

Phase II gbXML Geometry Validator Software Spec

Introduction

In the past, translation of geometry definitions from one CAD or BIM tool to energy modeling tools has meant creating an intermediate language definition of the geometry in a neutral format, like gbXML or ifcXML. There have been complications in this translation in the past. End users have complained that, instead of receiving a set of well-formed volumes that are easy to analyze, users have found that the translation from the BIM or CAD authoring tool to the neutral format is not successful: there may be surfaces missing, surfaces only partially defined, or there may be gaps or parts of the space missing, even sliver surfaces.

Placeholder for some images of translations gone wrong.



Many end-users have wanted a free method to check the quality of one of these intermediate XML file formats. The goal: to ensure that all the geometry that was present in the BIM or CAD authoring tool is still present after translation to an intermediate format like gbXML. That the geometry is complete is the primary requirement. Other requirements, which they may or not be aware of is necessary, are deeper checks to ensure the neutral format contains all the necessary information to be complete, relative to the XML schema. In the case of gbXML, the XML file should adhere to the requirements of the latest available XSD.

The goal of this validator is to provide this to end users, for free, in a web-based format that can serve users with the information that they need to make judgement calls as to the accuracy of the geometry after translation from the original authoring tool into gbXML. The validator identifies errors, and reports these errors back to the user, allowing the user the chance to correct these errors.

Requirements for the Validator

Enclosure Checking discussion....leading into what enclosure checking is

To help you understand the purpose of enclosure validation and how it might work, we can look at a simple example:

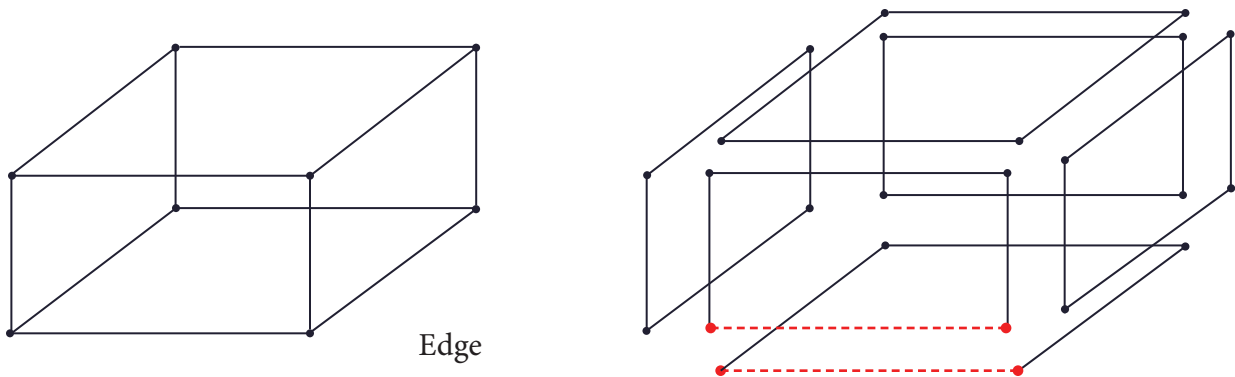
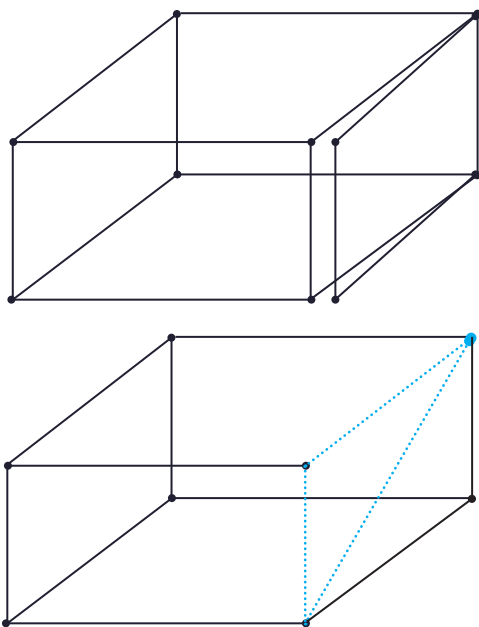


Figure 1 - Volumes are really a collection of surfaces

When we view an enclosed volume like the cube on the left in Figure 1 above, we see surfaces with nicely-aligned edges. If we could imagine that each surface is instead standalone, then we may see something more like the exploded view of the cube on the right of the figure above. This exploded view, is how gbXML stores its volume definition. The volume is made up of completely-defined surfaces, and each surface definition is made up of a PolyLoop that contains of a set of 3D cartesian coordinates (in axes x,y,z) that make up the definition of each surface. What this means, is that each edge is actually defined twice in a gbXML volume definition because each surface is totally defined as a standalone object in a volume definition. So in total, in gbXML, a cube would be defined by 24 edges instead of the 12 edges that would be counted in the volume on the left.

