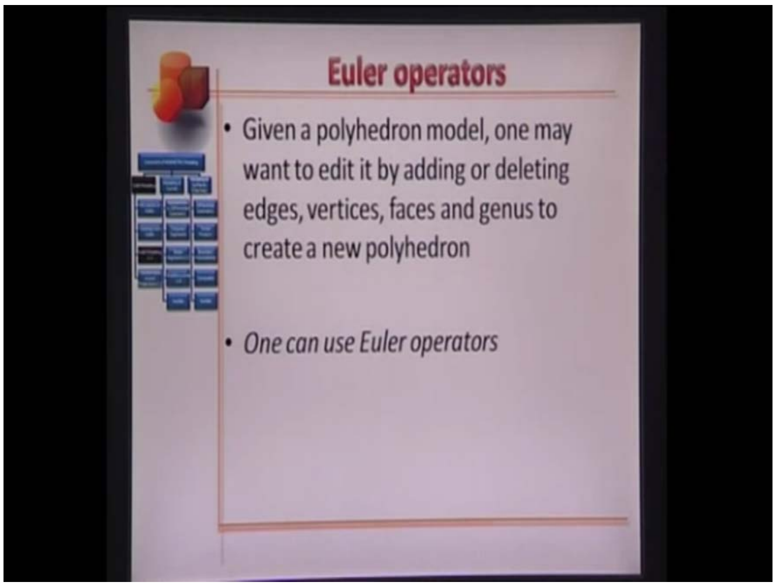
# Computer Aided Engineering Design

**Prof. Anupam Saxena**
**Department of Mechanical Engineering**
**Indian Institute of Technology, Kanpur**

## Lecture - 7

Welcome to lecture 7 of CAED NP-TEL video series. Here we are going to be discussing certain techniques to design solids. This is the second of the lectures in solid modeling. We can use Euler operators to design solids.

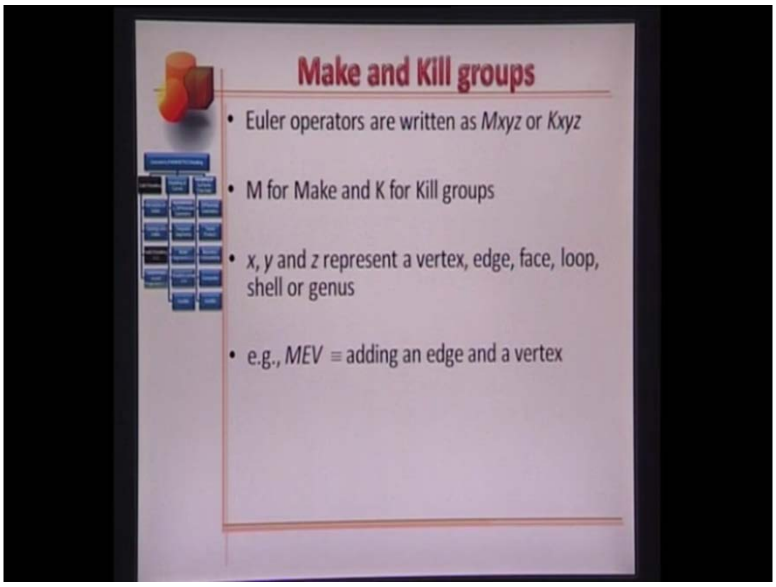
(Refer Slide Time: 00:43)



### Euler operators

- Given a polyhedron model, one may want to edit it by adding or deleting edges, vertices, faces and genus to create a new polyhedron
- *One can use Euler operators*

(Refer Slide Time: 01:02)

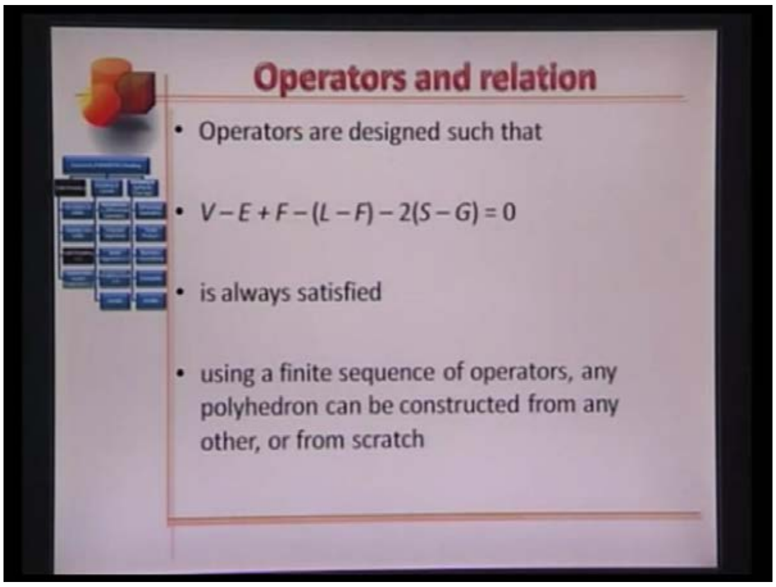


### Make and Kill groups

- Euler operators are written as *Mxyz* or *Kxyz*
- M for Make and K for Kill groups
- *x*, *y* and *z* represent a vertex, edge, face, loop, shell or genus
- e.g., *MEV* ≡ adding an edge and a vertex

Given a polyhedron model, one may want to edit it by adding or deleting edges, vertices, faces, and genus to create a new polyhedron. One can use Euler operators for this purpose. The two groups of Euler operators namely the Make groups and the Kill group. Euler operators are written as Mxyz or Kxyz; M represents the make group, and k represents the kill group, x y and z can represent any other features, vertex, edge, face, loop, shell or genus. For example, M E V implies make an edge and a vertex or in other words we add an edge and a vertex to a polyhedron model. K E V represents killing or deleting an edge and a vertex.

(Refer Slide Time: 02:04)

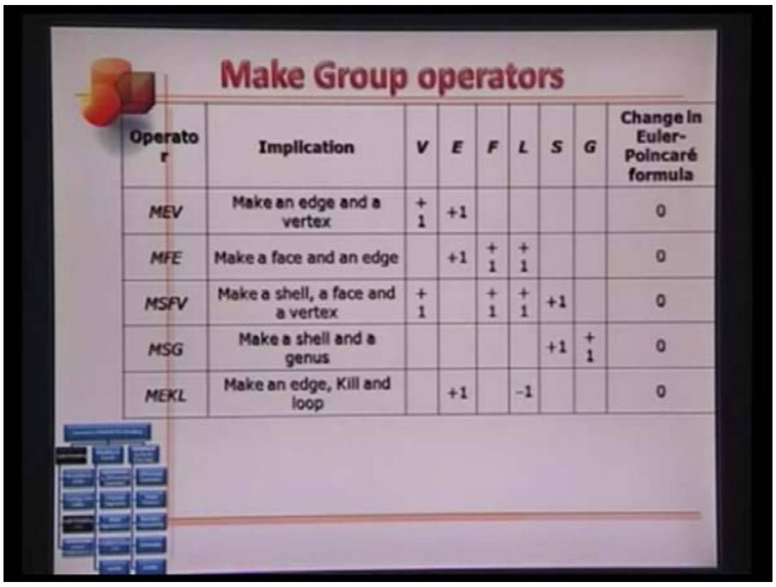


## Operators and relation

- Operators are designed such that
- $V - E + F - (L - F) - 2(S - G) = 0$
- is always satisfied
- using a finite sequence of operators, any polyhedron can be constructed from any other, or from scratch

