Sketch Input of Engineering Solid Models

2. Wireframe drawings

Pedro Company

Peter Varley

Introduction

Divide-and-conquer helps in
isolating those open problems which
prevent SBM tools from being used

We shall describe
the **main stages** of an SBM process

Then we shall detail several **algorithms**
for solving some representative problems of the
different stages:

1. Finding faces for polyhedral shapes
2. Inflating polyhedral shapes
3. Rounds and fillets

Main stages

The following main stages are often considered in Sketch-Based Modelling:

1. 2D sketching
2. 2D beautification or tidying up
3. Extraction of geometrical and perceptual information
4. Inflating a rough 3D model
5. 3D model refinement

They are not strictly sequential, the order may change, and some of them are sometimes unnecessary!

(More details in Annex 6)

Main stages

We shall describe algorithms representative of the current state of the art in these stages:

1. 2D sketching
2. 2D beautification or tidying up
3. Extraction of geometrical and perceptual information
4. Inflating a rough 3D model
5. 3D model refinement

Learn more on segmentation:

Xiong, Y. LaViola J. (2010) A ShortStraw-Based Algorithm for Corner Finding in Sketch-Based Interfaces, *Computers and Graphics, 34(5):513-527*

(See a demo in Demo 1)

We shall describe our algorithm for **finding faces**

We shall describe our algorithm for **inflating quasi-normalons**

We shall describe our algorithm for **optimisation-based inflation**

We shall describe our algorithm for **finding rounds and fillets**

Finding Faces in Wireframes

Perception is the stage where information required to produce a 3D model out of the 2D input is sought

Relevant information may be:

- explicit

  e.g. edges connected to the same vertices

  Not so difficult!

- implicit

  e.g. faces of a polyhedral shape

  Not so easy!

  Some heuristics are required to extract this information!

Finding Faces in Wireframes

Perception is the stage where information required to produce a 3D model out of the 2D input is sought

Relevant information may be:

- explicit

e.g. edges connected to the same vertices

Not so difficult!

- implicit

e.g. faces of a polyhedral shape

Not so easy!

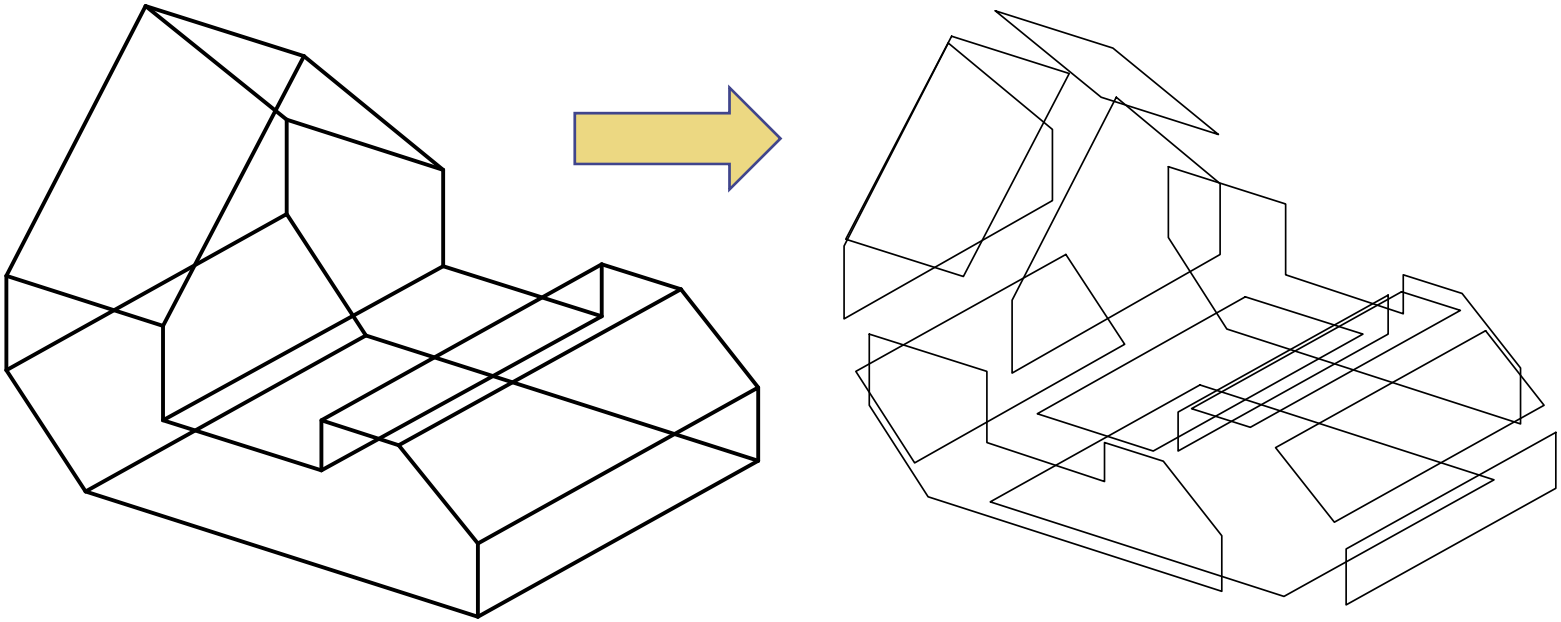
Some heuristics are required to extract this information!

Let us study the finding faces problem!

Finding Faces in Wireframes

Problem:

Given a wireframe line drawing of a polyhedral object, determine a list of loops of edges which correspond to faces of the object



Finding Faces in Wireframes

If we have accurate 3D coordinates for the vertices, it is not so difficult

G. Markowsky and M.A. Wesley, 1980. Fleshing Out Wire Frames, IBM Journal of Research and Development, 24(5) 582–597.]

Without accurate 3D coordinates, it is not so easy

Shpitalni and Lipson determine all possible sets of loops of edges, and then use heuristics to pick the best one

This works, more or less, but is very slow (of the order of *days*)

M. Shpitalni and H. Lipson, 1996. Identification of Faces in a 2D Line Drawing Projection of a Wireframe Object, IEEE Transactions on Pattern Analysis and Machine Intelligence 18(10), 1000–1012.

