Asociación Argentina
de Mecánica Computacional

Mecánica Computacional Vol XXIX, págs. 6065-6089 (artículo completo)
Eduardo Dvorkin, Marcela Goldschmit, Mario Storti (Eds.)
Buenos Aires, Argentina, 15-18 Noviembre 2010

ADAPTIVE PRECISION BASED FAST ALGORITHMS FOR ROBUST SURFACE INTERSECTIONS

Ricardo Marques[a], André Pereira[a,b], Luiz Fernando Martha[a], Antonio Miranda[c], Marcelo Gattass[a]

[a]*Tecgraf, Pontifical Catholic University, Rua Marques de Sao Vicente 225, 22451-900 Rio de Janeiro, Brazil, lfm@tecgraf.puc-rio.br, http://www.tecgraf.puc-rio.br*

[b]*Department of Civil Engineering, Fluminense Federal University, Rua Passo da Patria 156, 24210-240 Niterói, Brazil, andrebrabo@vm.uff.br, http://www.tecgraf.puc-rio.br/~brabo*

[c]*Department of Civil Engineering, University of Brasilia, Campus Darcy Ribeiro, 70910-900 Brasilia, Brazil, acmiranda@unb.br, http://www.tecgraf.puc-rio.br/~amiranda*

Keywords: Surfaces, Intersections, Meshes, Domain Partition, Geometric Predicates, Exact Arithmetic.

Abstract. Most available algorithms for performing surface intersections are very versatile and cover a wide range of particular cases. However, the main problem of such algorithms arises already during the implementations, since they involve geometric calculations that require high degree of accuracy (i.e. most of these algorithms were not designed to work with special codes capable to deal with exact computations). Therefore, such algorithms need to be reformulated in order to consider exact computations, satisfying some error bound, which avoid possible crashes of the system. Thus, a set of fast algorithms for robust surface intersections is proposed in this paper. Three different algorithms are proposed, one which deals with intersections of a plane with a surface; the second algorithm allows inserting a curve on a given surface; whereas the third one is responsible for performing intersections between two surfaces. All the algorithms were adapted in order to allow adaptive precision based fast computations by the direct use of robust and adaptive geometric predicate algorithms. Examples are presented to validate and to demonstrate the capabilities of the proposed algorithms.

1 INTRODUCTION

All science fields and engineering branches demand the construction of models in order to analyze or solve a problem. The models may commonly have a geometric representation. More realistic representations of the real world demand three-dimensional models. There are several ways to represent 3D geometric models in the literature. One of the most used solid representations is the well-known boundary representation. In such representation, the boundaries of the models are keys of the solid representation, which are constituted of a set of geometric entities called surfaces. In the geometric modeling of solids, the surfaces are usually represented in two different fashions: with a parametric representation or with a set of triangular facets (a triangulation).

The construction of complex solids requires several manipulations and operations with surfaces, such as intersections of surfaces with other surfaces (surface-surface intersections), intersections of surfaces with planes (surface-plane intersections), or still the insertion of a curve on a surface or the tearing of a surface with a curve (curve-on-surface insertion). Any procedure or algorithm which involves intersection of surfaces with other geometric entities (e.g. surface, plane or curve) is referred in this paper as "surface intersections" algorithms.

Therefore, surface intersections are one of the most important tools which a 3D geometric modeler must offer. Its robustness, however, is still a problem to be faced. As the model must be consistent, the precision of the intersection computations is critical. This is required not only to obtain, in the end, the exact position of points in space, but especially to, during the intersection calculations, classify correctly relative position of points in space and properly treat its intersection case. Moreover, a solution with a low computational cost and a high performance is also desired.

In this paper, a set of three fast and robust surface intersections algorithms is proposed. The first algorithm treats Surface-Plane Intersection problems, the second allows inserting interactively a curve on a given surface and the third one solves Surface-Surface Intersection problems. In the proposed approaches, surfaces are represented by triangular meshes and the intersection curves are polygonal curves composed by all intersection points.

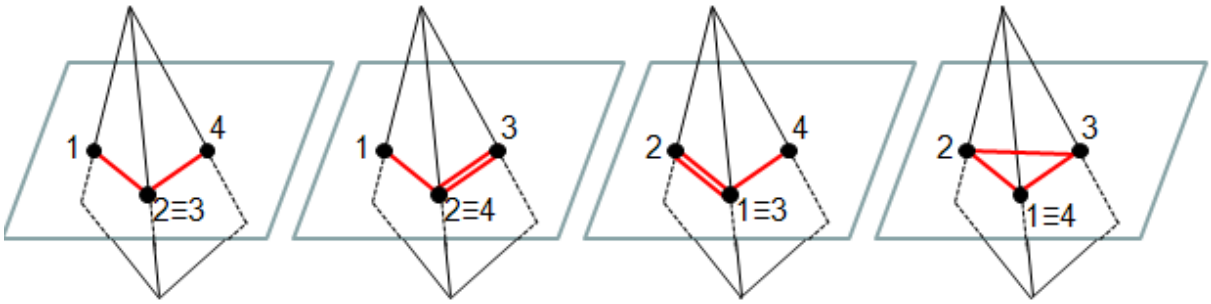
Basically, the implementations take advantage of two main features: the adaptive exact arithmetic geometric predicates (Shewchuk 1996a, b) and an adjacency-walking mesh structure. Since the predicates use exact arithmetic, they assure robustness of the calculations and intersection cases classifications. Besides, they are calculated adaptively taking a short amount of time. The employment of an adjacency-walking mesh structure facilitates the search for mesh edges during the intersection computations, once that it is possible to walk on the mesh edge by edge, as it were a graph. As a result, it is not necessary to check if all edges are intersected, but just a few connected edges. In addition, the walk on this graph enables to create the intersection curve with its points already sorted. In the end, the intersection curve creation process is speed up.

2 TRIANGLE-PLANE VERSUS SEGMENT-PLANE

In all proposed algorithms, the computation of the intersection between a cutting plane and an edge of the surface mesh is necessary. Some algorithms require few adaptations in the procedure, but its basis is common for all algorithms. Nevertheless, in any case, a robust Plane-Segment Intersection algorithm is needed.

Initially, the direct intersections between triangles and the cutting plane were also tested as an alternative, but their computational cost was higher. In other words, calculating triangle-plane intersections was slower than calculating the segment-plane intersections. This happens

because adjacent triangles have edges in common and when we calculate a triangle-plane intersection, we obtain two intersection points at once. So two adjacent triangles when intersected by the same plane will give us 4 intersection points, of which 2 are coincident. In fact, there is only 3 intersection points. In a triangular mesh, where a triangle has many adjacencies, these redundant calculations have a considerable cost. And, the worst, once that we cannot assure the curve points are in the correct order only by walking on adjacent intersected triangles, triangle-plane intersection requires that we check the orientation of every pair of intersection points in the curve.



(a) (b) (c) (d)

Figure 1: Triangle-Plane Intersection: (a) Curve Points Correctly Sorted (b) Two Last Curve Points Inverted (c) Two First Curve Points Inverted (d) All Curve Points Incorrectly Sorted

Sorting curve points means to minimize the total length of the polygonal (sum of the distances between every pair of consecutive points). The cost of this sorting is comparable to finding the shortest path in a graph (as in the Travelling Salesman problem). If we are already walking in a graph, we should be able to add points to the polygonal in the order they must be. Using triangles, we could check which edge should have the first intersection point of that triangle, but this would force us to walk on edges too. Thus, walking exclusively edge by edge seems a good method.

However, a very simple data structure that implements this adjacency-walking mesh structure does not offer a method that returns an adjacent edge given another edge. Consequently, we will walk on a triangle edges and on these edges' adjacent triangles edges. As in a graph, we will mark the edges we had already visited. So, when we enter an unvisited triangle, one of its edges was previously visited. Then, there will be only one more edge in intersection, whose adjacent triangle is the next to be visited.

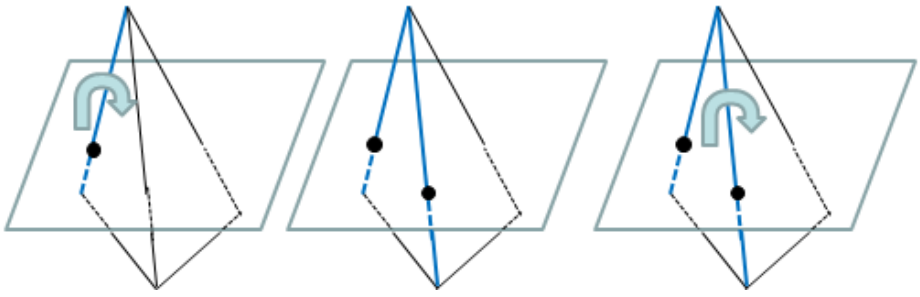


Figure 2: Walking on the Mesh. Visited Edges are in blue.

