ACM Symposium on Solid Modeling and Applications (2004)
G. Elber, N. Patrikalakis, P. Brunet (Editors)

Progressive Dimension-Independent Boolean Operations

A. Paoluzzi[1] and V. Pascucci[2] and G. Scorzelli[1]

[1] Dip. di Informatica e Automazione, Università Roma Tre, Via della Vasca Navale 79, 00146 Roma, Italy
[2] Center for Applied Scientific Computing, Lawrence Livermore National Laboratory, Livermore, CA 94551, USA

Abstract

This paper introduces a new progressive multi-resolution approach for representating and processing polyhedral objects of any dimension. Our representation, a variant of BSP *trees [Nay90] combined with the Split scheme introduced in [BP96], allows progressive streaming and rendering of solid models at multiple levels of detail (*LOD*). Boolean set operations are computed progressively by reading in input a stream of incremental refinements of the operands. Each refinement of the input is mapped immediately to a refinement of the output so that the result is also represented as a stream of progressive refinements. The computation of complex models results in a tree of pipelined processes that make continuous progress concurrently, so that coarse approximations of the final results are obtained nearly instantly, long before the input operands are fully processed. We demonstrate the practical effectiveness of this approach with models constructed with our prototype system.*

Categories and Subject Descriptors (according to ACM CCS): I.3.5 [Computational Geometry and Object Modeling]: Geometric algorithms, languages, and systems

1. Introduction

A progressive multi-resolution representation of polyhedra of any dimension is discussed in this paper. The goal is to represent smooth objects by a limiting series of refined polyhedral approximations. Our representation is a quite simple variation of standard BSP trees [Nay90], and allows to progressively stream and display a curved solid object at different levels of detail, as well as to progressively compute Boolean set operations, so that the computation proceeds by streaming subsequent refined LOD representations of the result. A stratification of a BSP tree into an ordered family of subtrees starting from the root is used, that flows through a computational pipeline where only the portion of the tree between two *frontiers* giving different levels of detail is actually stored and processed in each time step. The proposed approach also enables a pipelined and progressive computation of Boolean operations over a distributed system.

Standard BSP trees are normally used as spatial indexes to efficiently represent and traverse polyhedral point sets and cell decompositions, and do not store topology information. A different approach is assumed here, in order to efficiently update the geometry of solid cells at every level of detail. We locally combine the BSP representation of bounded cells with the *Split* representation [BP96], based on the incidence lattice of faces of a set of polytopal cells, that is an incremental method used to progressively discover and update both the geometry and the topology of each cell split by a hyperplane. The domain of our scheme contains, but is not limited to, the set of CSG expressions of convex primitives, enriched with further operators, like Cartesian product, projection and affine transformations. When computing a Boolean operation between large-scale objects, the result with the Naylor's algorithm [NAT90] is only generated at the end of the whole computation, and may require an intolerable time with large-scale models. With our approach, a continuously refined estimate of the Boolean result is available from the very beginning. If the output appears unsatisfactory, the task can be stopped without waiting for termination. The computation can be also terminated using a local threshold for the approximation error, possibly depending on the viewer position.

Boolean operations with solid models at interactive rates have been attracting a great research effort in recent years, using techniques based on convex sets, voxel and slice based discretization, and oriented points. Rappoport and Spitz [RS97] present a method for interactive display of CSG models that enables the user to interactively modify the affine transformations associated with CSG subobjects, using graph re-writing techniques, geometric algorithms on convex objects and a built-in hierarchical acceleration scheme. CSG modeling and rendering is achieved at interactive rates by Chen and Fang [CF99] by interactively generating a volume representation of a CSG model in 3D texture memory. A volume scene tree is used by Liao and Fang [LF02] where each leaf node represents an input dataset or synthetic geometric model, and each interior node represents an operator such as blending or filtering. Their algorithm uses a pipeline for volume rendering by sweeping a volume slice. Adams and Dutré [AD03] perform interactive Boolean operations on free-form solids bounded by surfels, oriented points in 3D space, that approximate the local orientation of the surface they represent.

The method discussed in the present paper, according with the standard use of BSP trees as solid representations, only works when

© The Eurographics Association 2004.

the intrinsic dimension of the object, say its number of coordinates in a chart, is equal to the dimension of the embedding space. A possible extension of this dimension-independent method to non regular polyhedra using ternary trees would allow to computing the semi-algebraic set that satisfies a Boolean formula representing a complex query concerning scalar, vector or tensor fields over manifolds.

2. Outline of a progressive geometric framework

The aim of this section is to outline a computational framework for progressive polyhedral approximations of geometric expressions, that produce polyhedral point-sets of arbitrary dimension d when evaluated. The set of tasks encoding a symbolic description of the computation can be visualized as a graph (see Figure 1), and the computational model closely resembles a data-flow diagram, where processes transform the incoming data flow into the outgoing data flow. The geometric computing provided by our prototype kernel supports several geometric operators, including affine transformations, Boolean operations, Cartesian products, primitive generators and others. In the remainder of this paper we discuss our approach to *progressive Booleans* using progressive BSP trees, later abbreviated as PBSP.

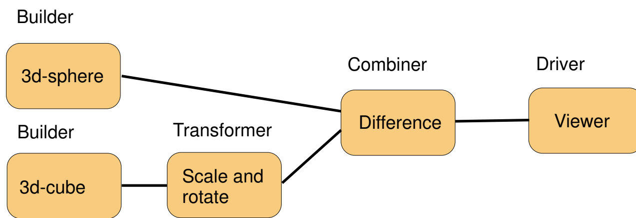

Figure 1: *Task diagram of the computation*

Representations For this purpose we use an "ad hoc" variation of the *double representation* of polyhedral sets. It is well known that a convex polyhedron can be represented either as a system of linear inequalities (*on-face* rep) or as the convex hull of extremal points (*on-vertex* rep) [Zie95, BFM98, Pao03]. In our case we enforce the hyperplane representation by using a progressive type of BSP-*trees*, while making also use of a rich representation (describing both topology and geometry) of the face lattice of the associated *weak complex*, supported by the *Split* data structure and algorithm [BP96], and discussed in the next section.