Operators are designed to satisfy the Euler-Poincare formula, which is V minus E plus F minus within parenthesis L minus F minus 2 times S minus G equal 0. We have seen this relation in the previous lecture. Retransfer vertices E for edges, S faces, L for loops, S for the number of shells and G for the number of handles in case; we are dealing with solids, which are homeomorpically the connected some of toroide. Euler operators ensure that the Euler-Poincare formula is always satisfied. That is using a finite sequence of operators; any polyhedron can be constructed from any other polyhedron or from scratch. You might want to think about using Euler operators to construct a cube, from a tetrahedron or visa a versa; that is to construct a tetrahedron from a cube. You might as well want to think about constructing a tetrahedron from scratch.

(Refer Slide Time: 03:37)



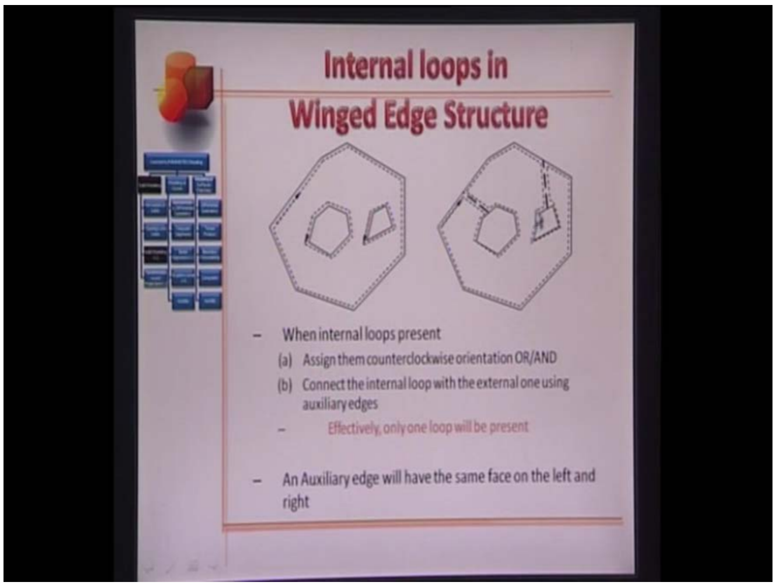
## Make Group operators

| Operator | Implication | V | E | F | L | S | G | Change in Euler-Poincaré formula |
|---|---|---|---|---|---|---|---|---|
| MEV | Make an edge and a vertex | +1 | +1 | | | | | 0 |
| MFE | Make a face and an edge | | +1 | +1 | +1 | | | 0 |
| MSFV | Make a shell, a face and a vertex | +1 | | +1 | +1 | +1 | | 0 |
| MSG | Make a shell and a genus | | | | | +1 | +1 | 0 |
| MEKL | Make an edge, Kill and loop | | +1 | | -1 | | | 0 |

Here are some details about the make group operators. M E V stands for make an edge and a vertex, this operator introduces a vertex in an edge. And all the other features are not introduced. Correspondingly there is no change in the Euler-Poincare formula. The second operator make a face in an edge M F E this operator introduces a face in an edge, the number of edges get incremented by 1, the number faces also get incremented by 1 and since loops are associated with faces, the number of loops get incremented; there is no increase in the shell value nor there is an increase in the genus value.

Once again there is no change in the Euler-Poincare formula. M S F V make a shell, a face and a vertex; the number of vertices increases by 1, there is no increase in the number of edges with this formula; the number of faces increases by 1, the number of loops get increased by 1 and the number of shells would increase by 1 with no change and genus value; again no change in the formula. M S G make a shell and a genus, no change in the number of vertices, in the number of edges, in faces and a loops ,but the number of shells increase by 1, and so does the genus value, again there is no change in the formula. Finally make an edge and kill a loop, the operator M E K L there is an increase in the number of edges, no increase in the number of faces, the number of loops get decremented by 1; no increases in the number of shells nor the genus; once again there is no change in the Euler-Poincare formula. This operator has a specific purpose. Recall our discussion from the previous lecture on internal loops.

(Refer Slide Time: 06:46)



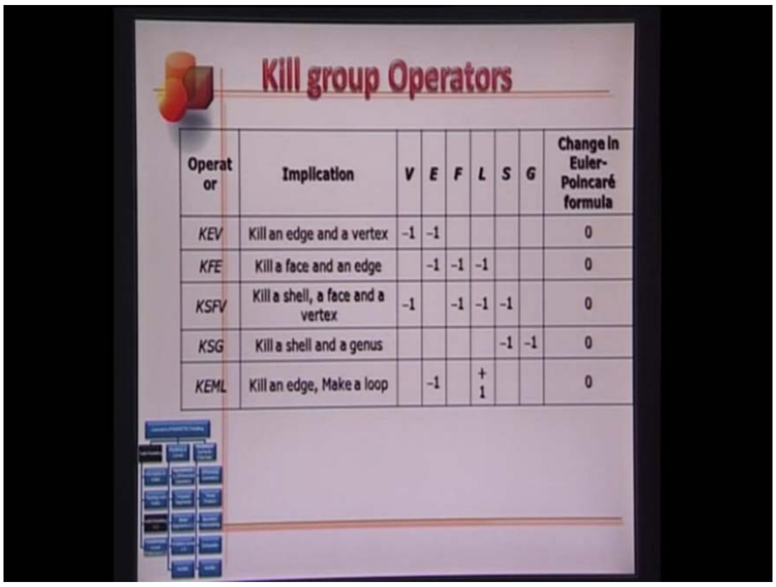
## Internal loops in Winged Edge Structure

– When internal loops present
  (a) Assign them counterclockwise orientation OR/AND
  (b) Connect the internal loop with the external one using auxiliary edges
  – Effectively, only one loop will be present

– An Auxiliary edge will have the same face on the left and right

And recall how we introduced auxiliary edges, to merge the internal loops with the external one. This was the first auxiliary edge and this was the second auxiliary edge. So remember what we are trying to do here, We are trying to construct in edge, or make an edge and in the process kill this internal loop; once again, we are trying to make an edge, or construct an edge and in the process we trying to kill this internal loop.

