UNIT 5

RAPID PROTOTYPING DATA FORMATS

6.1 STL FORMAT

Representation methods used to describe CAD geometry vary from one system to another. A standard interface is needed to convey geometric descriptions from various CAD packages to rapid prototyping systems. The STL (STereoLithography) file, as the *de facto* standard, has been used in many, if not all, rapid prototyping systems.

The STL file [1–3], conceived by the 3D Systems, USA, is created from the CAD database via an interface on the CAD system. This file consists of an unordered list of triangular facets representing the outside skin of an object. There are two formats to the STL file. One is the ASCII format and the other is the binary format. The size of the ASCII STL file is larger than that of the binary format but is human readable. In a STL file, triangular facets are described by a set of *X*, *Y* and *Z* coordinates for each of the three vertices and a unit normal vector with *X*, *Y* and *Z* to indicate which side of facet is an object. An example is shown in Figure 6.1.

Because the STL file is a facet model derived from precise CAD models, it is, therefore, an approximate model of a part. Besides, many commercial CAD models are not robust enough to generate the facet model (STL file) and frequently have problems.

Nevertheless, there are several advantages of the STL file. First, it provides a simple method of representing 3D CAD data. Second, it is already a *de facto* standard and has been used by most CAD systems and rapid prototyping systems. Finally, it can provide small and accurate files for data transfer for certain shapes.

On the other hand, several disadvantages of the STL file exist. First, the STL file is many times larger than the original CAD data file for a given accuracy parameter. The STL file carries much redundancy

```
solid print
 facet normal 0.00000e+00 1.00000e+00 0.00000e+00
  outer loop
    vertex   0.00000e+00 0.00000e+00 2.00000e+01
    vertex   0.00000e+00 0.00000e+00 0.00000e+00
    vertex   1.00000e+01 0.00000e+00 2.00000e+01
  endloop
 endfacet
 facet normal 0.00000e+00 1.00000e+00 0.00000e+00
  outer loop
    vertex   1.00000e+01 0.00000e+00 2.00000e+01
    vertex   0.00000e+00 0.00000e+00 0.00000e+00
    vertex   1.00000e+01 0.00000e+00 0.00000e+00
  endloop
 endfacet
.....
```

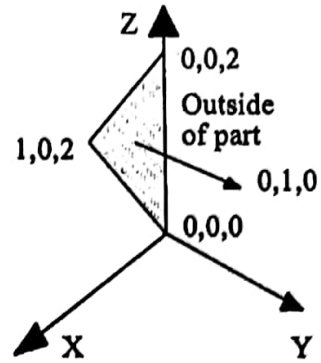


Figure 6.1: A sample STL file

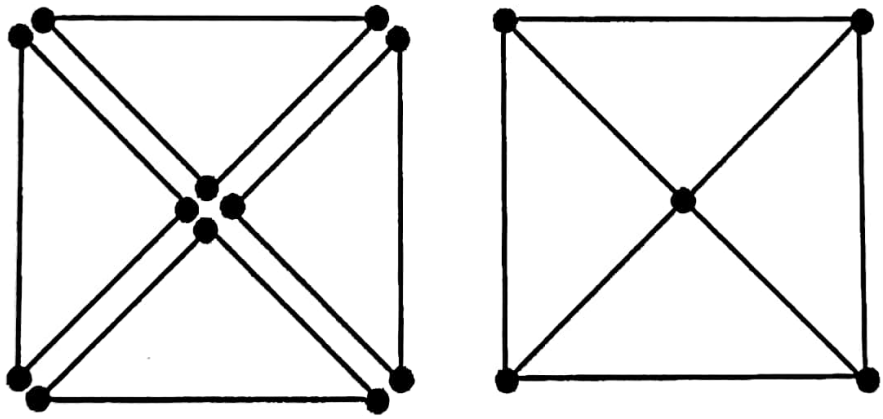


Figure 6.2: Edge and vertex redundancy in STL format

information such as duplicate vertices and edges shown in Figure 6.2. Second, the geometry flaws exist in the STL file because many commercial tessellation algorithms used by CAD vendor today are not robust. This gives rise to the need for a "repair software" which slows the production cycle time. Finally, the subsequent slicing of large STL files can take many hours. However, some RP processes can slice while they are building the previous layer and this will alleviate this disadvantage.

STL FILE PROBLEMS

Several problems plague STL files and they are due to the very nature of STL files as they contain no topological data. Many commercial tessellation algorithms used by CAD vendors today are also not robust [4–6], and as a result they tend to create polygonal approximation models which exhibit the following types of errors:

(1) Gaps (cracks, holes, punctures) that is, missing facets.
(2) Degenerate facets (where all its edges are collinear).
(3) Overlapping facets.
(4) Non-manifold topology conditions.

The underlying problem is due, in part, to the difficulties encountered in tessellating trimmed surfaces, surface intersections and controlling numerical errors. This inability of the commercial tessellation algorithm to generate valid facet model tessellations makes it necessary to perform model validity checks before the tessellated model is sent to the Rapid Prototyping equipment for manufacturing. If the tessellated model is invalid, procedures become necessary to determine the specific problems, whether they are due to gaps, degenerate facets or overlapping facets, etc.

Early research has shown that repairing invalid models is difficult and not at all obvious [7]. However, before proceeding any further into discussing the procedures that are generated to resolve these difficulties, the following sections shall clarify the problems, as mentioned earlier. In addition, an illustration would be presented to show the consequences brought about by a model having a missing facet, that is, a gap in the tessellated model.

Missing Facets or Gaps

Tessellation of surfaces with large curvature can result in errors at the intersections between such surfaces, leaving gaps or holes along edges of the part model [8]. A surface intersection anomaly which results in a gap is shown in Figure 6.3.

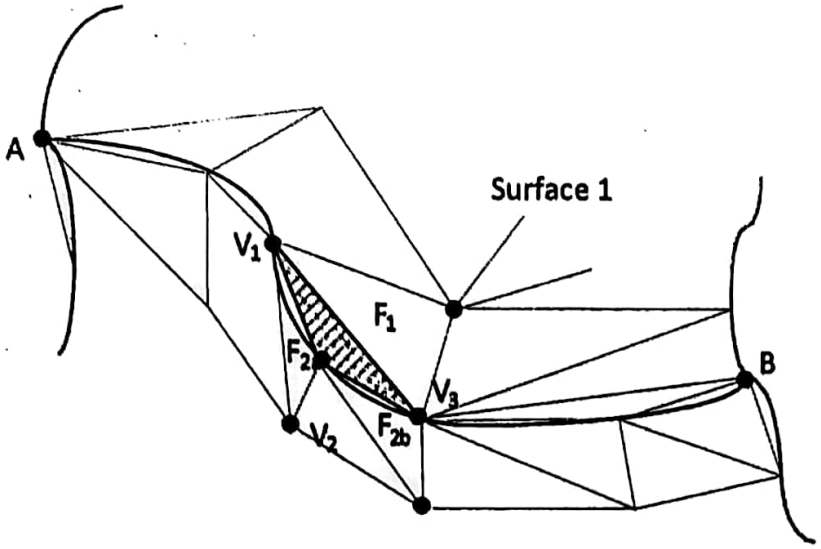


Figure 6.3: Gaps due to missing facets [4]

Degenerate Facets

A geometrical degeneracy of a facet occurs when all of the facets' edges are collinear even though all its vertices are distinct. This might be caused by stitching algorithms that attempt to avoid shell punctures as shown in Figure 6.4(a) below [9].

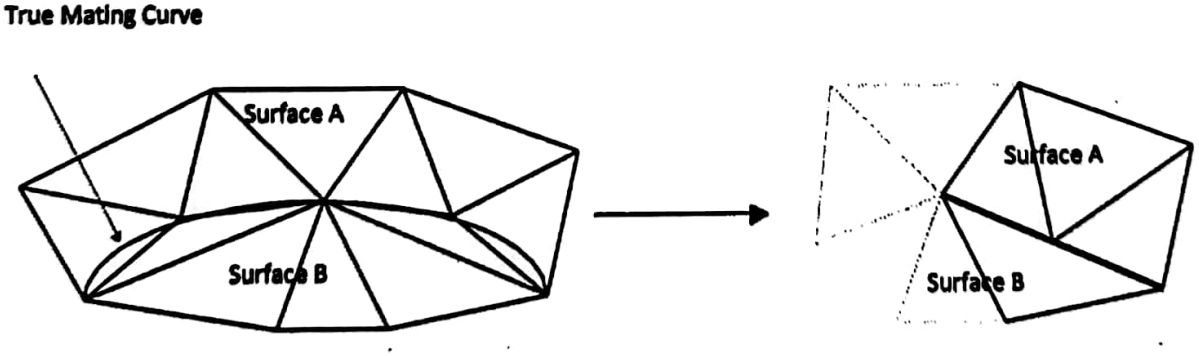


Figure 6.4(a): Shell punctures created by unequal tessellation of two adjacent surface patches along their common mating curve

Figure 6.4(b): Shell punctures elimlnated at the expense of adding degenerate facet

The resulting facets generated, shown in Figure 6.4(b), eliminate the shell punctures. However, this is done at the expense of adding a degenerate facet. While degenerate facets do not contain valid surface

normals, they do represent implicit topological information on how two surfaces mated. This important information is consequently stored prior to discarding the degenerate facet.

Overlapping Facets

Overlapping facets may be generated due to numerical round-off errors occurring during tessellation. The vertices are represented in 3D space as floating point numbers instead of integers. Thus the numerical round-off can cause facets to overlap if tolerances are set too liberally. An example of an overlapping facet is illustrated in Figure 6.5.

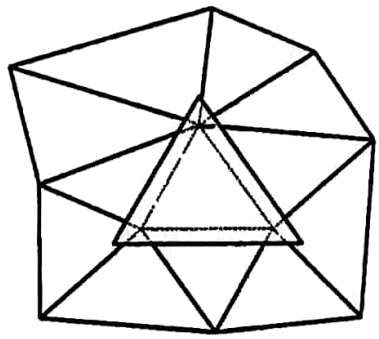


Figure 6.5: Overlapping facets

Non-manifold Conditions

There are three types of non-manifold conditions, namely:

(1) A non-manifold edge.
(2) A non-manifold point.
(3) A non-manifold face.

These may be generated because tessellation of the fine features are susceptible to round-off errors. An illustration of a non-manifold edge is shown in Figure 6.6(a). Here, the non-manifold edge is actually shared by four different facets as shown in Figure 6.6(b). A valid model would be one whose facets have only an adjacent facet each, that is, one edge is shared by two facets only. Hence the non-manifold edges must be resolved such that each facet has only one neighboring facet

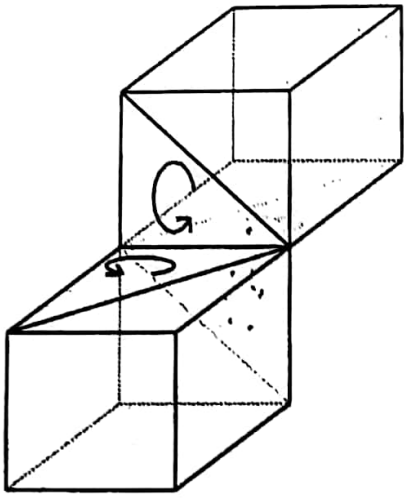


Figure 6.6(a): A non-manifold edge whereby two imaginary minute cubes share a common edge

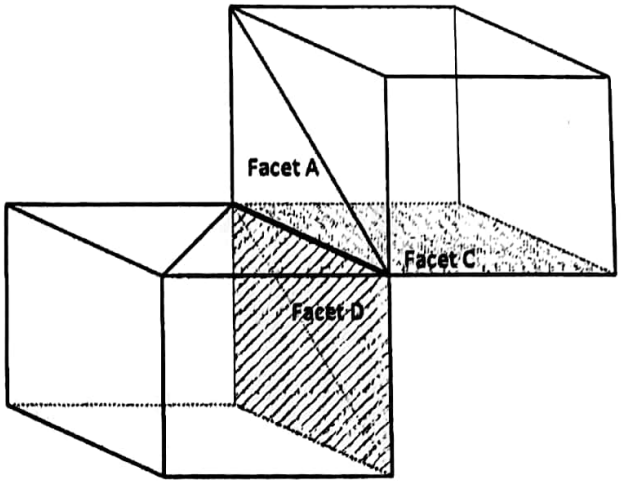


Figure 6.6(b): A non-manifold edge whereby four facets share a common edge

after tessellation

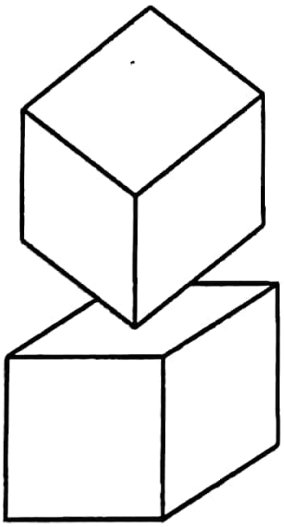


Figure 6.6(c): Non-manifold point

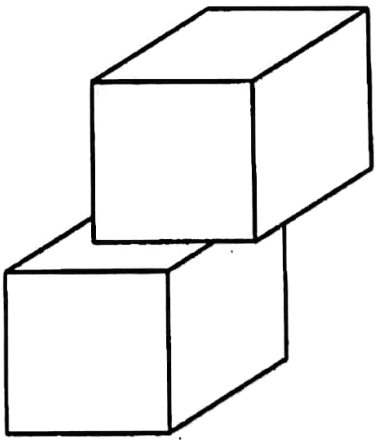


Figure 6.6(d): Non-manifold face

along each edge, that is, by reconstructing a topologically manifold surface [4]. In Figures 6.6(c) and 6.6(d), two other types of non-manifold conditions are shown.

