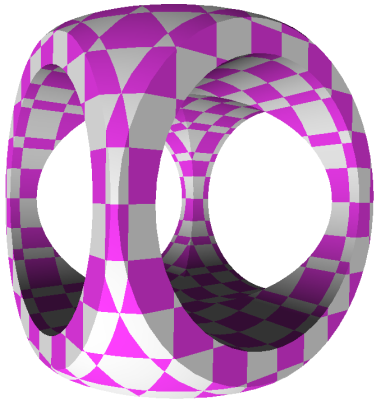
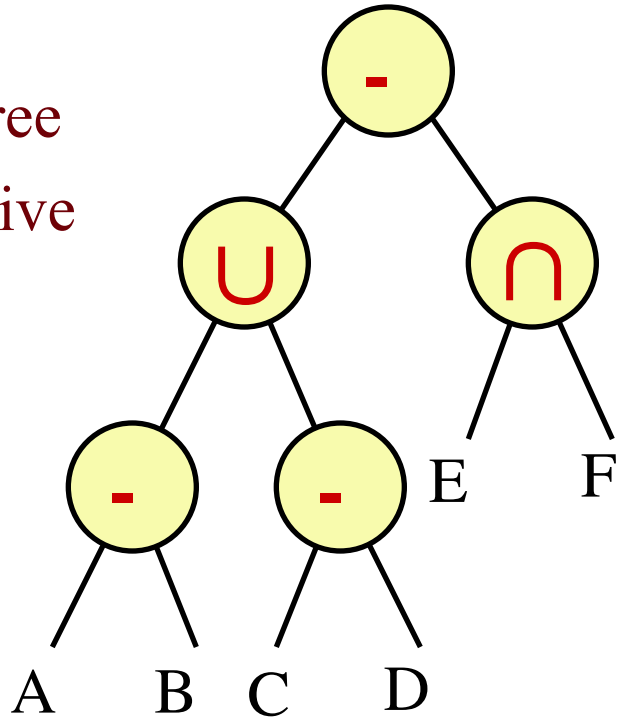
CSG rendering

Georgia Tech College of Computing
Interactive & Intelligent Computing

- Booleans
- Space partitions
- CSG & BSP
- CSG expression/tree
- Positive form of a CSG tree
- Active zone Z of a primitive
- Classifying surfels
- Blist form
- Blister rendering



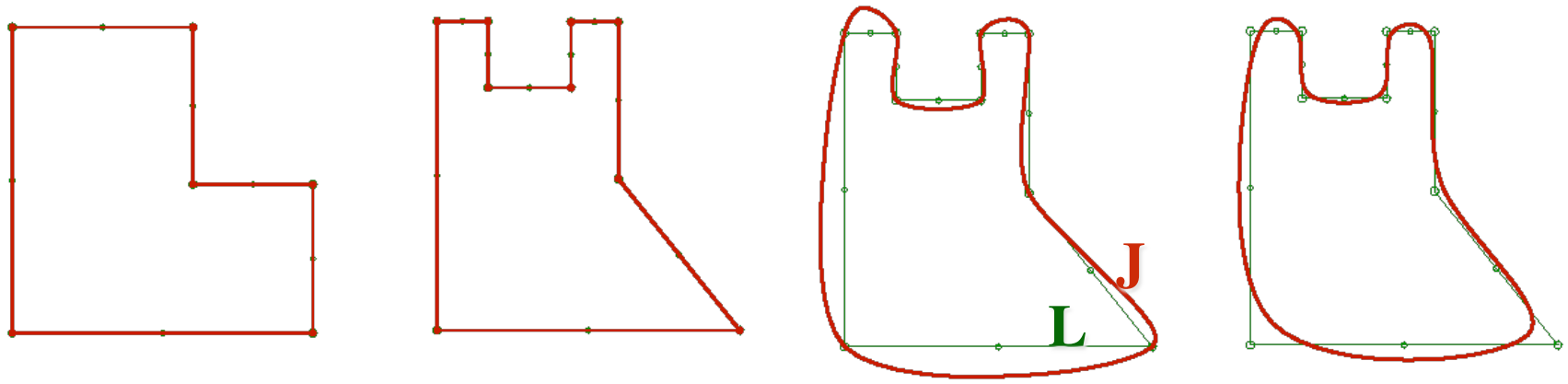


What is a polyloop L?

- A **closed loop curve** defined by a **cyclically ordered** set of **control vertices**
- It is convenient to have **next** (in) and **previous** (ip) functions for accessing them

```
int vn =6;                    // number of control vertices
pt[] P = new pt [vn];         // vertices of the control polyloop
int in(int j) {  if (j==vn-1) {return (0);}  else {return(j+1);}  };  // next vertex in control loop
int ip(int j) {  if (j==0) {return (vn-1);}  else {return(j-1);}  };  // previous vertex in control loop
```

- The user should be able to **insert**, **delete**, and **move** the control vertices…
- and get a smooth curve J **interpolating** or **approximating** the polyloop

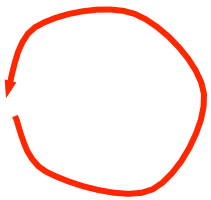


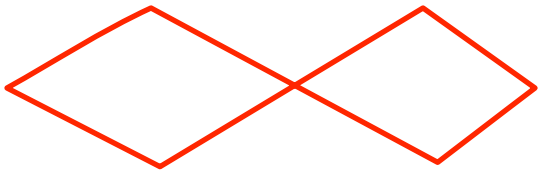
Concavity and inflection

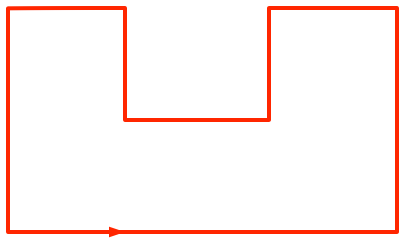
- How to test whether corner (A,B,C) is a left turn?

  $BC \bullet AB.left > 0$

- A curve is convex when all corners are left turns or all corners are right turns
  - CW: clockwise, CCW: counterclockwise
  - Convention: we **orient** the curve ccw
  - Some curves cannot be oriented (which?)
- An inflection edge joins a left and a right turn corner (example?)







Concave vertices?

Inflection edges?

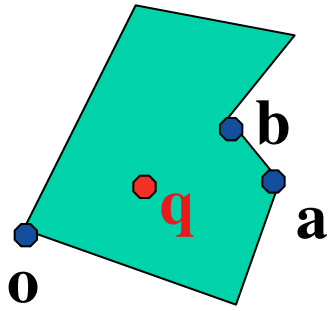
Point-in-polygon test in 2D

- Given a polygon P and a point q in the plane of P, how would you test whether q lies inside P?

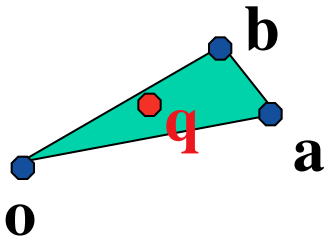
Point-in-polygon test

- Algorithm for testing whether point q is inside polygon P

```
in:=false;
for each edge (a,b) of P { if (PinT(q,a,b,o)) in := !in; };
return in;
```

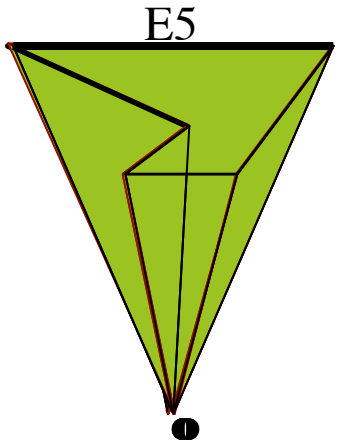


- PinT(q,a,b,o) was studied earlier

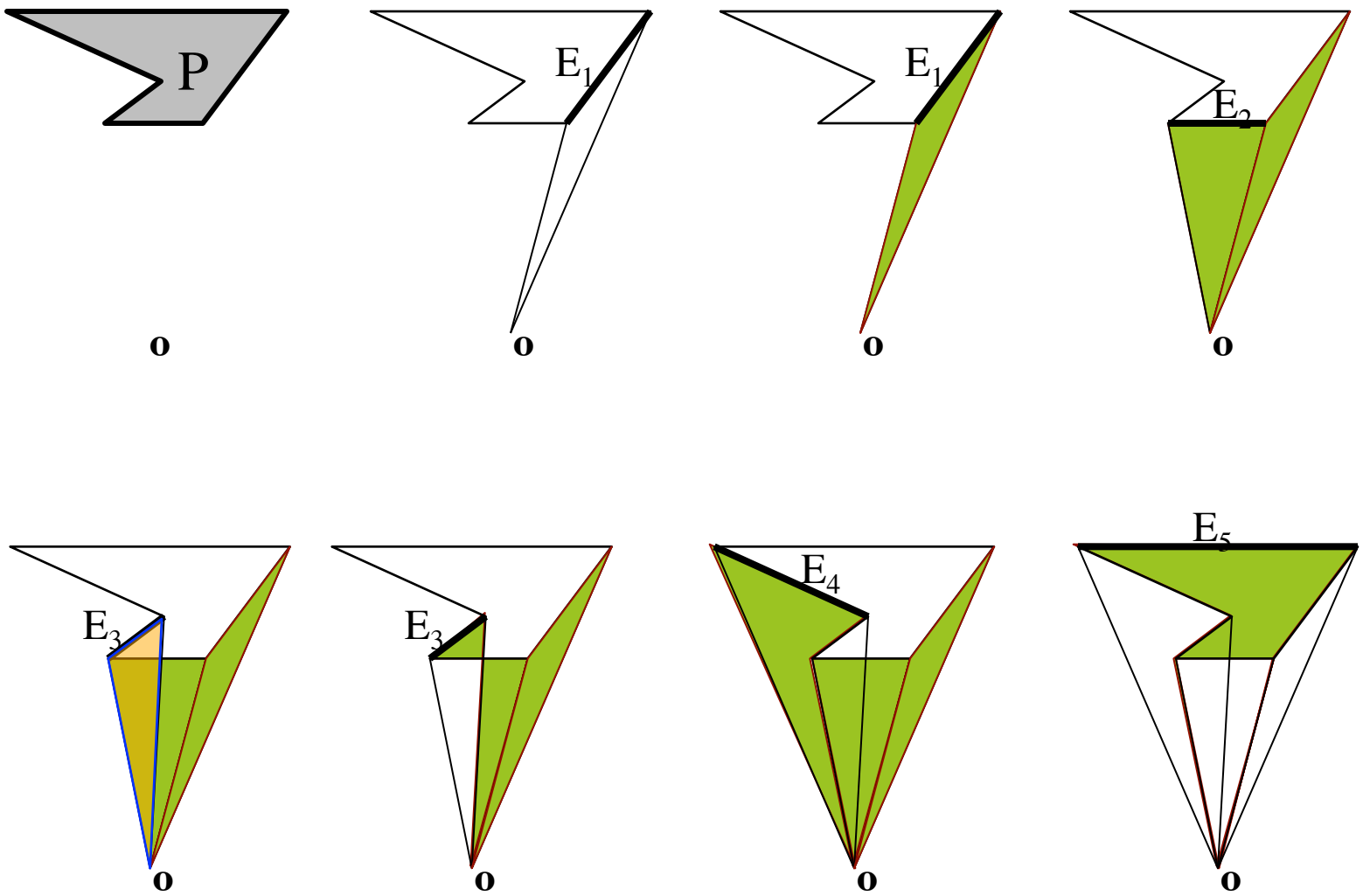


XOR shading of a polygon in 2D

- **A⊗B is the set of points that lie in A or in B, but not in both**
  - **A⊗B := {p: p∈A != p∈B }**
- **Let A, B, C, … N be primitives, then A⊗B⊗C⊗… ⊗N is the set of points contained in an odd number of these primitives**
  - **A⊗B := {p: (p∈A) XOR (p∈B) XOR (p∈C) XOR … XOR (p∈N) }**
- **How to shade a polygon A in 2D:**
  - **Given a polygon A, with edges $E_1, E_2, \ldots E_n$, let T_i denote the triangle having as vertices an** arbitrary **origin** o **and the two endpoints of E_i.**
  - **The interior of A is $T_1 \otimes T_2 \otimes T_3 \otimes \ldots \otimes T_n$**
    - *If you ignore the cracks*
  - To shade A:
    - Initialize all pixels to be background
    - Shade all the T_i using XOR
      - Toggle status of each visited pixel



Example of XOR polygon filling



Relation to ray-casting approach?

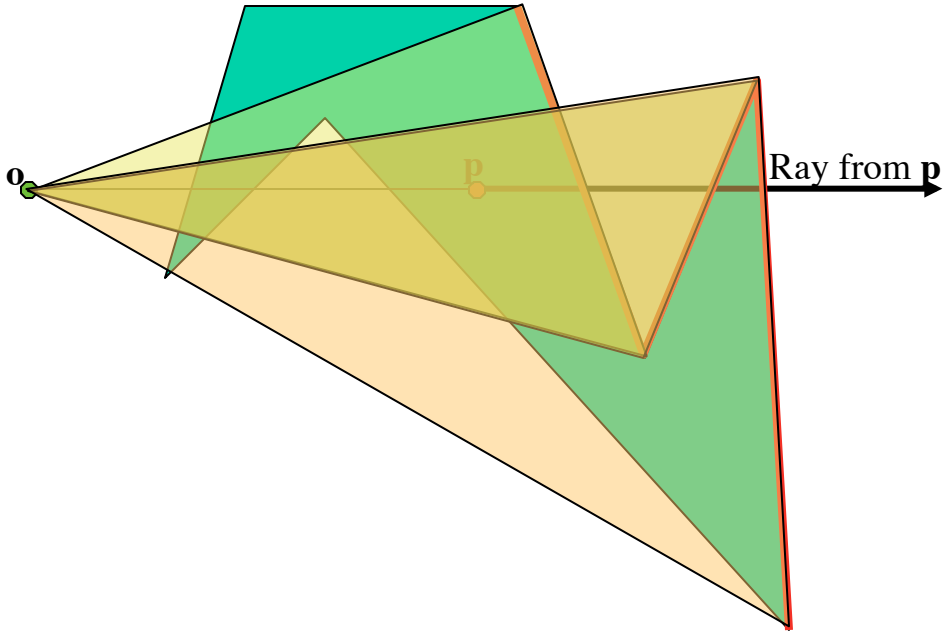
A point p lies in a set A if a ray from p intersects the boundary of A an odd number of times

If your ray hits a vertex or edge or is tangent to a surface, pick another ray

We do the same thing. Look at an example in 2D. Here:

Only edges E_1, E_2, E_3 intersect the ray from p in the opposite direction to o

Thus only triangles T_1, T_2, T_3 contain p



p is in because it is contained in an **odd** number of triangles

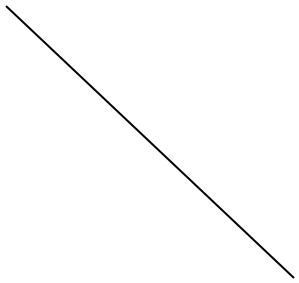
Convex sets and convex hulls

- A set S is convex if for any points a and b in S, the segment ab is in S.
- The convex hull H(S) of a set S is the smallest convex set containing S.

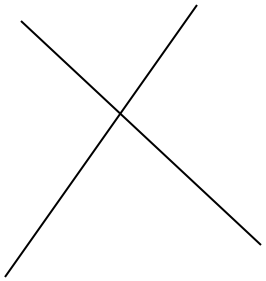
- Is it true that "for a planar set S, H(S) is the union of all line segments ab with a and b in S"?

Line arrangements in the plane

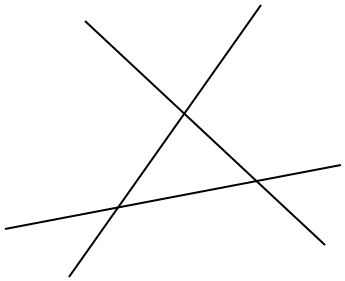
- N lines divide the plane into cells
  - Convex 2d regions bounded by line segments (some unbounded)
- Assume lines are in general position
  - no 2 are parallel, no 3 are coincident
- How many cells do we have?
  - Do a few examples
  - Establish a recursive formulation
  - Solve it
  - Remember the essence of your approach



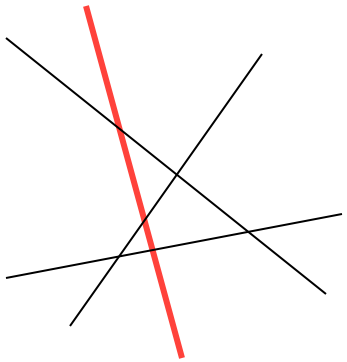
L=1, C(1)=2



L=2, C(2)=4



L=3, C(3)=7



L=4, C(4)=

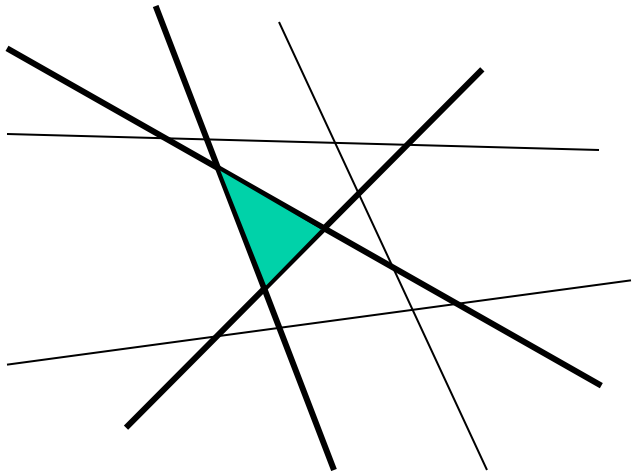
Counting cells in line arrangements

- Obviously C(0)=1.
- For n>0, C(n)=C(n–1)+n,
  - because the line bounding the new half-space is intersected at most n points by previous lines and thus divided into at most n+1 segments.
  - Each segment splits a previous cell into 2, and thus they all create at most n+1 new cells.
- The recursive formula yields C(n)=1+(1+2+3+4+5…+n),
- which can be solved for C(n)=1+(n+1)n/2,
  - using the fact that 1+2+3+4+…+n = n(n+1)/2
- It may also be written as $C(n)=(n^2+n+2)/2$.
  - Indeed C(2)=1+3*(2/2)=4 and C(3)=7.

How to identify a cell?

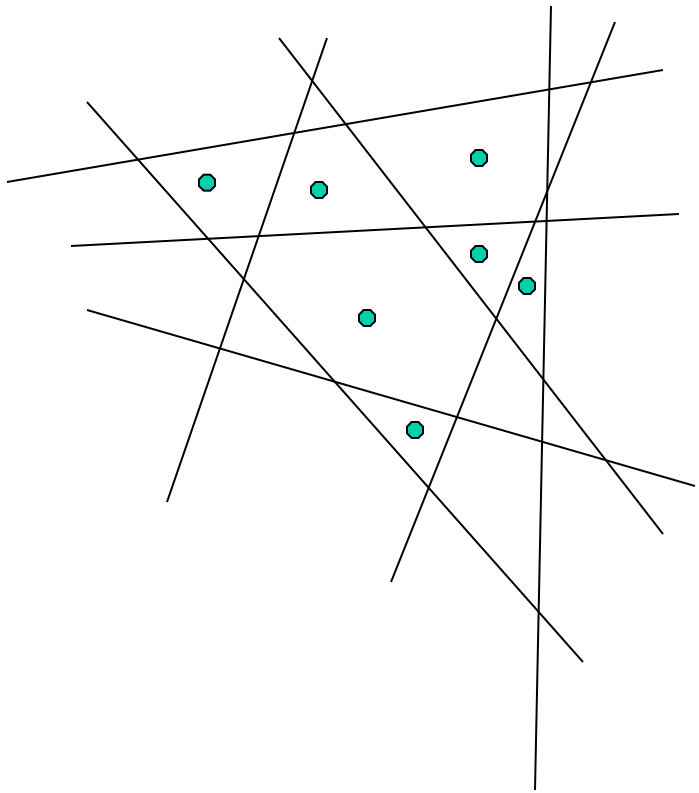
- How can we identify a cell?
  - Label the lines: A, B, C
  - How can you unambiguously reference a single cell?

  - How many bits do you need for a cell name?
  - What is the theoretically minimal number of bits?



How to specify **collections** of cells?