(Refer Slide Time: 07:38)



## Kill group Operators

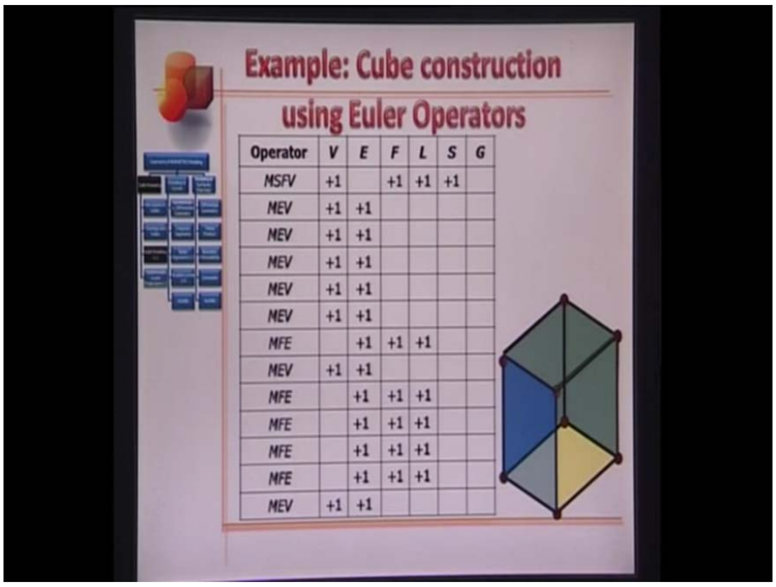
| Operator | Implication | V | E | F | L | S | G | Change in Euler-Poincaré formula |
|---|---|---|---|---|---|---|---|---|
| *KEV* | Kill an edge and a vertex | -1 | -1 | | | | | 0 |
| *KFE* | Kill a face and an edge | | -1 | -1 | -1 | | | 0 |
| *KSFV* | Kill a shell, a face and a vertex | -1 | | -1 | -1 | -1 | | 0 |
| *KSG* | Kill a shell and a genus | | | | | -1 | -1 | 0 |
| *KEML* | Kill an edge, Make a loop | | -1 | | +1 | | | 0 |

The operator M E K L is specifically designed for auxiliary edges. Now kill group of Euler operators. The first is K E V that is kill an edge and a vertex; the number of vertices would decrease by 1 and the number of edges would decrease by 1. Correspondingly there is no change in the formula. The second kill a face and an edge K F E no change in the number of vertices, the number of edges decrease by 1, the number of faces decrease by 1 and so does the number of loops; no change in the number of shells nor the number of genus. And therefore there is no change in the Euler-Poincare formula.

The third one K S F V kill a shell, a face and a vertex. The number of vertices decrease by 1, the number of faces decrease by 1; and so does the number of loops, the number of shells decrease by 1; no change in the Euler-Poincare formula. The fourth kill a shell and a genus; the number of shells decrease by 1, and so does the number of genus or the number of handles; no change in the Euler-Poincare formula. Finally, we have K E M L which stands for kill an edge and make a loop. The number of edges go down by 1, the number of loops increase by 1, and there is no change in the Euler-Poincare formula.

If you think about it, the make group and the kill group of Euler operators are designed such that the Euler-Poincare formula is always satisfied. What would that mean? Theoretically this would mean, that if one would start with a polyhedron and start constructing or start using these operators; namely, the make group operators and the kill group operators. One would assume that all intermediate results would be valid solids.

(Refer Slide Time: 10:51)



## Example: Cube construction using Euler Operators

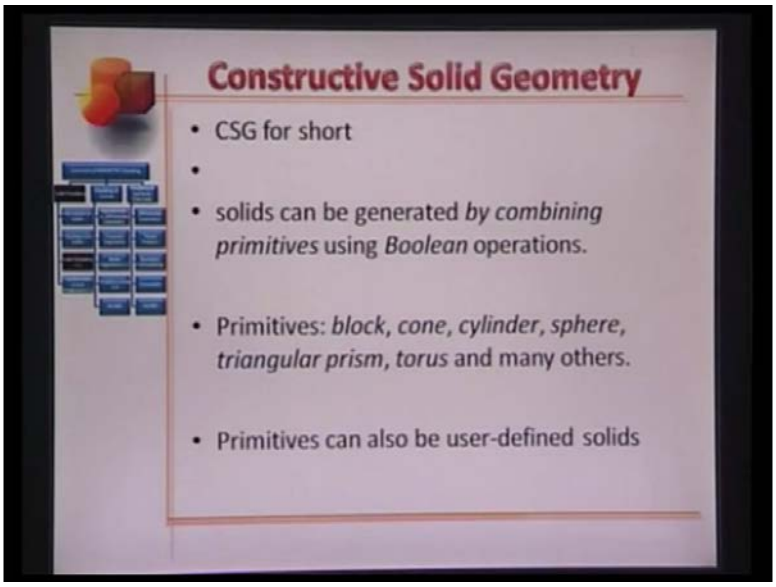
| Operator | V | E | F | L | S | G |
|---|---|---|---|---|---|---|
| MSFV | +1 | | +1 | +1 | +1 | |
| MEV | +1 | +1 | | | | |
| MEV | +1 | +1 | | | | |
| MEV | +1 | +1 | | | | |
| MEV | +1 | +1 | | | | |
| MEV | +1 | +1 | | | | |
| MFE | | +1 | +1 | +1 | | |
| MEV | +1 | +1 | | | | |
| MFE | | +1 | +1 | +1 | | |
| MFE | | +1 | +1 | +1 | | |
| MFE | | +1 | +1 | +1 | | |
| MFE | | +1 | +1 | +1 | | |
| MEV | +1 | +1 | | | | |

In general this may not be true, but still one would have that impression, that all results are valid solids. Eventually, when the final polyhedron is constructed it would be ensured that it is a valid solid. Let us try now to construct a cube using Euler operators. The first step make a shell, face and a vertex; these are the respective increments in the number of vertices, edges, faces, loops, shells and genus. What we have a done is, we have hypothetically constructed a shell we still have to find it with the inclosing faces. But for now, let us assume that this is a hypothetical shell and we have introduced a face at the bottom and a vertex writes here. The next step make an edge and a vertex; this would increase the number of edges and vertices.

Correspondingly the construction is shown here; this edge and this vertex get newly introduced. Try to appreciate the intermediate results, at this time none of them would be valid solids, but still the Euler-Poincare formula will be satisfied. The third step, once again make an edge and a vertex, the number of vertices and edges increase by 1 each, this is the new edge and a new vertex introduced. Next step make an edge and a vertex again; the next step make an edge and make a vertex, once again make an edge and make a vertex. Once again now make a face and an edge, the number of edges increase by 1, the number of faces and the number of loops they both increase by 1.

We have introduced the top face and the right most edge on the top face, make an edge and a vertex, introducing a vertex and an edge at the right most corner of the cube; make a face and an edge, a new face and a new edge gets introduced here. Make a face and an edge again, this is back face right here that gets introduced. Make a face and an edge again, it is this face now that is getting introduced make face and an edge, it is a face closes to you getting introduced. And finally, make an edge a vertex. If you look at this polyhedral model closely, what remains is construct this edge and this vertex. We have just about finished the construction of a cube using Euler operators.