Liu and Tang use a genetic algorithm

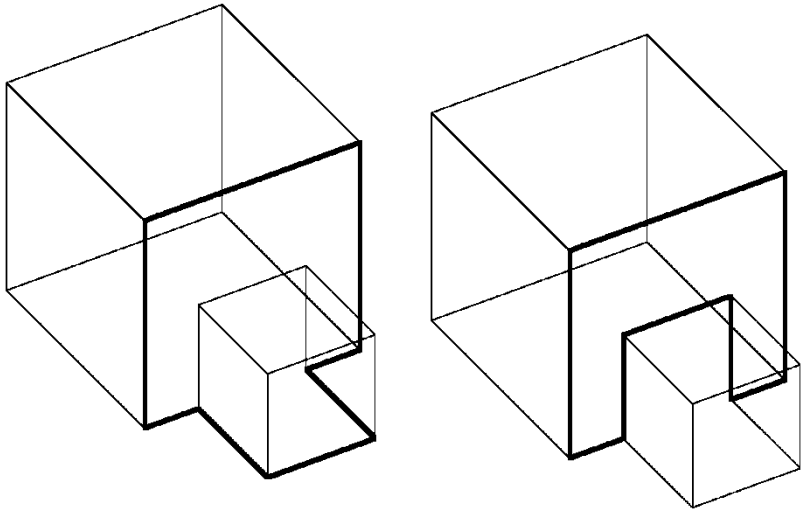
This also works, more or less, but it can never be fully reliable (it is driven by random numbers)

It also needs a lot of tuning

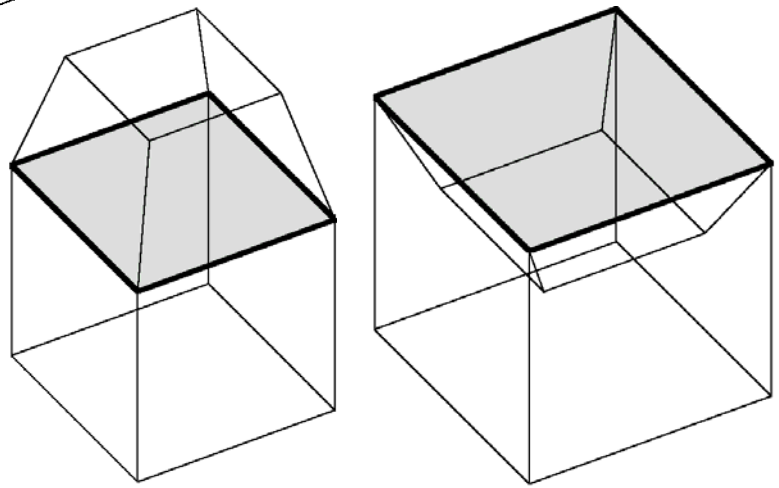
J. Liu and X. Tang, 2005. Evolutionary Search for Faces from Line Drawings, IEEE Transactions on Pattern Analysis and Machine Intelligence 27(6), 861–872.

Finding Faces in Wireframes

So what are the difficulties?



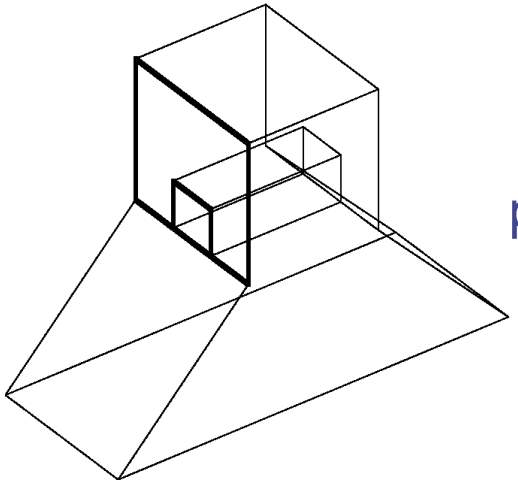
Loop Ambiguity: Wrong and Right Choices



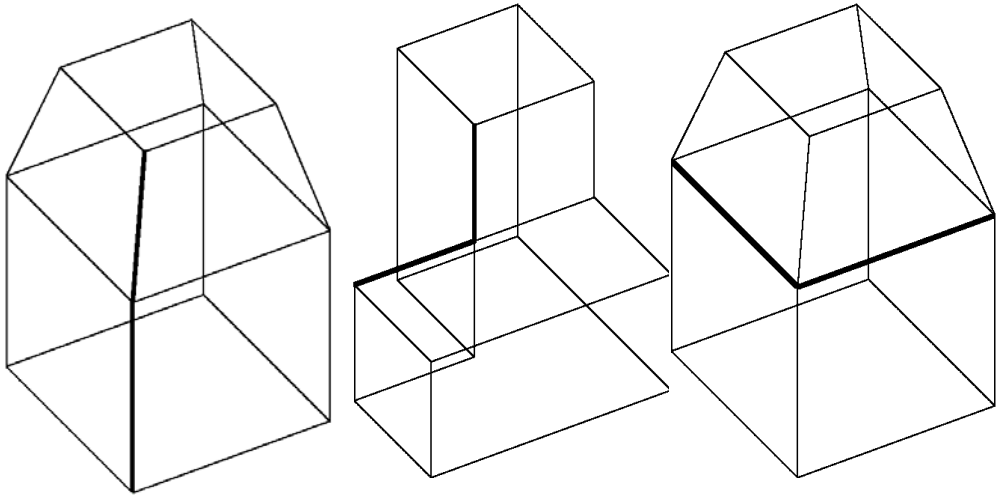
Internal Faces

Finding Faces in Wireframes

More difficulties



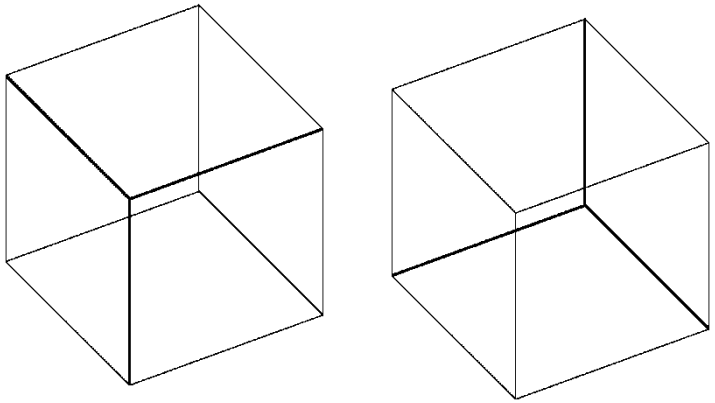
Multiple circuits in a planar edge subgraph