This underlying logic of gbXML's basic PolyLoop structure is the basis of the enclosure algorithm has been created, and the enabling factor that makes enclosure validation possible. The purpose of the validator is to ensure that PolyLoop coordinate definitions create a well-formed volume, with the edges of each surface definition properly aligned. The validator algorithm attempts to locate each edge of a volume, and find a "perfect match" for it in another PolyLoop definition. If it can successfully find a matching edge for all edges in a volume, the validator will report a message of assurance to any party using the algorithm. An example of two edges that are perfect matches are shown as red dotted lines in the exploded view on the right.

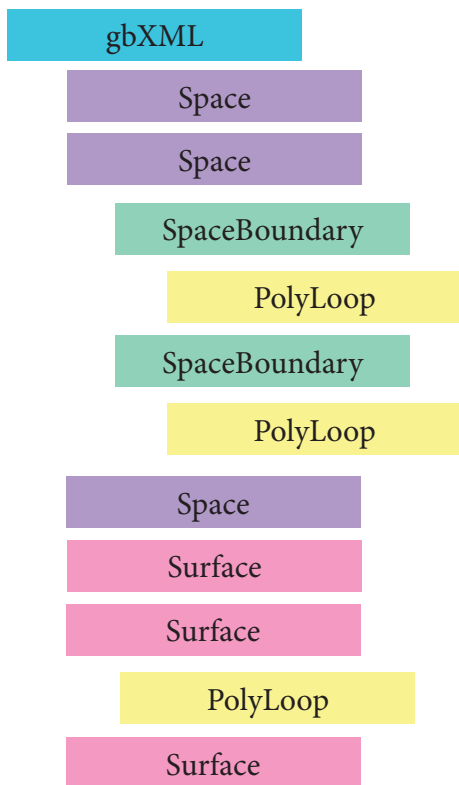


The validator enclosure-checking algorithm looks for situations like the ones in Figure 2 to the left. Where a surface may not be well-aligned with the other surfaces, or where the surface definitions may be incomplete. In the top example of Figure 2, the surface edges are out of alignment with the others, which results in gaps existing between surfaces.

In the bottom example, this surface is only described with three points instead of four, creating a situation with a single diagonal edge that has no relationship to all of the other edges. As well, two other edges also do not have any matches. All of these lonely edges, shown in blue, are an indication that the volume is not perfectly enclosed.

The validator should in these cases indicate that something is wrong with the volume definition.

Figure 2 - Examples of Improperly-formed Volumes



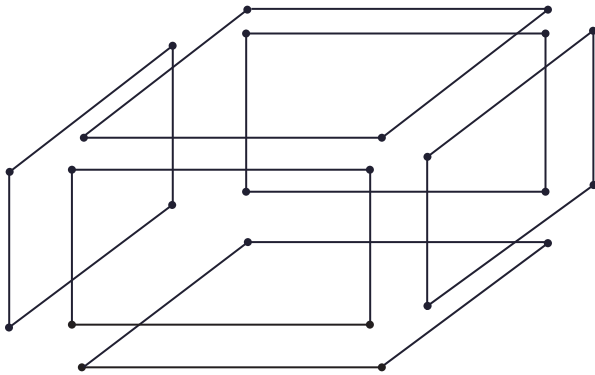
The enclosure-checking algorithm assumes that gbXML has already been formed into logical groupings. Unfortunately, the algorithm does not attempt to search the entire space of the gbXML file to uncover how each surface in the gbXML file may be related to another surface. Instead, it relies on the structure of gbXML to determine which surfaces have been logically grouped into individual, stand-alone volumes. It then groups these surfaces into a collection that *should* create a tightly-enclosed volume, and checks these against one another, in the algorithm described below.

