Recursive hidden-line elimination on an infinite surface

Philip Greenspun and James J. Little*
Massachusetts Institute of Technology
Artificial Intelligence Laboratory
545 Technology Square
Cambridge, Massachusetts 02139 USA

August 19, 1988

Introduction

Algorithms that display a piece of an arbitrary, infinite surface with a vector display or pen plotter are well-known (Figure 1). Iterative algorithms can be time-consuming to implement in Fortran or PL/1. We designed, coded and debugged a substantially more perspicuous implementation in Common Lisp in about two hours on a Symbolics 3670.

Shortly thereafter, Greenspun was teaching a Lisp programming course to a group of Fortran graphics programmers at a company producing computer-aided drafting software. Due to Greenspun's fine pedagogical skills, after eight weeks many students maintained that one could program any algorithm just as easily in C or Fortran as in Lisp. This company had spent over a year trying to debug a multi-thousand line Fortran surface displayer. We decided to write the following paper describing our 70-line Lisp implementation of the same algorithm.

Greenspun was trying to convince these Fortran users of the advantages of the philosophies espoused in *Structure and Interpretation of Computer Programs* by Abelson and Sussman (McGraw Hill: New York 1985). Traditional programs, such as multi-pass compilers, serially convert data from one format to another until the desired final format is achieved. The most important features of Lisp for our algorithm are recursion and dynamic storage allocation. Using these mechanisms, the program can decompose the problem as it recurses, then incrementally construct the solution as it returns. This technique is similar to that employed by simple (and somewhat inefficient) compilers that associate code with each node in a parse tree and then collect that code as the tree is collapsed.

Restrictions on surfaces

Let us restrict ourselves to surfaces defined by $z(x, y)$ so that the surface has a unique elevation for a given point in the x-y plane. In surface topography, for example, this restriction would prevent us from modelling overhangs and caves.

We shall assume that our surface has infinite extent, even if our data represents only a finite portion of the surface. This assumption enables us to worry only about showing one side of the surface. In computer cartography this assumption reflects reality. For example,

*© 1988 Philip Greenspun and James J. Little

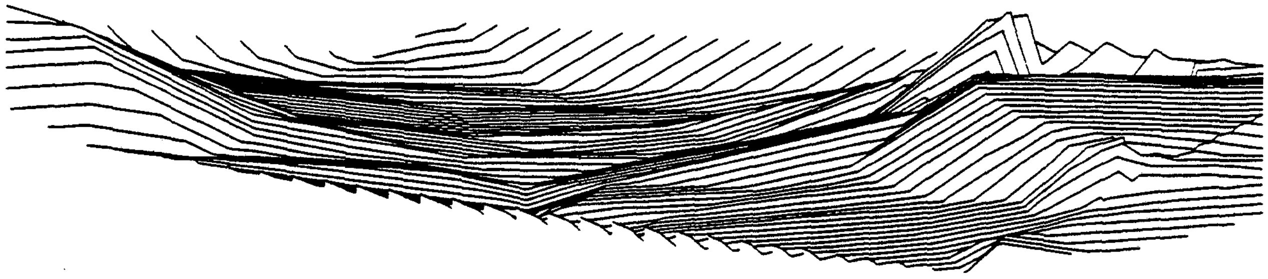

Figure 1: Perspective view of a site in New Hampshire

if the surface is a small oil field in Texas with a 5,000 meter deep hole in the ground, we cannot see the exterior of the hole poking out from underneath our surface if our view point is above the surface. Even if the exterior bottom of the hole does not project (in the direction of the viewing plane normal) against a part of the surface covered by our model, we can assume that its projection reaches the underside of the Earth before reaching the viewing plane.

General procedure

First, we intersect the surface with a series of vertical planes orthogonal to the projection of the viewing direction onto the x-y plane (Figure 2). We then convert each intersection into a list of line segments (if the surface is not piecewise planar we must approximate higher-order curves in a piecewise linear fashion, something we would have to do anyway for most output devices) and call the list a *profile*. We make a list of the profiles by stepping in fixed increments along the viewing direction projection (if profile A is closer to the view point than profile B, then A precedes B in the list).

