April 6, 2011

A New Line Drawing Algorithm Based on Sample Rate Conversion

©2002, C. Bond. All rights reserved.

1 Overview

In this paper, a new method for drawing straight lines suitable for use on raster scan displays and plotters is presented. Unlike previous methods which are based on DDA or midpoint algorithms, this technique is based on signal processing concepts related to resampling, multirate processing and sample rate conversion generally, and decimation in particular.

An algorithm based on spatial decimation is developed which has attractive features including the elimination of test, compare and branch operations within the inner plotting loop. Furthermore, all multiplies, divides, shifts and other complex CPU operations are eliminated from the inner loop, as well.

Lines can be drawn in any orientation with 'nearest pixel' accuracy using only the primitive CPU operations of addition (subtraction) and incrementing (decrementing). With some CPU architectures, including current Intel and AMD processors, it is possible to hold all variables in CPU registers so that the only memory accesses required in the inner loop relate to the plotting function.

2 Background

A variety of line drawing algorithms have been published in the literature. Among these are algorithms by Bresenham [1], Wu [5] and others. The central problem solved by these algorithms is to find a 'best fit' to an ideal line, given the constraints imposed by a raster scan or integer grid limited display. See also Pitteway [3], and Foley [2].

These *prior art* solutions are typically derived from the differential quantities associated with a straight line and focus on correct pixel selection or

improvements in efficiency.

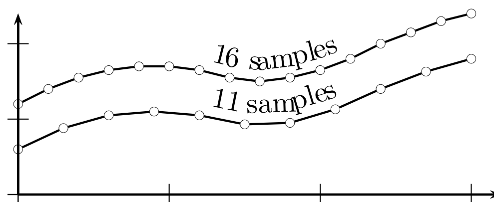
The new method is based on signal processing concepts. For general information about signal processing see Ifeachor [6] and Oppenheim [7].

3 The Algorithm

The derivation of the new algorithm will focus on applying decimation or rate multiplying methods to the problem of drawing a line in the first octant. The other octants can be supported by changing the signs of x and y, or exchanging the roles of x and y, or both.

3.1 Decimation

In signal processing, a significant and recurrent problem has to do with aligning data sets associated with different sample rates. For example, it may be necessary to convert a data set sampled at 48 kilosamples per second to data set with equivalent sampling at 44.1 kilosamples per second. The challenge is to align the ends of the source and target data sets, and devise a resampling strategy which produces the target set from the source set with minimum error. When the target set has a lower sample rate (fewer samples per second) this process is called decimation. The following figure illustrates decimation for a short signal fragment.



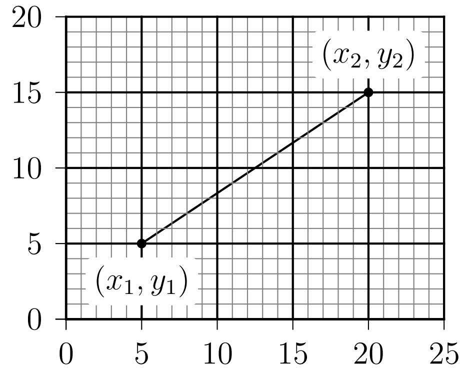
In this figure, the upper trace represents a source data set consisting of 16 samples. The lower trace represents the same signal decimated (downsampled) to 11 samples. It is offset vertically for clarity.

Decimation in a sampled digital environment is a *dirty* process, in the sense that sometimes dubious assumptions are made about the behavior of the signal between sample points. Nevertheless, most signals of interest are rea-

sonably smooth, so the errors are generally acceptable.

Since we will be applying decimation techniques to straight lines, rather than arbitrary curves, there are no assumptions required and the process introduces no errors other those inherent in the scan conversion.

How is decimation applied to the line drawing problem? Consider the line in the figure below.



The x coordinates of each pixel can be viewed as a uniform sample set. The y coordinates represent another sample set. Assuming, without loss of generality, that the slope of the line is within the first octant, the x set will be the larger of the two. Given the coordinates of the endpoints and using decimation, we can generate the y coordinates by resampling the set of x coordinates. In a sense, we have rotated the resampled sequence by 90 degrees.

