A RADAR-VALIDATION SERVER

Zhiyong Li, Peter Mills, Lars Nyland, Jan Prins, John Reif, Bill Yakowenko
- Revised: Sun Sep 25 19:02:58 1994 by nyland@cs.unc.edu

Function

The radar-validation server computes a "coverage map", showing the amount of coverage of the radar over a given doctrine. The coverage varies due to the limit in height of the radar and the horizon shadow (inability to see past the horizon) imposed on the radar. The coverage map indicates the amount of the doctrine covered by the radar, computed over a grid of a particular resolution.

An example problem is

![Diagram showing radar coverage and doctrine geometry]

The result of doctrine validation is

![Image showing a spherical radar region in blue (intensity varies with height), and a spherical doctrine that is greenish where it is fully covered by the radar, yellow-orange where it is partially covered, and red where it is not covered at all. Any activity in the red area would go undetected by the radar shown.]

This image shows a spherical radar region in blue (intensity varies with height), and a spherical doctrine that is greenish where it is fully covered by the radar, yellow-orange where it is partially covered, and red where it is not covered at all. Any activity in the red area would go undetected by the radar shown.

Interfaces

The radar-validation server retrieves data from several sources in order to compute the coverage-map. It outputs only the coverage-map. All data acquisition and distribution is done over TCP sockets, communicating with other processes across a TCP/IP network.

Input
Input data comes from many sources. Initiation by a user or periodic process starts the radar-validation computation. Data required for the computation come from the doctrine-server (giving the geometry of a doctrine), the track-server (locating objects, such as the ship on which the radar is located, on the Earth), and the radar-parameter-server (yielding the geometry of the radar coverage). Each of these arrives as a textual lisp-like S-expression, and the format of each is described.

**Initiation Request**

The radar-validation server is waiting for a command to arrive to indicate what doctrine to validate against the radar. The arrival of a message on a specific port begins the coverage map computation. The format is

**VERSION 2,** in use, 10/15/94

(validate doctrine-name dam-timestamp track-server-timestamp)

**VERSION 1,** no longer in use.

(validate doctrine-name (clip-region) (resolution)
  track-server-port
document-server-port
display-server-port)

where the doctrine-name is the name of a doctrine in the current data, and the 2 timestamps that follow are timestamps to ask the doctrine-server and the track-server for data at a particular time. For version 1, the doctrine-name matches a name in the doctrine dataset; the clip-region is a list of 2 positions (lat/long) on the Earth that define a bounding region; the resolution is a pair of numbers, indicating the number of sample points in the rectangular coverage-map; and the three ports indicate where to get/send the data (ports have the form ":(machine-id port-number)").

An example is

For version 2

(validate carrier 120006 120001)

or, for version 1

(validate carrier ((22.1 24.8) (49.5 54.0)) (64 64)
  (alpha.ship.navy.mil 9991)
  (beta.ship.navy.mil 9001)
  (gamma.ship.navy.mil 9500))

**Doctrine Data**

Given the initiation command, we need to acquire the region that the doctrine covers. This can be done by sending the doctrine-authoring-module (a prototype developed by NYU Griffin team) a "get_doctrine_at" command with the name of the doctrine and a time-stamp value. The DAM then returns the doctrine in a lisp-like format that has been well-specified by Malcolm Harrison.

**Track Data**

Track data is required to position the radar geometry at the ship's location and in the case of slaved doctrine validation, the position of the object on which the doctrine is slaved. The
geometric information retrieved from the DAM may be in relative coordinates, thus a basis must be determined and reflected in the geometric information. Of course, there must be coordination of the identifiers used in the doctrine data with the identifiers used for each track datum so that the necessary track data can be extracted.

No specification of where we should get track-data was ever supplied. As an exercise, we developed a track-server in Proteus, and extended it to cover all of the scenarios given in the Scenario Descriptions. We provided this service for our own purposes, and also invited other ProtoTech teams to use it as well. Intermetrics used it extensively, as did the NSWC developers working on the graphical output system. As the project was coming to a close, Bob Balzer of ISI supplied an enhanced version, which all of us used since 9/26/94.

