A Lab Exercise for 2D Line Clipping

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

Abstract: *Line clipping is a fundamental topic in an introductory graphics course. The simplicity and elegance of the classic Cohen-Sutherland 2D Line Clipping Algorithm makes it suitable for implementation by the student in a lab exercise. An understanding of the algorithm is reinforced by having students write actual code and see the results. A code framework is provided that allows an instructor to focus student effort on the algorithm while avoiding the details of the visualization API used to render the results.*

Keywords: 2D modeling, algorithm development, rendering.

1 Introduction

Fundamental graphics techniques are a core topic in the computing body of knowledge. Teaching these techniques to today's computer science undergraduates presents a pedagogical challenge, however: theory is not so captivating when the spectacular graphics seen in movies, advertising and games is viewed by the typical graphics student as the expected norm. Course critiques reveal student dissatisfaction with the stale nature of theory presented without "live" examples. The approach described here attempts to make the mathematical and algorithmic aspects of computer graphics more palatable through short programming exercises. Having gained a basic understanding of an algorithm or technique through lecture and reading, a student typically can program an implementation during a one to two hour lab period when provided with scaffold code. This active learning approach thus serves to put theory into practice in a manner that rewards effort with immediate visual results. Moreover, the programming framework permits focusing student effort on the algorithm rather than ancillary details of the visualization API. This paper describes one exercise in a series.

2 Educational Goals

A student who completes this lab exercise will have demonstrated an understanding of 2D line clipping by implementing the Cohen-Sutherland algorithm from pseudo-code. The student should be able to explain how the algorithm works, discuss how the algorithm achieves its speed efficiency, and relate it to similar algorithms such as the Midpoint Subdivision technique. As a practical benefit, the student will gain further experience making the transition from a pseudo-code specification to an actual implementation. With working code as an artifact, the student should then be able to extend the method by implementing the 3D case, perhaps as a component of a 3D viewing pipeline.

3 Methodology

The target audience consists of undergraduate students taking an introductory computer graphics course. The lab exercise is one component of the following general approach. Following an out-of-class reading assignment, lecture material is used to motivate the general topic, as illustrated through pseudo-code with examples worked by hand, as appropriate. The student is expected to take notes by fleshing out details on a minimal handout that is provided. After this approximately one-hour classroom lecture, the lab assignment is introduced and the students begin work. 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 *g++* and *make*, and in a Windows PC lab using Microsoft Visual Studio C++ 6.0, with OpenGL/GLUT3.7 as the graphics API. Any platform that supports the latter could be used.

Complete the code provided to implement the Cohen-Sutherland 2D line clipping algorithm.

```
// Returns the 'outcode' for point ( x, y ) with respect to
// the upright clipping rectangle ( L,R,B,T ).
GLubyte CSoutcode( double x, double y,
                   double L, double R, double B, double T );

// Clips a line with endpoints ( x0, y0 ), ( x1, y1 ) to the upright
// clipping rectangle ( L,R,B,T ), setting the flag 'visible' if the
// line is not completely outside the clipping rectangle.
void CS_LineClip2D( double& x0, double& y0, double& x1, double& y1,
                    double L, double R, double B, double T,
                    bool& visible );
```

Figure 1. 2D line clipping assignment. Students are given scaffold code that must be completed.

Students are provided an archive (tar or zip) via the course website, containing the lab assignment document, an executable example of a correct solution, and source 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
`render[.h, cpp]`	- display and idle callbacks
`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
`init.[h, cpp]`	- GLUT, OpenGL, and application setup

Course policy requires students to adhere to this organization. This arrangement allows everyone to refer to a common framework, makes it easy to locate specific code according to its functionality, and better affords collaborative work when such is desired. This code organization is introduced early in the course in conjunction with the discussion of event-driven programming. As a practical matter it also makes grading assignments much easier.

Figure 2. A clipping rectangle is defined by dragging the cursor. Resulting clipped lines.

```
GLubyte CSoutcode( double x, double y,
                   double L, double R, double B, double T )
{
  GLubyte code = 0x0;
  if( x < L ) code = LEFT; // x can't be both < L and > R
  else ;          // RIGHT
  if( y > T ) code |= TOP; // y can't be both > T and < B
  else ;         // BOTTOM
  return ( code );
}
```

Figure 3. Example scaffold code given to the student.