We accomplish hidden-line elimination by two dimensional clipping against a horizon on the view plane. We display only those line segments (or portions of line segments) whose projections are above a horizon established by projecting and drawing previous profiles. Our initial horizon is a "segment" from $(-\infty, -\infty)$ to $(+\infty, -\infty)$ so that anything in the first profile is guaranteed to be drawn. (The Symbolics implementation of Common Lisp allows the notation "`1e∞`" for positive infinity.)

The core function, `clip`, takes a profile and existing horizon and returns that portion of the profile that was visible and a new horizon. `Outer-loop` starts off the recursion with all the profiles and the initial horizon by calling `outer-loop-1`. `Outer-loop-1` checks to see if there are any profiles left and, if so, calls `clip` to clip the first remaining profile against the horizon. `Outer-loop-1` then draws whatever visible segments `clip` returned and recurses with the unprocessed elements of the profile list (the `cdr`) and the new horizon returned by the call to `clip`.

Recursive clipping

`Clip` takes two arguments. The first argument is a list of profile segments and the second is a list of horizon segments. `Clip` uses the Common Lisp multiple value facility to return two values, a list of segments to draw (that part of the profile that was visible) and a new horizon (also a list of segments).

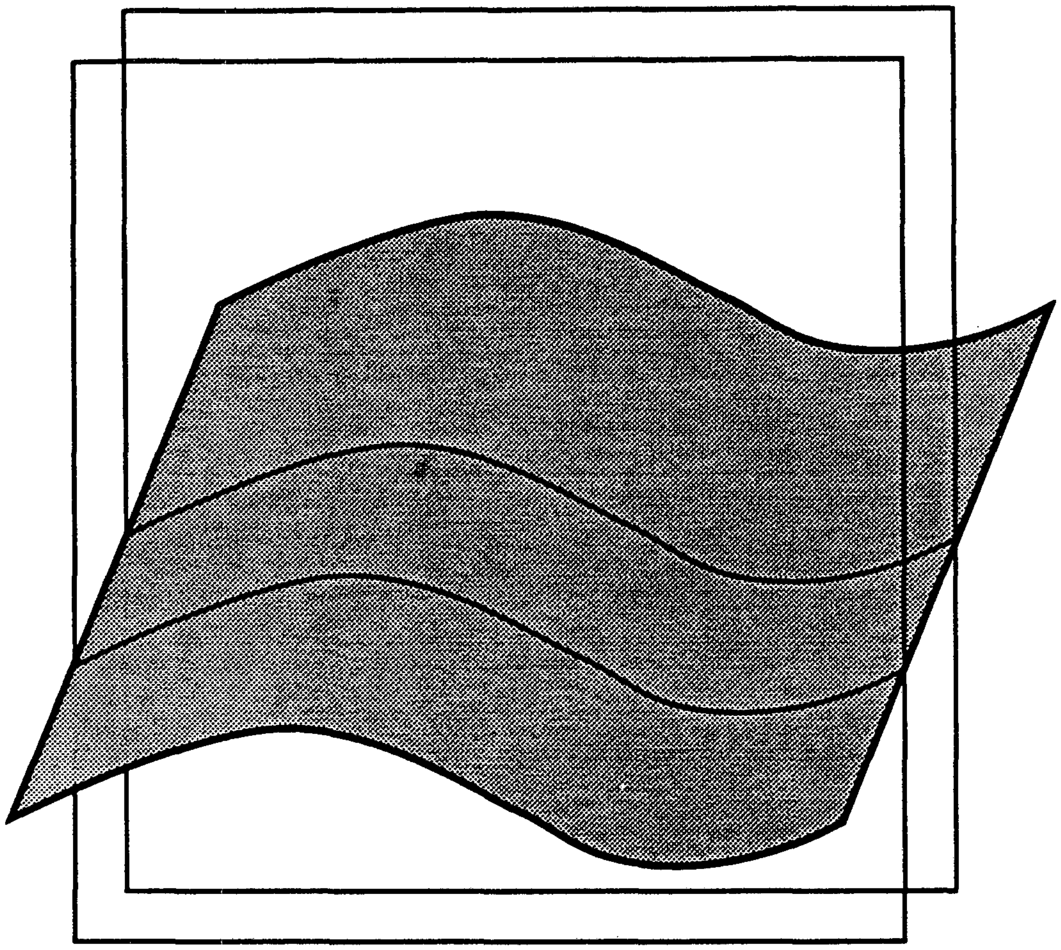