(Refer Slide Time: 14:34)

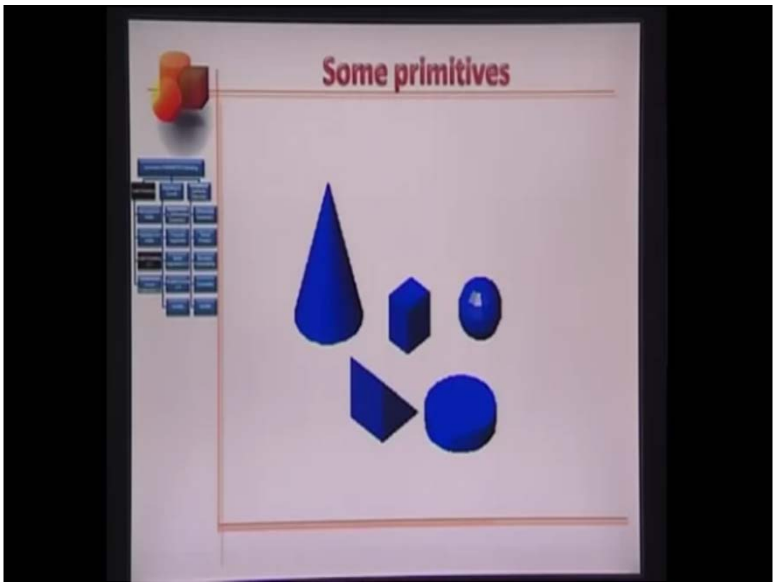


## Constructive Solid Geometry

- CSG for short
-
- solids can be generated *by combining primitives* using *Boolean* operations.
- Primitives: *block, cone, cylinder, sphere, triangular prism, torus* and many others.
- Primitives can also be user-defined solids

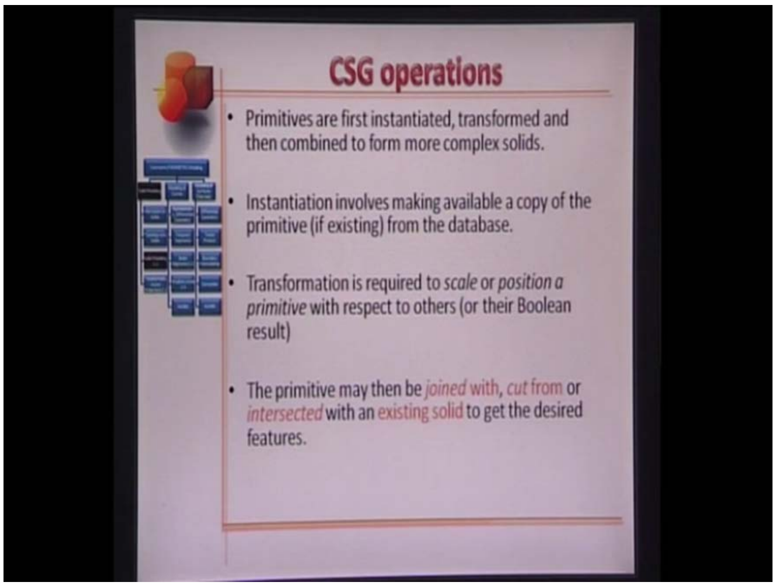
The next technique is called the constructive solid geometry CSG for short. It is based on this (( )), that solids can be generated by combining primitives using Boolean or set operations. We are seen those operations before, I refer to addition, subtraction, union and intersection like set operation here. Primitives can be block, cone, cylinder, sphere, a triangular prism of which a torus and many others. Primitives can also be user-defined solids that one can design and store in library of a solid modular.

(Refer Slide Time: 15:32)



## Some primitives

(Refer Slide Time: 15:40)

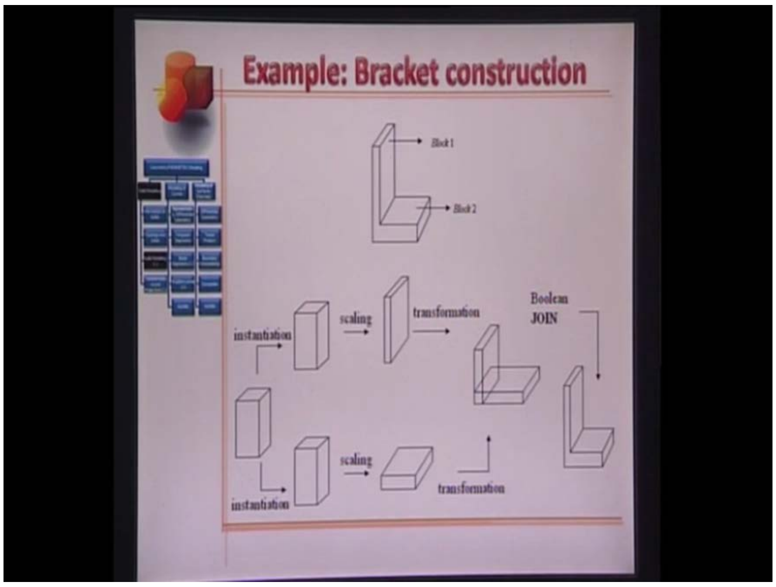


## CSG operations

- Primitives are first instantiated, transformed and then combined to form more complex solids.
- Instantiation involves making available a copy of the primitive (if existing) from the database.
- Transformation is required to *scale* or *position a primitive* with respect to others (or their Boolean result)
- The primitive may then be *joined with*, *cut from* or *intersected* with an existing solid to get the desired features.

These are some examples of basic or analytical primitives. We see a cone, a block, a sphere, a edge, a cylinder and there could be others. Let us talk about constructive solid geometry operations. Step 1: primitives are first instantiated which means copied and then transformed and then combined to form more complex solids. Instantiation involves making available a copy of the primitive from the database; I will explain this step you later. Transformation is required to scale or position a primitive with respect to a few others which are their available in front of your screen.

(Refer Slide Time: 16:55)

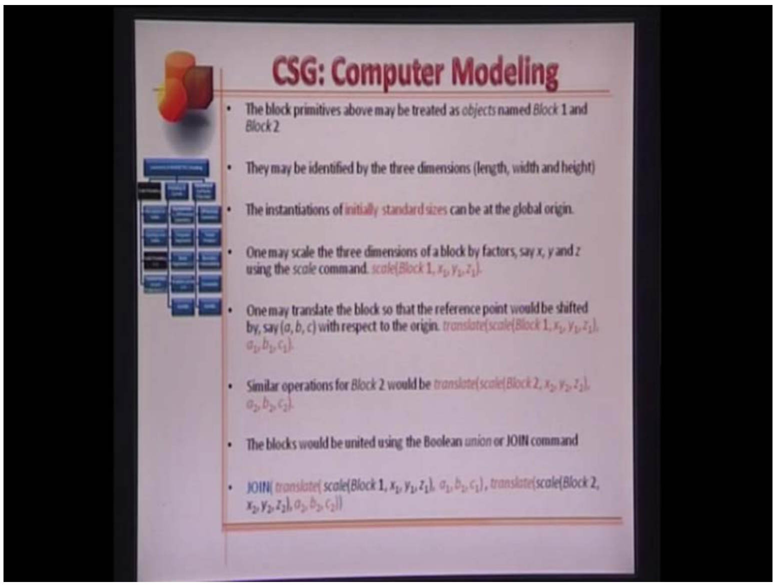


## Example: Bracket construction

The primitive may then be joined with, cut from or intersected with an existing solid to get the desired features. These are the set operations. Let us see the example of how to construct a bracket using constructive solid geometry. If you think about it and l bracket can be thought of a union of two blocks; appropriately positioned with respect to each other. Let us take the block as a primitive; let us assume that this primitive is available with database. I first copied on to the workspace which is the process instantiation; I perform scaling to get block 1, and then I perform transformation to appropriately face it in the workspace.