The provided source code is typically created by the instructor from a fully commented, working solution. Key parts of the algorithm are removed, requiring students to 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.

4 Assessment

The pedagogical approach we describe has met our goals. In several offerings of an introductory graphics course using the approach described here, not a single student has been unable to complete the lab exercise on 2D line clipping. 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. Furthermore, after completing the 2D clipping exercise, students are able to extend the algorithm to 3D with minimal additional guidance.

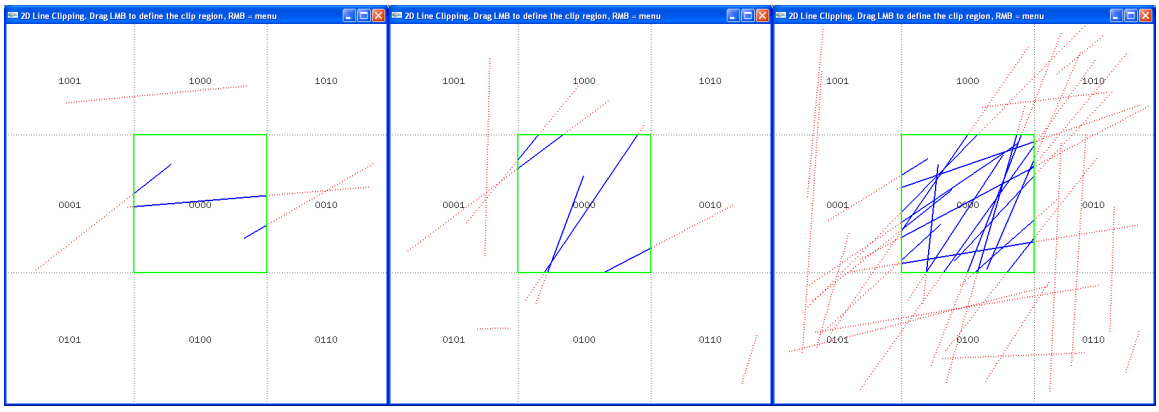


Figure 4. A correct implementation will clip any number of random lines.

Simple line clipping algorithm:
Clip line $(x_a, y_a) \rightarrow (x_b, y_b)$ to the rectangle (l,r,b,t)

1. Try to trivially accept. Test these conditions:

$(x_a < l \;||\; x_a > r)$
$(x_b < l \;||\; x_b > r)$
$(y_a < b \;||\; y_a > t)$
$(y_b < b \;||\; y_b > t)$

If ALL tests FAIL then line is TOTALLY visible i.e., trivial *accept* (done)

If ANY test SUCCEEDS then line may be PARTIALLY visible i.e., might have to clip (goto 2)

2. Try to trivially reject. Test these conditions:

$(x_a < l \;\&\&\; x_b < l)$
$(x_a > r \;\&\&\; x_b > r)$
$(y_a < b \;\&\&\; y_b < b)$
$(y_a > t \;\&\&\; y_b > t)$

If ANY test SUCCEEDS then line is TOTALLY invisible i.e., trivial *reject* (done)

If ALL tests FAIL then line may be PARTIALLY visible or TOTALLY invisible (go to 3)

3. Calculate the intersection of the line with the clipping edges.

Figure 5. Cohen-Sutherland 2D line clipping algorithm in pseudo-code.