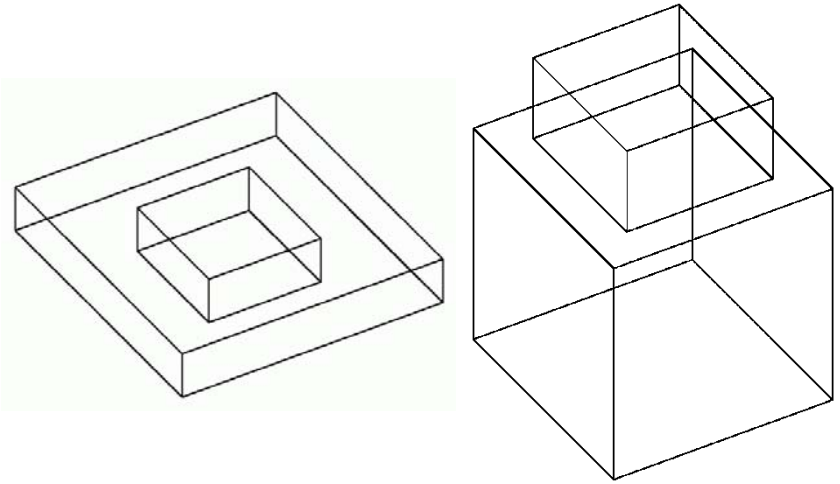
Edge pairs not in true faces

Finding Faces in Wireframes

More difficulties



Necker reversal



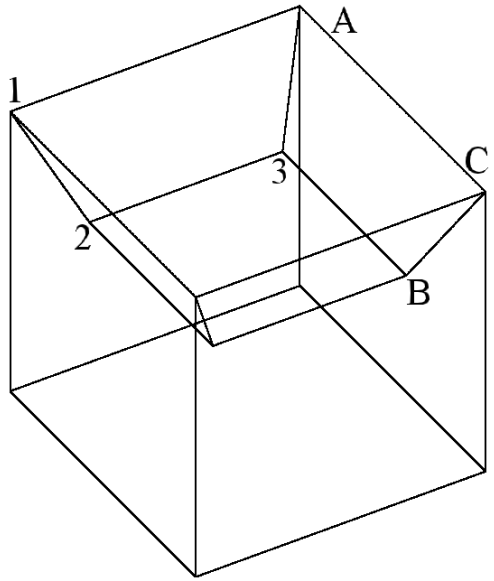
Objects with distinct subgraphs

Finding Faces in Wireframes

It is a graph theory problem

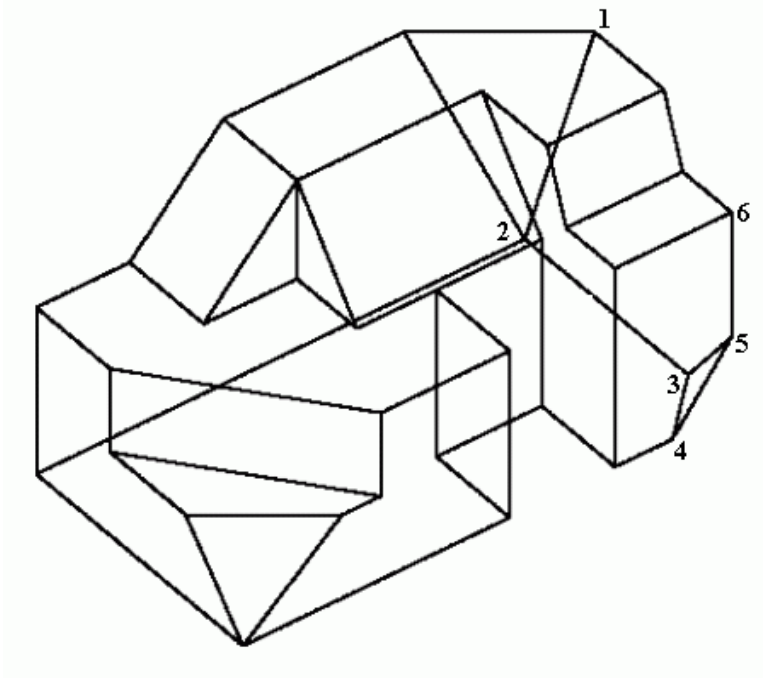
↳ Why not use Dijkstra's Algorithm (or something like it) to pick off loops one by one?

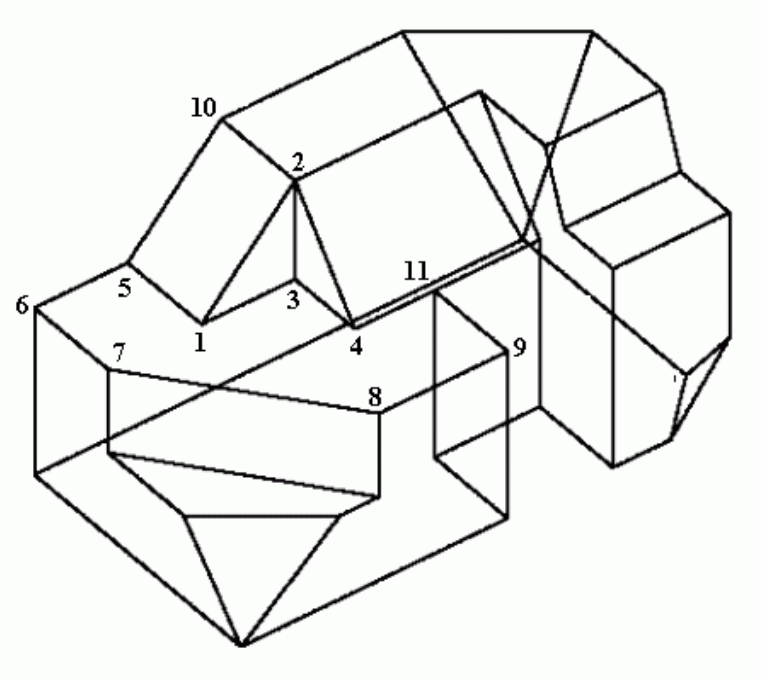
↳ This often works, but sometimes leads to problems



Finding Faces in Wireframes

More problems with Dijkstra's Algorithm approaches





Finding Faces in Wireframes

So what is going wrong?

The fundamental problem with Dijkstra's Algorithm approaches is that they assume a fixed cost for traversing any edge, irrespective of the route taken to reach the edge

We do not want this
we want the cost of traversing an edge to be a function of how well it fits in with any particular loop, taking into account the route taken to reach the edge

What we want is a graph algorithm which allows for the cost of traversing an edge to be context-dependent

We could not find one in the literature, so we came up with our own

Finding Faces in Wireframes

Data Structures:

Strings are *concatenated* sequences of half-edges

The shortest possible *strings* are single half-edges

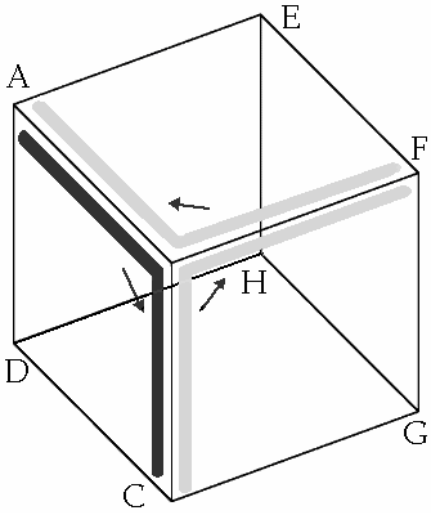
Operations:

Two *strings* can be *concatenated* if the final vertex of the first *string* is also the start vertex of the second *string*, except that:

1. two *strings* cannot be *concatenated* if any other vertex appears in both *strings*
2. two *strings* cannot be *concatenated* if the new triple of three consecutive vertices appears in reverse order in any existing face or already-concatenated *string*

Finding Faces in Wireframes

Data Structures and Operations (examples)



Starting with only the half-edges, we can concatenate AH and HC to give AHC

Once we have AHC, we cannot concatenate CH and HA

In fact, the only possible concatenation of CH is with HF, to give CHF

Similarly, the only possible concatenation of HA is with FH, to give FHA

Finding Faces in Wireframes

Data Structures (continued):

