Volume xxx, (1997) number yyy pp. 000–000

Interactive Boundary Computation of Boolean Combinations of Sculptured Solids †

S. Krishnan M. Gopi D. Manocha M. Mine

Department of Computer Science
University of North Carolina
Chapel Hill, NC 27599-3175
{krishnas,gopi,manocha,mine}@cs.unc.edu

Abstract

We present algorithms and systems for interactive boundary computation of Boolean combinations of sculptured solids. The algorithm is applicable to all spline solids and computes an accurate boundary representation. To speed up the computation, the algorithm exploits parallelism at all stages. It has been implemented on a multi-processor SGI *and takes one second on average per boolean operation to compute the boundary of high degree primitives. The system has also been integrated with an immersive design and manipulation environment. The resulting system is able to interactively evaluate boundaries of the models, display them for model validation and place them at appropriate position using collision detection algorithms.*

1. Introduction

Interactive computer-aided design and modeling systems are being developed to help designers create parts of a model as part of the design/engineering process. A major goal of these systems has been to provide an integrated and interactive three-dimensional environment in which users can experiment with different shapes, interactively display them for model validation, place them in correct positions and check for interference with other parts in a large model. In this paper, we restrict ourselves to parts designed using boolean operations on primitive solids. The set of primitives include polyhedra, quadrics, surface of revolutions and sculptured solids (whose boundaries can be represented as NURBS surfaces). Boolean combinations of such solids are used in most CAD and modeling systems. For example, the Bradley fighting vehicle (shown in Figure 1), has been modeled using boolean operations. The model consists of more than $5,000$ solids, each designed using 5 to 8 boolean operations.

† Supported in part by an Alfred P. Sloan Foundation Fellowship, ARO Contract DAAH04-96-1-0257, NSF Grant CCR-9319957, NSF Grant CCR-9625217 , ONR Young Investigator Award, Intel, DARPA Contract DABT63-93-C-0048 and NSF/ARPA Center for Computer Graphics a nd Scientific Visualization

Interactive Boundary Computation : Interactive three-dimensional design and modeling systems can benefit greatly from fast, accurate boundary computation. Typically the design process goes through a number of iterations before converging on the correct model and placement. In such cases, the turn-around time per iteration should be minimized. In addition, the increased emphasis on 3-D interaction techniques and immersive interfaces further increase the need for interactive boundary computation. Current techniques for direct rendering of constructive solid geometry solids are either too slow (e.g. ray-tracing) or restrict the number of boolean operations or the degrees of the primitives. Given the boundary, interactive systems for rendering the surface description are available on current graphics systems [29, 16]. Furthermore, algorithms for interactive collision detection and object placement need an exact description of the boundary [18]. Other applications of boundary representation (B-rep) include computation of solid's mass properties.

Main Contribution: We present fast, parallel algorithms and systems for computing the boundary of boolean combinations of sculptured solids. The resulting boundary is represented in terms of trimmed Bézier patches. Given two prim-

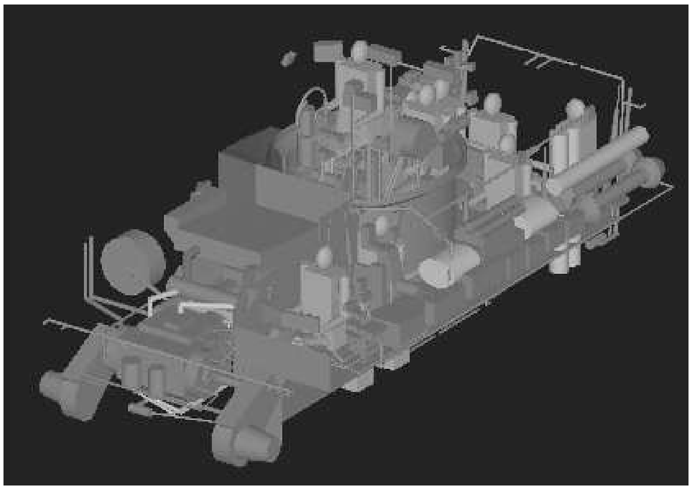


Figure 1: Interior of the Bradley fighting vehicle composed of over 5000 solids

itives, our algorithm involves evaluating surface intersections, curve-merging and component classification. To speed up these computations, we parallelize and distribute them among multiple processors. The main contributions in this paper are:

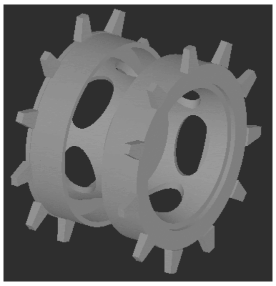
- **Accuracy:** Each intermediate primitive and the resulting B-rep solid is represented as a collection of trimmed Bézier patches and an adjacency graph. The trimming curves are the result of accurate surface intersection computation. These trimming curves are stored, for storage efficiency, as spline approximations. We take special care in preventing cracks in the resulting model.
- **Parallel Implementation with Load Balancing:** Our algorithm exploits parallelism at all stages of boundary computation. Our system can work on any shared memory parallel system. We also present algorithms to perform load balancing with minimum use of locking while parallelizing the boundary computation algorithm among various processors. We have applied it to interactively compute the boundaries of the track (shown in Figure 2) and other parts of the Bradley fighting vehicle. Each solid is defined using $5-8$ boolean operations on high degree primitives.
- **Interactivity:** Our system uses an efficient and accurate surface-surface intersection algorithm and topological information to achieve optimal ray shooting tests during component classification. It performs only one ray-shooting for each boolean operation. In practice, it takes about *one second on average* to perform one boolean operation on high-degree sculptured solids (on a multi-processor SGI configuration).
- **Interface with Immersive Environment:** We have integrated our boundary evaluation system with CHIMP [24], an immersive design and manipulation environment. In the resulting environment, an immersed user specifies the primitives and their location using a 3-D interface. The system is able to interactively evaluate its boundary, display it and place it at an appropriate position using collision detection algorithms. Figure 7 shows one such part designed in this integrated environment.