This way, we can add curve points without duplication and sorted. Now, we can build our basic algorithm that searches for intersections (Figure 3). It follows the style of a recursive depth search in a graph.

```
Search_Intersections(T):
For each unvisited edge e of triangle T
  Mark e as visited
  If e is intersected then
    Calculate the Intersection Point
    Search for more intersections in T
    Search for more intersections in the triangles adjacent to e
  End If
End For
```

Figure 3: Basic Search Intersections

3 ROBUST PLANE-SEGMENT INTERSECTION

In this section, we will firstly introduce a classical solution for the Plane-Segment Intersection and understand its robustness problem. Then, we will present the exact arithmetic predicates and deeply explore their geometric meaning so that we can, finally, build an exact arithmetic solution.

3.1 Plane-Segment Intersection

A plane π and a segment e, standing for an edge of a mesh, can be defined respectively as

$$\pi = \{\mathbf{X} \in \Re^3 \,/\, \mathbf{X}\cdot\mathbf{n}_\pi + d_\pi = 0\} \tag{1}$$

$$e = \{\mathbf{X} \in \Re^3 \,/\, \mathbf{X} = \mathbf{R} + t(\mathbf{S}-\mathbf{R})\}, \quad \forall t \in \Re \,/\, 0 \le t \le 1 \tag{2}$$

where $\mathbf{n}_\pi$ is the normal vector to π, d_π is the fourth plane coefficient and **R** and **S** are the vertices/endpoints of the edge/segment e. Therefore, an intersection point **P** must satisfy

$$\begin{cases} \mathbf{P}\cdot\mathbf{n}_\pi + d_\pi = 0 \\ \mathbf{P} = \mathbf{R} + t_P(\mathbf{S}-\mathbf{R}) \end{cases}, \quad 0 \le t_P \le 1 \tag{3}$$

Isolating t_P, it yields

$$t_P = \frac{\mathbf{R}\cdot\mathbf{n}_\pi + d_\pi}{(\mathbf{R}-\mathbf{S})\cdot\mathbf{n}_\pi}, \quad 0 \le t_P \le 1 \tag{4}$$

Finally, **P** may be calculated by replacing t_P in the equation below

$$\mathbf{P} = \mathbf{R} + t_P(\mathbf{S}-\mathbf{R}) \tag{5}$$

Note that if **R** belongs to plane, then the numerator of t_P will be 0, t_P will be 0 and **P** will be equal to **R**. If **S** belongs to plane, then the numerator of t_P will be equal to the denominator of t_P, so t_P will be 1 and **P** will be equal to **S**. If **R** and S belong to plane, then the numerator and the denominator of t_P will be 0. If t_P belongs to the interval [0,1], then there is one intersection point between **R** and **S** in the segment. Otherwise (t_P is out of the interval [0,1]) then the segment e does not intersect the plane, although a line in *RS* direction does. In fact, if t_P is negative or greater than 1, the segment lies strictly over or strictly under the plane.

3.2 The Robustness Problem

However, in the computation of t_P which precision becomes a problem. Suppose t_P is an extremely small positive number instead of 0, so the intersection point **P** will be very close to

R, but won't be properly **R**. Then, if the tolerance of the system is too low, **P** will be mistakenly considered a different point from **R**. What is more, if we increase the tolerance of the system to force **P** and **R** be coincident, others errors may occur because this tolerance may be too high for other operations of the system. This problem will be called "Positive Zero" problem. In spite of its name, mind that it also can happen when t_P is a little lower than 1, but not exactly 1, as a consequence that the numerator did not match precisely the denominator.

A greater problem appears if a t_P that should be 0 is an extremely small negative number ("Negative Zero" problem). If this t_P lies in the tolerance range of the system, an intersection point **P** will be calculated. In this case, **P** would be a point very close to **R** belonging to an edge adjacent to **R** and parallel to e which may not even exist. Definitely, **P** would be misplaced. On the other hand, if this t_P lies outside the tolerance range of the system, the edge e will be classified as not intersected and the intersection curve would miss one of its points. Being so, a small piece of the curve could not fit the surface. A similar problem can happen if a t_P is slightly greater than 1.

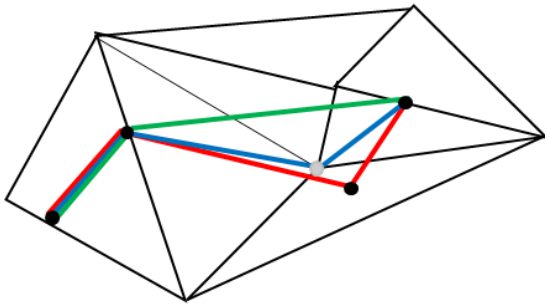


Figure 4: The "Negative Zero" Problem: The Blue curve represents the ideal curve, passing through **R** (the gray point). The red curve has a misplaced intersection point and the green curve has discarded an intersection point (note that the red and the green curve have parts that do not fit the surface).

As we have seen before, if the numerator and the denominator of t_P are 0, the intersection points are **R** and **S**. (In truth, in this condition, all intermediate points in the segment e belong to the plane, but we need only **R** and **S** as intersection points to build the curve.) Suppose that the denominator and the numerator of t_P are close to 0. If the denominator is considered as 0, two situations may happen:

- If the numerator is also considered as 0, both **R** and **S** will be included in the curve
- If the numerator is not considered as 0, e will be classified as not intersected.

Otherwise, if the denominator is not considered as 0,

- If the numerator is considered as 0, only **R** will be included in the curve, which would miss the point **S**.
- If the numerator is not considered as 0, t_P will be calculated. One of the previously mentioned "Negative Zero" or "Positive Zero" problems may happen. Anyway, or an intermediate point to segment RS will be included in the curve, or e will be classified as not intersected.

Hence, to solve these potential problems, the need of exact arithmetic comes.

3.3 The Exact Arithmetic Predicates

Shewchuk (1996a, b) provides four geometric orientation predicates using adaptive exact arithmetic:

- *orient2d*: determines if a point lies above, under or belongs to a line in 2D space
- *orient3d*: determines if a point lies above, under or belongs to a plane in 3D space

- *incircle*: determines if a point lies inside, outside or belongs to a circle in 2D space
- *insphere*: determines if a point lies inside,outside or belongs to a sphere in 3D space

In these predicates, all floating-point numbers are split in two components of non-overlapping floating point numbers with different order of magnitudes. Arithmetic operations, as sum, subtraction and multiplication, were redesigned deal with these components. Once these operations always receive two components and produce two components, roundoff errors are avoided. As a result, we have a higher degree of precision than the machine offers.

In addition, these predicates are calculated adaptively. This way, we speed up exact arithmetic calculations, which would be, at first, very expensive. When a low degree of precision is required, the calculations stop early, taking a minimal cost. And when more precision is needed, approximations are continuously refined until they reach a trustable error bound.

As we are interested in solving segment-plane intersections, we will focus on *orient3d*, which is the most useful for us. Consider a point **D** in 3D space and a plane π containing the points **A**, **B** and **C**.

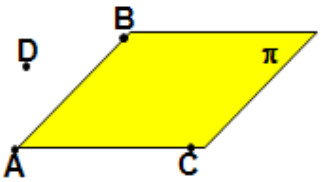


Figure 5: The plane π defined by the points **A**, **B** and **C** and the point **D** to be oriented in relation to π

Then, *orient3d* calculates the following determinant

$$orient3d(\mathbf{A},\mathbf{B},\mathbf{C},\mathbf{D}) = \begin{vmatrix} A_x & A_y & A_z & 1 \\ B_x & B_y & B_z & 1 \\ C_x & C_y & C_z & 1 \\ D_x & D_Z & D_z & 1 \end{vmatrix} = \begin{vmatrix} A_x - D_x & A_y - D_y & A_z - D_z \\ B_x - D_x & B_y - D_y & B_z - D_z \\ C_x - D_x & C_y - D_y & C_z - D_z \end{vmatrix} \tag{6}$$

The predicate's sign allows us to classify the position of the point **D** in relation with the plane π. If **A**, **B** and **C** are given clockwise, then

$$orient3d(\mathbf{A},\mathbf{B},\mathbf{C},\mathbf{D}) \begin{cases} > 0 \ \ if \ \ \mathbf{D} \ \ lies \ \ over \ \ \pi \\ = 0 \ \ if \ \ \mathbf{D} \ \ belongs \ to \ \ \pi \\ < 0 \ \ if \ \ \mathbf{D} \ lies \ under \ \pi \end{cases} \tag{7}$$

The predicate's module can be also geometrically interpreted. Rewriting *orient3d* using the mixed product notation

$$\begin{aligned} orient3d(\mathbf{A},\mathbf{B},\mathbf{C},\mathbf{D}) &= \langle \mathbf{A}-\mathbf{D}, \mathbf{B}-\mathbf{D}, \mathbf{C}-\mathbf{D} \rangle = \langle \mathbf{r}_{DA}, \mathbf{r}_{DB}, \mathbf{r}_{DC} \rangle \\ &= \langle \mathbf{D}-\mathbf{A}, \mathbf{B}-\mathbf{A}, \mathbf{C}-\mathbf{A} \rangle = \langle \mathbf{r}_{AD}, \mathbf{r}_{AB}, \mathbf{r}_{AC} \rangle \end{aligned} \tag{8}$$

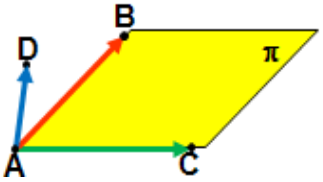


Figure 6: The vectors $\mathbf{r}_{AD}$, $\mathbf{r}_{AB}$ and $\mathbf{r}_{AC}$

Therefore, by a geometric interpretation of the mixed product, the predicate corresponds to the signed volume of a parallelepiped with basis defined by $\mathbf{r}_{AB}$ and $\mathbf{r}_{AC}$ vectors and height given by $\mathbf{r}_{AD}$ vector. Or better, the module of *orient3d* is equal to 6 times the volume of the tetrahedron defined by **A**, **B**, **C** and **D**.