All problems that have been mentioned previously are difficult for most slicing algorithms to handle and they do cause fabrication problems for RP processes which essentially require valid tessellated solids as input. Moreover, these problems arise because tessellation is

a first-order approximation of more complex geometric entities. Thus, such problems have become almost inevitable as long as the representation of the solid model is done using the STL format which inherently has these limitations.

CONSEQUENCES OF BUILDING A VALID AND INVALID TESSELLATED MODEL

The following sections present an example each of the outcome of a model built using a valid and an invalid tessellated model as an input to the RP systems.

A Valid Model

A tessellated model is said to be valid if there are no missing facets, degenerate facets, overlapping facets or any other abnormalities. When a valid tessellated model (see Figure 6.7(a)) is used as an input, it will first be sliced into 2D layers, as shown in Figure 6.7(b). Each layer would then be converted into unidirectional (or 1D) scan lines for the laser or other RP techniques to commence building the model as shown in Figure 6.7(c).

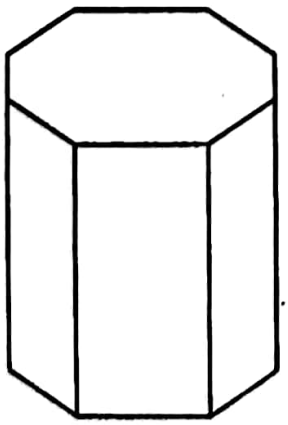


Figure 6.7(a): A valid 3D model

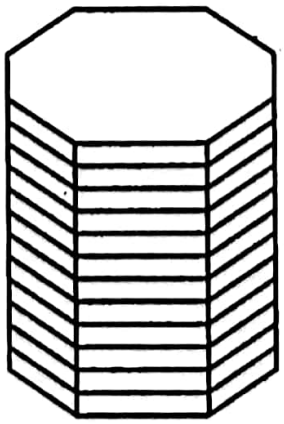


Figure 6.7(b): A 3D model sliced into 2D planar layers

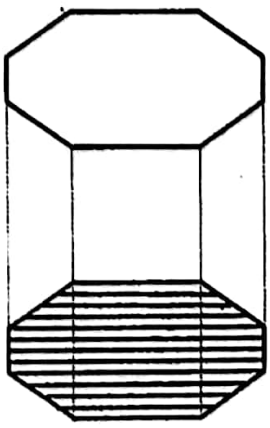


Figure 6.7(c): Conversion of 2D layers into 1D scan lines

The scan lines would act as on/off points for the laser beam controller so that the part model can be built accordingly without any problems.

An Invalid Model

However, if the tessellated model is invalid, a situation may develop as shown in Figure 6.8.

A solid model is tessellated non-robustly and results in a gap as shown in Figure 6.8(a). If this error is not corrected and the model is subsequently sliced, as shown in Figure 6.8(b), in preparation for it to

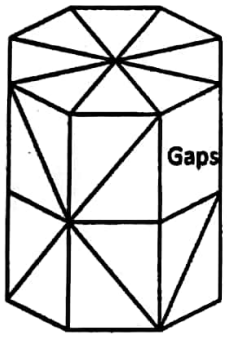


Figure 6.8(a): An invalid tessellated model

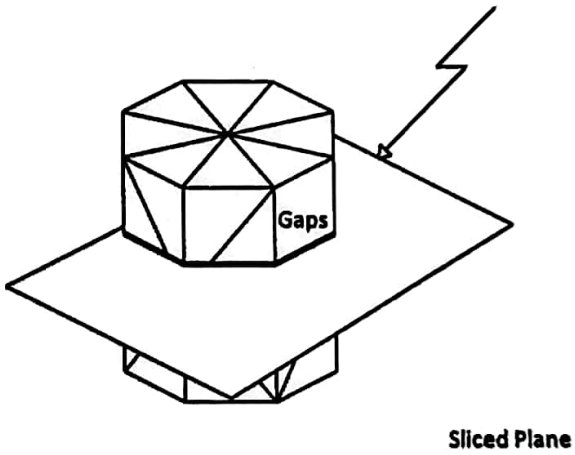


Figure 6.8(b): An invalid model being sliced

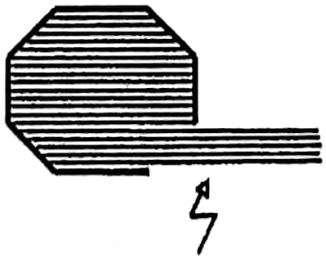


Figure 6.8(c): A layer of an invalid model being scanned

be built layer by layer, the missing facet in the geometrical model would cause the system to have no predefined stopping boundary on the particular slice, thus the building process would continue right to the physical limit of the RP machine, creating a stray physical solid line and ruining the part being produced, as illustrated in Figure 6.8(c).

Therefore, it is of paramount importance that the model be "repaired" before it is sent for building. Thus, the model validation and repair problem is stated as follows:

> ***Given a facet model (a set of triangles defined by their vertices), in which there are gaps, i.e., missing one or more sets of polygons, generate "suitable" triangular surfaces which "fill" the gaps [4].***

STL FILE REPAIR

The STL file repair can be implemented using a generic solution and dedicated solutions for special cases.

Generic Solution

In order to ensure that the model is valid and can be robustly tessellated, one solution is to check the validity of all the tessellated triangles in the model. This section presents the basic problem of missing facets and a proposed generic solution to solve the problem with this approach.

In existing RP systems, when a punctured shell is encountered, the course of action taken usually requires a skilled technician to manually repair the shell. This manual shell repair is frequently done without any knowledge of the designer's intent. The work can be very time-consuming and tedious, thus negating the advantages of rapid prototyping as the cost would increase and the time taken might be longer than that taken if traditional prototyping processes were used.

The main problem of repairing the invalid tessellated model would be that of matching the solution to the designer's intent when it may have been lost in the overall process. Without the knowledge of the

designer's intent, it would indeed be difficult to determine what the "right" solution should be. Hence, an "educated" guess is usually made when faced with ambiguities of the invalid model. The algorithm in this report aims to match, if not exceed, the quality of repair done manually by a skilled technician when information of the designer's intent is not available.

The basic approach of the algorithm to solve the "missing facets" problem would be to detect and identify the boundaries of all the gaps in the model. Once the boundaries of the gap are identified, suitable facets would then be generated to repair and "patch up" these gaps. The size of the generated facets would be restricted by the gap's boundaries while the orientation of its normal would be controlled by comparing it with the rest of the shell. This is to ensure that the generated facets' orientation are correct and consistent throughout the gap closure process.

The orientation of the shell's facets can be obtained from the STL file which lists its vertices in an ordered manner following Mobius' rule. The algorithm exploits this feature so that the repair carried out on the invalid model, using suitably created facets, would have the correct orientation.

Thus, this generic algorithm can be said to have the ability to make an inference from the information contained in the STL file so that the following two conditions can be ensured:

(1) The orientation of the generated facet is correct and compatible with the rest of the model.
(2) Any contoured surface of the model would be followed closely by the generated facets due to the smaller facet generated. This is in contrast to manual repair whereby, in order to save time, fewer facets generated to close the gaps are desired, resulting in large generated facets that do not follow closely to the contoured surfaces.

Finally, the basis for the working of the algorithm is due to the fact that in a valid tessellated model, there must only be two facets sharing every edge. If this condition is not fulfilled, then this indicates that there are some missing facets. With the detection and subsequent repair

of these missing facets, the problems associated with the invalid model can then be eliminated.

Solving the "Missing Facets" Problem

The following procedure illustrates the detection of gaps in the tessellated model and its subsequent repair. It is carried out in four steps.

(I) Step 1: Checking for Approved Edges with Adjacent Facets

The checking routine executes as follows for Facet A as seen in Figure 6.9:

(a) (i) Read in first edge {vertex 1-2} from the STL file.
(ii) Search file for a similar edge in the opposite direction {vertex 2-1}.
(iii) If edge exists, store this under a temporary file (e.g., file B) for approved edges.
(iv) Do the same for 2 and 3 below.

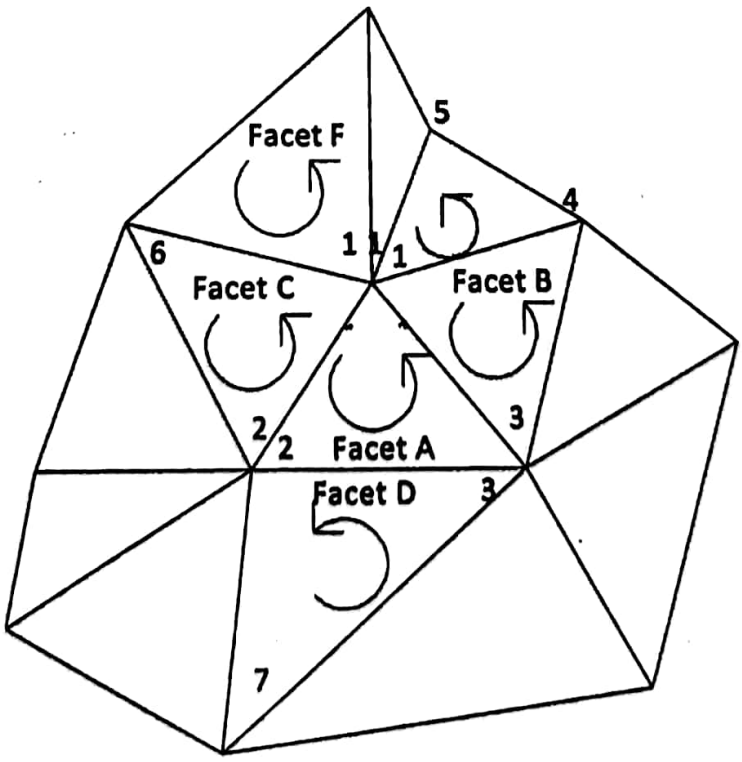


Figure 6.9: A representation of a portion of a tessellated surface without any gaps

(b) (i) Read in second edge {vertex 2-3} from the STL file.
(ii) Search file for a similar edge in the opposite direction {vertex 3-2}.
(iii) Perform as in (a)(iii) above.

(c) (i) Read in third {vertex 3-1} from the STL file.
(ii) Search file for a similar edge in the opposite direction {vertex 1-3}.
(iii) Perform as in (a)(iii) above.

This process is repeated for the next facet until all the facets have been searched.

(2) Step 2: Detection of Gaps in the Tessellated Model

The detection routine executes as follows:

For Facet A (please refer to Figure 6.10):

(a) (i) Read in edge {vertex 2-3} from the STL file.
(ii) Search file for a similar edge in the opposite direction {vertex 3-2}.
(iii) If edge does not exist, store edge {vertex 3-2} in another temporary file (e.g., file C) for suspected gap's bounding edges and store vertex 2-3 in file B1 for existing edges without adjacent facets (this would be used later for checking the generated facet orientation).

For Facet B,

(b) (i) Read in edge {vertex 5-2} from the STL file.
(ii) Search file for a similar edge in the opposite direction {vertex 2-5}.
(iii) If it does not exist, perform as in (a)(iii) above.

(c) (i) Repeat for edges: 5-2 ; 7-5 ; 9-7 ; 11-9 ; 3-11.
(ii) Search for edges: 2-5 ; 5-7 ; 7-9 ; 9-11 ; 11-3.
(iii) Store all the edges in that temporary file B1 for edges without any adjacent facet and store all the suspected bounding edges of the gap in temporary file C. File B1 can appear as in Table 6.1.

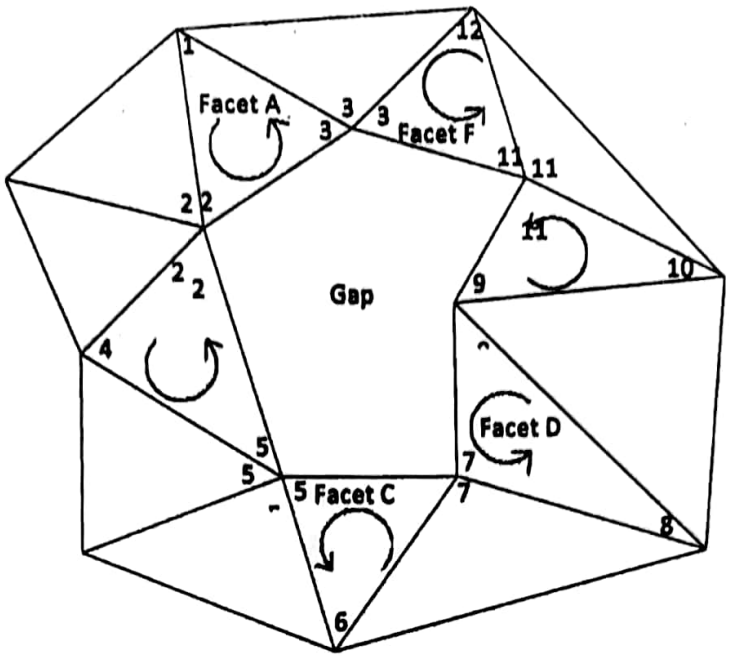


Figure 6.10: A representation of a portion of a tessellated surface with a gap present

Table 6.1: File B1 contains existing edges without adjacent facets

Vertex	Edge					
	First	Second	Third	Fourth	Fifth	Sixth
First	2	7	3	5	9	11
Second	3	5	11	2	7	9

(3) Step 3: Sorting of Erroneous Edges into a Closed Loop

When the checking and storing of edges (both with and without adjacent facets) are completed, a sort would be carried out to group all the edges without adjacent facets to form a closed loop. This closed loop would represent the gap detected and be stored in another temporary file (e.g., file D) for further processing. The following is a simple illustration of what could be stored in file C for edges that do not have an adjacent edge.