Organization: The rest of the paper is organized in the following manner. A brief discussion of previous work in the area of boundary computation and interactive rendering of CSG solids is given in Section 2. The representation of each solid in our system is explained in Section 3. Section 4 describes our B-rep generation algorithm. Section 5 describes our load balancing scheme for the parallel implementation. The interface of our system with an immersive design environment is discussed in Section 6. Section 7 shows the performance of our system on some models and the speed-up due to parallelism.

2. Previous Work

There is a vast amount of literature on integrated three-dimensional modeling environments, direct rendering of CSG primitives and boundary computation. The Alpha_1 CAD system developed at the University of Utah has many such features. An interactive approach for design, analysis and illustration of assemblies has been presented in [3, 30].

Some graphics systems include capabilities to directly render CSG models [12, 8]. However, they either restrict the number of CSG operations or the degrees of the primitives (e.g. only quadrics) or are not able to render complex models. Algorithms based on enhanced Z-buffer often require multiple rendering passes along with fast rasterizing systems for such primitives [31, 30]. Other techniques and systems for direct rendering are based on ray-casting [11, 32]. More recently, researchers at Army Labs have developed real-time ray tracing



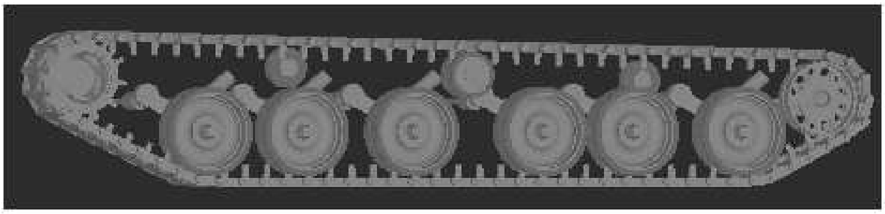


Figure 2: B-rep of the drivewheel model(left): 44 Boolean operations took 49 secs. to generate on our system. Track (right) showing the placement of the drivewheel model

systems to render complex CSG models on high-end configurations consisting of $50-100$ processors [25]. Ray representations along with specialized parallel architectures [22, 4, 23] like the RayCasting engine and 'Solids engine' were used to achieve interactive solid modeling on low-degree primitives like quadrics. Mantyla and Ranta[20] describe methods to perform interactive solid modeling using HutDesign.

Generating B-reps from CSG representations of solids has been a classic problem in solid modeling [10, 19, 27, 28, 26]. Most of the earliest work in generating B-reps concentrated on polyhedral models. Weiler[35] proposed edge-based data structures for non-manifold modeling in curved surface environments. Casale[2] and Menon[23] proposed algorithms for performing analytic solid modeling for quadric and freeform surfaces. Most of the recent work in the literature on Boolean combinations of curved models has focussed on computing the surface intersection between a pair of B-spline or Bézier surfaces [33, 13, 14]. They concentrate on resolving all the components of the intersection curve and finding convenient representations for it in modeling applications. Even though a great deal of work has been done in this area, we are not aware of any parallel system which explicitly computes the boundary of CSG solids. Rossignac[30] presents algorithms for interactive inspection of cross-sections and interference between solids of bounded degree and limited height of CSG trees.

Though rendering of low degree CSG models can be done at interactive rates, we are not aware of any system which evaluates B-rep accurately at interactive rates. Further, even if the model can be visualized at interactive rates without the boundary evaluation, by using direct rendering methods, it requires very high end graphics machines or dedicated hardware systems. Due to the lack of boundary representation, the versatility of such systems is lost. For example, these systems cannot be used in places like immersive virtual environment of assembly unit of a tank, where collision detection is imminent. In this paper, we fill this gap by presenting algorithms and systems to evaluate B-rep at interactive rates and display them for model verification.

3. Representation of Solids

In this section, we present our representation for a solid. Our algorithms assume that all B-rep solids are specified in this format.

Each solid is represented as a set of *trimmed* parametric surface (tensor-product Bézier) patches which define the solid boundary. Each patch $\mathbf{F}(s,t)$ is defined over the parameters s and t. This can be viewed as a mapping of the unit square $0 \leq s,t \leq 1$ in the (s,t)-plane (also called the *domain* of the Bézier patch) to $\mathcal{R}^3$. We also maintain topological information in the form of an adjacency graph.

Topological information of the solid is maintained in terms of an adjacency graph. It is similar to the winged-edge data structure [10, 21]. To start with, we assume that each of the input objects has *manifold* boundaries, and the Boolean operation is *regularized* [19]. While it is possible to generate non-manifold objects from regularized Booleans on manifold solids, we assume for the sake of simplicity that such cases do not occur. It is a well-known fact that, while dealing with topological representation of curved objects, global resolution of edge ambiguities cannot be guaranteed at times [10]. Given these assumptions, it can be shown that an unambiguous topological representation is possible for a solid.

A trimmed patch consists of a sequence of curves defined in the domain of the patch such that they form a closed curve ($\mathbf{c_i}$'s in Figure 3). In the figure, the $\mathbf{c_i}$ refer to the algebraic curve segments forming the trimming boundary. The portion of the patch that lies in the interior of this closed curve is retained. Most of these trimming curves correspond to intersection curves between two surfaces. Therefore, these curves are typically algebraic curves that do not admit a rational parametrization ([1]). We represent these curve segments ($\mathbf{c_i}$) by their algebraic equation (for accuracy), and a spline approximation (for efficient computation) and the two endpoints ($\mathbf{p_i}$ and $\mathbf{p_{i+1}}$).