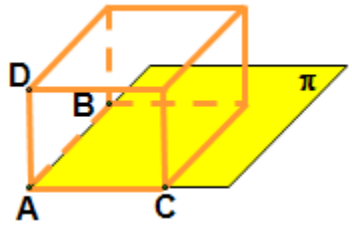


Figure 7: The parallelepiped with basis $\mathbf{r}_{AB}$ x $\mathbf{r}_{AC}$ and height $\mathbf{r}_{AD}$

Alternatively, if $\mathbf{r}_{AB}$ and $\mathbf{r}_{AC}$ vectors are normalized, the predicate module is equivalent to the distance from the point **D** to the plane π.

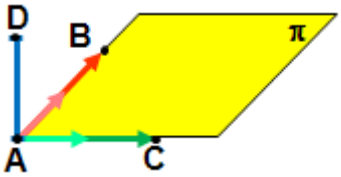


Figure 8: The vectors $\mathbf{r}_{AB}$ and $\mathbf{r}_{AC}$ normalized and the distance from the point **D** to the plane π.

Demonstration: To prove this statement, let's see the classic formula of the distance from a point to a plane. Given the four plane π coefficients *a*, *b*, *c* and *d* and one point **D**, the signed distance from the point **D** to the plane π is:

$$dist(\mathbf{D},\pi) = \frac{a \cdot D_x + b \cdot D_y + c \cdot D_z + d}{\sqrt{a^2 + b^2 + c^2}} \tag{9}$$

As the vector (*a*,*b*,*c*) is the normal of the plane π, follows that

$$dist(\mathbf{D},\pi) = \frac{\mathbf{n}_\pi \cdot \mathbf{D} + d_\pi}{\|\mathbf{n}_\pi\|}, \text{ where } d_\pi = d \tag{10}$$

If the plane π is defined by the points **A**, **B** and **C** (or by the vectors $\mathbf{r}_{AB}$ and $\mathbf{r}_{AC}$) one might say that

$$\mathbf{n}_\pi = \mathbf{r}_{AB} \times \mathbf{r}_{AC} \tag{11}$$

In addition, we can determine d_π once **A** belongs to π

$$\begin{aligned} &\mathbf{n}_\pi \cdot \mathbf{A} + d_\pi = 0 \therefore \\ &d_\pi = -\mathbf{A} \cdot \mathbf{n}_\pi \therefore \\ &d_\pi = -\mathbf{A} \cdot (\mathbf{r}_{AB} \times \mathbf{r}_{AC}) \end{aligned} \tag{12}$$

Therefore

$$dist(\mathbf{D},\pi) = \frac{\mathbf{D} \cdot (\mathbf{r}_{AB} \times \mathbf{r}_{AC}) - \mathbf{A} \cdot (\mathbf{r}_{AB} \times \mathbf{r}_{AC})}{\|\mathbf{r}_{AB} \times \mathbf{r}_{AC}\|} = \frac{(\mathbf{D} - \mathbf{A}) \cdot (\mathbf{r}_{AB} \times \mathbf{r}_{AC})}{\|\mathbf{r}_{AB} \times \mathbf{r}_{AC}\|} = \frac{\langle \mathbf{r}_{AD}, \mathbf{r}_{AB}, \mathbf{r}_{AC} \rangle}{\|\mathbf{r}_{AB} \times \mathbf{r}_{AC}\|} = \frac{orient3d(\mathbf{A},\mathbf{B},\mathbf{C},\mathbf{D})}{\|\mathbf{r}_{AB} \times \mathbf{r}_{AC}\|}$$

For any $\mathbf{r}_{AB}$ and $\mathbf{r}_{AC}$ vectors

$$orient3d(\mathbf{A},\mathbf{B},\mathbf{C},\mathbf{D}) = \left\|\mathbf{r}_{AB}\times\mathbf{r}_{AC}\right\|\cdot dist(\mathbf{D},\pi) \tag{13}$$

This means that, if $\mathbf{r}_{AB}$ and $\mathbf{r}_{AC}$ are not normalized, *orient3d* will return a scaled signed distance from **D** to the plane π. This scale factor is the area of the parallelogram given by $\mathbf{r}_{AB}$ and $\mathbf{r}_{AC}$ vectors. So, as mentioned before, *orient3d* really calculates the volume of a parallelogram, by multiplying its basis area (scale factor) for its height (distance to plane).

In Eq. (13), however, if $\mathbf{r}_{AB}$ and $\mathbf{r}_{AC}$ are unitary vectors, their norm is 1, so their cross product will result in a vector of norm 1 and consequently

$$orient3d(\mathbf{A},\mathbf{B},\mathbf{C},\mathbf{D}) = dist(\mathbf{D},\pi) \tag{14}$$

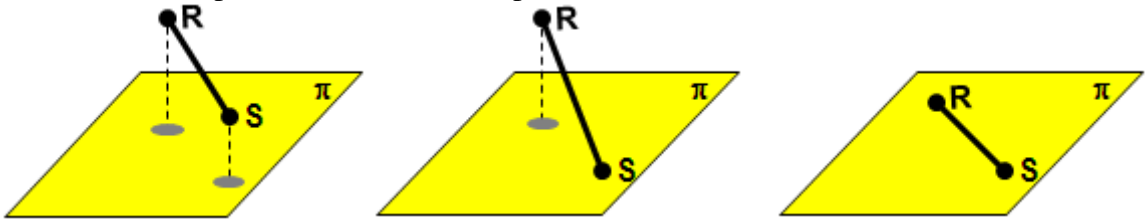
3.4 The Exact Arithmetic Solution

To build our exact arithmetic solution, first we will classify the intersection case of the considered edge. Doing this, we can rapidly identify edges that are not in intersection with the plane and avoid some calculations as well. For example, if we know that an endpoint is the intersection point, we do not have to calculate t_P (see Eq. (4)) to figure out it is 0 or 1 and apply this t_P to Eq. (5). In fact, t_P needs to be calculated only if an intermediate point of the segment intersects the plane.

Latter, we will conclude that any intermediate point in the segment *RS* in intersection with the plane π can be expressed using the predicates. And, finally, we will present a robust algorithm for a segment-plane intersection using adaptive exact arithmetic.

3.4.1 Intersection Classification

For now, using the predicate's sign (see Eq. (7)), we know how to classify whether a point is over, under or exactly in a plane. Every segment whose endpoints lay both above or both below the plane, surely does not intersect the plane. If a segment endpoint lies exactly in the plane, then this endpoint is an intersection point.



(a) (b) (c)

Figure 9: Some Segment-Plane Intersection cases: (a) Segment strictly over the plane. (b) Segment over the plane with an endpoint in intersection (c) Segment contained in the plane

The unique case of segment-plane intersection which we cannot treat yet is when there is an endpoint over the plane and another under the plane. This way the intersection point will be a segment intermediate point.

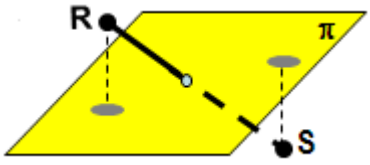


Figure 10: Segment with an intermediate point in intersection with the plane

In short, Segment-Plane intersection cases are:

Classification	*orient3d* sign of R	*orient3d* sign of S	Intersection Point
Strictly Over	+	+	None
Over with intersection	0	+	R
	+	0	S
Contained	0	0	R and S
Under with intersection	0	-	R
	-	0	S
Strictly Under	-	-	None
Through	+	-	Between R and S
	-	+	Between R and S

Table 1: Segment-Plane Intersection Cases.