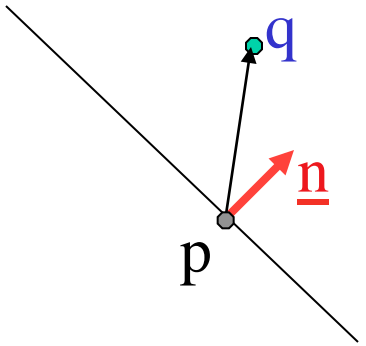
- Geometric: a point in each cell of the collection
  - Any point will do
  - Not convenient for drawing a cell



- Semi-algebraic
  - In terms of inequalities
  - as a Boolean combination of
  - Half-spaces

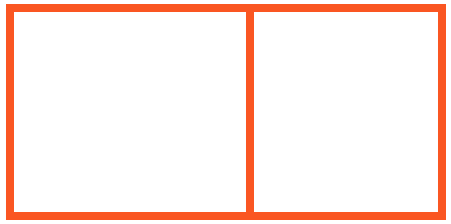
What is a half-space

- A half-space A separates space in 2 parts: A and !A (complement)
- Examples of algebraic half-spaces (polynomial inequality)
  - A sphere, $S(p,r)=\{q: \|pq\|<r\}$
  - A linear half-space, $H(p,\underline{n})=\{q: pq \bullet \underline{n}>0\}$



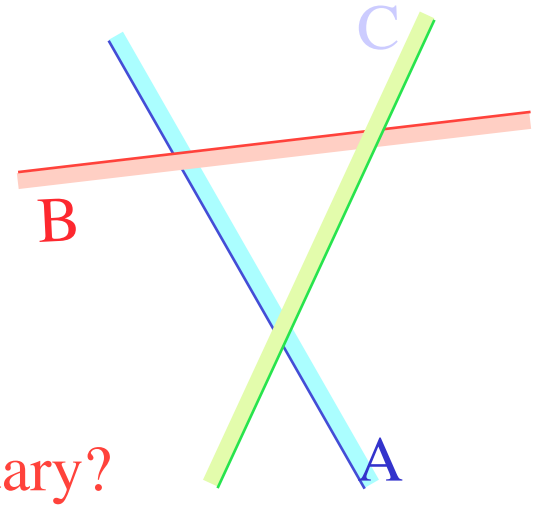
- Examples of non-algebraic half-spaces
  - Genus zero surface created by a Split&Tweak
  - Butterfly subdivision of a water tight triangle mesh
  - Space swept by a polygon moving and rotating in 3D

- Examples of things that are not half-spaces
  - Curves with borders (holes). Not water tight
  - Some non-manifold surfaces

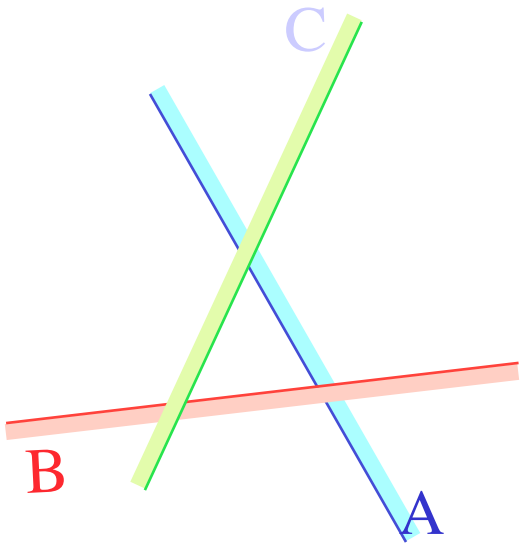


Intersection of linear half-spaces

- It is a convex set bounded by planar faces
  - Some of the faces may be unbounded
  - The intersection may be empty
- How to test whether point q is inside?



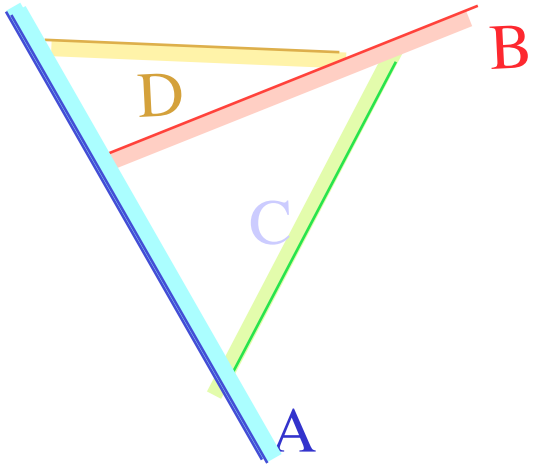
- How to test whether point q is on the boundary?

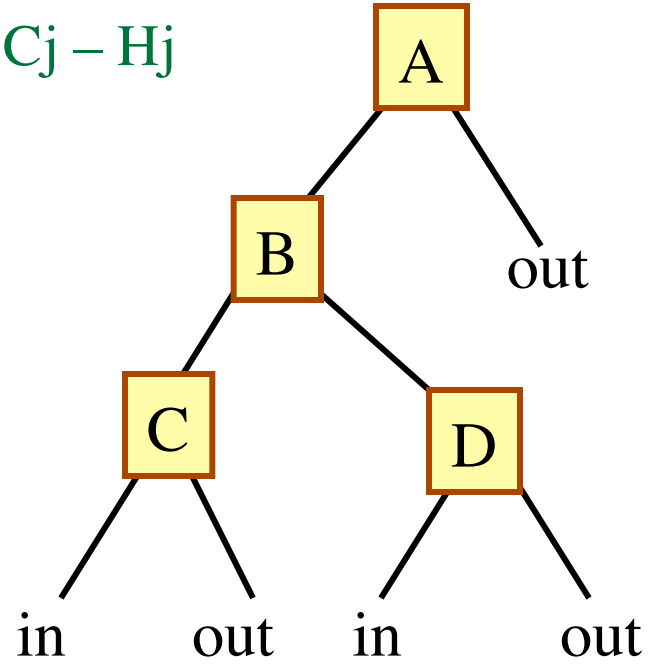


- *The intersection needs not be closed*
  - *Complement of a closed half-space is open*
- Each cell of an arrangement of lines
  - is the intersection of linear half-spaces,
  - each half-sapce is bounded by one of the lines

BSP (Binary Space-partition Tree)

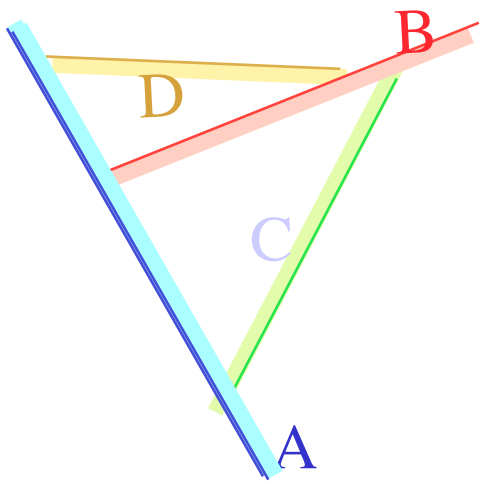
- Binary tree
- Each node Nj represents a convex cell Cj
  - It is associated with a half space Hj
  - The left child is associated with a cell Cj Hj
  - The right child is associated with a cell Cj – Hj
- Leaves that are left children are in
  - Other leaves are out

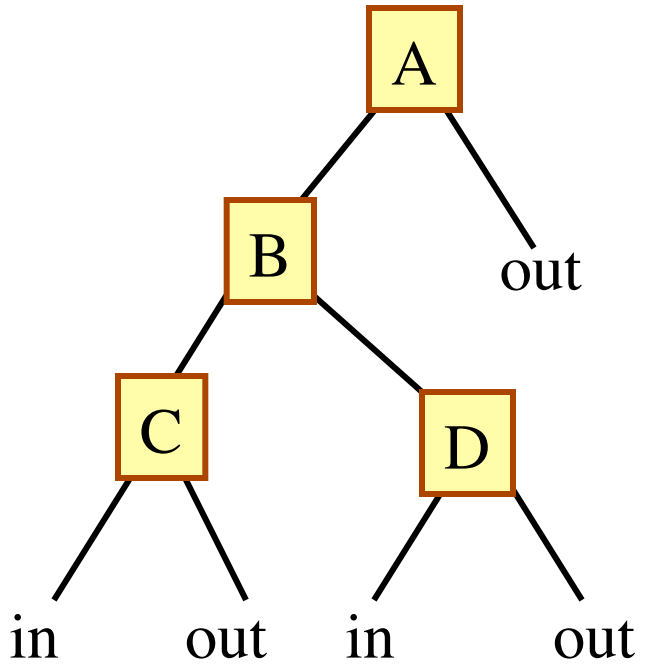




How to test a point **p** against a BSP

- **Start at the root**
- **When you reach a leaf**
  - **return (leaf == leaf.parent.left)**
- **At an internal node**
  - **If p in node.halfspace go left**
  - **Else go right**



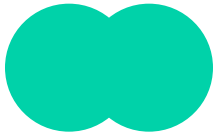


BSP for back-to-front ordering of faces

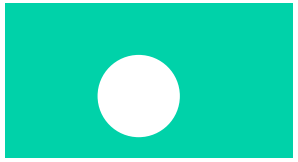
- HSR = Hidden Surface Removal
  - Currently done with z-buffer in contemporary hardware
  - Previously done using painter's algorithm (paint back-to-front)
  - Back-to-front order is view dependent: use BSP to re-compute quickly
- BSP for back-to-front: At each internal node n store
  - Half-space n.H
  - Pointers to children n.L (in H) and n.R (out of H)
  - List of front faces n.F in the boundary of H facing interior of H
  - List of back faces n.B in the boundary of H facing exterior of H
- Render (node n, viewpoint v)
  - If (n==nil) RETURN;
  - IF (v in n.H) {Render(n.R, v); Display(n.F); Render (n.L, v)}
  - ELSE {Render(n.L, v); Display(n.B); Render (n.R, v)};

Set theoretic Boolean operations

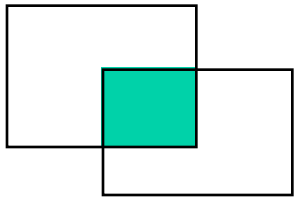
- 16 total, only 5 of interest



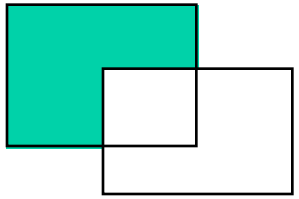
union A+B = {s: s in A or s in B}



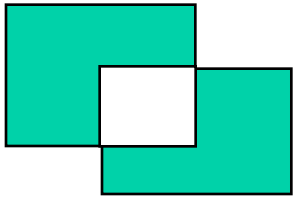
complement !S = {s: s not in S}



intersection AB = {s: s in A and s in B}

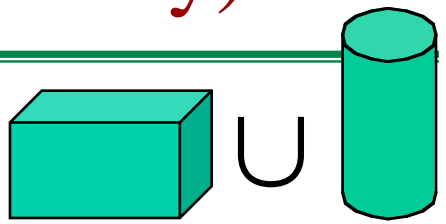


difference A–B = {s: s in A and s not in B}

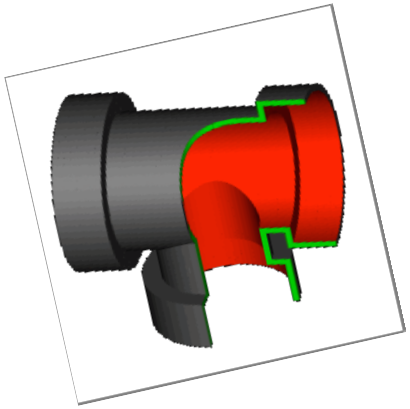


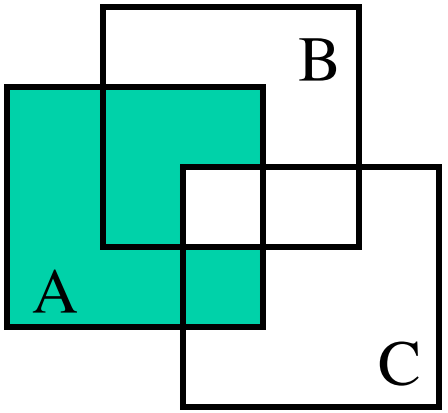
xor A⊕B =(A+B)–AB

CSG (Constructive Solid Geometry)



- Set theoretical Boolean expression
- Has half-spaces as primitives
  - Bounded solids: Blocks, spheres, cylinders, cones, tori
  - Transformed by a rigid body motion
- Uses only union, intersection, and difference operators
  - Priority: intersection higher than union and difference
  - Example: (A+B)-(CD-E) is ((A+B)-((CD)-E))

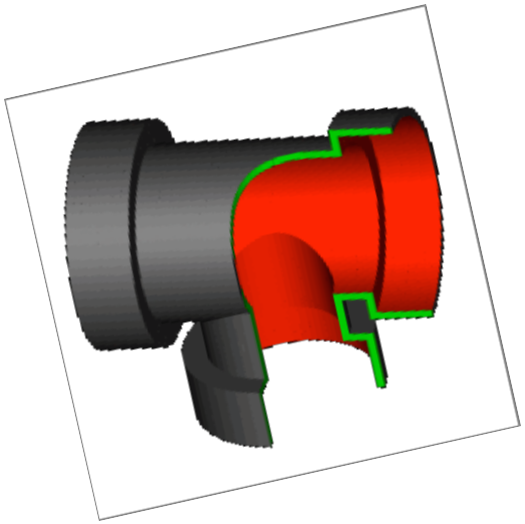


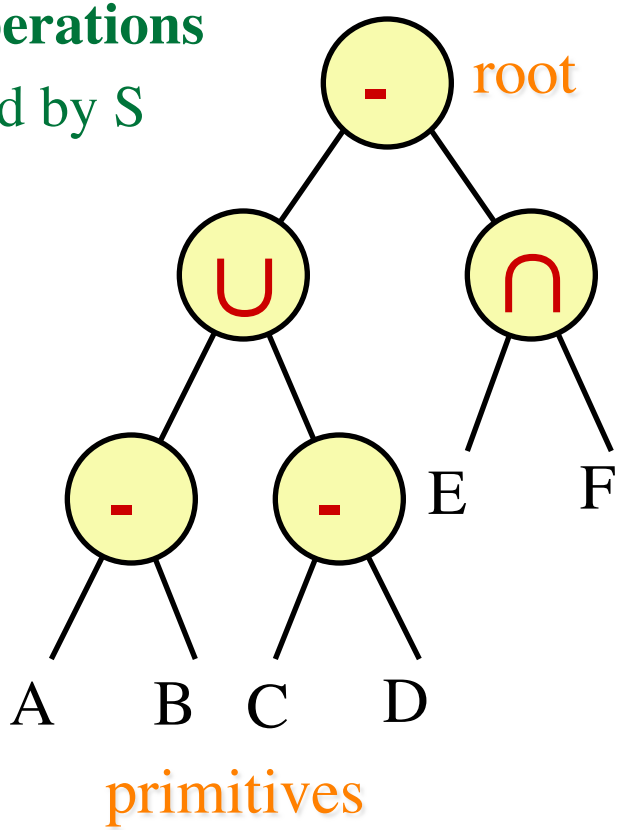


- Write the simplest expression for the figure: A B C

What is a CSG **tree**

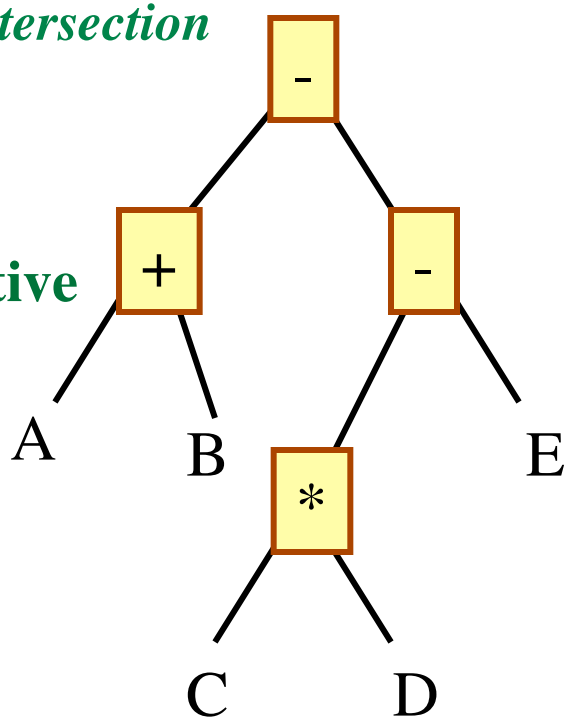
- Consider a CSG expression: S=(A–B)+(C–D)–(EF)
- It may be parsed into a binary tree
  - Leaves correspond to **primitive** shapes
  - Nodes represent Boolean **operations**
  - **Root** represents solid defined by S





How to store a CSG expression

- Parsing a CSG expression yields a binary tree
  - S = ((A+B)–((C*D)–E)), **** stands for intersection***
- Store it as an table of nodes:
  - Type: **+, *, –, prim Id**
  - Left: **Id of left child node or of primitive**
  - Right: **Id of right child node**
- Plus a table of primitives:
  - Type: **block, cylinder…**
  - Dimensions: **radius, length…**
  - Transformation: **3x4 matrix**
    - Product of rotations, translations…



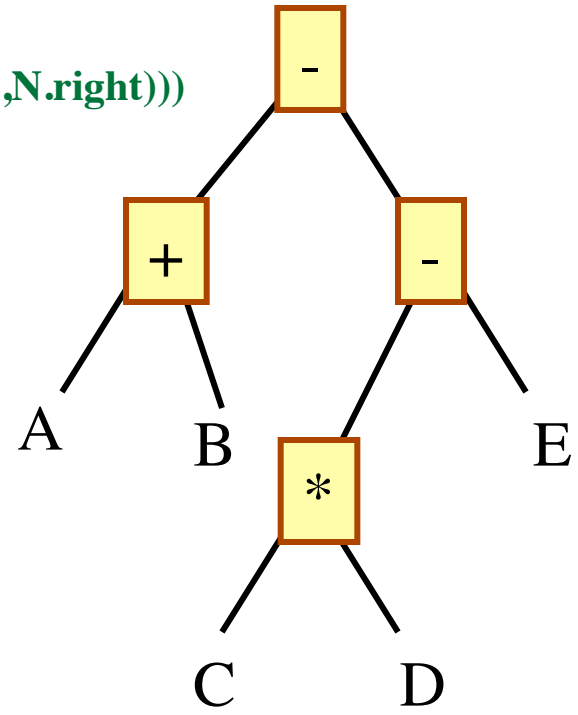
Point-in-solid test for CSG

- Test the point for inclusion in each primitive P
  - Produces Boolean result (True/False) p
  - Ignore for now "on" cases
- Evaluate Boolean expression
  - Substitute P in the CSG expression by p
  - Substitute Set theoretic operators by Boolean operators
    - Union is OR, ||
    - Intersection is AND, &&
    - Difference is AND NOT, &&!
- Example: ((A+B)–((CD)–E))
  - yields ((a OR b) AND NOT ((c AND d) AND NOT e))

Details of point-in-CSG

- PinCSG(point p, node N) {
  if N.type in {+,*,-}
  then return (combine**(N.operator,** PinCSG**(p,N.left),** PinCSG**(p,N.right)))**
  else return PinP**(p, N.type, N.dimensions, N.transformation) }**

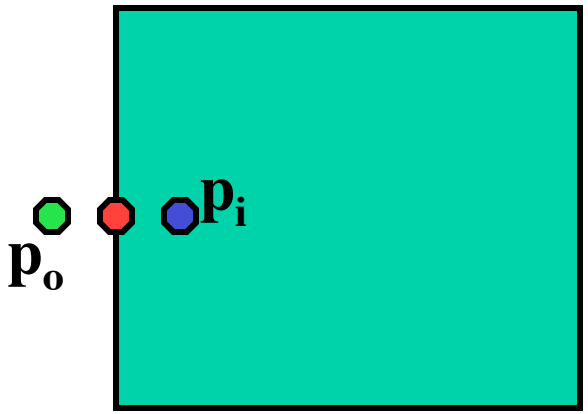
- Combine **(op, left, right) {**
  if op== '+' then return (left OR right);
  if op== '*' then return (left AND right);
  if op== '-' then return (left AND NOT right);}



- PinP **(p, type, Dim, xform) {**
  - **(x,y,z):=apply(p,inverse(xform));**
  - **If type=sphere then return($x^2+y^2+z^2<Dim[1]^2$);**
  - **If type=cylinder then return($x^2+y^2<Dim[1]^2$ AND 0<z AND z<DIM[2]);**
  - **…}**

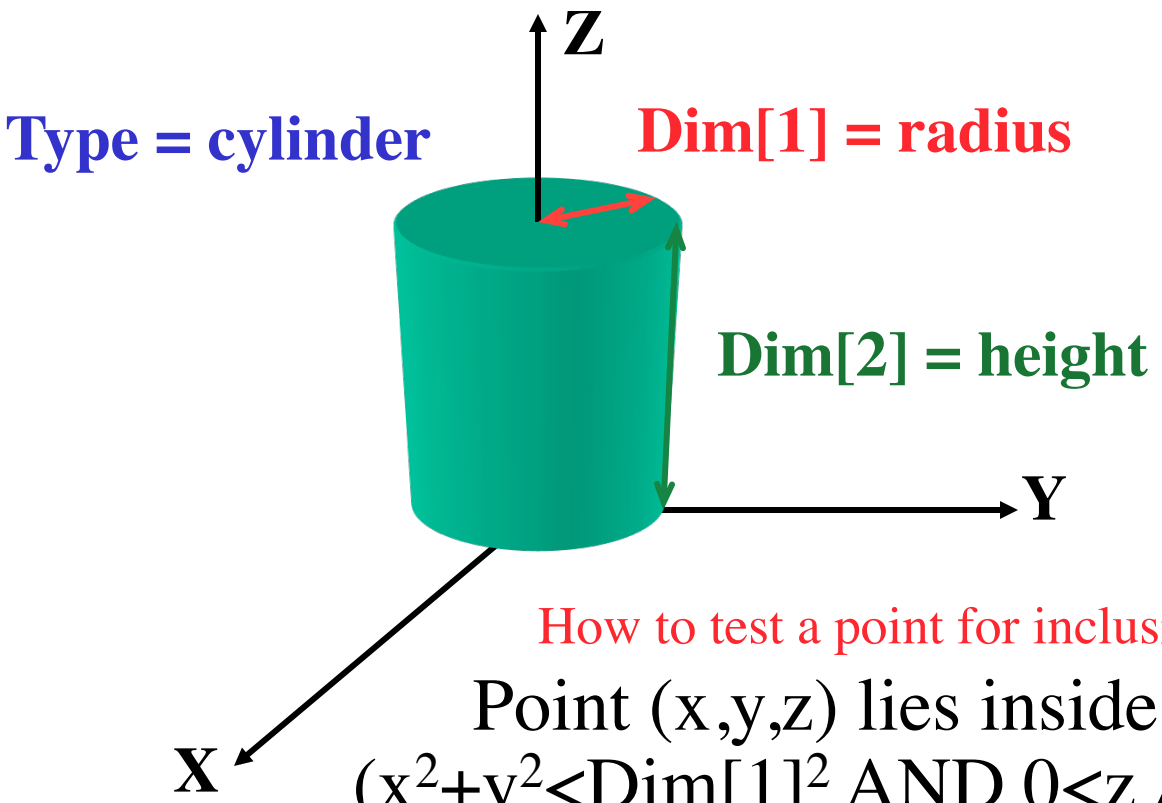
How to test a point **p** on a primitive