This representation of a solid lends itself to a description in terms of *faces, edges, and vertices* analogous to the polyhedral case. Each *face* is a trimmed patch. Each of the trimming curves form an *edge*, and are formed as an intersec-

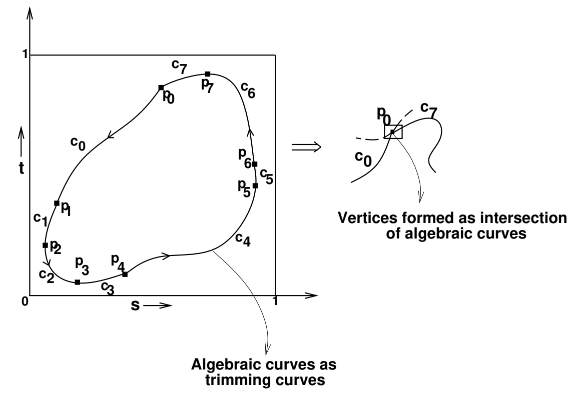


Figure 3: Representation of a trimmed patch as algebraic curve segments

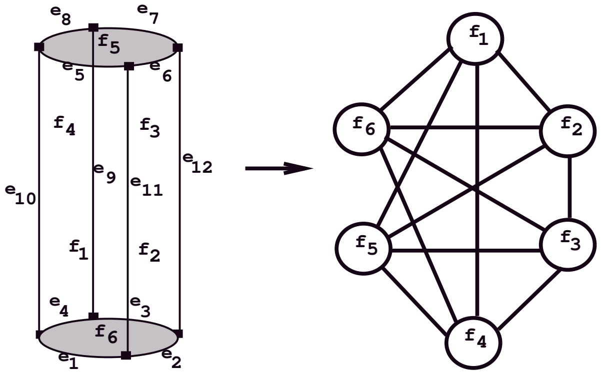


Figure 4: A cylinder and its face connectivity structure

tion of two surfaces (faces). Finally, endpoints of edges form the *vertices*. They can be represented as an intersection of three surfaces. Figure 4 shows an example solid and the face connectivity structure that we maintain. We also maintain the two faces that are adjacent to each edge, and an anticlockwise order of faces around each vertex.

4. Algorithm Overview

In this section, we describe the algorithm to evaluate the boundary. Let the number of patches in one solid be m and those in the second solid be n and let the degree of each patch be $d_s \times d_t$.

The algorithm to evaluate the Boolean operation between the two solids runs in seven stages. We now give an overview of each stage.

Stage 1: (Figure 5) A main part of the algorithm is to compute the intersection curve between the two solids. Hence each patch of one solid has to be checked for intersection with each patch of the other solid. However, not all the mn pairs would intersect typically. We prune out most of the non-intersecting pairs using a two step process.

Initially, we compute the 3D bounding box for each patch. This is done in parallel as the construction of bounding boxes for each of the patches can be done independently. If a pair of bounding boxes do not intersect, the corresponding patches are also non-intersecting (*convex hull property* of Bézier patches [5]). All the redundant pairs are removed using a simple sort on all the bounding boxes. The next step of pruning uses linear programming. We formulate the linear programming problem as follows. Two patches do not intersect if there exists a separating plane between them. Thus we eliminate the patch pairs whose bounding boxes have a separating plane between them. We use Mike Hohmeyer's implementation of the linear programming algorithm by Seidel [34]. By applying these two methods on the two solids, we are left with few pairs of patches that are most likely to intersect.

Stage 2: (Figure 5) As each patch-pair intersection is independent of the other, this operation is parallelized over all patch pairs. Stage 2 of Figure 5 shows the distribution of patch-pairs to various processors.