3.4.2 The Intermediate Point Hypotheses

If the intermediate point could also be calculated using some of the information provided by the predicates, our algorithm would gain a lot in exactness and speed. In fact, we can raise some hypotheses based on the geometric interpretations of the predicates.

Hypothesis I: (Interpolating Signed Volumes) When the predicates are applied to a plane and each segment endpoint, we have the values of two signed parallelepipeds. A segment which has an intermediate point as the intersection point has two orient3d values (applied to its endpoints) with different signs. We know that for any point in the plane, its parallelepiped has volume zero. Considering an intersection point given by the parametric equation of its segment (see Eq. (2)), shouldn't the parameter t_P (see Eq. (4)) be proportional to these signed volumes?

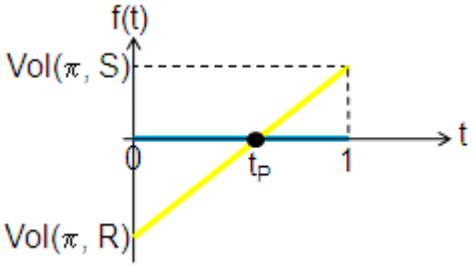


Figure 11: The hypothetic signed volumes interpolation function

Intuitively it seems so. Then, suppose a linear function f

$$f : [0,1] \in \Re \rightarrow \Re \,/\, f(t) = \alpha t + \beta \tag{15}$$

that correlates the parameter t with the signed volumes, as following:

$$\begin{cases} f(0) = Vol(\pi, \mathbf{R}) \\ f(1) = Vol(\pi, \mathbf{S}) \end{cases} \tag{16}$$

Let's solve a system to find out the expression of f:

$$\begin{cases} \alpha \cdot 0 + \beta = Vol(\pi, \mathbf{R}) \\ \alpha \cdot 1 + \beta = Vol(\pi, \mathbf{S}) \end{cases} \therefore$$
$$\begin{cases} \beta = Vol(\pi, \mathbf{R}) \\ \alpha + \beta = Vol(\pi, \mathbf{S}) \end{cases} \therefore$$
$$\alpha = Vol(\pi, \mathbf{S}) - Vol(\pi, \mathbf{R})$$
$$\beta = Vol(\pi, \mathbf{R})$$

Therefore, f is

$$f(t) = \left(Vol(\pi, \mathbf{S}) - Vol(\pi, \mathbf{R})\right) t + Vol(\pi, \mathbf{R}) \tag{17}$$

But if t_P is the t of the intersection point P, which makes volume zero with the plane, then

$$f(t_p) = 0 \tag{18}$$

So, we can find an expression for t_P

$$\left(Vol(\pi, S) - Vol(\pi, R)\right) t_p + Vol(\pi, R) = 0 \therefore$$
$$\left(Vol(\pi, R) - Vol(\pi, S)\right) t_p = Vol(\pi, R) \therefore$$

$$t_p = \frac{Vol(\pi, R)}{Vol(\pi, R) - Vol(\pi, S)} \tag{19}$$

If the points A, B and C belong to π, then

$$t_p = \frac{orient3d(A, B, C, R)}{orient3d(A, B, C, R) - orient3d(A, B, C, S)} \tag{20}$$

Hypothesis II: (Interpolating Equally Scaled Signed Distances) When the predicates are applied to a plane and each segment endpoint, we have the values of two equally scaled signed distances from this plane to each segment endpoint. We know that these distances are equally scaled because they refer to the same plane points, Eq. (13). Considering an intersection point given by the parametric equation of its segment, as in Eq. (2), shouldn't the parameter t_P (see Eq. (4)) be proportional to these scaled signed distances?

Again, it seems so. Considering λ as the common scale factor, we could interpolate these scaled distances exactly as we did before.

$$f : [0,1] \in \Re \rightarrow \Re \,/\, f(t) = \alpha t + \beta \tag{21}$$

$$\begin{cases} f(0) = \lambda \cdot Dist(\pi, R) \\ f(1) = \lambda \cdot Dist(\pi, S) \end{cases}$$
$$\begin{cases} \alpha \cdot 0 + \beta = \lambda \cdot Dist(\pi, R) \\ \alpha \cdot 1 + \beta = \lambda \cdot Dist(\pi, S) \end{cases} \therefore$$
$$\begin{cases} \beta = \lambda \cdot Dist(\pi, R) \\ \alpha + \beta = \lambda \cdot Dist(\pi, S) \end{cases} \therefore \tag{22}$$
$$\alpha = \lambda \cdot Dist(\pi, S) - \lambda \cdot Dist(\pi, R)$$
$$\beta = \lambda \cdot Dist(\pi, R)$$

$$f(t) = \left(\lambda \cdot Dist(\pi, S) - \lambda \cdot Dist(\pi, R)\right) t + \lambda \cdot Dist(\pi, R) \tag{23}$$

As the distance of the intersection point to the plane is 0,

$$\left(\lambda \cdot Dist(\pi, S) - \lambda \cdot Dist(\pi, R)\right) t_p + \lambda \cdot Dist(\pi, R) = 0 \tag{24}$$

$$\left(\lambda \cdot Dist(\pi, R) - \lambda \cdot Dist(\pi, S)\right) t_p = \lambda \cdot Dist(\pi, R) \therefore$$

$$t_p = \frac{\lambda \cdot Dist(\pi, R)}{\lambda \cdot Dist(\pi, R) - \lambda \cdot Dist(\pi, S)} \therefore$$

$$t_p = \frac{Dist(\pi, R)}{Dist(\pi, R) - Dist(\pi, S)} \tag{25}$$

If the points A, B and C belong to π, then

$$t_p = \frac{orient3d(A, B, C, R)}{orient3d(A, B, C, R) - orient3d(A, B, C, S)} \tag{26}$$

Note that as the distances were equally scaled, the scale factor had influence on the interpolation. This non-scaled result also seems reasonable.

Hypothesis III: (Triangles Similarity) Suppose that when the predicates are applied to a plane and each segment endpoint, we have the values of two non-scaled signed distances from this plane to each segment endpoint. Considering an intersection point given by the parametric equation of its segment (see Eq. (2)), shouldn't the parameter t_P (see Eq. (4)) be proportional to these signed distances?

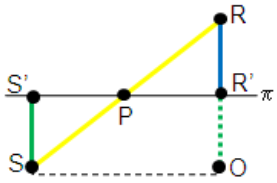


Figure 12: Orthogonal View of the plane π intersected by RS segment with non-scaled signed distances

We can see that the triangle RPR' is similar to the triangle RSO. Therefore

$$\Delta RPR' \sim \Delta RSO \therefore$$

$$\frac{\|RP\|}{\|RR'\|} = \frac{\|RS\|}{\|RO\|} \tag{27}$$

Discarding the norm of the numerators to obtain a vectorial expression:

$$\frac{P - R}{\|R' - R\|} = \frac{S - R}{\|O - R\|} \tag{28}$$

$$\frac{P - R}{|Dist(\pi, R)|} = \frac{S - R}{|Dist(\pi, R)| + |Dist(\pi, S)|} \tag{29}$$

$$P = R + \frac{|Dist(\pi, R)|}{|Dist(\pi, R)| + |Dist(\pi, S)|}(S - R) \tag{30}$$

Matching Eq. (30) and Eq. (5), we conclude that

$$t_p = \frac{|Dist(\pi, R)|}{|Dist(\pi, R)| + |Dist(\pi, S)|} \tag{31}$$

But as $Dist(\pi,R)$ and $Dist(\pi,S)$ have opposite signs, we can say that

$$t_p = \frac{Dist(\pi, R)}{Dist(\pi, R) - Dist(\pi, S)} \tag{32}$$

which is same equation that we have obtained on Hypothesis II, Eq. (25). Again, if the points A, B and C belong to π, then

$$t_p = \frac{orient3d(A, B, C, R)}{orient3d(A, B, C, R) - orient3d(A, B, C, S)} \tag{33}$$

Note that if *orient3d* returned two equally scaled signed distances instead of two non-scaled signed distances, the scale factor would be eliminated and the result of t_P would not be affected. Actually, we would have another pair of similar triangles, but its hypotenuses would still intersect the plane π in the point P. Assuming that λ is a scale factor, then we would have the following situation

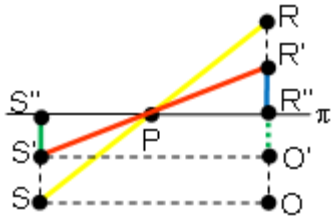


Figure 13: Orthogonal View of the plane π intersected by RS segment with a λ-scaled signed distances, 0<λ<1

Our old triangle similarity is still valid (notice the change of the points' names):

$$\Delta RPR'' \sim \Delta RSO \therefore$$
$$\frac{\|RP\|}{\|RR''\|} = \frac{\|RS\|}{\|RO\|} \tag{34}$$