Tasks The main components of the proposed framework are four types of processes, that we call *builders*, *transformers*, *combinators* and *drivers*, respectively. Such components either generate, transform or combine PBSP-trees, and maintain a suitable internal state, depending on the work they must perform. A useful classification of processes may be given depending on the cardinality of their i/o connections. In particular:

1. A **builder** is a task with no input and one output. It generates progressive polyhedral approximations, at subsequent levels of detail, of some specific type of geometric object. Presently we have dimension-independent builders for parallelepipeds, simplexes, cylinders, cones, spheres and toruses.
2. A **transformer** is a task with one input and one output. Our current transformers either apply affine transformations (dimension-independent rotation, translation, scaling or shearing) or produce the complement, projection, or extrusion of their input.
3. A **combiner** is a task with more than one input and a single output. These tasks combine the input trees to produce the output tree. In particular, we have n-ary combiners for Boolean set union, intersection, difference and symmetric difference (i.e. xor), as well as for Cartesian product of polyhedral point sets.
4. A **driver** is a task with one input and no output. The status of a driver contains a *Split* data structure, and allows to compute model properties and to visualize it. Based on the computed properties, a driver task is able to decide which cells of its input should be expanded, i.e. further detailed. The decision criteria may concern suitable ratios of volumes of cells, the position of the viewer, the approximation error, and so on. In some sense, a driver task is the container of a progressively refined geometric data base.

3. Background

We shortly recall here the definitions and main properties of the reference dimension-independent data structures used by our progressive approach, say the BSP decompositive representation and the *Split* data structure storing the topology and geometry of the cell decomposition.

3.1. Binary space partition

Binary Space Partitioning (BSP) of objects support a "decompositive" representation, where a solid model is partitioned into a set of cells. More precisely, BSP trees are binary trees where each internal node has an associated boundary hyperplane of the model, and each leaf node represents a convex cell — which may be either full or empty — in the space partition induced by such a set of hyperplanes. Given a set of hyperplanes in d-dimensional Euclidean space, a BSP-tree on such hyperplanes establishes a hierarchical partitioning of E^d.

Each node ν of such a binary tree represents a convex and possibly unbounded cell of E^d denoted as R_ν. The two children of an internal node ν are denoted as *below*(ν) and *above*(ν), respectively. Leaves correspond to unpartitioned cells, which are called *empty* (OUT) or *full* (IN) *cells*. Each internal node ν of the tree is associated with a *partitioning hyperplane* h_ν, which intersects the interior of R_ν. The hyperplane h_ν partitions R_ν into three subsets: (a) the cell $R^0_\nu = R_\nu \cap h_\nu$ of dimension $d-1$; (b) the cell $R^-_\nu = R_\nu \cap h^-_\nu$, where h^-_ν is the negative half-space of h_ν; (c) the cell $R^+_\nu = R_\nu \cap h^+_\nu$, where h^+_ν is the positive half-space of h_ν. For each node ν in a BSP tree, the cell R_ν is the intersection of the closed half-spaces on the path from the root to ν. The cell described by the ν node is

$$R_\nu = \bigcap_{e \in E(\nu)} h^\pm_e \qquad (1)$$

where $E(\nu)$ is the edge set on the path from the root to ν and $h^\pm_e$ is the half-space associated with the e edge.

© The Eurographics Association 2004.

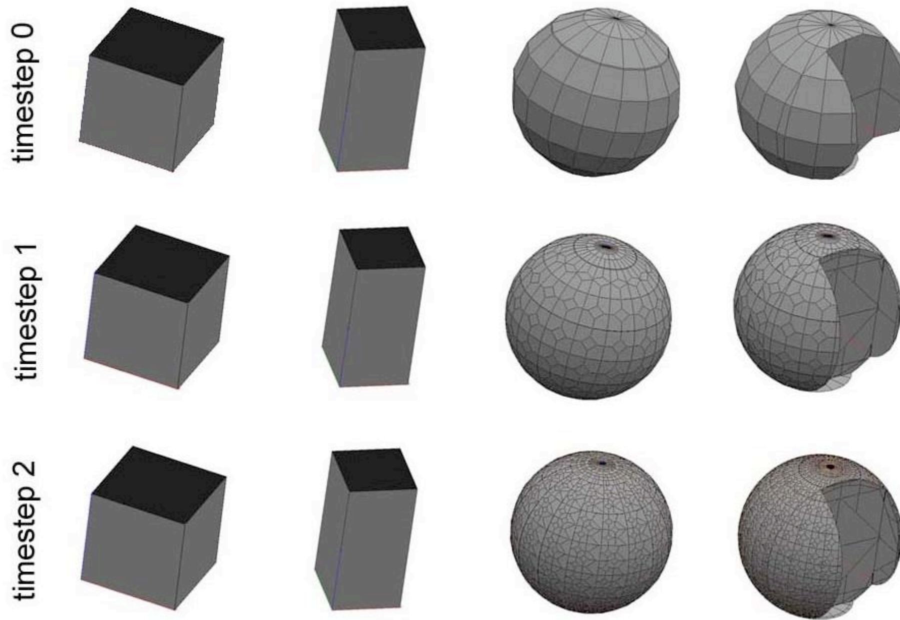

Figure 2: *Progressive computation of different levels of detail of a 3D model*

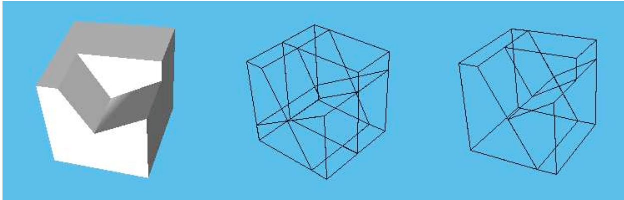

Figure 3: *(a) Set difference between two cubes represented as* BSP *trees; (b) cells of the space partition induced by the boundary planes; (c) full (*IN*) cells after* tree pruning