Stage 3: (Figure 5) The evaluation of the intersection

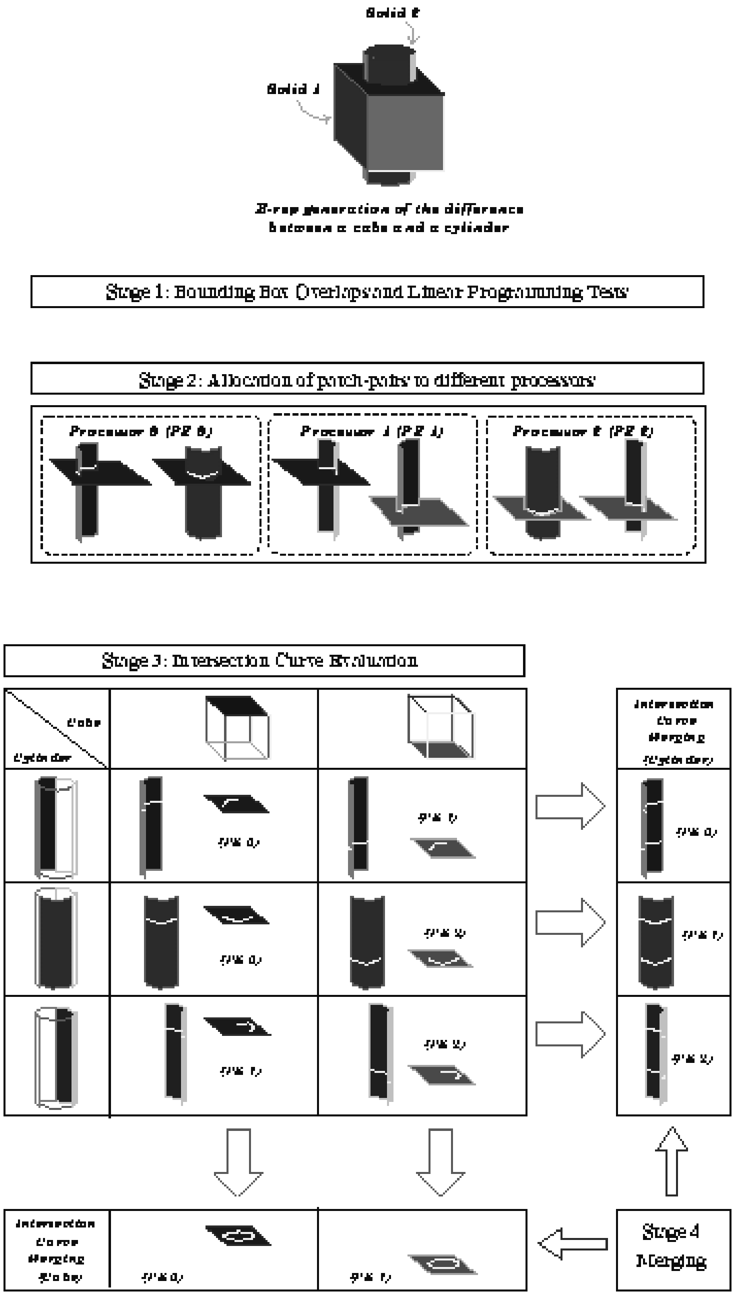


Figure 5: Intersection curve computation and curve merging

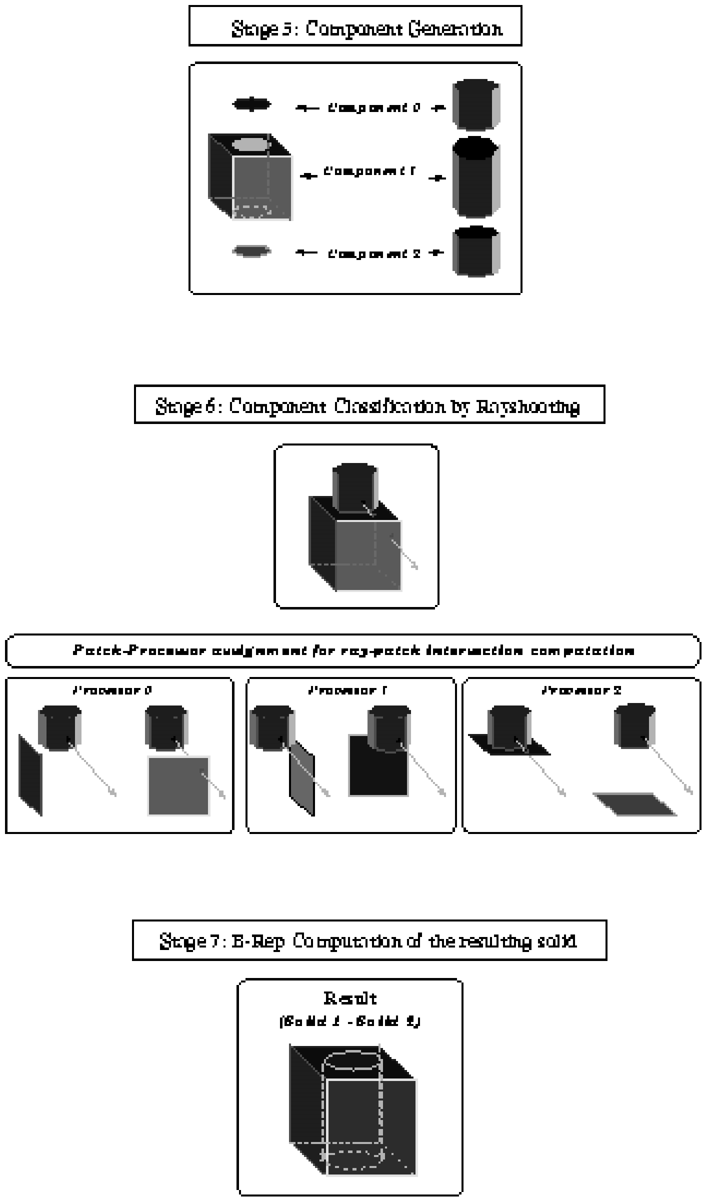


Figure 6: Component generation, classification and B-rep computation

curves between the pairs of patches remaining after Stage 1 is shown in Stage 3 (Figure 5). We use a recently developed algorithm [14] to compute the intersection curve between two parametric patches. The algorithm is based on a combination of algebraic and numerical techniques, and new methods of loop detection, singularity detection, and curve tracing for accurate and efficient computation of the intersection curve.

Stage 4: (Figure 5) B-rep evaluation involves merging of the intersection curves computed in the previous stage. It can be shown that for closed C^0 continuous solids, the intersection curve between them must form a collection of closed curves in space for regularized Boolean operations. Merging is the process of collecting different pieces of the intersection curve and ordering them in sequence to form closed curves in space. The first step of merging is to merge the curves within a patch. The merging of curves between patches in each solid is performed next.

Stage 5: (Figure 6) The merged intersection curve partitions the boundary of the solid into various *components*. The generation of these components is done in Stage 5 of Figure 6. The components are generated by a simple graph traversal algorithm using the existing topological information in each solid.

Stage 6: (Figure 6) Each component has the property that all the patches corresponding to them are either completely *inside* or *outside* the other solid. Therefore, it suffices to compute the inside-outside information of exactly one point in each component. If a solid is closed and not self-intersecting, then this query is answered by computing the number of intersections of a ray, emanating from that point, with the solid. If the number is odd, then the point is inside, otherwise it is outside the solid. To find this number, the ray-surface intersection algorithm has to be run for all the patches on the other solid. To avoid ray-surface intersection computation (an expensive operation) repeatedly over all components of one solid against all patches of the other solid, we use the connectivity information between components (also computed during component generation stage) to perform just one operation per solid [15]. This component classification by rayshooting is done in parallel over all the patches, and forms Stage 6 (Figure 6) of our algorithm. Just like the pruning step for patch-pair intersection (Stage 1), we perform bounding box tests and linear programming to speed up this computation also.

Stage 7: (Figure 6) The particular set operation performed on the solids, and the inside/outside classification of the component, determine if a component is part of the new solid. The algorithm to generate the new solid forms the last stage, Stage 7, of our algorithm. The connectivity information between various trimmed patches of the new solid is found using the topological information of the original solids and the intersection curves.

