Geometry-Free Polygon Splitting

Sherif Ghali*

Abstract

A polygon splitting algorithm is a combinatorial recipe. The description and the implementation of polygon splitting should not depend on the embedding geometry. Whether a polygon is being split in Euclidean, in spherical, in oriented projective, or in hyperbolic geometry should not be part of the description of the algorithm. The algorithm should be purely combinatorial, or *geometry free*.

The geometry ultimately needs to be specified, and the geometric predicates can only be implemented after specifying the coordinate type and the number type. But the geometry, along with the coordinates and number type in that geometry, remain a late "plug-in", to be added only to the finished algorithm.

We describe a kernel for hyperbolic geometry. Once classes and predicates in that geometry are developed, hyperbolic geometry can be used as a plug-in to polygon splitting alongside other geometries.

We also describe an algorithm for the splitting of a polygon represented using its bounding lines. The use of this dual representation ensures that all predicates are computed directly from input data. This remains the case even if the same polygon is split multiple times, as occurs in BSP tree construction.

1 Introduction

Polygon clipping and splitting algorithms are described in the literature for a specific geometry. An algorithm is described either for Euclidean geometry [10] or for oriented projective geometry [17, 2]. Initially, intersections in oriented projective space were performed by making observations about the homogenizing coordinate, w. As kernels for different geometries were developed, it became better understood that intersection operations can be performed while the coordinates remain invisible [2, 15, 6].

Yet there is no reason for clipping and splitting algorithms not to be designed and implemented as purely combinatorial algorithms. The geometry remains a variable, one that is bound to the algorithm at a late stage during compilation.

The art of geometric computing has been scattered, with computational geometry mainly seeking solutions in Euclidean spaces and with computer graphics seeking ones in oriented projective space. Recent work has shown that geometric algorithms can be made neutral [5, 4]. The same algorithm can be instantiated in either Euclidean or oriented projective geometry. We take at present another step and show that a kernel for hyperbolic geometry can also be defined. We show how a geometry kernel can be a late addition to an algorithm to produce a concrete algorithm in that geometry.

The problem addressed here is polygon splitting—two parts result from the split. If only one of the two parts is needed, the problem is termed polygon clipping instead. Given a splitting algorithm, regardless of whether it is geometry free, one can easily produce a clipping algorithm by removing the algorithm subset that generates the part that is not needed.

1.1 Number-Type, Coordinate, and Dimension Freedom

By liberating an algorithm from its number type, coordinates, dimension, or from geometry, an algorithm becomes number-type free, coordinate free, dimension free, or geometry free, respectively.

Number-type freedom refers to the ability to modify an implementation by changing as little as one program line, to make the implementation operate on one number type or another [13]. Minimal modification is important. Modifying an algorithm to use 'float' instead of 'double', for example, can in general not be performed simply by replacing one string with another. One must also confirm that each instance does indeed represent a coordinate in the geometric system.

Coordinate freedom [11] is as important to writing maintainable geometric systems as number-type freedom. A geometric system is said to be *coordinate free* if coordinate manipulation is restricted. Coordinates are needed in the input and output stages of an algorithm, but the intermediate stages of an algorithm are designed and implemented such that coordinates are not accessed. Aside from the objectives of generality and reuse, coordinate freedom promotes the use of a vectorial language to resolve geometric predicates [5, Chap. 17].

Dimension freedom involves defining a geometric algorithm that can operate in any dimension. The only algorithms that appear to be amenable to dimension freedom at this time are BSP algorithms.

*shghali@gmail.com

1.2 Geometry Freedom

Geometric freedom proposes to turn a geometric algorithm into a purely combinatorial one [5, Chap. 29]. Figure 1 illustrates a few low-dimensional geometries.

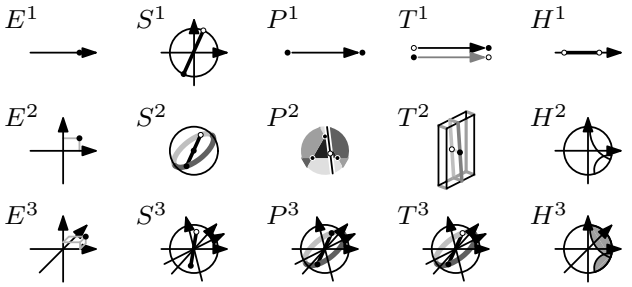

Figure 1: Low-dimensional kernels for Euclidean, spherical, projective, oriented projective, and hyperbolic geometries.

Consider that we have defined classes (datatypes and functions) for objects in each geometry. In the real Euclidean plane $\mathbb{R}E^2$ we will define classes for a point, a line, and a polygon—called, respectively, Point_E2, Line_E2, and Polygon_E2. Likewise in the real oriented projective plane $\mathbb{R}T^2$ we will define the classes Point_T2, Line_T2, and Polygon_T2, and so on.

Even though in a geometric system we have no need for creating a concrete instance of Euclidean, spherical, or hyperbolic plane geometry, we define a datatype for each geometry [4]. The datatypes remain abstract—no instance is ever created. They serve in acting as a parameter to a combinatorial algorithm. During compilation the generic geometry is replaced by a concrete one, and the resulting implementation is as efficient as one hand-tailored for a particular geometry.

If 'double' is chosen as the number type, the class for 2D Euclidean geometry becomes Geometry_E2<**double**>, that for 2D hyperbolic geometry Geometry_H2<**double**>, and so on.