- **Test an offset point p_i inside and an offset point p_o outside**
  - **The offset path may not traverse any other boundary**
- **If the results disagree, p is ON the CSG solid**



Primitive(cylinder,radius,height)

- Expressed in their local coordinate system
- Defined by their intrinsic parameters

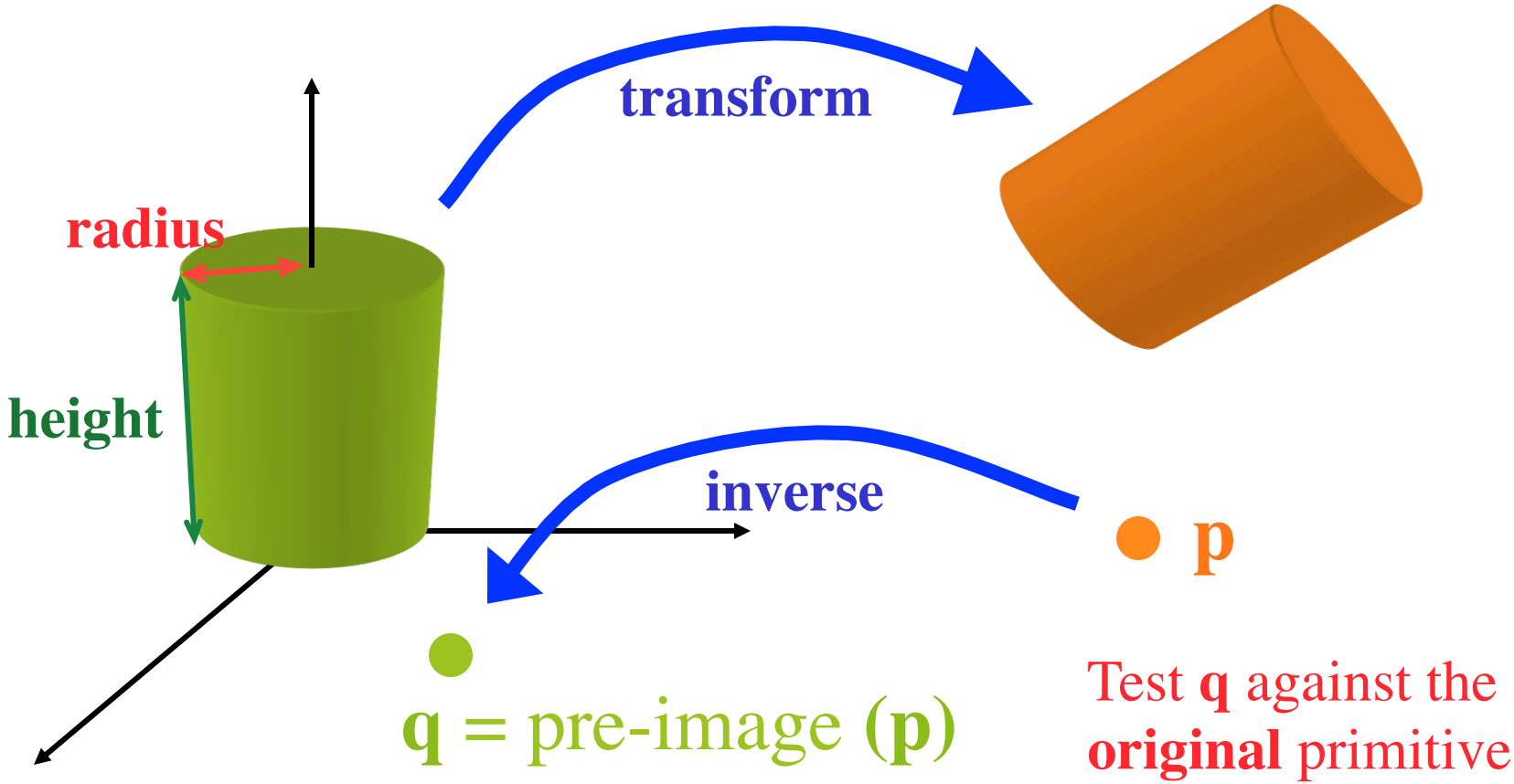


How to test a point for inclusion in the cylinder?

Point (x,y,z) lies inside if and only if
($x^2+y^2<Dim[1]^2$ AND 0<z AND z<DIM[2]);

Rigid transformations

- Specifies where the primitive is:
- How to test for inclusion in the transformed primitive?



Issues

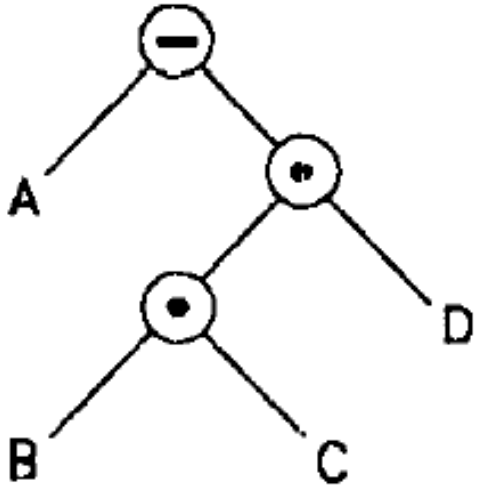
- How to convert from BSP to CSG ?
- How to convert from CSG to BSP ?
- Which is faster for Point-in-Solid tests

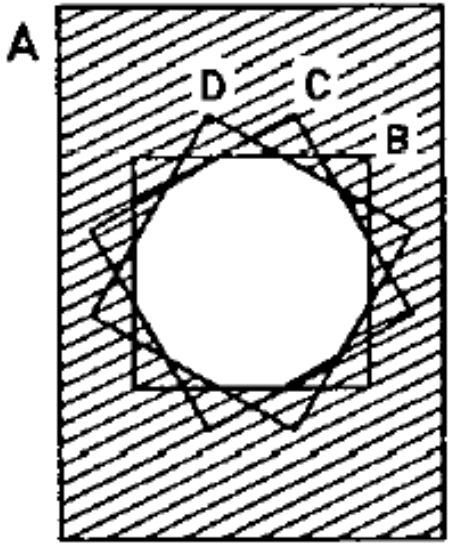
What you must remember

- Definitions of Boolean operators
- Definitions of linear half-spaces
- Point-in-half space tests for planes, spheres, cylinders
- What are BSP and CSG
- How to represent them
- How to perform point-in-solid tests on them
- How to come up with number of cells in line arrangement

A 2D example

```
A=REC(3,4);
B=REC(2,2);
C=B.turn(60);
D=C.turn(60);
S=A-B•C•D;
```





A shape has many CSG reps

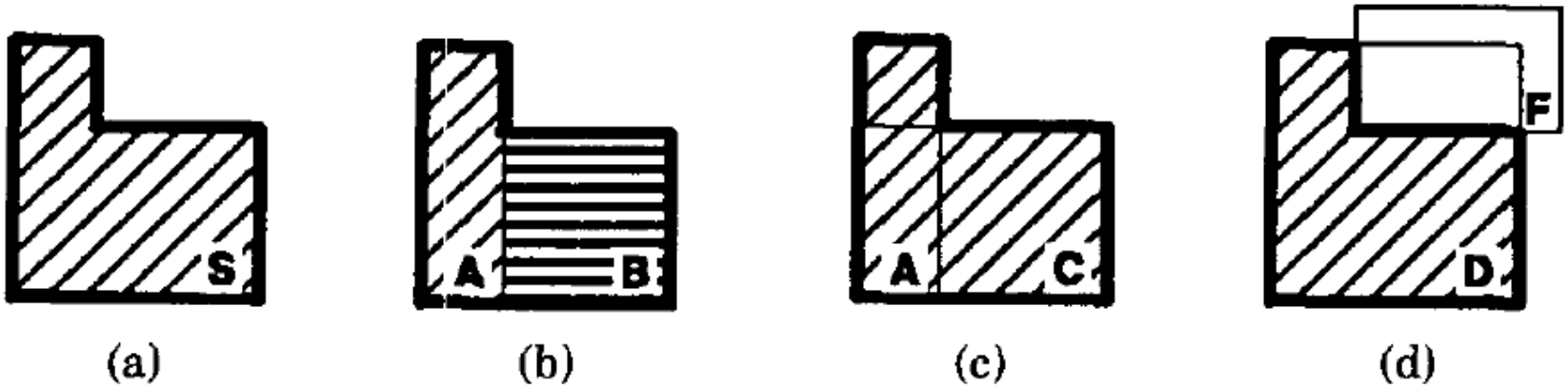


Fig. 2. A CSG representation is not unique: A solid (a) can be represented as $A + B$ (b), $A + C$ (c), $D - F$ (d), and so on.

When is a primitive **redundant**

A primitive A is redundant in S if S can be expressed using the other primitves, without A.

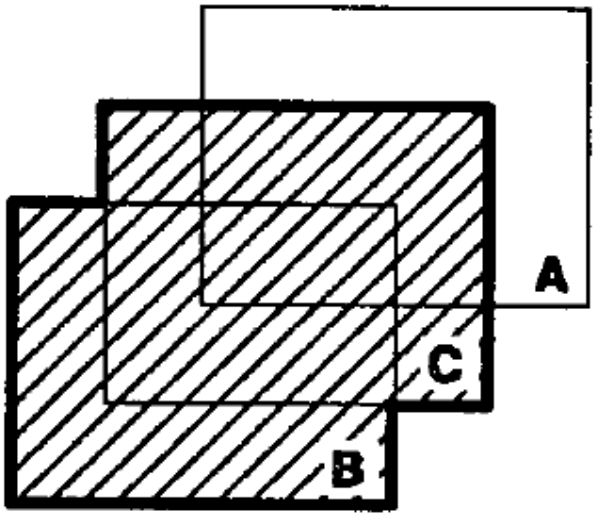


Fig. 3. A redundant primitive: In the solid defined in CSG by $(B - A) + C$, the primitive A is redundant because the same solid can be defined by $B + C$.

Is A redundant if δA adds nothing to δS?

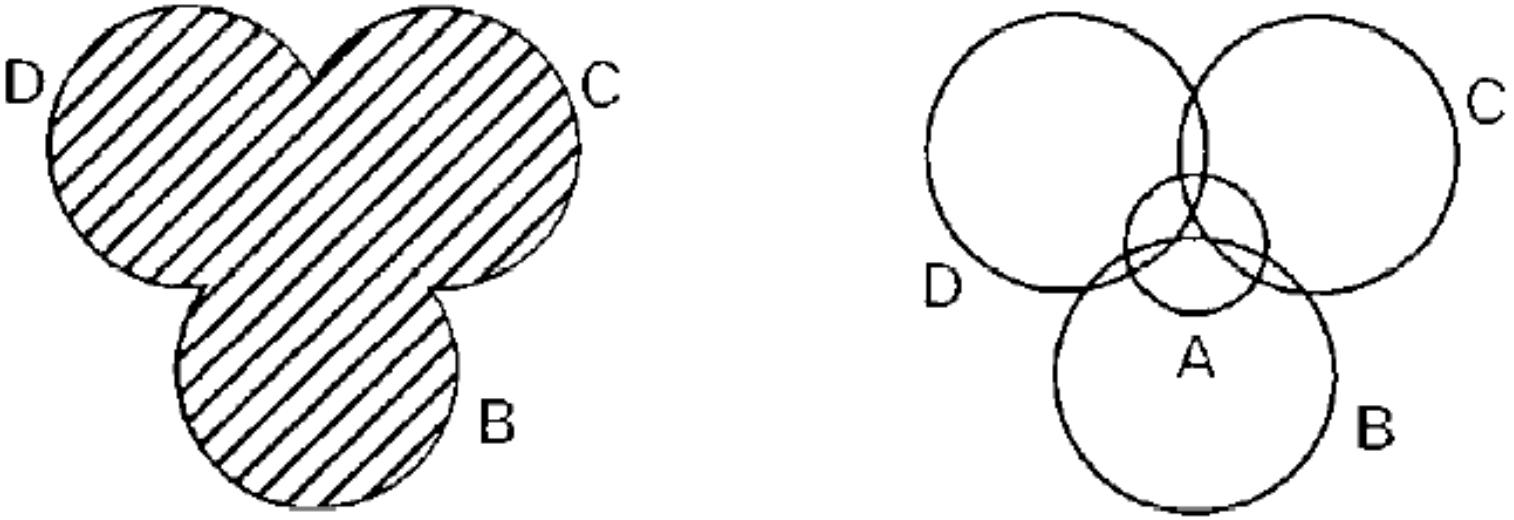


Fig. 5. Boundary evaluation and redundancy detection: The region S (left) is defined in CSG as the union of four disk primitives A, B, C, and D (right). A is not redundant, even though ∂A does not intersect ∂S.

How to convert a CSG to its **positive form**?

Apply de Morgan laws and push the complement down to the primitives

$$X - Y \rightarrow X \cdot \bar{Y},$$
$$\overline{X + Y} \rightarrow \bar{X} \cdot \bar{Y},$$
$$\overline{X \cdot Y} \rightarrow \bar{X} + \bar{Y}$$
$$\bar{\bar{X}} \rightarrow X.$$

Primitives complemented an odd number of times are negative (and hence unbounded)

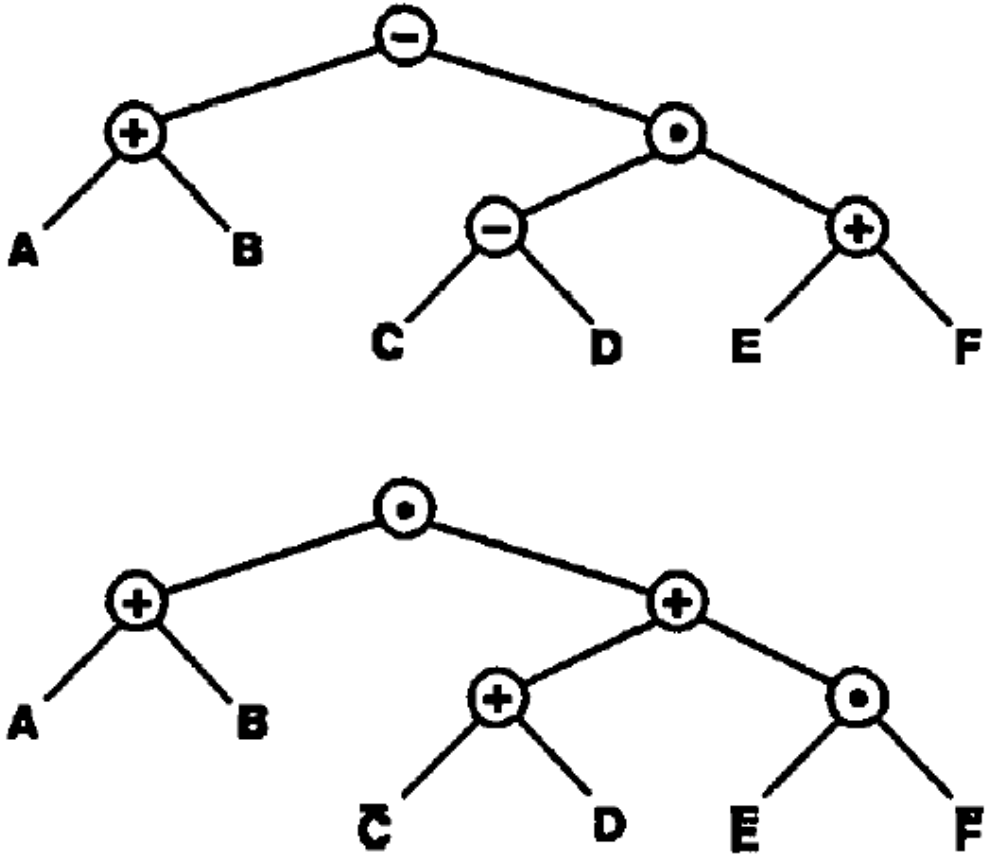
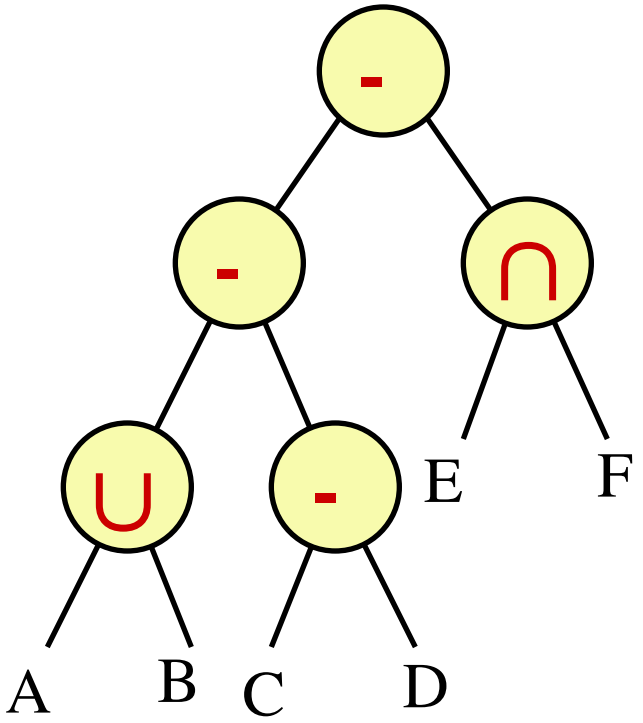


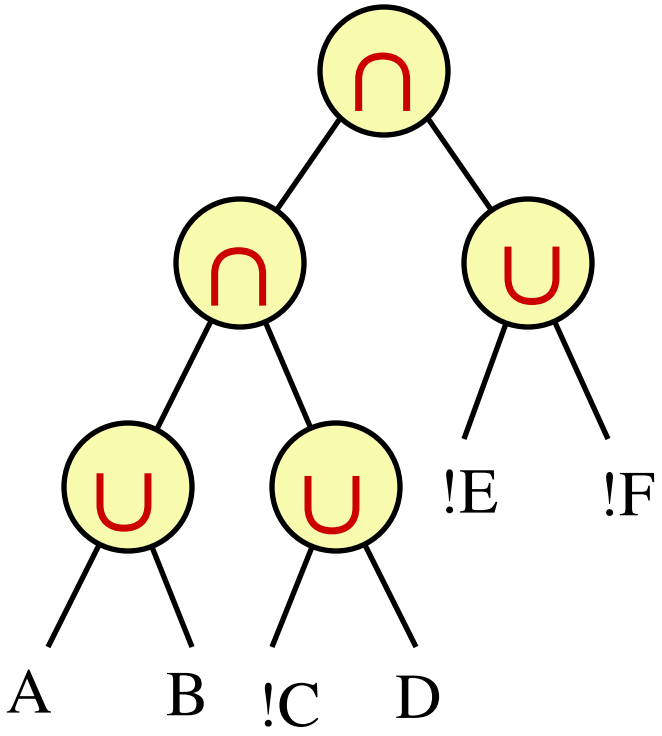
Fig. 6. Tree reformulation: $(A + B) - (C - D) \cdot (E + F)$ is transformed into its positive form, $(A + B) \cdot ((\bar{C} + D) + \bar{E} \cdot \bar{F})$.

What is the **positive form** of S?

S=(A+B)–(C–D)–(EF)

S = (A+B) (!C–D) (!E+!F)



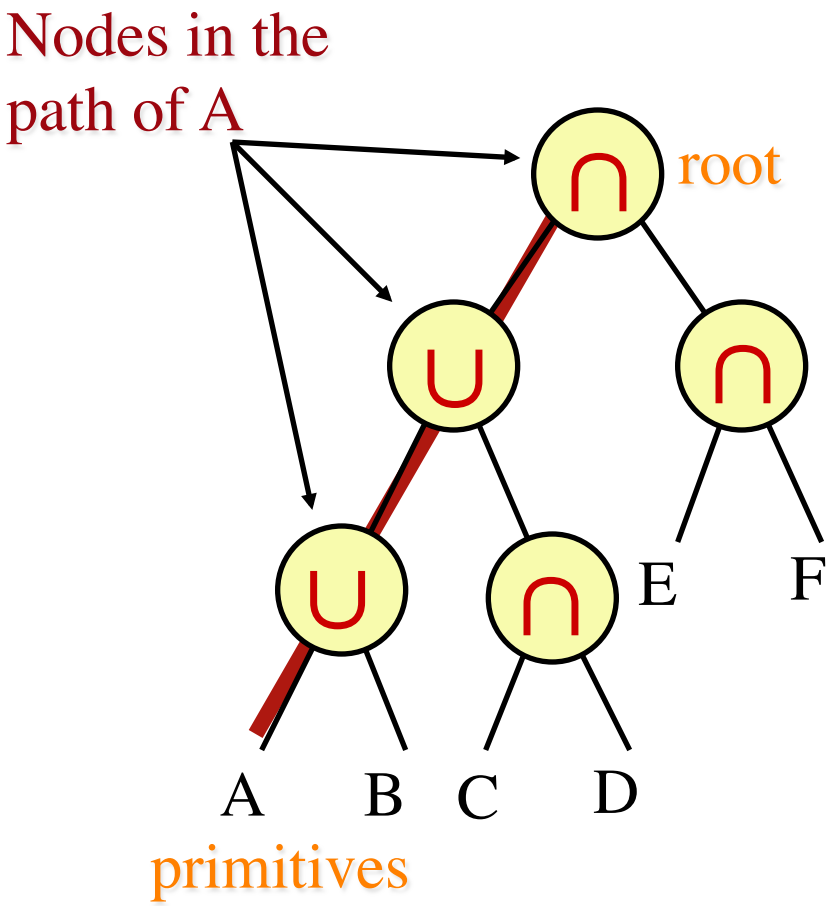


Now assume CSG is in **positive form**

- Rename the negative primitives with new names so that they become positive (although infinite) primitives
- A CSG tree in positive form has only union and intersection operators
- Hence, in what follows we need not discuss what to do with difference operators
- We assume that the CSG tree has been converted into its positive form and that all primitives are positive (even though some may be unbounded)

What is the **path** of a primitive A?