Cyclisation is a double-*concatenation* where the final vertex of each *string* is the same as the start vertex of the other *string*

Cyclisation produces faces

Operations (continued):

the same rules apply to *cyclisation* as to *concatenation*

since the purpose of the algorithm is to produce faces, *cyclisation* takes priority over other operations

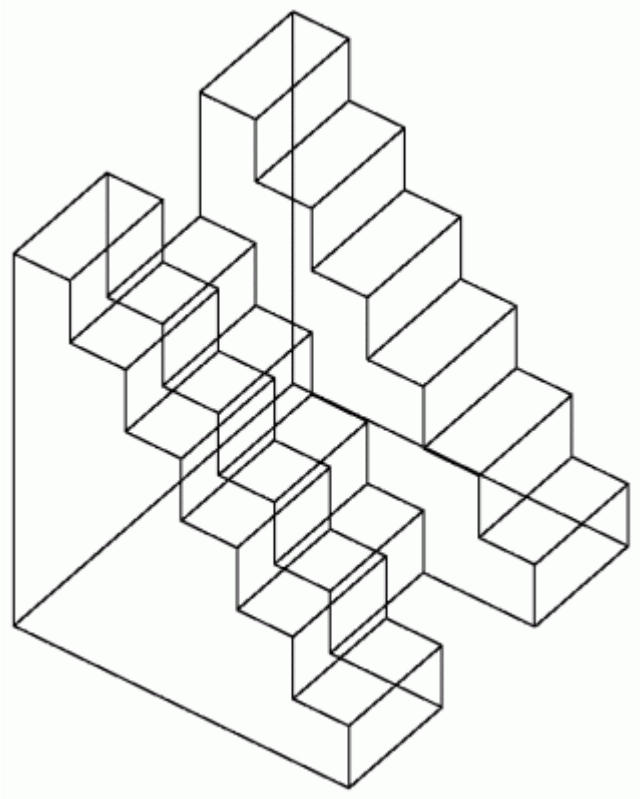
Finding Faces in Wireframes

Finding small (often quadrilateral) faces is easy

Finding large (often irregularly-shaped) faces is not so easy – there is more opportunity to go wrong

So we want to find small faces first, leaving the larger faces until the end

But how do we know which face loops are going to be the small ones?



Finding Faces in Wireframes

The procedure may be summarised as follows:

1. Start with several seeds
2. Add to each seed simultaneously (or in turn) until one of them (the smallest) turns out to be a face loop

> Don't worry if one of them takes a wrong turn somewhere (one of the others will generally finish first)

3. When we have a face loop, discard the rest and start again
4. We implement this by maintaining two lists of strings:
   - ✓ The master list records only things which must be true
   - ✓ The working list is used to explore hypotheses

   When the hypotheses produce a face loop, add this to the master list and throw away the rest of the current working list

(More details in ...) Varley P.A.C. and Company P. (2010) A new algorithm for finding faces in wireframes. Computer-Aided Design 42 (4), 279-309

Finding Faces in Wireframes

The resulting algorithm is short and easy to implement:

(Top level of algorithm)

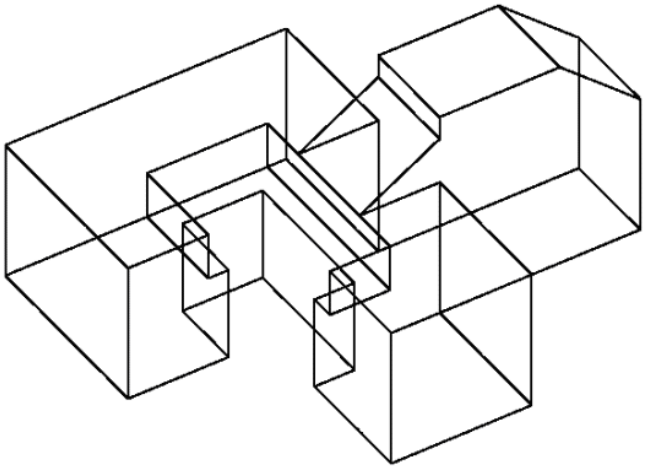
- Create initial master string list, two entries per edge
- Assign priorities to all strings in the list
- Choose a trihedral vertex, and concatenate two strings at this vertex
- While there are strings remaining in the master list
    - Examine the master list for the presence of forced concatenations
    - If there are forced concatenations, perform them
    - Otherwise, examine the master list for the presence of voluntary mergers
        - If there are voluntary mergers, perform them
        - Otherwise, examine hypotheses (see next column)

(Subroutine: Examine Hypotheses)

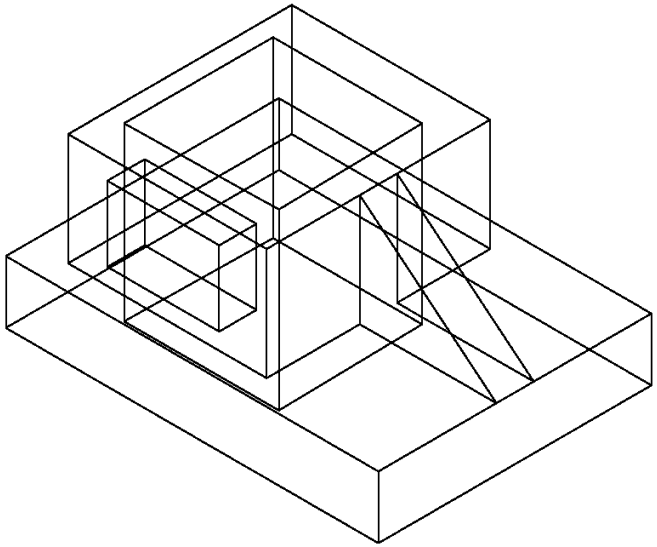
- Take a working copy of the master string list
- Repeat
    - Take the highest-priority string *S* in the working string list
    - Find the string *T* which has the best *mating value* with *S*
    - Concatenate *S* and *T*, and reduce the priority of the resulting string
    - Repeat
        - Examine the working string list for the presence a forced concatenation
        - If there is a forced concatenation, perform it
            - If the forced concatenation created a new face, update the master list accordingly and exit this subroutine
        - If there is no forced concatenation
            - If there is a voluntary merger available, create the face, update the master list accordingly and exit this subroutine
            - Otherwise exit this inner loop

Finding Faces in Wireframes

When tested on 84 drawings, the new algorithm got them all right except for these two:



A previous approach using Dijkstra's Algorithm failed altogether on 19 drawings and got the wrong answer on another 2



The other previous state-of-the-art approach, a genetic algorithm by Liu and Tang, has not been tested on complex drawings such as the two for which our new algorithm fails – every drawing which the genetic algorithm processes correctly is also processed correctly by the new algorithm

