A Lab Exercise for Rasterizing Lines

Dr. David J Stahl, Jr.
US Naval Academy
stahl@usna.edu

Abstract: *Rasterizing lines is one of many fundamental topics in an introductory graphics course, with Bresenham's Algorithm particularly well suited for student implementation. By having students complete carefully prepared scaffold code, understanding is reinforced by means of an exercise no more difficult than a short lab assignment. To accomplish this a particular code framework is imposed that allows an instructor to focus student effort on the algorithm while avoiding details of the visualization API.*

Keywords: 2D modeling, algorithm development, rendering.

1 Introduction

Educators in computer graphics must deal with a difficult challenge as the discipline matures. Faced with the seductive allure of spectacular - and commonplace – computer generated imagery, it becomes increasingly difficult to capture student attention and motivate enthusiasm for fundamental concepts. Student critiques in introductory graphics courses reveal a general lack of interest with the underlying mathematics and theory that produce impressive visuals. They desire an ability to produce similar results without having to understand the algorithms or mathematics on which they are based. Since fundamental graphics techniques remain a core topic in the computing body of knowledge, however, the challenge is to make the theoretical underpinnings of computer graphics more palatable. This paper describes one exercise in a series that approaches the problem with active learning via short programming exercises. After gaining a basic understanding of an algorithm or technique through the classical approach of lecture and reading, a student can typically complete an implementation during a one to two hour lab period when provided with carefully prepared scaffold code. This approach puts theory into practice in a manner that rewards effort with immediate visual results. Moreover, the programming framework imposed on student solutions permits focusing their effort on the algorithm rather than the API used to render it.

2 Educational Goals

A student who completes this lab exercise will have demonstrated an understanding of line rasterization by implementing both a Digital Differential Analyzer and Bresenham's Algorithm from pseudo-code. The student should be able to explain the general concept of line rasterization, discuss how the particular technique of Bresenham's Algorithm achieves its speed efficiency, and extrapolate it to rasterizing circles. Students will benefit from the additional practical programming experience gained. The student's working code could easily form one module of a rendering API they further implement using this same approach.

3 Methodology

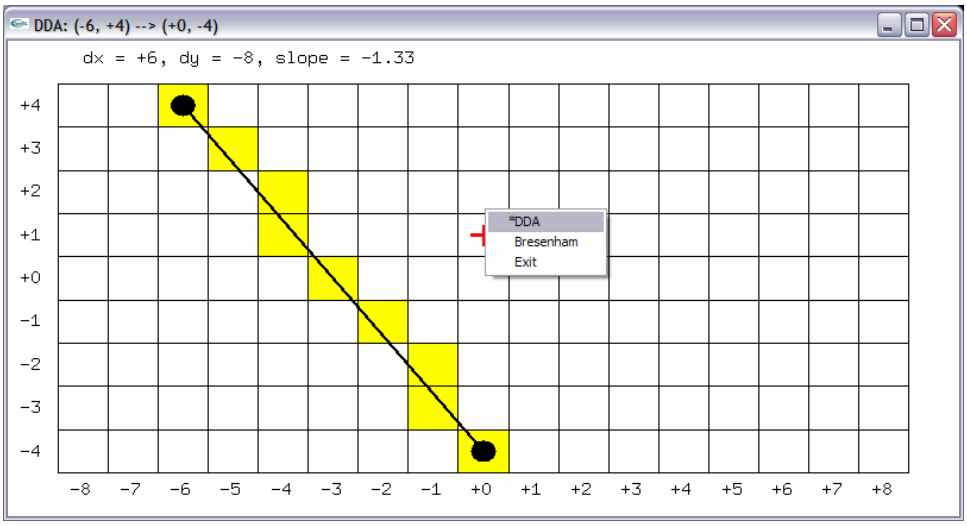
The target audience consists of undergraduate students taking an introductory computer graphics course. The lab exercise is one component of the following pedagogical approach used throughout the course. A lecture that follows an out-of-class reading assignment is used to motivate the general topic. As students will eventually implement some aspect of the current topic, as far as possible we illustrate concepts through pseudo-code and examples worked by hand, with students expected to take notes by fleshing out details on a minimal handout. Immediately following this approximately one-hour classroom lecture, the lab assignment is introduced and the students begin programming. We have used this and similar exercises in a 3-credit hour course meeting twice a week. The schedule is nominally one hour of lecture followed immediately by one hour of lab with lecture topic and lab closely tied. This arrangement allows the flexibility of increasing or decreasing either lecture or lab time as needed.

