# Optimized Algorithms for Displaying 16-bit Gray Scale Images on 8-bit Computer Graphic Systems

Thurman Gillespy III

**Most personal computers contain 8-bit graphic display hardware, whereas most medical gray scale images are stored at 16-bit per pixel integers. To display medical gray scale images on such computers, the 16-bit image data must be remapped into 8-bit gray scale images. This report presents the algorithms and computer code that allow very rapid 16-bit to 8-bit image data transformation. These algorithms are helpful in allowing personal computers with at least the performance of a Macintosh II (Apple Computer, Cupertino, CA) computer to function as low-end picture archiving communication systems or personal workstations.**

***Copyright © 1993 by W.B. Saunders Company***

**KEY WORDS: window width, window level, personal computer, Macintosh, workstation.**

MOST DIGITAL radiographic images are represented as 16-bit (2 byte) integers per pixel element. The actual dynamic range of the image data usually varies from 9 to 12 bits, depending on the imaging modality. When such images are displayed on imaging consoles or picture archiving communication systems (PACS)/teleradiology workstations, a window and level control is usually provided to allow visualization of the full-image dataset. This function is usually implemented with special graphic hardware that allows window and level settings to be applied in real time. However, most personal computers and some low-end PACS/teleradiology workstations have standard 8-bit graphic systems. To display 16-bit images on 8-bit graphic systems, at least two steps are required: First, the 16-bit image data must be remapped to an 8-bit representation using user selected window and level settings. Second, the 8-bit data must be translated into different colors or gray scales (most commonly in hardware using a lookup table) before being projected onto the display monitor. The first step is performed in software and is usually computationally expensive. The second step is performed quite rapidly, usually in real time. This report presents algorithms and C programming language examples that optimize the remapping of 16-bit image data into an 8-bit color/gray scale representation.

## DIRECT CALCULATION TRANSFORMATION

The 16-bit to 8-bit transformation requires that an 8-bit value be substituted for each 16-bit value in the image dataset (Fig 1). Typically this is not a one to one transformation; often image values less than a certain number are clipped to black (in gray scale images), and images values greater than a certain number are clipped to white. Two parameters are usually chosen that determine the slope and end points of this transformation function: the window and level (also known as the window width and window level). From Fig 1, the following definitions are derived:

$$window = x2 - x1 \quad \text{(Equation 1)}$$
$$level = x1 + (window/2) \quad \text{(Equation 2)}$$
$$x1 = level - (window/2) \quad \text{(Equation 3)}$$

The equation for a line on an $x,y$-coordinate system is defined as:

$$y = (m * x) + b \quad \text{(Equation 4)}$$

For the present discussion, $y$ is the 8-bit gray scale/color value of interest, $x$ is the known 16-bit image data value, $m$ is the slope of the line, and $b$ is the $y$-intercept. Thus from Equation 4 the following formulas can be derived:

$$m = \Delta y / \Delta x \quad \text{(Equation 5)}$$
$$m = GRAYSCALES/window \quad \text{(Equation 6)}$$

where *GRAYSCALES* is the difference between the lowest and highest gray scale value (usually 255 Gy).

Furthermore, if $x = x1$, then $y = 0$ (Fig 1). Thus, substituting for Equation 4 we find the following:

$$0 = (m * x1) + b \quad \text{(Equation 7)}$$
$$b = -(m * x1) \quad \text{(Equation 8)}$$
$$b = -(GRAYSCALES/window) * x1 \text{ (substituting Equation 6)} \quad \text{(Equation 9)}$$

*From the Department of Radiology, University of Washington, Seattle, WA.*

*Address reprint requests to Thurman Gillespy III, MD, Department of Radiology, SB-05, University of Washington, Seattle, WA 98195.*

*Copyright © 1993 by W.B. Saunders Company*

*0897-1889/93/0601-0003$03.00/0*

Thus, given the values for *window*, *level*, and *GRAYSCALES*, the 16-bit data can be remapped into 8-bit data as follows:

```
[Code 1]
  #define RSIZE /* size of image data, in pixels, defined
                   as rows × columns of the image matrix*\
  unsigned short    src[RSIZE];  /* array of 16-bit image
                    data         */
  unsigned char     img[RSIZE]; /* array of 8-bit
                    remapped image data */
  short             i, window, level, temp;
  short             b; /* y-intercept, as defined in Eq 8 */
  float             m; /* slope, as defined in Eq 9      */

  for (i = 0; i < RSIZE; i++) {
     temp = (m * src[i]) + b;   /* y = (m * x) + b */
     if (temp > 255 )
        temp = 255 ;
     else if (temp < 0)
        temp = 0;
     img[i] = temp;
  }
```

Although many refinements on the above algorithm are possible, this approach is unavoidably computationally expensive when performed on the data sets that are commonly encountered in medical imaging.