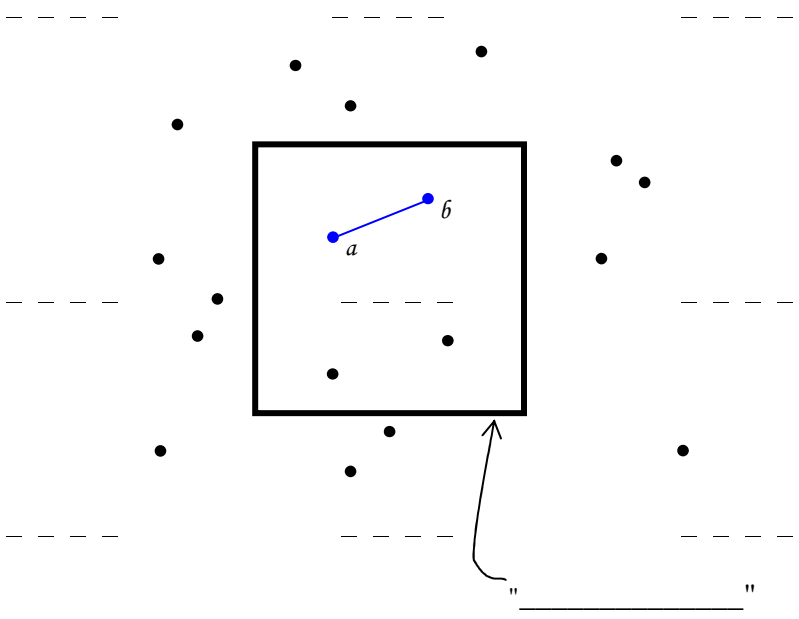
5 Conclusions

Developing graphics course content so as to present theory in a palatable, engaging manner via short lab exercises is no small task. 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 real implementations. The specific lab exercise topic described here, the 2D Cohen-Sutherland Line Clipping Algorithm, is one such example. Other fundamental techniques such as polygon rasterization and clipping 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 3.9, pp. 101-107.

Follow the lecture to label this diagram. Then fill in the table below.



? A line is **completely inside** if both endpoints are inside ? T / F
? A line is **completely outside** if both end endpoints are outside ? T / F

TA = "_____ " TR = "_____ "

Line	code1	code2	code1 | code 2	code1 & code2	Which do we do?		
					TA	TR	Calc. intsx
ab							
ij ①							
ij ②							
ij ③							
ij ④							
cd							
ef							
gh							
kl							

Figure 6. Minimal handout a student must fill in during the lecture.

```
void CS_LineClip2D( double& x1, double& y1, double& x2, double& y2,
                    double L, double R, double B, double T,
                    bool& visible)
{
  // Get the 'outcodes' for the line endpoints
  GLubyte code1 = 0; // Outcode for (x1,y1)
  GLubyte code2 = 0; // Outcode for (x2,y2)
  // code1 == 0000 => (x1,y1) is inside the clipping rectangle
  // code2 == 0000 => (x2,y2) is inside the clipping rectangle

  GLubyte code;    // Indicates an outside endpoint
  visible = false; // Start by assuming we can trivial reject

  // 1. Try to trivially accept. If we can trivially accept,
  //    the loop exits. What should the 'while' test be?
  while ( 0 )
  {
    // 2. Try to trivially reject (if we can, we're done):

    // Can't accept or reject => at least one endpoint is outside
    code = code1 ? code1 : code2;  // code1 == 0 => (x2,y2) outside
                                   // code1 != 0 => (x1,y1) outside

    // 'code' is now the outcode of the outside endpoint. Use it to
    // determine which edge to intersect. Intersection will be (x,y)
    double x,y;

    if( code & LEFT ) // Derivation done in class
    {
      x = L; // intersection with left edge
      y = y1 + ((y2-y1)/(x2-x1))*(x-x1);
    }
    else if( 1 ) { ; } // You must derive and implement RIGHT
    else if( 1 ) { ; } // You must derive and implement TOP
    else if( 1 ) { ; } // You must derive and implement BOTTOM

    if( code == code1 ) // (x1,y1) was outside.
    {
      // Discard this part of the line: (x1,y1) -> (x,y)
      x1 = x;
      y1 = y;
      // Get the outcode for new endpoint:
      code1 = CSoutcode( x1, y1, L, R, B, T );
    }
    else // (x2,y2) was outside. What part of the line gets discarded?
    {    //                      What is the outcode for new endpoint?
    }
  }
  visible = true; // We get here via TRIVIAL ACCEPT if the line was
                  // COMPLETELY VISIBLE to begin with, or it was
                  // PARTIALLY VISIBLE but we clipped it.
}
```

Figure 7. Scaffold code for the Cohen-Sutherland 2D line clipping algorithm.