**Radar Parameters**

The last datum that needs to be obtained is the shape of the radar. This has not been specified (other than pictorially), thus we obtain this information by creating it in our own code. Since it is nothing more than a geometric region, we have asked that it be stored with the other geometric regions by the DAM, and thus can retrieve it in a manner similar to doctrine-retrieval.

**Output**

The output of the radar-validation server is called a coverage-map. It is a grid of values indicating the coverage of the radar at each point in the grid. This information is sent to a display-server, a process that will display it visually.

Currently, the format is a list of values, where each value is a pair consisting of a region and it's coverage value. More precisely,

Version 2, since 10/24/94, flattened at Balzer's request

```
coverage-map ::= (element ...)
element ::= (region value)
region ::= (min_lat min_long max_lat max_long)
value ::= (radar_depth doctrine_depth intersection_depth)
```

Version 1

```
coverage-map ::= (column ...)
column ::= (element ...)
element ::= (region value)
region ::= (min_lat min_long max_lat max_long)
value ::= (radar_depth doctrine_depth intersection_depth)
```

By sending data in this form (rather than a grid of pixel values), the display function can be tailored to find the best method of converting this information into a meaningful representation. Our current display-server stub generates a portable-pix-map (ppm) file that shows areas where the doctrine is covered in shades of one color (doctrine_depth > 0 and intersection_depth > 0), where it is not covered at all in an eye-catching color (doctrine_depth > 0 and intersection_depth = 0), and the rest of the radar coverage in shades of another color (radar_depth > 0 and intersection_depth = 0).
Implementations

There are two parts to prototyping the radar-validation server. One is the interface to the rest of the AEGIS Hiper-D system, providing the ability to communicate with other processes over TCP/IP sockets. The second part computes coverage-map, a task where many different algorithms can apply. To this end, we have developed three different strategies of computing the coverage-map, one uses a ray-casting method from the center of the Earth, while the others decompose the problem into hierarchical regions (an oct-tree) where the coverage is simpler to compute within one small sub-region. Either approach can be used within the communications framework we have established.

- **Ray-Casting (by Bill Yakovenko, UNC)**
- **Static Oct-tree Decomposition (by Zhiyong Li, Duke University)**
- **Adaptive data-parallel Oct-tree Decomposition (by Peter Mills, Duke University)**

Conclusions

Metrics

Design Commentary

Two types of problems have plagued us throughout the development of this prototype. They both have to do with the data supplied, but in different ways. The first has to do with the myriad of meanings of geographic measurements (which is still in flux). We've had measurements in feet, yards, K-feet, miles and data-miles, sometimes mapped to a flat surface, sometimes a sphere. It seems that it is of the utmost importance to provide a consistent measurement system, along with a full set of translation mechanisms (such as longitude/latitude/altitude to radius/theta/phi spherical coordinates), and distance measurements as well (for a variety of position representations). If we had had such an interface, as well as clarity in the specification and scenarios, many of our design meetings would have been significantly shorter.

The other specification with which we have had difficulty is in actually receiving the data. No formats were provided for incoming data supplied by NSWC until 8/29/94, and when it was announced, it was in a form that did not interface well with the prototyping systems being used. It also had the serious drawback that one process failing to read the incoming data could lock up the entire system with overfull buffers. This situation has evidently uncovered many of the problems of interfacing typical prototyping language components into a production system (especially a system with real-time deadlines). Prototyping systems do not typically allow a programmer full access to the machine (there are no high-level models for expressing some of the low-level capabilities), and prototyping systems typically run slower than production systems. We provided NSWC with an *intermediate server* that would consume all data from the real-time system, and provide data as needed to the prototype modules.
In addition, there are two components in this prototyping scenario that both require absolute geographical information, yet both parts must receive relative information and track information and then convert that into absolute information. There may be good reason for acquiring the track and doctrine data from separate sources, but it seems that one repository for geographical information would allow a single centralized place for the manipulation of geographical data.

**Integration in a Production Environment**

As of 9/27/94, our integration is not yet complete. We are using servers provided by other ProtoTech members, but receive/supply no service to NSWC modules.