3.2. Split representation

In our approach to progressive combination and visualization of geometric expressions, we make use of the representation of polytopal cell complexes [Zie95] introduced by Bajaj and Pascucci and named *Split* data structure [BP96]. This representation allows one to solve in a localized way the problem of splitting such kind of complex with a hyperplane.

In particular, the set $\mathcal{F}(c)$ of faces of a c polytope, including c and $\emptyset$, is a lattice, partially ordered with respect to the inclusion relation. The polytopal cell complex generated by a pruned BSP tree (see Figure 3c) is called *weak complex* by [BP96], and defined as follows:

Definition 3.1 (Weak complex) *A weak complex $\mathcal{C}$ is a set of polytopes, such that if $c \in \mathcal{C}$ then $\mathcal{F}(c) \subseteq \mathcal{C}$, and, for each pair $c, d \in \mathcal{C}$, it is either $c \cap d = \emptyset$ or*

$$c \cap d = \partial c \cap \partial d = f_1 \cup \cdots \cup f_k,$$

where $\{f_1, \ldots, f_k\} = \mathcal{F}(c) \cap \mathcal{F}(d)$.

Closure of splitting The set of weak complexes is closed with respect to the *Split*ting of a polytopal cell with a hyperplane. This *local splitting* property does not hold for standard cell complexes. It is easy to show that the set of cells of the space partition produced by a BSP-tree is a weak complex (see Figure 3c). A weak complex, in particular, is generated when a BSP-tree is *pruned* according to the Naylor's approach. If pruning is not allowed, a standard polytopal complex[†] is associated to the tree, but every standard polytopal complex is also a weak complex. In other words, the set of BSP-trees and the set of weak complexes are isomorphic.

Weak complex representation Using the *Split* data structure [BP96], a weak complex $\mathcal{C} = \{c_i\}$ can be represented as an undirected graph $\mathcal{S}(\mathcal{C}) = (N, A)$, where

$$N = \bigcup_{c_i \in \mathcal{C}} \mathcal{F}(c_i),$$

and

$$A = \{(c_i, c_j) \mid c_i \prec c_j \vee c_j \prec c_i\}$$

where $c_i \prec c_j$ means that c_i is *directly contained* within the boundary ∂c_j of c_j, i.e. does not exists another face c_k such that $c_i \subset \partial c_k$ and $c_k \subset \partial c_j$. Notice that, for each $(c_i, c_j) \in A$ it is

$$|\dim c_j - \dim c_i| = 1.$$

Split algorithm

The aim of the generic step of the *Split* algorithm is to subdivide the face complex $\mathcal{F}(c)$ of a c polytope with a single h hyperplane.

† Where the intersection of any pair of cells is either empty or is a face for both.

© The Eurographics Association 2004.

As in the original formulation [BP96], let us consider two different cases:

(1) the splitting hyperplane does not contain any of c vertices, and, as a consequence, does not contain any of higher-dimensional faces of c;
(2) the splitting hyperplane does contain some of c vertices.

In the (1) case, the splitting of a d-cell c is quite simple and robust. It can be summarized as follows: (a) classify the c vertices with respect to the h subspaces, thus adding either a $+$ or a $-$ label to each vertex; (b) for each k-cell f, $1 \leq k \leq d$, look at the set of $(k-1)$-cells contained in f and, if such set contains both $+$ and $-$ labels, then (i) split f and substitute it by its parts f^+ and f^-, such that $f^+ \subset h^+$, $f^- \subset h^-$, (ii) create a $(k-1)$-cell $f^=$ such that $f^+ \cup f^= \cup f^- = f$, (iii) suitably link $f^+, f^=, f^-$ to both their super- and sub-cells.

The case (2) above is numerically unstable. Three types of labels are used for vertex classification, labeling a vertex as $v^=$ when it is contained on the h hyperplane. No problems would arise in a computation with infinite precision. Unfortunately, real computers are not infinitely precise, so that some vertex classification can be inexact. In order to recover from wrong vertex classifications and to consistently compute the split complex, further information concerning topological structure must be used.

Actually, a consistency problem only arises when some vertices are classified at the same time in each of the three classes, i.e. when both v_i^+, v_j^- and $v_k^=$ vertices contemporary exist. In fact, if vertices are only of v_i^+ and $v_i^=$ type (or v_j^- and $v_k^=$) the cell is not split.

4. Progressive trees

First we give the definitions of progressive binary space partitioning trees and some related concepts, then we introduce a convergence property to be satisfied by useful progressive algorithms. For the sake of brevity, we often denote a BSP-tree as BSP, and more in general, we omit the -tree specification, that should be understood from the context.

4.1. Definitions

In particular, a new definition of BSP, with three kind of leaves, is given below.

Definition 4.1 (BSP) *A BSP is a complete binary tree, made by three different types of nodes, that are labeled white, black and gray (for either OUT or IN or UNDECIDED), respectively.*

Notice that we do not enforce the usual constraint that the BSP leaves contain a final information (IN or OUT label) about the spatial region they describe. A convex cell associated to a leaf node may be not yet detailed, and in this sense is UNDECIDED. Thus we introduce the following definition.

