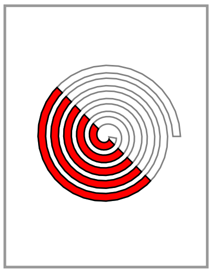
Supplementary Note

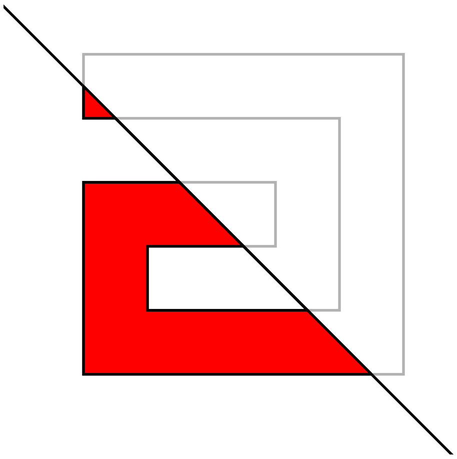
Clipping



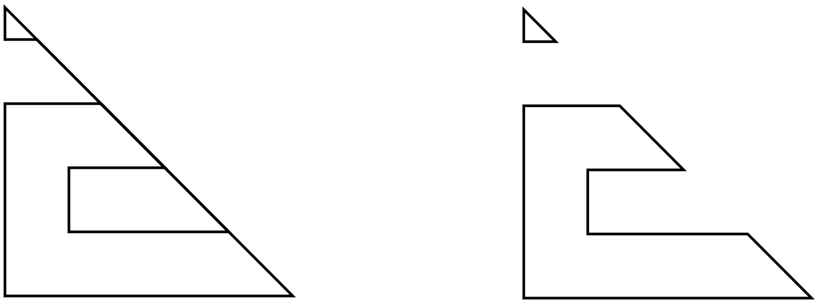
In Chapter 4 the Hodgman-Sutherland clipping algorithm was explained. Given an affine function $f(x)$ and a polygonal path γ, it returns the truncation of γ to the region $f \leq 0$. It has one major fault, however—if the path being clipped does not enclose a convex region and when clipped is really a union of distinct components, this algorithm returns a single path instead of the collection of paths one would expect. We'll see a new version in this note, one that can also be used with minor modification to draw regions on spheres, clipping them to the visible side of the sphere.

1. The problem

This figure explains the problem:



The Hodgman-Sutherland algorithm described in Chapter 4 returns here a single path instead of two—the figure on the left instead of that on the right:



2. A technical tool

In the procedures we'll use, it will be useful to have available the notion of a **stack** in PostScript. This will be an array of two items, an array a of the items on the stack and an integer n indicating the current height of the stack. Relevant procedures associated to this are:

Arguments	Command	Left on stack; side effects
n	`stack`	a stack of potential size n
x s	`spush`	pushes x on the stack s
s	`spop`	pops the top item off s and returns it
s	`slength`	returns the current number of items on s
s	`sempty`	returns whether the size is 0 or not
s	`sfull`	returns whether s is about to overflow
s	`sextend`	doubles the potential size of s
s	`sarray`	returns an array made up of the current stack items

The stack doesn't handle overflow or underflow automatically, but it is possible to detect them by using `sempty` and `sfull`. If overflow is a problem use `sextend` to double the potential size of the stack. All of these implemented purely by PostScript stack manipulations for efficiency. In many programs, a stack will be used primarily as an extensible array. The procedure `sarray` returns the active stack as a fixed array.

3. The solution

Input: the path γ, the (oriented) boundary of a region Γ; and a function f with certain auxiliary properties: (1) given a point, a procedure that returns a value; (2) given two points P and Q with values $f(P) \le 0$, $f(Q) > 0$, a function that returns a unique point of intersection where $f = 0$; (3) a way to sort any finite set of points on $f = 0$. For a line, the first two are straightforward, and if the linear function is $Ax + By + C$ then we sort according to the value of $-Bx + Ay$, which is $[A, B]$ rotated by $90°$. This idea can be used also in 3D where the region is on a sphere, and the truncation takes place with respect to the visible part of the sphere, or across the plane separating visible from invisible. In that case, as things move, say rotating the sphere with the region drawn on it, the calculation will take place dynamically, so it had better be reasonably fast.