I make another copy of the block available to me in the workspace. In other words I insatiate this block again, I scale it appropriately to the dimensions of block 2 which is seen in the top figure. I transform it to place it properly with respect to block 1 and then I perform Boolean join; which is a union of two blocks to get the l bracket. How would this operation be seen by a solid modular or how would this operation be simulated by a solid model, let us try to understand that.

(Refer Slide Time: 18:46)



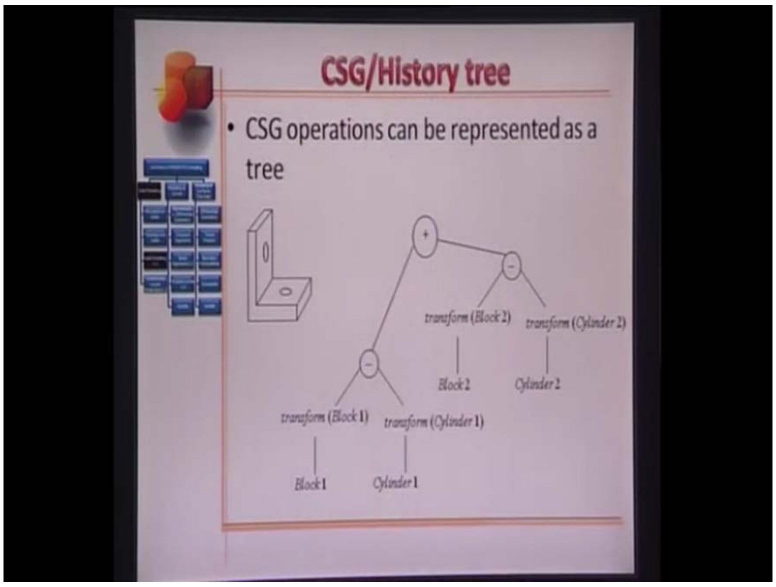
## CSG: Computer Modeling

- The block primitives above may be treated as *objects* named *Block* 1 and *Block* 2
- They may be identified by the three dimensions (length, width and height)
- The instantiations of initially standard sizes can be at the global origin.
- One may scale the three dimensions of a block by factors, say *x*, *y* and *z* using the *scale* command. *scale*(*Block* 1, $x_1, y_1, z_1$).
- One may translate the block so that the reference point would be shifted by, say ($a, b, c$) with respect to the origin. *translate*(*scale*(*Block* 1, $x_1, y_1, z_1$), $a_1, b_1, c_1$).
- Similar operations for *Block* 2 would be *translate*(*scale*(*Block* 2, $x_2, y_2, z_2$), $a_2, b_2, c_2$).
- The blocks would be united using the Boolean *union* or JOIN command
- JOIN( *translate*( scale(Block 1, $x_1, y_1, z_1$), $a_1, b_1, c_1$), *translate*(scale(Block 2, $x_2, y_2, z_2$), $a_2, b_2, c_2$))

The block primitives above may be treated as objects named block 1 and block 2. They may be identified respectively by three dimensions; the length, the width, and the height of the blocks. The copies of the initially standard sizes can be at the global origin. One may scale the three dimensions of a block by factors say x, y and z using say the scale command. Look at the representations in the red; it says, scale block 1 by factors x 1, y 1 and z 1, this is the scaling operation.

One may translate the block, so that the reference point would be shifted by say, coordinates a, b and c with respect to the global origin. This command is represented by translate scale block 1 with factors x, y, z and translate by coordinates a 1, b 1 and c 1. Similar operations for block 2 will be to translate within parenthesis scale block 2 by factors x 2, y 2 and z 2; and translate by coordinates a 2, b 2 and c 2. The two blocks would be united using the Boolean union or join command. This is how the entire operation would look like in text. We can represent the same operation graphically using the history tree.

(Refer Slide Time: 21:10)



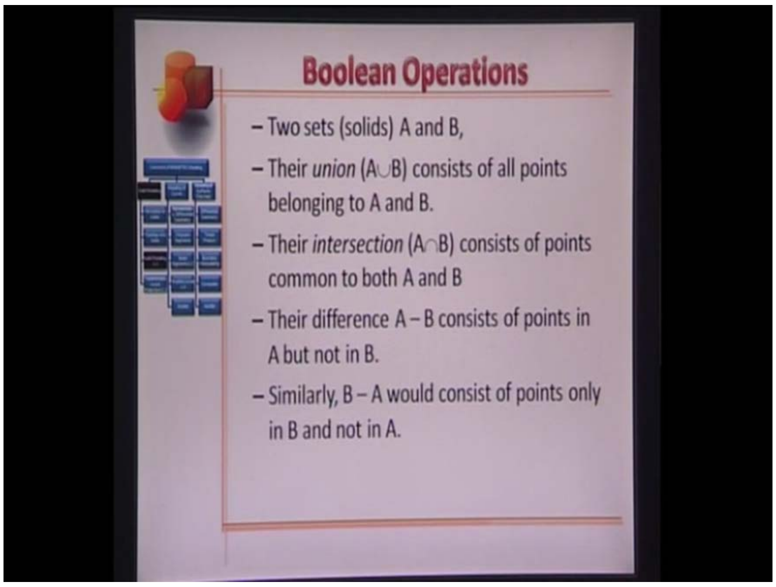
## CSG/History tree

- CSG operations can be represented as a tree

Like a set C S G operations can represented as a tree graphically. Let us take a look at a slightly different example. This is an l bracket with two true holes. We will start with block 1, we will transform it we will take cylinder one, we will transform it and we will perform a subtraction operation. So if we imagine that this is block 1, and this is cylinder 1. What we have done here? Is we have first made a copy available of block 1, in the workspace transformed it appropriately. And then we have introduced a copy of cylinder 1, in the workspace transformed it appropriately; and then we have subtracted the cylindrical feature form this block.

We can do something very similar for the second block and second cylinder; that is we take block 2 transform it, we take cylinder 2 transform it and cut the cylinder from the block. Doing so would give me a block with a cylindrical hole here. And then I can perform Boolean join operation to join this block with this block.