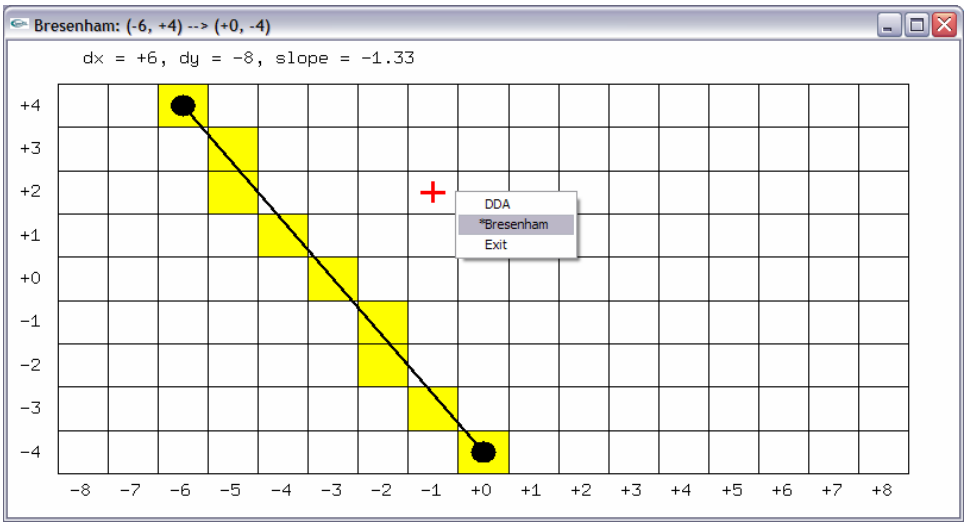
The course has been taught in a Sun workstation lab using *Solaris*, *g++*, *xemacs* and *make*, and in a PC lab using *Windows XP*, and *Microsoft Visual Studio C++ 6.0*. We use OpenGL as the graphics API, and GLUT3.7 for interfacing with the windowing system. Any platform that supports GLUT could be used. In fact, it is usually the case where at least one student feels more comfortable working on a Macintosh laptop brought to the lab.

Lab assignment: Rasterizing Lines

Using pseudo-code and your notes from the lecture, complete the following two functions to rasterize a line from (X1, Y1) to (X2, Y2). You only need to add code to **lines.cpp**: no other file should be modified.

```
void dda( GLint X1, GLint Y1, GLint X2, GLint Y2 );
void bres( GLint X1, GLint Y1, GLint X2, GLint Y2 );
```





Notice how these two algorithms rasterize the line differently!

Figure 1. Line rasterization assignment. Students are given scaffold code that must be completed.

To begin work on the exercise a student downloads from the course website an archive (tar or zip) containing the lab assignment document, an executable example of a correct solution with sample data as needed, and one or more source code files that includes scaffold code. The provided code is structured to enforce separation by file according to logical purpose, with a separate header and source file pair for each set of related GLUT callbacks. The typical set of files consists of:

`main.[h, cpp]`	- event-driven loop entry
`init.[h, cpp]`	- GLUT, OpenGL, and application setup
`render[.h, cpp]`	- display callback and idle callback
`view.[h, cpp]`	- reshape, visibility and entry callbacks
`mouse.[h, cpp]`	- mouse interaction callbacks
`keyboard.[h, cpp]`	- keyboard interaction callbacks
`menu.[h, cpp]`	- menu callbacks

Course policy requires students to adhere to this organization. Enforcing this arrangement has several benefits. The common framework makes it easy to locate specific code according to its functionality and affords collaborative work when such is desired. As a practical matter it makes grading assignments much easier and simplifies the Makefile when the development environment requires one. Most importantly however, an easily understood common code framework obviates students having to spend time and effort in designing their own high-level code structure, but rather allows them to focus effort on implementing the algorithm. This code organization is introduced early in the course in conjunction with the discussion of event-driven programming. Students quickly become familiar with the code structure, easily moving from file to file as needed.