Finding Faces in Wireframes

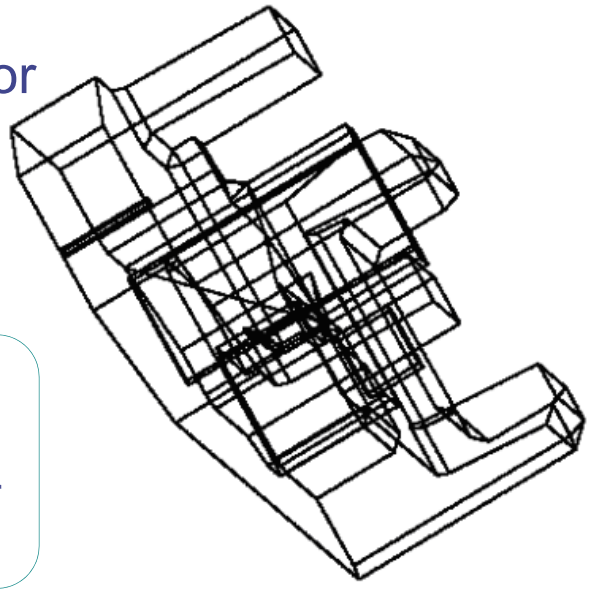
How fast is it?

- ✓ The algorithm is polynomial - counting loops would suggest a worst-case performance of $O(n^5)$

  In practice, processing a sequence of similar drawings, the time complexity is around $e^{2.7}$ (where e is the number of edges)

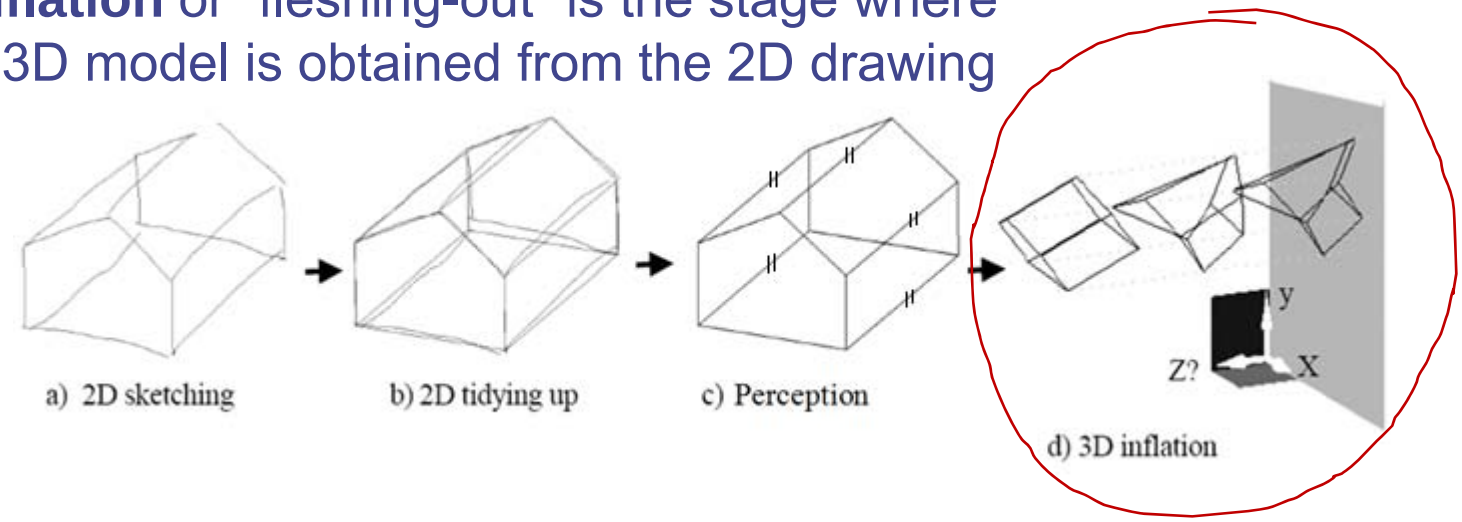
- ✓ Dijkstra's Algorithm is noticeably faster for all drawings because of its low time constant, but the difference is greater for smaller drawings - our new algorithm actually has a *better* practical time complexity

  When applied to our most complex drawing (251 edges), the new algorithm took slightly more than one second – it is fast enough for use in interactive systems



Inflation

Inflation or “fleshing-out” is the stage where a 3D model is obtained from the 2D drawing



Two different strategies coexist:

1. Direct inflation — Without intermediate solutions
2. Iterative inflation — When tentative solutions are tested en route to the final solution

Inflation

Two kinds of direct inflation approaches can be considered:

1 When enough information is available

i.e. Line drawings of **semi-normalons** allow direct inflation

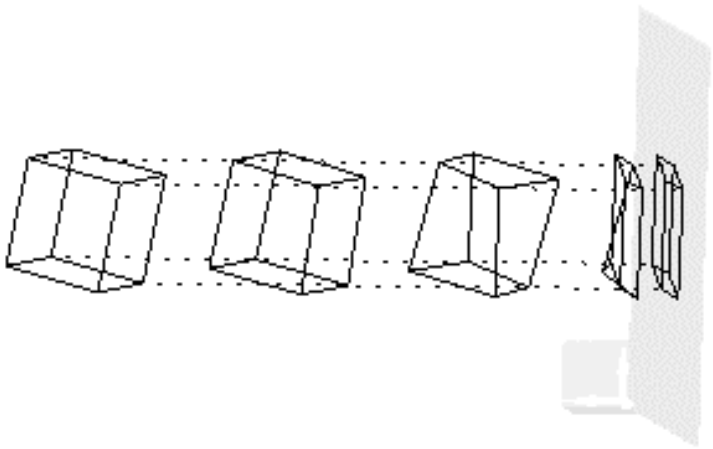
2 When geometrical information is incomplete and perceptual information is ambiguous

i.e. **Linear programming**

(More details Annex 7)

Inflation

When direct inflation does not work, **iterative approaches** are used



The most frequent strategy is:

1. The multiple heuristics are formulated as **compliance functions**
2. The compliance functions are combined to produce a single **objective function**
3. The solution which minimises/maximises the objective function is sought by way of **mathematical optimisation** strategies

Inflation

The procedure may be summarised as follows:

1. An “inflation” reference system is defined

2. (x_i, y_i) coordinates of every junction in the 3D model are made equal to (x_i, y_i) coordinates of the corresponding vertex in the drawing