(Refer Slide Time: 22:55)



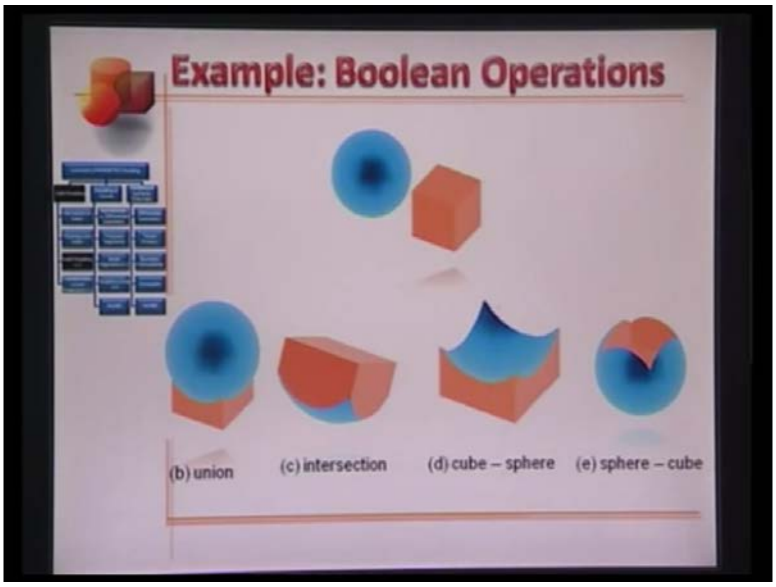
## Boolean Operations

- Two sets (solids) A and B,
- Their *union* (A∪B) consists of all points belonging to A and B.
- Their *intersection* (A∩B) consists of points common to both A and B
- Their difference A – B consists of points in A but not in B.
- Similarly, B – A would consist of points only in B and not in A.

We have seen two kinds of Boolean operations already; namely, the subtraction and the Boolean join. Let us talk about some more operations in detail. Two sets or solids A and B; if we have two solids A and B; their union which is represented by A union B, consists of all points belonging to the two solids. Their intersection represented by A intersection B would consist of points which are common to both A and B. Their difference A minus B would consists of points in A, but not in B.

Similarly, the difference B minus A would consist of points only in B and not in A. Try to appreciate the difference between the intersection operation and the subtraction operation. I emphasis again that intersection operation would give us points, which are common to both solids; participating in Boolean operations. Whereas in the subtraction operation for example, A minus B it will give us solid with points only in the primitive A, but not in the primitive B.

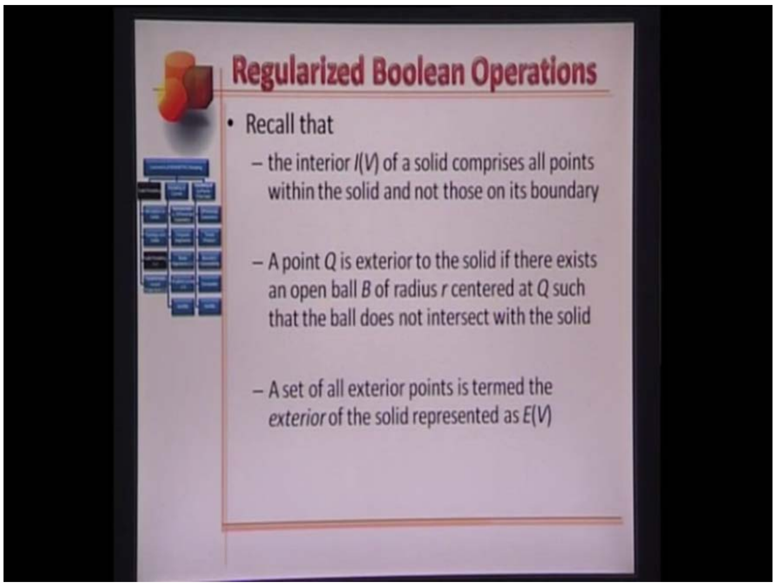
(Refer Slide Time: 24:42)



## Example: Boolean Operations

Let us take a few examples on Boolean operations, for a simple ones. Let us say a sphere and a cube are interacting with each other; why different operations, why different set operations are Boolean operations. Imagine there are transformed this sphere to lie over the top face of the cube. In such a way, that the center of the sphere rests on the top face of the cube. The union of the two primitives will look like this.

(Refer Slide Time: 26:20)



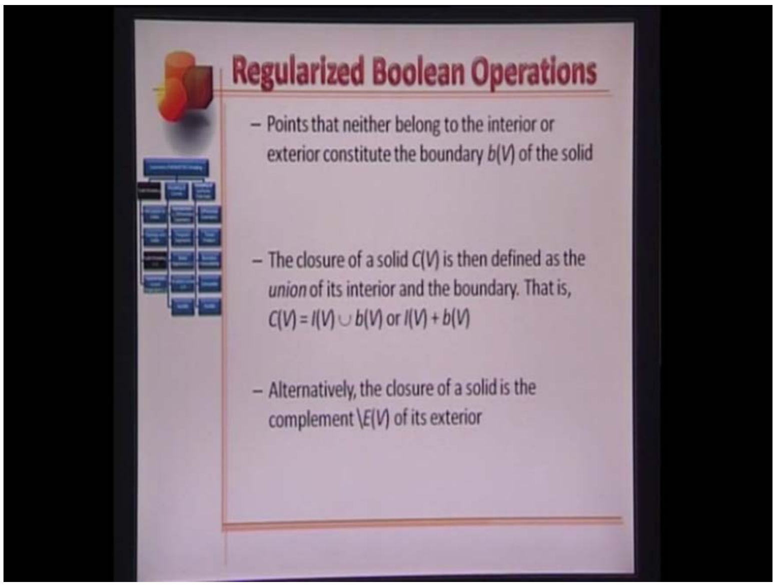
## Regularized Boolean Operations

- Recall that
  - the interior *I*(*V*) of a solid comprises all points within the solid and not those on its boundary
  - A point *Q* is exterior to the solid if there exists an open ball *B* of radius *r* centered at *Q* such that the ball does not intersect with the solid
  - A set of all exterior points is termed the *exterior* of the solid represented as *E*(*V*)

The intersection would look something like this; I have rotated the view here, to show the result better. The result cube minus sphere will look something like this; and the other way round that is sphere minus cube will look some like this. So this is very simple example that stimulates interaction between two very basic primitives this sphere and the cube; can result in different solids, if different set operations of a formed with them. Let us now, talk about regularized Boolean operations. I will tell you a little later why the Boolean operations cannot be applied in the row form the way we discussed in the previous slide.

Recall that the interior I V of a solid includes all points in the solid and not those on its boundary. A point Q is exterior to the solid if there exists an open ball of radius r centered at Q such that; the ball does not intersect with the solid. A set of all exterior points is termed as the exterior of the solid represented as E of V; where V would be a set of all points belonging to the solid. This is a definition that we have seen in one of the previous lectures.