Assuming all the "erroneous" edges are stored according to the detection routine (see Figure 6.10 for all the erroneous edges), then file C can appear as in Table 6.2.

Table 6.2: File C containing all the "Erroneous" edges that would form the boundary of each gaps

Vertex	Edge								
	First	Second	Third	Fourth	Fifth	Sixth	Seventh	Eighth	Ninth
First	3	5	*	11	2	7	*	9	*
Second	2	7	*	3	5	9	*	11	*

*Represent all the other edges that would form the boundaries of other gaps

As can be seen in Table 6.2, all the edges are unordered. Hence, a sort would have to be carried out to group all the edges into a closed loop. When the edges have been sorted, it would then be stored in a temporary file, say file D. Table 6.3 is an illustration of what could be stored in file D.

Table 6.3: File D containing sorted edges

	First edge	Second edge	Third edge	Fourth edge	Fifth edge	Sixth edge
First vertex	3	2	5	7	9	11
Second vertex	2	5	7	9	11	3

Figure 6.11 is a representation of the gap, with all the edges forming a sorted closed loop.

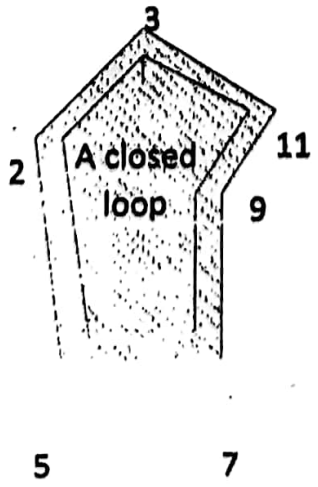


Figure 6.11: A representation of a gap bounded by all the sorted edges

(4) Step 4: Generation of Facets for the Repair of the Gaps

When the closed loop of the gap is established with its vertices known, facets are generated one at a time to fill up the gap. This process is summarized in Table 6.4 and illustrated in Figure 6.12.

Table 6.4: Process of facet generation

		V3	V2	V5	V7	V9	V11
Generation of facets	F1	1	2	—	—	—	3
	F2	E	1	—	—	2	3
	F3	E	1	2	—	3	E
	F4	E	E	1	2	3	E

V = vertex, F = facet, E = eliminated from the process of facet generation

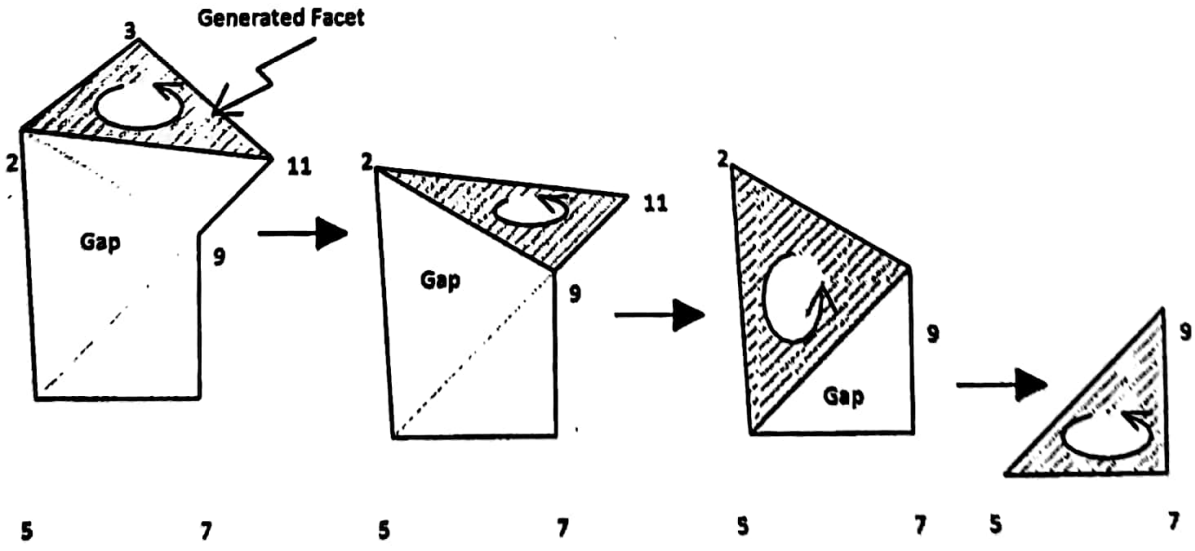


Figure 6.12(a): First facet generated

Figure 6.12(b): Second facet generated

Figure 6.12(c): Third facet generated

Figure 6.12(d): Fourth facet generated

With reference to File D,

(a) Generating the first facet: First two vertices (V3 and V2) in the first two edges of file D will be connected to the first vertex in the last edge (V11) in file D and the facet is stored in a temporary file E (see Table 6.5 on how the first generated facet would be stored in file E). The facet is then checked for its orientation using theinformation stored in file B1. Once its orientation is determined to be correct, the first vertex (V3) from file D will be temporarily removed.

(b) Generating the second facet: Of the remaining vertices in file D, the previous second vertex (V2) will become the first edge of file D. The second facet is formed by connecting the first vertex (V2) of the first edge with that of the last two vertices in file D (V9, V11), and the facet is stored in temporary file E. It is then checked to confirm if its orientation is correct. Once it is determined to be correct, the vertex (V11) of the last edge in file D is then removed temporarily.

(c) Generating the third facet: The whole process is repeated as it was done in the generation of facets 1 and 2. The first vertex of the first two edges (V2, V5) is connected to the first vertex of the last edge (V9) and the facet is stored in temporary file E. Once its orientation is confirmed, the first vertex of the first edge (V2) will be removed from file D temporarily.

(d) Generating the fourth facet: The first vertex in the first edge will then be connected to the first vertices of the last two edges to form the fourth facet and it will again be stored in the temporary file E. Once the number of edges in file D is less than three, the process of facet generation will be terminated. After the last facet is generated, the data in file E will be written to file A and its content (file E's) will be subsequently deleted. Table 6.5 shows how file E may appear.

Table 6.5: Illustration of how data could be stored in File E

Generated facet	First edge		Second edge		Third edge	
	First vertex	Second vertex	First vertex	Second vertex	First vertex	Second vertex
First	V3	V2	V2	V5	V5	V3
Second	V2	V9	V9	V11	V11	V2
Third	V2	V5	V5	V9	V9	V2
Fourth	V5	V7	V7	V9	V9	V5

The above procedures work for both types of gaps whose boundaries consist either of odd or even number of edges. Figure 6.13 and Table 6.6 illustrate how the algorithm works for an *even* number of edges or vertices in file D.

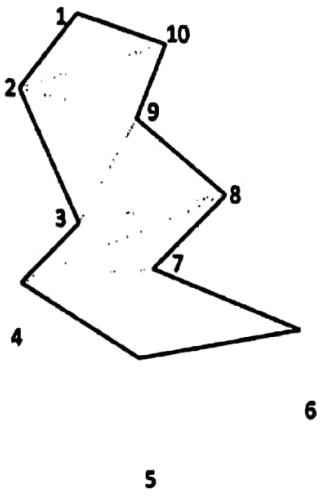


Figure 6.13: Gaps with even number of edges

Table 6.6: Process of facet generation for gaps with even number of edges

Facets	Vertices									
	V1	V2	V3	V4	V5	V6	V7	V8	V9	V10
F1	1	2								3
F2	E	1							2	3
F3	E	1	2						3	E

F4	E	E	1					2	3	E
F5	E	E	1	2				3	E	E
F6	E	E	E	1			2	3	E	E
F7	E	E	E	1	2		3	E	E	E
F8	E	E	E	E	1	2	3	E	E	E

With reference to Table 6.6,

First facet generated:	Second facet generated:
Edge 1 → V1, V2	Edge 1 → V2, V9
Edge 2 → V2, V10	Edge 2 → V9, V10
Edge 3 → V10, V1	Edge 3 → V10, V1

and so on until the whole gap is covered. Similarly, Figure 6.14 and Table 6.7 illustrate how the algorithm works for an *odd* number of edges or vertices in file D.

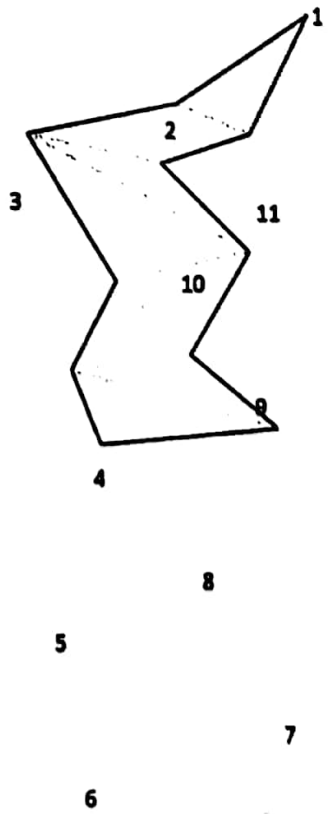


Figure 6.14: Gaps with odd number of edges

Table 6.7: Process of facet generation for gaps with odd number of edges

Facets	Vertices										
	V1	V2	V3	V4	V5	V6	V7	V8	V9	V10	V11
F1	1	2									3
F2	E	1								2	3
F3	E	1	2							3	E
F4	E	E	1						2	3	E
F5	E	E	1	2					3	E	E
F6	E	E	E	1				2	3	E	E
F7	E	E	E	1	2			3	E	E	E
F8	E	E	E	E	1		2	3	E	E	E
F9	E	E	E	E	1	2	3	E	E	E	E

The process of facet generation for ***odd*** vertices are also done in the same way as ***even*** vertices. The process of facet generation has the following pattern:

F1 → First and second vertices are combined with the last vertex. Once completed, eliminate first vertex. The remainder is ten vertices.

F2 → First vertex is combined with the last two vertices. Once completed, eliminate the last vertex. The remainder is nine vertices.

F3 → First and second vertices are combined with the last vertex. Once completed, eliminate first vertex. The remainder is eight vertices.

F4 → First vertex is combined with last two vertices. Once completed, eliminate the last vertex. The remainder is seven vertices.

This process is continued until all the gaps are patched.

Solving the "Wrong Orientation of Facets" Problem

In the case when the generated facet's orientation is wrong, the

algorithm should be able to detect it and corrective action can be taken to rectify this error. Figure 6.15 shows how a generated facet with a wrong orientation can be corrected.

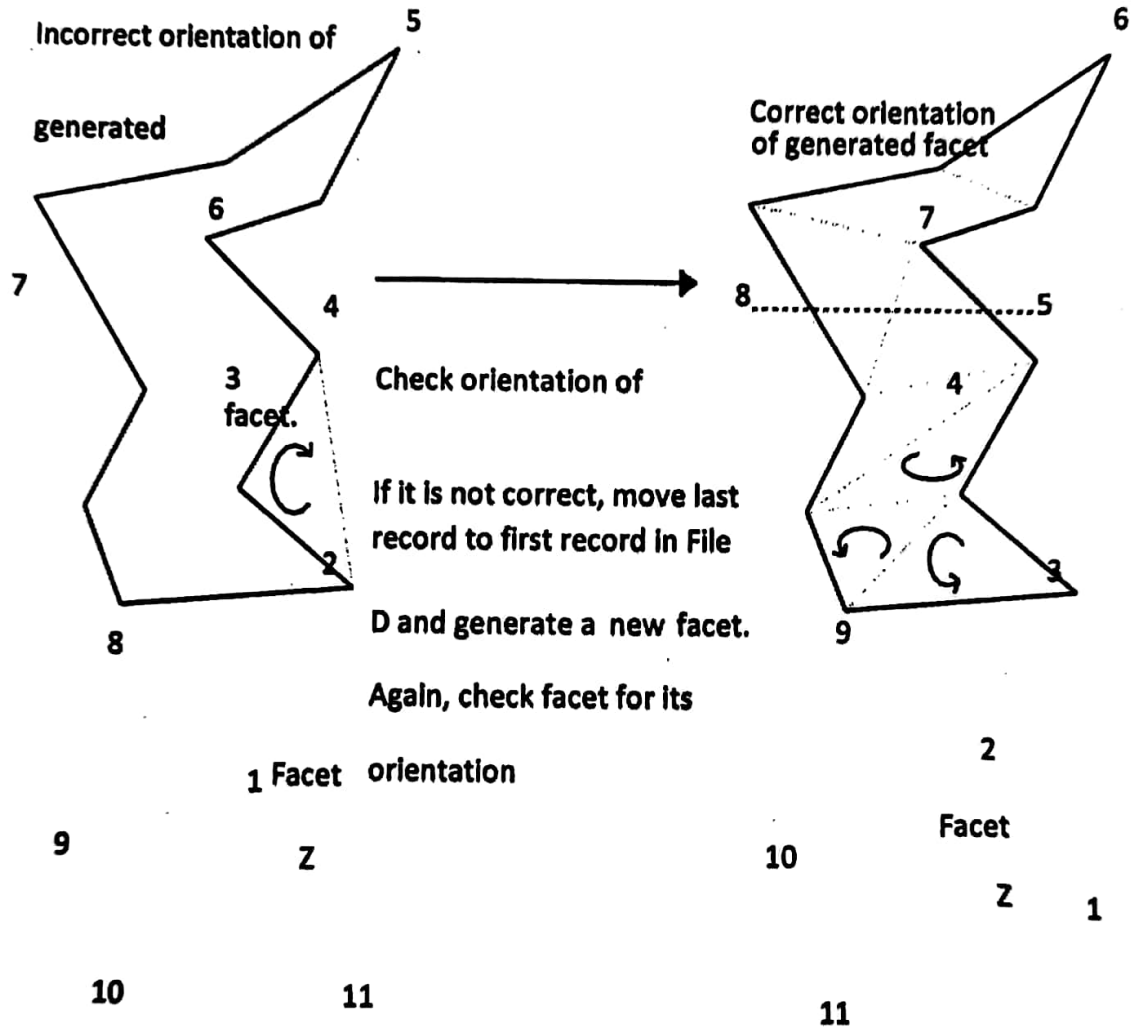


Figure 6.15: Incorrectly generated facet's orientation and its repair

It can be seen that facet Z (vertices 1, 2, 11) is oriented in a clockwise direction and this contradicts the right-hand rule adopted by the STL format. Thus, this is not acceptable and needs corrections.