But we can see that

$$\|RO\| = \|RR''\| + \|R''O\| = \|RR''\| + \|SS''\| \tag{35}$$

By the common scale factor λ, we also know that

$$\frac{\|RR''\|}{\|R'R''\|} = \frac{\|SS''\|}{\|S'S''\|} = \lambda \tag{36}$$

Therefore,

$$\|RR''\| = \lambda\|R'R''\|$$
$$\|RO\| = \lambda\|R'R''\| + \lambda\|S'S''\| \tag{37}$$

Combining Eq. (33) with Eq. (36) and simplifying

$$\frac{\|RP\|}{\lambda\|R'R''\|}=\frac{\|RS\|}{\lambda\|R'R''\|+\lambda\|S'S''\|}\quad\therefore$$
$$\frac{\|RP\|}{\|R'R''\|}=\frac{\|RS\|}{\|R'R''\|+\|S'S''\|}\quad\therefore$$
$$\frac{RP}{\|R'R''\|}=\frac{RS}{\|R'R''\|+\|S'S''\|}\quad\therefore$$
$$P=R+\frac{\|R'R''\|}{\|R'R''\|+\|S'S''\|}(S-R)\quad\therefore$$
$$t_p=\frac{\|R'R''\|}{\|R'R''\|+\|S'S''\|}\quad\therefore$$
$$t_p=\frac{|orient3d(A,B,C,R)|}{|orient3d(A,B,C,R)|+|orient3d(A,B,C,S)|}\quad\therefore \tag{38}$$

And, at last,

$$t_p=\frac{orient3d(A,B,C,R)}{orient3d(A,B,C,R)-orient3d(A,B,C,S)} \tag{39}$$

3.4.3 The Hypotheses Proof

So far, we have seen that all hypotheses have reached the same expression, see Eqs. (39), (33), (26) and (20). But, does this expression really match the algebraic formula for t_P (see Eq. (4))? If it does, it is proven that t_P can be expressed with predicates, and thus, any intermediate point **P** in segment RS in intersection with a plane π can be expressed with predicates.

Proof: (Expressing an Intermediate point with predicates) We must simply prove that:

$$t_p=\frac{R\cdot n_\pi+d_\pi}{(R-S)\cdot n_\pi}=\frac{orient3d(A,B,C,R)}{orient3d(A,B,C,R)-orient3d(A,B,C,S)} \tag{40}$$

With the Eq. (4) and Eq. (11),

$$t_p=\frac{R\cdot n_\pi+d_\pi}{(R-S)\cdot n_\pi}=\frac{R\cdot n_\pi-A\cdot n_\pi}{(R-S)\cdot n_\pi} \tag{41}$$

Therefore

$$\begin{aligned}t_p&=\frac{(R-A)\cdot n_\pi}{(R-S)\cdot n_\pi}=\frac{(R-A)\cdot(B-A)\times(C-A)}{(R-S)\cdot(B-A)\times(C-A)}=\frac{\langle R-A,B-A,C-A\rangle}{\langle R-S,B-A,C-A\rangle}\\&=\frac{\langle R-A,B-A,C-A\rangle}{\langle(R-A)-(S-A),B-A,C-A\rangle}\\&=\frac{\langle R-A,B-A,C-A\rangle}{\langle R-A,B-A,C-A\rangle-\langle S-A,B-A,C-A\rangle}=\frac{\langle AR,AB,AC\rangle}{\langle AR,AB,AC\rangle-\langle AS,AB,AC\rangle}\\&=\frac{\langle RA,BA,CA\rangle}{\langle RA,BA,CA\rangle-\langle SA,BA,CA\rangle}\end{aligned} \tag{42}$$

By the Eq. (8)

$$t_p = \frac{\langle RA, BA, CA \rangle}{\langle RA, BA, CA \rangle - \langle SA, BA, CA \rangle} = \frac{orient3d(A,B,C,R)}{orient3d(A,B,C,R) - orient3d(S,B,C,R)} \tag{43}$$

3.4.4 The Robust Algorithm

Given 3 plane points and both segment endpoints, we calculate two predicates, each one relative to one endpoint. Based on these predicates signs, we check in which intersection case this segment is with this plane: if the segment is strictly under the plane, strictly over, over with one intersection point, under with one intersection point, contained on the plane or if the segment cuts the plane. In the latter case, we need to calculate t_P as in Eq. (39) or Eq. (33), or (26), or Eq. (20) and then calculate the intersection point **P** as in Eq. (5). In all the other cases where there are intersection points, we do not have to calculate them, because they will be a known endpoint. Finally, we present our robust Segment-Plane Intersection algorithm.

```
Segment_Plane_Intersection(e, π):
If both endpoints of e are over π then
    return false
End If
If both endpoints of e are under π then
    return false
End If
If both endpoints do not belong to π then
    Calculate tp
    If 0 ≤ tp ≤ 1 then
        Calculate P
        Add P to curve
        return true
    Else
        return false
    End If
End If
If first endpoint belongs to π then
    Add first endpoint to curve
    If second endpoint belongs to π then
        Add second endpoint to curve
        return true
    End If
End If
If second endpoint belong to π then
    Add second endpoint to curve
    return true
End If
return false
```

Figure 14: Robust Segment-Plane Intersection

4 SURFACE-PLANE INTERSECTION

In the Surface-Plane intersection, the given plane is already the cut plane. So, given a triangular mesh representing a surface and the four coefficients of the cut plane, the resultant curve will be a polygonal composed by all intersection points between the cut plane and the mesh edges. Particularly in the Surface-Plane intersection, once that our cut plane is the infinite plane given, the intersection curves

- Or will traverse the surface from a boundary edge to another boundary edge
- Or will contour a peak

If the given surface contains holes, the edges in the border of the hole will be handled as boundary edges and our previous statement is still valid. Therefore, start looking for edges in intersection from an intersected boundary edge is very suitable.

Initially, all edges of the mesh are unvisited. Thus, for every unvisited boundary edge in intersection, we calculate the intersection point (see Figure 14), mark this edge as visited and start the search for intersections (as in Figure 3) from this edge's triangle. This method builds an intersection curve end to end (finishing at another boundary edge), marking all the edges this curve passes through as visited. So, all intersection points calculated after calling this method must be inserted in a new intersection curve.

This way, we have built all curves that traverse the surface from a boundary edge to another. After all intersected boundary edges were visited(and some other internal edges too), if any other unvisited internal edge has an intersection, the curve to be built will be a closed polyline, what means that this internal edge must be visited twice. In order to avoid a point repetition in the intersection curve and avoid an eternal loop, unvisited edges can be traversed just once and visited edges cannot be traversed at all. However, in this case we must be able to revisit the same edge, otherwise, we will not close the polyline. So, the unvisited intersected internal edge we have found will be marked as "to revisit". These edges can be visited and after its adjacent triangles where visited, the intersection point must be re-added to the curve, which must have a break. Then, for every unvisited edge in intersection, we calculate the intersection point, mark this edge as "to revisit" and start the search for intersection from this edge's triangle. Note that, to treat the peaks case, our algorithm has to visit all edges of the mesh, so it is $\theta(n_e)$.

An improvement can be made if, instead of calculating the predicates in the intersection method, we calculate all predicates previously for each vertex (node) of the mesh. Doing this, we avoid calculating the same predicate more than one time, as would happen in adjacent edges. According to Jiménez et al(2009), Segura and Feito(1998, 2001) stored all triangle normals and all signed volumes, being extremely fast, especially with static objects, as our case is. To calculate the predicates, given the four plane coefficients, a,b,c and d, we calculate 3 trivial plane points:

$$P_0 = \left(0,1,\frac{d+b}{c}\right); P_1 = \left(0,-1,\frac{-d+b}{c}\right); P_2 = \left(1,0,\frac{d+a}{c}\right) \tag{44}$$

Note that these points are oriented clockwise, as our predicates demand (see Eq. (7)). Hence, we can describe our Surface-Plane Intersection Algorithm as bellow:

```
Build_Curves( ):
For each edge e of the mesh
  Mark e as unvisited
End For
For each unvisited boundary edge e of the mesh
  If e is intersected then
    Calculate the Intersection Point
    Mark e as visited
    Search_Intersections(Triangle Adjacent to e)
  End If
End For
For each unvisited edge e of the mesh
  If e is intersected then
    Calculate the Intersection Point
    Mark e as "to revisit"
    Search_Intersections(Triangle Adjacent to e)
  End If
End For
```

Figure 15: Surface-Plane Build Curves

```
Search_Intersections(T):
For each unvisited or "to revisit" edge e of triangle T
  Mark e as visited
  If e is intersected then
    Calculate the Intersection Point
    Search for more intersections in T
    Search for more intersections in the triangles adjacent to e
     If new points were not added to the curve and e is a Boundary Edge then
        End the current curve and start creating a new curve
     End If
    If e was "to revisit"
      Calculate the Intersection Point
      End the current curve and start creating a new curve
     End If
End If
End For
```

Figure 16: Surface-Plane Search Intersections

With this algorithm, we can, for example, create isocurves of many levels as in the following pictures.