Definition 4.2 (PBSP) *a PROGRESSIVE BSP (PBSP for short) is a non-empty sequence $\mathcal{T} = (T_0, T_1, \ldots, T_k, \ldots)$ of BSP trees ordered by reversed inclusion and with the same root, i.e. such that:*

$$T_0 \subset T_1 \subset \cdots \subset T_k \subset \cdots$$

It may be useful to see the index $k \in \{0,1,2,\ldots\}$ as a discrete time index.

Definition 4.3 (Frontier) *the k-*frontier *of the $\mathcal{T}$ PBSP, denoted as $\mathcal{F}(T_k)$, is the set of leaves of the T_k BSP.*

Notice that every k-frontier, $k \in \{0,1,2,\ldots\}$, is a weak complex, and that $\mathcal{F}(T_{k+1})$ is a *refinement* of $\mathcal{F}(T_k)$, for every k, where some UNDECIDED cell is split.

Definition 4.4 (Support) *the* support *$|\mathcal{F}(T)|$ of the BSP T is the point-set union of the cells R_ν, associated to the leaves $\nu \in \mathcal{F}(T)$. In formal terms*

$$|\mathcal{F}(T)| = \bigcup_{\nu \in \mathcal{F}(T)} R_\nu,$$

where R_ν is defined by (1).

Making use of the Requicha's definition [Req80] of a *representation scheme* as a mapping $s: M \to R$ between a set of models M and a set of representations R , we distinguish between $m \in M$ as a point set, and its computer representation. In our approach, the model m is approximated by the limit, for $k \to \infty$, of the support sets of the frontiers of a PBSP. In this sense, a *progressive scheme* is an ordered parametrization of a set of maps $M \to R$. More formally, we can write:

$$s_k(m) := \mathcal{F}(T_k)$$

such that

$$m = \lim_{k \to \infty} |s_k(m)|.$$

4.2. Convergent approximations

Only a subset of PBSP trees are really useful as progressive solid representations. In particular, a progressive scheme is meaningful iff it satisfies the convergence property discussed below.

By definition, each tree in a PBSP $\mathcal{T} = (T_0, T_1, \ldots, T_k, \ldots)$ gives a partition of the embedding space $\mathbb{E}^d$ with convex cells of non-zero volume in the relative topology. Let us assume that the *modeling space* $S = |T_0| \subset \mathbb{E}^d$ is bounded. In particular, assume, without loss of generality, that S is a hyper-cuboid that contains the object m to be modeled. This assumption will allow us to considered only cell decompositions T_k where all cells have bounded volume.

First notice that in such bounded modeling space S, the sum of volumes of empty, full and undecided cells is constant, and is equal to $\mathrm{vol}\, S$. Then consider a weak complex $\mathcal{C}_k := \mathcal{F}(T_k)$ that gives a partition of S at some time k. At the very beginning, with $\mathcal{C}_0 = \{c \mid c := S\}$, c is UNDECIDED, i.e. the root of T_0 is gray. By definition, in each transition between $\mathcal{F}(T_k)$ and $\mathcal{F}(T_{k+1})$, the volume of gray cells diminishes, while the sum of volumes of white and black cells increases. Notice also that the empty (OUT) and full (IN) cells are not further detailed in subsequent trees (see Figure 6). On the limit, the IN volume goes to $\mathrm{vol}\, m$, whereas the OUT volume goes to $\mathrm{vol}\,(S-m)$.

In other words, every point in $S - \partial m$, at some time k, will necessarily belong to some cell that is either IN or OUT. In this sense we may state that our progressive representation converges to the (interior of) curved object m.

© The Eurographics Association 2004.

4.3. Cell representation

With standard BSP trees, a cell R_ν of the space decomposition induced by a BSP is implicitly described by the the tree path connecting the ν node to the root. In our progressive approach, a cell of the current frontier is explicitly stored in a geometry database using the *Split* representation, and is represented by a pair of pointers to (a) its boundary geometry and to (b) the PBSP node associated to the cell. This one may be either a *leaf* or a *non-leaf* node. In the first case it is characterized by a label in the set $\{\text{IN}, \text{OUT}\}$; in the second one it is a triple $t_\nu = (h, t_\nu^-, t_\nu^+)$, where h is the hyperplane splitting R_ν, and t_ν^-, t_ν^+ are the pointers to the trees decomposing R_ν^- and R_ν^+, respectively. Formally, we can write:

$$\begin{aligned} \langle\text{tree}\rangle &::= \langle\text{leaf}\rangle \mid \langle\text{non-leaf}\rangle \\ \langle\text{leaf}\rangle &::= \text{IN} \mid \text{OUT} \mid \text{UNDECIDED} \\ \langle\text{non-leaf}\rangle &::= (h, \langle\text{tree}\rangle^-, \langle\text{tree}\rangle^+) \end{aligned}$$

5. Progressive Booleans

The supported operations are *union*, *intersection*, *difference*, *xor* (symmetric difference) and *complement*. The unary complement is trivial and just requires to change the leaf labels from IN to OUT and vice-versa. The other operations are n-ary, i.e. may be applied to an arbitrary number of input objects. The n-ary difference is defined as

$$\begin{aligned} &\text{difference}(a,b,c,\ldots) \\ &\quad := \text{difference}(a, \text{union}(b,c,\ldots)) \end{aligned}$$

5.1. Preview

The proposed algorithm for Boolean combination of PBSP trees works by recursively *merging* the roots of its argument trees as shown by Figure 4. The basic case of recursion, i.e. the *termination* condition, depends on the value of the executed

$$\langle op \rangle \in \{\text{union}, \text{intersection}, \text{difference}, \text{xor}\}.$$

A further *simplification* step allows to speed-up the algorithm, and also depends on the executed operation. The simplification and termination conditions are discussed in the following subsection. Then we give a formal description of the whole algorithm. It is very important to understand that not all the generated nodes are expanded, but only those requested by the *driver* task, that knows the geometry associated to each node, and solves the decision problem if its cell should be either split or not, depending on the involved linear inequalities .