The provided source code is typically created by the instructor from a fully commented, working solution. Key parts of the algorithm are removed, leaving a scaffold. Students must refer to notes or texts to finish the implementation. We have found that giving the students all or most of the code not directly related to the algorithm or technique in question avoids the frustration of getting bogged down in details ancillary to the problem at hand. For the most part this also results in short lab exercises that students are able to complete in the hour following presentation of the theoretical material.

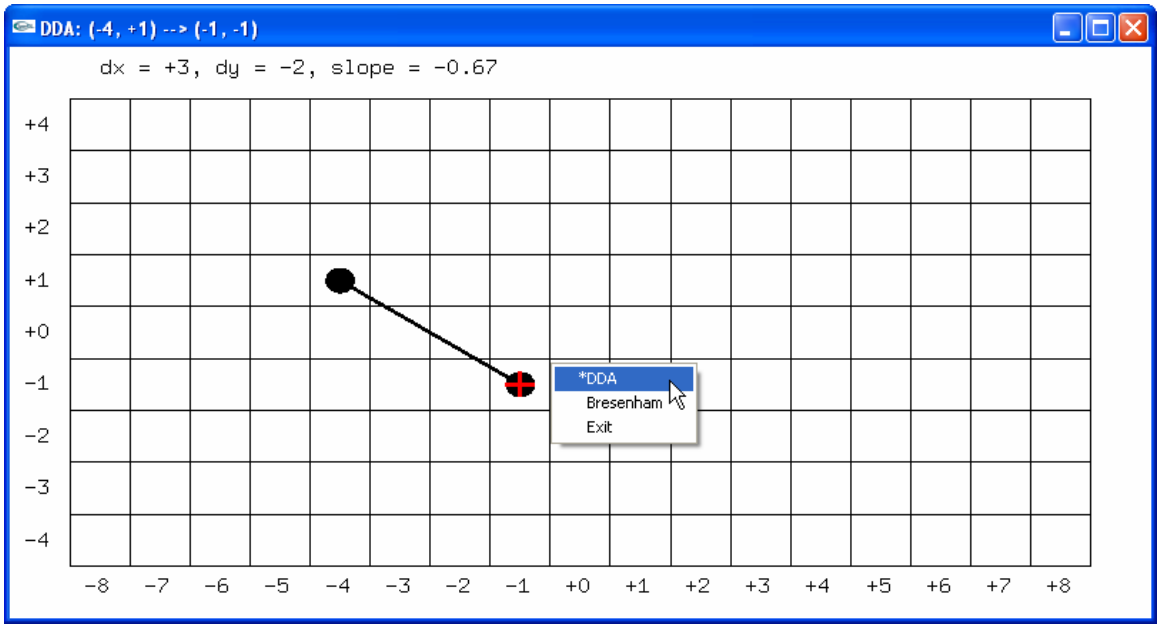


Figure 2. Output of compiled scaffold code as given to the students, showing a 17 x 9 raster with two pixels selected as endpoints of a line from `(-4, +1)` to `(-1, -1)`. The student must implement code to rasterize the line using a DDA algorithm and Bresenham's Algorithm.

```
// DDA(x1, y1, x2, y2), all octants:

compute dy
compute dx
plot(x1, y1)

if –1 < slope < +1      // Shallow slope case.
  compute m             // Calculate in the usual way: dy/dx.

  // 1st endpoint could be on the left or the right ...
  // should we increment x (+1) or decrement x (–1)?
  dx < 0 ⇒ dx = –1, otherwise dx = +1

  m *= dx  // Same variable m used for positive and negative slopes.
  y = y1   // Start at 1st endpoint's scanline.

  // Go from the 1st endpoint to the 2nd.
  while( x1 != x2 )
    increment x1 by dx
    increment y by m
    plot( x1, floor(y + 0.5) )

else // Steep slope case.
  // same algorithm as above, except
  // switch x's and y's, dx's and dy's
  plot(floor(x + 0.5), y1 )
```

```
Bresenham(x1, y1, x2, y2) // All octants

 1. steep = |y2 – y1| > |x2 – x1|
 2. if( steep )         // Reflecting about y = x switches x's and y's
 3.   swap(x1, y1)      // to convert a steep slope case to a shallow
 4.   swap(x2, y2)      // slope case.

4a. if( x1 > x2 )       // Swap endpoints so algorithm only has to deal
4b.   swap(x1, x2)      // with going from left to right in x
4c.   swap(y1, y2)

 5. dx = x2 – x1        // Line 4b ⇒ absolute value not needed
 6. dy = |y2 – y1|      // (saves one operation)

 7.  e = 0              // Initialize the error term, the error term
 8. de = dy             // increment, and the initial scanline.
 9.  y = y1             // ystep accounts for + or – slopes.