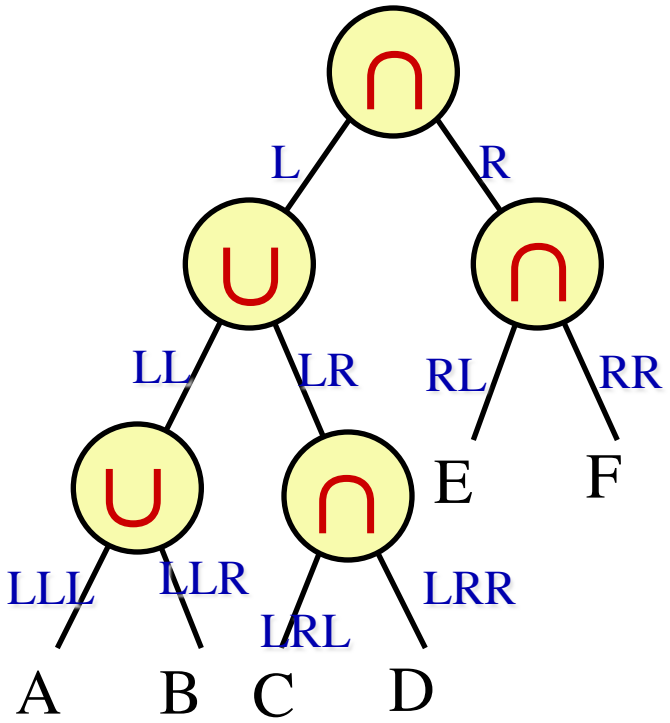
- The parent and ancestors of A in the CSG tree.



How to **compute** the **path**?

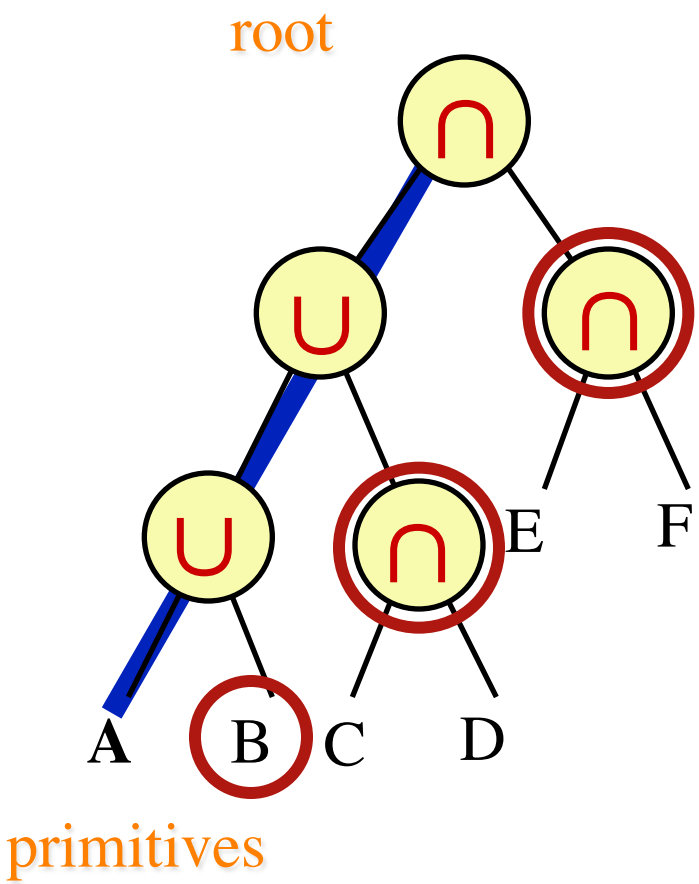
Path will be represented as a string of L or R symbols

```
p=emptyString; path(root,p);
path(n,p) {
    if (n.isPrimitive) {
        n.myPath=p;
    else {
        path(n.left, p+'L');
        path(n.right, p+'R');
        };
    };
```



What are the **branching nodes**?

- Siblings of path nodes

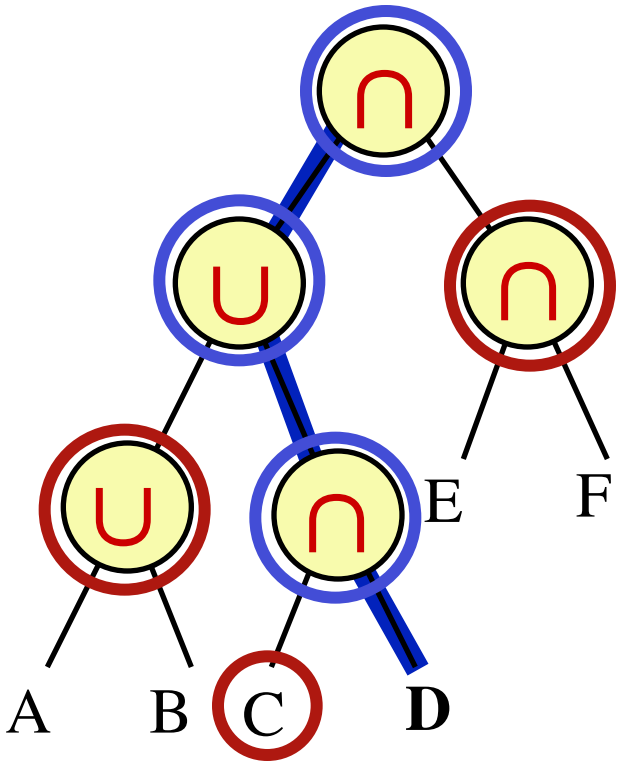


Branching nodes of A

What are the path and branching nodes of D?

Path nodes

Branching nodes



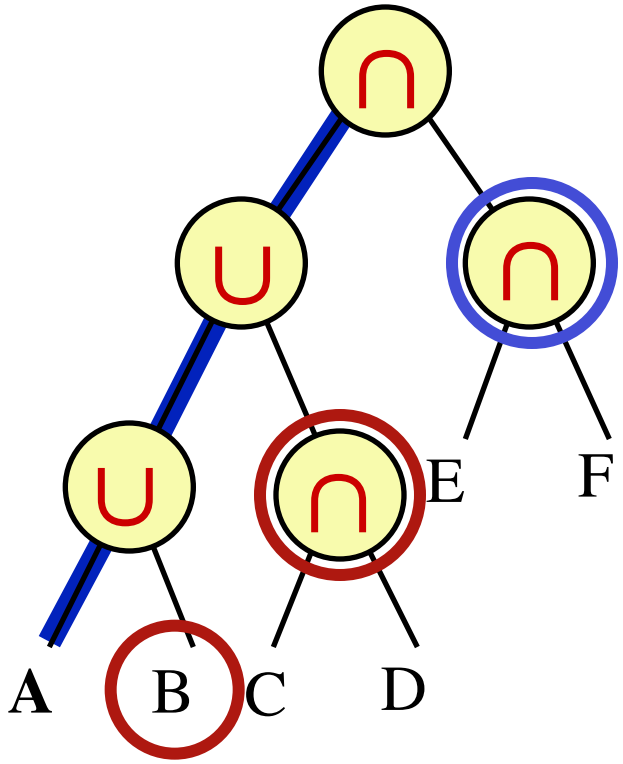
What are **i-nodes** and **u-nodes** of A?

i-nodes = branching nodes that are children of an intersection

u-nodes = branching nodes that are children of a union

Circle the i-nodes of A?

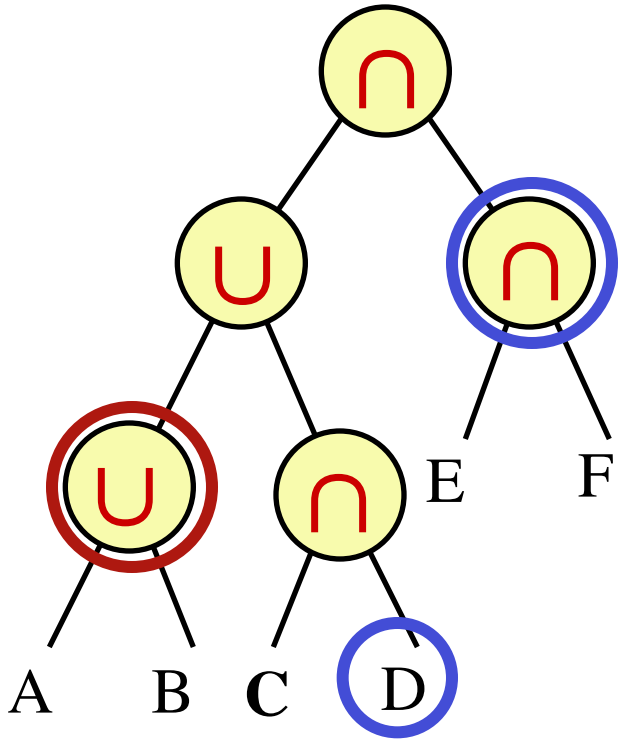
Circle the u-nodes of A?



Practice i-nodes and u-nodes

Circle the i-nodes of C?

Circle the u-nodes of C?



What is the **active zone** of A in S?

The active zone Z of primitive A in a given CSG expression of S is the region in which changes to A affect S.

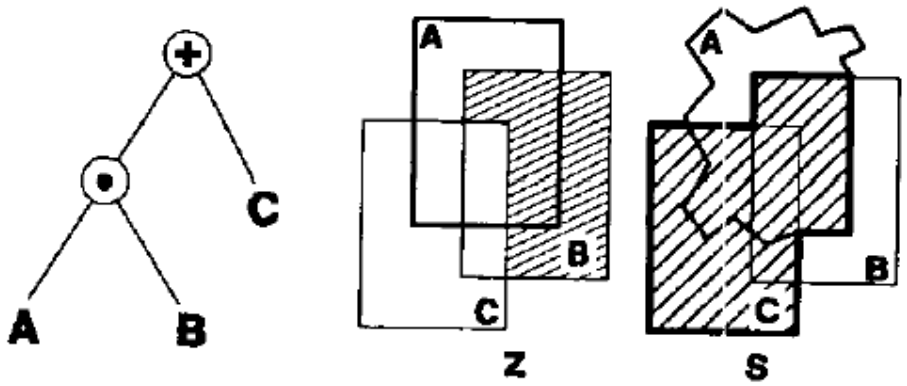


Fig. 7. Simple active zones: In $S = A \cdot B$ (left), the active zone Z of A is B, because modifications to A out of B do not affect S and modifications to A in B affect S. Similarly, in $S = A + B$ (right), the active zone of A is $\bar{B}$.

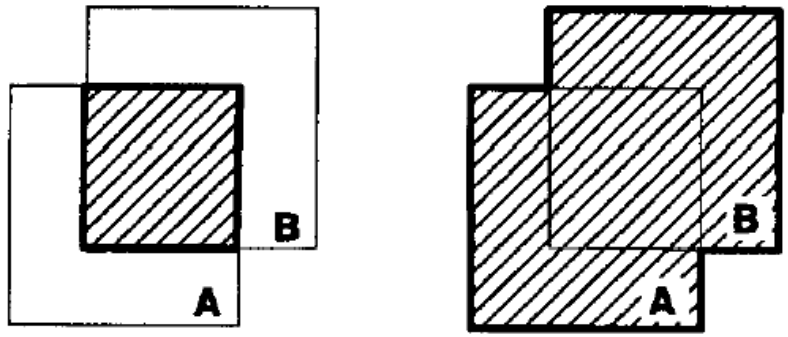


Fig. 8. Combining active zones: In the tree $A \cdot B + C$ of S (left), the active zone of A is $Z = B \cdot \bar{C} = B - C$ (center). Changes to the shape of A outside of Z do not affect S, as shown by the broken boundary line of A outside of Z (right).

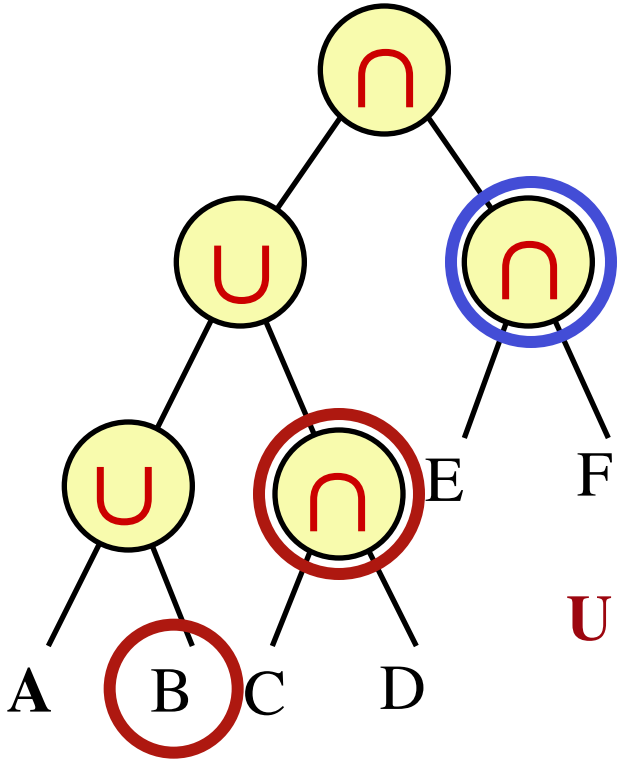
How are I, U, and Z of A defined?

I-zone: I = intersection of the i-nodes **(or the universe w if none)**

U-zone: U = union of the u-nodes

Active zone: Z = I – U

Compute I, U, and Z for A



I = EF

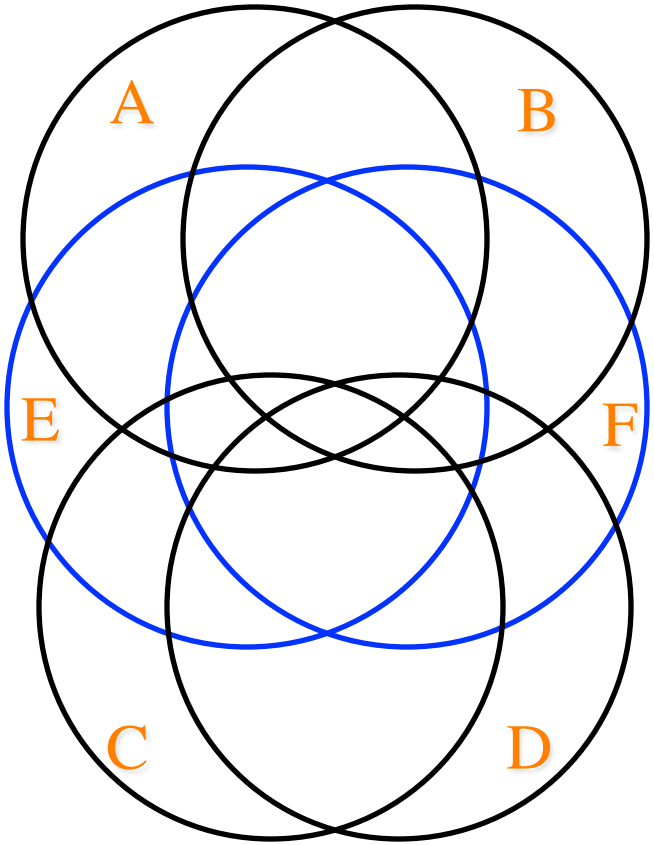
U = B+CD

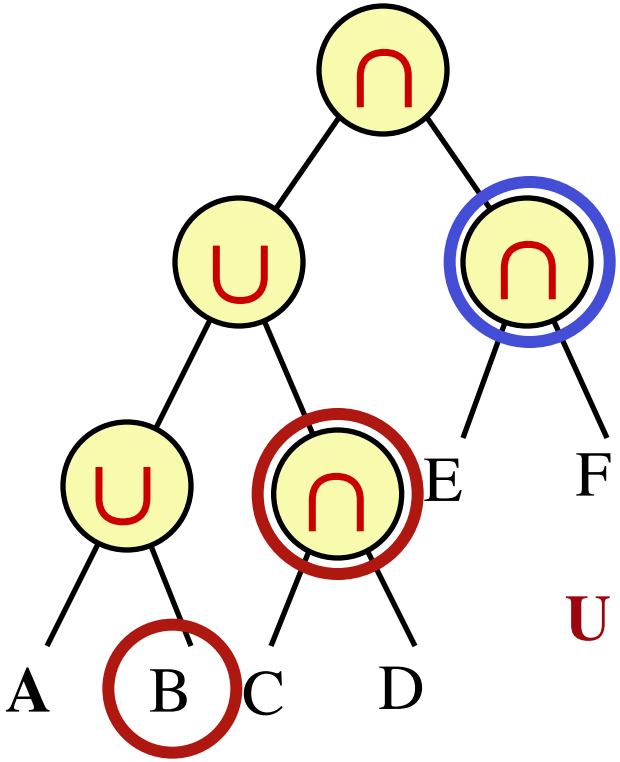
Z = EF – (B+CD) = EF– B –(CD)

Practice: Shade S and Z

IDENTIFY S AND Z for primitive A

Verify that changing A out of Z does not change S





I = EF

U = B+CD

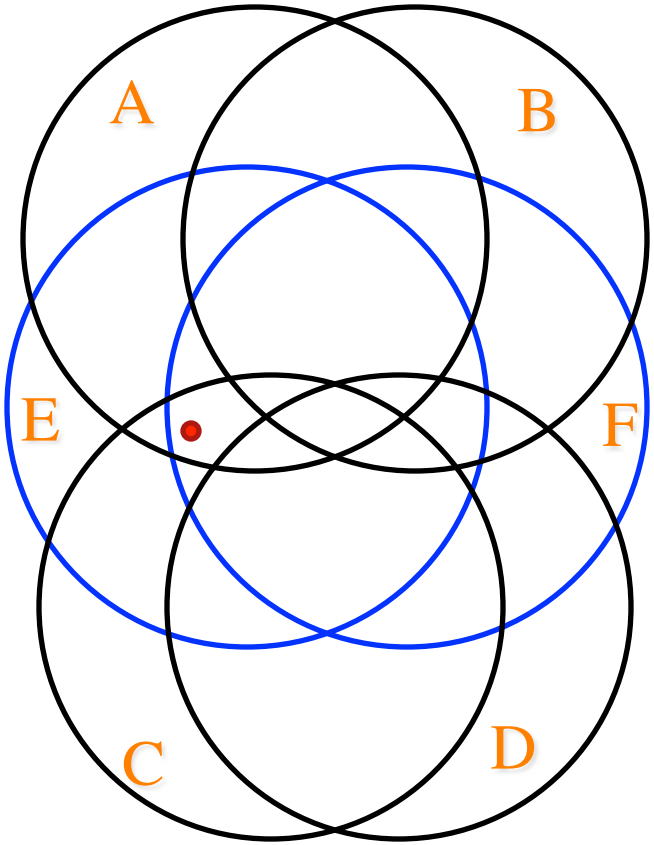
Z = EF – (B+CD) = EF– B –(CD)

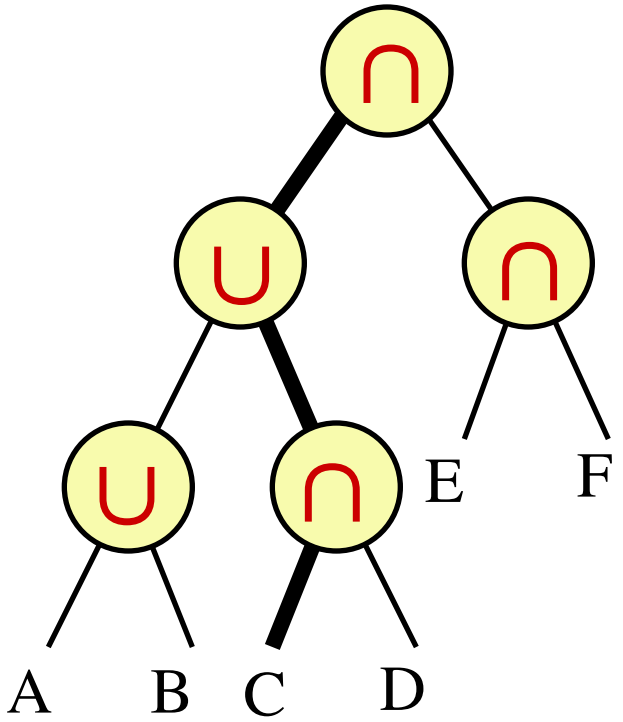
Point-in-solid test on CSG

Classify(point P, node N)

IF N is primitive, THEN RETURN(PMC(P,N))

ELSE RETURN(Combine(N.op, PMC(P,N.left),PMC(P,N.right)));





How to test whether a **surfel is on?**

Surfel = point E on the boundary of a primitive A.

Does it contribute to the boundary of S? Or can A be changed around E without affecting S?