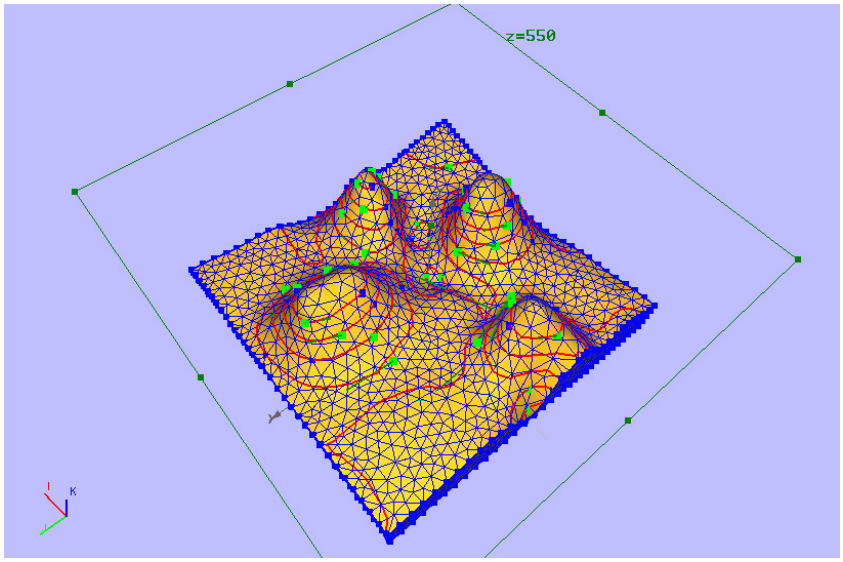


Figure 17: Isocurves with peaks

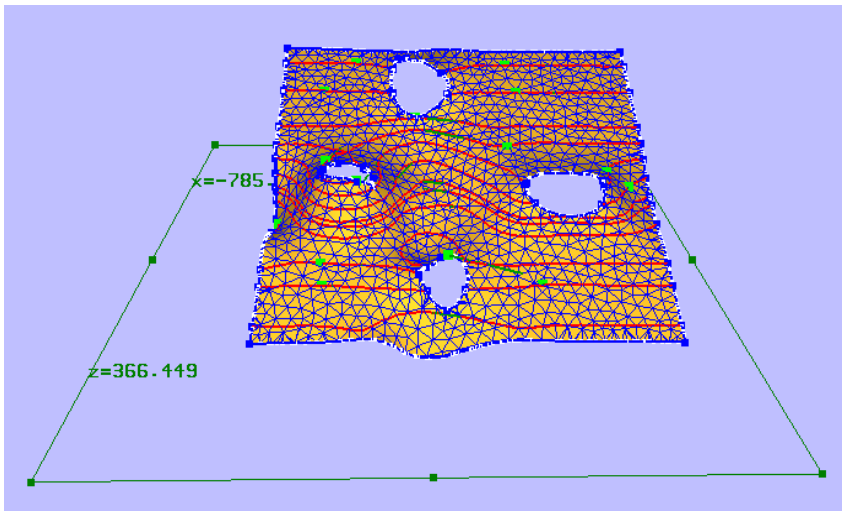


Figure 18: Isocurves bypassing holes

5 CURVE ON SURFACE

To insert interactively a curve on a surface, the user must, using a "Curve On Surface" tool, click 2 or more times on the surface. All these points will be attracted to the surface (as pick points) and then will be used as control points to build our curve on surface. For every pair of consecutive control points, we will build a cut plane and two delimiter planes. Then, a curve segment, starting with a control point and ending with the next control point, will be generated. In the end all curve segments will compose our Curve On Surface.

```
Build_Curves( ):
Add the first Control Point to the Curve
For each pair of consecutive Control Points
   Build the new Cut Plane
   Build the new Delimiter Planes
   Generate Curve Segment
   Add the next Control Point
End For
```

Figure 19: Curve On Surface Build Curves

5.1 Building the Cut Plane

Once we use the predicates, building a plane, actually means, defining 3 of its points. The cut plane must be orthogonal to the mesh and also contain both current control points. Let U and V be two consecutive control points. Known in which mesh triangle these points are, we can determine their normal vectors $\mathbf{n}_U$ and $\mathbf{n}_V$. Supposing that the surface is continuous, a good normal to this cut plane should be orthogonal to an average vector of $\mathbf{n}_U$ and $\mathbf{n}_V$.

$$n_m = \frac{n_u + n_v}{2} \tag{45}$$

and orthogonal to the vector UV. So, the normal of the cut plane π is:

$$n_\pi = UV \times n_m \tag{46}$$

Thus, 3 simple points that define π are:

$$\begin{aligned} P_0 &= U \\ P_1 &= U + n_m \\ P_2 &= V \end{aligned} \tag{47}$$

Note that these points are oriented clockwise, as our predicates demand (see Eq. (7)).

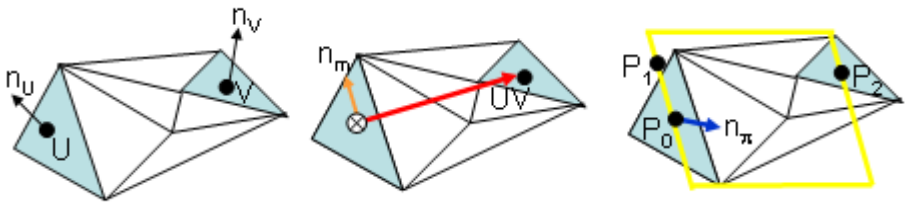


Figure 20: Defining the cut plane

Then, we calculate the predicate value for each vertex (node) of the mesh in relation with this cut plane and store them.

5.2 Building the Delimiter Planes

The delimiter planes are important because our cut plane is infinite and only intersection points between our control points must be added to the curve. In other words, intersection points before our first control point or after our second control point shall be rejected. Otherwise, we would have a full Plane-Surface intersection per each pair of consecutive control points.

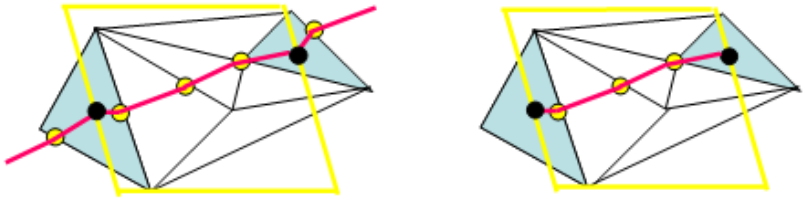


Figure 21: A non-delimited a Curve on Surface segment and a delimited one

There will be two delimiter planes: the left delimiter plane will discard intersection points before our first control point and the right delimiter plane will discard intersection points after our second control point.

The Delimiter planes must be orthogonal to the cut plane and must contain a control point. So, three left delimiter plane points can be:

$$\begin{aligned} L_0 &= U \\ L_1 &= U + n_m \\ L_2 &= U + n_\pi \end{aligned} \tag{48}$$

And three right delimiter plane points can be:

$$\begin{aligned} R_0 &= V \\ R_1 &= V - n_\pi \\ R_2 &= V - n_m \end{aligned} \tag{49}$$

Note that the left delimiter points are given counterclockwise and the right delimiter points are given clockwise.

