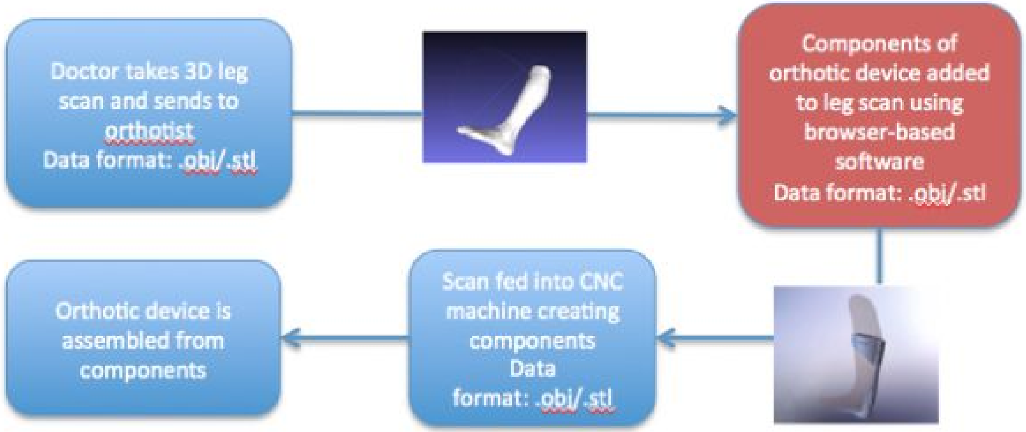
Set Operations on Polyhedra using Binary Space Partitioning Trees

Vikram Chandrashekhar
Group 16

Project Summary

Browser Based Constructive Solid Geometry for Anatomical Models

- Orthoses for cerebral palsy patients
- Fusiform developed a process to reduce waste, reduce time and increase efficiency of orthotic design/fabrication
- Currently: ~10 hour process to create orthotic in SolidWorks
- Browser based software to add pre-designed orthotic components



Paper

Thibault, W. C., & Naylor, B. F. (1987). Set operations on polyhedra using binary space partitioning trees. *ACM SIGGRAPH Computer Graphics SIGGRAPH Comput. Graph., 21*(4), 153-162. doi:10.1145/37402.37421

Goal of Paper:

- Use binary space partitioning trees (BSPTs) to perform constructive solid geometry (CSG) operations
    - Why: BSPTs are much faster and more unified than other methods to compute CSG operations

Application to Project:

- Modify current CSG package to optimize nodes of BSPT

Some terms and Definitions

- **Boundary representation (B-rep)** - A d-dimensional solid represented as a collection of (d-1)-polyhedra (also called faces) represented by (d-2)-polyhedra until d = 0
- **Binary space partitioning tree (BSPT)** - a binary tree whose non-leaf nodes are labeled with hyperplanes and whose leaf nodes correspond to cells of a partitioned d-space.
- **Cell** - area enclosed by splitting hyperplanes
- **Hyperplane** - a (d-1)-dimensional subspace in d-space (e.g. a 2D plane in 3D space but generalized to any higher-dimensional space)
- **Half-space** - either of the 2 parts into which a (hyper)plane divides a d-dimensional space

BSPT terminology

- Each internal node *v* of BSPT represents region of space R(v)
- R(v) is the intersection of open halfspaces on the path from the root to v
- Associated with partitioning hyperplane H_v
- 3 regions
  - $R(v) \cap H_v^+$
  - $R(v) \cap H_v^-$
  - $R(v) \cap H_v$
- **Sub-hyperplane** (SHp(v)) - $R(v) \cap H_v$

More formally, for a hyperplane

$$H = \{(x_1, ..., x_d) | a_1 x_1 + \cdots + a_d x_d + a_{d+1} = 0\},$$

the *right* (or in B-rep parlance, the "front") halfspace of H is

$$H^+ = \{(x_1, ..., x_d) | a_1 x_1 + \cdots + a_d x_d + a_{d+1} > 0\},$$

and the *left* (or "back") halfspace of H is

$$H^- = \{(x_1, ..., x_d) | a_1 x_1 + \cdots + a_d x_d + a_{d+1} < 0\}).$$

The right side of H lies to the side of H in the direction of the hyperplane's normal, $(a_1,...,a_d)$.