This can be done by shifting the last record in file D of Table 6.8 to the position of the first edge in file D of Table 6.9. All the edges, including the initial first one will be shifted one position to the right (assuming that the records are stored in the left to right structure). Once this is done, step 4 of facet generation can be implemented.

Before the shift:

Table 6.8: Illustration showing how file D is manipulated to solve orientation problems

Vertex	Edge										
	First	Second	Third	Fourth	Fifth	Sixth	Seventh	Eighth	Ninth	Tenth	Eleventh
First	1	2	3	4	5	6	7	8	9	10	11
Second	2	3	4	5	6	7	8	9	10	11	1

After the shift:

Table 6.9: Illustration showing the result of the shift to correct the facet orientation

Vertex	Edge										
	First	Second	Third	Fourth	Fifth	Sixth	Seventh	Eighth	Ninth	Tenth	Eleventh
First	11	1	2	3	4	5	6	7	8	9	10
Second	1	2	3	4	5	6	7	8	9	10	11

As can be seen from the above example, vertices 1 and 2 are used initially as the first edge to form a facet. However, this resulted in a facet having a clockwise direction. After the shift, vertices 11 and 1 are used as the first edge to form a facet.

Facet Z, as shown on the right-hand-side of Figure 6.15, is again generated (vertices 1, 2, 11) and checked for its orientation. When its orientation is correct (i.e., in the anti-clockwise direction), it is saved and stored in temporary file E.

All subsequent facets are then generated and checked for its orientation. If any of its subsequently generated facets has an incorrect orientation, the whole process would be restarted using the initial temporary file D. If all the facets are in the right orientation, it will then

be written to the original file A.

Comparison with an Existing Algorithm that Performs Facet Generation

An illustration of an existing algorithm that might cause a very narrow facet (shaded) to be generated is shown in Figure 6.16.

This results from using an algorithm that uses the smallest angle to generate a facet. In essence, the problem is caused by the algorithm's search for a time-local rather than a global-optimum solution [10]. Also, calculation of the smallest angle in 3D space is very difficult.

Figure 6.16 is similar to Figure 6.14. However, in this case, the facet generated (shaded) can be very narrow. In comparing the algorithms, the result obtained would match, if not, exceed the algorithm that uses the smallest angle to generate a facet.

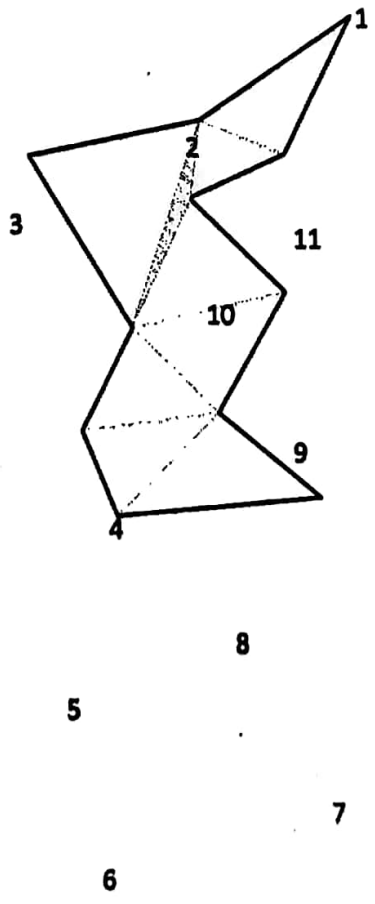


Figure 6.16: Generation of facets using an algorithm that uses the smallest angle between edges

Special Algorithms

The generic solution presented could only cater to gaps (whether simple or complex) that are isolated from one another. However, should any of the gaps were to meet at a common vertex, the algorithm may not be able to work properly. In this section, the algorithm would be expanded to include solving some of these special cases. These special cases include:

(1) Two or more gaps formed from a coincidental vertex.
(2) Degenerate facets.
(3) Overlapping facets.

The special cases are classified as such because these errors are not commonly encountered in the tessellated model. Hence it is not advisable to include this expanded algorithm in the generic solution as it can be very time consuming to apply in a during normal search. However, if there are still problems in the tessellated model after the generic solution's repair, the expanded algorithm could then be used to detect and solve the special case problems.

Special Case 1: Two or More Gaps are Formed from a Coincidental Vertex

The first special case deals with problems where two or more gaps are formed from a coincidental vertex. Appropriate modifications to the general solution may be made according to the solutions discussed as follows.

As can be seen from Figure 6.17, there exists two gaps that are connected to vertex 1. The algorithm given in Paper I would have a problem identifying which vertex to go to when the search reaches vertex 1 (either vertex 2 in gap 1 or vertex 5 in gap 2).

Table 6.10 illustrates what file C would look like, given the two gaps.

When the search starts to find all the edges that would form a closed loop, the previous algorithm might mistakenly connect edges: 3-4; 4-1; and *1-5*; instead of *1-2*. This is clearly an error as the edge that is

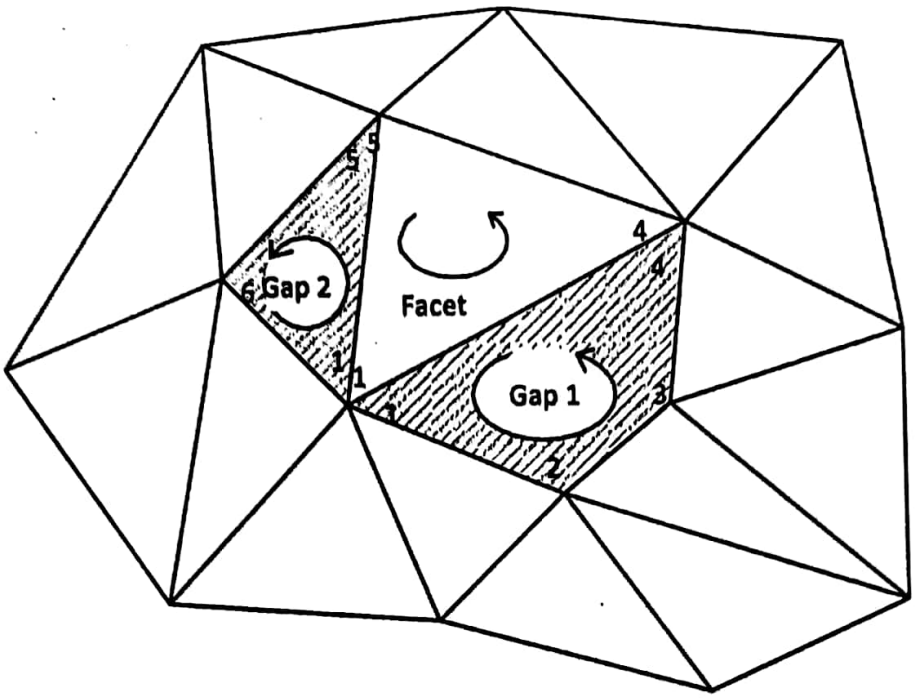


Figure 6.17: Two gaps sharing one coincidental vertex

Table 6.10: File C containing "Erroneous" edges that would form the boundaries of gaps

File C :

Vertex	Edge						
	First	Second	Third	Fourth	Fifth	Sixth	Seventh
First	V3	V5	V4	V7	V6	V2	V7
Second	V4	V6	V1	V5	V1	V3	V2

supposed to be included in file D should be *1-2, and not 1-5*. It is therefore pertinent that for every edge searched, the second vertex (e.g., V1 of the third edge, shaded, in Table 6.10) of that edge should be searched against the first vertex of subsequent edges (e.g., V1 of edge 1-5 in the fourth edge) and this should not be halted the first time the first vertex of subsequent edges are found. The search should continue to check if there are other edges with V1 (e.g., edge 1-2, in the seventh edge). Every time, say for example, vertex 1 is found, there should be a count. When the count is more than two, that would indicate that there

is more than one gap sharing the same vertex; this may be called the coincidental vertex. Once this happens, the following procedure would then be used.

(1) Step 1: To Conduct Normal Search (for the Boundary of Gap 1)

At the start of the normal search, the first edge of file C, vertices 3 and 4, as seen as in Table 6.11(a), is saved into a temporary file C1.

Table 6.11(a): Representation of how files C and C1 would look like

File C:

Vertex	Edge						
	First	Second	Third	Fourth	Fifth	Sixth	Seventh
First	V3	V5	V4	V1	V6	V2	V1
Second	V4	V6	V1	V5	V1	V3	V2

File C1:

Vertex	Edge		
	First	Second	Third
First	V3	?	?
Second	V4	?	?

The second vertex in the first edge (V4) of file C1, is searched against the first vertex of subsequent edge in file C (refer to Table 6.11(a), shaded box). Once it is found to be the same vertex (V4) and that there are no other edges sharing the same vertex, the edge (i.e., vertex 4-1) is stored as the second edge in file C1 (refer to Table 6.11(b)).

(2) Step 2: To Detect More Than One Gap

The second vertex of the second edge (V1) in file C1 is searched for an equivalent first vertex of subsequent edges in file C (refer to Table 6.11(b), shaded box containing vertex). Once it is found (V1, first vertex of fourth edge), a count of one is registered and at the same time, that edge is noted.

Table 6.11(b): Representation on how files C and C1 would look like during the normal search (special case)

Count 1 ⇓ Count 2 ⇓

File C:

Vertex	Edge						
	First	Second	Third	Fourth	Fifth	Sixth	Seventh
First	V3	V5	V4	V1	V6	V2	V1
Second	V4	V6	V1	V5	V1	V3	V2

File C1:

Vertex	Edge		
	First	Second	Third
First	V3	V4	?
Second	V4	V1	?

The search for the same vertex is continued to determine if there are other edges sharing the same vertex 1. If there is an additional edge sharing the same vertex 1 (Table 6.11(b), seventh edge), another count is registered, making a total of two counts. Similarly, the particular record in which it happens again is noted. The search is continued until there are no further edges sharing the same vertex. When completed, the following is carried out for the third edge in file C1.

For the first count, reading from file C, all the edges that would form the first alternative closed loop are sorted. These first alternatives in file C2 (refer to Table 6.11(c) and Figure 6.18(a) for a graphical representation of how the first alternatives might look like) are then saved.

A closed loop is established once the second vertex of the last edge (V3) is the same as the first vertex of the first edge (V3). ***For the second count***, the edges are sorted to form a second alternative of a closed loop that will represent the boundary of the gap. These edges are then saved in another temporary file C3 (refer to Table 6.11(d) and Figure 6.18(b) for a graphical representation of how the second

Table 6.11(c): First alternative closed loop that may represent the boundary of the gap

File C2:

Vertex	Edge						
	First	Second	Third	Fourth	Fifth	Sixth	Seventh
First	V3	V4	V1	V5	V6	V1	V2
Second	V4	V1	V5	V6	V1	V2	V3

Table 6.11(d): Second alternative closed loop that may represent the boundary of the gap

File C3:

Vertex	Edge			
	First	Second	Third	Fourth
First	V3	V4	V1	V2
Second	V4	V1	V2	V3

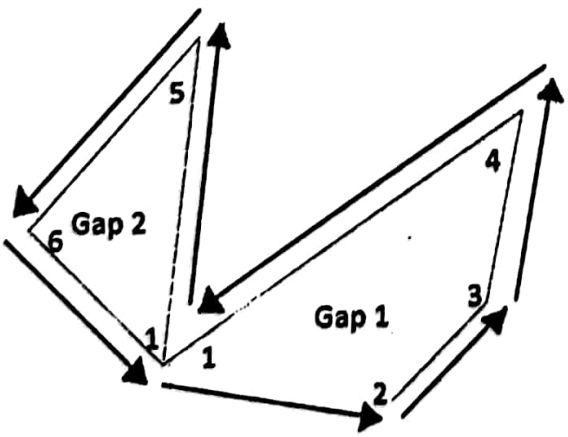


Figure 6.18(a): Graphical representation of the two gaps sharing a coincidental vertex (first alternative)

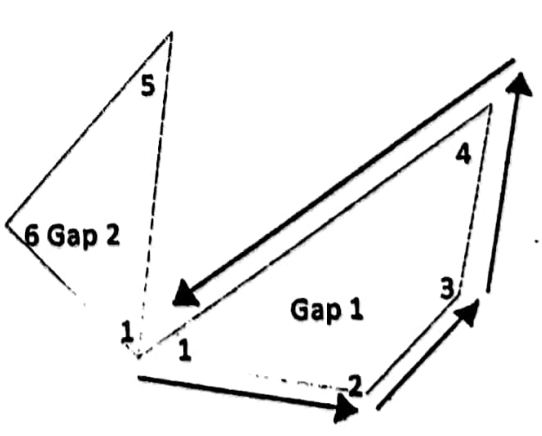


Figure 6.18(b): Graphical representation of the two gaps sharing the same vertex (second alternative)

alternatives may look like). Once a closed loop is established, the search is stopped.