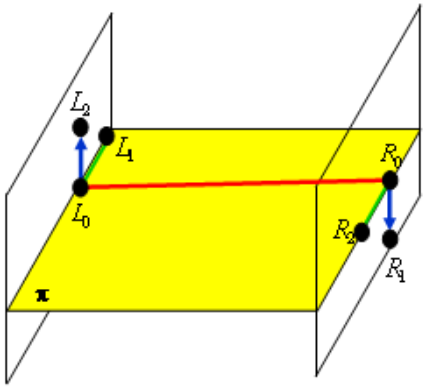


Figure 22: The Delimiter Plane Points

So, an intersection point D inside the limits (between the delimiter planes) must satisfy:

$$\begin{cases} orient3d(L_0, L_1, L_2, D) \geq 0 \\ orient3d(R_0, R_1, R_2, D) \geq 0 \end{cases} \tag{50}$$

Thus, our basic Segment-Plane Intersection methods changes a little.

```
Segment_Plane_Intersection(e, π):
If both endpoints of e are over π then
    return false
End If
If both endpoints of e are under π then
    return false
End If
If both endpoints do not belong to π then
    Calculate t_p
    If 0 ≤ t_p ≤ 1 then
        Calculate P
        If P lies between delimiter planes then
            Add P to curve
            return true
        Else
            return false
        End If
    Else
        return false
    End If
End If
If first endpoint belongs to π and
  first endpoint lies between delimiter planes then
    Add first endpoint to curve
    If second endpoint belongs to π and
       second endpoint lies between delimiter planes then
        Add second endpoint to curve
        return true
    End If
End If
If second endpoint belong to π and
  second endpoint lies between delimiter planes then
    Add second endpoint to curve
    return true
End If
return false
```

Figure 23: Segment-Plane Intersection for Curve On Surface

5.3 Building the Curve Segment

When we start building a new Curve segment, all edges of the mesh are unvisited. Then, if we know the triangles where our control points are, we just have to start searching for intersections in the mesh from this triangle (as in Figure 3). The last point of this segment will be at an edge of the triangle where the next control point is. So, add the next control point to the curve, get the next pair of consecutive control points, update all planes and build the next curve segment (as in Figure Figure 19)

Sadly, if the mesh contains a hole and our cut planes passes though it, our search would end in the border of the hole, once that no more connected edges would be in intersection. However, the edges that are in the border of the hole are considered boundary edges of the mesh. So, when our search reaches a boundary edge and no more intersection points were added, probably, we have found a hole. We skirt this hole by running along this edge's adjacent boundary edges (marked as skirted). When we find a boundary edge in intersection, we mark it as visited, calculate its intersection point, add this point to a new curve and restart our search from this edge's triangle. Note that, if our mesh is too irregular, we may have many intersection curves. In this case, our curves would split, not in holes, but in the border of the mesh, which are boundary edges anyway. So, this strategy would also work.

```
Search_Intersections(T):
For each unvisited or skirted edge e of triangle T
   If e is intersected then
    Mark e as visited
    Calculate the Intersection Point
    Search for more intersections in T
    Search for more intersections in the triangles adjacent to e
     If new points were not added to the curve and e is a Boundary Edge then
        End the current curve and start creating a new curve
        Skirt_Hole(T)
     End If
End If
End For
```

Figure 24: Curve On Surface Search Intersections

```
Skirt_Hole(T):
For each unvisited edge e of triangle T
   Mark e as skirted
   If e is a boundary edge and e is intersected then
    Calculate the Intersection Point
    Add the intersection point to the curve
    Search_Intersections(T)
    return
   End If
   If e is an edge with 1 or 2 boundary nodes then
    Skirt_Hole(Triangle Adjacent to e)
   End If
End For
```

Figure 25: Curve On Surface Skirt Hole

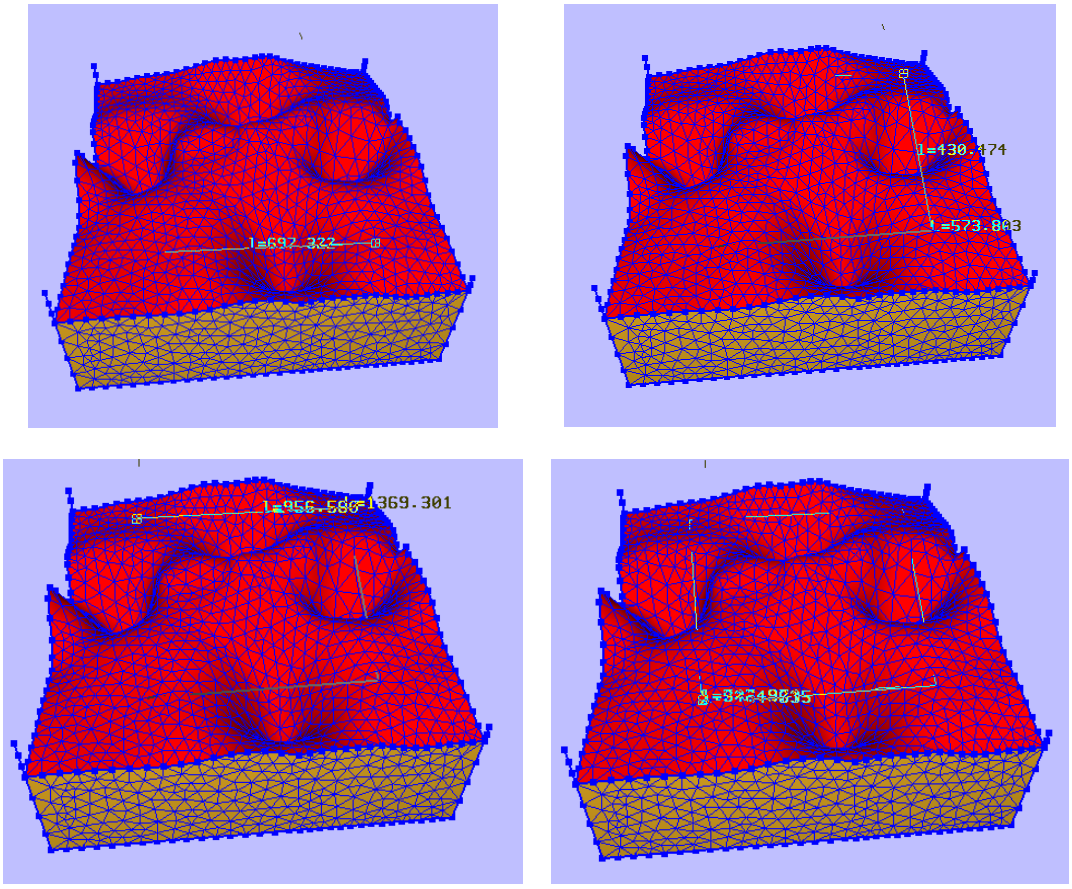


Figure 26: Creating a Curve On Surface

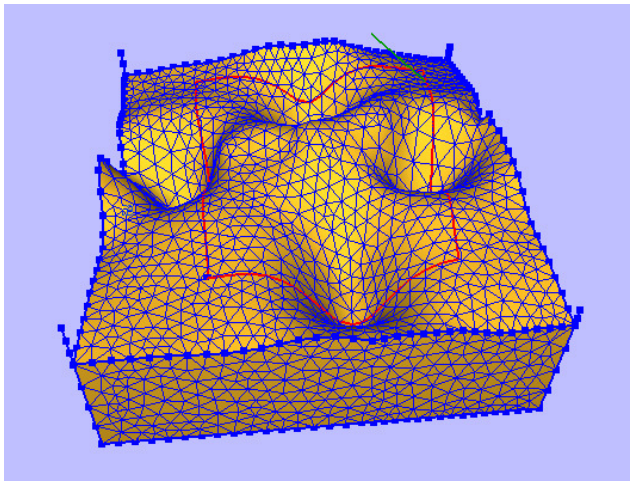


Figure 27: Curve On Surface attracted to the surface

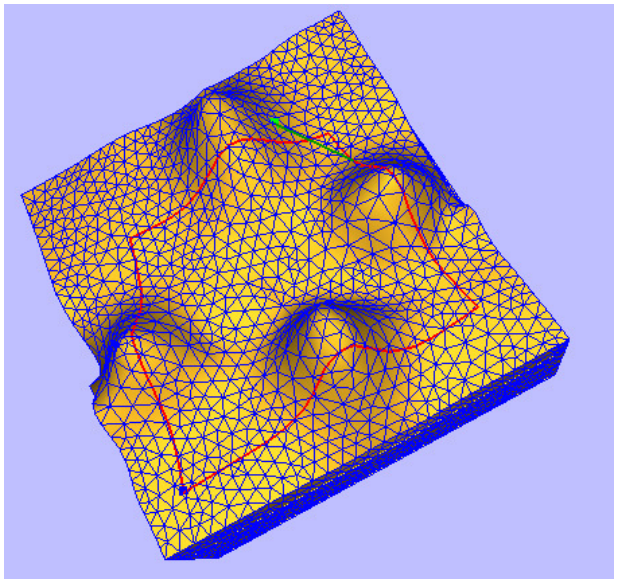


Figure 28: Curve On Surface attracted to peaks

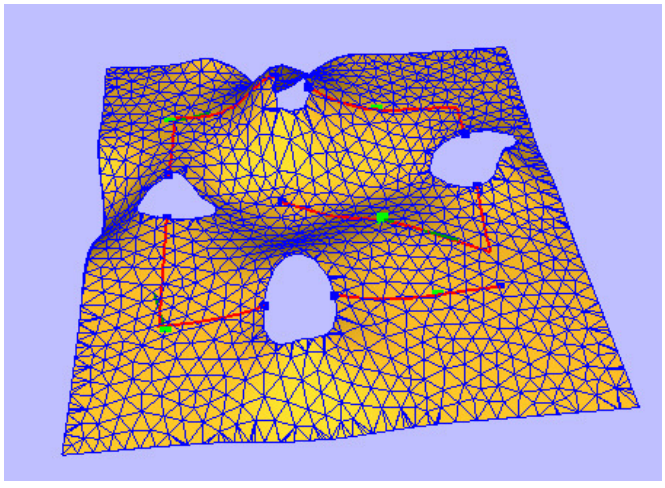


Figure 29: Curve On Surface bypassing holes

6 SURFACE-SURFACE INTERSECTION

Surface-Surface Intersection involves intersecting triangles. To keep our strategy of walking by edges, we will intersect segments with triangles. Segura and Feito(1998, 2001) have developed a fast algorithm for testing them which uses signed volumes. However, the point has to be calculated using a classical segment-plane intersection algorithm. Jiménez et al(2009), using barycentric coordinates and signed volumes have built an algorithm which calculates segment-triangle intersection points even faster than Segura and Feito(1998, 2001). A curious fact is that Jiménez et al(2009), in search of robustness, have minimized the number of arithmetic operations that would be performed in the determinants calculations. Besides, they have avoided divisions, which is a relatively slow and imprecise operation, and used a tolerance range in their comparisons.