**Log Data**

7/14/94 (lsn) Added socket capabilities to Proteus interpreter (2 days)
7/15/94 Discussed several designs with Proteus group members
8/08/94 (lsn) Finished initial dist. framework w/stubs
8/09/94 (lsn) Asked questions about architecture, user-interface
8/10/94 (lsn) Built several stubs, set up my own architecture
8/12/94 (lsn) Inquired about "radar parameters"
8/17/94 (lsn) Revamped code structure s.t. servers could be restarted at will
8/18/94 (lsn) Inquired about what to do with generated coverage map
8/19/94 (lsn) Got radar parameters picture, required new geometries.
8/23/94 (lsn) Received NYU structure of system, talked with Harrison
8/23/94 (lsn) Sought/recv info on track-data service from INMET & NSWC
8/25/94 (lsn) Recv an Ada header of graphical interface package
8/25/94 (lsn) Revised architecture, in light of M. Harrison's DAM desc.
8/26/94 (lsn) Set up track-data server, for use by us and INMET
8/29/94 (lsn) Discussed the optionality of 3d info from my track-server
8/29/94 (lsn) Recv track-data spec from NSWC-- binary, periodic data
8/30/94 (lsn) Req ascii, on-demand track-data, consistent with NYU-DAM
8/30/94 (lsn) Explored track-data service problems (buffer o-flow, lock-up)
8/30/94 (lsn) Requested our two compute engines to have the same interface
8/30/94 (lsn) Discussed who should take care of providing track-data svc
8/30/94 (lsn) Started discussion of process priority, concurrent i/o
8/31/94 (lsn) NSWC track-data format and values require clarification
9/01/94 (lsn) Provided NSWC with track-data conversion program
9/12/94 (lsn) Made commands case-insensitive
Ray-Casting (by Bill Yakowenko, UNC)

There are three major steps to the algorithm:

1. Prepare for parallel processing:
   - Normalize the CSG (Constructive Solid Geometry) representations of doctrine and radar to the smaller of CNF or DNF (Conjunctive or Disjunctive Normal Form).
   - Sort all CSG nodes by primitive type, grouping each type together into a sequence for parallel processing.

2. Compute depth maps of doctrines and radar:
   - For each primitive type
     - For all CSG nodes of that type, in parallel:
       - For all pixels, in parallel:
         - Compute the intersection of the ray through that pixel and the CSG node.
     end
   end

3. For all pixels, in parallel:
   - Compute the intersection of the doctrine and radar depths.

Because the CSG object tree was normalized, all of the intersections can be done in a small number of passes, one for each kind of CSG primitive. Currently, there are eight supported primitives: polygon, plane, wedge, circle, rect, negation, union, and intersection. In each pass, all CSG nodes of some type are evaluated at all pixels. On a machine with enough parallelism, all pixels could be evaluated at once, and the total time would be based on the number of distinct primitives, which is a constant.

The intersection of a ray and CSG primitive object take advantage of the knowledge that, in this algorithm, rays are vertical. That is, they pass through the center of the Earth, and are perpendicular to the surface at the point where they cross it. Because of this, intersections of rays with leaf CSG nodes are simple: if the ray is within the boundary of the object, its elevation bounds are just those given for that object; if not within the bounds of the object, the intersection is null. Intersections with logical constructors are just the intersections of the segments resulting from the lower-level (and therefore previously computed) nodes. Although the same algorithm could work with parallel rays (ie: not all passing through the Earth's center), the computation of intersection between rays and leaf objects would likely be more difficult to program.

The algorithm went through two major versions. Initially, adaptive subdivision of the map using objects' bounding boxes was seen as a way to reduce the amount of work done. This was developed and refined in several ways to increase its speed or efficiency. Later, it was decided that this approach made extreme parallelization difficult.

In response to that, it was decided to simply evaluate every pixel in parallel, and every object of each CSG type in parallel. This is possible only by normalizing the CSG tree, so there is a specific sequence of primitive types can be evaluated in order, and never need results of a type before it is ready. Although this wastes some amount of work, in practice the clipping region
would not be much greater than were the bounding boxes used in adaptive subdivision, so the waste is small. Furthermore, this allows for a great deal more parallelism, and therefore speed.