Conceptually, it may be easier to think of this as a enclosure-validator for each individual space in a gbXML file. Comparisons of all surfaces against one another is possible, but beyond the scope of time for this phase of the work plan.

The inherent structure of gbXML makes it nearly trivial to pull out groups of surfaces that should make up one tightly-bound enclosed volume. As shown on the right in Figure X, each Space element of gbXML contains an optional set of Space Boundary elements already nested below, each containing a PolyLoop for each surface said to be related to the Space parent. This logical grouping makes it possible to easily find each logical grouping of PolyLoops for each Space.

Alternatively, the Surface elements each contain a PolyLoop, that is referenced to volumes via the AdjacentSpaceId attribute, again making it trivial to pull out surfaces related to each volume. These logical groupings form the basis for which surfaces belong to which enclosed volumes, as defined by the gbXML file.

How the Enclosure Algorithm for gbXML Works



**Figure X - gbXML definition of a Volume:
a series of Polyloops**

As we discussed on page X, a gbXML file's structure stores each volume as a collection of PolyLoops that are supposed to make up a well-formed volume. These PolyLoop collections are bundled in the gbXML schema in one of two ways, either as a series of PolyLoops inside of each Space definition, or as a set of Surface PolyLoop objects connected by their AdjacentSpaceId values. In both cases, the PolyLoops can be thought of as a grouped set of surfaces, which the gbXML structure declares as connected to form a volume.

In Figure X to the left (exploded for clarity), there would be 6 (six) PolyLoop definitions that are declared in gbXML as connected to form the cube you see, each with 4 (four) edges. Every edge in this volume definition should have a perfect match.

The enclosure-checking algorithm's task is to keep track of every edge of every surface that makes up the definition of volume, and find all of the perfect matches that exist for each edge. It should then report to the user if all edges have perfect matches, and if not, then report where the edges have failed to match.

The case of the cube is one of the more simple cases that the validator can successfully find all edges and their perfect matches. More complex cases, described below, required that the enclosure check be flexible enough to handle a wide variety of shapes and conditions and still work successfully and seamlessly to the user.

There were two main challenges in creating an robust enclosure checker. First, was to create an algorithm that would be capable of handling the wide variety of conditions that it may face. The second, was to develop a reporting system decent enough to successfully report meaningful information back to the user. First, is a description of how the enclosure algorithm was made robust enough to handle the edge-tracking requirements to successfully match edges to one another for the wide variety of use cases that may be encountered in practice.

After reviewing how gbXML volums are created and stored in XML format, it became clear that there may be a number of cases that would make validation more difficult, particularly as shapes become more complicated. The simple cube example used in Figure X above is a very simple case of the type of information that could be stored in gbXML. The enclosure-checking algorithm is required to handle these simple and more complex use cases.

Space 1	Space 2
	Space 3
	Space 4

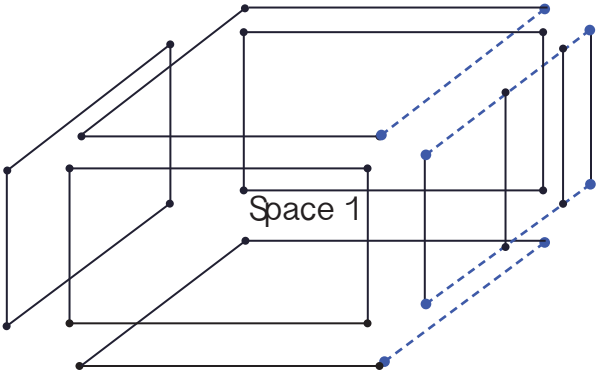


Figure X - Important Use Case #1

The complication that arises for the enclosure-checking algorithm is presented when any of these three smaller walls attempts to match its bottom and top edge to the ceiling and floor, or vice-versa, when one of the edges of the ceiling or floor attempt to match to the shorter edges of the wall. These edges are related but do not share coordinates in a standard way. Therefore, a simple coordinate matching algorithm would not be sufficient in this case. How to successfully validate this enclosure given that a “perfect match” using simple coordinate comparisons is not possible for these edges? (These edges are represented by blue dashed lines in Figure X above.)