There are two main differences with the standard BSP approach [NAT90]. First of all, the combinatorics of the Boolean algorithm is completely separated from the representation of topology and geometry. Furthermore, the construction and detailing of the space decomposition proceeds by (subsets of) levels of the resulting PBSP tree, instead than in a *depth first search* way.

5.2. Termination and simplification rules

First of all, we distinguish between the simple cases corresponding to (a) the basic recursion case and (b) the simplification of input arguments, and (c) the combinatorial step.

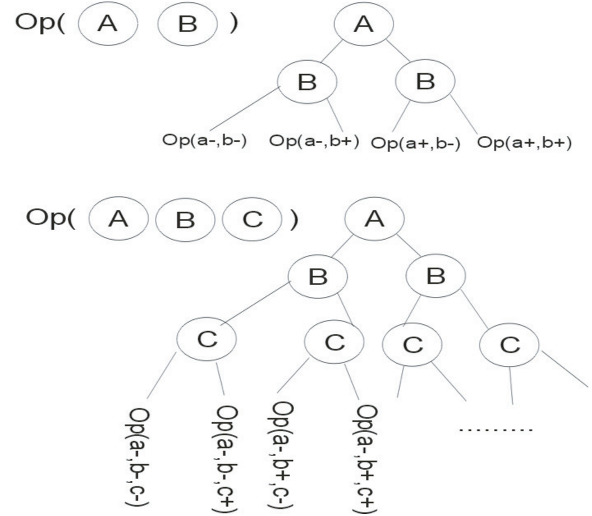

Figure 4: *Complete binary trees produced by the combination of 2 or 3 nodes*

Basic case of recursion

It is pretty simple, and consists in checking the $\langle op \rangle$ arguments looking for values inducing the algorithm termination. A termination is induced when either a IN or a OUT node may be produced. Consider the operation

$$\langle op \rangle(a_1, a_2, \ldots, a_n)$$

where $a_1, a_2, \ldots, a_n$ are BSP trees. Then we have:

Union If at least one of the arguments $a_1, a_2, \ldots, a_n$ is IN, then return a IN node;

Intersection If at least one of the arguments $a_1, a_2, \ldots, a_n$ is OUT, then produce a OUT node;

Difference If at least one of the arguments $a_2, \ldots, a_n$, but not a_1, is IN, then produce a IN node; if the a_1 argument is OUT, then the result is also OUT.

Two "special" cases require a transformation of the operation $\langle op \rangle$ into an operation $\langle op' \rangle$:

Xor If the a_i argument is IN, then

$$\begin{aligned} &\text{XOR}(a_1, a_2, \ldots, a_i, \ldots, a_n) \rightarrow \\ &\quad \text{COMPLEMENT}(\text{UNION}(\\ &\quad\quad a_1, a_2, \ldots, a_{i-1}, a_{i+1}, \ldots, a_n)) \end{aligned}$$

Difference If the a_1 argument is IN, then

$$\begin{aligned} &\text{DIFFERENCE}(a_1, a_2, \ldots, a_n) \rightarrow \\ &\quad \text{COMPLEMENT}(\text{UNION}(a_2, \ldots, a_n)) \end{aligned}$$

Simplification of arguments

In this step we search the $\langle op \rangle$ arguments, to see if some of them do not change the result, so that they can be simplified. For the sake of space, we give the simplification rules without proof.

Union Simplify the arguments with OUT value; if all are simplified, then the output node is OUT;

© The Eurographics Association 2004.

Intersection Simplify the IN arguments; if all the arguments are simplified, then the output node is IN;

Difference Simplify the OUT arguments, but not the first one; if only the first argument is not simplified, then return a copy of the first argument;

Xor Simplify the arguments with OUT value; if all are simplified, then the resulting node is OUT.

5.3. Algorithm

As we already said in the preview section, the *Combine* algorithm to return a specified Boolean combination $\langle op \rangle$ of BSP trees $a_1, \ldots, a_n$ is very simple, and correspond to orderly execute (a) a check for termination, (b) a simplification of arguments, and (c) the recursive merging of the input trees, calling itself on the leaf nodes of the output BSP *tree*. The *build_merge* function just combinatorially produces a complete binary tree of depth n starting from n nodes, without taking into account neither the operation $\langle op \rangle$ to execute nor if the current node is incompatible (does not have solutions). The merging process is graphically displayed in Figure 4; the *Combine* algorithm is given in Figure 5.

5.4. Geometric data base

A *driver* process, at the end of the computational pipeline, produces an *expanded* binary tree to store the topological and geometrical data structure, based on the frontier of the progressively generated BSP tree and on a lattice-based *Split* data structure of the cell decomposition. At the same time the driver *pulls* the computation, by deciding what UNDECIDED nodes of the current PBSP frontier to expand and make more detailed.

In particular, the *Split* algorithm starts with an input tree constituted by the only root node. the bounding box $B := [min - real, max - real]^d$ of the whole space $\mathbb{E}^d$ is initially stored within the node, so that we have:

$$tree = (root, B)$$

When the root node has been completed by the other pipeline processes as

$$root := non_leaf(h, tree^-, tree^+)$$

then it become possible for the *Split* process to really subdivide the B cell according to the h hyperplane, so generating the B^- and the B^+ cells, as well as their lower-dimensional faces, and to store B^- with $tree^-$ and B^+ with $tree^+$. At this point it is possible to continue recursively the computation on the expanded nodes $(tree^-, B^-)$ and $(tree^+, B^+)$.