Since we are dealing with sculptured solids with trimmed Bézier patches, as opposed to polyhedral solids, the complexity of the whole system is increased significantly. The time taken for the surface-surface intersection algorithm can be a cubic function of the degree of the patch in the worst case. Further, in Stage 6, the complexity of ray-patch intersection evaluation is again dependent on the degree of the patch. These parts are computationally intensive and form the main bottleneck in terms of system performance. Hence parallelizing these stages of the system becomes crucial in achieving interactive rates of B-rep computation. While our system can also be used with polyhedral primitives, its performance is more pronounced for sculptured solids. Parallelizing other stages like Stages 1 and 4 of the algorithm has also made a significant contribution to the performance improvement.

5. Load Balancing Algorithm

The problem of load balancing arises when an algorithm has to be parallelized among a number of processors. The running time of the parallel algorithm is directly related to the maximum execution time of the task at a single processor. It is clear that the most effective parallel algorithm is one where the tasks are equally distributed among all the processors. The problem of load balancing has received considerable attention for a long time due to the fact that a single scheme is not applicable for parallelizing all algorithms [17, 7, 9]. The effectiveness of different techniques varies with the nature of the problem it is used for. Hence there arises a need for newer problem specific analysis methods which help in choosing the most effective load balancing technique. We shall now describe three such techniques that we use to shared memory multiprocessor architectures for boundary computation.

- **Static load balancing:** Static load balancing is done by dividing the given problem consisting of n tasks into p (number of processors) parts and submitting each part to a single processor. The size of each problem piece is precomputed and is not changed during execution. This technique works best when the processing time of each of the tasks is known, and the number of tasks does not change during execution. Extracting parallelism in our B-rep converter starts from computing the bounding boxes for all the patches (Stage 1 of our algorithm). As the bounding box computation for each patch is independent of the other, this can be easily parallelized. Further as the amount of work that is to be done for the bounding box computation for each patch is approximately the same, load balancing is achieved statically. Once the bounding boxes for all the patches have been computed, the overlap tests is also performed in parallel.
- **Global queue:** In many algorithms, it is not possible to estimate the execution time of each task. For example, execution time for computing the intersection curve between two surfaces can vary depending on the number of curve components, and length (in terms of number of points traced) of each component. In this technique, when one

processor is accessing the task queue, the queue should be locked to ensure exclusive access (mutual exclusion). This technique achieves the best load balancing, though the extra work done for balancing the load in the form of locks might offset its advantage. In our system, using global queues with locks to perform load balancing was not as efficient as dynamic load balancing (described below). We believe it is because of the reasons cited above.

- **Dynamic load balancing:** In this technique, a local job queue is maintained for each processor. Initially, tasks are assigned to every processor similar to the static load balancing scheme. However, due to suboptimal task division, some processors might complete their tasks before others. In this scenario, the idle processors share the load with the busy processors, thereby balancing the load dynamically. If we can ensure that each busy processor is accessed by only one idle processor at any time, then a lock-free implementation of this scheme is possible. We can also ensure that each task is processed only once, and no task is left out. In our application, load balancing is efficiently achieved by minimal use of locks. Therefore, we use this approach for our most computationally intensive tasks like surface-surface intersection (Stage 3) and ray-shooting computation (Stage 6).

If we ensure that only one idle processor will access a particular busy processor, then a lock free implementation of dynamic load balancing is possible. We enforce a unique one to one correspondence between an idle and busy processor using the following algorithm. A shared global variable *WhichIdleProc* stores the id of the idle processor, which now has the chance to choose its busy processor. This serializes the operation of finding an idle-busy processor pair. In our implementation, we choose a single lock to guard this critical section because the computation time for surface-surface intersection and ray-shooting dominates one locking operation. With each busy processor, we associate a shared variable *MyIdleProc*, which stores the idle processor id that has been paired up with that particular busy processor. These variables are initialized to **NIL**, referring to none of the processors. Each processor also maintains its processor number in a local variable **myid**. Whenever a processor becomes idle, it executes the following code:

```
{
 If (WhichIdleProc == NIL) then {
   GetLock( GetMeAccess);
     if(GetMeAccess == NIL) {
       GetMeAccess = myid;
       WhichIdleProc = myid;
     }
   ReleaseLock(GetMeAccess);
 }
 /* Waiting for my chance */
 while (WhichIdleProc ≠ myid);

 /* All tasks completed */
 If (NoMoreBusyProc()) then exit;

 /* All Busy processors are being load
 balanced by some idle processor */
 while (GetBusyProc() == NIL);

 /* Got a Busy Processor to pair up with */
 MyBusyProc = GetBusyProc();

 /* Make sure no one else captures this busy processor */
 MyIdleProc[MyBusyProc] = myid;

 /* Give chance to next idle proc to find its partner */
 If (NextIdleProc()) then WhichIdleProc = NextIdleProc();

 /* No one to grab the chance */
                    else {
                     GetLock(GetMeAccess);
                        WhichIdleProc = NIL;
                        GetMeAccess = NIL;
                     ReleaseLock(GetMeAccess);
                    }

 /* Balancing the load with the partner */
 LoadBalance(MyBusyProc);

 /* Finished load sharing; Freeing my partner */
 MyIdleProc[MyBusyProc] = NIL;

 /* Register myself as busy */
 If (IHaveLoad()) BUSY[myid] = TRUE;

 /* Work on new list of tasks */
 PerformSurfaceIntersection(); or PerformRayShooting();

 /* Register myself as idle */
 BUSY[myid] = FALSE;
}
```

Initially, the variable *WhichIdleProc* has to be set by the idle processor to gain access to the list of busy processors. Race condition occurs only when the variable *WhichIdleProc* is **NIL** and more than one idle processor try to access it. By making *WhichIdleProc* a critical resource, we can ensure mutual exclusion while setting this variable. This can be achieved by using locks. The number of locking operations can be reduced by allowing free access to *WhichIdleProc* and introducing a new shared variable *GetMeAccess*, which is locked only when a race condition occurs. Locks can be totally avoided by maintaining a random permutation of the busy processor list locally in every processor. This does not guarantee that a single idle processor captures a busy processor, however, the probability of a race condition is very small.