We can further complicate matters by imagining the scenario where there are two spaces beneath space one, separated by a wall represented with a dotted line in Figure X to the right. Although Space 1 still remains one volume in gbXML, the relationship to the spaces below forces the floor to be broken into two separate surfaces to satisfy second-level space boundary requirements.

In this case, the floor, previously a single element in Figure x above, is now two separate floors as in Figure X to the right. Now several overlapping edges do not have any easily-defined relationship to one another that is precise, other than the fact that they are parallel and occupy the same space in 3D.

Space 1 is now represented by 9 PolyLoop definitions in gbXML.

Space 1	Space 2
	Space 3
	Space 4

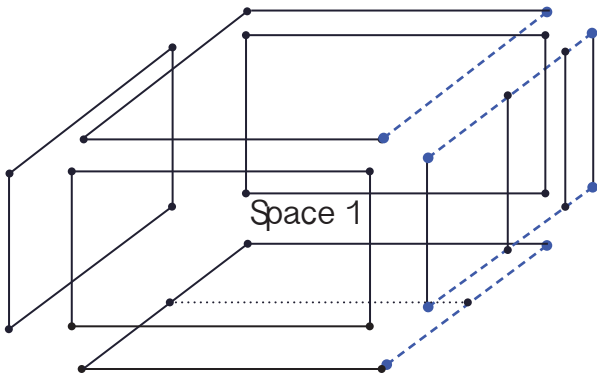
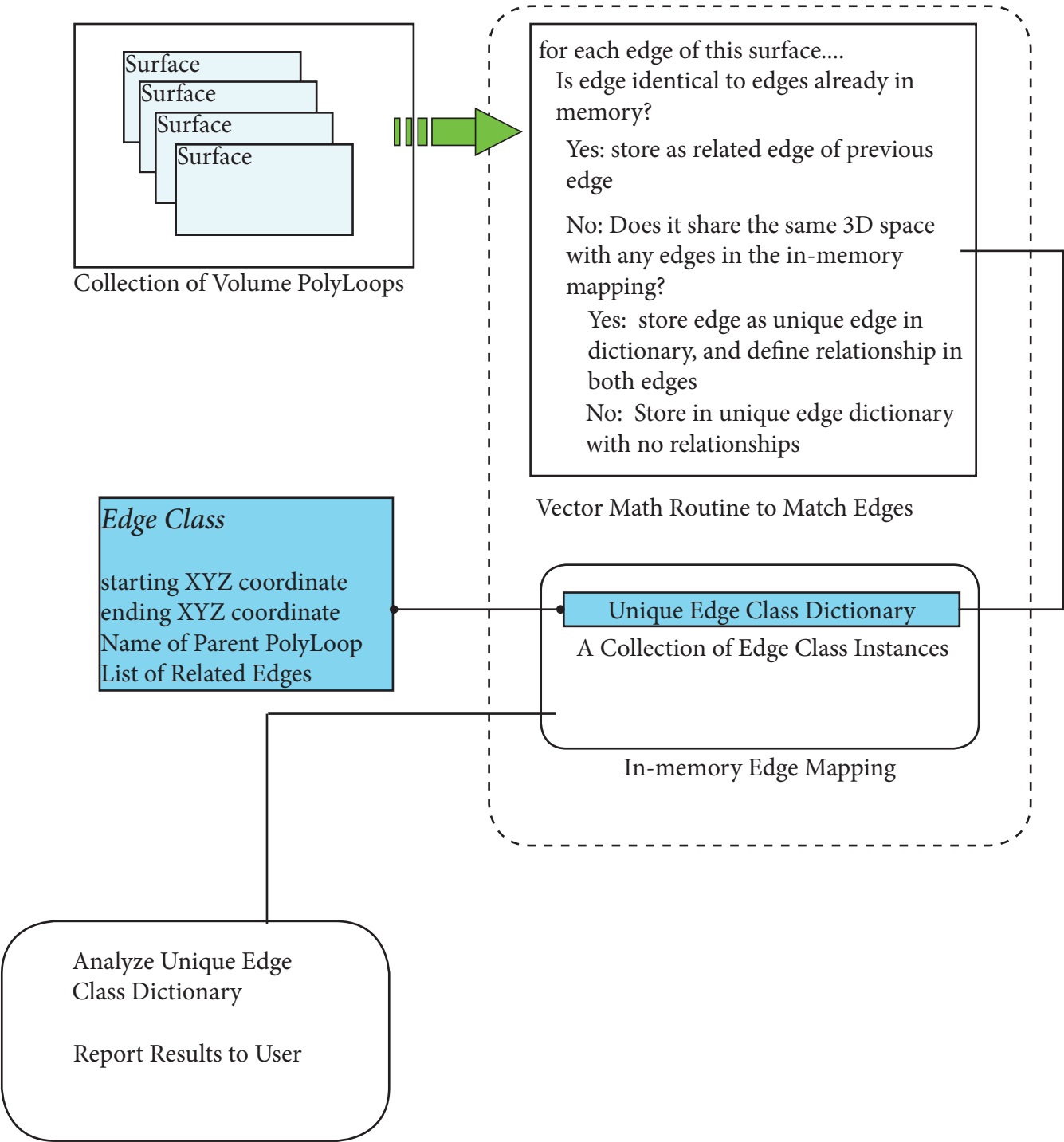


