATH('glowing_gas.jpg', $
   SUBDIRECTORY = ['examples', 'data'])

; Import image from file into IDL.
READ_JPEG, cosmicFile, cosmicImage
```

```
; Determine size of image.
cosmicSize = SIZE(cosmicImage, /DIMENSIONS)

; Initialize objects.
; Initialize display.
oWindow = OBJ_NEW('IDLgrWindow', RETAIN = 2, $
   DIMENSIONS = [cosmicSize[1], cosmicSize[2]], $
   TITLE = 'glowing_gas.jpg')
oView = OBJ_NEW('IDLgrView', $
   VIEWPLANE_RECT = [0., 0., cosmicSize[1], $
   cosmicSize[2]])
oModel = OBJ_NEW('IDLgrModel')
; Initialize image.
oImage = OBJ_NEW('IDLgrImage', cosmicImage, $
   INTERLEAVE = 0, DIMENSIONS = [cosmicSize[1], $
   cosmicSize[2]])

; Add image to model, which is added to view, and then
; display view in window.
oModel -> Add, oImage
oView -> Add, oModel
oWindow -> Draw, oView

; Initialize color parameter.
fillColor = [[0, 0, 0], $ ; black
   [255, 0, 0], $ ; red
   [255, 255, 0], $ ; yellow
   [0, 255, 0], $ ; green
   [0, 255, 255], $ ; cyan
   [0, 0, 255], $ ; blue
   [255, 0, 255], $ ; magenta
   [255, 255, 255]] ; white

; Initialize polygon location parameters.
x = [5., 25., 25., 5., 5.]
y = [5., 5., 25., 25., 5.] + 5.
offset = 20.*FINDGEN(9) + 5.

; Initialize location of colorbar border.
x_border = [x[0] + offset[0], x[1] + offset[7], $
   x[2] + offset[7], x[3] + offset[0], x[4] + offset[0]]

; Initialize polygon objects.
oPolygon = OBJARR(8)
FOR i = 0, (N_ELEMENTS(oPolygon) - 1) DO oPolygon[i] = $
   OBJ_NEW('IDLgrPolygon', x + offset[i], y, $
   COLOR = fillColor[*, i])

; Initialize polyline (border) object.
```

```
z = [0.001, 0.001, 0.001, 0.001, 0.001]
oPolyline = OBJ_NEW('IDLgrPolyline', x_border, y, z, $
   COLOR = [255, 255, 255])

; Add polgons and polyline to model and then re-display
; view in window.
oModel -> Add, oPolygon
oModel -> Add, oPolyline
oWindow -> Draw, oView

; Clean-up object references.
OBJ_DESTROY, [oView]

END
```