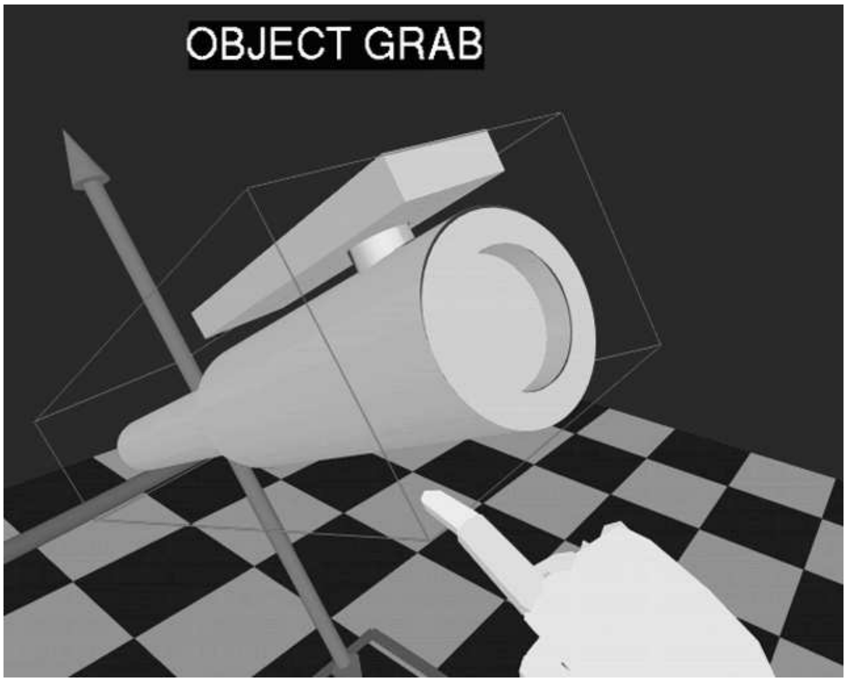


Figure 7: Design of a pen in an immersive modeling environment. The immersive environment is used for model validation and placement using accurate collision detection.

6. Interactive Modeling in Immersive Environments

The ability to perform boundary computations of Boolean combinations of sculptured solids in real-time has many potential applications in the field of interactive design. To demonstrate this, we have combined our system with the Chapel Hill Immersive Modeling Program (CHIMP), a virtual environment application for the preliminary phases of architectural design developed at UNC [24].

In the CHIMP system users create and manipulate models while immersed within a virtual world, as opposed to the customary through-the-window computer-aided design (CAD) environment. CHIMP uses an immersive head-mounted display, a magnetic tracking system with three 6 degree-of-freedom (DOF) sensors (one for the head and one for each hand), two separate input devices (one for each hand) and a high speed graphics computer (Pixel-Planes 5, see [6]). CHIMP includes both one and two-handed interaction techniques, minimizes unnecessary user interaction and takes advantage of the head-tracking and immersion provided by the virtual environment system.

Using the CHIMP system, designers can create complex shapes by specifying Boolean operations on a standard set of simple primitive solids which they interactively manipulate in the virtual world. Information on the types of solids, their position, orientation, and extents, and the desired Boolean operation are sent via standard Unix sockets to the B-rep generator running independently on an SGI. The B-rep generator computes the resulting boundary representation which can then be loaded into the CHIMP environment for further modification and/or exploration. Figure 7 shows the design of a pen inside CHIMP. We are working on improving the interface between the two systems so that design can become much easier.

7. Implementation and Performance

The contribution of this work includes an implementation of the above parallel algorithm on shared memory multiprocessor architectures. Our experiments currently run on an SGI-ONYX with four R10000 processors each with a 194 MHz clock.

In Stage 2 of our algorithm, we allocate patch pairs for exact surface-surface intersection computation. The surface intersection computation and the ray shooting operation in Stage 6 are the two costly operations where the time taken cannot be predetermined. As described in section 5, an implementation of the dynamic load balancing scheme performs better than the centralized queue model with locks. Hence for Stages 3 and 6, we use this method for distributing the patch pairs and ray-patch pairs.

All the processors are given an almost equal number of patch pairs for intersection computation. Lock free implementation is achieved by a carefully chosen data structure. A