The basic cases of the recursion clearly deal with IN and OUT cells. A IN cell should not be (usually) further split, and can be used for either visualization purposes or other computational tasks. When a cell is OUT, its expanded node is not used anymore, and the corresponding store may be garbage collected. Notice that the computation may proceed in parallel by using a distributed pipeline, either multi-process or multi-host.

6. Examples

A progressive Boolean result is only produced when combining two or more progressively generated arguments. Hence we provided some progressive constructors, including the generator of the d-dimensional sphere discussed below. The generation of d-cylinders is by Cartesian product of a $(d-1)$-sphere times a 1D interval complex, toruses are produced by product of circles followed by projection, and so on.

Progressive d-sphere

The d-dimensional sphere S_d of unit radius can be generated by applying d reflections about coordinate hyperplanes followed by unions[‡], to the *circular sector* described by the chart (U_d, ϕ_d), where

$$U_d = \{x \geq 0 \mid (x-0)^2 \leq 1\} \subset \mathbb{E}^d$$

and

$$\phi_d : U_d \rightarrow [0,1] \times \left[0, \frac{\pi}{2}\right)^{d-1},$$

with ϕ_d^{-1} defined inductively as

$$\phi_2^{-1}(r, \alpha_1) = r\left(\cos\alpha_1 \begin{bmatrix} 1 \\ 0 \end{bmatrix} + \sin\alpha_1 \begin{bmatrix} 0 \\ 1 \end{bmatrix}\right),$$

$$\phi_d^{-1}(r, \alpha_1, \ldots, \alpha_{d-1}) = r\left(\cos\alpha_{d-1} \begin{bmatrix} x_{d-1} \\ 0 \end{bmatrix} + \sin\alpha_{d-1} \begin{bmatrix} 0_{d-1} \\ 1 \end{bmatrix}\right),$$

where $x_{d-1} = \phi_{d-1}^{-1}(1, \alpha_1, \ldots, \alpha_{d-1}) \in \partial U_{d-1}$, and where 0_{d-1} is the origin of $\mathbb{E}^{d-1}$.

Three snapshots of the union of translated 2D spheres with same radius are given in Figure 6. The first picture shows with different gray tones the IN and UNDECIDED cells of the union after few progressive steps. The second and third pictures show the refined models after some more steps. Notice that the IN cells previously computed are not further detailed, ie that the computation proceeds only within the undecided region.

Other examples

Other snapshots of the progressive generation of the classic CSG example given by the difference and union of a cube with three orthogonal cylinders are shown in Figure 7.

The ray-traced and textured rendering of approximations at different levels of detail of a CSG "temple" made by union and differences of parallelepipeds and cylinders are given in Figure 8. It may be interesting to notice that our progressive method may automatically generate LOD nodes of virtual environments.

In Table 1 we show the timing of the three examples given in this paper, executed on a IBM Intellistation M-Pro Pentium 4 3GHz. Each record refers to the time spent to compute the progressive geometry of the models, without considering the display time.

‡ In the actual implementation, set unions are substituted by tree merges, since the various parts of the assembly are quasi-disjoint.

© The Eurographics Association 2004.

```
Algorithm Combine (⟨op⟩, a_1:BSP, a_2:BSP, ..., a_n:BSP)
  BspTree tree;

    # STEP 1: check for termination ...
    for  (i=0; i<n; i++)  do
      if  (termination(⟨op⟩, i, a_i)) then
        # compute leaf depending on ⟨op⟩ and on arguments
        ...
        return tree;
      fi
    done

    # STEP 2: simplification of arguments ...
    for (i=0; i<n; i++)  do
      if  (simplification(⟨op⟩, i, a_i)  then
        return  Combine(⟨op⟩,a_1,a_2,...,a_{i-1},a_{i+1},...,a_n)
      fi
    done

    # STEP 3: merge of nodes; recursively generates new tree levels
    tree = build_merge(a_1,a_2,,...a_n)
    tree.leaf[0] = Combine(⟨op⟩,a_1^-,a_2^-,...,a_n^-)
    tree.leaf[1] = Combine(⟨op⟩,a_1^-,a_2^-,...,a_n^+)
    ...
    tree.leaf[2^n - 1] = Combine(⟨op⟩,a_1^+,a_2^+,...,a_n^+)
    return  tree
  end
```

Figure 5: *Algorithm for progressive Boolean combination of* PBSP *trees*

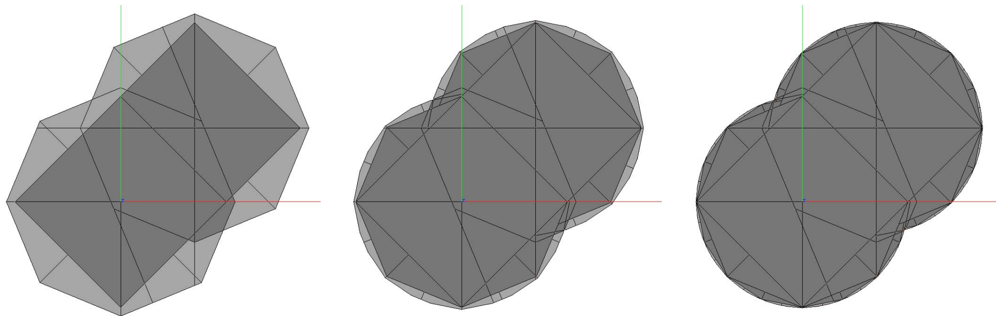

Figure 6: *Progressive union of 2D spheres at different resolutions. Both* IN *and* UNDECIDED *cells are displayed*

7. Conclusion

In this paper a progressive approach to Boolean operations on polyhedra of any dimension is proposed, using a pipelined version of standard BSP, named PROGRESSIVE BSP tree, which takes into account the combinatorial aspect of the Boolean operations, and stores a *Split* representation of topology and geometry of the corresponding *weak* polytopal complexes. The computation may proceed in parallel, by using a distributed pipeline, either multiprocess or multi-host.