Figure X - Important Use Case #2

To solve these problems, the algorithm employs a vector algebra routine and an in-memory mapping of edges. The vector algebra routine finds edges that are parallel and in the same location in space. The in-memory mapping holds the relationships between edges. A separate reporting routine then reads the in-memory map of edge relationships and returns an answer to the user in comprehensible language that can be displayed on a page or in a 3D viewer.

Whether looking at PolyLoops located in the Space definition, or PolyLoops associated with the Surface elements in gbXML, the algorithm for both is essentially identical. Sending each surface:



In-Memory Mapping of Edge Relationships

The job of the in-memory edge store is to hold all edges that are unique, and the edges that are related to this edge.

The in-memory mapping stores each edge of the volume that is unique, and its related edges (the edges that align with it in 3D), using instances of the edge class. Related edges are also called overlapping edges. After all unique edges have been stored in memory, an analysis routine reviews each edge in the memory store, and evaluates whether the edge is properly-aligned with the edges that have been defined as related.

To create the in-memory mapping, each surface of the volume is sent to a routine that subsequently removes each edge in turn, comparing this edge to edges already in the in-memory store. This is shown schematically in Figure X to the right.

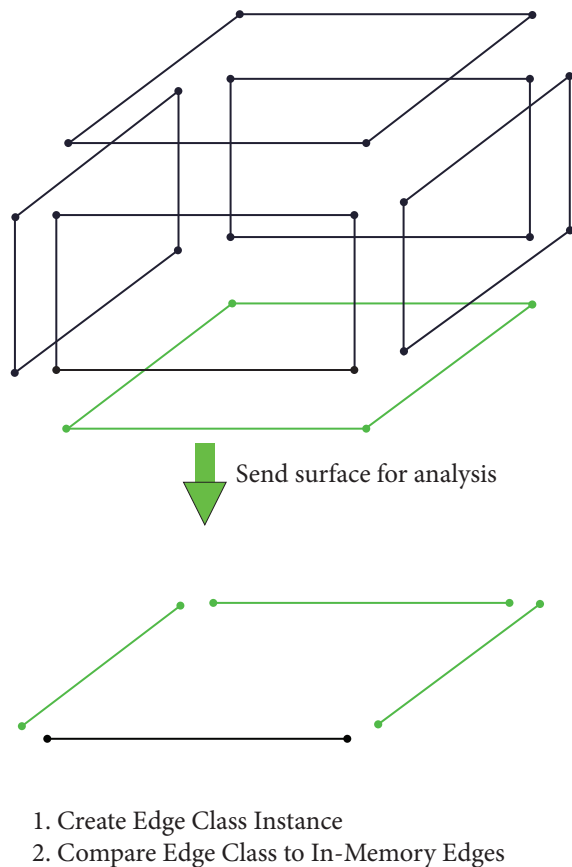
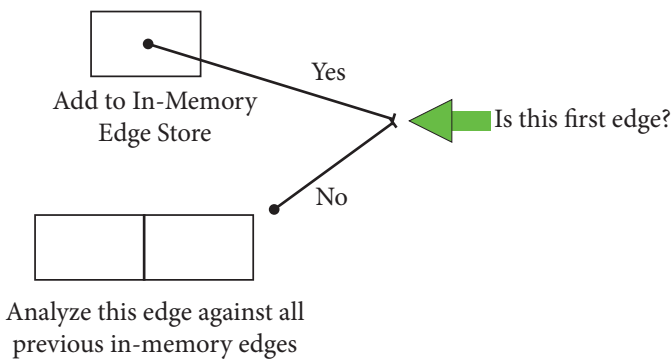


Figure X - Important Use Case #2

The in-memory store holds a history of all edges that are unique, and the edges that the algorithm finds overlaps this edge. The analysis to determine how edges may be related to one another is described in greater detail below. Here, we show how an in-memory edge or the edge in question may be added to the in-memory store.