(3) Step 3: To Compare Which Alternative Has the Least Record to Form the Boundary of the Gap

From file C2 (first alternative) and file C3 (second alternative), it can be seen that the second alternative has the least record to form the boundary of Gap 1. Hence the second alternative data would be written to file D for the next stage of facet generation and the two temporary files C2 and C3 would be discarded.

Once gap 1 is repaired, gap 2 can be repaired by using the generic solution.

Further Ilustrations on How the Algorithm Works

The algorithm can cater to more than two gaps sharing the same vertex, as shown in Figure 6.19(a), or even three gaps arranged differently, as shown in Figure 6.19(b).

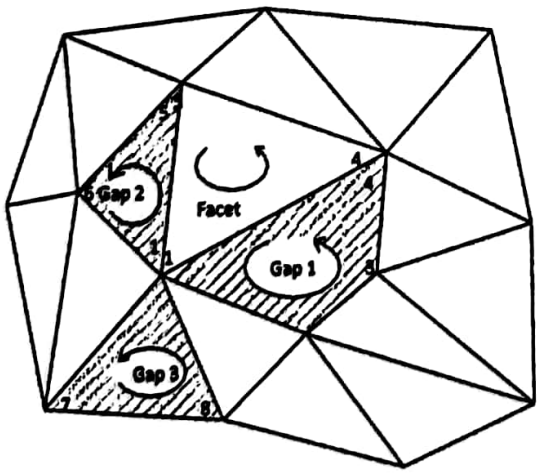


Figure 6.19(a): Three facets sharing one coincidental vertex

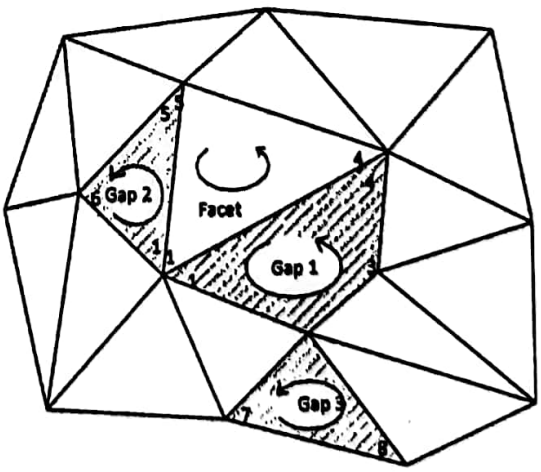


Figure 6.19(b): Three facets sharing two coincidental vertices

The solution is illustrated using the condition shown in Figure 6.20(b). Table 6.12(a) shows how file C may appear.

Table 6.12(a): Illustration on how file C may appear given three gaps sharing the same two vertices

File C:

	Count 1 for V1 ⇓				Count 1 for V2 ⇓	Count 2 for V1 ⇓	Count 2 for V2 ⇓		
	Edge								
Vertex	First	Second	Third	Fourth	Fifth	Sixth	Seventh	Eighth	Ninth	Tenth
First	V3	V5	V4	V1	V6	V2	V1	V2	V8	V7
Second	V4	V6	V1	V5	V1	V3	V2	V7	V2	V8

(1) Step 1: Normal Search

Table 6.12(b): File C1 during normal search

File C1:

	Edge			
Vertex	First	Second	Third	Fourth
First	V3	V4	?	?
Second	V4	V1	?	?

(2) Step 2: Detection

Referring to Table 6.12(a), it can be seen that:

For V1 → there are two counts
For V2 → there are two counts

<u>For Count 1 for V1 and Count 1 for V2</u>

Thus from file C (Table 6.12(a)) the first alternative closed loop can be generated and the file is shown in Table 6.12(c).

Table 6.12(c): First alternative of closed loop that may represent the boundary of a gap

	Edge									
Vertex	First	Second	Third	Fourth	Fifth	Sixth	Seventh	Eighth	Ninth	Tenth
First	V3	V4	V1	V5	V6	V1	V2	—	—	—
Second	V4	V1	V5	V6	V1	V2	V3	—	—	—

Figure 6.20(a) illustrates a graphical representation of the gap's boundary.

For Count 1 for V1 and Count 2 for V2

The second alternative of a closed loop can be sorted and is shown in Table 6.12(d).

Table 6.12(d): Second alternative of closed loop that may represent the boundary of a gap

	Edge									
Vertex	First	Second	Third	Fourth	Fifth	Sixth	Seventh	Eighth	Ninth	Tenth
First	V3	V4	V1	V5	V6	V1	V2	V7	V8	V2
Second	V4	V1	V5	V6	V1	V2	V7	V8	V2	V3

Figure 6.20(b) illustrates a graphical representation of the gap's boundary.

For Count 2 for V1 and Count 1 for V2

The third alternative of a closed loop can again be sorted and is shown in Table 6.12(e).

Table 6.12(e): Third alternative of closed loop that may represent the boundary of a gap

	Edge									
Vertex	First	Second	Third	Fourth	Fifth	Sixth	Seventh	Eighth	Ninth	Tenth
First	V3	V4	V1	V2	—	—	—	—	—	—
Second	V4	V1	V2	V3	—	—	—	—	—	—

Figure 6.20(c) illustrates a graphical representation of the gap's boundary.

For Count 2 for V1 and Count 2 for V2

The fourth alternative of a closed loop can also be sorted and is shown in Table 6.12(f).

Table 6.12(f): Fourth alternative of closed loop that may represent the boundary of a gap

Vertex	Edge									
	First	Second	Third	Fourth	Fifth	Sixth	Seventh	Eighth	Ninth	Tenth
First	V3	V4	V1	V2	V7	V8	V2	—	—	—
Second	V4	V1	V2	V7	V8	V2	V3	—	—	—

Figure 6.20(d) illustrates a graphical representation of the gap's boundary.

(3) Step 3: Comparison of the Four Alternatives

As can be seen from the four alternatives (see Figure 6.20), the third alternative, as shown in Figure 6.20(c), is considered the best solution and the correct solution to fill gap 1. This correct solution can be found by comparing which alternative uses the least edges to fill gap 1 up. Once the solution is found, the edges would be saved to file D for the next stage, that of facet generation.

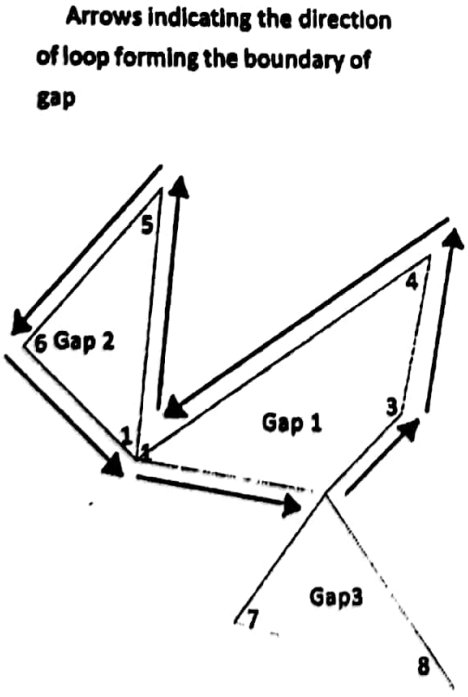


Figure 6.20(a): First alternative — graphical representation of the three gaps sharing two coincident vertices

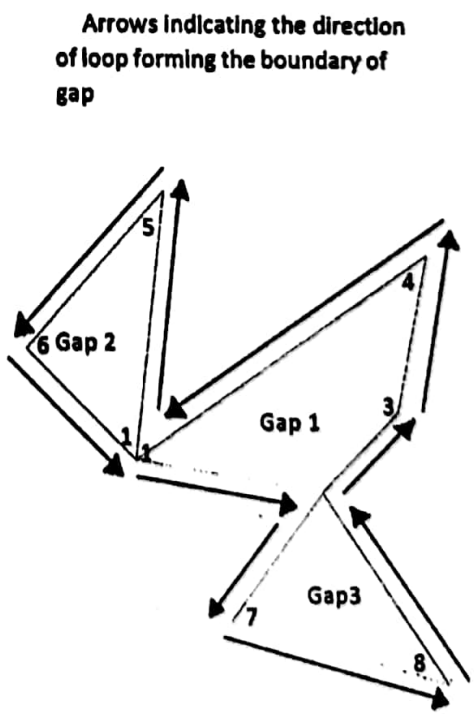


Figure 6.20(b): Second alternative — graphical representation of the three gaps sharing two coincident vertices

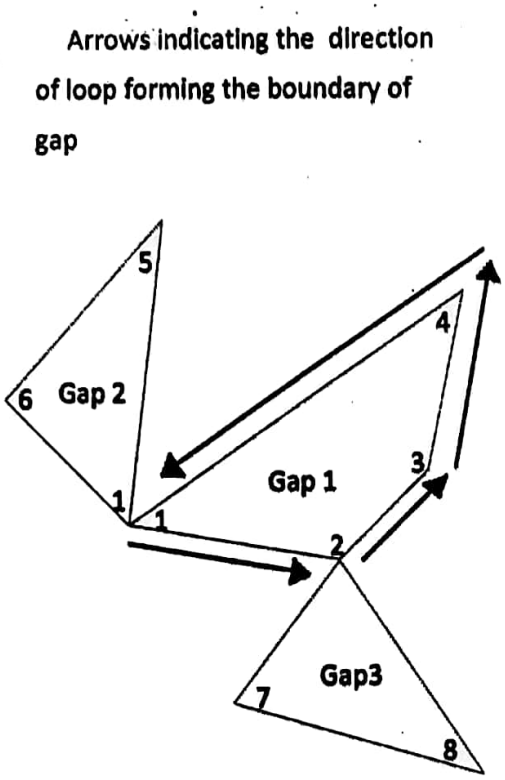


Figure 6.20(c): Third alternative — graphical representation of the three gaps sharing two coincident vertices

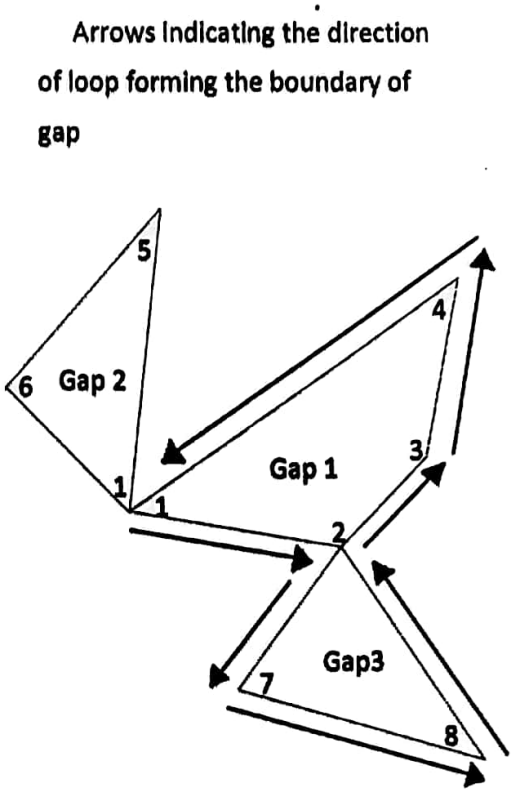


Figure 6.20(d): Fourth alternative — graphical representation of the three gaps sharing two coincident vertices

After gap 1 is filled, gaps 2 and 3 can then be repaired using the basic generic solution.

Special Case 2: Two Facets Sharing a Common Edge

When a degenerate facet such as the one shown in Figure 6.21 is encountered, vector algebra is applied. The following steps are taken:

(1) Edge a-c is converted into vectors,

$$(c_1\mathbf{i} + c_2\mathbf{j} + c_3\mathbf{k}) - (a_1\mathbf{i} + a_2\mathbf{j} + a_3\mathbf{k})$$

(2) (i) Collinear vectors are checked. Let

$$\mathbf{x} = ac \Rightarrow x_1\mathbf{i} + x_2\mathbf{j} + x_3\mathbf{k}; \quad (6.1)$$

$$\mathbf{y} = ce \Rightarrow y_1\mathbf{i} + y_2\mathbf{j} + y_3\mathbf{k};$$

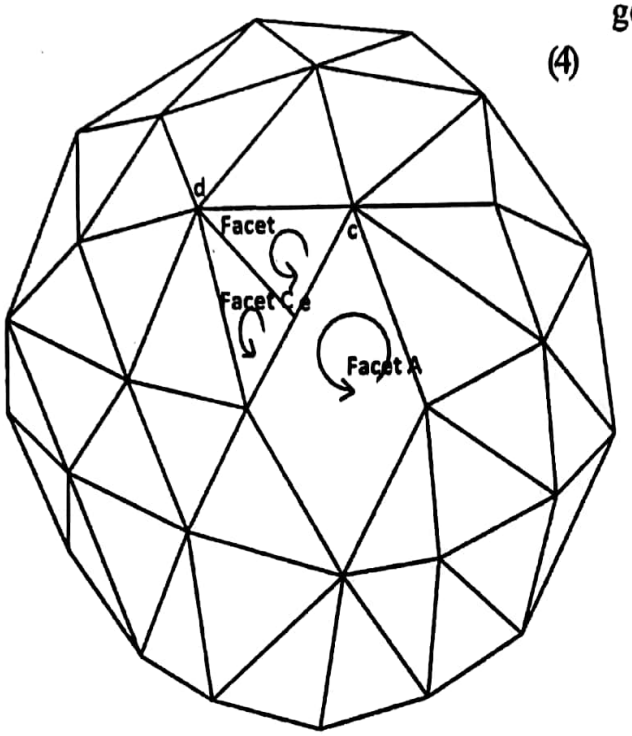


Figure 6.21: An illustration showing a degenerate facet

(ii) By mathematical definition, the two vectors x and y are said to be collinear if there exists scalars s and t, both non-zero, such that:

$$sx + ty = 0. \quad (6.4)$$

But when applied in a computer, it is only necessary to have:

$$sx + ty \leq \varepsilon, \quad (6.5)$$

where ε is a definable tolerance.