The approach is very different from the Naylor's one. No geometric comparison of hyperplanes is executed to decide if either splitting or not the current node. Furthermore, the *Combine* algorithm knows nothing about the object geometry. It only combines some labeled tree nodes according to the rules we discussed in the previous section. This approach has several advantages. In particular, the driver process may pull the computation by deciding about what nodes to expand, and with what resolution, depending on special decision rules, e.g. on the viewpoint position and distance from the cell.

The main open problem concerns the geometric robustness of the approach. Since the undecided cells are more and more splitted, and their volume continuosly decreases, the computation of degenerate splitted cells due to floating errors becomes more common. Our current efforts are devoted to cope with topological inconsistencies and to find new builders. A further problem concerns the production of local artifacts corresponding to UNDECIDED cells, usually

© The Eurographics Association 2004.

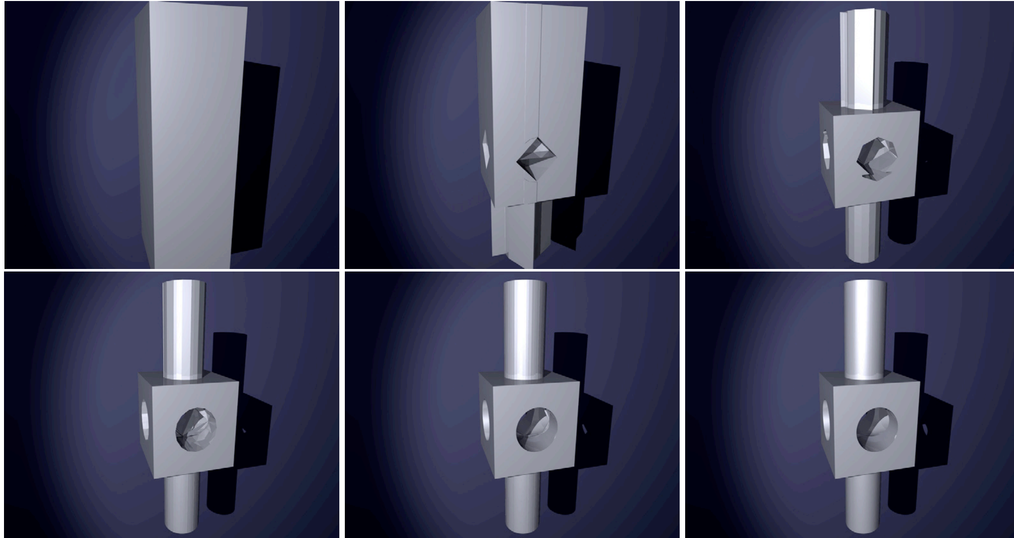

Figure 7: *Progressive construction of the mechanical joint with perpendicular cylindrical holes*

Table 1: *Timings of the progressive execution of given examples. (a) 2d circle union (b) mechanical part (c) temple*

	(a)		(b)		(c)
1	0.0010	1	0.0005	1	0.0640
2	0.0025	2	0.0300	2	0.2105
3	0.0050	3	0.1035	3	0.3550
		4	0.2350	4	0.7105
		5	0.3205	5	1.2040
		6	0.6150	6	4.1035

displayed as IN cells. Such visual effect clearly decreases as the computation goes on.

The approach introduced here is the first prototype implementation of a new distributed kernel to be used by the `PLaSM` language for symbolic geometric design [PPV95, Pao03]. Such prototype kernel already supports several geometric operators, including affine transformations, Boolean operations, Cartesian products, primitive generators and others. Before being appropriate as a general purpose geometry engine, it should also support *embedded* polyhedra of any dimension by using ternary space partition trees, where cells are intersection of linear equalities and inequalities, and progressive polyhedral approximation of curved manifold, probably by using the new Peters' efficient technique, named *sleeve*, for linearizing curved spline geometry [Pet03]. In few world, our mid-term goal is to implement a progressive geometry engine that should be able to refine with increasing detail semi-algebraic multi-dimensional curved manifolds embedded in higher-dimensional spaces.

References

[AD03] ADAMS B., DUTRŐ P.: Interactive boolean operations on surfel-bounded solids. *ACM Transactions on Graphics (TOG) 22*, 3 (2003), 651–656. 1

[BFM98] BREMNER D., FUKUDA K., MARZETTA A.: Primal-dual methods for vertex and facet enumeration. *Discrete and Computational Geometry 20* (1998), 333–357. 2

[BP96] BAJAJ C. L., PASCUCCI V.: Splitting a complex of convex polytopes in any dimension. In *Proceedings of the twelfth annual symposium on Computational geometry* (1996), ACM Press, pp. 88–97. 1, 2, 3, 4

[CF99] CHEN H., FANG S.: A volumetic approach to interactive csg modeling and rendering. In *Proceedings of the fifth ACM symposium on Solid modeling and applications* (1999), ACM Press, pp. 318–319. 1

[LF02] LIAO D., FANG S.: Fast volumetric csg modeling using standard graphics system. In *Proceedings of the seventh ACM symposium on Solid modeling and applications* (2002), ACM Press, pp. 204–211. 1

[NAT90] NAYLOR B. F., AMANATIDES J., THIBAULT W.: Merging BSP trees yields polyhedral set operations. *Computer Graphics 24*, 4 (Aug. 1990), 115–124. Proc. of ACM Siggraph'90. 1, 5

[Nay90] NAYLOR B. F.: Binary space partitioning trees as an alternative representation of polytopes. *Computer Aided Design 22*, 4 (1990), 250–252. 1

[Pao03] PAOLUZZI A.: *Geometric Programming for Computer Aided Design*. John Wiley & Sons, Chichester, UK, 2003. 2, 8

[Pet03] PETERS J.: Efficient one-sided linearization of spline geometry. In *Mathematics of Surfaces X* (2003), Martin R., Wilson M., (Eds.), Lecture Notes in Computer Science, Springer. 8

© The Eurographics Association 2004.