Figure 2: An example surface intersected by two vertical planes

`Clip` assumes that the segments within each list either do not overlap or overlap just at their endpoints. The segments are assumed to be arranged from left to right (in order of increasing x) and it is assumed that the horizon extends at least as far in both directions as the profile.

As in any recursion, we first check for termination conditions. If the list of horizon segments is empty, we signal an error (since the horizon should extend to $+\infty$). If the list of profile segments is empty, we are done and return `nil` (the empty list) as our first value (the list of segments to draw) and the remaining horizon list as the new horizon. These values will be added to by surrounding calls to `clip`.

Another easy case is when the first profile segment is completely to the right of the first horizon segment (Figure 3). In this case we recurse, passing down the complete profile segments list, but stripping off the first horizon segment. The value returned by this call to `clip` cannot simply be the value of the recursive call, however, since that would result in the loss of part of the horizon. The **`segments-to-draw`** value is simply whatever the recursion returned, but the new horizon is a list

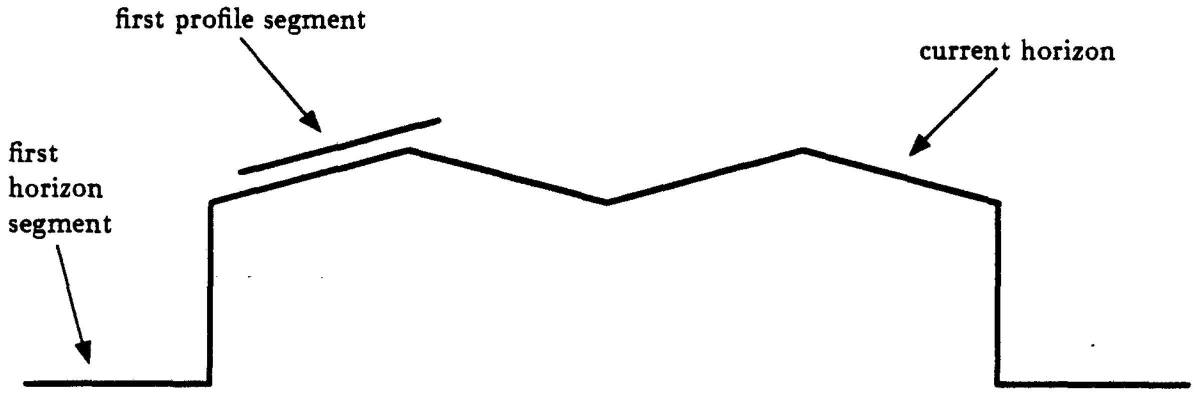

Figure 3: The first profile segment does not overlap the first horizon segment

whose `car` is the horizon segment we stripped off and whose `cdr` contains the horizon segments returned by the recursive call.