9a. if( y1 < y2 ) ystep = +1 else ystep = –1

10. for( x from x1 to x2 )    // Same loop for all cases!
11.   if( steep ) plot(y,x) else plot(x,y)
12.   e = e + de
13.   if( 2e ≥ dx )
14.     y += ystep
15.     e = e – dx
```

Figure 3. Pseudo- code for line drawing algorithms.

```
// Rasterize a line from (X1,Y1) to (X2,Y2)
void dda( GLint X1, GLint Y1, GLint X2, GLint Y2 )
{
  int dy = 0;  // = ? (initialize from endpoints)
  int dx = 0;  // = ? (initialize from endpoints)
  double m;    // slope of the line
  double x, y; // algorithm increments x or y, depending on slope

  ; // draw the pixel corresponding to the left endpoint (see draw_pixel() in render.cpp)

  // Shallow slope case: start at X1 and on scanline Y1, then
  // increment X1, going to the next scanline when necessary,
  // until we get to the second endpoint (X2,Y2)
  if( 0 ) // test for |slope| < 1 ... => shallow slope
  {
    m  = 0;  // = ? (compute slope in the usual way: rise over run)
    dx = 0;  // = ? (should we increment X1 by -1 (go left) or by +1 (go right) ?)
             //     (That depends on the left-right order of X1 and X2           )

    m *= dx; // (slope can be positive or negative)
    y  = Y1; // (start on the first endpoint's scanline)

    // Going from X1 to X2 ...
    while( 0 /* ... we haven't yet reached the second endpoint */ )
    {
      ; // (step to next x value)
      ; // (add slope to y value)
      ; // (draw this pixel)
    }
  }
  // Steep slope case: switch x's and y's, dx's and dy's from the shallow case.
  // Start on scanline Y1 at X1, then increment Y1 (move one scanline at a time),
  // going to the next X value when necessary, until we get to the 2nd endpoint (X2,Y2)
  else // |slope| >= 1 ... => steep slope
  {
    m  = 0;  // = ? (compute slope as "rise" over "run"; dx and dy are switched!)
    dy = 0;  // = ? (should we increment Y1 by -1 (go down) or by +1 (go up) ?  )
             //     (That depends on the top-bottom order of Y1 and Y2          )

    m *= dy; // (slope can be positive or negative)
    x  = X1; // (start at the first endpoint in x)

    // Going from Y1 to Y2 ...
    while( 0 /* ... we haven't yet reached the second endpoint */ )
    {
      ; // (step to next y value)
      ; // (add slope to x value)
      ; // (draw the pixel)
    }
  }
}
```