The decimator is the key to making this process efficient. Its purpose is to step along the same interval at two different rates. As applied here, it works by algorithmically matching the y coordinate set to the x set by taking unit steps along the x coordinate and stepping at a different rate along the y axis. Since the y steps typically involve a fractional quantity, fixed point math will be used to track the *exact* y value.

The integer part of y, which is the only part usable on the raster grid, is found by updating a control variable which represents the fractional part of the y coordinate at each step. When the control variable passes through the next integer value, the y coordinate is incremented. The update value is simply the ratio the total y distance over the total x distance. Note that this process bears a superficial resemblance to line drawing by DDA methods,

but the emphasis on decimation suggests a new solution for the y coordinate update problem.

3.2 The New Solution

The y coordinate is incremented when the fractional part of the control variable overflows, increasing the integer part by unity. Now identify the y coordinate with the integer part of the control variable and allocate a separate integer (register) for the fractional part. If the fractional part is scaled so that it occupies an entire unsigned integer, its overflow will generate a CPU carry. By simply adding the carry to the y coordinate, we can update it properly without any need for correcting the fraction or testing any of the other variable quantities.

To understand how this is done, we must see how to scale the fractional portion of the update quantity to fit an integer. Let the control variable be **cvar** and the update quantity be **incr**. Assuming 16-bit integers,[1] the generating equation for **incr** is:

$$\frac{y_2 - y_1}{x_2 - x_1} = \frac{\Delta y}{\Delta x} \quad (1)$$

$$\frac{\Delta y}{\Delta x} = \frac{\mathbf{incr}}{65536} \quad (2)$$

$$\mathbf{incr} = \frac{\Delta y}{\Delta x} \cdot 65536. \quad (3)$$

The multiplication by 65536 is even simpler than a word shift, because 16-bit integer division on most processors leaves the fractional part in a specific register as a remainder. Just take that register as **incr**. **cvar** is initialized to 32768 (0x8000), which is the fractional equivalent of 0.5.

The new algorithm is summarized in the following pseudo-code for the inner drawing loop:

[1] 16-bit integers are not required by the algorithm. 32-bit integers could also be used.

1. **cvar**$\leftarrow$ `0x8000`, **incr**$\leftarrow \Delta y/\Delta x$,
2. $\mathbf{x} \leftarrow \mathbf{x_1}$; $\mathbf{y} \leftarrow \mathbf{y_1}$,
3. `plot` **x**,**y**,
4. `add` **incr** `to` **cvar**,
5. `add carry to` **y**,
6. `increment` **x**,
7. `repeat 3-6 until done` $(\mathbf{x} = \mathbf{x_2})$.

Observe that in step 5, if no carry is generated, the y coordinate will be unchanged.

4 Conclusion

It has been shown that it is possible to devise a line drawing algorithm such that the inner drawing loop requires no test, compare or branch instructions. In addition, there are no complex CPU instructions, such as multiplies, divides or shifts.

For the case shown, with the line in the first octant, we have identified the y coordinate with the integer portion of the control variable, and allocated a separate integer for the fractional part. By separating the integer and fractional parts, we have found a means to employ the CPU carry bit as a convenient, efficient device to transfer integer updates to the y coordinate without auxiliary tests, branches or correction factors.

The following quantities are required to support the algorithm

x: The x coordinate,

y: The y coordinate,

incr: The update quantity,

cvar: The control variable.

All these values can be held in processor registers during the plotting process.

4.1 Optimization

For the basic algorithm operating in octant I, the x value and the loop control variable both increment in unit steps at each iteration. Hence, it may be possible to combine the two operations in such a way as to reduce the number of increments by one, per iteration. There are several ways to do this, and the performance advantage of any particular method is processor dependent.

An even more efficient method results from allocating a 16-bit control variable in memory followed by the 16-bit y-coordinate. Then, if the control variable is updated using 32-bit adds, the y value is automatically updated without additional operations. This eliminates step 5 of the algorithm.