Real work

We've now dealt with the easy cases and must consider the case where there is some overlap between the first horizon segment and the first profile segment (Figure 4 illustrates two representative cases). The key idea here is to deal with as small a part of the problem as possible and let the recursion do the rest of the work. `Clip` calls `clip-segments` to compare the two segments until it hits the right edge of either. `Clip-segments` returns four values. The first is a segment to draw (at most one can result from the intersection of a single profile segment and a single horizon segment). The second is a list of new horizon segments lying horizontally between the left edge of the original horizon segment and the leftmost right edge of either segment. This list will contain between one and three elements (one if the profile segment was entirely below the horizon, two if the left edges of both segments were aligned and they intersected somewhere in the middle, and three if the profile segment's left edge was to the right of the horizon segment's and they intersected similarly).

`Clip-segments` returns as its third value that portion of the profile segment that extended beyond the right edge of the horizon segment. The fourth value is that portion of the horizon segment that extended beyond the right edge of the profile segment. At most one of these will be non-`nil` (both will be `nil` if the segments' right edges are aligned, otherwise one segment must extend beyond the other).

Recursive algorithms in Lisp generally completely process one element of a list, strip that off and recurse with the rest of the list. `Clip` is unusual in that it sometimes doesn't completely process the first element, so it has to recurse with the unhandled piece of the first element stuck (consed) onto the rest of the list. `Clip` is also unusual in that it is recursing down two lists at once and returning two values.

So `clip` calls itself recursively, with the first argument (profile segments) being the result of a conditional. If the profile segment processed on this call extended beyond the horizon segment processed, we strip off the entire first profile segment (using `cdr`), but stick on the piece of the first profile segment that we didn't handle (the `remaining-profile-segment` value returned by `clip-segments`). Otherwise, we have completely finished with

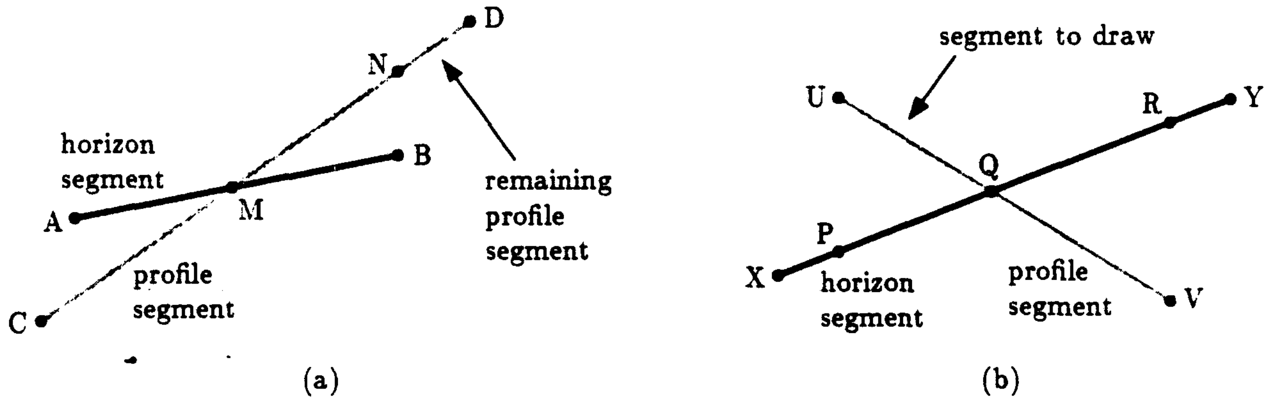

Figure 4: The operation of `clip-segments`

a) AB is the input horizon segment, CD is the input profile segment, MN is the `segment-to-draw`, AM and MN are the `new-horizon-segments`, and ND is the `remaining-profile-segment`. The `remaining-horizon-segment` is `nil`.

b) XY is the input horizon segment, UV is the input profile segment, UQ is the `segment-to-draw`, XP, UQ, and QR are the `new-horizon-segments`, and RY is the `remaining-horizon-segment`.

the first profile segment and can recurse after stripping it off. The horizon segments argument for the recursion is similarly calculated.

