Subdivision Surface Interrogations

Much of the material on curve interrogations was cast in terms of point sets, and all of this applies equally to surfaces. It will also be found that the ideas form interrogation of parametric surfaces have relevance.

However, there are a few new ideas in the subdivision surface interrogation context, and these make it worth spending one session on this topic.

The major headings are:-

- API
- rendering
- plane sections
- nearest point
- intersections
- offset surfaces
- implementation issues.

API

The set of enquiries which can be made of a surface object look exactly the same as those for a curve object, although there are some differences of interpretation.

- Are you simple ?
- What is your approximation ?
- What is your hull ?
- What are your halves ?

with possibly some additional enquiries to make certain interrogations more efficient.

Are you simple/What is your approximation

The approximation here is now a plane[1] facet approximating the surface piece. The *'are you simple ?'* question returns true only if the approximation is close enough, *'close enough'* again meaning that the Hausdorff distance between the facet and its piece of surface should be less than some tolerance. In some implementations the additional requirement could be imposed that the projection of the facet on to its nominal plane should be convex.

[1] Maybe a facet approximating planarity is more practical.

Hulls

The definition of hull for a piece of surface is exactly the same as for the curve case: a convex set containing the point set. The choice between the three main contenders, the convex hull (the tightest), the bounding box (the cheapest) and the quantised hull (in between) is a system-level decision and will apply to both curve and surface handling within a system.

Halves

The halves, S_1 and S_2, of a surface S, are two surfaces which together make up S.

Technically their union is S and their intersection is $\emptyset$, but in practice we let them share an edge curve, so that we can treat them consistently as closed sets.

Again, it is reasonable to ask that if S is connected, S_1 and S_2 should each be connected.

There is the definite requirement for termination of the interrogation algorithms, that halving should give rise to simple pieces in a finite (preferably small, for efficient interrogation) number of steps.

For curves it was easily assumed that the approximations of the subcurves should meet in a common point, and also easily achieved, by implementing the scheme in such a way that the approximation of a curve piece was always the chord between its end-points.

For surfaces it is less easy to define a good condition, and we shall see that definite difficulties emerge from this. However, the connectedness condition mentioned above leads naturally to the condition that the approximations of the pieces should contain the approximation of the common edge.

Two new questions arise in the bivariate context.

- Why divide a surface piece into two and not four ?

The standard response to this is that interrogations are essentially pruned tree-searches, and these can be more efficient if the tree is deep rather than wide.

Clearly it would not be efficient to implement an intersection of two surfaces by splitting each into a very large number of pieces. However, I am not aware of any published work reporting actual experiments on this.

- If we are splitting into two, which way do we split ?

The simple plausible rule is to split whichever way reduces the maximum extent most.

Defining a good split for a complex surface which includes extraordinary points (so that there is no obvious choice of iso-u versus iso-v) is an even bigger question. This will be returned to under implementation issues below.

rendering

The essence of this problem is sending to the graphics card a set of polygonal facets whose union is close enough to S. The tolerance can reasonably be half a pixel, which is a relatively coarse requirement.

The obvious algorithm, by analogy with that for drawing a subdivision curve, is

```
Render(Surface: S)
Surface: S1, S2
Begin If      S is simple
        Then Render(Approximation of S)
        Else  S1, S2 := halves of S
              Render(S1)
              Render(S2)
        Endif
End
```

This does work, but it has a big problem with **glitches**. A glitch occurs when the depth of subdivision varies between parts of the surface. Suppose that a piece has been subdivided in u and that one half is accurate enough, while the other half is subdivided in v and both sub-halves are then found accurate enough. The two cuts form a T-shape, and it is likely that there will be a narrow triangular sliver of a gap between the pieces sent to the graphics card. This is a simple example, but is fairly typical.

The solution to this is not to send the approximations direct to the graphics card, but to salt them away. Then the coarser approximation can be modified when the extra vertex is found on the other side of the resolution mismatch. The detail of this fix-up can either replace the original coarse facet by splitting it up, or else can just issue a not-quite plane facet, which most graphics cards are perfectly happy to accept.