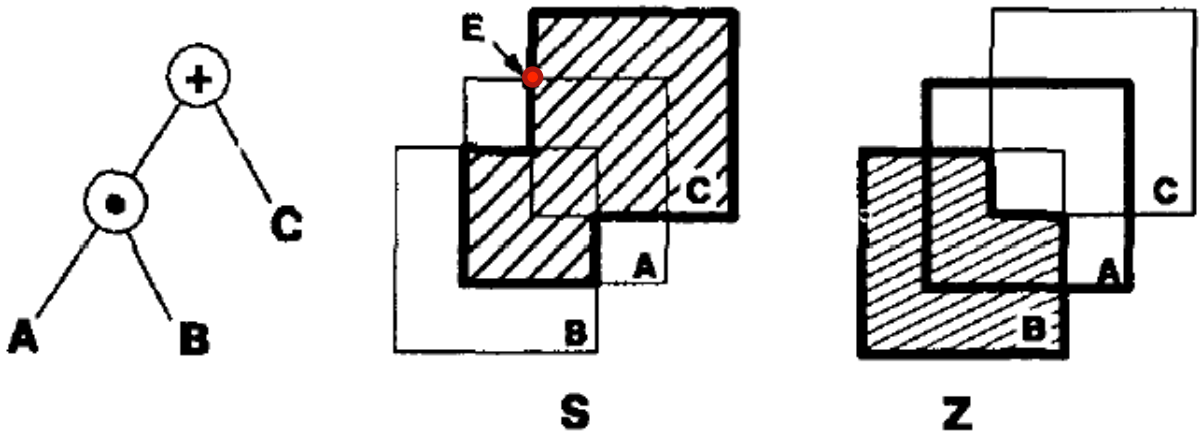


Fig. 10. Redundant boundary element: Given $S = A \cdot B + C$ (left), the point E of ∂A is on ∂S (center). E, however, is out of the active zone, $Z = B - C$, of A in S (right), and thus could be removed from ∂A without changing S.

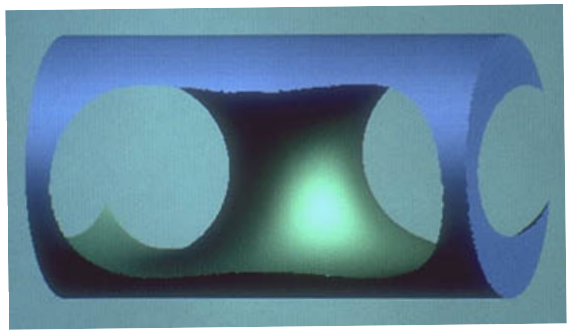
Strategy for CSG rendering

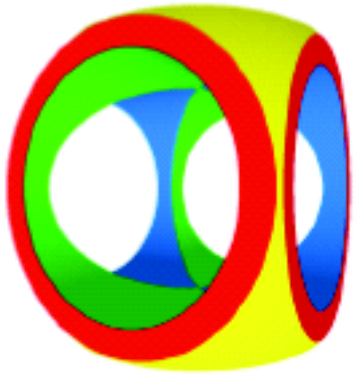
For each primitive A

- Generate a sufficiently dense set of surfels on the boundary of A

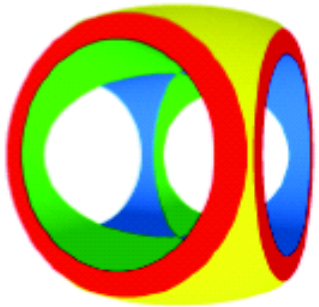
  For a cylinder, place surfels on a circle and slide the circle …

- Classify surfels against Z
- Render those in Z





Blister: GPU-based rendering of Boolean combinations of free-form triangulated shapes

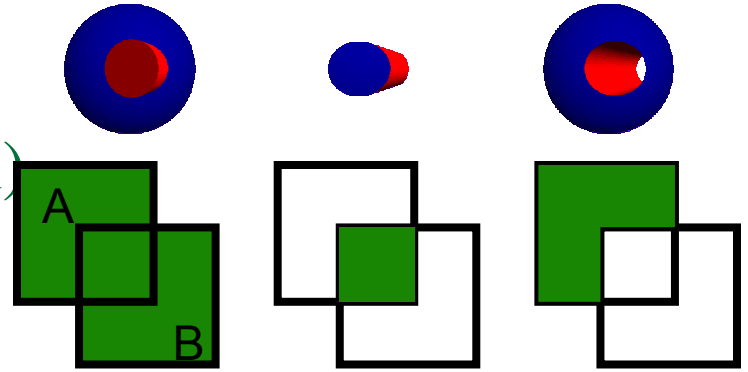


John Hable & **Jarek Rossignac**
College of Computing, IRIS cluster, GVU Center
Georgia Tech, Atlanta, USA

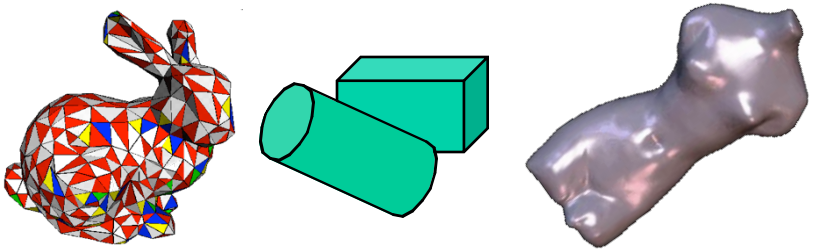
http://www.gvu.gatech.edu/~jarek/blister/

CSG (Constructive Solid Geometry)

- Operators:
  - A+B (Union)
  - AB or A*B (Intersection)
  - !A (Complement)
  - A–B=A!B (Difference)

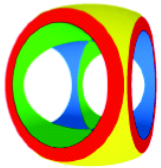


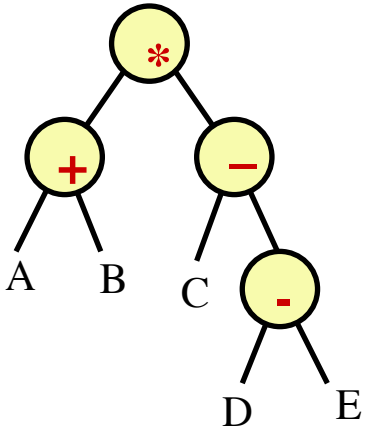
- Primitives (solids):
  - **Bunnies, Triangle meshes**
  - Natural quadrics
  - Subdivision shells



Water-tight boundaries

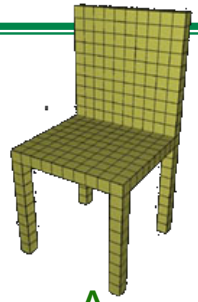
- CSG expression: (A+B)(C– (D–E))
  Define solids
  Parsing them yields a binary tree

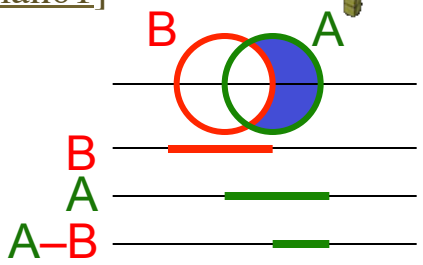


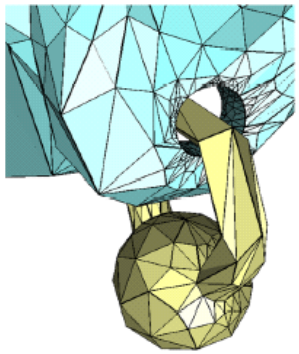


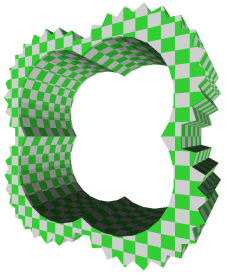
Examples of prior approaches

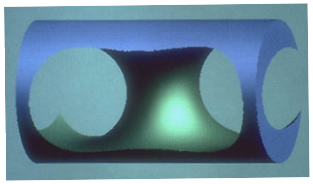
- 3-D: **Voxels** (classify wrt primitives, merge)
  [Meagher84, Breen89, Fang98, Chen00]
- 2-D: Trimmed **faces** (boundary evaluation)
  [Mantyla86, Hoffmann89, Krishnan97, Bierman01]
- 1-D: **Rays** from eye through pixels
  [Roth82, Ellis91, Pfister00]
- 0-D: Surface **samples**
  - Model space (**surfels**)
    [Rossignac86, Bronsvoort87, Breen91]
  - Image space (**fragments**)
    [Okino84, Sato85, Goldfeather86-89, Epstein89, Wiegant96, McReynolds96, Jansen95-87, Rossignac92, Stewart98, Fang00, Stewart02]







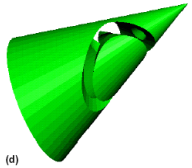




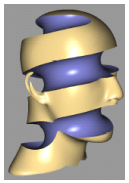
→ **Disjunctive form: 31,104 products**

- Mixed

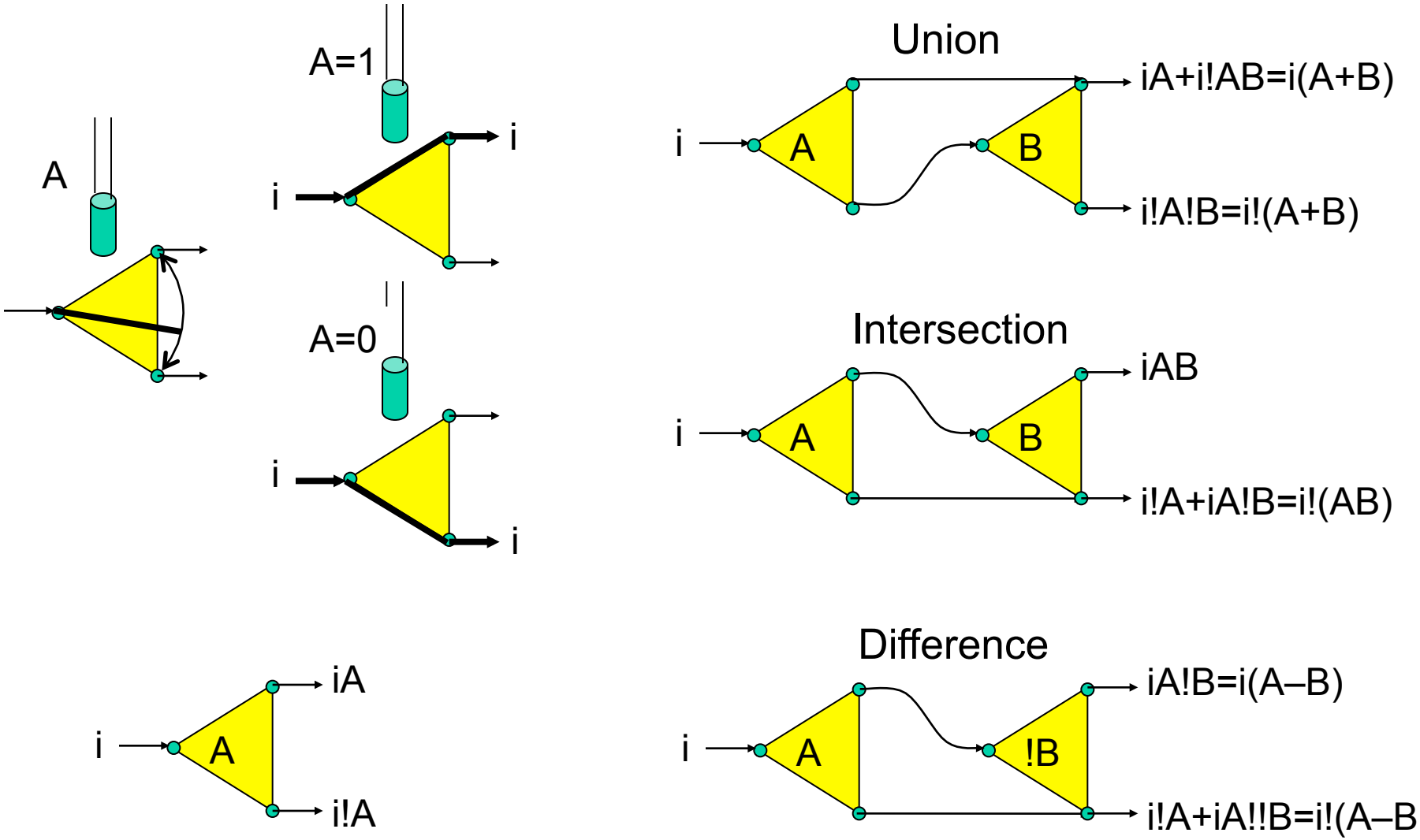
Faces+fragments [Rappoport97]



Samples+voxels [Adams03]

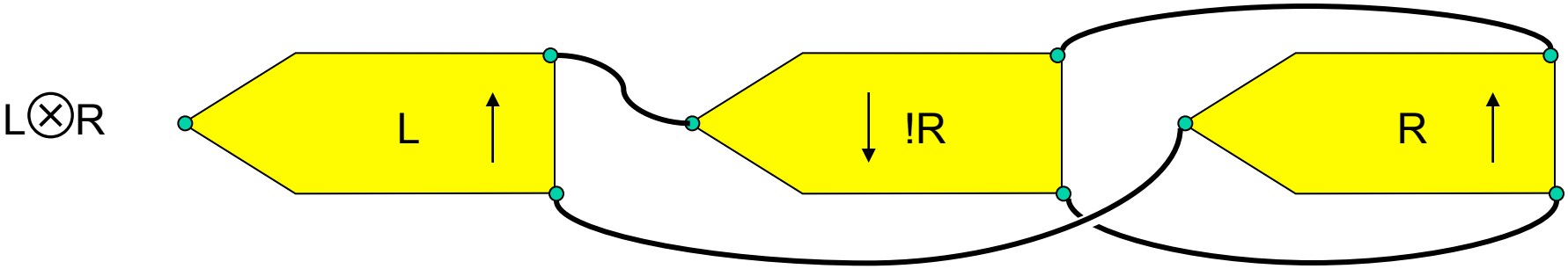


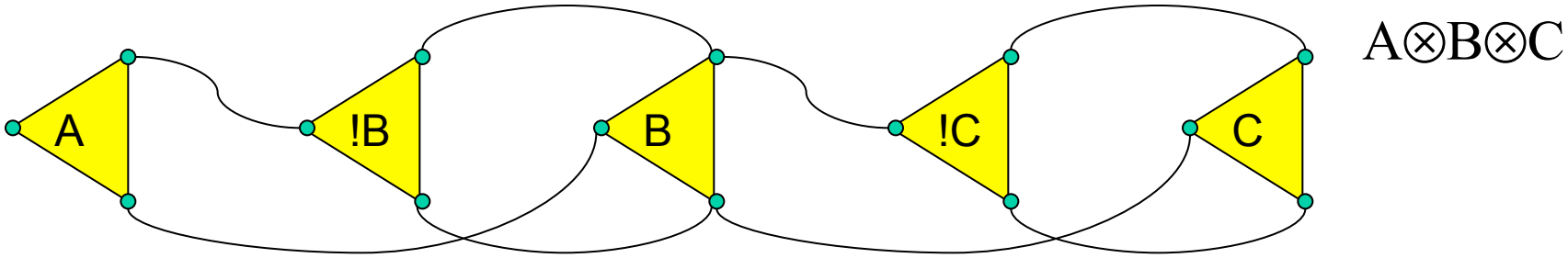
Blist form of a CSG tree



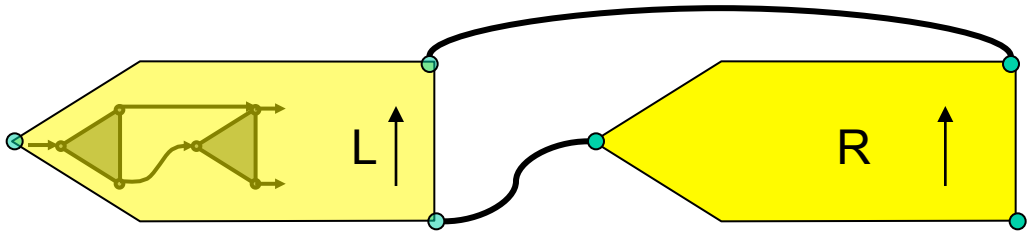
Blist of a symmetric difference

Simple to support, but reduces the size of the worst case CSG models that we can handle.

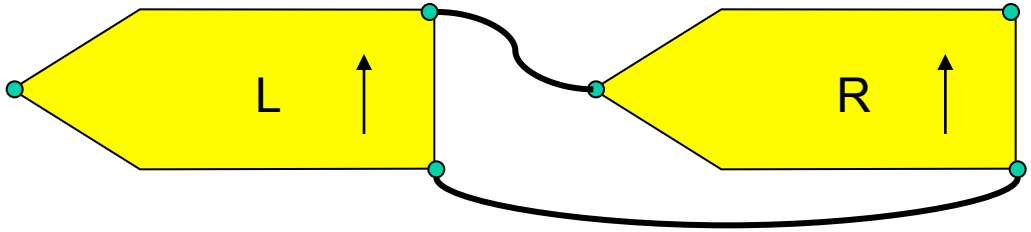




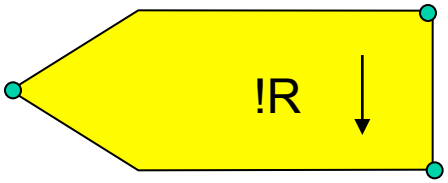
Merging Blist sub-expressions



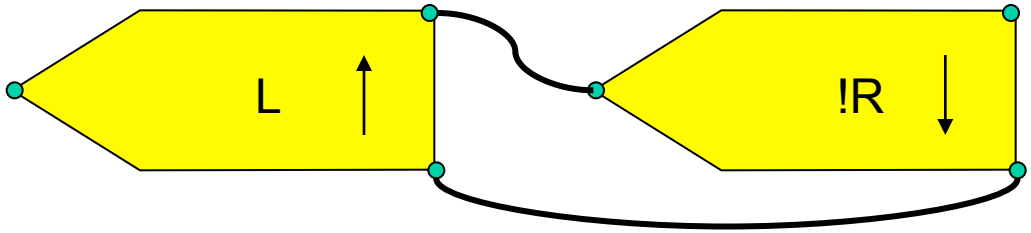
Union: L+R



Intersection: LR

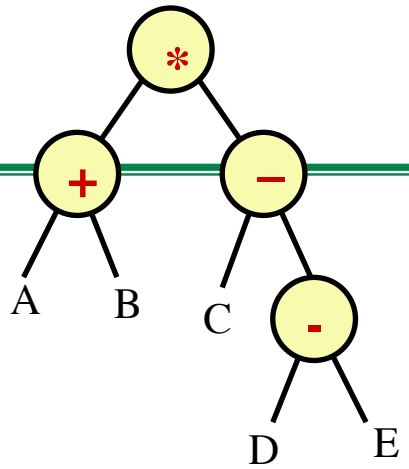


Complement: !R

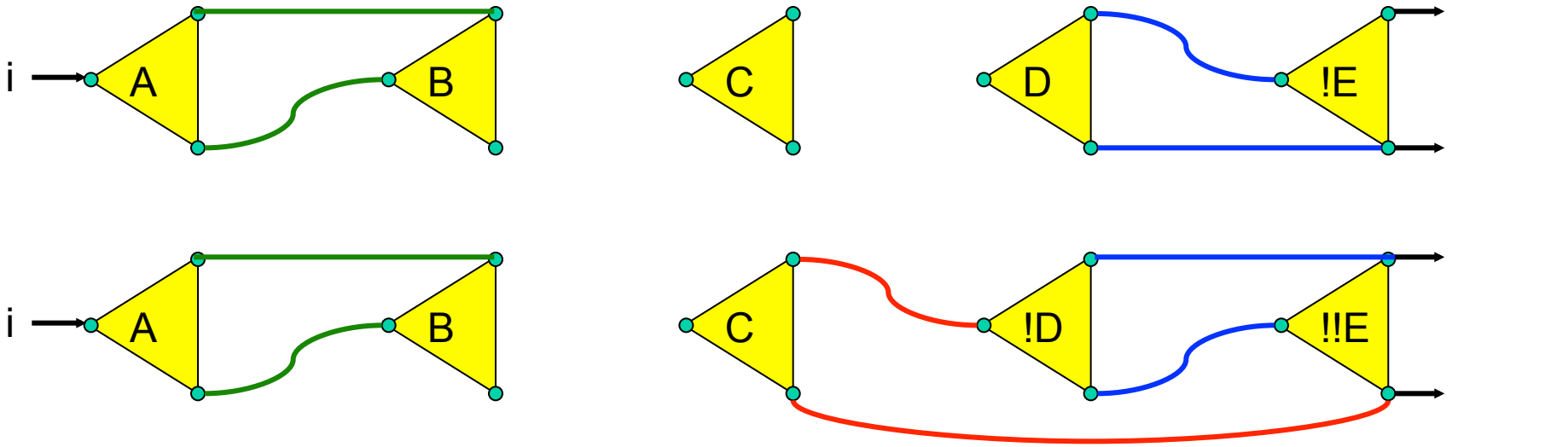


Difference: L–R=L!R

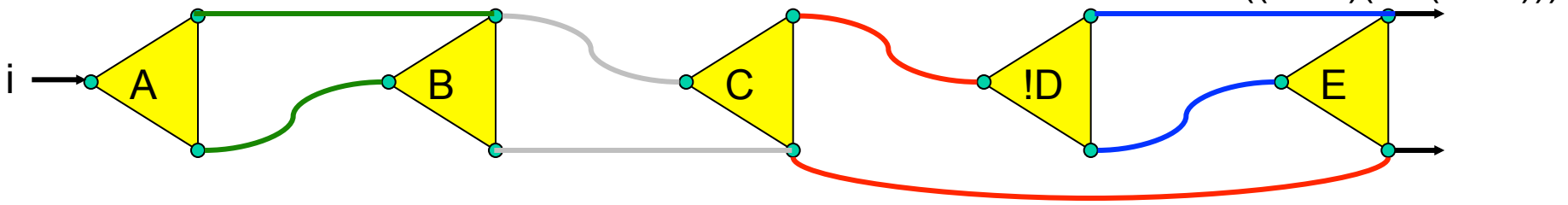
Example of Blist



(A+B)*(C–(D–E))



i((A+B)(C–(D–E)))



i!((A+B)(C–(D–E)))