## LOOKUP TABLE TRANSFORMATION

A better approach is to consider the $x$-axis of Fig 1 to represent an array, where each position on the $x$-axis is an element of the array, and the $y$-value (color/gray scale value) is the value of that element (Fig 2). The calculation of this array is as follows:

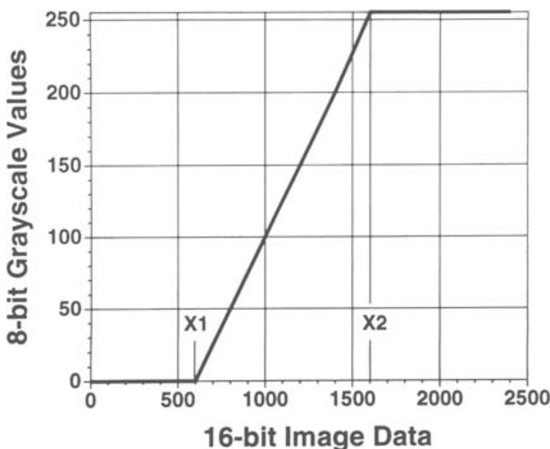


Fig 1. The transformation of 16-bit image data into 8-bit gray scale values.

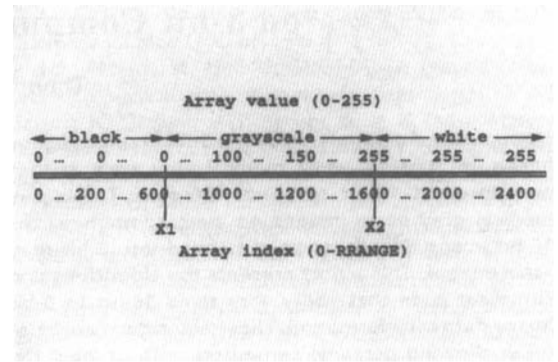


Fig 2. The *lut[ ]* array. Compare *X1* and *X2* with Fig 1.

```
[Code 2]
  #define RRANGE /* the absolute difference between
                    the smallest and largest image
                    data value */
  unsigned char     lut [RRANGE];
  for (i = 0; i < RRANGE; i++) {
    temp = (m * i) + b;
    if (temp > 255 )
       temp = 255 ;
    else if (temp < 0)
       temp = 0;
    lut[i] = temp;
```

Further refinements in the calculation of *lut[]* are possible, but are not essential to the present discussion. The array *lut[]* now contains an ordered set of integers (cast as unsigned chars) from (and usually including) black (0) to white (255) (Fig 2). The 16-bit to 8-bit transformation can now be performed by using the 16-bit image data value of each pixel as the index for *lut[]* to obtain the 8-bit value of interest, as follows:

```
[Code 3]
  for (i = 0; i < RSIZE; i++)
    img[i] = lut[src[i]];
```

This is a common programming algorithm known as a lookup table. Code 3 can profitably be rewritten using pointers and a *do . . . while* loop as in the following code:

```
[Code 4]
  unsigned char     *ip = img;
  unsigned short    *sp = src;
  i = RSIZE;
  do {

    *ip++ = *(lut + *sp++);
  } while (--i);
```

This algorithm is very fast because the only calculations required are a counter decrement, two pointer address increments, and memory transfers. A further slight improvement is possible by "uncoiling" the loop:

*[Code 5]*

```
i = RSIZE/16;
do{
    *ip++ = *(lut + *sp++); *ip++ = *(lut + *sp++);
    *ip++ = *(lut + *sp++); *ip++ = *(lut + *sp++);
    *ip++ = *(lut + *sp++); *ip++ = *(lut + *sp++);
    *ip++ = *(lut + *sp++); *ip++ = *(lut + *sp++);
    *ip++ = *(lut + *sp++); *ip++ = *(lut + *sp++);
    *ip++ = *(lut + *sp++); *ip++ = *(lut + *sp++);
    *ip++ = *(lut + *sp++); *ip++ = *(lut + *sp++);
    *ip++ = *(lut + *sp++); *ip++ = *(lut + *sp++);
} while (--i);
```

## FURTHER OPTIMIZATION

Whereas the above algorithm is a significant improvement, it still requires up to *RRANGE* number of floating point and integer calculations to calculate a new *lut []* for a new window and/or level setting. These calculations are a potential performance bottleneck if nearly instantaneous window and level adjustments are desired. For further optimization, consider the following propositions to be true: (1) Continuous adjustment of the window setting is not required, ie, preset window settings are clinically acceptable (and in fact have been used on many clinical systems in the past). (2) However, continuous or nearly continuous adjustment of the level setting is desirable. (3) The maximum range of level settings required is the difference between those settings necessary to convert the image from complete black to complete white (this range is not always needed, but is the maximum that would be required). (4) Furthermore, consider that for a given window, a change in the level setting only "shifts" the values in *lut []* to different positions within the array (Fig 3).

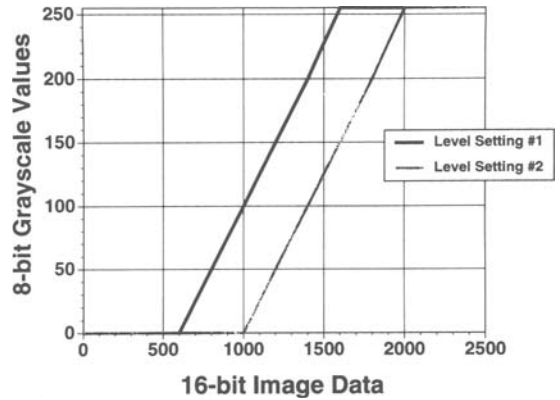


**Fig 3. Different level settings.**

The following data structure is then suggested:

*[Code 6]*

```
unsigned char    baseLUT[ ];
```

This array has 3 regions (Fig 4): (1) The first region is *RRANGE* elements long. Element *baseLUT [RRANGE − 1]*, hereafter labeled *W1*, is fixed. For computed tomographic images, *RRANGE* can be set to 4,096 (12-bit dynamic range). (2) The region from *W1* to *W2* is as large as the current window setting; it contains an ordered set of integers that progresses from black to white. It is initially set to *MAX_WIND_SIZE* (the largest window setting possible) elements long. Whereas the array position *W1* is fixed, *W2* will vary depending on the size of the current window. (3) The region from *W2* to *END* is at least *RRANGE* elements long, and at most *RRANGE* + *MAX_WIND_SIZE* elements long (ie, when *window* = 0).

The *baseLUT[]* array performs the same transformation function as *lut[]* above, but is constructed and initialized differently. At program startup, *baseLUT[]* is allocated (2* *RRANGE*) + *MAX_WIND_SIZE* elements. The elements from *baseLUT[0]* to *baseLUT[W1 − 1]* are set to black (0), and the elements from *baseLUT[W1]* to *baseLUT[END]* are set to white (255).

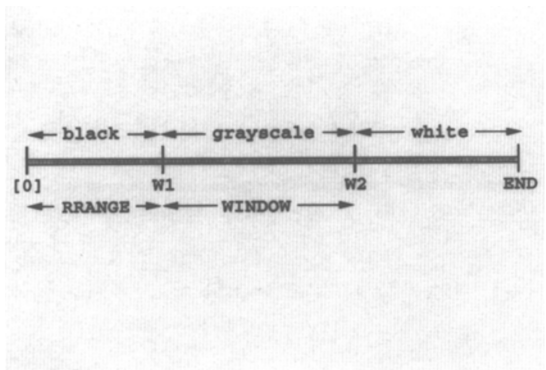


**Fig 4. The *baseLUT[]* array.**

Next, an array of unsigned char arrays is initialized using predefined window settings, where each subarray is defined from a predefined list of window settings. However, unlike *lut[]* above, each array contains at most one white and black gray scale value: the "flat" or clipped portions of the transformation curve on Fig 1 are not included.

```
[Code 7]
  #define NUM_OF_WINDS /* number of window
    settings */
  short myWinds [NUM_OF_WINDS]
    ={100, 200, . . . , 2000}; /* array of defined
    window settings */
  unsigned char * myWind Array[NUM_OF_WINDS];
    /* array of unsigned char arrays, defined from
    myWinds[ ] */
  for (i = 0; i < NUM_OF_WINDS; i++)
    myWindArray[i] = makeLUT(myWinds[i]);
```