After receiving the values returned by the recursive call, we must add to them the pertinent information from the section this call handled. If `clip-segments` found a visible segment, it is consed onto the list of segments to draw returned by the recursive call; otherwise, `clip` just returns the result of the recursion. Because every call to `clip` is guaranteed to make some progress to the right, some horizon is generated for each call. `Clip` calculates the new horizon value by appending the `new-horizon-segments` returned by `clip-segments` and the new horizon returned by the recursive call.

One valid objection to this algorithm is that it tends to produce fragmented horizons. This is easily dealt with by making each subsegment point to the segment from which it was created. Subsegments that were created from the same segment are guaranteed to be colinear and hence can be collapsed by `outer-loop-1`. In one example, when displaying a triangulated digital terrain model surface, the horizon following the clipping and display of 60 profiles was a list of 750 segments. Collapsing segments after each profile was displayed reduced this to 60 and dramatically reduced run time.

A nice feature of this algorithm is its ability to deal with different segment representations. Only the functions `make-segment` and `clip-segments` need to know how the data are structured and both are straightforward. This algorithm can be generalized to finite surfaces by maintaining both top and bottom "horizons." ()

```
(defun outer-loop (profile-list)
  (when profile-list
    (let* ((initial-horizon-segment (make-segment (make-point -1e∞ -1e∞)
                                                  (make-point +1e∞ -1e∞)))
           (initial-horizon (list initial-horizon-segment)))
      (outer-loop-1 profile-list
                    initial-horizon))))

(defun outer-loop-1 (profile-list horizon)
  (when profile-list                          ; This will terminate when we
                                              ;   run out of profiles.
    (multiple-value-bind (segments-to-draw new-horizon)
        ;; CLIP takes the next profile on the list and the existing
        ;; horizon and returns SEGMENTS-TO-DRAW and the new horizon.
        (clip (car profile-list)
              horizon)
      (draw-segments segments-to-draw)        ; Display the segments on some
                                              ;   output device.
      (outer-loop-1 (cdr profile-list)        ; Strip off the profile we've just
                    new-horizon))))           ;   handled

(defun clip (profile-segments horizon-segments)
  (declare (values segments-to-draw new-horizon-segments))
  (cond ((null horizon-segments)
         (error "Ran out of horizon"))
        ((null profile-segments)
          ;; We have run out of profile segments, so we return a NIL to
          ;; CONS onto as SEGMENTS-TO-DRAW and return the remaining
          ;; horizon.
          (values nil horizon-segments))
        ((not (overlap? (car profile-segments) (car horizon-segments)))
          ;; The first profile segment is completely to the right of the
          ;; first horizon segment
          (multiple-value-bind (rest-segments-to-draw
                                rest-new-horizon-segments)
              (clip profile-segments                ; We haven't processed any,
                                                    ;   so we pass through.
                    (cdr horizon-segments))         ; Strip off the first one.
            (values rest-segments-to-draw           ; We have nothing new to draw,
                    (cons (car horizon-segments)   ;  but must return the
                                                    ;  new horizon piece
                          rest-new-horizon-segments))))
(:else
 ;; The left endpoint of the first profile segment falls within
 ;; the first horizon segment.
 (let ((profile-segment (car profile-segments))
       (horizon-segment (car horizon-segments)))
   (multiple-value-bind (segment-to-draw
                         new-horizon-segments
                         remaining-profile-segment
                         remaining-horizon-segment)
       (clip-segments profile-segment horizon-segment)
     ;; We now have all the information we need for the recursive
     ;; call.
     (multiple-value-bind (rest-segments-to-draw
                           rest-new-horizon-segments)
         (clip
           ;; Calculate the PROFILE-SEGMENTS to send down.
           (if remaining-profile-segment ; If some of the profile
                                         ;   segment wasn't handled
               (cons remaining-profile-segment
                     (cdr profile-segments))
             (cdr profile-segments))
           ;; Calculate the HORIZON-SEGMENTS to send down
           (if remaining-horizon-segment ; If some of the horizon
                                         ;   segment wasn't handled
               (cons remaining-horizon-segment
                     (cdr horizon-segments))
             (cdr horizon-segments)))
       (values
         ;; Calculate the SEGMENTS-TO-DRAW to return.
         (if segment-to-draw
             (cons segment-to-draw       ; Our contribution.
                   rest-segments-to-draw) ; Result of recursion.
           rest-segments-to-draw)        ; This call contributes
                                         ;   nothing.
         (append new-horizon-segments    ; We always contribute
                                         ;   something to the
                                         ;   to the horizon.
                 rest-new-horizon-segments)))))))
```

