Chapter 5.
Spatial Data Manipulation

Geographic databases — particularly those supporting three-dimensional data — provide the means to visualize and analyze the world around us in ways that until recently were only dreamed of. Yet the vast volumes of data inherent to these databases can slow processing times down to a crawl. And although performance of computing machines continues to improve, these improvements consistently fail to meet the growing need to process even more data more rapidly. Recognizing this problem, Guenther and Buchmann [GB90] suggest that algorithms using a data filtering approach could greatly improve the performance of many spatial operations.

5.1. Benefits of spatial filtering

Hierarchical data structures have been used in the past to reduce computation for a number of applications. These include pyramid decompositions for image understanding [TP75], geometry hierarchies for computing visible surfaces [Cla76], BSP trees for shadow generation [CF89], and triangular decompositions

of the sphere for geographic indexing [Goo89, Fek90]. Samet provides an excellent overview of hierarchical data structures in [Sam90a] and their applications in [Sam90b].

My approach is to take advantage of the adaptive hierarchical triangulations' tree structure using levels of detail as data filters. Because each triangle in a coarser level of detail represents a generalization of its children, the tree may be effectively pruned to provide detail where it's needed and generalizations where it's not. Parent triangles also provide enough information about their children to provide approximate answers for analytical queries.

To demonstrate the ability of this structure to support both visualization and analysis, I developed manipulation techniques for three specific applications: zoom, multi-resolution display, and line-of-sight calculation. The first two — zoom and multi-resolution display — aid rapid visualization of the data. The third — line-of-sight — represents a more analytical application. These are just a few examples of how this structure can be used to improve performance of spatial operations. All three are described in the remainder of this chapter.

5.2. Zoom

Zooming in on a model allows one to go from a generalized overview of an object to a detailed look at some small part. Only the salient features need appear in the generalized model, for smaller details are insignificant at that point. As the viewer zooms in, more and more details must become apparent. For example, one could start with a map of New York, zoom in on Long Island, zoom in further for a view of Stony Brook, and end with a map of the campus.

Adaptive hierarchical triangulation, described in chapter 3.2, is ideal for this type of operation. The hierarchy contains fixed levels of detail in which only the least significant details are eliminated from the coarser levels. Hence major features do not suddenly appear or disappear in the transition from one level of detail to another.

I achieve a smooth transition from one level of detail to another with a method that is similar to that used in Evans and Sutherland flight simulators [CMR90]. Each triangulation essentially has two states: a finished state in which all vertices are in their correct positions, and a reduced state where those vertices are projected onto the parent triangles.Because no child edge crosses a parent edge in this reduced state, it is visually equivalent to the parent triangulation. Hence the transition is perfectly smooth, with no sudden changes from level to level. An example of the types of views this might produce is shown in figure 5.1. The algorithm is presented in figure 5.2.