The advantages and disadvantages of this algorithm both stem from the fact that it analytically computes depths at each pixel. It gains speed and flexibility by taking advantage of the fact that object boundaries are oriented with the projected rays defining each pixel. Therefore, it needs only a small amount of computation per pixel per object, and produces very accurate intersection depths. The disadvantages are that the analytic solution to other primitives might prove difficult. For instance, spheres should be easy to handle, because each pixel could determine its upper and lower limits based only on distance from the sphere's center. But a cylinder whose axis is skewed, neither vertical nor horizontal, might be a little more challenging, and an arbitrarily rotated complex shape could prove difficult.

Conclusions

Metrics

Discussion : 20
Reading : 4
Design : 5
Coding : 70
Testing/Debugging : 30
Total : 129

Buzzwords

- CSG tree normalized to smaller of CNF or DNF
- Leaves stripped & sorted into sequences by type
- Parallel evaluation of one sequence for each primitive type
- Analytic solution takes advantage of Earth-centered projection
- Only one "contained or not" test needed per pixel per CSG leaf

Log Data

7/11/94 (wfy) first heard of the project
7/15/94 (wfy) meeting w/Lars & Zhi-yong, algorithm outline decided
(wfy) coding begun
7/18/94 (wfy) initial coding done; testing/debugging begun
7/20/94 (wfy) first operational program
(wfy) added print_map function
7/21/94 (wfy) added ppm_map function
(wfy) adjusted to track radar-only regions separately
8/03/94 (wfy) adding doctrine-per-region tracking during subdivision
8/18/94 (wfy) first integrated with validate.pro
(wfy) CSG normalization (flattening) routine
(wfy) skip adaptive subdivision, flatten CSG tree
(wfy) put leaves in table, sequentializing evaluation of shapes
(wfy) index_leaves() procedure,
handling of primitives and constructors made identical
9/06/94 (wjq) flattened-CSG code operational
9/07/94 (wjq) removed bounding boxes from CSG object definitions
9/10/94 (wjq) new primitive objects added; old ones commented out
9/12/94 (wjq) added ascii2csg() functions
9/14/94 (wjq) added a level of nesting to output, pixels, and srays
   (wjq) modified print_map() and ppm_map() to accept this
9/23/94 (wjq) adjusted to use correct units for distances, angles, etc.
   (wjq) memory-saving modifications:
   (wjq) allowed to normalize to smaller of CNF/DNF
   (wjq) some date redundancy eliminated
   (wjq) deallocation of dead data space
Coverage Map Calculation Using Oct-tree Approach

by Zhiyong Li and Peter Mills, Duke University

Problem Description

Given a CSG representations of radar and doctrine, determine the radar coverage. More exactly, determine the volume ratio of (radar \^ doctrine) with doctrine.

Algorithm Design and Data Structure

We use an oct-tree to represent the space decomposition, and compare each sub-cube against the CSG representation. There are four possible results, contained, disjoint, overlap and maybe, and each guides us to further divide the sub-space or not. In order to complish this goal, the following algorithms are designed.

Data Structure

This consists of the choice of CSG primitives and their representation, which are listed below,

- point : (x,y,z)
- sphere : (center, radius)
- cylinder : (center, center, radius)
- halfspace : (point, normal, direction)
- cube : (center, radius)

And we make following assumptions,

1. All objects are convex.
2. A cylinder is infinite in current implementation. We also tried the finite cylinder and have developed algorithms for both of them.
3. A cube is assumed orthogonal to a Cartesian coordinate system.

Space decomposition

The idea is to derive an oct-tree whose leaves comprise cubes interior to CSG object. Derivation proceeds by recursively subdividing initial bounding cube of CSG object, oriented orthogonally to object base. Recursive subdivision proceeds by testing if cube<=object (contained in), using a quaternary logic (detailed later) for this test (Yes,No(disjoint),Overlapping, or Maybe) where subdivision proceeds on Overlapping or Maybe. The main trick is that we do not really to build a oct-tree, but use a stack to do the depth-first search of the oct-tree tree.