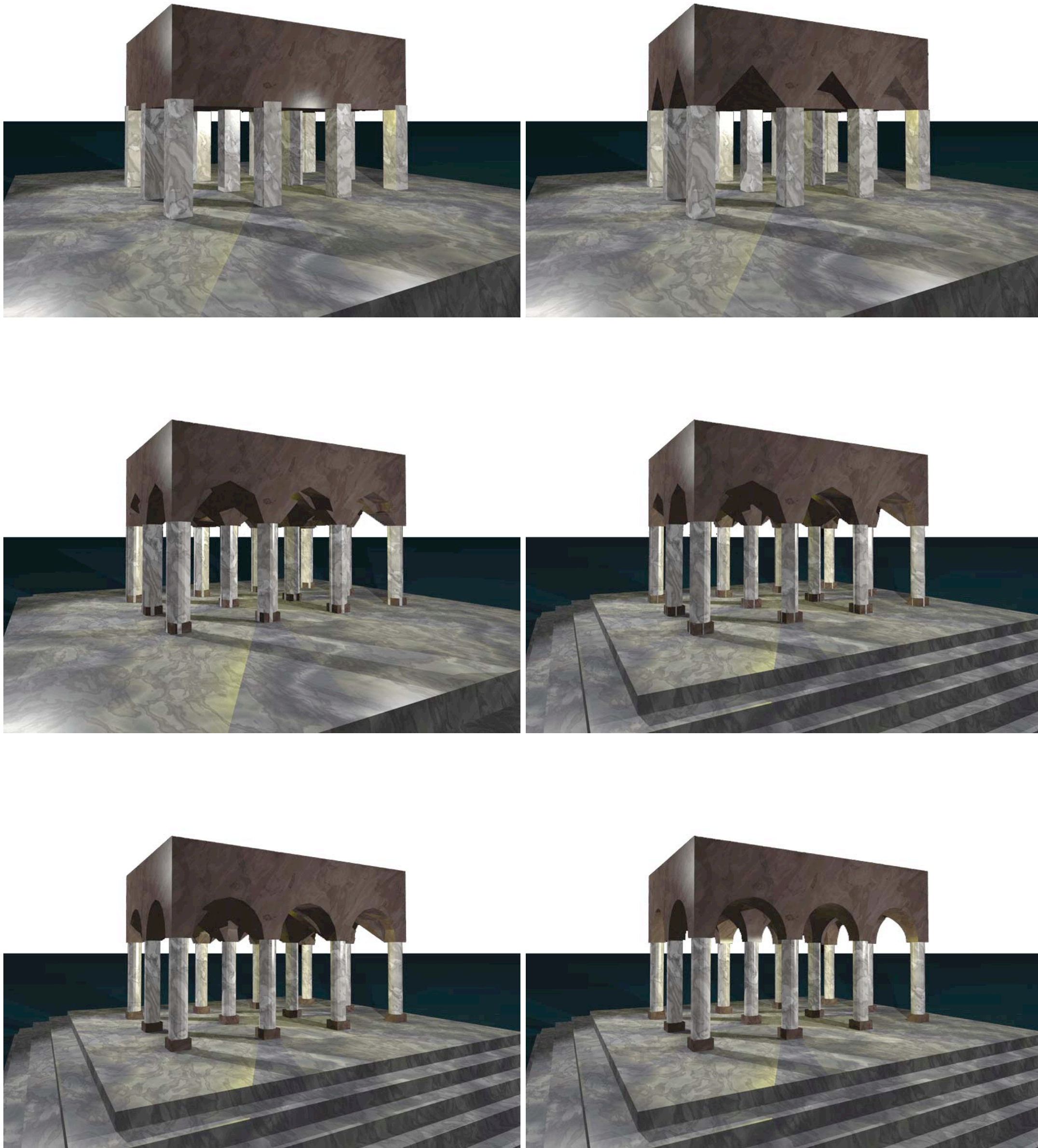

Figure 8: *Ray-traced rendering of some subsequent approximations of a progressive temple*

[PPV95] PAOLUZZI A., PASCUCCI V., VICENTINO M.: Geometric programming: a programming approach to geometric design. *ACM Transactions on Graphics 14*, 3 (July 1995), 266–306. 8

[Req80] REQUICHA A.: Representations for rigid solids: Theory, methods and systems. *ACM Computing Surveys 12*, 4 (Dec. 1980), 437–464. 4

[RS97] RAPPOPORT A., SPITZ S.: Interactive boolean operations for conceptual design of 3-d solids. In *Proceedings of the 24th annual conference on Computer graphics and interactive techniques* (1997), ACM Press/Addison-Wesley Publishing Co., pp. 269–278. 1

[Zie95] ZIEGLER G. M.: *Lectures on polytopes*. No. 152 in Graduate texts in mathematics. Springer-Verlag, New York, NY, 1995. 2, 3

© The Eurographics Association 2004.