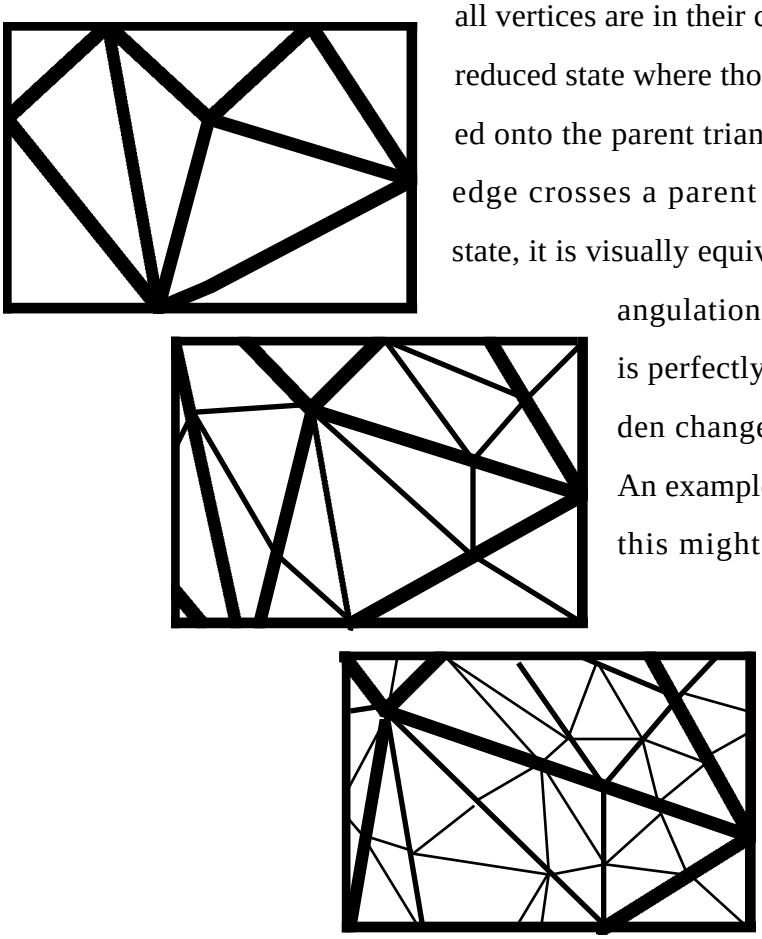

Figure 5.1. Example of zoom

I select levels of detail for this algorithm using a

```
procedure zoom (i, zooming_in, steps );
Input :     Triangulation hierarchy T, level of detail i., a zooming_in flag
            indicating which way the zoom is moving, and a number of iteration
            steps for the transition.
Output :    Animation showing the smooth transition from level i to level i±1. begin
    display n_i triangles [T_{i,1} .. T_{i,n_i} ] at level i ;
    if zooming_in then begin    (*show surface at a finer level of detail*)
        get all n_{i+1} vertices at level i+1 ;
        project those vertices onto triangles [T_{i,1} .. T_{i,n_i} ] ;
        display the n_{i+1} child triangles [T_{i+1,1} .. T_{i+1,n_{i+1}} ] from level i+1
            with their vertices projected to the parent triangles;
        calculate an interpolation vector for each vertex;
        for each of the interpolation steps do
            shift each vertex to the next position along its vector;
    end;
    else begin                  (*show surface at a coarser level of detail*)
        project n_i vertices at level i onto triangles [T_{i-1,1} .. T_{i-1,n_{i-1}} ] at level i-1 ;
        calculate a vector for each of those n_i vertices, and divide it into steps
        for each of the interpolation steps do
            shift each vertex to the next position along its vector;
        display n_{i-1} parent triangles [T_{i-1,1} .. T_{i-1,n_{i-1}} ] at level i-1 ;
    end;
end;
```

Figure 5.2. Algorithm for zoom

simple heuristic based on viewing distance. As noted by the computer graphics community, the degree of accuracy required in a rendered image is proportional to the projected size of a pixel [Bar86, HD91]. Likewise, errors in the model that translate to less than a pixel in size are negligible. I therefore take a similar approach, selecting a level of detail where the error is always less than the width of a pixel projected onto the surface. For example, a 1˚x1˚ cell is approximately 120 kilometers on each side. Mapped to a 1024x1024 display, this level of detail

should have no error greater than 117 meters. A 15'x15' map in the same space must have less than 29 meters error. And so on.

5.3. Multi-resolution views

As noted by Devarajan and McArthur [DM93], an ideal terrain model for real-time simulation is

- a tree, where all possible prunings are valid terrain models,
- continuous over the entire surface for all prunings, and
- single-valued (i.e. $z = f(x,y)$).

Multi-resolution display, like zoom, allows the use of fewer, more generalized triangles to represent areas that are further from the camera or less important to the viewer. Unlike zoom, multi-resolution views combine different levels of detail in a single seamless model. Multi-resolution display is useful for rendering perspective views of a scene where the foreground shows greater detail than the background. This is similar to a strategy currently employed in simulators using quadtrees [CMR90]. Another application of multi-resolution display is the bull's-eye view [TSDB88] where a roving window shows a portion of the scene in crisp detail against a generalized background. In both applications, using multiple levels of detail improves performance.

Multi-resolution displays essentially show temporary models that combine different levels of detail corresponding to different levels of importance or distance. I begin by partitioning the model into triangular patches corresponding to some coarse level of detail. The tree-like structure of my triangulation hierarchy allows us to then use each patch to represent a different level of detail as

required. This partitioning may also be repeated recursively within the triangular patches. Adjoining patches from different levels of detail are seamed together by forcing high-resolution triangle vertices along the shared edge to lie on that edge. Once the initial model is built, shifting the focus is achieved by raising or lowering the level of detail in triangles along the focal boundaries. Because most triangles in the model are unaffected by such changes, this may be done relatively quickly.