*makeLUT()* is a function defined as follows:

```
[Code 8]
  unsigned char * makeLUT(short); /* prototype */
  unsigned char * makeLUT(short window)
  {
    short           i;
    float           m = -(GRAYSCALES/window);
                    /* see Eq. 6 */
    unsigned char   lut[window];
    /* note that b = 0 */
    i = 0;
    while (i < window)
      lut[i++] = m * i;
    return (lut);
  }
```

Finally, a global pointer is used to set thecurrent level setting (Fig 5):

```
[Code 9]
  unsigned char      *gLevel;
```

**Fig 5. The *gLevel* pointer.**

**Table 1. Algorithm Performance**

| Macintosh Computer Model | Central Processing Unit | Time (s) | % Time |
|---|---|---|---|
| II | 16 MHz 68020 | 0.51 | 67 |
| IIci, no cache | 25 MHz 68030 | 0.41 | 65 |
| IIci, 32k cache | 25 MHz 68030 | 0.31 | 63 |
| Quadra 700 | 25 MHz 68040 | 0.17 | 50 |

NOTE: Includes total time to change level control with mouse, calculate new level setting, remap new 8-bit gray scale image from 16-bit image data, update display. Times recorded using profiler option in Think C. The "uncoiled" version of the algorithm was used (code 5), which was 10% to 20% faster than the standard version (code 4), depending on specific Macintosh configuration.

*The percent of the total time spent in the subroutine performing the 16-bit to 8-bit transformation.

Once a window setting is selected, the corresponding unsigned char array in *myWindArr[]* is copied to *baseLUT[]*, starting at position *W1*. To set or change the level setting, the *gLevel* pointer is set to the appropriate position within *baseLUT[]*.

```
[Code 10]
  x1 = level - (window/2);
  gLevel = &baseLUT[W1 - x1];
```

The design of *baseLUT[]* allows continuous level settings with only a change to the *gLevel* pointer address. Furthermore, the array design allows level settings to be selected that set the image from complete black (*&baseLUT[0]*) to complete white (*&baseLUT[W1 + window]*). This algorithm also permits the use of nonlinear lookup tables with no performance penalty.

## IMPLEMENTATION

Using the algorithms above, window and level adjustments might be performed as the following loop:

(1) Store the old selector position (mouse, trackball, dial, etc).

(2) Get the current selector position.

(3) Calculate the difference between the old and new selector positions.

(4) If the difference is not zero, use the difference to calculate a new window and/or level setting, else repeat loop.

(5) If the window setting has changed, copy the appropriate array from *myWindArray[]* to *baseLUT[]*. If the level has changed, calculate *x1*, then change the address of the gLevel pointer within *baseLUT[]*.

(6) Remap the 16-bit data into 8-bit data using the portion of *baseLUT[]* pointed to by *gLevel.*

(7) Update the displayed image using the revised 8-bit data.

The optimized algorithms described above have been implemented in a sample application on the Macintosh (Apple Computer, Cupertino, CA) platform using Think C 5.02 (Symantec Corp, Cupertino, CA). On a Macintosh II class computer, the algorithms allow very rapid window and level adjustments (Table 1). On a Macintosh Quadra computer, the window and level adjustments are performed almost instantaneously. As expected, algorithm performance is strongly influenced by microprocessor performance.

## CONCLUSIONS

The algorithms described above allow nearly real time window and level adjustment of 16-bit gray scale images on computer systems that contain standard 8-bit graphic hardware. Therefore, personal computers with at least the performance of a Macintosh II computer may perform adequately as personal/low-end PACS workstations without the additional expense of special graphic hardware.