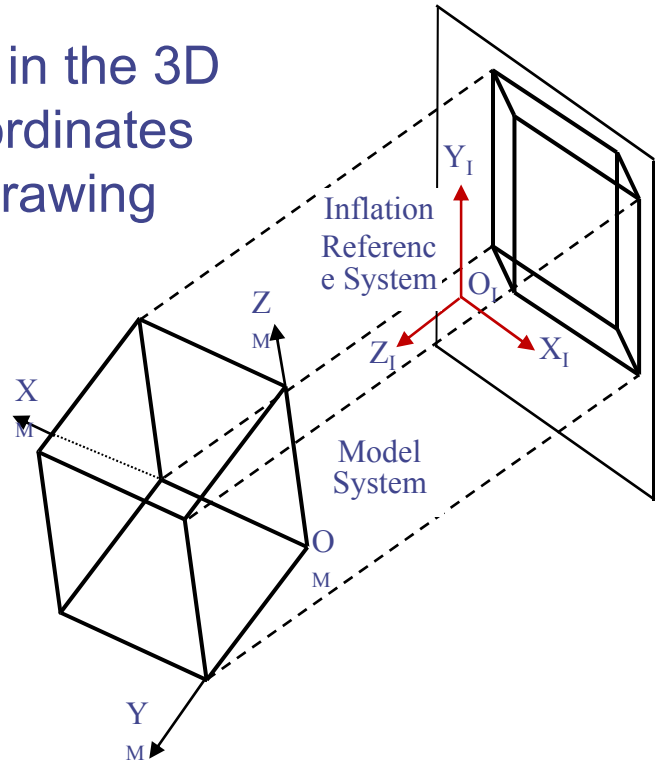
3. The z coordinates of the nodes are used as independent variables in the Objective Function:

$$F(\mathbf{z}) = \sum \alpha_j R_j(\mathbf{z})$$

where α_j is the j-th weighting coefficient, and $R_j(z)$ is the j-th heuristic, expressed in terms of the independent variables z

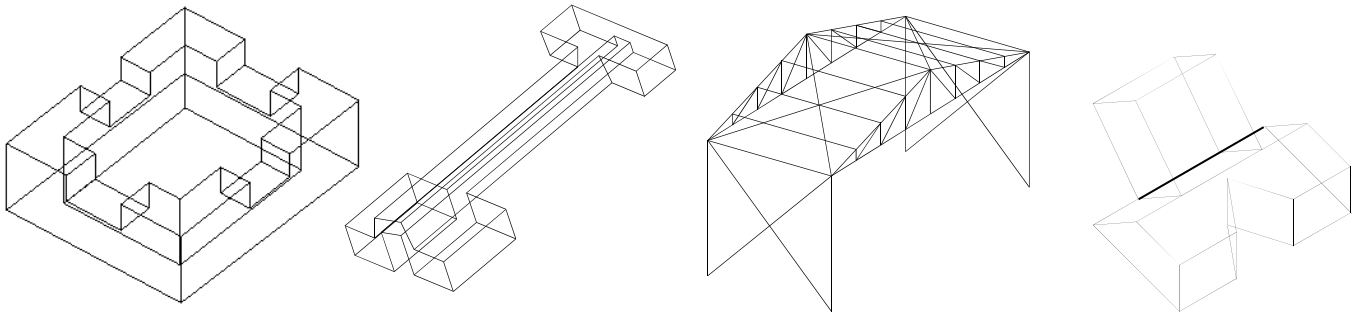
Heuristics must be formulated so as to be equal to zero when complete compliance of the condition is achieved, and very different from zero for clear non-compliance

4. Solve **z** that minimises F



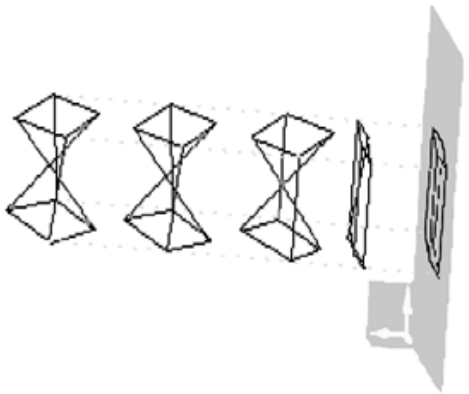
Inflation

The approach succeeds in inflating different shapes...



...but many bottlenecks prevent it from becoming **robust**:

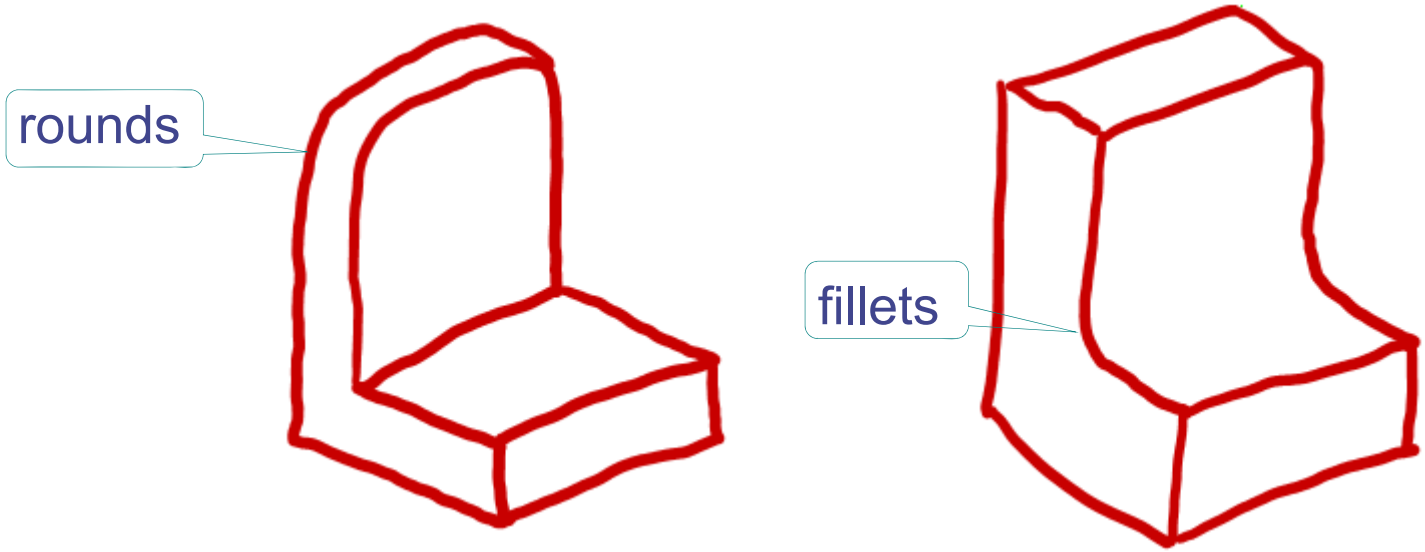
- ✗ Complex or poorly defined compliance functions are not mathematically resolvable
- ✗ Failures in perceiving design intent prevent the Objective Function from conveying some shapes
- ✗ Failures in optimisation algorithms give rise to local minima



(More details in Annex 7)

Rounds and fillets

Some features are better added during **refinement** of a previously produced 3D model:



...embedded in polyhedral shapes

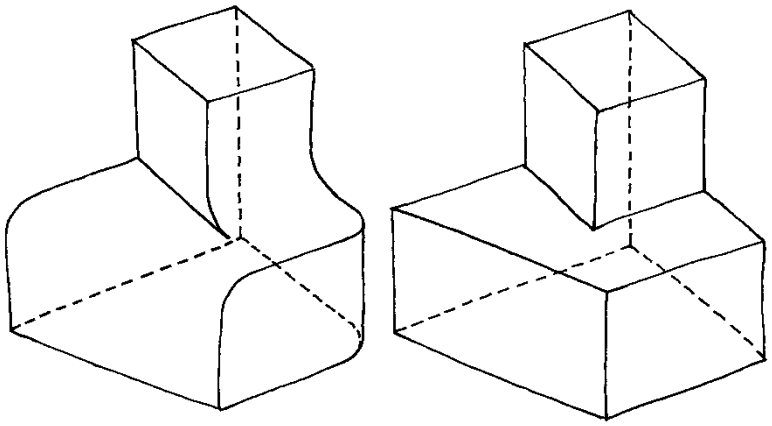
Rounds and fillets

Adding them automatically at the end will give us two **advantages**:

1 reduces the workload of the designer

2 isolates features which play specific roles in designed parts

Avoiding the step of obtaining the mind's eye image of the polyhedral skeleton



When designer wants this…

…draws this…

… and then adds rounds and fillets to the final model

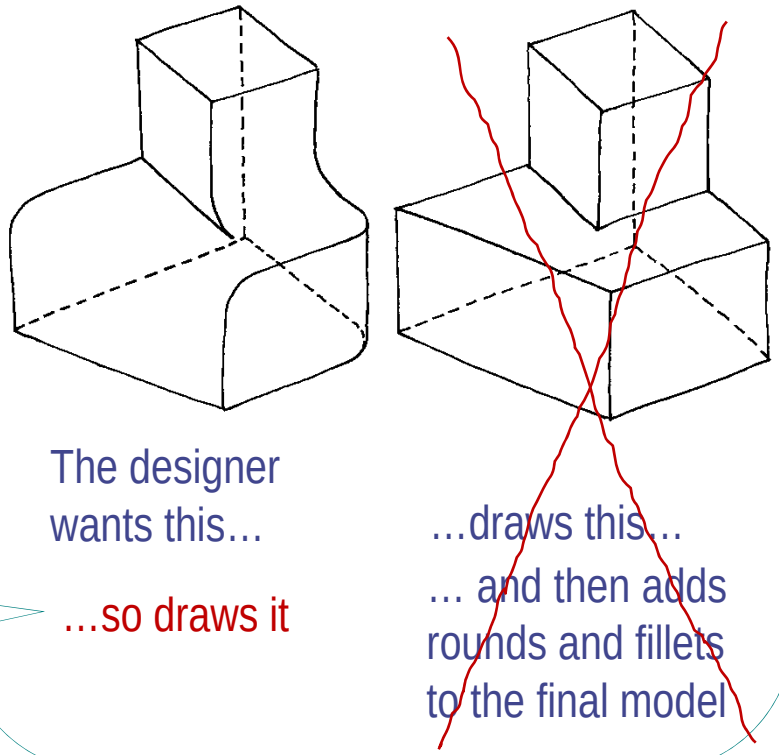
Rounds and fillets

Adding them automatically at the end will give us two **advantages**:

1 reduces the workload of the designer

2 isolates features which play specific roles in designed parts

Avoiding the step of obtaining the mind's eye image of the polyhedral skeleton



Some current approaches allow this strategy… but they miss the second advantage

Rounds and fillets

Adding them automatically at the end will give us two **advantages**:

1 reduces the workload of the designer

2 isolates features which play specific roles in designed parts

Such features can be efficiently managed as independent features by current geometrical engines

http://www.plm.automation.siemens.com/en_us/products/open/parasolid/functionality/index.shtml

Solving them through reconstruction strategies is inefficient!

Rounds and fillets

The algorithm has four main stages:

1. detect rounded edges and fillets
2. obtain the polyhedral skeleton
3. reconstruct the skeleton
4. add rounded edges and fillets

Rounds and fillets

The algorithm has four main stages:

1. detect rounded edges and fillets
2. obtain the polyhedral skeleton
3. reconstruct the skeleton
4. add rounded edges and fillets

We described this part earlier

Rounds and fillets

The algorithm has four main stages:

1. detect rounded edges and fillets
2. obtain the polyhedral skeleton
3. reconstruct the skeleton
4. add rounded edges and fillets

This is a common task in current CAD modelling environments, which encourage the users to create “skeletons” and then add rounds manually

Current geometrical engines of CAD applications are quite efficient in managing rounds as separate features added on top of model trees

Current academic interest deals with more complex and subtle variations such as infinitely sharp and semi-sharp edges

Rounds and fillets

The algorithm has four main stages:

1. detect rounded edges and fillets
2. obtain the polyhedral skeleton
3. reconstruct the skeleton
4. add rounded edges and fillets

Stages 1 and 2 are new
and it is these we describe in more detail

Rounds and fillets

1 The two steps to detect rounded edges and fillets are:

1 Detect circular arcs

Circular arcs are projected as elliptical arcs

Detecting them is easy after segmenting the sketch strokes into simple lines – this is a solved problem for tidied hand-drawn line-drawings

2 Form pairs of circular arcs

Rounds and fillets

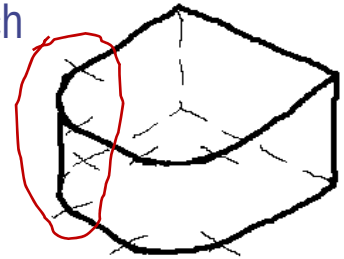
1 The two steps to detect rounded edges and fillets are:

1 Detect circular arcs

2 Form pairs of circular arcs

This task is done as follows:

1 Pair those arcs which are contained in parallel faces and share a tangent contour line



2 For the remaining arcs, pair those arcs which are:

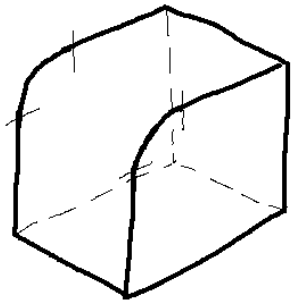
- ✓ contained in parallel faces
- ✓ similar in size and orientation
- ✓ connected to mutually parallel lines

Rounds and fillets

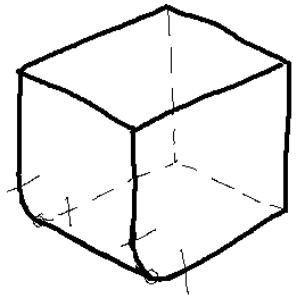
Rounds applied to single edges in quasi-normalon shapes fall in one of three categories:

a) both arcs are fully visible, and no one line connects them

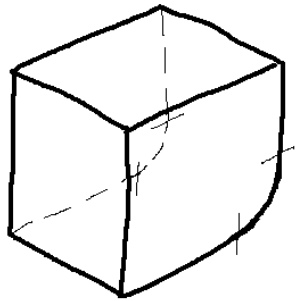
since the edge has disappeared because of the rounding operation



b) one arc is fully visible, the other is partially occluded and one contour line is tangent to both