We proceed through the edges of the polygonal path γ, which is to say of pairs P, Q of successive points from γ, starting with $P = \gamma_{n-1}$. I assume that γ is oriented positively, so that the region Γ lies to its left. I also assume that γ is simple and closed. We then let Q be in turn γ_0 etc., keeping track of $f(P)$ and $f(Q)$. We maintain three stacks as we go. The first A is of all points on the new paths, the second B classifies those in the first, according to whether they are intersections of in,out; out,in; or neither, which means they are just inside. The third stack C is of indices of the points which are of the first or second kind. The first array is made up of the points themselves,the third of indices in the first (as well as second) array. All points in the third array are **boundary points** lying on the curve $f = 0$ and are calculated as intersections of a segment with the boundary curve. Cases for P, Q:

(1) $f(P) \le 0$, $f(Q) \le 0$: we add Q to our arrays A and B.
(2) $f(P) \le 0$, $f(Q) > 0$: we add the point of intersection R to all three arrays.
(3) $f(P) > 0$, $f(Q) > 0$: we do nothing.
(4) $f(P) > 0$, $f(Q) \le 0$: if R is the point of intersection, then we add R and Q to the first two arrays, R to the third.

These rules differ from the ones applied in the earlier version of Hodgman-Sutherland in the special case where one of P and Q lie in the plane $f = 0$. In the previous algorithm, this was dealt with so as not to add a point twice, but here, on the other hand, the rules will produce three copies of the same point if we get a triple of successive vertices of γ where the first and last points lie in the region $f > 0$ but the middle point P satisfies $f(P) = 0$. This case is treated no differently from the case $f(P) < 0$, and therefore places three copies of P on the curve to be drawn, two of which are tagged as boundary points. Of course in the real world the probability that P will fall exactly on the curve $f = 0$ is infinitesimal, but in theoretical constructions it happens often. One consequence of the rules applied here here is that boundary points always come in pairs, one ingoing and one outgoing.

At the end, after this pass through the edges of the polygon γ, we sort the points in the third array. This sorting is first of all geometrical, compatible with The sorting has to be compatible with orientation—so if P is outgoing and Q the next (ingoing) point in our sorted list, then PQ has the region on the left.

Make up a fourth array of the same length as the first, giving for every boundary point its index in the sorted boundary point array, and $\emptyset$ for every point that's not a boundary point.

We now want to locate the components of the original region cut off by $f \leq 0$. We loop, building one component in each loop. At the beginning of a loop we search through the boundary points to find an outgoing point that hasn't been assigned yet. Then we build the component boundary starting with this point, adding one point at a time. How we choose the next point (i.e. its index) dpends on the type of the current point. If it is outgoing, we choose the next (ingoing) point in the boundary list; if it is ingoing we just choose the next point in the path; and similarly if it is an interior point. We stop when the next point is our starting point, in which case we have a component path. As we add points we keep count, and when all point shave been put on some componnet we stop.

When we add a point to a component, we mark it as used.

What we return finally is an array of polygons, the oriented boundaries γ_i of regions Γ_i.

4. Other applications

The same technique can be used to draw regions on a sphere, for example countries on a map. In this case, we first calculate the plane separating the visible region of the sphere from the rest, and then truncate when a segment of the path crosses this. As for sorting, we pick one outgoing point, and the sort by angular distance (clockwise, as seen by the eye) away from this. This angular distance can be calculated by means of dot products with a plane vector to this point and this vector rotated $90°$. Otherwise the whole algorithm looks pretty much the same. If we use the path we get as a 2D clipping path, we can even produce good shading.

What surfaces will this work on? Presumably convex ones for which we can compute the horizon. It remains difficult to find something that works for others.

5. Code

The relevant code for this new version of Hodgman-Sutherland is in the file `hs.inc`, which contains the single procedure `hs`. This has the path and the line as arguments, and returns the array of component boundary paths as polygons.

The stack procedures are contained in the file `stack.inc`. In order to access the code defined there after running this file, place `stackdict begin` in your program as well, and (of course) an `end` when you don't need stacks anymore.