Another optimization we have done is to keep track of the relationship of the cube with each sub-formula of the CSG expression. When the cube is subdivided, we may not need to run the checking algorithm in some cases. Even though we do get slightly speed up, we finally gave up
on this effort because this approach requires vast information transferred between functions. This not only slows down the program execution, it also complicates the code and makes it extremely hard to read. An important lesson is that it is complicated to keep track of states of programs in a functional languages.

**Quaternary Logic**

The idea is that cube<=object relation is answerable as Y/N/O for primitive objects, but for non-primitive the answer may not be immediately obtainable (e.g., for object composed using intersection), in which case we can subdivide cube and try again. For non-primitives the answer to cube<=object proceeds by recursive structural expansion of CSG expression, using <= test on subcomponents whose results (with values of Y/N/O/M) are combined using logic operators AND, OR, and NOT conservatively extended for quaternary values.

Tables: (think of as: is cube<=A^B?, etc.)

\[
\begin{align*}
M \land Y &= M \\
M \land N &= N \\
M \land M &= M \\
M \lor Y &= Y \\
M \lor N &= M \\
M \lor M &= M \\
\lnot M &= M \\
\lnot N &= Y \\
\lnot Y &= N \\
\lnot O &= N \\
O \land Y &= M \\
O \land N &= N \\
O \land M &= M \\
O \land \lnot O &= M \\
O \lor Y &= Y \\
O \lor N &= M \\
O \lor M &= M \\
O \lor \lnot O &= O \\
\end{align*}
\]

(chosen so that \(^{lnot}\)cube<=B\(^{lnot}\)iff cube*B=empty)

**Geometric Algorithm**

The algorithms include the checking of relationships of cube with sphere, halfspace, infinite cylinder and finite cylinder. We have designed several algorithms especially for the checking of a cylinder with a cube, ranging from very accurate and slow to very fast but not accurate.

**Implementation**

The current implementation includes three module: interface, main and geometric algorithms.
**Interface**

The interface takes the input and transforms it into the format which can be dealt with by our geometric algorithms. It includes the complete transformation sub-module from longitude/latitude representation to Cartesian coordination system which is orthogonal to the clipping region. A flow is shown below,

(\text{lat, long}) \ldots \text{on surface of the earth} \\
| \\
| \\
| spherical coordinate (w/ origin at the center of the earth) \\
| \\
| Cartesian coordinate (including rotation and transformation of the coordinate system to the center of the clipping_region and orthogonal to that plane)

It also transforms the cylinder as the intersection of infinite cylinder with two spheres, and the polygon to the intersection of a series halfspaces with two spheres.

**Main**

This is the main control module. It calls the interface to make the format transformation and calls the geometric module to calculate the volume. It also includes the oct-tree decomposition algorithms and output generation functions. The general flow is shown below, Call interface

For radar, doctrine and (radar^doctrine) do

- Initialize pixmap to 0.
- Push initial cube (bbox of doctrine) on stack.
- While stack not empty:
  - pop cube from stack.
  - cube <= object? \ldots \text{call geometric module}
  - if no => continue
  - if maybe || overlap =>
    - subdivide cube (if >= threshold)
    - push subdivisions on stack
  - if yes => drop cube onto pixmap
    (i.e., pixmap[i,j] += cube_radius for i,j in cube bottom)

Generate the output

**Geometric Module**

Besides the implementation of the geometric checking algorithms, it also includes the functions for quaternary logic.

**Discussion**

**Pros and Cons of oct-tree approach**
The objects defined in our geometric algorithms are quite general and can deal with most of the random shapes and rotated 3D objects. The oct-tree representation is easy to adapt to the data parallel execution and presumably it can need more accurate result and efficient execution. Another interesting of this prototyping exercise is that we developed the quartenary logic to fully handle the logic operations in CSG expressions and the coordinate system transformation itself could be a useful sub-module for other experiment.