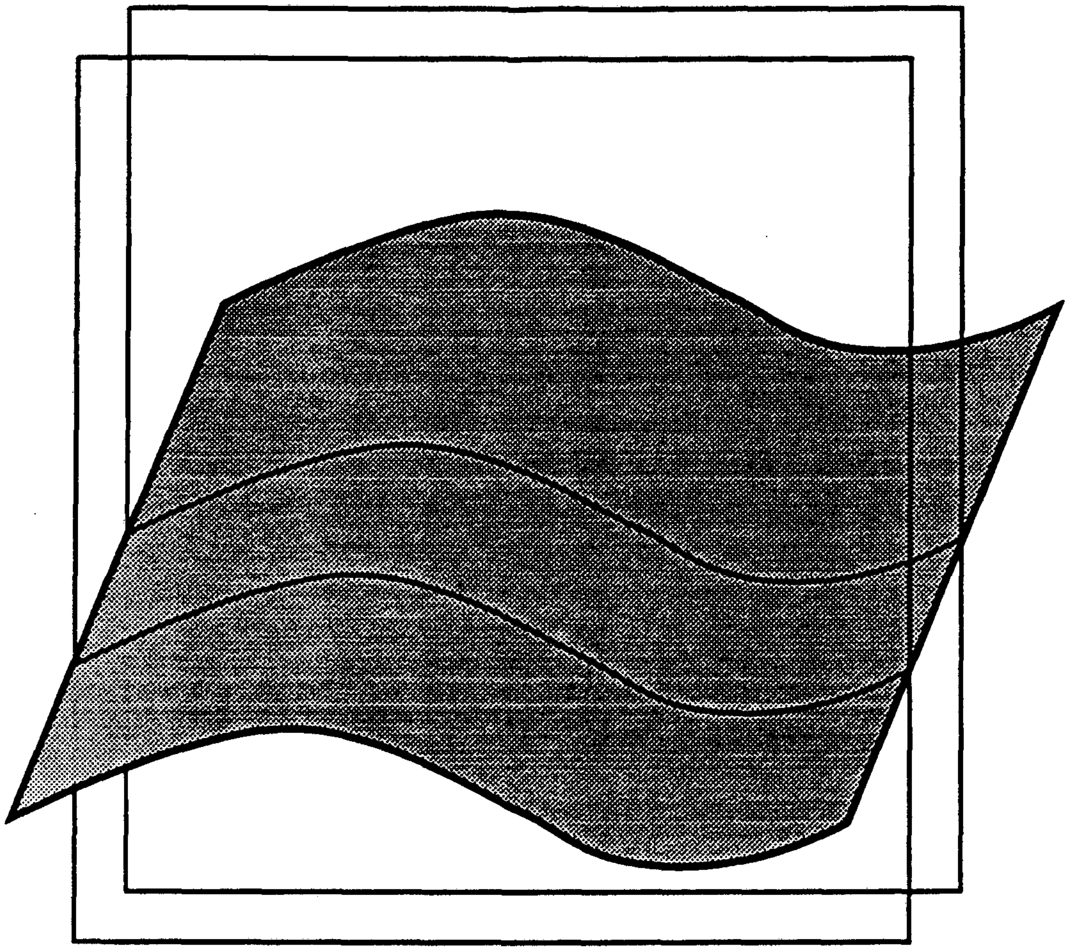

Figure 2