

5

If, step 29, the error term is less than half scale, step 36, the error term is decreased by 100 and increased by two scale, and the count is increased by 2 at step 37. If the count is now greater than 4, step 38, step 39, set the mode to the count of 4 or 5 (output a line) and subtract 4 from the count, step 40. If, step 38, the count is less than 4, step 41, the mode is 3 (no output).

This algorithm produces output lines that are the closest approximation to the ideal result. To explain this, the following two examples are given to show how the error dispersion is optimized.

(1) For a magnified image with Scale=120, the objective of the algorithm is to delete 20 lines for every 120 input lines to restore the input image to the original size of Scale=100 and also to ensure that the deleted lines are evenly distributed across the 120 input lines to maintain the integrity of the input image.

For every Scale/(Scale-100) input lines, delete 1 line. In the case of Scale=120, for every 6 lines, delete 1 line.

The number of lines not deleted is:

$\{\text{Scale} / (\text{Scale}-100) = 100 / (\text{Scale}-100)$. For Scale=120, the number of lines not deleted is 5. In other words, keep the first 5 lines and delete the 6th line.

Define a variable err_y, for each line, increase the variable err_y by

(Scale-100), and the next line is the line to be deleted.

When err_y>60, decrease err_y by 100 and start over again.

The algorithm starts out with err_y=0 as an initial condition. For each line kept, increase err_y by (Scale-100) until err_y exceeds $\frac{1}{2}$ Scale. Delete the next line and decrease err_y by 100 and start over again. Essentially, err_y is offset from $(\frac{1}{2} \text{Scale}-100)$ to $\frac{1}{2}$ Scale.

For the case of Scale=120, $100 / (\text{Scale}-100) = 5$ after 5 lines, and err_y is increased to its full range of 100. For the next line, delete the 6th line and decrease err_y by 100 and there is no left over in err_y.

For Scale=115, $100 / (\text{Scale}-100) = 6$ with a remainder of 10. After 7 lines, err_y is increased to an absolute range of 105. Delete the 8th line and decrease err_y by 100. The value of 5 is the error diffused to the next round of computation.

(2) For a reduced image with Scale=80, the objective of the algorithm is to duplicate 20 lines for every 80 input lines to restore the input image to the original size of Scale=100 and also to ensure that the duplicated lines are evenly distributed across the 80 input lines to maintain the integrity of the input image.

For every Scale/(100-Scale) input lines, duplicate 1 line.

In the case of Scale=80, for every 4 lines, duplicate 1 line. The number of lines not duplicated is:

$\{\text{Scale} / (100-\text{Scale}) - 1\} = (2 * \text{Scale} - 100) / (100 - \text{Scale})$. For Scale=80, the number of lines not duplicated is 3. In other words, keep the first 3 lines. At the end of the 4th line, duplicate the 4th line.

Define a variable err_y, for each line. Decrease the variable err_y by

(Scale-100). Until err_y<(2*Scale-100), the next line is the line to be duplicated. Add err_y by (2*Scale-100) and start over again.

The algorithm starts out with err_y=0 as an initial condition. For each line, decrease err_y by (Scale-100) until err_y is less than $-\frac{1}{2}$ Scale. Duplicate the next line at the end of the next line and add err_y by (2*Scale-100) and start over again. Essentially the range of err_y is offset from $-\frac{1}{2}$ Scale to $[-\frac{1}{2} \text{Scale} + (2 * \text{Scale} - 100)]$.

6

For the case of Scale=80, $(2 * \text{Scale} - 100) / (100 - \text{Scale}) = 3$ after 3 lines, and err_y is decreased the full range of 60. For the next line, duplicate the 4th line and increase err_y by 60 and there is no left over in err_y.

For Scale=85, $(2 * \text{Scale} - 100) / (100 - \text{Scale}) = 4$ with a remainder of 10. After 5 lines, err_y is decreased to a range of -75. At the end of 6th line duplicate the 6th line and increase err_y by 70.

The value of -5 is the error diffused to the next round of computation.

The circuit for accomplishing the averaging, subsampling and resizing is the two-stage pipe-lined circuit of FIG. 2. The pixels are input at line 50. The first of the four input pixels of each block bypasses the adder 52 and is applied through the mux 51 to load the register 53. On the next three clocks, the pixels are used as one input to the adder 52 and the register content is used to supply the running sum to the other input. After the four input pixels of the block are summed into a ten bit number, the sum is divided by 16 at the shifter 54, and stored through mux 55 into the line buffer 56. (To get an average of four pixels, one would normally divide by four, but here it is known that eventually four lines of four pixels each will finally be summed, so the division by sixteen is done here in one step). At the end of the first scan line, the line buffer has a partial sum for each pixel.

For the next scan lines the sum from shifter 54 is applied to one input of adder 58 and the other adder input receives the partial sum from the line buffer transferred through mux 59 to the output register 60. If this result is yet another partial sum, it is loaded back into the line buffer 56 through mux 55. If this is the final averaged pixel, it is (usually) output through register 60.

A complication is that in mode 0 the entire scan line is ignored, and the line counter is not incremented. In all cases 16 pixels need to be averaged for each output pixel. In this Mode 0 case, there will be one output for five input scans, but only four will be used to supply the required total of 16 pixels.

Similarly, in Table 2 there are times when there is an output every three scan lines. To get 16 pixels in this case, one set is sent to a doubler 61 and added in adder 62 to get the correct output

In addition, the modulo 3 counter 64 is used in the algorithm to generate the mode numbers as shown in the flow chart of FIG. 2, and a RAM address counter 63 is used to count down scan line clocks to generate an address pointer for RAM 56.

This entire circuit uses four clocks to average the four input pixels, and the remainder of the circuit is pipe-lined so that it will execute on the first clock of the next block.

While the invention has been described with reference to a specific embodiment, it will be understood by those skilled in the art that various changes may be made and equivalents may be substituted for elements thereof without departing from the true spirit and scope of the invention. In addition, many modifications may be made without departing from the essential teachings of the invention.

What is claimed is:

1. A method of resizing and subsampling a video image of pixels by either outputting or not outputting each scan line of pixels, where the subsampling is by a factor of n, where n is an integer, comprising:

going to a first scan line, setting a scan count to zero, and setting an error value;

a) going to a next scan line and computing an error value that is a function of the error that would result if the next scan line were to be output,