(Refer Slide Time: 27:13)

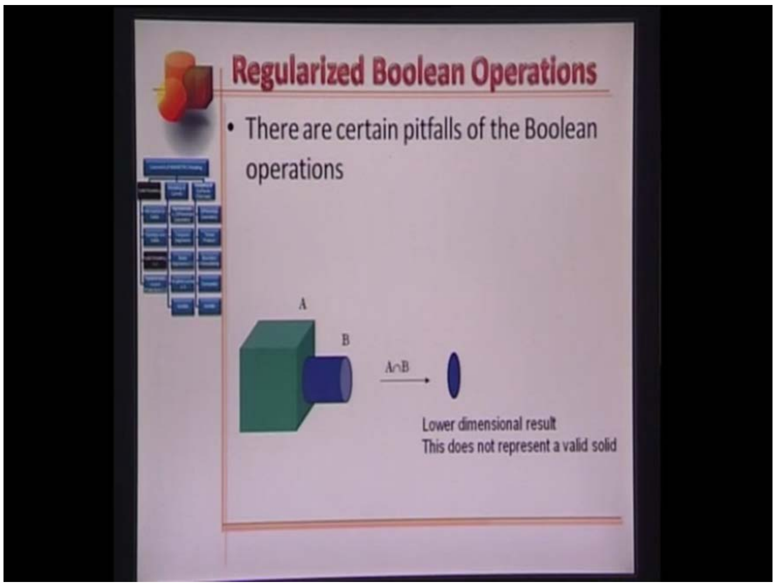


## Regularized Boolean Operations

- Points that neither belong to the interior or exterior constitute the boundary $b(V)$ of the solid
- The closure of a solid $C(V)$ is then defined as the *union* of its interior and the boundary. That is, $C(V) = I(V) \cup b(V)$ or $I(V) + b(V)$
- Alternatively, the closure of a solid is the complement $\backslash E(V)$ of its exterior

Points that neither belongs to the interior or exterior constitute the boundary b V of the solid. The closure of a solid C V is defined as the union of its interior and the boundary. That is, C V equals i V union b V or I V plus b V. Note that in general union an addition operations might give us different results, but in our case the sets i V and b V are disjoint. They would not have any element common between them, for which reason the union of the interior of the solid and the boundary of the solid or the addition of the interior and the boundary of the solid would give the same result. Alternatively the closure of a solid is the complement slash E V of its exterior.

(Refer Slide Time: 28:40)

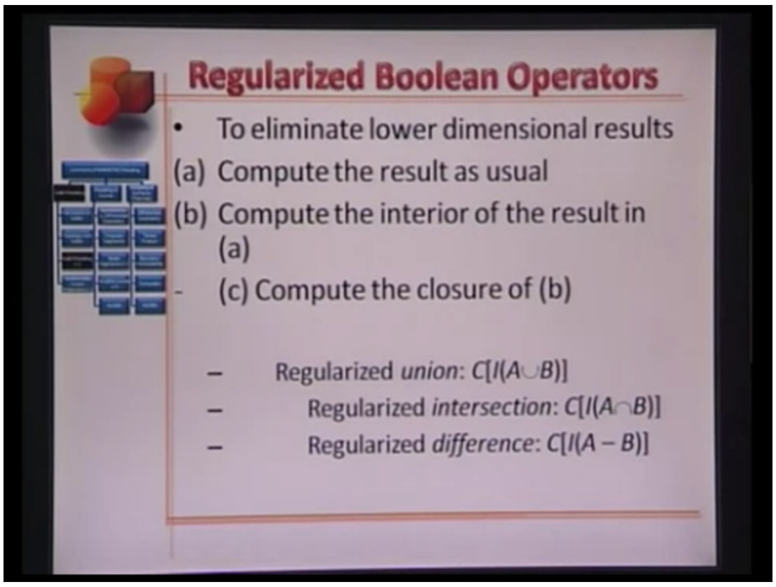


## Regularized Boolean Operations

- There are certain pitfalls of the Boolean operations

A
B
A∩B
Lower dimensional result
This does not represent a valid solid

(Refer Slide Time: 29:57)



## Regularized Boolean Operators

- To eliminate lower dimensional results

(a) Compute the result as usual

(b) Compute the interior of the result in (a)

(c) Compute the closure of (b)

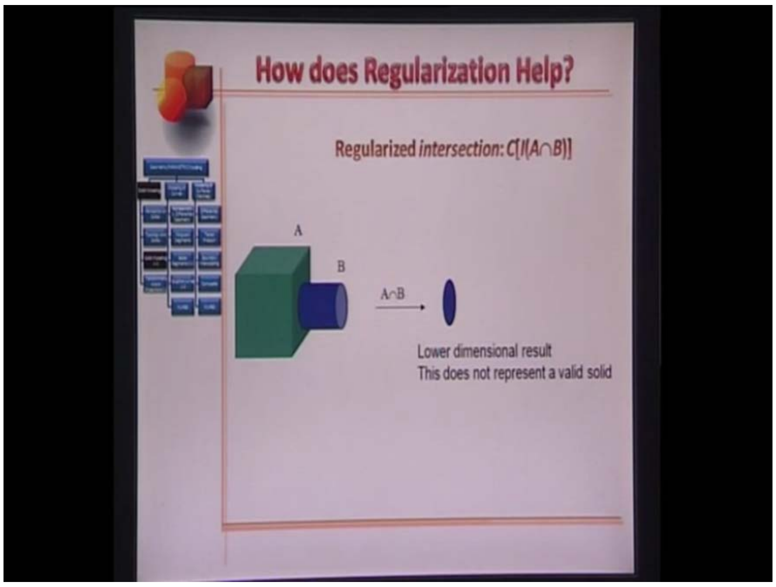
- Regularized *union*: $C[I(A \cup B)]$
- Regularized *intersection*: $C[I(A \cap B)]$
- Regularized *difference*: $C[I(A - B)]$

Here is where I will explain why row Boolean operations may not be used. There certain draw backs of the Boolean operations. Let us take this example, we have a cube and we have a cylinder; we have positioned this cylinder, in such a way that one of its faces rests on one of the faces of the cube. Say the cube is solid A and the cylinder is solid B. If we perform the intersection operation between the two will be left with a desk and this is a lower dimensional result. It is two-dimensional results and it does not represent a valid solid, this desk has no thickness. So a row intersection operation for example, in this case can lead to lower dimensional results. That is why we need to regularize Boolean operation. This is how we do it.

To eliminate lower dimensional solids, which are not solids at all. We compute the result as usual, that is we compute the result as if we are performing a row Boolean operation. And then we compute the interior of that result. We compute the closure of the result in step b for example; a regularized union will look some like this. We have two solids A and B, we perform the union like a row union, we compute the interior of this result and then we perform closers operation, which would be the union of the interior of A union B and the boundary of A union B.

Likewise a regularized intersection operation will look some like this; we perform row intersection between the two solids, we compute the interior of this result and then we perform the closer operation. Here we perform the union between the interior of A and B and the boundary of A intersection B. A regularized difference can be thought of in a similar manner, compute the difference as usual in the row form A minus B compute the interior of A minus B and then perform the closer operation.