Figure X below shows how the in-memory store is updated ff the edge in question (black) is found to overlap an in-mem-ory edge (blue dotted), AND perfectly aligns with the in-memory edge. The edge in question is simply added to the in-memory edge's related edge list. The edge in question is not added to the in-memory store because it is not unique.

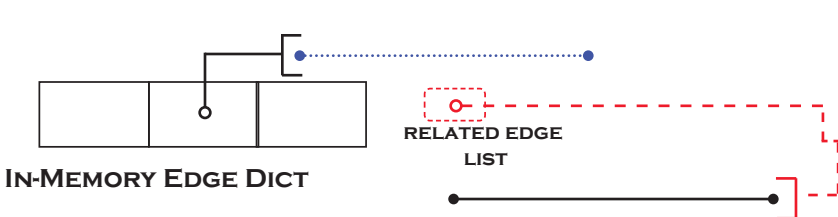


Figure X - Important Use Case #2

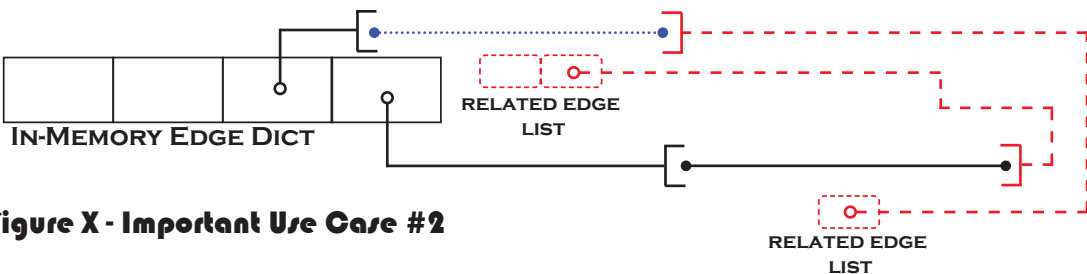


Figure X - Important Use Case #2

However, if the edge in question is found to overlap an in-memory edge, but NOT perfectly, then the edges are added to each other's related edge list and the edge in question is also added to the in-memory store, because it is unique. This is described in Figure X below. Otherwise, the edge in question will be added to the in-memory store.

Description of In-Memory Edge Store Analysis

Edge Class

starting XYZ coordinate
ending XYZ coordinate
Name of Parent PolyLoop
List of Related Edges

for each edge of this surface....

Is edge identical to edges already in memory?

Yes: store as related edge of previous edge

No: Does it share the same 3D space with any edges in the in-memory mapping?

Yes: store edge as unique edge in dictionary, and define relationship in both edges

No: Store in unique edge dictionary with no relationships

The heart of the algorithm is a set of logic and vector algebra that attempts to understand the nature of the relationship between surface edges already encountered, and the edge currently under analysis, through a comparison with historical edge class in-memory storage.


Each edge is defined as an instance of an Edge Class: at its most basic the class instance contains a starting and end point, the name of the surface it belongs to, and a list of all related edges. This list can grow dynamically depending upon the use case.

As the algorithm loops through each successive edge that it encounters when scanning each PolyLoop edge, it compares the current edge to the Unique Edge Class Dictionary. If no relationship can be established to previously-analyzed edges, the algorithm stores the edge. In this way, every unique edge found is stored in the dictionary and is used in successive comparisons with new edges as the algorithm moves forward. As you might intuitively imagine, the in-memory dictionary begins by storing many new edges, but as it proceeds, it begins to find more relationships that are stored in the List of Related Edges.

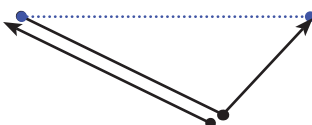
For more on how the Unique Edge Class Dictionary works, see below under the discussion of the dictionary and in-memory storage.