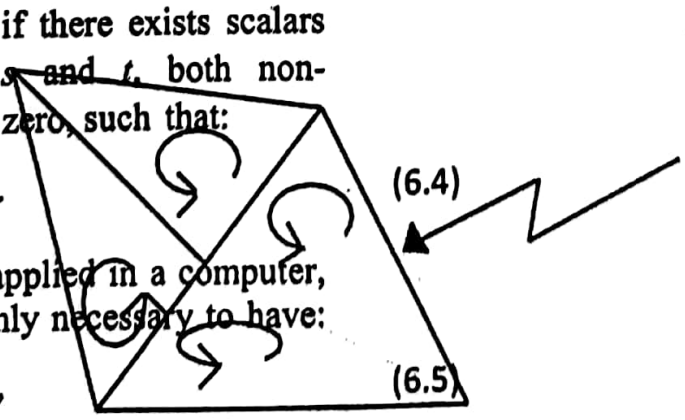
(3) If the two vectors are found to be collinear vectors, the position of vertex *e* is generated.

(4) (i) Facet A is split into two facets (see Figure 6.22). The two facets are generated using the three vertices in facet A.
First facet vertices are: *a*, *b*, *e*
Second facet vertices are: *b*, *c*, *e*

(ii) The orientation is checked.

(iii) New facets are stored in a temporary file.

(5) Search and delete facet A from original file.



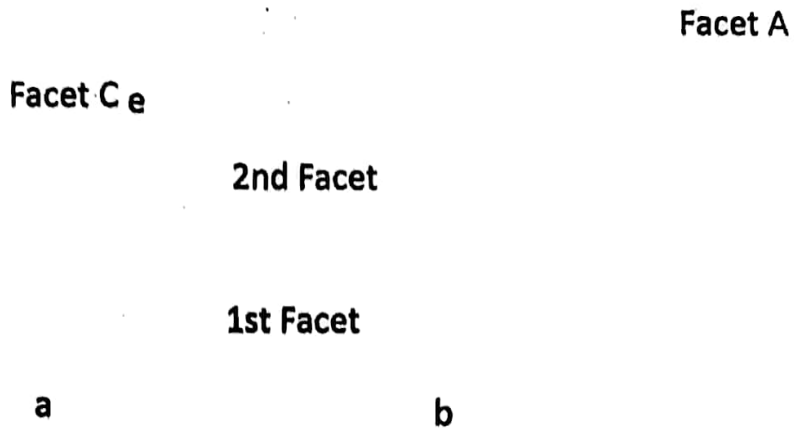


Figure 6.22: Illustration on how a degenerate facet is solved

(6) (i) The two new facets are stored in the original file.
(ii) Data are deleted from the temporary file.

Special Case 3: Overlapping Facets

The condition of overlapping facets can be caused by errors introduced by inconsistent numerical round-off. This problem can be resolved through vertex merging where vertices within a predetermined numerical round-off tolerance of one another can be merged into just one vertex. Figure 6.23 illustrates one example of how this solution can be applied. Figure 6.24 illustrates another example of an overlapping facet.

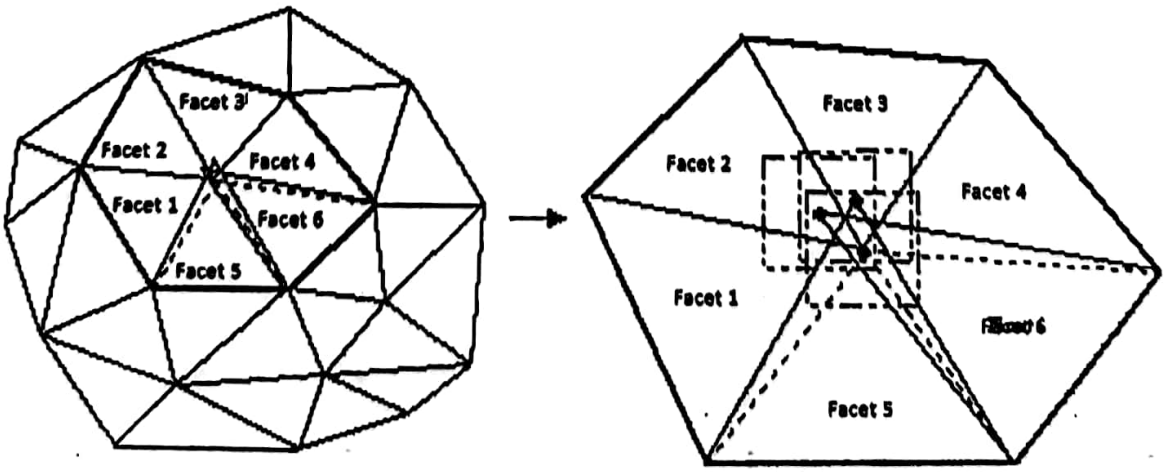


Figure 6.23(a): Overlapping facets

Figure 6.23(b): Numerical round-off equivalence region

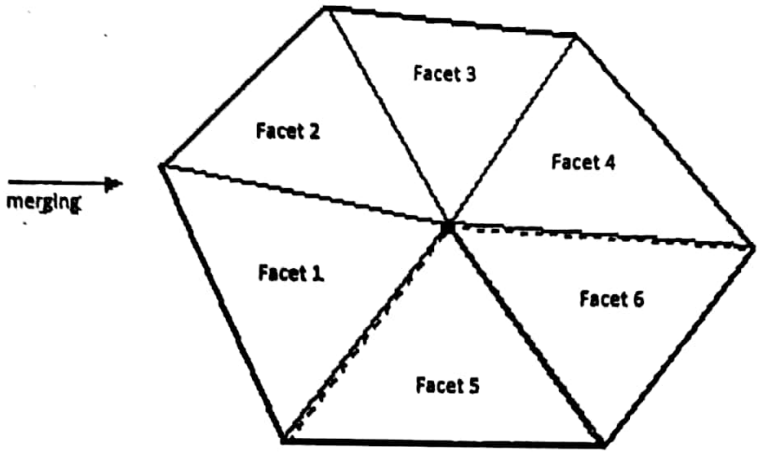


Figure 6.23(c): Vertices merged

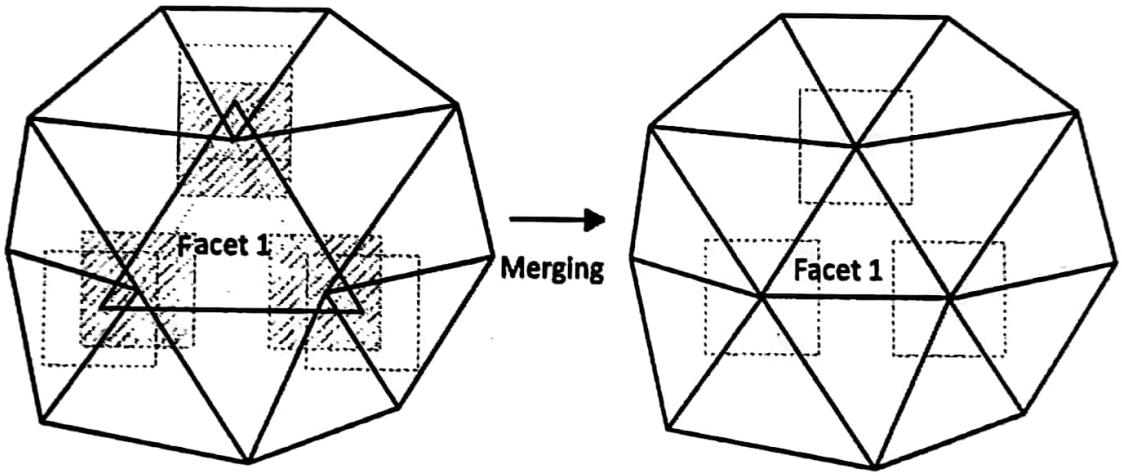


Figure 6.24(a): An overlapping facet

Figure 6.24(b): Facet's vertices merged with vertices of neighboring facets

It is recommended that this merging of vertices be done before the searching of the model for gaps. This will eliminate unnecessary detection of erroneous edges and save substantial computational time expended in checking whether the edges can be used to generate another facet.

Performance Evaluation

Computational efficiency is an issue whenever CAD-model repair of

solids that have been finely tessellated is considered. This is due to the fact that for every unit increase in the number of facets (finer tessellation), the additional increase in the number of edges is three. Thus, the computational time required for the checking of erroneous edges would correspondingly increase.

Efficiency of the Detection Routine

Assuming that there are twelve triangles in the cube (Figure 6.25), the number of edges = 12 × 3 = 36. The number of searches is computed as follows:

(1) Read first edge, search 35 and remove two edges.
(2) Read second edge, search 33 and remove two edges.
(3) Read third edge, search 31 and remove two edges.
(4) Number of searches = $1 + 3 + \cdots + 35$
In general, the number of searches = $1 + \cdots + (n-1)$

$$= n^2/4 \qquad (6.6)$$

where n = number of edges.

Although this result does not seem satisfactory, it is both an optimum and a robust solution that can be obtained given the inherent nature of the STL format (such as its lack of topological information). However, if topological information is available, the efficiency of the routine used to detect the erroneous edges can definitely be increased significantly. Some additional points worth noting are: First, binary files are far more efficient than ASCII files because they are only 20–25% as large and thus reduce the amount of physical data that needs to be transferred and because they do not require subsequent

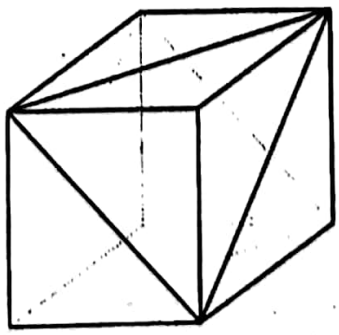


Figure 6.25: A cube tessellated into 12 triangles

translation into a binary representation. Consequently, one easily saves several minutes per file by using binary instead of ASCII file formats [10].

As for vertex merging, the use of one-dimensional AVL-tree can significantly reduce the search-time for sufficiently identical vertices [10]. The AVL-trees, which are usually twelve to sixteen levels deep, reduces each search from $0(n)$ to $0(\log n)$ complexity, and the total search-time from close to an hour to less than a second.

Estimated Computational Time for Shell Closure

The computational time required for shell closure is relatively fast. Estimated time can range from a few seconds to less than a minute and is arrived at based on the processing time obtained by Jan Helge Bohn [10] that uses a similar shell closure algorithm.

Limitations of Current Shell Closure Process

The shell closure process developed thus far does not have the ability to detect or solve the problems posed by any of the non-manifold conditions. However, the detection of non-manifold conditions and their subsequent solutions would be the next focus of the ongoing research. A limitation of the algorithm involves the solving of co-planar (see Figure 6.26(a)) and non co-planar facets (see Figure 6.26(b)) whose intersections result in another facet. The reasons for such errors are related to the application that generated the faceted model, the application that generated the original 3D CAD model, and the user.

Another limitation involves the incorrect triangulation of parametric surface (see Figure 6.27). One of the overlapping triangles, T_b = BCD should not be present and should thus be removed while the other triangle, T_a = ABC should be split into two triangles so as to maintain the correct contoured surface. The proposed algorithm is presently unable to solve this problem [11].

Finally, as mentioned earlier, the efficiency of $n^2/4$ (when n is the number of edges) is a major limitation especially when the number of

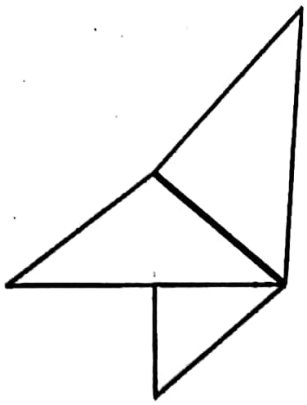


Figure 6.26(a): Incorrect triangulation (co-planar facet)

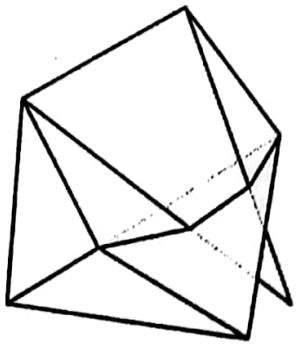


Figure 6.26(b): Non co-planar whereby facets are split after being intersected

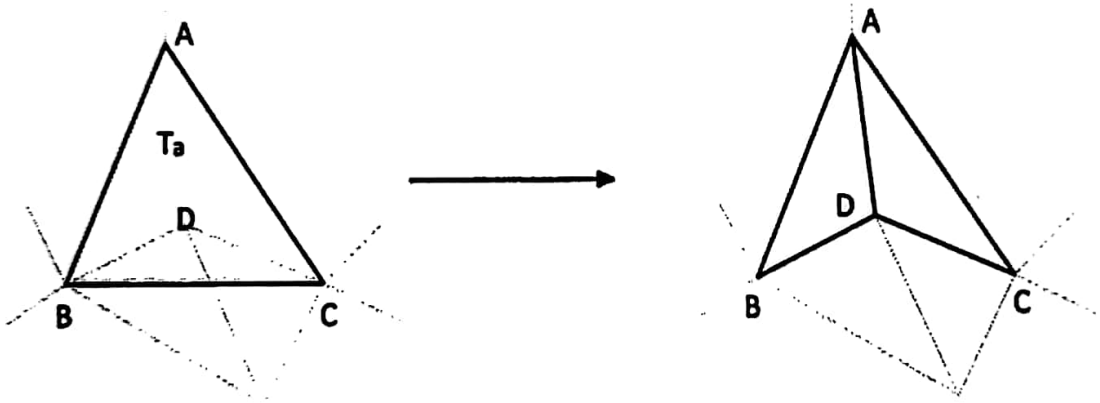


Figure 6.27: Incorrect triangulation of parametric surface

facets in the tessellated model becomes very large (e.g., greater than 40 000). There are differences in the optimum use of computational resources. However, further work is being carried out to ease this problem by using available topological information which are available in the original CAD model.