Figure 4. Scaffold code provided to the student for the DDA algorithm.

```
// Rasterize a line from (X1, Y1) to (X2, Y2)
void bres( GLint X1, GLint Y1, GLint X2, GLint Y2 )
{
  bool steep = 0; // ? (check for steep slope case)

  if( steep ) // switch our notion of x and y for each endpoint
  {
    ; // (first endpoint)
    ; // (second endpoint)
  }
  if( 0 /* first endpoint is to the "right" of the second )
  {
    ; // reverse the endpoints
    ;
  }
  int dx = 0; // = ? (computed as usual)
  int dy = 0; // = ? (computed as an absolute value; ystep takes care of direction)
  int e  = 0; // (no error, initially)
  int de = 0; // = ? (value to adjust error term by)
  int y  = 0; // = ? (start on first endpoint's scanline)

  int ystep; // (set this below, based on endpoint Y values)
  if( 0 /* first endpoint is "below" second endpoint */ )
    ystep; // = ? (y increment will be positive)
  else
    ystep; // = ? (y increment will be negative)

  for(int x; 0 ; ++x ) // go from X1 to X2 ('X' really is 'Y' if this is steep slope case)
  {
    // Draw the pixel (see draw_pixel() in render.cpp)
    // remember: steep slope case swaps x and y!

    // Add to the error term

    if( 0  ) // See if y increments
    {
      ;  // Increment y
      ;  // Reset error term
    }
  }
}
```

Figure 5. Scaffold code provided to the student for Bresenham's Algorithm.

4 Assessment

Student feedback across our computer science curriculum consistently expressed a desire for more "hands-on" programming experience and less exposure to math and theory. In addressing this a common challenge was to be faced regardless of the course: designing practical exercises that are interesting and not "toy" problems, that are focused tightly on the subject matter to minimize effort spent on ancillary implementation details, yet are simple enough to reasonably be accomplished within the time span of a scheduled lab period. Providing this in an introductory graphics course is perhaps easier than in other domains as the subject matter by nature lends itself to visualization: we might just design lab exercises that have them "draw things". Yet students continually are surprised by the details that must be addressed to simply draw a line with the mouse in a window on a raster display. Forcing students in an undergraduate course to deal with all of the details of windowing, interaction, and rendering often leads to frustration.

The pedagogical approach we describe in this paper meets the expressed desire of our students for more practical programming, and it does so in a manner that appears to work: in several offerings of an introductory graphics course using this approach not a single student has been unable to complete the lab exercise on rasterizing lines. With theoretical material presented first, being able to work exclusively on implementing an algorithm and having immediate visual feedback - whether it indicates error or success in the implementation - seems to make students more willing to absorb the underlying theory. The latter conclusion has been evidenced by a steady decrease in the number of complaints about "too much theory!", in comments made on student course critiques as the number of course topics employing this approach has increased, year to year.

5 Conclusions

Developing graphics course content so as to present theory in a palatable, engaging manner via short lab exercises is no small task. It takes considerable instructor time to carefully prepare an integrated set of lecture notes, handout materials, and scaffold code designed to allow students to reasonably complete an algorithm implementation in one lab period. Yet we have discovered the payoff is worth the effort: while perhaps not enthusiastically embracing the theoretical underpinnings of key graphics techniques and algorithms, undergraduate students are at least able to apply theoretical knowledge to create correct implementations. The specific lab exercise topic described here, the Bresenham's Algorithm and the DDA algorithm on which it is conceptually based, is one such example. Other fundamental techniques such as polygon rasterization and 3D viewing could lend themselves to this approach as well.

References

Foley, J.D., Van Dam, A., Feiner, S. K., Hughes, J.F., Phillips, R.L., 1994. Introduction to Computer Graphics, Addison-Wesley, New York, Chapter 3.2, pp. 72-79.

Handout: Rasterizing Lines

A. Naïve approach: use line equation: _____

```
// first octant, 0 < m < 1
m = _____
b = _____
for( x = ____; x ≤ _____; ++x )      ←
  y = _____
  plot( x, _____ )      ← choose the pixel _____
```

Inefficient due to _____ .

B. Improvement:

$y_i = $ _____

$y_{i+1} = $ _____ + b

$= $ ____ (_____) + b

$= $ _____

$= $ _____

$y_{i+1} = $ _____ ← next value of y obtained by _____

C. Pseudo-code:

```
// first octant, 0 < m < 1, line from (x1,y1) to (x2,y2)
compute ____
compute ____
plot( __, __ )
compute ____
y = __
while( _____ )
  increment x1 by _
  increment y by _
  plot( x1, _____ )
```

No multiplies ⇒ this is called an

"_____"

or, a "DDA":

"_____"

Figure 6. Part of a handout tied to subject material presented to students in a lecture. Completing the handout facilitates completing the associated lab assignment.