4.2 Extension to other Octants

Interchanging the roles of x and y will map the algorithm for octant I to octant II. Changing the signs the update quantities for x and/or y will map to other quadrants. These modifications in combination will extend the basic algorithm to work in any octant.

There are several high level strategies possible for implementing a general line drawing algorithm. An obvious method is to implement 8 versions of the algorithm, one for each octant, and traverse a decision tree to determine the appropriate octant from the given line coordinates. Another is to derive signed update quantities based on the required octant and use them for incrementing (decrementing) and adding (subtracting) in the inner loop.

These methods have been used successfully by other line drawing algorithms and are not detailed here.

5 Comments

5.1 Downsampling

The line drawing algorithm presented above is based on an efficient linear downsampler (decimator) which has been used for many years by the author. An observation that there was an equivalence between the integer indices used

in sample sequences, and the integer grid coordinates of the display inspired the adaptation. A brief description of the downsampler should clarify its functional equivalence with the line drawing routine.

In operation, the signal decimator uses the integer x and y values as indices into source and target sample vectors, $z[n]$ and $\overline{z}[m]$. **incr** is initialized to m/n, and x, y and **cvar** are initially set to zero. To start, $\overline{z}[0] \leftarrow z[0]$. Operations may be performed in floating point or fixed point math, but recall that *cvar* is a fraction.

In the inner decimation loop, x is incremented and **cvar** is updated by the addition of **incr**. When **cvar** overflows, y is incremented and the array entry for $\overline{z}[y]$ is set to $z[x] + (z[x+1] - z[x]) * \textbf{cvar}$. Iteration is continued until $x = n$.

5.1.1 Decomposition of Control Variable

There are several ways to interpret the new algorithm. Its basic strategy is to separate the integer and fractional parts of the control variable so the y coordinate is identified with the integer part and is immediately accessible to the plot routine without modification. The fractional part is **cvar**.

This decomposition operates the same way for the two 16-bit integers as would a 32-bit fixed point control variable consisting of a 16-bit upper integer part and a lower 16-bit fractional part. For the 32-bit variable, updating the upper part occurs automatically with the addition of **incr**. However, the y coordinate cannot be directly accessed except by right shifting the variable by 16 bits. Furthermore, since the control variable must be maintained from one iteration to the next, the right shifting must be done on a temporary copy, involving another operation. Hence, the 32-bit solution requires a copy operation followed by a 16-bit right shift. For comparative purposes the new algorithm replaces the copy and right shift with a simple add instruction.

For Intel and AMD processors the code sequence using a 32-bit control variable is:

```
mov tmpreg,rega     ; tmp ← cvar
shr tmpreg,16       ; recover y coordinate
```

For the new algorithm, with separated integer and fractional parts, this is replaced by:

```
adc yreg,0            ; update y
```

References

[1] J. E. Bresenham, "Algorithm for Computer Control of a Digital Plotter," *IBM Systems Journal*, 4(1), 1965, pp.25-30.

[2] James D. Foley, et al., "Computer Graphics, PRINCIPLES AND PRACTICES," *Addison-Wesley Publishing Company*, 1997, pp. 72-81.

[3] M. L. V. Pitteway, "Algorithm for Drawing Ellipses and Hyperbolae with a Digital Plotter," *Computer J.*, 10(3), Nov. 1967, pp. 282-289.

[4] J. C. Rokne, et al., "Fast Line Scan Conversion," *ACM Transactions on Graphics*, 1990.

[5] X. Wu and J. G. Rokne, "Double-Step Incremental Generation of Lines and Circles," *Computer Vision, Graphics and Image Processing*, 37, pp. 331-334.

[6] Emmanuel C. Ifeachor and Barrie W. Jervis, "Digital Signal Processing, A Practical Approach," *Addison-Wesley Publishing Company*, 1993.

[7] Alan V. Oppenheim and Ronald W. Schafer, "Digital Signal Processing," *Prentice-Hall, Inc.*, 1975.