Whenever the algorithm is sent a new edge, the first relationship test is to identify if the current edge is identical to a previously identified edge already in the Unique Edge Class Dictionary. This test is a simple coordinate comparison where the coordinates of the current edge under analysis is checked against all previous edges in the memory store. If the coordinates match (within some allowable floating point tolerance) in X,Y,and Z, then the current edge is considered to be non-unique, and is only stored as a related edge to its complement already in the Unique Edge Class Dictionary. (it should now make sense why the dictionary is called a Unique Edge Class dictionary.


If the edge cannot find a perfect match, then the next test is to determine if the edge shares anything in common with edges already in the Unique Edge dictionary. To do so, an algorithm was developed to determine if two edges when compared to one another are parallel, and if parallel, if the two edges are found to mathematically overlap in any way. If both of these test prove true, then the edges are considered related. Because both edges are technically unique, the current edge is stored in the dictionary, and the edge found to match is stored as a related edge. As well, the current edge is stored as a related edge in the dictionary




Edges share one coordinate, current edge overlaps some



Edges share one coordinate, current edge not parallel



Edges share one coordinate, but don't overlap



Edges share one coordinate, current edge complete overlap

Figure X - Important Use Case #2

The first two situations are the easiest to determine, because a dot product between the two formed vectors will quickly reveal the truth. If the dot product returns -1, then the vectors are anti-parallel, which means that the edges do overlap some. If the dot product returns neither 0 nor -1, then we have a case where the edges are not parallel.

But if the vectors' dot product is equal to zero, then either of the last two situations could be true. In this case, a second test is conducted to determine which of the situations is at hand. One of several techniques could be used. The algorithm employs a technique where the edge stored in memory (dotted blue) is turned into a vector, and the dot product is taken with the previous vectors.¹



Figure X - Important Use Case #2

If the dot product reveals the vectors are antiparallel, then it is concluded that they overlap. If the vectors are parallel, then one coordinate is shared, but do not overlap. In any case where the two edges are determined to overlap, then the in-memory dictionary is updated with new edge relationships:

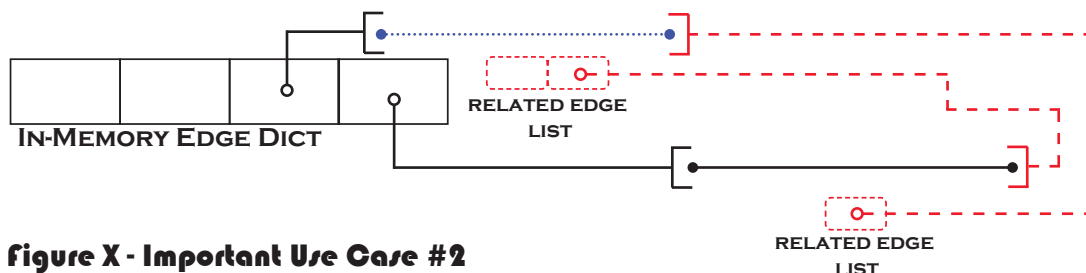


Figure X - Important Use Case #2

Case 1: Edges share one coordinate but do not share the second. Do the edges overlap in 3D space, or not?

The two edges (the current edge in question is in black, the edge stored in-memory that has matched is a dotted blue line) could take on one of four different relationships

In this case, the algorithm knows that the two edges are not identical, but they do share a common coordinate. The question then becomes, which situation (shown to the right) is occurring?

To determine this, two vectors are drawn from the second coordinate of the edge in question (black solid line). One vector is drawn to the second coordinate of the in-memory edge (blue line), the other vector is drawn to the shared identical coordinate.

¹ An alternative technique that could have been used is to compare the magnitude of vectors.