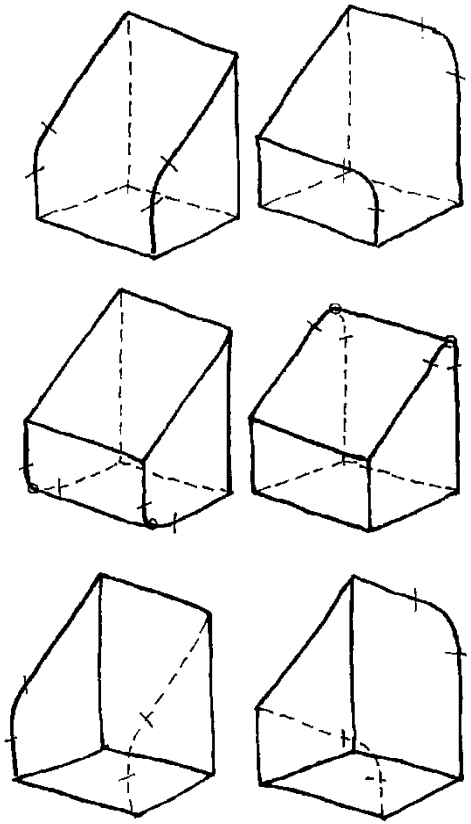


c) one arc is fully visible, the other is fully occluded and no one line connects both



Rounds and fillets

Rounds in oblique edges of quasi-normalon shapes can be classified into the same three categories



It does not matter whether or not the rounded edge meets at 90° with the other edges connected to the same junction

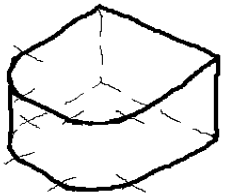
Fillets in quasi-normalon shapes can only be classified into the second and third categories

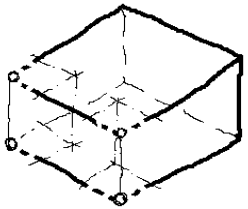
since tangent contour lines may never appear (as fillets are concave shapes and may not belong to the contour of quasi-normalon shapes)

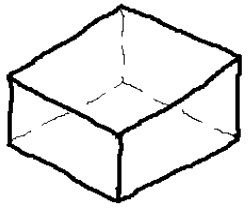
Rounds and fillets

2 The sequence to obtain the polyhedral skeleton is:

- Repeat for every pair of arcs
  - Repeat for both arcs in the pair
    - Suppress the arc
    - Extend the lines connected to its ends until they intersect
    - The intersection point is one new vertex
  - Add an edge connecting the new vertices



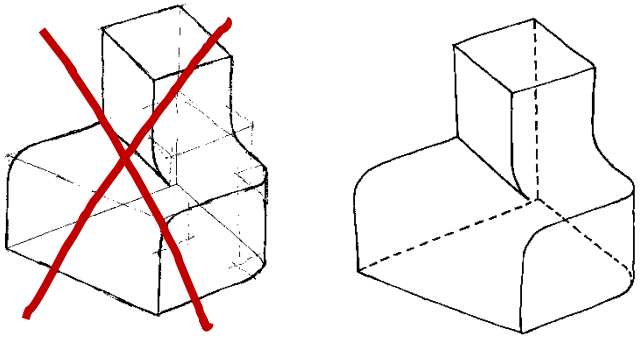




Rounds and fillets

Our current approach has some obvious limitations:

1 Our inputs are tidied up line drawings



2 Polyhedrons must be normalon or quasi-normalon

3 Drawings must be wireframe

Summary

We have described
the **main stages** in an SBM process

1. 2D sketching
2. 2D beautification or tidying up
3. Extraction of geometrical and perceptual information
4. Inflating a rough 3D model
5. 3D model refinement

We have also described several **algorithms** for solving critical stages when the inputs are **wireframe drawings**:

1. Finding faces for polyhedral shapes
2. Inflating polyhedral shapes
3. Rounds and fillets

Summary

Problem

If we have accurate 3D coordinates for the vertices, **finding faces** is not so difficult

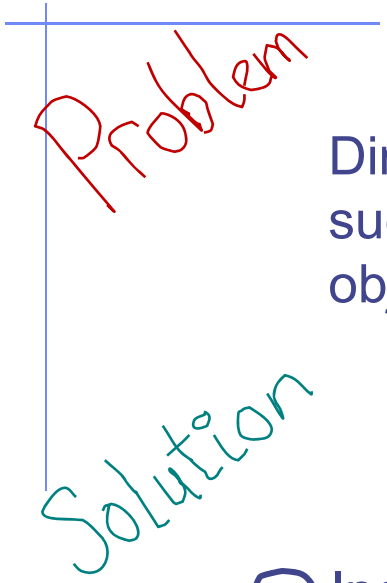
Without accurate 3D coordinates, it is not so easy

Solution

We have described a new algorithm:

- ✓ It is fast enough for an interactive system
- ✓ Not as fast as Dijkstra's Algorithm for smaller drawings, but has a *better* practical time complexity

Summary



Direct and iterative **inflation** strategies have a low success ratio and only solve particular types of object

? Incremental improvements may help?

? A new paradigm is required?

Summary

Rounds and fillets are currently added after skeleton-like polyhedral shapes are automatically modeled from sketches

This strategy clearly overloads the designer

The designer is forced to:

- ✓ convert the original shape formed in his mind's eye into a simplified skeleton
- ✓ depict it as an input sketch for the reconstruction system
- ✓ wait for the reconstruction system to output a 3D model
- ✓ remember where rounded edges and fillets were located and add them through current CAD application operations

Summary

Solution

Our novel algorithm automatically reconstructs polyhedral shapes with rounds and fillets

The algorithm works in four main stages:

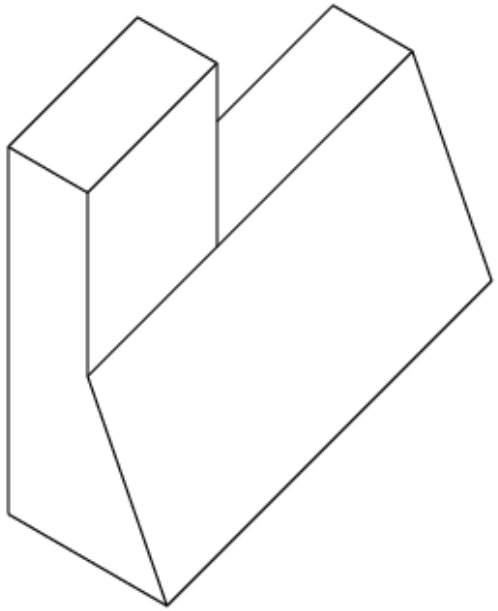
1. detect rounded edges and fillets
2. obtain the polyhedral skeleton
3. reconstruct the skeleton
4. add rounded edges and fillets

New!

Next presentations

In the **third** presentation we will describe some **algorithms** for solving stages of an SBM process when the inputs are **natural drawings**:

1. Line Labelling
2. Inflation to 2½D
3. Hidden Topology



Starts ** time **