Sensible indexing can make the finding of the facets to be modified a reasonably efficient process, but that is a computer science issue, not a mathematics one.

plane sections

The essence of this problem is finding the set of points which lie both in S and in the plane. It is strongly analogous to the finding of points in which a curve cuts a plane. The new feature which rendering does not have, is again the use of the hull at high levels in the tree to avoid subdivision of parts which cannot contribute.

```
pointset: Plnint(Surface: S)
Surface: S1, S2
Begin If    hull of S intersects the plane
      Then If    S is simple
           Then Plnint(approximation of S)
           Else  S1, S2 := halves of S
                 Union ( Plnint(S1), Plnint(S2))
           Endif
      Else  ∅
      Endif
End
```

This is again inadequate for any purposes except graphics, because it does not generate the curve segments in curve sequence, which is required for many purposes.

It also has a problem closely related to the rendering glitches, in that a cut across the glitch will have the two parts not meeting at what should be a common point.

The extra complication for both of these can be hidden inside the Union operator, and requires a fair amount of programming and indexing of computed pieces from the cuts between the pieces of surface.

Careful maintenance of orientation can also help to make this process not too much of an overhead.

Because plane sections are often required, and in large numbers (think of a contour map) with the same plane orientation, it is worth thinking of using a very special hull, a quantised hull, which has two entries; plus and minus the plane normal. This is cheap to evaluate and is probably worth cacheing.

nearest point

This is closely related to the Nearest Approach algorithm for two curves, and, like that algorithm, benefits from being implemented in terms of explicit iteration, with a priority queue instead of a stack.

It is interesting to compare this with the nearest point algorithm for parametric surfaces. The subdivision algorithm really is going for the nearest point directly, rather than via the foot-point problem, and, as a result, it does not need to consider derivatives at all.

On the other hand, it is not able to benefit from using coherence between nearby target points.

surface-surface intersections

This picks up the other concept introduced under closest approach of two curves, that what goes on to the stack is not a piece of geometry, but a sub-problem (which may be characterised by a piece of geometry).

It shares the glitch problem and the need to reconnect the resulting curve pieces into curve sequence with the plane section problem, and essentially the same solution applies. It is therefore important to maintain orientation, always keeping the pieces of the same surface first in the recursive calls.

In the literature it is reported that it is worth while using **normal hulls** to check whether it is known that a particular subproblem can only have a single, open piece of curve. If that is the case, then it may be possible to switch early to an alternative (parametric) algorithm long before the surfaces are simple enough for their intersection to be just a piece of straight line.

The new question here is *'which surface to split ?'*. If both surface are still complex, the answer is simple: *'whichever has the larger hull.'*. If one is simple, there is probably little point in splitting it, and indeed, it is probably worth-while to have special-case code which is entered as soon as it is known that one surface is simple.

normal hulls

A normal hull is a convex set containing all possible surface normals of a piece of surface. There are two obvious variants, one being the hull of normals at whatever magnitude the code produces, and the other being the hull of unit surface normals. The latter can reasonably be expected to give somewhat tighter results for the bounds on direction of tangent vector which is what we require in the surface intersection context.

offset surfaces

An offset surface is one which is defined with reference to its **base surface** by adding a displacement to each point. The appropriate displacement usually varies with the surface normal vector.

These arise in three frequent contexts

- where one side of a skin has its shape defined, but the other side has supporting structure which needs to be determined.
- where we need to determine tool-centre paths for machining of dies. The tool centre lies on an offset surface.
- where we seek the centre line for a rolling-ball blend. This centre line lies on two surfaces offset from the surfaces between which the blend is to span.

As in the parametric case, the offset surface can either be set up by explicit approximation, or by computation on the fly by making reference back to the base surface.

It is less obvious how this can be done for subdivision surfaces.

The offset is defined in terms of an **offset form**. For skin thickness and blend computation this is just a sphere of appropriate radius, but for machining with efficient cutters it is definitely a function from surface normal to a displacement. We can clearly take the hull of all possible such displacements.