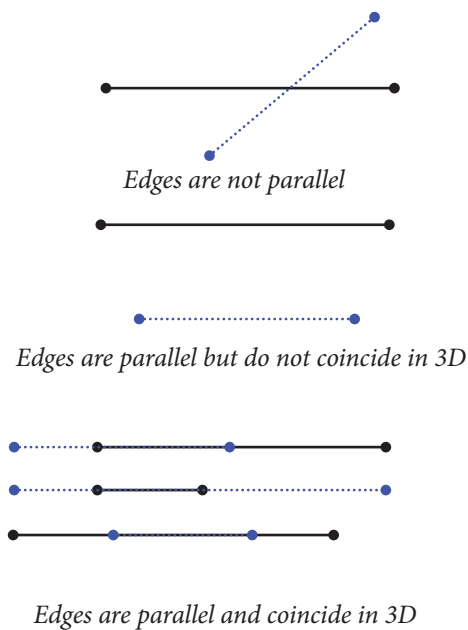


Figure X - Important Use Case #2

Like in case 1, the algorithm relies on dot products and vector math to determine if the parallel vectors share the same 3D space. Taking two edges that overlap as an example, the algorithm performs the following checks to find overlaps in 3D space:



Figure X - Important Use Case #2

Draw two vectors from each of the coordinates of the edge in question to one of the coordinates of the in-memory edge. The choice of coordinate of the in-memory edge is arbitrary, setting two different logic paths depending upon which coordinate in the in-memory edge is randomly chosen to draw the vectors. In one case, the vectors may be drawn anti-parallel, in the other case, the vectors drawn are parallel. This case is shown in the left of Figure X above, where the vectors drawn depend on which coordinate in the in-memory edge is chosen to draw the vectors. As this coordinate is arbitrary, it is conceivable for the vectors to be drawn forming a parallel pair or an anti-parallel pair.

If the resulting vectors, and the dot product taken of these vectors, returns an anti-parallel relationship, it is safe to conclude that the two edges overlap. However the case where the vectors return as parallel will require a second test, as it is not clear whether the two edges overlap or not, based on this simple test. This is shown in Figure X above where in two separate cases, resultant parallel vectors can result in both the overlap and non-overlap case,. Though in both cases the edges are aligned, in both cases they do not necessarily overlap.

Case 2: There are no shared coordinates between the edges. Do the edges overlap in 3D space, or not?

The two edges (the current edge in question is in black, the edge stored in-memory that has matched is a dotted blue line) could take on a multitude of relationships, where the following on the left are representative of these relationship conditions.

In the case where there is no coincident coordinate, the algorithm first attempts to see if the two edges are parallel. If parallel, the question then becomes, which situation of parallelism is occurring?

On the one hand, it is possible for edges to be parallel but in no way coincident. On the other hand the edges can be parallel and coincident, and coincident in one of several formations, all of which are considered to form neighboring relationships.

When the resulting vectors are parallel, one of three conditions may be possible, two of which occur in the overlap condition, one which occurs in a non-overlap. Figure X above showed two of these conditions, which will be repeated below in Figure X. The third condition will also be shown.

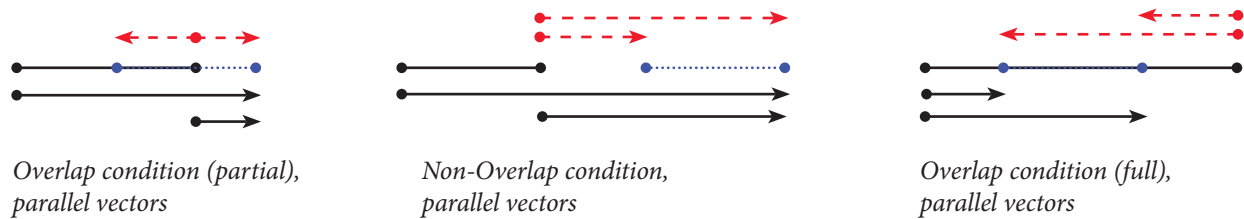


Figure X - Important Use Case #2

The second test that is performed is to draw two more vectors from the second coordinate of the edge in question (black solid line) to each of the coordinates of the in-memory edge (dotted blue line). These vectors are drawn in the Figure above as red dotted line vectors.

Again, if the resulting vectors come back from this test indicating that the resultant dot product is -1, indicating the vectors are anti-parallel, then the two edges overlap. This is shown in the Overlap condition (partial) example in Figure X above. However, in the Non-Overlap and Overlap condition (full), again this test comes back with two sets of parallel vectors. A final test is to take the cross product of a black vector and a red dotted vector. If the resulting dot product shows the two vectors are anti-parallel, then the condition is a full overlap condition, shown at the far right of Figure X above. Otherwise, it can be concluded that the edges are in a Non-Overlap condition (middle diagram of Figure X above).