Generic BSPTs

- Recursive, hierarchical partitioning of d-dimensional space
- Nodes store splitting hyperplanes
- Distinction between halfspaces determined by normal vector - arbitrary choice
- Right subtree - region lying on side pointed to by normal
- Left subtree - the other region

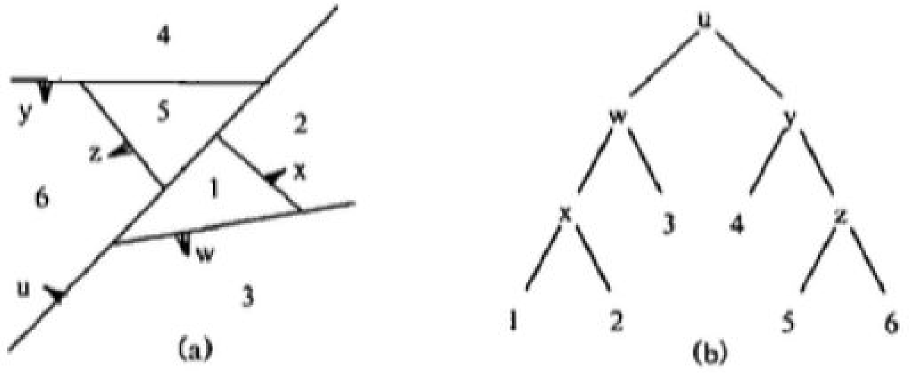


Figure BSPT. Geometry of a 2D partitioning (a) and its BSP tree (b).

B-rep -> BSPT

- Requirements
  - All points on boundary of polyhedra lie in sub-hyperplanes of the resulting tree
    - Embed faces
  - Correct classification of cells
    - “In” vs “out”
- Algorithm
  - Choose hyperplane H
  - Partition faces left of, right of, or coincident with H
    - When empty, we know that region is homogenous
  - Recursively apply to left and right face subtrees

```
procedure Build_BSPT ( F : set of faces ) returns BSPTreeNode

  Choose a hyperplane H that embeds a face of F;
  new_BSP := a new BSP tree node with H as its
             partitioning plane;
  <F_right, F_left, F_coincident, > := partition faces of F with H;
  Append each face of F_coincident to the appropriate face list
    of new_BSP;

  if (F_left is empty) then
    if (F_coincident has the same orientation as H) then
      (* faces point "outward" *)
      new_BSP.left := "in";
    else   new_BSP.left := "out";
  else
    new_BSP.left := build_BSPT( F_left );

  if (F_right is empty) then
    if (F_coincident has the same orientation as H) then
      new_BSP.right := "out";
    else   new_BSP.right := "in";
  else
    new_BSP.right := build_BSPT( F_right );

  return new_BSP;
end;  (* Build_BSPT *)
```

Inserting a face

1. Let *v* be some node in the tree (initially equal to root) and *f* be some face to add
2. Partition *f* by H_v and pass the part of *f* lying to the left of H_v to v.left and part of *f* lying to the right to v.right
3. Repeat this process until part of *f* reaches a leaf (create a new node)

Using this process one can go from a trivial BSP to BSP tree representing polyhedra

Evaluating Set Operations

- **Regular set** - set that consists of its interior and its boundary
- Partition space into regions such that at least one operand is homogenous in each region (e.g. ext(S) or int(S))

op	left operand	right operand	result
$\cup^*$	S	in	in
	S	out	S
	in	S	in
	out	S	S
$\cap^*$	S	in	S
	S	out	out
	in	S	S
	out	S	out
$-^*$	S	in	out
	S	out	S
	in	S	$\sim^*$ S
	out	S	out

Figure SIMPLIFY. Expression simplification rules. S is an arbitrary regular set.

Evaluating Set Operations

Given BSP tree T' representing polyhedron T and B-rep (or BSPT) B' representing polyhedron B.
Perform T -* B:

1. Insert all faces of B' into T'
2. If at some node *v*, no part of B' is found to lie on one side of H_v (let's say left) then R(v.left) is homogenous
3. Determine whether the region is "in" or "out" of B

```
procedure Incremental_Set_op
              ( op : set_operation ; v : BSPTreeNode ;
                B : set of Face ) returns BSPTreeNode
 if v is a leaf then
   case op of
     ∪* : case v.value of
       in  : return v
       out : return Build_BSPT( B )
     ∩* : case v.value of
       in  : return Build_BSPT( B )
       out : return v
 else
   <B_left, B_right, B_coincident> := partition B with H_v
   if B_left has no faces then
     status := Test_in/out(H_v, B_coincident, B_right)
     case op of
       ∪* : case status of
         in  : discard_BSPT( v.left )
               v.left := new "in" leaf
         out : do nothing
       ∩* : case status of
         in  : do nothing
         out : discard_BSPT( v.left )
               v.left := new "out" leaf
   else
     v.left := Incremental_Set_op( op, v.left, B_left )

   if B_right has no faces then
     (* similar to above *)
   else
     v.right := Incremental_Set_op(op, v.right, B_right)
   return v

end Incremental_Set_op ;
```