However, the generality also leads the inefficiency for this particular problem (look at out ray-tracing approach). There is quite a bit of transformation and adjustments to be made to the original input, obscuring the beauty of our original design. Well, part of the argument for this is that we did not know what is the input and output was going to be when we began development. The productive result from this is that our prototyping system is quite flexible which can relatively easy to adapt itself to the changed requirements, demonstrated here by the changing geometric primitives.

**Development time and effort metrics**

The time I spent is divided into following five parts (unit : hours),

- Discussion : 16
- Reading and Literature Searching : 21
- Algorithm Design : 48
- Coding : 22
- Debug : 33

Total : 140

Warning: this is really vague statistic especially there are lots of time when the problem was hanging in our mind which is hard to count.
An Adaptive Data-Parallel Oct-Tree Solution

by Peter H. Mills

The objective of doctrine validation is to compute the percentage coverage of the doctrine by a radar volume, and ideally to produce a 2D picture showing this percentage of coverage. The heart of doctrine validation consists of computing the volume of the intersection of the radar and doctrine shapes. We explored at a high-level several simple algorithmic variants for this volume computation using the Proteus system. In particular we explored two techniques for volume computation: ray casting and recursive spatial decomposition using octrees. In both cases technically the result we were after was the projection of the volume onto a 2D surface to produce a 2D picture. We successfully designed and prototyped in Proteus algorithms for both a sequential and data-parallel version of volume computation using adaptive octree decomposition.

The key idea underlying volume computation using octrees is to recursively subdivide the space into 8 orthogonal cubes (octants) and keep only cubes interior to an object, thus deriving an "oct-tree" whose leaves comprise the volume. The technique is "adaptive" in the sense that at each level "empty" leaves -- those exterior to the object -- are thrown away. Our sequential algorithm for octree decomposition performed a depth-first derivation, and was unique in that it used a 4-valued logic for deciding if a cube was interior to object. More precisely, since the object was represented as a CSG ("constructive solid geometry") expression built from union, intersection, and complement operators and primitive shapes, we evaluated whether a cube was interior to a CSG expression by comparing whether the cube as contained in each primitive in the expression and then combining the resulting values -- which were one of "contained", "disjoint", "overlapping", or "unknown" -- by structurally reducing the CSG expression using and, or, and not logic operators conservatively extended for these 4 quaternary values. Subdivision then proceeds on overlapping or unknown comparison results, and the entire process recurses until a desired level of accuracy is reached. To derive a picture from the derived volume, the octree cubes found to be contained in the object had their heights added to, or "dropped" onto, a 2D pixmap. Thus the algorithm was informally termed a "chop-and-drop" approach.

The above algorithm was successfully prototyped in Proteus. However it became rapidly apparent that volume computation is a particularly computationally demanding subtask. Thus we pursued the development and implementation of a data-parallel variant of the algorithm, with the strategy of first expressing the nested data-parallel algorithm in Proteus and validating it using the Proteus interpreter, and then realizing a parallel implementation using the Proteus transformation machinery to translate the nested data-parallel core to the C+DPL vector library. The resultant vector code would be invoked from the Proteus interpreter using the "prog" MIF construct, and would execute in parallel on a MasPar MP-2 or other parallel platform.

The data-parallel algorithm for octree decomposition that we successfully developed and validated using the Proteus interpreter was unique in several respects. First, it was a functional program that used single assignment (a "let" construct). Second, to increase the expressed parallelism the CSG expression was represented in a fixed-depth normal form as a disjunction of conjunctions of positive or complemented primitives. The algorithm then proceeded in
log(sqrt(#pixels)) steps of recursive spatial subdivision, at each step processing all the octants in parallel. At each step, each octant was compared with all CSG primitives in parallel, and the result was reduced using the normal form expression to yield a containment decision for each octant. As before, subdivision proceeded on a result of overlapping or unknown. A further algorithmic refinement was to use in containment tests spheres which bounded the octants. The entire amount of code for the data-parallel algorithm expressed in Proteus was ~ 280 lines. Currently we are exercising the transformation machinery and shortly expect to have a functioning parallel implementation.