Use a positive CSG form (for simplicity)

negative (complemented) primitive

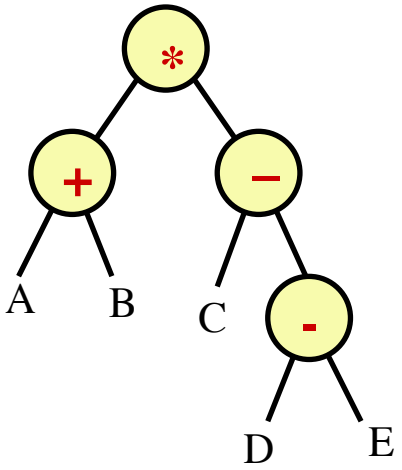
A–B=A!B

!(A+B)=!A!B

!(AB)=!A+!B

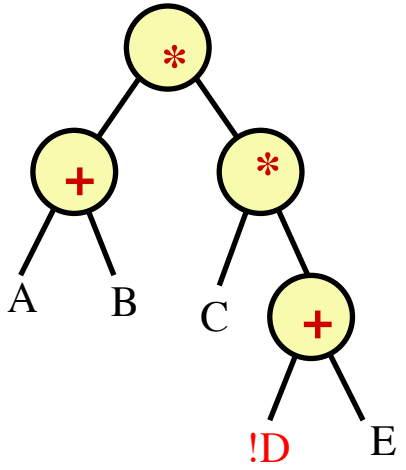
!!A=A

(A+B)(C–(D–E))



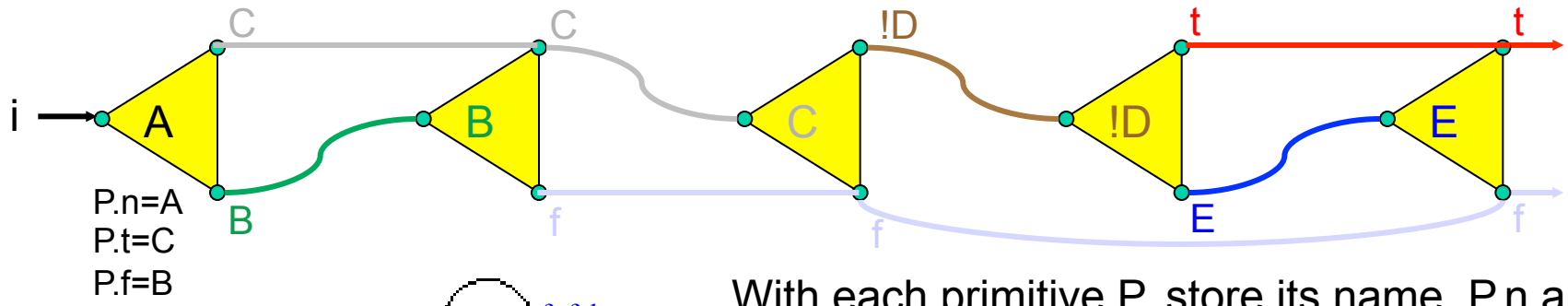
Conversion to positive form is a trivial recursion

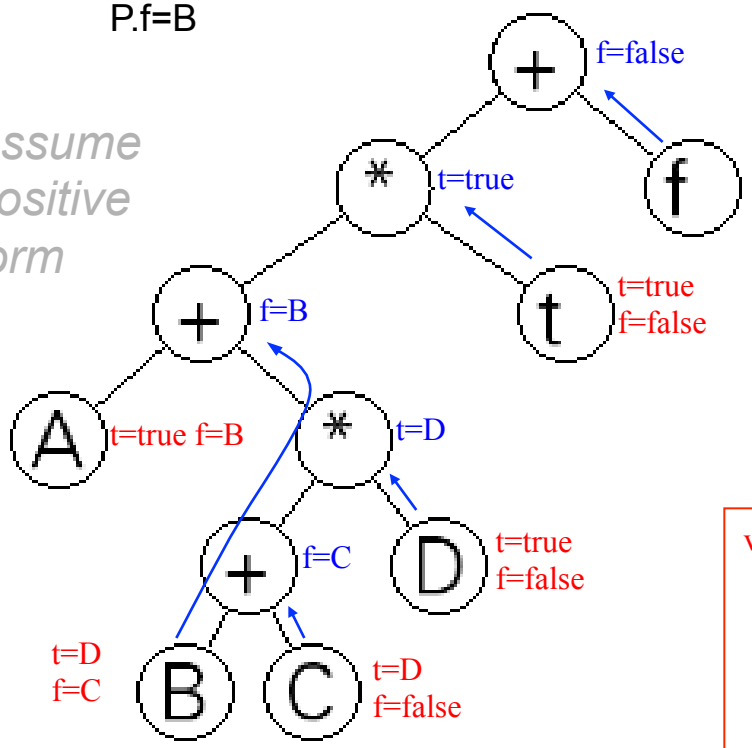
(A+B)(C(!D+E))



Algorithm for computing Blist labels

Assigns to each link the label (color) of its final destination primitive





Assume positive form

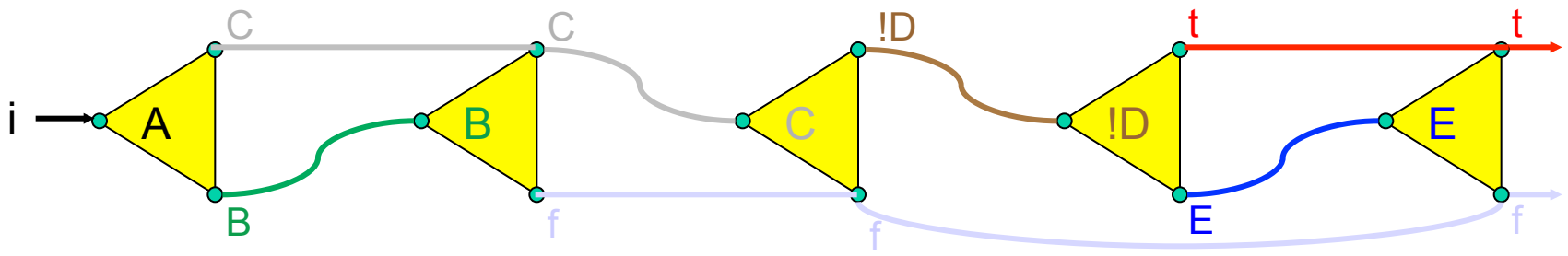
With each primitive P, store its name, P.n and its P.t (upper) and P.f (lower) labels

```
char lml (int m) {              # left most leaf of right child
  char leftMost;
  if((o[m]!='*')&&(o[m]!='+')) {leftMost=o[m];}
  else {if (o[m]=='+') {f[m]=lml(r[m]); }
        else {t[m]=lml(r[m]); }; leftMost=lml(l[m]); };
  return(leftMost); }
```

Going right: inherit t and f. Going left: pick up blue value

```
void mb (int m, char pt, char pf) {
 if((o[m]!='*')&&(o[m]!='+')) {
   blistName[pc]=o[m];  blistIfTrue[pc]=pt;  blistIfFalse[pc]=pf;
   t[m]=pt;  f[m]=pf;   pc++;}
  else {if (o[m]=='+') {mb(l[m],pt,f[m]); } else  {mb(l[m],t[m],pf); };
       mb(r[m],pt,pf); };        }
```

Point-in-CSG classification using Blist



Workspace: Boolean parity; 6-bit integer next;

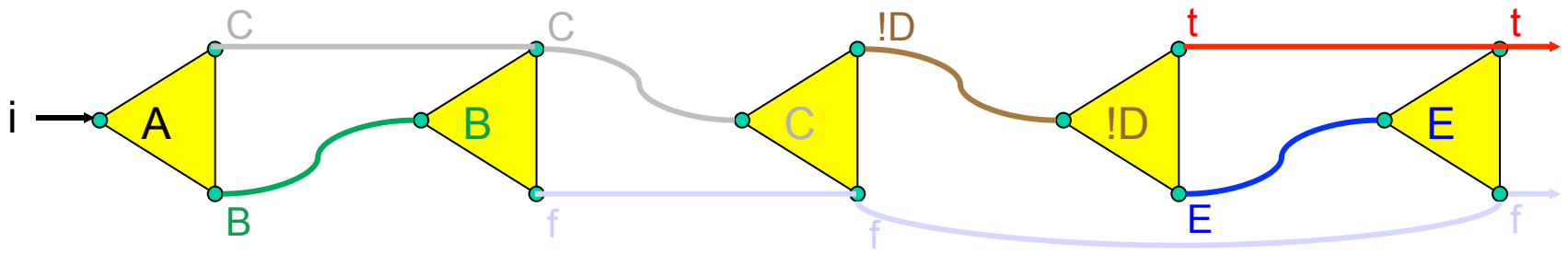
For each primitive P in Blist do:

parity := true if point is in-or-on P;

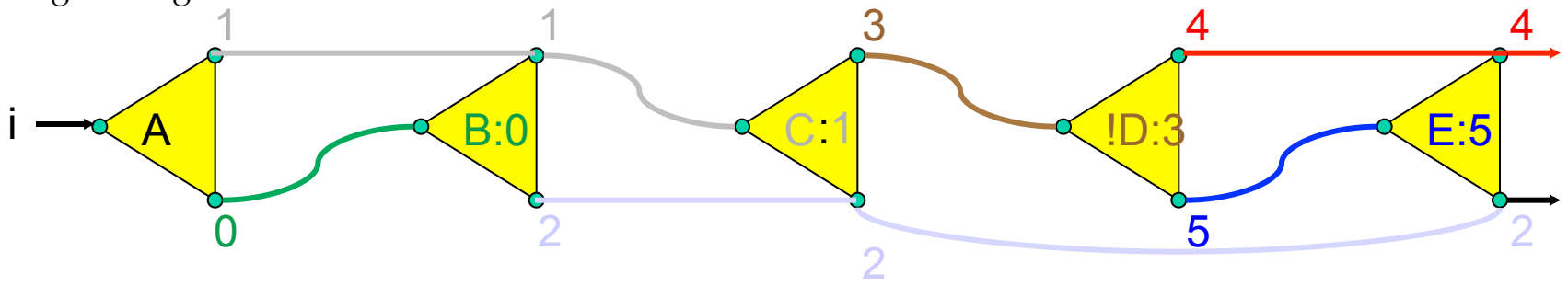
If (next==P.n) {if (parity) {next=P.t} else {next=P.f}};

The final content of next indicates whether the CSG expression is true or false

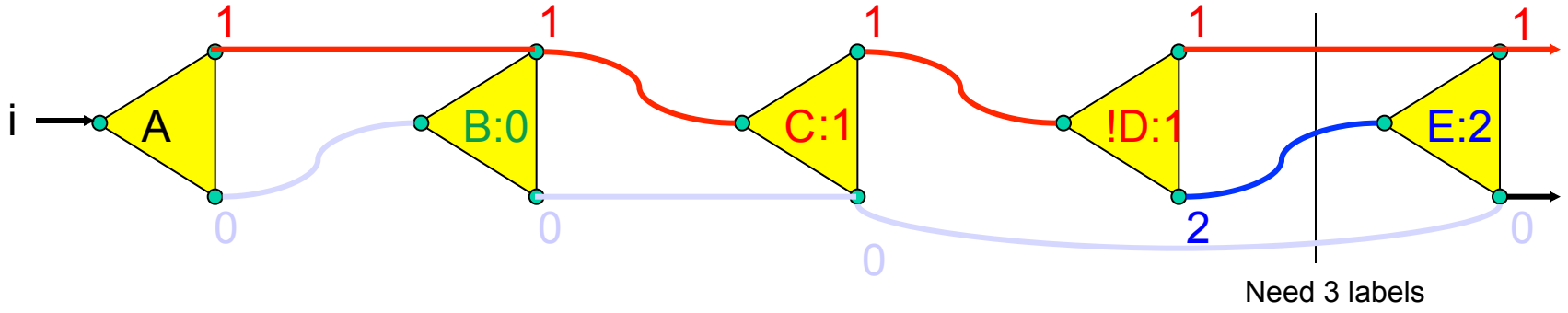
We can only afford 6-bit labels



Assign integers as needed.



Reuse labels of primitives that have already been reached.

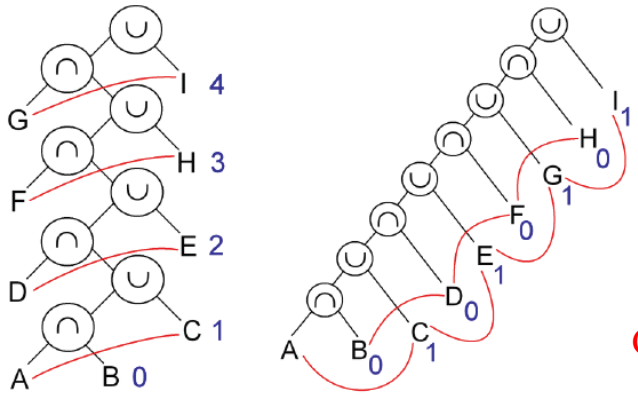


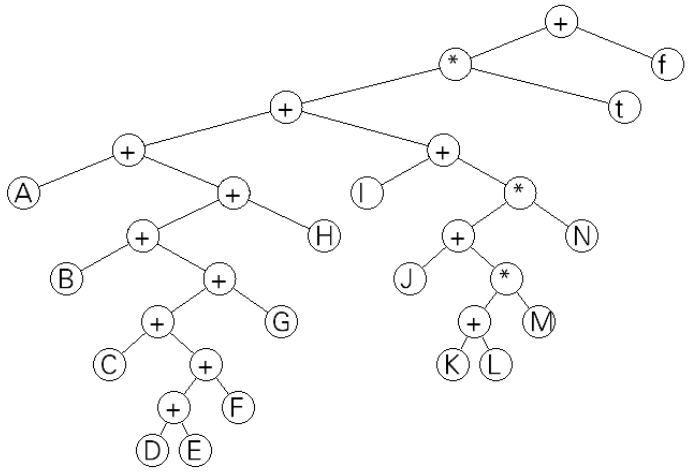
Minimizing the number of labels

Original: (A+((B+((C+((D+E)+F))+G))+H))+(I+(J+(K+L)M)N): 5 labels

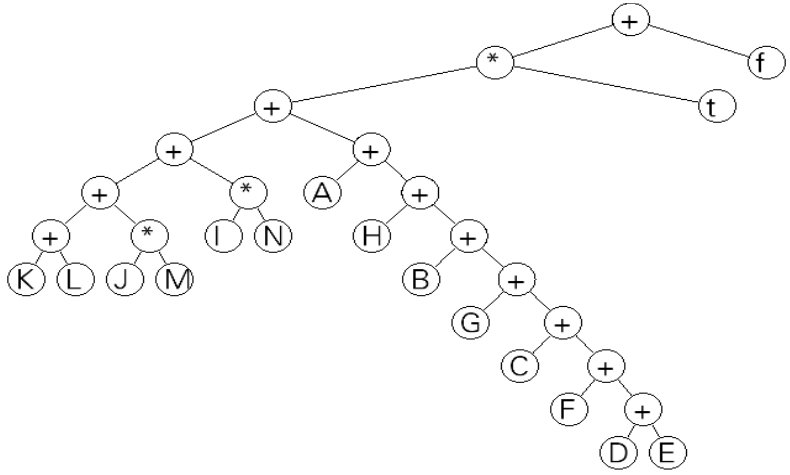
Can swap left and right arguments.
Strategy: swap to make tree **left-heavy**

A+B → B+A, AB → BA





Cost swaps: (K+L)+JM+IN+(A+(H+(B+(G+(C+(F+(D+E))))))) :3 labels



There are 64 6-bit labels. We prove that no CSG tree with < **3909** primitives requires more than 64 labels

Recently developed **linear optimization**:
No CSG trees with < 2^{63} primitives requires more than 64 labels.

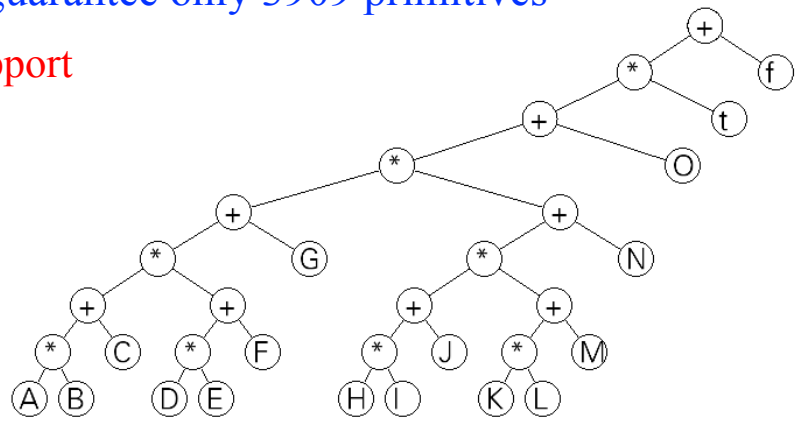
Improving Blist worst-case footprint

Height-based pivoting (described in the paper) could guarantee only 3909 primitives

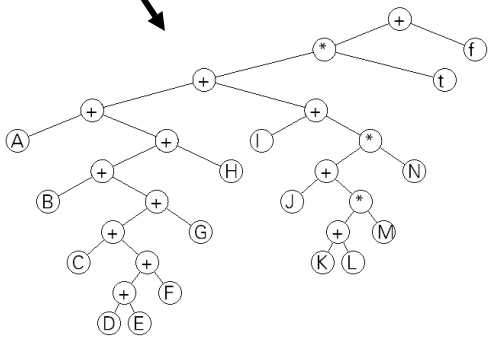
New **cost-based pivoting** with 6 stencil bits guarantees support for all CSG trees with less than 2^{63} primitives!

In general b stencil-bits support $n=2^b$ labels and n labels suffice for all Blist expressions with up to $p=2^{n-1}-1$ primitives

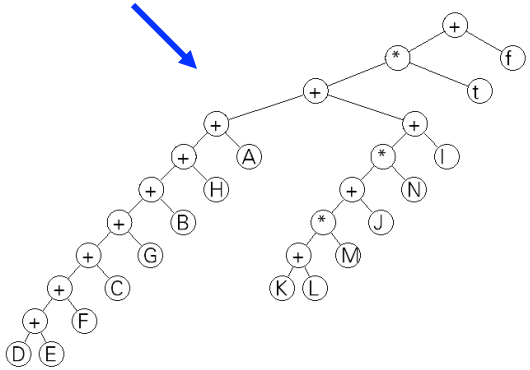
Smallest expression requiring 5 labels has 15 primitives: ((AB+C)(DE+F)+G)((HI+J)(KL+M)+N)+O

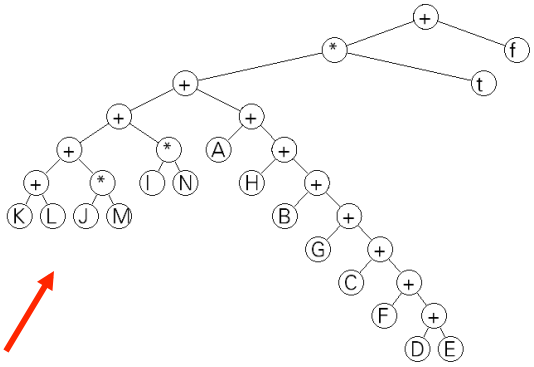


Original (5 labels): (A+((B+((C+((D+E)+F))+G))+H))+(I+(J+(K+L)M)N)



Height swaps (4 labels): D+E+F+C+G+B+H+A+(((K+L)M+J)N+I)



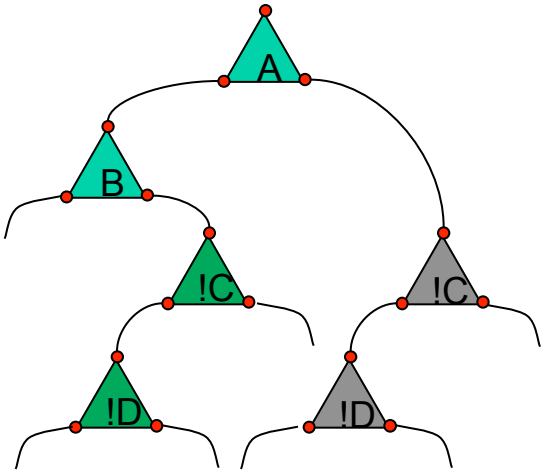


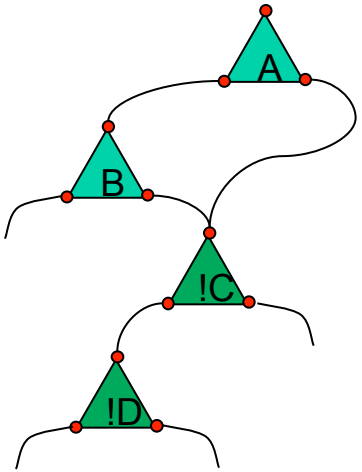
Cost swaps (3 labels): (K+L)+JM+IN+(A+(H+(B+(G+(C+(F+(D+E)))))))

