Subdivision Curve Interrogations

Subdivision API

The API for subdivision curves consists of the following methods:-

- Is the curve simple ?

This enquiry returns true or false, and reports whether the approximation which the next method calls is an adequate approximation within a required tolerance.

- Approximation

This method returns something simpler (typically a straight line segment between the ends of the curve).

- Hull

This returns a convex set which contains the curve. The code of interrogations will be more efficient if the hull is tight, not containing more non-curve than necessary, but this has to be traded off against the cost of calculating the hull and using it.

- Halves

This returns two curves, which together make up the original curve.

- ...

Other methods can be added, which may make some interrogations more efficient, but they are speed-ups, not essentials, and will be introduced at the appropriate time.

Examples

Drawing a Curve

The essence of drawing a curve is to issue to the drawing device a polygon - a sequence of straight line segments - whose Hausdorff distance[1] from the curve is less than some agreed tolerance, typically half a pixel on the screen.

[1] The Hausdorff distance $h(S_1, S_2)$ between two point sets, S_1 and S_2 is defined by

$$h = max(max_{P \in S_1}(min_{Q \in S_2}|P - Q|), max_{Q \in S_2}(min_{P \in S_1}|P - Q|)\,)$$

The subdivision approach to this is

```
Draw(curve:a)
curve:b, c
begin   if      simple(a)
        then    draw(approximation(a))
        else    b, c := halves(a)
                draw(b)
                draw(c)
        endif
end
```

For this to be correct (i.e. always working and always giving a result, correct as per the definition of drawing above) we require

- that the recursive calls should themselves be correct,
- that the drawing of the approximating segment should be correct,
- that the process of halving should produce sets whose union is the original point-set,
- that the test for simplicity should only return true if the Hausdorff distance between the approximation and the curve is less than the required tolerance.
- and that eventually the process of halving should result in pieces all of which pass the test for simplicity.

Given those axioms, proving the correctness of the pseudo-code is a useful exercise.

An example of this algorithm in action is the Douglas-Peucker simplifier of complicated curves, widely used in GIS systems to produce sparse curves (for small-scale maps) from dense data. This takes a given curve and finds the point furthest from the chord between the ends. If this point is within the appropriate tolerance from the chord, the chord is drawn. If not, the curve is split into two at that point, and the same procedure applied to each part.

There are efficiency issues.

- How few line segments are issued ? Is this minimal ?

No, it is not minimal but is never too bad. There is a counterexample where the number of segments issued is just under twice the minimum number.

- How expensive is the computation ?

In the Douglas-Peucker context, each of the dense points has to be scanned at each level in the recursion, and so the time taken is proportional to the number of original points times the logarithm to base 2 of the number of segments issued.

Curve-Plane Intersection

The essence of curve-plane intersection is to return all those points of the curve which also lie on the plane. This is untenable in a floating point computation context, and so in practice we have to introduce a tolerance again.

The raw algorithm has exactly the same shape as the drawing algorithm. It takes the form of a method returning a set of points

```
planecut(curve:a)
curve:b, c
begin    if      simple(a)
         then    planecut(approximation(a))
         else    b, c := halves(a)
                 Union(planecut(b),
                       planecut(c))
         endif
end
```

Correctness now places an additional axiom that if the two halves of a curve are connected at their meet, their approximations should be also.

The raw algorithm can be made significantly more efficient by adding a level of culling, whereby pieces that are nowhere near the plane are not subdivided down to the tolerance level.

```
planecut(curve:a)
curve:b, c
begin   if      hull(a) cuts the plane
        then    if      simple(a)
                then    planecut(approximation(a))
                else    b, c := halves(a)
                        Union(planecut(b),
                              planecut(c))
                endif
        else    emptyset
        endif
end
```

The proof that this is as correct as the previous version requires the axiom that

- no point of a lies outside hull(a).

This is effectively the definition of the hull.

With this approach we have the opportunity of handling degeneracy correctly, when part of the curve lies identically in (or within tolerance of) the plane. If the planecut(approximation) code detects that both ends of the approximation are within tolerance of the plane, then it can return, not a point, but the entire approximation. If the code for taking the union of two pointsets can accept sets with either discrete points or connected pointsets, the structure for computing degenerate intersections is all available.

Closest Approach

The subdivision approach is as applicable to this as to the previous examples of interrogations, but it requires a slightly deeper understanding of what is going on during the recursive calls so glibly made above.

What a recursive call is doing is placing the current operation instance in a **stack**[2] while the new instance is being computed. We can achieve exactly the same effect explicitly, by having an explicit stack of problems to solve.

The equivalent iterative algorithm for drawing is

```
Draw(curve:a)
curve:b, c
stack of curve:S
begin   push a into S

        until S is empty
        do      pop a from S
                if      simple(a)
                then    draw(approximation(a))
                else    b, c := halves(a)
                        push c into S
                        push b into S
                endif
        enddo
end
```

For closest approach we have to recognise that what we are putting into the stack are not bits of curve, but **sub-problems**. In the case of drawing and plane cut, the sub-problems were characterised adequately by curves: in the case of closest approach a subproblem is characterised by a pair of curve segments.