My algorithm for producing an initial bull's-eye view model is given in figure 5.3. Levels of detail for the focal area (*n*) and surrounding areas (*m*) are selected by the user. The polygon defining the focal area is represented by *P*. The

```
procedure multi_res (n, m, P );
Input :     Triangulation hierarchy T,  level of detail range n  to m.
            (from finest to coarsest), and polygon P  outlining area to contain
            finest level of detail.
Output :    Multi-resolution surface model T.
begin
    initialize view model to empty set of triangles: T  = [] ;
    for each level of detail i  from n  down to m  do begin
        find the set T'  of triangles at level i-1  crossing or inside P ;
        create polygon Q  from the perimeter of set T'  ;
        find the set of triangles T" at level i  that are children of the triangles in T'
            but  outside boundary T ;
        project all boundary vertices of T'  onto Q ;
        add triangles of T"  to view model T  ;
        P  := Q ;
    end;
    complete the view model T  with triangles from level m  that are outside P ;
end;
```

Figure 5.3. Algorithm for producing multi-resolution bull's-eye model

algorithm for finding all triangles inside or crossing this polygon is given in the next chapter. Figure 5.4 shows an example of a multi-resolution bull's-eye model. Here the shaded polygon is the area that must be represented with high precision. Darker lines outline triangles from coarser lines in the hierarchy.

Perspective viewing models, which differ only in the number of focal viewing ranges to be considered, are easily produced with an extension to this algorithm. For perspective views, as with zoom, I select levels of detail at varying distances such that each level's error tolerance is no more than the width of the area covered by a pixel.

5.4. Line-of-sight

Given two points *p1* and *p2* in space and an object model, line-of-sight calculation determines whether those two points can “see” each other, i.e. whether or not their view is obstructed by the object. Figure 5.5 illustrates this problem. Typically, line-of-sight is calculated by traversing the path from one point to the other. Each surface patch encountered is tested for obscuration. In a high-precision model, the number of patches examined could be very large indeed.

Line-of-sight calculation is an important analysis function that has merited study in the past. For example, Petty et al. [PCFP92] recently described three line-of-sight algorithms. Significantly, one of the better-performing algorithms

uses a triangulated surface model. Clarke [Cla90] discusses other intervisibility algorithms. Yet none of these takes advantage of a hierarchy to further improve performance. Mine does.

I use my hierarchical triangulation model to reduce the number of triangles that must be examined. The error tolerance specified for each level of detail guarantees that no point on the actual object will be further than that from the triangulated model at that level. Therefore if a triangle at level *i* definitely obstructs the view between *p1* and *p2* — taking into consideration any possible errors at that level — then there is no sense examining level *i*+1: *p1* and *p2* cannot see one another. Likewise, if a triangle at level *i* cannot obstruct the view, then its children need not be examined. Although this algorithm can cause more triangles to be examined in the worst case, on average the time savings should be significant. My algorithm works as shown in figure 5.6.

```
function line_of_sight (p1, p2 );
Input :     Triangulation hierarchy T, and 2 points in space — p1 to p2 —
            for which intervisibility is a question.
Output :    True — if p1 and p2 can see one another — or False.
begin
    find triangle A in finest level of detail n containing point p1 ;
    find triangle B in finest level of detail n containing point p2 ;
    if (A = B ) then (* there's nothing to block the view *)
        return True;
    else begin (* need to check surface patches between them *)
        find triangles A_i and B_i , ancestors of A and B at level of detail i
            such that A_i ≠ B_i and parent_of (A_i ) = parent_of (B_i ) ;
        for each edge between p1 and p2 do
            put edge, i on queue;
        repeat (* see if any edge in the queue can block the view *)
            get edge e and level of detail i from queue;
            elevate p1 and p2 , adding error tolerance for level i ;
            if elevated points can't see over edge e then (* view is blocked *)
                return False;
            else begin
                lower p1 and p2 , subtracting error tolerance for level i ;
                if elevated points can't see over the edge then (* check further *)
                    put e, n+1 on queue;
            end;
        until queue is empty;
    end;
    return True; (* nothing is blocking the view *)
end;
```

Figure 5.6. Line of sight algorithm