OTHER TRANSLATORS

IGES File

IGES (Initial Graphics Exchange Specification) is a standard used to exchange graphics information between commercial CAD systems. It

was set up as an American National Standard in 1981 [12, 13]. The IGES file can precisely represent CAD models. It includes not only the geometry information (Parameter Data Section) but also topological information (Directory Entry Section). In the IGES, surface modeling, constructive solid geometry (CSG) and boundary representation (B-rep) are introduced. Especially, the ways of representing the regularized operations for union, intersection, and difference have also been defined.

The advantages of the IGES standard are its wide adoption and comprehensive coverage. Since IGES was set up as American National Standard, virtually every commercial CAD/CAM system has adopted IGES implementations. Furthermore, it provides the entities of points, lines, arcs, splines, NURBS surfaces and solid elements. Therefore, it can precisely represent CAD model.

However, several disadvantages of the IGES standard in relation to its use as a RP format include the following objections:

(1) Because IGES is the standard format to exchange data between CAD systems, it also includes much redundant information that is not needed for rapid prototyping systems.
(2) The algorithms for slicing an IGES file are more complex than the algorithms slicing a STL file.
(3) The support structures needed in RP systems such as the SLA cannot be created according to the IGES format.

IGES is a generally used data transfer medium which interfaces with various CAD systems. It can precisely represent a CAD model. Advantages of using IGES over current approximate methods include precise geometry representations, few data conversions, smaller data files and simpler control strategies. However, the problems are the lack of transfer standards for a variety of CAD systems and system complexities.

HP/GL File

HP/GL (Hewlett-Packard Graphics Language) is a standard data format for graphic plotters [1, 2]. Data types are all two-dimensional, including

lines, circles, splines, texts, etc. The approach, as seen from a designer's point of view, would be to automate a slicing routine which generates a section slice, invoke the plotter routine to produce a plotter output file and then loop back to repeat the process.

The advantages of the HP/GL format are that a lot of commercial CAD systems have the interface to output the HP/GL format and it is a 2D geometry data format which does not need to be sliced.

However, there are two distinct disadvantages of the HP/GL format. First, because HP/GL is a 2D data format, the files would not be appended, potentially leaving hundreds of small files needing to be given logical names and then transferred. Second, all the support structures required must be generated in the CAD system and sliced in the same way.

CT Data

CT (Computerized Tomography) scan data is a particular approach for medical imaging [1, 14]. This is not standardized data. Formats are proprietary and somewhat unique from one CT scan machine to another. The scan generates data as a grid of three-dimensional points, where each point has a varying shade of gray indicating the density of the body tissue found at that particular point. Data from CT scans have been used to build skull, femur, knee, and other bone models on Stereolithography systems. Some of the reproductions were used to generate implants, which have been successfully installed in patients. The CT data consist essentially of raster images of the physical objects being imaged. It is used to produce models of human temporal bones.

There are three approaches to making models out of CT scan information: (1) Via CAD Systems (2) STL-interfacing and (3) Direct Interfacing. The main advantage of using CT data as an interface of rapid prototyping is that it is possible to produce structures of the human body by the rapid prototyping systems. But, disadvantages of CT data include firstly, the increased difficulty in dealing with image data as compared with STL data and secondly, the need for a special interpreter to process CT data.

NEWLY PROPOSED FORMATS

As seen above, the STL file — a collection of coordinate values of triangles — is not ideal and has inherent problems in this format. As a result, researchers including the inventor of STL, 3D Systems Inc., USA, have in recent years proposed several new formats and these are discussed in the following sections. However, none of these has been accepted yet as a replacement of STL. STL files are still widely used today.

SLC File

The SLC (StereoLithography Contour) file format is developed at 3D Systems, USA [15]. It addresses a number of problems associated with the STL format. An STL file is a triangular surface representation of a CAD model. Since the CAD data must be translated to this faceted representation, the surface of the STL file is only an approximation of the real surface of an object. The facets created by STL translation are sometimes noticeable on rapid prototyping parts (such as the AutoCAD Designer part). When the number of STL triangles is increased to produce smoother part surfaces, STL files become very large and the time required for a rapid prototyping system to calculate the slices can increase.

SLC attempts to solve these problems by taking two-dimensional slices directly from a CAD model instead of using an intermediate tessellated STL model. According to 3D Systems, these slices eliminate the facets associated with STL files because they approximate the contours of the actual geometry.

Three problems may arise from this new approach. Firstly, in slicing a CAD model, it is not always necessarily more accurate as the contours of each slice are still approximations of the geometry. Secondly, slicing in this manner requires much more complicated calculations (and therefore, is very time-consuming) when compared to the relatively straightforward STL files. Thirdly, a feature of a CAD model which falls between two slices, but is just under the tolerances set for inclusion on either of the adjacent slices, may simply disappear.

SLC File Specification

The SLC file format is a "21/2D" contour representation of a CAD model. It consists of successive cross-sections taken at ascending Z intervals in which solid material is represented by interior and exterior boundary polylines. SLC data can be generated from various sources, either by conversion from CAD solid or surface models, or more directly from systems which produce data arranged in layers, such as CT-scanners.

Definition of Terms

Segment

A segment is a straight line connecting two *X*/*Y* vertice points.

Polyline

A polyline is an ordered list of *X*/*Y* vertice points connected continuously by each successive line segment. The polyline must be closed whereby the last point must equal the first point in the vertice list.

Contour boundary

A boundary is a closed polyline representing interior or exterior solid material. An exterior boundary has its polyline list in counter-clockwise order. The solid material is inside the polyline. An interior boundary has its polyline list in clockwise order and solid material is outside the polyline. Figure 6.28 shows a description of the contour boundary.

Contour layer

A contour layer is a list of exterior and interior boundaries representing the solid material at a specified Z cross-section of the CAD model. The cross-section slice is taken parallel to the *X*/*Y* plane and has a specified layer thickness.

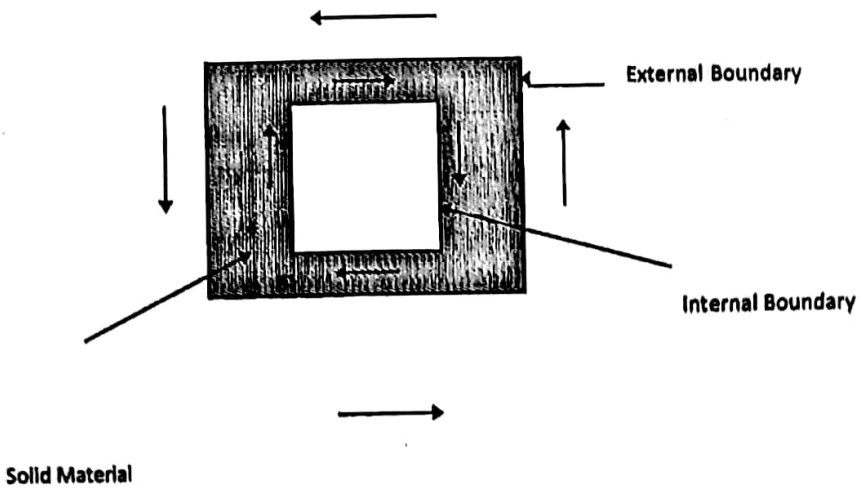


Figure 6.28: Contour boundary description

Data Formats

Byte	8 bits
Character	1 Byte
Unsigned Integer	4 Bytes
Float	4 Bytes IEEE Format

The most significant byte of FLOAT is specified in the highest addressed byte. The byte ordering follows the Intel PC Little Indian/Big Indian scheme.

Address	0	1	2	3
	Low Word		High Word	
	LSB	MSB	LSB	MSB

Most UNIX RISC Workstations are Big Indian/Little Indian, therefore they need to byte swap all Unsigned Integers and Floats before outputting to the SLC file.

Overview of the SLC File Structure

The SLC file is divided into a header section, a 3D reserved section, a sample table section, and the contour data section.

Header section

The Header section is an ASCII character string containing global information about the part and how it was prepared.

The header is terminated by a carrage return, line feed and control-Z character (0x0d,0x0a,0x1a) and can be a maximum of 2048 bytes including the termination characters.

The syntax of the header section is a keyword followed by its appropriate parameter.

Header keywords

- "-SLCVER <X.X>" specifies the SLC file format version number. The version number of this specification is 2.0.
- "-UNIT <INCH/MM>" indicates which units the SLC data is represented.
- "-TYPE <PART/SUPPORT/WEB>" specifies the CAD model type. PART and SUPPORT must be closed contours. WEB types can be open polylines or line segments.
- "-PACKAGE <vendor specific>" identifies the vendor package and version number which produced the SLC file. A maximum of 32 bytes.
- "-EXTENTS <minx,maxx miny,maxy minz,maxz>" describes the *X*, *Y*, and *Z* extents of the CAD model.
- "-CHORDDEV <value>" specifies the cordal deviation, if used, to generate the SLC data.
- "-ARCRES <value in degrees>" specifies the arc resolution, if used, to generate the SLC data.
- "-SURFTOL <value>" specifies the surface tolerance, if used, to generate the SLC data.
- "-GAPTOL <value>" specifies the gap tolerance, if used, to generate the SLC data.
- "-MAXGAPFOUND <value>" specifies the maximum gap size found when generating the SLC data.
- "-EXTLWC <value>" specifies, if any, line width compensation has been applied to the SLC data by the CAD vendor.

3D reserved section

This 256 byte section is reserved for future use.

Sampling table section

The sample table describes the sampling thicknesses (layer thickness or slice thickness) of the part. There can be up to 256 entries in the table. Each entry descibes the Z start, the slice thickness, and what line width compensation is desired for that sampling range.

Sampling Table Size	1 Byte
Sampling Table Entry	4 Floats
Minimum Z Level	1 Float
Layer Thickness	1 Float
Line Width Compensation	1 Float
Reserved	1 Float

The first sampling table entry Z start value must be the very first Z contour layer. For example, if the cross-sections were produced with a single thickness of 0.006 inches and the first Z level of the part is 0.4 inches and a line width compensation value of 0.005 is desired, then the sampling table will look like the following:

Sample Table Size	1
Sample Table Entry	0.4 0.006 0.005 0.0

If for example, the part was sliced with two different layer thicknesses, the sample table could look like the following:

Sample Table Size	2
Sample Table Entry 1	0.4 0.005 0.004 0.0
Sample Table Entry 2	2.0 0.010 0.005 0.0

Slice thicknesses must be even multiples of one other to avoid processing problems.

Contour data section

The contour data section is a series of successive ascending Z cross-sections or layers with the accompaning contour data. Each contour layer contains the minimum Z layer value, number of

boundaries followed by the list of individual boundary data. The boundary data contains the number of x, y vertices for that boundary, the number of gaps, and finally the list of floating point vertice points.

The location of a gap can be determined when a vertice point repeats itself.

To illustrate, given the contour layer in Section 2.4 the contour section could be as follows:

Z Layer	0.4
Number of Boundaries	2
Number of Vertices for the 1st Boundary	5
Number of Gaps for the 1st Boundary	0
Vertex List for 1st Boundary	0.0, 0.0
	1.0, 0.0
	1.0, 1.0
	0.0, 1.0
	0.0, 0.0

*Notice the direction of the vertice list is counter-clockwise indicating that the solid material is inside the polylist. Also, notice that the polylist is closed because the last vertice is equal to the first vertice.

Number of Vertices for the 2nd Boundary	5
Number of Gaps for the 2nd Boundary	0
Vertex List for 2nd Boundary	0.2, 0.2
	0.2, 0.8
	0.8, 0.8
	0.8, 0.2
	0.2, 0.2

*Notice the direction of the vertice list is clockwise indicating the solid material is outside the polylist. Also, notice that the polylist is closed because the last vertice is equal to the first vertice.

The contour layers are stacked in ascending order until the top of the part. The last layer or the top of the part is indicated by the Z level and a termination unsigned integer (0xFFFFFFFF).

Contour Layer Section Description

Contour Layer	
Minimum Z Level	Float
Number of Boundaries	Unsigned Integer
Number of Vertices	Unsigned Integer
Number of Gaps	Unsigned Integer
Vertices List (*X/Y*)	Number of Vertices * 2 Float
Repeat Number of Boundaries	1
Repeat Contour Layer until Top of Part	
Top of Part	
Maximum Z Level	1 Float
Termination Value	Unsigned Integer (0xFFFFFFFF)

Minimum Z Level for a Given Contour Layer

A one inch cube based at the origin 0, 0, 0 can be represented by only one contour layer and the Top of Part Layer data.