(a) Link model (b) Drivewheel model (c) Idlerwheel model

Figure 8: B-reps of some solids from the Bradley fighting vehicle

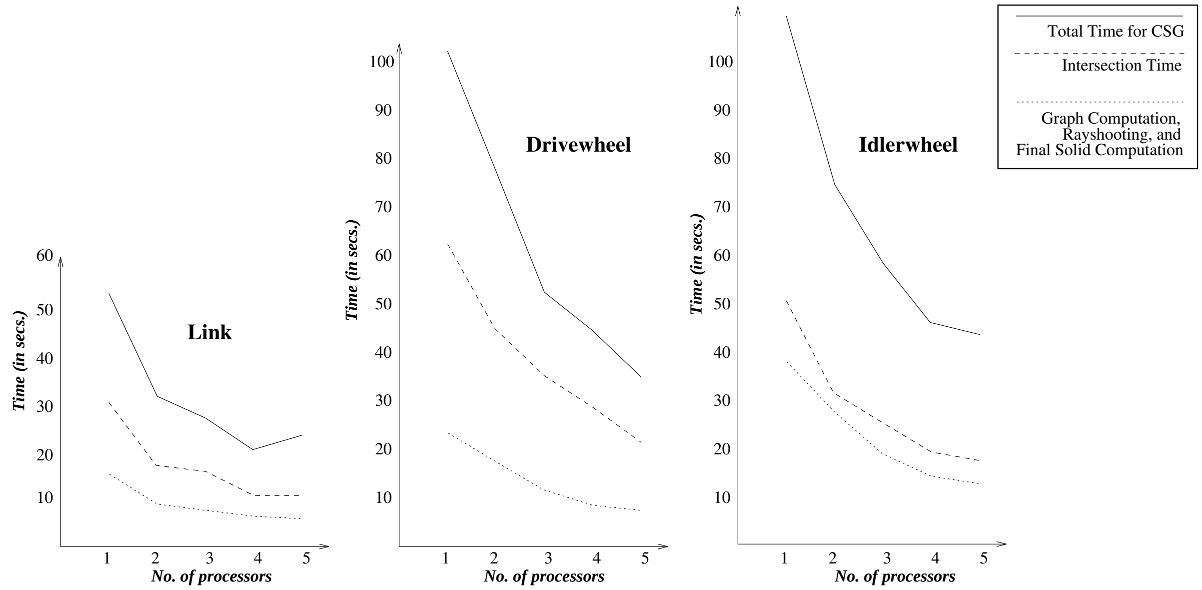


Figure 9: Performance of our system as a function of processor count

two dimensional array is used to hold the intersection curves between pairs of patches. This structure is used for curve merging in Stage 3. It ensures that only one processor is accessing any cell of the two dimensional array and thus, the intersection computation followed by curve merging is done by sharing data without memory contention.

Components classification by rayshooting is performed in Stage 6. Here, the patches of one solid are equally distributed to all the processors. Each processor computes the intersection of the ray with all the patches assigned to it and also counts the parity (odd/even) of number of intersections. These results are finally merged to find whether the total number of intersections is odd or even. This result is used to classify the component. This information is propagated along the adjacency graph of the solid to resolve the other components. The above operation is done once for each solid.

After all the components are classified, the components are chosen to be combined to form the final solid. As the B-rep of each patch of the component is known, the final solid is represented in its B-rep form. This stage is executed sequentially.

Robustness and Accuracy: One of the main problems in B-rep generation is robustness. We perform a number of checkpointing operations that control the accumulation of floating point error. We also handle special cases like face-face and edge-edge overlaps while performing regularized Boolean operations. These are handled as special cases in our system. In practice we have observed that our system generates accurate B-reps on most input cases. Since the implementation was done using floating point arithmetic, we also use tolerances to compare such values. Finding a tolerance that works for all models is very difficult. In some cases, we had to change tolerances to make our system work. Parallelization of the algorithm does not contribute to robustness problems. Currently, we are incorporating B-rep computation using exact arithmetic to prevent most robustness and accuracy problems. The use of exact arithmetic can slow down the computation time, however exploiting parallelism helps significantly in the overall speed.

The accuracy of the B-rep generated is determined by the accuracy of the intersection curves between solids. In our system, the accuracy of these curves can be controlled by the user. Depending on the application, our system can generate very accurate B-reps at the expense of computation time.

7.1. Performance of the algorithm

In this section, we highlight the performance of the algorithm on some real-world models. The model of the Bradley fighting vehicle was obtained from Army Research Laboratories. It is composed of more than 5000 solids each consisting of about 5-8 Boolean operations. We shall describe the performance of our algorithm on three of the solids in the Bradley fighting vehicle whose boundary consists of biquadric parametric surfaces.

- **Link model:** It consists of 16 Boolean operations and the B-rep contains 76 trimmed Bézier patches. Figure 8(a) shows the model. Below the figure is the graph that shows the performance of our system on varying number of processors. It can be seen that the performance becomes worse when we go from four to five processors. Since this is not a very complex model, the setup costs of using five processors outweigh the benefit of parallelism.
- **Drivewheel model:** This model is constructed using 44 Boolean operations. The B-rep is shown in Figure 8(b) and consists of 264 trimmed Bézier patches. The performance of our algorithm improves with increasing processor count.
- **Idlerwheel model:** The B-rep of the idlerwheel (composed of 235 trimmed Bézier patches) is shown in Figure 8(c) and took 48 Boolean operations to generate. Again increasing the processor count reduces the running time because of complexity of the model.

Table 1 shows the numbers corresponding to the graphs (see Figure 9) for the various models. It can be observed from the graphs and table that, in general, parallelism significantly reduces the execution time of our system, and for most practical models, B-rep generation can be made interactive. From the graphs in Figure 9, it is clear that for most models of this size and complexity we obtain almost linear speed-ups up to four processors. It does not help to use more processors.

Model	Total running time (in secs.)				
	1 proc.	2 proc.	3 proc.	4 proc.	5 proc.
Link	51.95	30.88	26.93	20.55	23.44
Drivewheel	102.32	77.39	53.39	49.02	35.67
Idlerwheel	112.40	74.51	58.96	46.10	44.23

Table 1: Performance of the our system

8. Conclusion

We have presented algorithms and a system for interactive boundary computation of Boolean combinations of sculptured solids. The algorithm is applicable to all spline solids and computes an accurate boundary representation. The interactivity is achieved by exploiting parallelism at various stages of the algorithm. It has been implemented on a shared memory multi-processor SGI and takes one second on average per boolean operation to compute the boundary of high degree primitives. The system has also been integrated with an immersive design and manipulation environment.

Acknowledgements

We would like to thank Army Research Labs for providing us with the model of the Bradley fighting vehicle, Lars Nyland for helping us with the parallel implementation, and Sumedh Barde for helping us with the graphical interface for our system.