(Refer Slide Time: 32:09)



## How does Regularization Help?

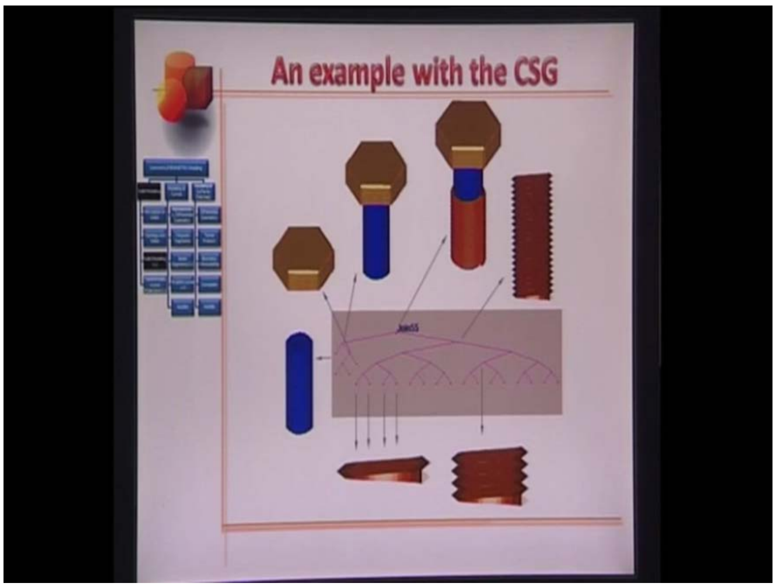
Regularized *intersection*: $C[I(A \cap B)]$

Lower dimensional result
This does not represent a valid solid

Using the cube and the cylinder example. Let us try to understand now, how would regularization help. Recall that we had computed the intersection between the cube A and this cylinder B, to get a low dimensional result which was a desk. Let us take into perspective regularized intersection.

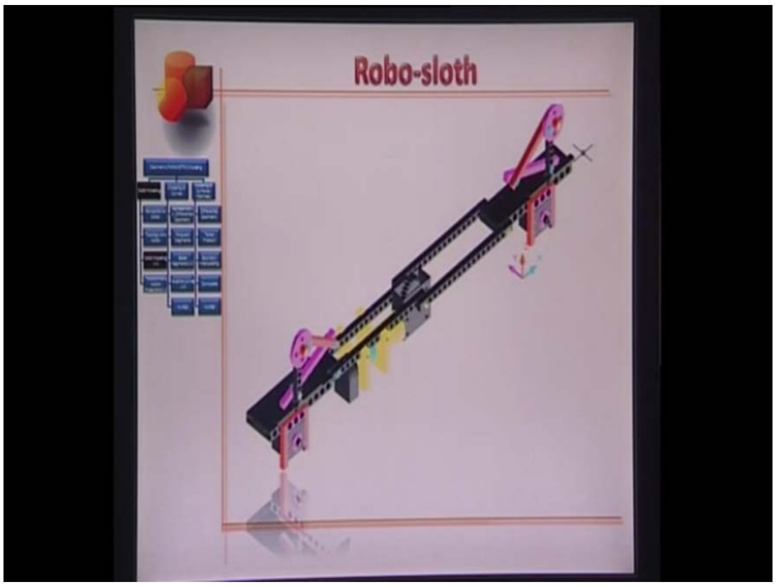
We compute the intersection of A and B in the row form to get a desk and if we now compute the interior of the desk, we will find that the result will be a null set and the closer of a null set would again be a null set. In the sense therefore, while row intersection gives me a low dimensional result; which is a desk a regularized intersection of the cube and a cylinder, post relative to each other in this manner will give me a null set which is what is exceptive. Let us now see an example with constructive solid geometry, with reference to the history tree which shown in the figure here.

(Refer Slide Time: 33:26)



We trying to construct a bolt; each of these Nodes here, represents the thread, revolved by 360 degrees and it has a pitch value p. These threads can be joined together these joins are represented by these nodes here to give us more threads. So, more of these threads can be joined together, to give the threaded structure which is now hollow from within. This Node here represents a cylindrical primitive, which is the sunk of the bolt. This Node here represents the hexagonal head of the bolt; they can be joined together to get this intermediate result. And these two results can be joined together, to give me a hexagonal bolt. In a similar manner one can think of constructing very complex solids, very complex features.

(Refer Slide Time: 35:11)

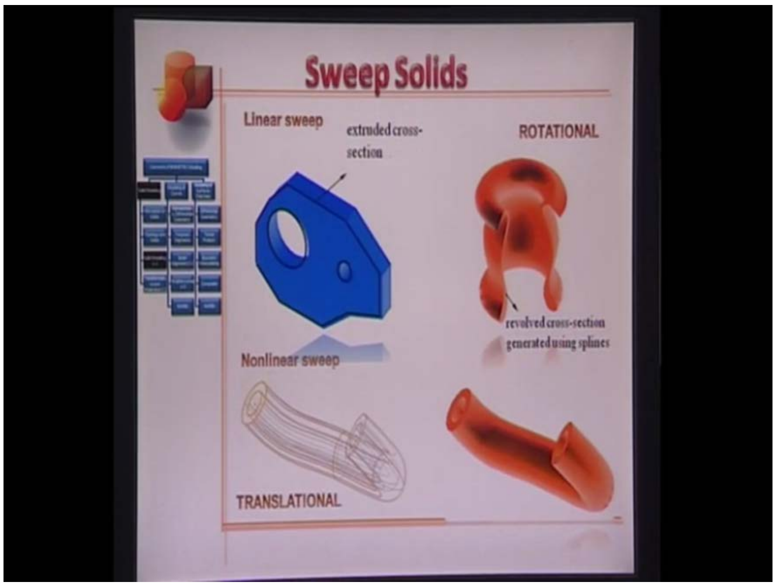


# Robo-sloth

For example, this robo-sloth, which was designed by a few of my B-Tech students in their final year project. These students analyzed this design in the soft form thoroughly before manufacturing. After verifying the model of a robo-sloth in soft form we prototyped and tested it. This is how the robot works, the robot is designed to scale a thin rope, it has two grippers at the end. One of the grippers would relies itself and move back and forth on the rope while the other grippers were ensured that one end of the robot is stationary; just like a natural sloth.

This is where the robot is scaling a vertical rope. The point I emphasize here is that it is always better to evaluate any design in soft form and ensure almost a 100 percent that the design is going to work before one can think of a prototyping here. And this is where solid modeling comes in very handy. The robot is not trying to get down. There are other techniques that different solid models tend to use to generate primitives. There essentially, categorized or classified as sweep operations.

(Refer Slide Time: 38:22)


## Sweep Solids

One can come up with a cross section something like this, with outer and may be a few in a loops and one can think of extruding this entire cross section in the third dimension to get primitive like this; this is called the linear sweep operation. Alternatively one can sweep a cross section in this case for example, two circles here along non-linear path in the third dimension; this is how the solid looks like; this is called the non-linear sweep operation. Both linear and non-linear sweep are categorized into translational sweep operations. They are translational sweep solids.

Likewise, one can think a rotational sweep as well. For example, one can come up with any cross section in particular this is a cross section obtained by a closed spline curve or a free from curve and one can revolve it about an axis in this case vertical to get this solid; this is the rotational sweep operation.