Suppose the cube was to be imaged in 0.010 layers. The sample table would have a single entry with its starting Z level at 0.0 and layer thickness at 0.01. The contour layer data section could be as follows:

Z Layer	0.0
Number of Boundaries	1
Number of Vertices for the 1st Boundary	5
Number of Gaps for the 1st Boundary	0
Vertex List for 1st Boundary	0.0, 0.0
	1.0, 0.0
	1.0, 1.0
	0.0, 1.0
	0.0, 0.0
Z Layer	1.0
Termination Value	0xFFFFFFFF

Notice, only one contour was necessary to describe the entire part. The initial contour will be imaged until the next minimum contour layer or the top of the part at the specified layer thickness described in the

sampling table. Now, this part could have 100 identical contour layers, but that would have been redundant. This is why the contour Z value is referred to as the minimum Z value. It gets repeated until the next contour or top of the part.

CLI File

The CLI (Common Layer Interface) format is developed in a Brite Euram project [2, 16] with the support of major European car manufacturers. The CLI format is meant as a vendor-independent format for layer by layer manufacturing technologies. In this format, a part is built by a succession of layer descriptions. The CLI file can be in binary or ASCII format. The geometry part of the file is organized in layers in the ascending order. Every layer is started by a layer command, giving the height of the layer.

The layers consist of series of geometric commands. The CLI format has two kinds of entities. One is the polyline. The polylines are closed, which means that they have a unique sense, either clockwise or anti-clockwise. This directional sense is used in the CLI format to state whether a polyline is on the outside of the part or surrounding a hole in the part. Counter-clockwise polylines surround the part, whereas clockwise polylines surround holes. This allows correct directions for beam offset.

The other is the hatching to distinguish between the inside and outside of the part. As this information is already present in the direction of polyline, and hatching takes up considerable file space, hatches have not been included into output files.

The advantages of the CLI format are given as follows:

(1) Since the CLI format only supports polyline entities, it is a simpler format compared to the HP/GL format.
(2) The slicing step can be avoided in some applications.
(3) The error in the layer information is much easier to be correct than that in the 3D information. Automated recovery procedures can be used and if required, editing is also not difficult.

However, there exists several disadvantages of the CLI format. They are given as follows:

(1) The CLI format only has the capability of producing polylines of the outline of the slice.
(2) Although the real outline of the part is obtained, by reducing the curve to segments of straight lines; the advantage over the STL format is lost.

The CLI format also includes the layer information like the HP/GL format. But, the CLI format only has polyline entities, while HP/GL supports arcs and lines. The CLI format is simpler than the HP/GL format and has been used by several rapid prototyping systems. It is hoped that the CLI format will become an industrial standard such as STL.

RPI File

The RPI (Rapid Prototyping Interface) format is designed by the Rensselaer Design Research Center [4, 7], Rensselaer Polytechnic Institute. It can be derived from currently accepted STL format data. The RPI format is capable of representing facet solids, but it includes additional information about the facet topology. Topological information is maintained by representing each facet solid entity with indexed lists of vertices, edges, and faces. Instead of explicitly specifying the vertex coordinates for each facet, a facet can refer to them by index numbers. This contributes to the goal of overall redundant information reduction.

The format is developed in ASCII to facilitate cross-platform data exchange and debugging. A RPI format file is composed of the collection of entities, each of which internally defines the data it contains. Each entity conforms to the syntax defined by the syntax diagram shown in Figure 6.29. Each entity is composed of an entity name, a record count, a schema definition, schema termination symbol, and the corresponding data. The data is logically subdivided into records which are made up of fields. Each record corresponds to one variable type in the type definition.

The RPI format includes the following four advantages:

(1) Topological information is added to the RPI format. As the result, flexibility is achieved. It allows users to balance storage and processing costs.
(2) Redundancy in the STL is removed and the size of file is compacted.
(3) Format extensibility is made possible by interleaving the format schema with data as shown in Figure 6.29.
(4) Representation of CSG primitives is provided, as capabilities to represent multiple instances of both facet and CSG solids.

Two disadvantages of the RPI format are given as follows:

(1) An interpreter which processes a format as flexible and extensible as the RPI format, is more complex than that for the STL format.
(2) Surface patches suitable for solid approximation cannot be identified in the RPI format.

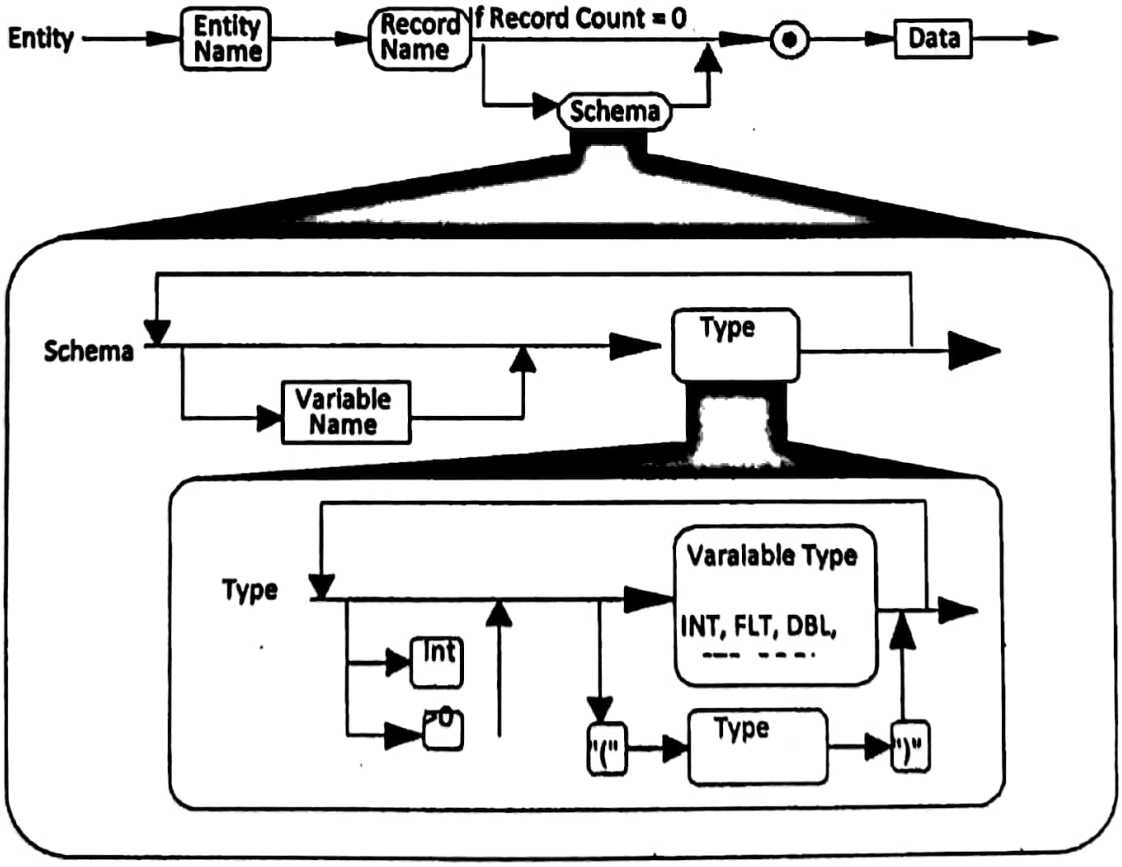


Figure 6.29: RPI format entity syntax diagram

The RPI format offers a number of features unavailable in the STL format. The format can represent CSG primitive models as well as facet models. Both can be operated by the Boolean union, intersection, and difference operators. Provisions for solid translation and multiple instancing are also provided. Process parameters, such as process types, scan methods, materials, and even machine operator instructions, can be included in the file. Facet models are more efficiently represented as redundancy is reduced. The flexible format definition allows storage and processing cost to be balanced.

LEAF File

The LEAF or Layer Exchange ASCII Format, is generated by Helsinki University of Technology [11]. To describe this data model, concepts from the object-oriented paradigm are borrowed. At the top level, there is an object called LMT-file (Layer Manufacture Technology file) that can contain parts which in turn are composed of other parts or by layers. Ultimately, layers are composed of 2D primitives and currently the only ones which are planned for implementation are polylines.

For example, an object of a given class is created. The object classes are organized in a simple tree shown in Figure 6.30. Attached to each object class is a collection of properties. A particular instance of an object specifies the values for each property. Objects inherit properties from their parents. In LEAF, the geometry of an object is simply one among several other properties.

In this example, the **object** is a LMT-file. It contains exactly one child, the object **P1**. **P1** is the combination of two parts, one of which is the support structures and the other one is **P2**, again a combination of two others. The objects at leaves of the tree — **P3**, **P4** and **S** — must have been, evidently, sliced with the same z-values so that the required operations, in this case **or** and **binary-or**, can be performed and the layers of **P1** and **P2** constructed.

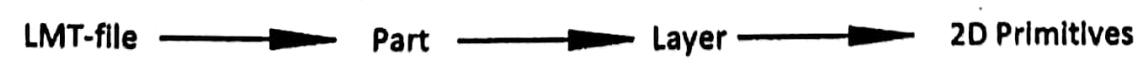


Figure 6.30: The object tree

```
(LMT-file
(name Object)(radix 85)(units 1mm)...
(Part (name P)...
(binary-or (Part (name S)(support-structure)(open)...
                (Layer...))
          (Part (name P2))...
                (or (Part (name P3)) ...
                    (Layer (name...)(polyline ...)))
                    (Part (name P4) ...
                    (Layer (name P4_L1) (ploYline ...)))
                )
          )
)))
```

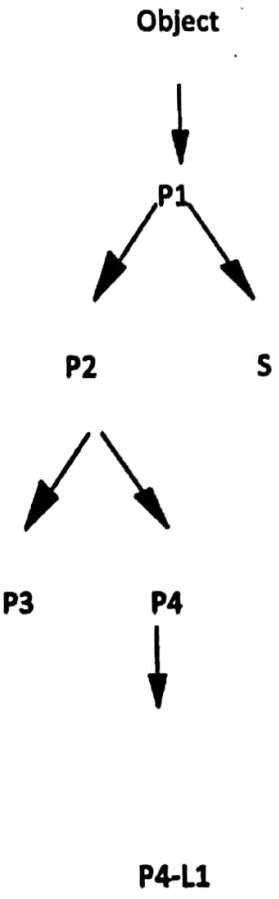


Figure 6.31: An Instancetree

In LEAF, the properties support-structure and open can also be attached to layer or even polyline objects allowing the sender to represent the original model and the support structures as one single part. In Figure 6.31, all parts inherit the properties of **object**, their ultimate parent. Likewise, all layers of the object S inherit the open property indicating that the contours in the layers are always interpreted as open, even if they are geometrically closed.

Amongst the many advantages of the LEAF format are:

(1) It is easy to implement and use.
(2) It is not ambiguous.
(3) It allows for data compression and for a human-readable representation.
(4) It is machine independent and LMT process independent.
(5) Slices of CSG models can be represented almost directly in LEAF.
(6) The part representing the support structures can be easily separated from the original part.

The disadvantages of the LEAF format include the following items:

(1) The new interpreter is needed for connecting the rapid prototyping systems.
(2) The structure of the format is more complicated than that of the STL format.
(3) The STL format cannot be changed into this format.

The LEAF format is described at several levels, mainly at a logical level using a data model based on object-oriented concepts, and at a physical level using a LISP-like syntax. At the physical level, the syntax rules are specified by several translation phases. Thus defined, it allows one to choose at which level, interaction with LEAF is desirable and at each level there is clear and easy-to-use interface. It is doubtful that LEAF currently supports the needs of all processes currently available but it is hoped it is a step forward in the direction.

STANDARD FOR REPRESENTING LAYERED MANUFACTURING OBJECTS

Currently, there is the ISO 10303, the international Standard for the Exchange of Product model data (STEP). This standard is intended for the computer-interpretable representation and exchange of product data for engineering purposes [17]. It comprises Generic Resources, Description Methods, Implementation Methods and Application Protocols (APs). However, at present there is no STEP AP that covers layered manufacturing.

The Rapid Prototyping and Layered Manufacturing (RPLM) group is proposing a New Work Item (NWI) for the development of an ISO 10303 AP for RPLM. This AP is to be a standard for product-related data and will support the RPLM process chain. It will comprise the design, process planning, RPLM process, post-processing and measurement of the product life-cycle. These components will require data representations for product geometry, product structure, configuration, tolerance information, product properties (e.g., appearance, color, identification of functional faces) and material data including the inhomogeneous and non-isotropic material distributions that RPLM processes can now generate [18].

Recently, there was a proposed standards-based approach for representing heterogeneous objects for layered manufacturing. The main

reason cited was that current solid modeling concentrates on the geometry and topology of the features, but does not include material gradation data [17]. Since layered manufacturing is able to fabricate an object with different materials and compositions, there is an urgent need for a standard to represent heterogeneous objects for layered manufacturing.

The following are several possible representations of the macro- structure of the material for modeling of heterogeneous objects [17]:

(1) tetrahedral decompositions,
(2) voxel-based representation,
(3) R-function method,
(4) r_m object model.

To represent heterogeneous objects in the ISO 10303 domain, a data planning model (DPM) is the required next step. A DPM shows the basic theories of the application domain and the general relationships among the main theories. After drawing up the DPM, the DPM must be validated. This was carried out by physically manufacturing the parts represented in the proposed STEP format. Representation of heterogeneous objects in a standardized format is a vital step in the successful physical realization of heterogeneous objects through layered manufacturing [17].

REFERENCES

[1] Jacobs, P.F., *Rapid Prototyping and Manufacturing*, Society of Manufacturing Engineers, 1992.

[2] Famieson, R. and Hacker, H., "Direct slicing of CAD models for rapid prototyping," *Rapid Prototyping Journal*, ISATA94, Aachen, Germany, 31 Oct to 4 Nov 94.

[3] Donahue, R.J., "CAD model and alternative methods of informa- tion transfer for rapid prototyping systems," *Proceedings of the Second International Conference on Rapid Prototyping*, 1991, pp. 217–235.