A simple, correct, algorithm is then to produce the offset surface object by inheriting the 'simple' and 'halves' enquiries from the base surface, by modifying the 'approximation' property by adding the displacements at the vertices, and by modifying the 'hull' property by merely taking the Minkowski sum of the base hull with the inverse[2] of the offset hull.

Theorem

Given: A surface S of hull H and an offset F of hull G

To Prove: that the offset surface lies inside $H \oplus G*$

Proof: by the definition of the Minkowski sum of two pointsets.

$$A \oplus B \equiv \{a + b | a \in A,\ b \in B\}$$

The offset surface consists of points each of which is a base point, plus an offset displacement. The base point lies in H and the offset displacement lies in $G*$. The sum therefore lies in their Minkowski sum.

∎

This algorithm is correct: it will ultimately terminate giving the correct result. However, it is extremely inefficient, because for tiny pieces of surface the hull of the offset surface is no smaller than the hull of the offset form. A typical run will therefore result in the surface being subdivided everywhere down to the 'simple' level, with no culling of irrelevant pieces.

[2] the inverse here is just the reflection through the origin of the shape of the cutter

It can be rescued, by using as the local hull for the offset surface the Minkowski sum of the base surface and the part of the offset form which is relevant to surface normals within the normal hull of the base surface.

implementation issues

A few years ago (between ten and twenty, but I do not have exact dates) there was great interest in subdivision interrogation because of the promised robustness.

In fact there are still problems which leak in through the use of tolerances and through the sharing of a curve by both halves of a surface, and it was also found that the performance was not as good as that of parametric techniques, which had been developed (like the internal combustion engine) to a good state of practical reliability. Interest therefore died down, leaving parametric techniques as the standard methods.

I believe that the balance of performance is about to change very dramatically, and that subdivision interrogations will begin to complete very strongly indeed.

The reason for this is the rapid growth in the sizes of real memories on everyday workstations over the last five or so years. We are approaching a threshold where profligate use of memory can make subdivision very fast indeed.

Here are three archetypical implementation designs for a subdivision-based system. The first is the simplest text-book version. The second identifies a few enhancements which cna be made and probably have been, and the third is the way that a 2002 system might be built.

Text-book system for tensor-product B-spline surfaces

- The master data, held in memory between interrogations is the original polyhedron. This might typically have a couple of hundred control points, and would occupy something of the order of 20K to 50K bytes. No problem on the machines of the mid 1980's with a Megabyte or two of memory.
- Refinement is by knot insertion whenever required during an interrogation.
- The hulls used are just the bounding boxes of the relevant control points for the piece of surface being considered.

Enhanced version, still for tensor-product B-splines

- The master data is still the original polyhedron.
- Refinement is by multiple knot insertion, so that the edges of each piece have Bezier edge conditions.
- S-bounds are used to reduce the hulls after subdivision has taken place.
- There may be some cacheing of intermediate data during, for example, the calculation of a nest of plane sections.

The performance gains are primarily from the Bezier edge conditions, which cause the hulls to be much tighter. The cacheing causes a significant increase in code complexity.

Version to exploit very large memories - interpolating refinement

- the master data held in memory between interrogations is a fully refined polyhedron. This may contain of the order of a million vertices and facets, and occupy a hundred Mbyte or so.
- the external definition is still the original polyhedron, so we do not face reading 100Mbytes from disc on loading.
- all the bounding boxes can be built bottom up, so there is no need for S-bounds.
- the same process determines the allocation of fine facets to pieces, so the partitioning actually fits bounding boxes rather efficiently. This operation takes of the order of $n\,lg(n)$ operations, which is still totally acceptable up to sizes of a few million.
- the vertices, facets, normals, hulls and a few other bits of information are all held in real memory, so that the actual interrogation processes consist only of a few tests, and can be extremely fast.

This style of implementation is just becoming feasible, because memories of the order of 1Gbyte are just now becoming standard on large PCs and on workstations. We may serious expect the 4Gbyte computer to be standard within a very few years, at the same price as a 256Mbyte one today.