Evaluating Set Operations

4. Determine what to do with the appropriate subtree (v.right or v. left) given the operation and type of region
5. If v is a leaf, then R(v) is homogenous and will either retain T's value or B's value

```
procedure Incremental_Set_op
                ( op : set_operation ; v : BSPTreeNode ;
                  B : set of Face ) returns BSPTreeNode
 if v is a leaf then
   case op of
     ∪* : case v.value of
       in  : return v
       out : return Build_BSPT( B )
     ∩* : case v.value of
       in  : return Build_BSPT( B )
       out : return v
 else
   <B_left, B_right, B_coincident> := partition B with H_v
   if B_left has no faces then
     status := Test_in/out(H_v, B_coincident, B_right)
     case op of
       ∪* : case status of
         in  : discard_BSPT( v.left )
               v.left := new "in" leaf
         out : do nothing
       ∩* : case status of
         in  : do nothing
         out : discard_BSPT( v.left )
               v.left := new "out" leaf
   else
     v.left := Incremental_Set_op( op, v.left, B_left )

   if B_right has no faces then
     (* similar to above *)
   else
     v.right := Incremental_Set_op(op, v.right, B_right)
   return v

end Incremental_Set_op ;
```

Evaluating Set Operations

- Perform T -* B

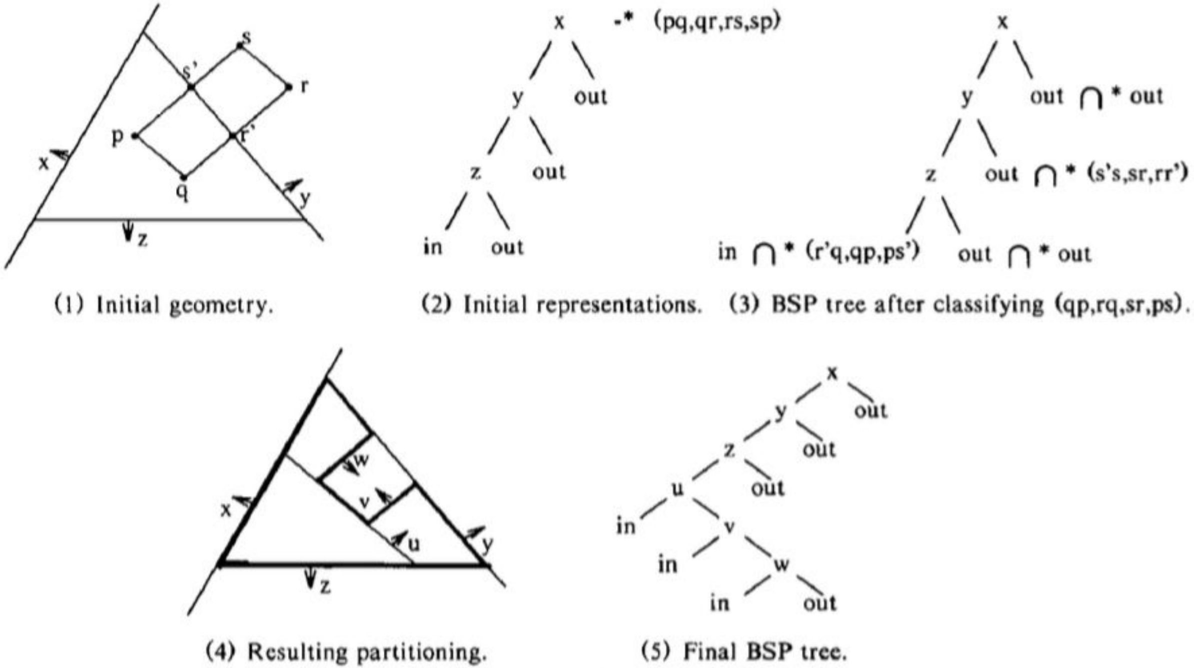
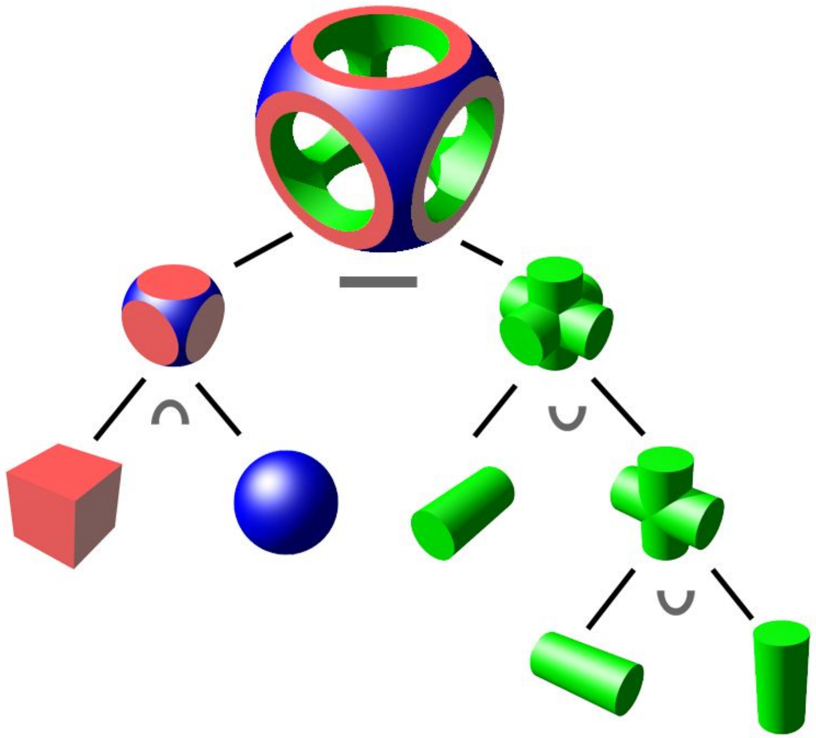