Blist is a very special decision graph

- Boolean functions: many representations [Moret82]
- Binary Decision Graphs (BDG) [Akers78]
- Reduced Function Graphs (RFG) [Bryant86]
  - Acyclic BDGs without repetition
  - Constructed through Shanon's Expansion [Shannon38]
  - Removing empty leaves yields BSP trees
  - Reduction: recursively tests graph isomorphisms
  - # nodes may be exponential in # primitives
  - Size depends on order [Payne77] (optimal is NP-hard)
  - Footprint is $\log_2 N$ bits
- Blist [Rossignac99]
  - RFG of a Boolean expression with +, *, – operators
  - Linear construction & optimization cost
  - # nodes = # primitives = N
  - Footprint may be compressed to $\log_2 \log_2 N$ bits

(A+B)–CD=(A+B)(!C+!D)
!A(B(!C+!D)+A(!C+!D)
!A(!B+B(!C+!D)+A(!C+!D)
!A(!B+B(C!D+!C)+A(C!D+!C)

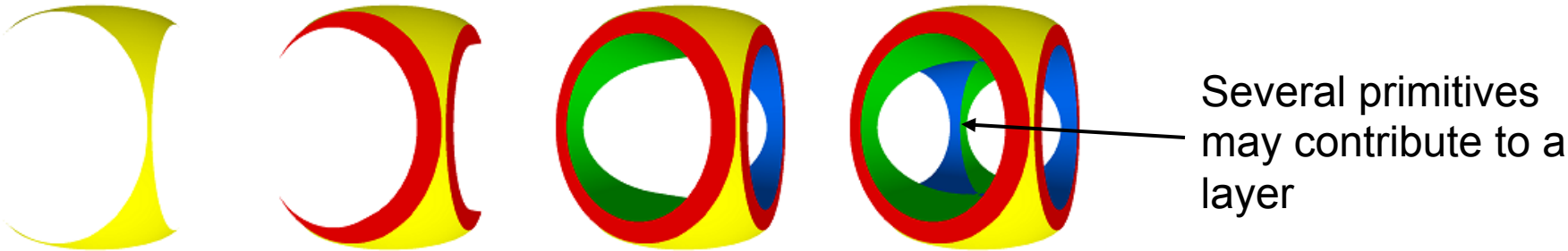




Blister Overview

- Repeat until all pixels are **locked**
  - **Peel**: Push the unlocked fragments to the next **layer**
    - Scan each primitive: compute foremost fragment behind current layer
  - **Lock**: Identify and **lock** fragments **on** the CSG solid
    - For each primitive P:
      - Classify active fragments by toggling parity bit
      - For all active pixels: Update the Blist next value

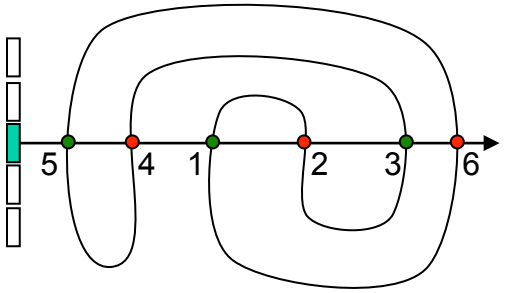
Contributions of layers (front-to-back)



Peel strategy

- UNC: Peel each primitive with a counter

  [Goldfeather-Hultquist-Fuchs SIG' 89]



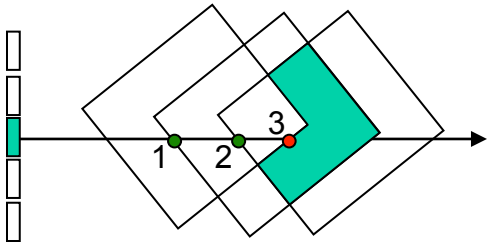
- IBM+: Peel a product with a Depth-Interval Buffer?

  F=previous layer, B=infinity

  If (F<z<B) {B:=z}

  [Epstein-Jansen-Rossignac' 89]

  [Stewart-Leach-John' 98, Guha et al.' 03]

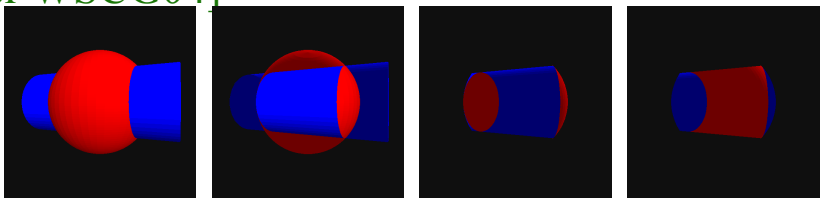


- GaTech: Peel it all using texture memory

  [Mammen CG&A89, Everitt' 02, Kirsch&Döllner WSCG04]

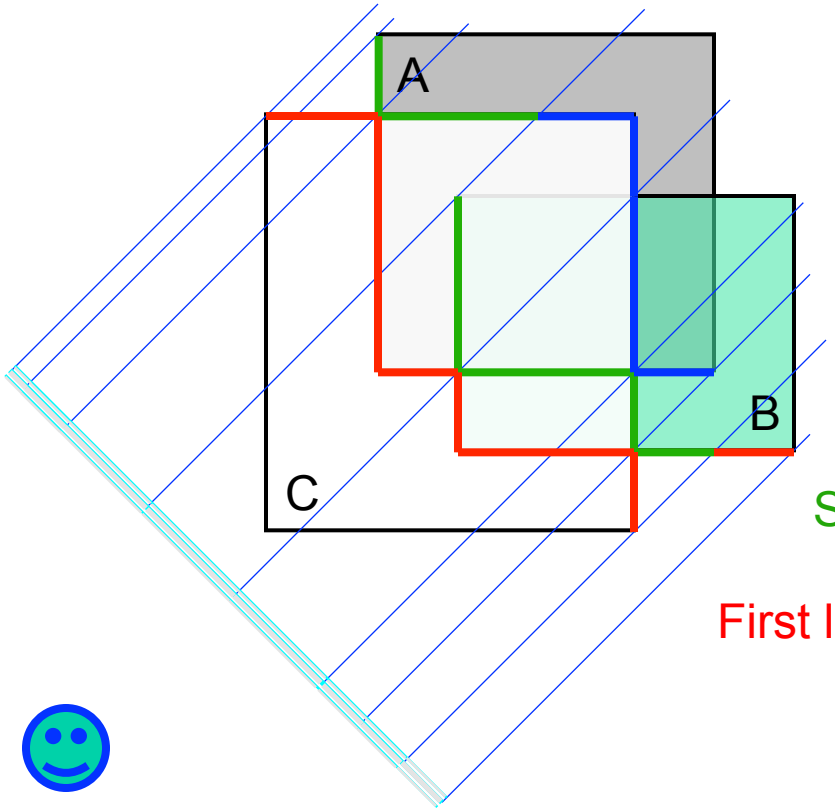
  Store F as depth texture

  B:=closest fragment behind F



Peel example

Peel the arrangements of **all primitives** taken together
For each layer, scan the front/back faces of positive/negative primitives
Use a Depth-Interval Buffer to advance the layer [Epstein-Jansen-Rossignac' 89]
Produces **layers** of fragments in front-to-back order



(A+B)–C

Negative primitive (A+B)C!

Third layer fragments

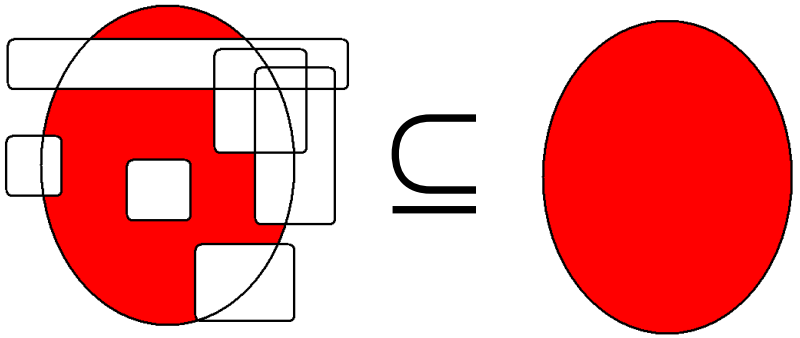
Second layer fragments

First layer fragments

Peeling: Implementation details

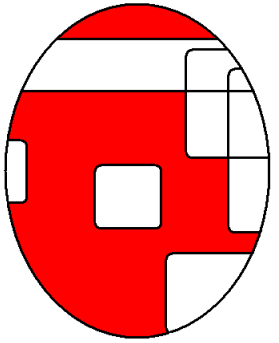
Initialization:

- Compute (small) set of primitives that cover the CSG solid
  - A–B →A, A+B→A+B, AB → smaller of A and B
- Store front F and back B' of the covering set in texture memory
  - Render with GL-LT and GL-GT



For each layer:

- F contains previous layer depth
- B is initialized to infinity
- Render primitives
  - In the pixel-shader (z=depth of fragment)
    - If (z<F || B' <z) discard fragment
    - If (pixel is locked) discard fragment
  - Standard depth-test on remaining fragments:
    - If (z<B) {B:=z}

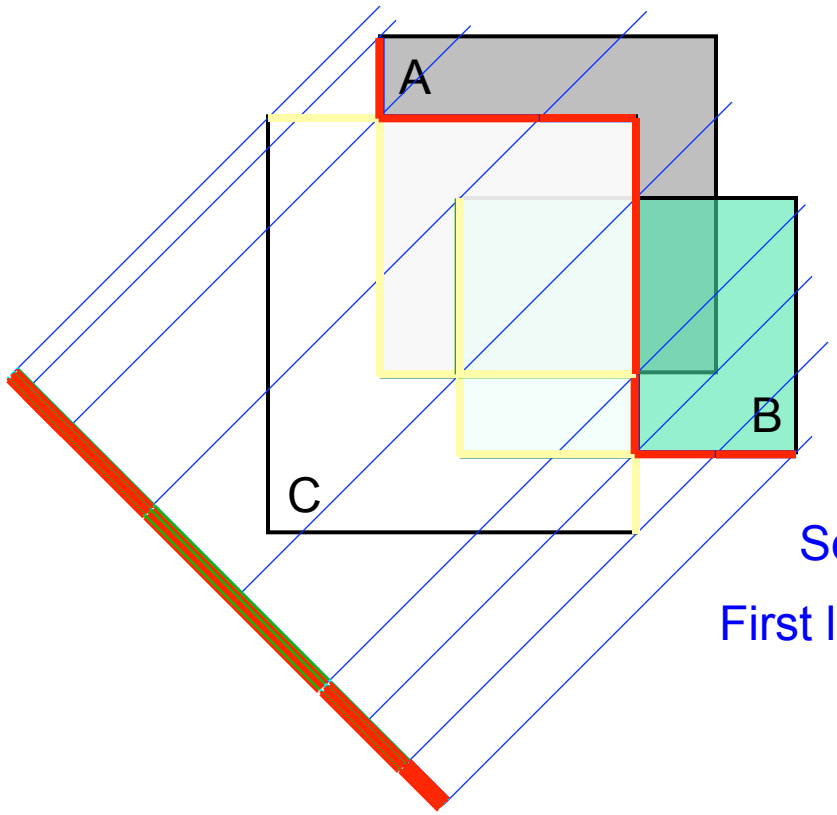


Peel & lock example

Each **new layer** is **trimmed**

Pixels with fragments **on** the CSG solid S are **locked**

The next layer is only computed at the **active** (unlocked) pixels



(A+B)–C

Third layer locked

Second layer locked

First layer locked

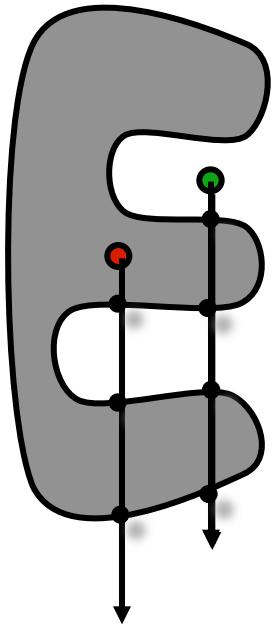
Fragment/primitive classification

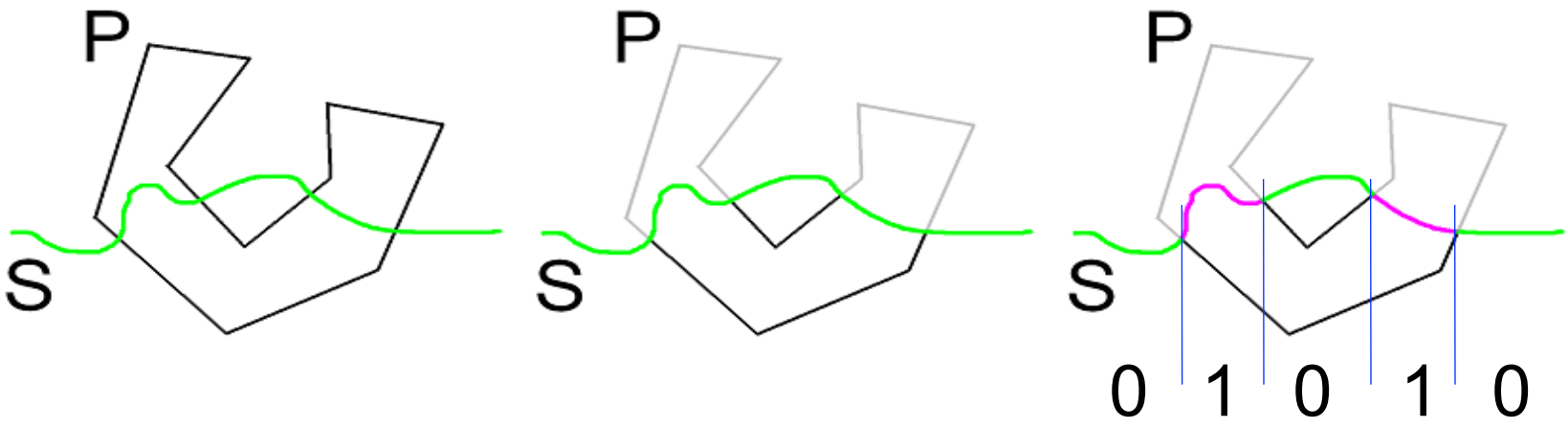
We want to classify the fragments of each active pixel q in the current layer whose depth is S[q]

Scan P generating fragments (pixel q, depth z)

- If ((q is active) && (z<S[q])) {parity[q]=parity[q]!}

Fragments of odd parity pixels are inside P.



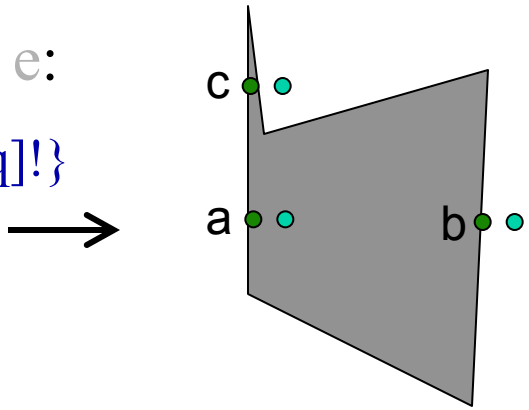


Classifying fragments on P

Prior art tested ghost fragment moved back by e:

If ((q is active) && (z<S[q])+e) {parity[q]=parity[q]!}

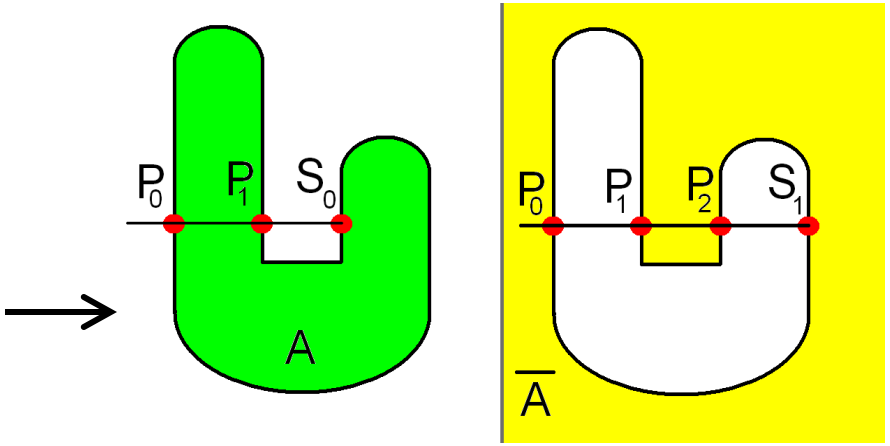
[Rossignac-Wu92]



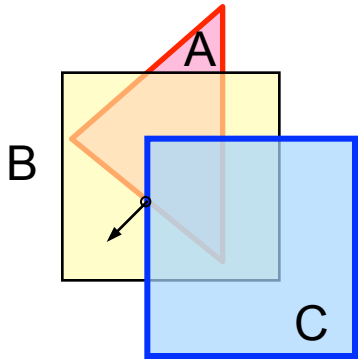
Possible artifacts in thin areas

We solve on-cases without fake displacement: Use GL_LEQUAL

Fragments on front-faces of positive primitive P or on the back-face of a negative primitive P must be classified as in-or-on P



ON/ON cases



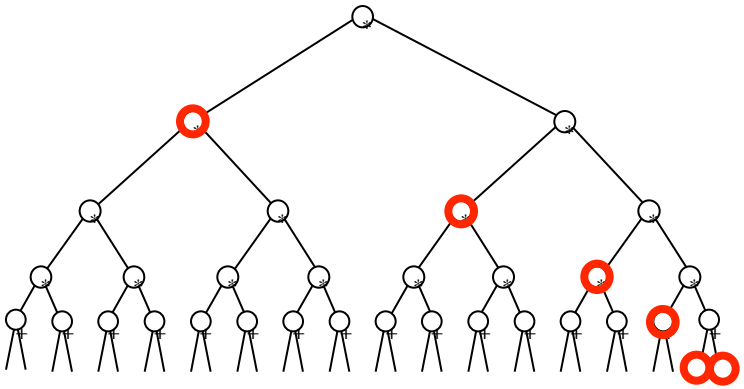
Can't remember past classifications?

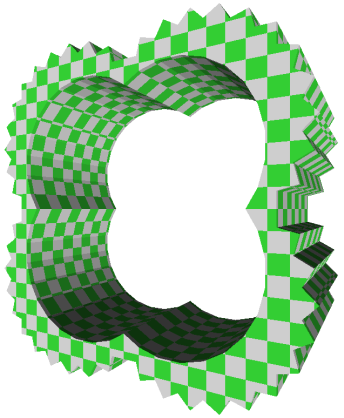
Only **6** stencil-bits per pixel to keep past classification results

- Storing Boolean for **each primitive**: **< 7** primitives
- Storing a **stack** of partially evaluated results **< 65** primitives
- Using a **disjunctive** form of CSG tree: no limit on N

  Goldfeather86-89, Epstein89, Wiegant96, Rossignac92, Stewart98, Stewart02…

  But… the number of products can grow exponentially!





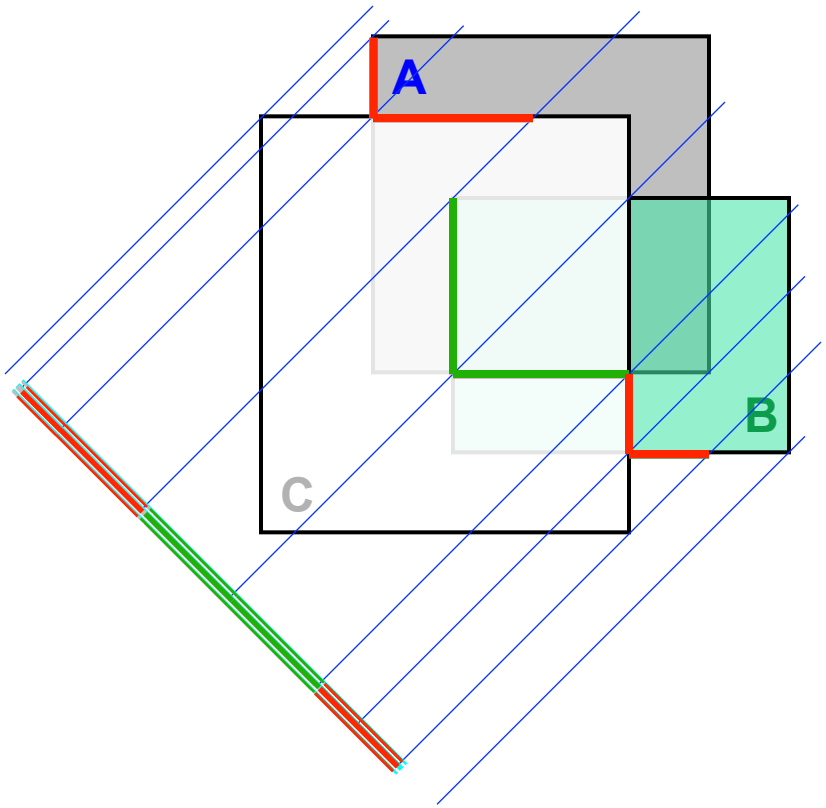
CSG expression has 48 primitives. Disjunctive form has **31,104 products** with 5 primitives per product

(No pruning possible)

So… use Blist instead!

Lock for layer 2 in the example

Fragments at all active pixels are classified in parallel against each primitive.
Their **stencils** are updated (based on the classification and their previous value).
After the last primitive, fragments on the CSG solid are locked, others stay active.