References

1. S.S. Abhyankar and C. Bajaj. Automatic Parametrizations of Rational Curves and Surfaces III: Algebraic Plane Curves. *Computer Aided Geometric Design*, 5:309–321, 1988.
2. M. S. Casale. Free-Form Solid Modeling with Trimmed Surface Patches. *IEEE Computer Graphics and Applications*, pages 33–43, January 1987.
3. E. Driskill and E. Cohen. Interactive Design, Analysis, and Illustration of Assemblies. In *Proc. of 1995 Symposium on Int. 3D Graphics*, pages 27–34, 1995.
4. J. L. Ellis, G. Kedem, T. C. Lyerly, D. G. Thielman, R. J. Marisa, J. P. Menon, and H. B. Voelcker. The Raycasting Engine and Ray Representations. In *Proceedings of Symposium on Solid Modeling Foundations and CAD/CAM Applications*, pages 255–267, 1991.
5. G. Farin. *Curves and Surfaces for Computer Aided Geometric Design: A Practical Guide*. Academic Press Inc., 1990.
6. H. Fuchs and J. Poulton et. Pixel-Planes 5: A Heterogeneous Multiprocessor Graphics System using

Processor-Enhanced Memories. In *Proc. of ACM Siggraph*, pages 79–88, 1989.

7. G. Georgiannakis and C. Houstis et. al. Description of the Adaptive Resource Management Problem, Cost Functions and Performance Objectives. *Technical Report TR 130, The Institute of Computer Science, Foundation for REsearch and Technology, Hellas, 1995*.

8. J. Goldfeather, S. Molnar, G. Turk, and H. Fuchs. Near Real-Time CSG Rendering using Tree Normalization and Geometric Pruning. *IEEE Computer Graphics and Applications*, 9(3):20–28, 1989.

9. R. Hendrickson and R. Leland. *A Multilevel algorithm of Partitioning Graphs*. In Proc. of Supercomputing '95, 1995.

10. C.M. Hoffmann. *Geometric and Solid Modeling*. Morgan Kaufmann, San Mateo, California, 1989.

11. G. Kedem and J.L. Ellis. The Ray-Casting Machine. In *Parallel Processing for Computer Vision and Display*, pages 378–401, Springer-Verlag, 1989.

12. M. Kelley and K. Gould et. al. Hardware Accelerated Rendering of CSG and Transparency. In *Proc. of ACM Siggraph*, pages 177–184, 1994.

13. G.A. Kriezis, N.M. Patrikalakis, and F.E. Wolter. Topological and Differential Equation Methods for Surface Intersections. *Computer-Aided Design*, 24(1):41–55, 1990.

14. S. Krishnan and D. Manocha. An Efficient Surface Intersection Algorithm based on the Lower Dimensional Formulation. *ACM Trans. on Computer Graphics*, 16(1):74–106, 1997.

15. S. Krishnan and D. Manocha. Efficient Representations and Techniques for Computing B-rep's of CSG models with NURBS primitives. In *Proceedings of CSG'96*, pages 101–122. Information Geometers Ltd, 1996.

16. S. Kumar, D. Manocha, and A. Lastra. Interactive display of large scale NURBS models. In *Proc. of ACM Interactive 3D Graphics Conference*, pages 51–58, 1995.

17. L. Lamport A fast mutual exclusion algorithm. *In ACM Transactions on Computer Systems*, 5(1):1–11, 1987.

18. M.C. Lin. *Efficient Collision Detection for Animation and Robotics*. PhD thesis, Department of Electrical Engineering and Computer Science, University of California, Berkeley, December 1993.

19. M. Mantyla. *An Introduction to Solid Modeling*. Computer Science Press, Rockville, Maryland, 1988.

20. M. Mantyla and M. Ranta. Interactive solid modeling in HutDesign. In *Proceedings of Computer Graphics, Tokyo*, 1986.

21. M. Mantyla and M. Tamminen. Localized set operations for Solid Modeling. In *Computer Graphics*, volume 17, pages 279–288, 1983.

22. D. J. Meagher. The Solids Engine: A Processor for Interactive Solid Modeling. In *Proceedings of Nicograph*, 1984.

23. J. Menon. *Constructive Shell Representations for Freeform Surfaces and Solids*. PhD thesis, Dept. of Computer Science, Cornell University, 1992.

24. M. Mine. Working in a Virtual World: Interaction Techniques used in the Chapel Hill Immersive Modeling Program. Technical report, Department of Computer Science, UNC Chapel Hill, TR96-029, 1996.

25. M.J. Muuss and L. A. Butler. Combinatorial Solid Geometry, Boundary Representations and Non-manifold geometry. In D. Rogers and R. Earnshaw, editors, *Advanced Computer Graphics Techniques*. Springer-Verlag, 1991.

26. B. Naylor. Interactive Solid Geometry via Partitioning Trees. In *Proc. of Graphics Interface*, pages 11–18, 1992.

27. A.A.G. Requicha and J.R. Rossignac. Solid Modeling and Beyond. *IEEE Computer Graphics and Applications*, pages 31–44, September 1992.

28. A.A.G. Requicha and H.B. Voelcker. Boolean Operations in Solid Modeling: Boundary Evaluation and Merging Algorithms. *Proceedings of the IEEE*, 73(1), 1985.

29. A. Rockwood, K. Heaton, and T. Davis. Real-time Rendering of Trimmed Surfaces. In *Proceedings of ACM Siggraph*, pages 107–17, 1989.

30. J. Rossignac, A. Megahed, and B.D. Schneider. Interactive Inspection of Solids: Cross-sections and Interferences. In *Proceedings of ACM Siggraph*, pages 353–60, 1992.

31. J. Rossignac and J. Wu. Correct Shading of Regularized CSG Solids using a Depth-Interval Buffer. In *Eurographics Workshop on Graphics Hardware*, 1990.

32. S. Roth. Ray Casting for Modeling Solids. *Computer Graphics and Image Processing*, 18(2):109–44, 1982.

33. T.W. Sederberg and T. Nishita. Geometric Hermite Approximation of Surface Patch Intersection Curves. *Computer Aided Geometric Design*, 8:97–114, 1991.

34. R. Seidel. Linear Programming and Convex Hulls made easy. In *Proc. 6th Ann. ACM Conf. on Computational Geometry*, pages 211–215, Berkeley, California, 1990.

35. Kevin J. Weiler. Edge-based Data Structures for Solid Modeling in Curved-Surface Environments. *IEEE Computer Graphics and Applications*, 5(1):21–40, January 1985.