[2] A stack is a collection of objects that you can put objects into and take objects out of, with the property that the object you get out is always the last one you put in. It has four methods

- empty the stack so there is nothing in it
- tell me whether it is empty
- **push** an object into the stack
- **pop** the most recently push object out again, returning it.

The previous examples had to return all the objects which satisfied the problem's conditions. The new concept in closest approach is that we are **selecting** a particular point pair from among a large set of pairs by comparison.

```
Closest approach(curve:a,b)
begin   push (a, b) into S
        best so far := a very big distance
        until S is empty
        do      pop (a,b) from S
                if      a and b both simple
                then    find actual distance between approximations
                        if dist less than best so far
                        then update best so far
                        endif
                else    make two sub-problems (by splitting a or b)
                        push first into S
                        push second into S
                endif
        enddo
        return best so far
end
```

We can still do useful culling by checking before attempting to solve a subproblem whether it can possibly provide a nearer pair than the best so far.

```
        until S is empty
        do      pop (a,b) from S
                if min dist between hulls of a and b is less than best so far
                then    if      a and b both simple
                        then    find actual distance between approximations
                                if dist less than best so far
                                then update best so far
                                endif
                        else    make two sub-problems (by splitting a or b)
                                push first into S
                                push second into S
                        endif
                endif
        enddo
end
```

The performance can be still further improved by replacing the stack with a **priority queue**[3], organised so that it always returns first, not the most recently added entry, but the entry with the smallest maximum distance between hulls.

[3] This gives a heuristic tree search, which was Advanced Artificial Intelligence in the 1960s

Finally, if we are prepared to assume that the original curves are C1, and we handle the case where the nearest approach involves one end of one or both curves separately, we can use **tangent hulls** of the curve segments to determine that a particular pair cannot contain the closest approach. The tangent hull of a C1 curve segment is a convex set containing all **unit** vectors tangent to the segment. We can also use these hulls to determine whether a pair has at most one common perpendicular. If that is the case, then it will possibly be worth switching to Newton iteration to find the exact answer directly. If there is only one root Newton can be expected to converge, and in the event of failure, all we have to do is fall back on subdivision. There is no need for a great deal of code to handle failure.

Hulls

Why hulls ? Because it is relatively easy to detect whether convex sets intersect. For some specific convex sets it is very easy indeed.

The early papers on subdivision used to refer almost exclusively to the **convex hull**, which is the set of all positive linear combinations of the points of a set. In 2D this is given by the interior of a rubber band stretched around the points and allowed to shrink. In 3D a rather more useful characterisation is that of the intersection of all half-spaces which contain all of the points in the set.

Unfortunately convex hulls can be rather complicated geometrically. The convex hull of a surface of positive Gaussian curvature can contain parts of the surface itself in its boundary, and so the objective of having something simpler to do tests on is not achieved.

What we can do is to take the intersection of only those half-spaces which have specific orientations, and the simplest hull (almost universally used in subdivision code) is the **bounding box** which is the intersection of all axially aligned half-spaces containing all of the points in the set.

This is much much simpler than the convex hull both to represent (6 numbers is enough) and to check for overlap of (at most 6 comparisons). The disadvantage is that it contains points which the convex hull does not, so that the culling in the plane-cut algorithm is not so effective, but the balance of effect on performance is definitely in favour.

It is possible to use more directions to reduce the excess volume over the convex hull, but this does not appear to give much advantage, if any.

S-bounds

Another way in which laxity of the hull can be used to give simpler computation is by taking the hull, not of the pointset of the curve itself, but of its control polygon. This can be applied whenever the basis is entirely non-negative.

If this is done, it is typical for the hulls of the halves to be closer fits, and taking the hull of these two hulls often gives a set which is tighter than the original.

Pre-subdivision

When subdivision methods were first explored, the typical workstation computer (personal computers were not yet heard of) had at most a megabyte of memory. It was therefore necessary to keep memory usage to a minimum.

Nowadays we have memories a thousand times larger, and we can exploit this to improve the speed of interrogation by retaining any subdivisions made. In fact we do not need to copy the actual data – just keep references to say which part of an original curve a particular piece is – but we can keep bounding boxes for every piece to save recomputing them.

It would then appear that subdivision should be the fastest interrogation algorithm almost all of the time.

Summary

Subdivision algorithms work by the divide-and-conquer strategy, solving directly any problems which are simple enough, and creating simpler sub-problems for those which are not. These sub-problems can be kept on a stack or in a priority queue.

This approach is capable of supporting careful reasoning about correctness and termination of algorithms, and therefore supports more reliable code. Actually building 'proofs of correctness' for the interrogations helps define the axioms on which those proofs are built, and which are themselves specifications for the routines which access specific curves.

It is also potentially capable of being extremely fast, by exploiting the large memories which are now widely available.