(A+B)–C

Active fragments of layer 2

Scan A

Out of A (for B)

In-or-on A (for C)

Scan B

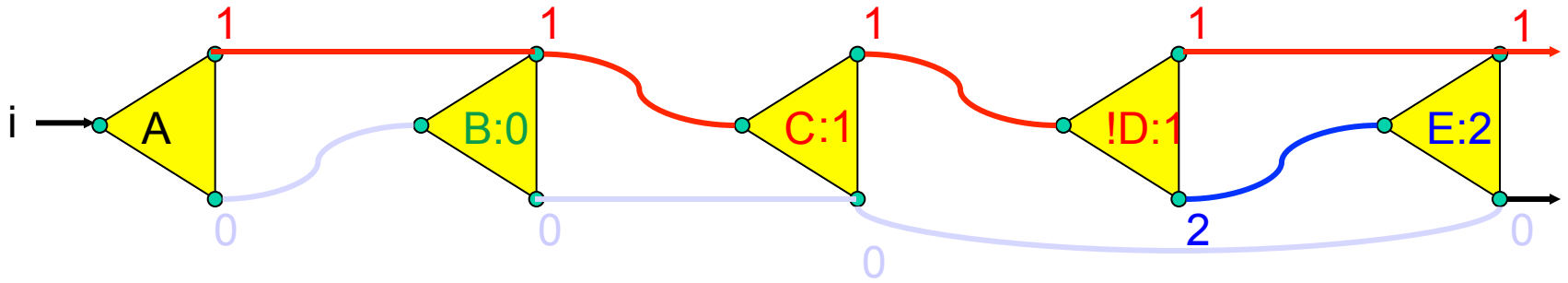
In-or-on B (for C)

Scan C

In-or-on C! (locked)

Out of C (active)

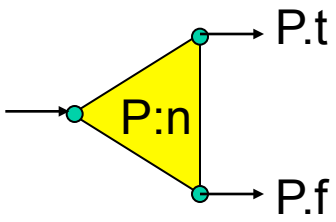
Use Blist to merge past classifications



Initialization:

Rearrange the CSG tree to minimize the number of labels

With each primitive, P, associate its 3 Blist labels: P.n, P.t, P.f



In the **stencil** of each pixel store: parity (bit-0), next (bits 1-6)

To classify and lock the fragments of the current layer

For each primitive P in Blist do:

Scan P {toggling parity when P is in front of the tested fragment}

Scan all active pixels {If (next==P.n) {if (parity) {next=P.t} else {next=P.f}}}

Updating the stencil bits of *next*

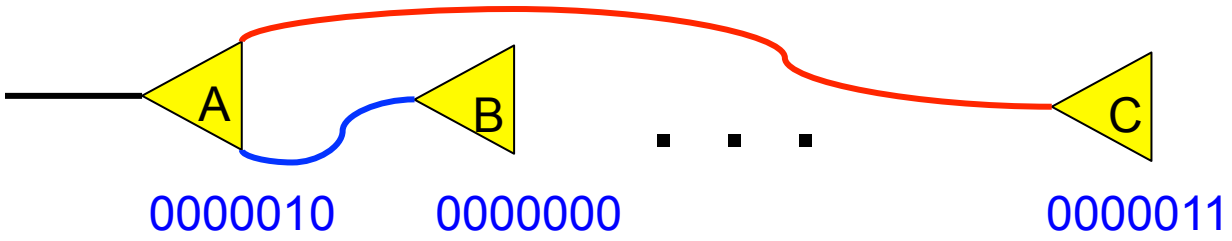
// Stencil=(parity-bit,next)

// if (parity-bit==1 and *next*==P.n) then *next*:= P.t

IF (Stencil==10000010) {Stencil:= 10000011;}

// if (parity-bit==0 and *next*==P.n) then *next*:= P.f

IF (Stencil==00000010) {Stencil:= 00000000;}

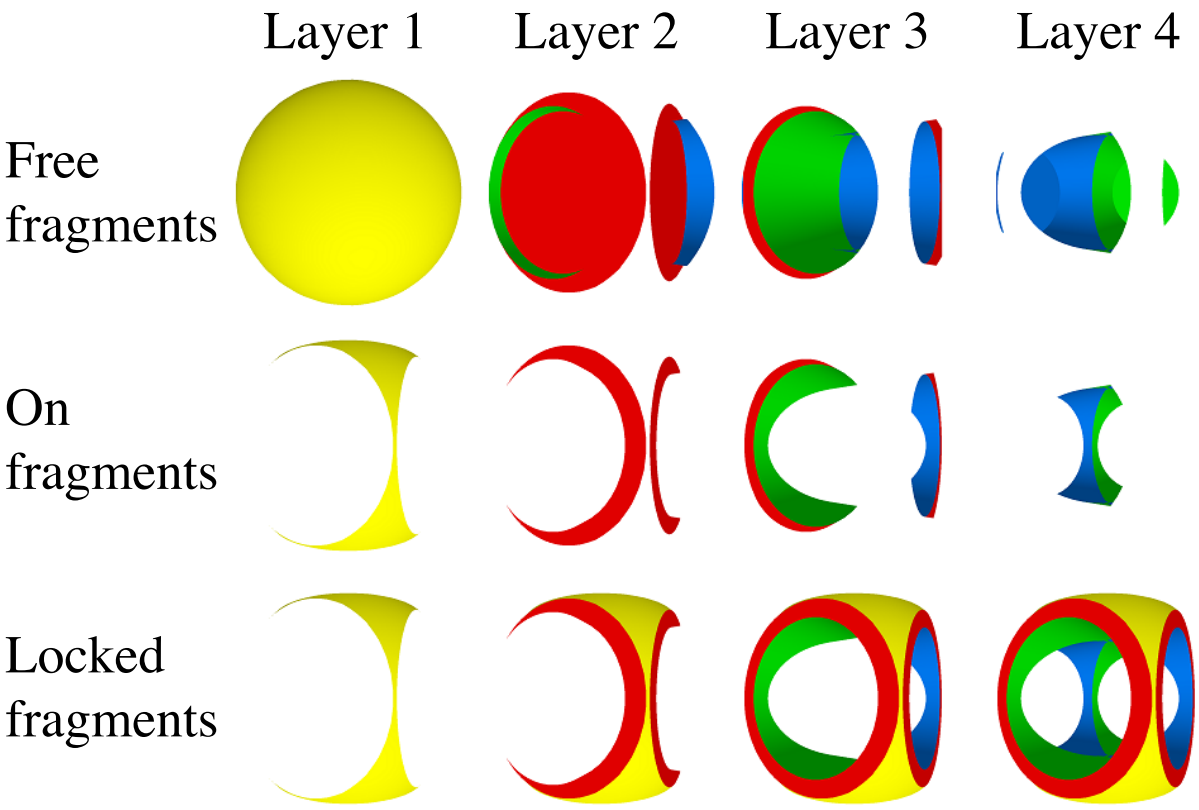


Speed-up suggested by **Mark Kilgard** (nVidia):

Mask=*next* XOR P.t; Then toggle next where it agrees with Mask

```
glStencilMask( P.t ~ next );
glStencilFunc( GL_EQUAL, P.t | 0x80, 0xff );
glStencilOp( GL_KEEP, GL_KEEP, GL_INVERT );
```

An example


Layer 1
Layer 2
Layer 3
Layer 4
Free fragments
On fragments
Locked fragments

Timing results

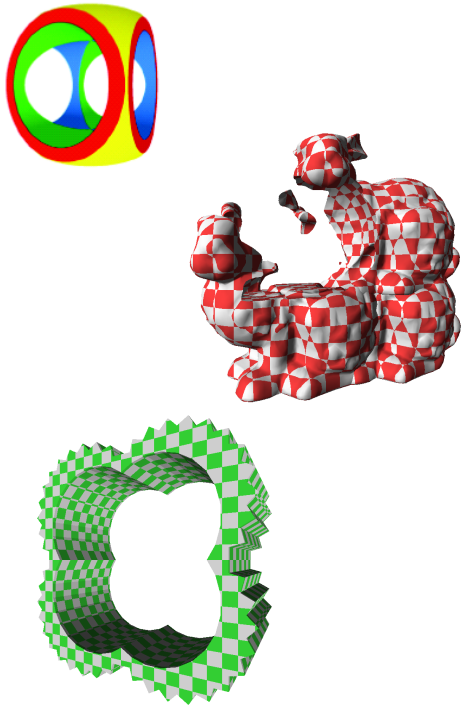
nVidia GeForce 6800

Bunny has 574K triangles, Bunny2 has 2.3M

Average over random directions

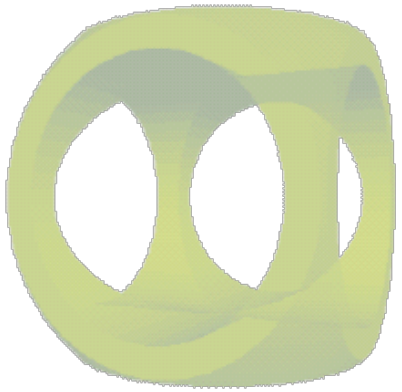
Actual layers processed when we prune active pixels

	prims	layers	k_f	**T** (sec)
Widget	4	4.00	3.85	**0.018**
Bunny	4	9.13	6.81	**0.325**
Bunny2	4	9.09	6.77	**1.200**
Complex	20	14.1	9.17	**0.093**
Gear	48	36.4	25.6	**0.460**

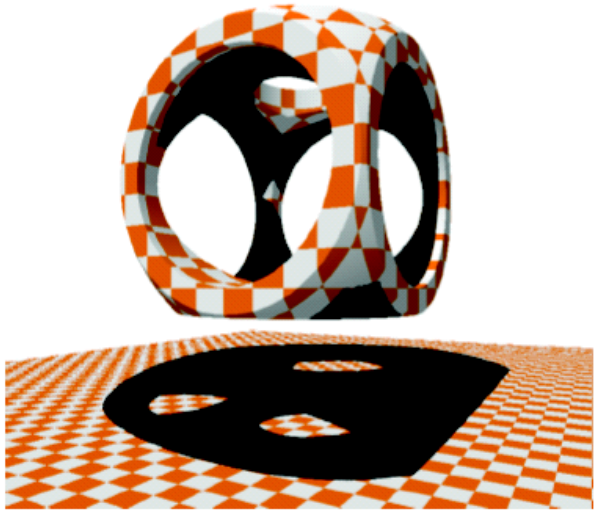


Extensions

- Transparency (not always correct)
  - Internal faces increase opacity



- Shadows (shadow buffer)

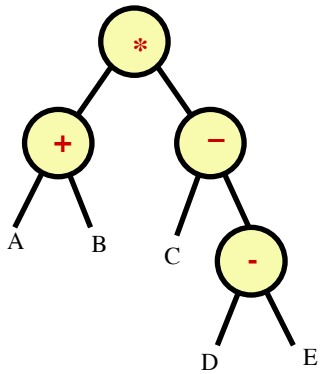


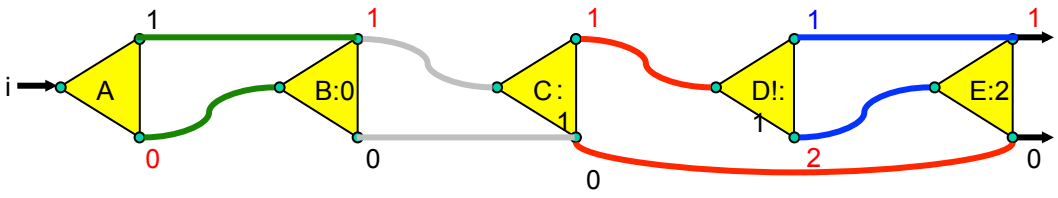
Ongoing & future work

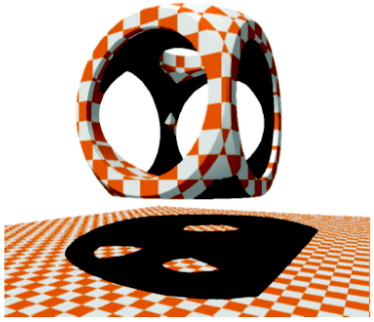
- Increase complexity for which Blister is guaranteed to work
  - Increased from 3909 to **2^{63} primitives** when using 6 stencil bits
- Improve performance
  - Preliminary Results: already **20% faster**
- Select "correct" color for fragments on 2 primitives
  - Use **Active Zones** [Rossignac&Voelcker 89]
- Correct surface transparency
  - Custom classification
- Voxelization / volumetric rendering
  - Interpolate / integrate between layers
- Parallelization (SLI/Crossfire)
  - AFR (Alternate Frame Rendering between two graphics cards)
  - Split Screen/Tiles, but shared texture

Summary of Blister

- Can render all CSG trees with up to 2^{63} primitives
  - Simple, efficient CSG-to-Blist conversion
- Time complexity: O(DepthComp*Primitives)
- Efficient implementation on GPU
  - Typical performance: 2 fps for CSG of 50 primitives
- Can be used for shadows (and "transparency")
- Many improvements/extensions underway







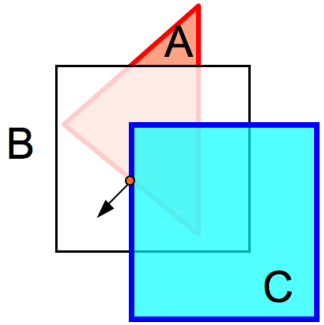
Constructive Solid Trimming (CST)

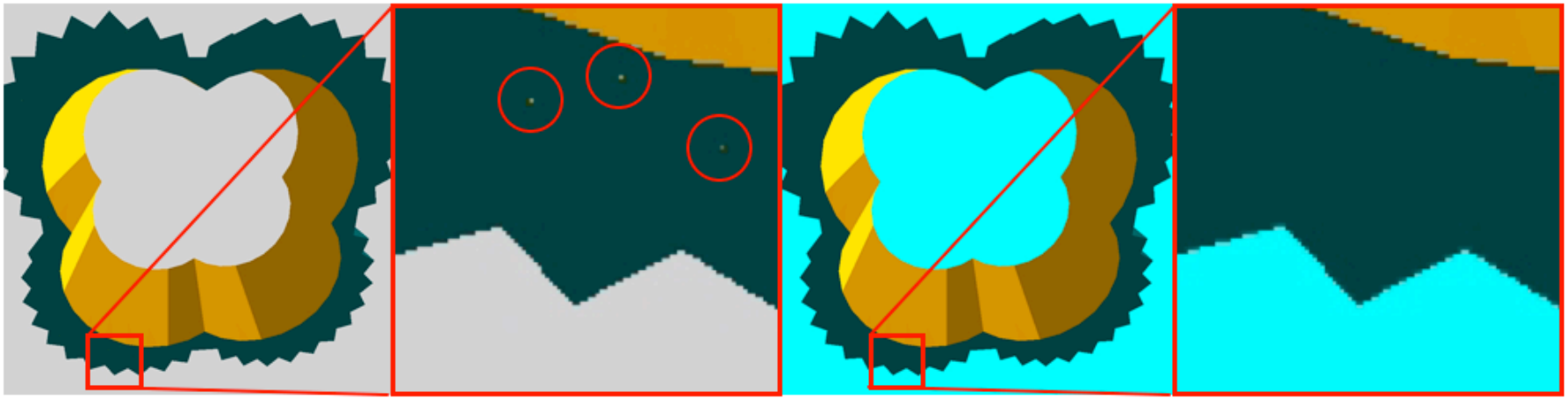
- Extension of CSG
- Use CSG expression for trimming surfaces
- GPU support for realtime rendering
- Use Active Zone Z of A as the CST for primitive A

- Advantages:
  - Faster
  - No ghost pixels
  - Internal structure
  - Correct transparency
  - Contacts / interferences
  - Cut-outs / painting

No ghost pixels

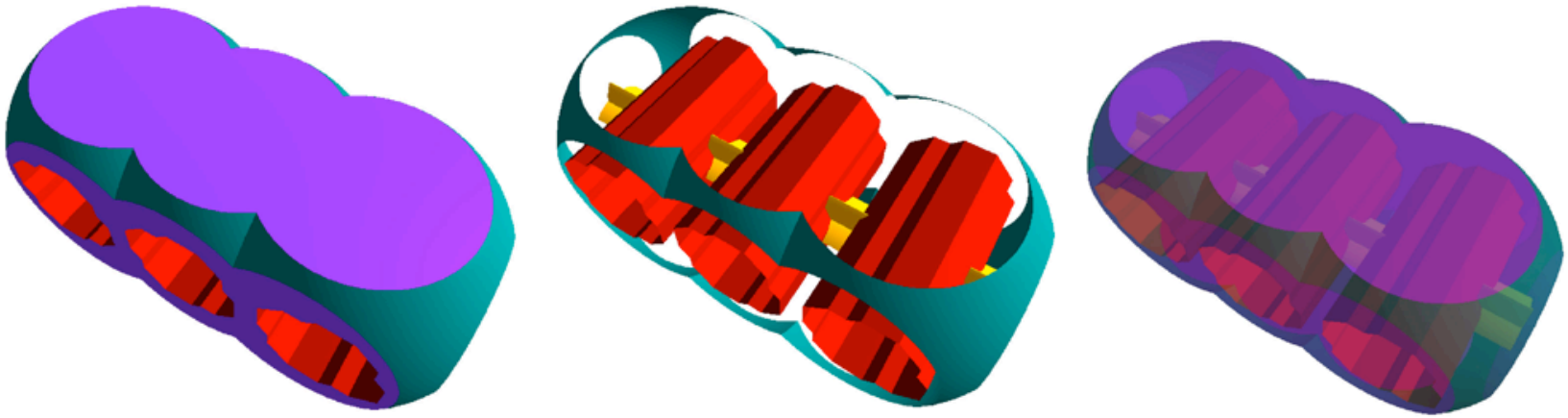
- Classifying surfels against active zones eliminates ghost pixels





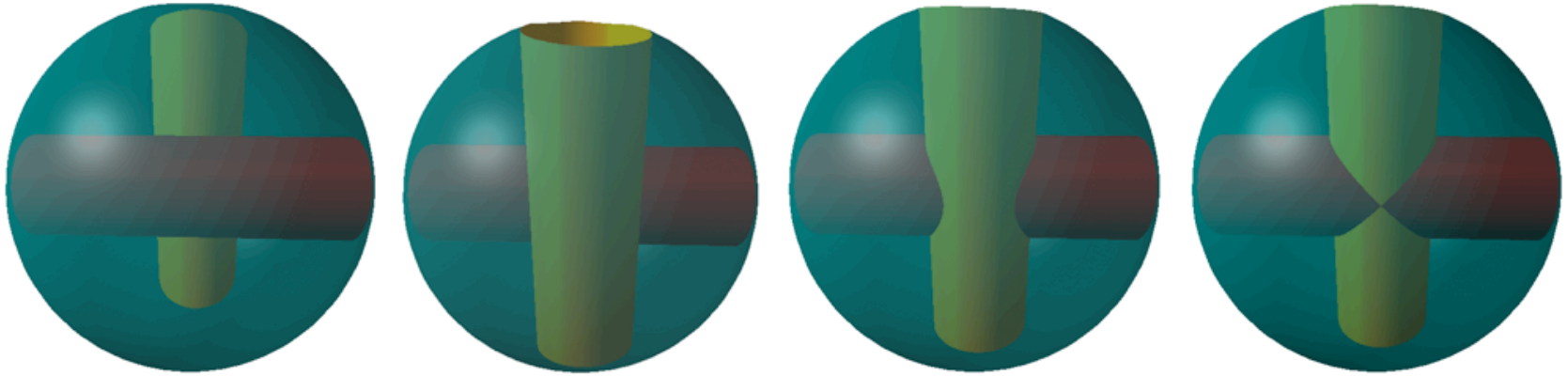
Reveal internal structure

- Make selected faces completely transparent reveal internal structure



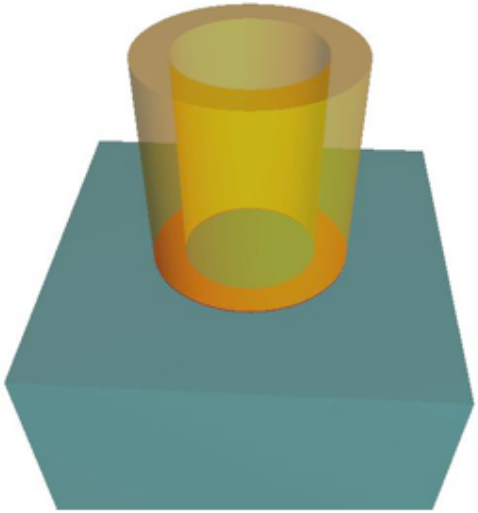
Correct transparency

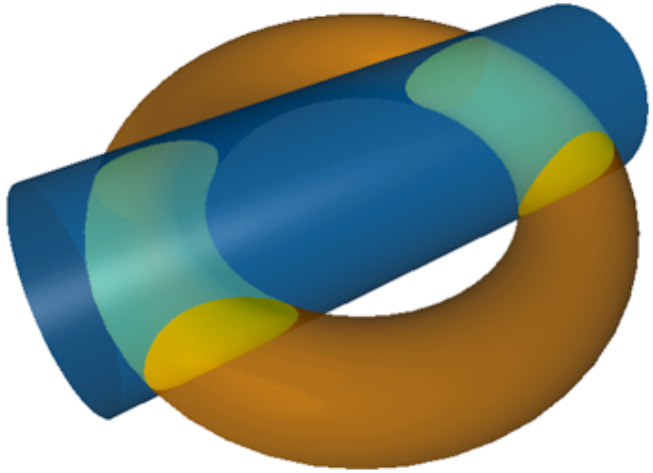
- Produce correct images of semitransparent CSG models



Contacts / interferences

- Highlight contacts and interferences for assembly design





Cut-outs / painting

Assigned Reading

Two papers:

- **Active Zones in CSG for Accelerating Boundary Evaluation, Redundancy Elimination, Interference Detection and Shading Algorithms**
  - Jarek Rossignac and Herbert Voelcker
  - ACM Transactions on Graphics, Vol. 8, pp. 51-87, 1989.
- **Blister: GPU-based rendering of Boolean combinations of free-form triangulated shapes**
  - John Hable and Jarek Rossignac
  - ACM Transactions on Graphics, *SIGGRAPH 2005*.