Definitely, exact arithmetic could be introduced in these algorithms. However, developing a new Robust Segment-Triangle Intersection Algorithm would be very interesting for study purposes. Once we have a Robust Segment-Plane Intersection Algorithm, all we have to do is assure that this intersection point lies inside the triangle borders. So, we will have 3 delimiter planes for each triangle we are intersecting.

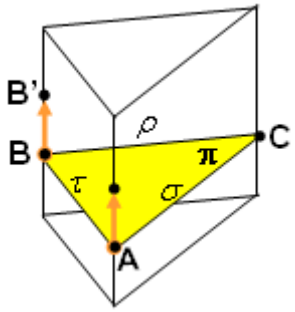


Figure 30: A Triangle Delimiter Planes ρ, σ, and τ

If π is the plane defined by the 3 vertexes of a triangle, we can define the points

$$\begin{aligned} A' &= A + n_\pi \\ B' &= B + n_\pi \end{aligned} \tag{51}$$

Then, our delimiter planes ρ, σ and τ will be defined by the points

$$\rho: \begin{cases} R_0 = B \\ R_1 = B', \\ R_2 = C \end{cases} \quad \sigma: \begin{cases} S_0 = C \\ S_1 = A', \\ S_2 = A \end{cases} \quad \tau: \begin{cases} T_0 = A \\ T_1 = A' \\ T_2 = B \end{cases} \tag{52}$$

Once these points are given counterclockwise, any point D inside these delimiter planes will satisfy:

$$\begin{cases} orient3d(R_0, R_1, R_2, D) \le 0 \\ orient3d(S_0, S_1, S_2, D) \le 0 \\ orient3d(T_0, T_1, T_2, D) \le 0 \end{cases} \tag{53}$$

Bellow, we present our Robust Segment-Triangle Intersection Algorithm.

```
Segment_Triangle_Intersection(e, T):
Build the plane π containing all T vertexes
Using the normal of T, build 3 delimiter planes
If both endpoints of e are over π then
    return false
End If
If both endpoints of e are under π then
    return false
End If

If both endpoints do not belong to π then
    Calculate t_p
    If 0 ≤ t_p ≤ 1 then
        Calculate P
        If P lies between delimiter planes then
            Add P to curve
            return true
        Else
            return false
        End If
    Else
        return false
    End If
End If

If first endpoint belongs to π and
  first endpoint lies between delimiter planes then
    Add first endpoint to curve
    If second endpoint belongs to π and
      second endpoint lies between delimiter planes then
        Add second endpoint to curve
        return true
    End If
End If

If second endpoint belong to π and
  second endpoint lies between delimiter planes then
    Add second endpoint to curve
    return true
End If

return false
```

Figure 31: Segment-Plane Intersection for Curve On Surface

Whenever a triangle edge intersects another triangle, the edges adjacent to this edge may also intersect this triangle. Besides, the triangles adjacent to the intersected triangles probably will intersect. Our search for intersections is based on these facts.

```
Search_Intersections(T_A, T_B):
For each unvisited edge e_A of triangle T_A
   Mark e_A as visited
   If e_A is intersected with T_B then
    Calculate the Intersection Point
    Search for more intersections between T_A and T_B
    Search for more intersections between the triangles adjacent to e_A and T_B
    Search for more intersections between T_A and the triangles adjacent to T_B
    Search for more intersections between the triangles adjacent to T_A and the triangles adjacent to
T_B
     End If
End For

For each unvisited edge e_B of triangle T_B
   Mark e_B as visited
   If e_B is intersected with T_A  then
    Calculate the Intersection Point
    Search for more intersections between T_B and T_A
    Search for more intersections between the triangles adjacent to e_B and T_A
    Search for more intersections between T_B and the triangles adjacent to T_A
    Search for more intersections between the triangles adjacent to T_B and the triangles adjacent to
T_A
```

Figure 32: Surface-Surface Search Intersections

For Surface-Surface Intersections, a good heuristic to start our search for intersections is similar to the used on Surface-Plane Intersection. First, we check all intersected boundary triangles, because two intersecting surfaces may result in a curve which starts and ends at boundary of a surface, which may be the same surface or not. At worst, the peaks of two surfaces may be in intersection and, then, the intersection curve is a closed polyline. So, checking the internal triangles and allowing revisiting is also required.

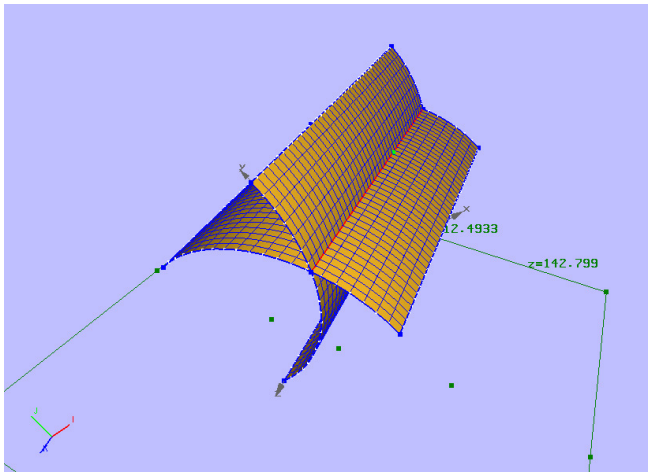


Figure 33: Two Quarter Cylinders Intersecting

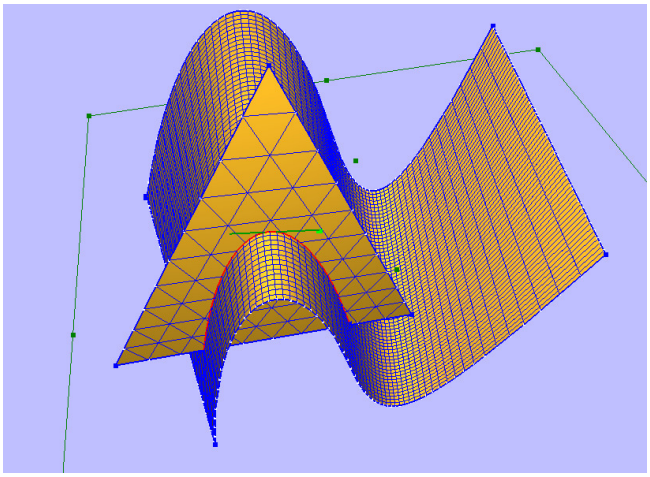


Figure 34: Sigmoidal Surface and a Triangle Intersecting

7 CONCLUSIONS

In this paper we have proposed three algorithms for fast and robust surface intersections. The first algorithm deals with intersections of a plane with a surface; the second algorithm allows inserting a curve on a given surface; whereas the third one is responsible for performing intersections between two surfaces. We show that all the algorithms were adapted in order to allow adaptive precision based fast computations by the direct use of robust and adaptive geometric predicate algorithms. Examples were presented to validate and to demonstrate the capabilities of the proposed algorithms.

REFERENCES

Jiménez, J.J., Segura, R.J., Feito, F.R., A Robust Segment-Triangle Intersection Algorithm for Interference Tests. Efficiency Study. *Computational Geometry Volume 43, Issue 5, Pages474-492,* 2009

Segura, R.J., Feito, F.R., Algorithms to Test Ray-Triangle Intersection. Comparative Study. *Journal of WSCG*, 2001

Segura, R.J., Feito, F.R., An Algorithm for Determining Intersection Segment-Polygon in 3D. *Computer & Graphics*. 1998

Shewchuk, J.R., Adaptive Precision Floating-Point Arithmetic and Fast Robust Geometric Predicates, 1996a

Shewchuk, J.R., Robust Adaptive Floating-Point Geometric Predicates, *Twelfth Annual Symposium on Computational Geometry*, May 1996b