Figure SET-OP. BSP tree —* B-rep → BSP tree.

CSG Trees

- A binary tree in which the internal nodes represent (regularized) set operations and leaves are instanced primitives
- Easier visual representation for complex objects
- Not particularly useful computationally (need to convert to BSPT)



BSP Tree Reduction

- Eliminate certain nodes without changing the set - reduction in memory
- Both subtrees of node v are cells with identical values
  - Replace subtree with single value
- Node that has one child and contains no part of the boundary (u)
  - Remove this node

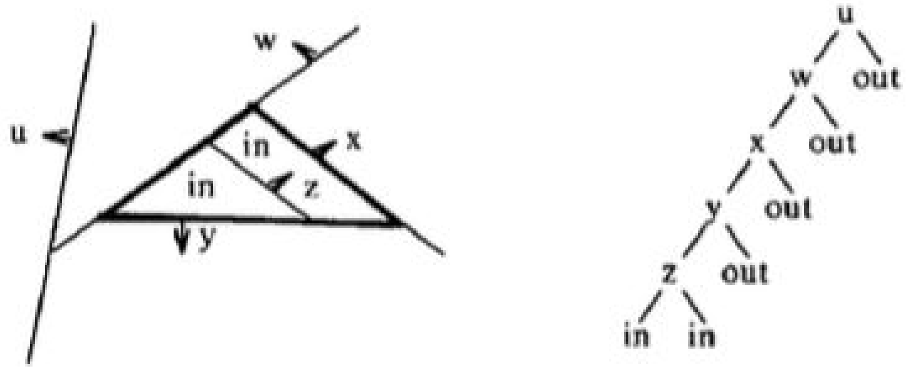


Figure REDUCE. Nodes u and z can be eliminated.

Conclusions

- Similarity between octrees and BSP trees
  - Recursively subdivide space
  - Assign values to leaves
  - Dimension independent
- Key difference: BSPT hyperplanes do not have to be axis-aligned
  - Octrees tend to be more verbose as a result (more memory)
- B-rep algorithms - independent search structure, set operations, and visible surface determination
- BSP tree -> all unified in a single structure
  - reduces the conceptual complexity and complexity of implementations

Assessment

Pros:

- Very detailed
- Not too complicated to follow
- Many diagrams to illustrate concepts
- Clear pseudocode

Cons:

- Could have provided more detail as to why approach is better
- Could have used better organization

Questions?