Computational geometry often uses mapping in general and projection in particular to reduce one problem to another. It is clear that a Euclidean geometry cannot replace a different geometry everywhere, but even if the topology is identical, the mapping may be undesirable. It is possible, for instance, to use stereographic projection to define a bijection between points on the extended complex plane and the Riemann sphere [9]. It then becomes possible to claim that a problem on the sphere can be solved by invoking an algorithm on the complex plane, along with appropriate handling for the ideal point. This may be satisfying in synthetic geometry, but it is not a useful solution from an algebraic or a computing perspective. No numerical precision would be adequate to capture points in proximity of the north pole.

2 Kernel Support for Polygon Splitting

As with any instance of introducing modularity into a software system, one must define the interface between two or more components. In the case of raising the abstraction of polygon splitting, we need to define the classes and the functions provided by the kernel and used by the implementation of polygon splitting.

The following C++ code illustrates the implementation of a 2D Euclidean geometry class. Itself parameterized by a number type NT, the class also acts as a parameter for geometry-free algorithms.

```
template<typename NT>
struct Geometry_E2
{
   typedef NT NumberType;

   typedef Point_E2<NT>  Point;
   typedef Line_E2<NT>  Hyperplane;
   typedef Polygon_E2<NT>  Polytope;
};
```

The code for other geometries is similar. The following code shows a class Geometry_H2 for 2D hyperbolic geometry.

```
template<typename NT>
struct Geometry_H2
{
   typedef NT NumberType;

   typedef Point_H2<NT>  Point;
   typedef Line_H2<NT>  Hyperplane;
   typedef Polygon_H2<NT>  Polytope;
};
```

The hyperbolic geometry kernel represents points by those in the interior of the Poincaré disk, where lines are oriented circles orthogonal to the unit circle [8]. Line intersection results in either no (real) points or in two points. In the first case the lines have no intersection in the hyperbolic plane and in the second they have one intersection. One point will be inside the unit disk and its inversion will be outside. A line joining two points will pass by the two points as well as their inversions. Adopting the Poincaré disk rather than Weierstrass coordinates [3] means that we sacrifice homogeneity, which we leave as a second step.

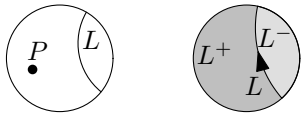

Figure 2: Separability of hyperbolic geometry

The only property of hyperbolic geometry on which polygon splitting depends is separability, illustrated in Figure 2. We say that a geometry is *separable* if the

removal of one hyperplane results in two disjoint sets. Separability is also the property of oriented projective geometry that is not satisfied by classical projective geometry and that makes it necessary to base geometric algorithms on the former. We have yet to identify a class of algorithms that can be naturally defined in classical projective geometry.

Each geometry in turn defines the concrete types for a point, a hyperplane, and a polytope in that geometry using traits [12]. A geometry-free algorithm is then written to use a point, a hyperplane, and a polytope without referring to a concrete type [4].

Traits are simply type mappings. A classical procedure performs mapping between objects. The procedure takes a set of parameters. When called, it evaluates a function and returns an object. Traits extend this notion to types. The 'typedef' statement in the C language already performs this mapping, although in the opposite order of what assignment statements in that language would suggest: the "l-type-value" appears on the right. The combination of type genericity and type mapping meant that traits have found wide applications in generic programming.

The polygon splitting implementation is a function split that is parameterized by the geometry.

```
template<typename Geometry>
void
split(const typename Geometry::Polytope & polytope,
      const typename Geometry::Hyperplane & hyperplane,
      typename Geometry::Polytope & positive_part,
      typename Geometry::Polytope & negative_part);
```

To split in a concrete geometry, it suffices to instantiate the generic function with a concrete geometry.

```
split<Geometry_E2<double> >(...);
split<Geometry_S2<double> >(...);
split<Geometry_H2<double> >(...);
```

Type safety is guaranteed. The function for splitting in one geometry will only accept polytope objects and a hyperplane in that geometry. In this abstraction we refer to polygons using the more general term polytope to facilitate dimension freedom.

Only one predicate function is needed by split: line-point sidedness. Visualization requires a second function: line-line intersection. Neither function is generic with respect to the geometry. To compile split in a given geometry, It is necessary to ensure that a concrete intersection function and sidedness predicate are available for that geometry. The declarations in the case of Euclidean geometry, for instance, are:

```
template<typename NT>
Oriented_side
oriented_side(const Line_E2<NT>& L1,
              const Line_E2<NT>& L2,
              const Line_E2<NT>& L3);

template<typename NT>
Point_E2<NT>
intersection(const Line_E2<NT> & L1,
             const Line_E2<NT> & L2);
```

3 Geometry-Free Polygon Splitting

3.1 A Dual Representation for Polygons

Our main application for polygon splitting is the computation of Boolean operations. A polygon is recursively split by the partitioning hyperplanes in a binary tree. When a fragment of the polygon reaches a leaf node, the Boolean operation is evaluated and the fragment is either discarded or used to construct a subtree at a leaf [18].

Suppose that the operation we wish to compute is Boolean union, and that we have inserted into an initially empty tree the three dark-shaded polygons shown in Figure 3 (a). Our BSP tree will at this time include some leaf node N representing the light-shaded triangle in the center. Suppose that we then insert a fourth polygon defined by the three circular markers and the dashed lines. That polygon will be split by the interior nodes, and all fragments but one will be eliminated as redundant (because they will be already flagged as belonging to the point set). Only the fragment at the center will remain.

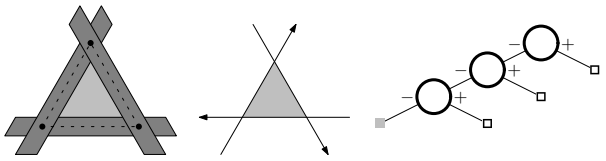

Figure 3: (a) Computing the union of four polygons; (b) the boundary of the fragment remaining of the fourth polygon; (c) the corresponding subtree

The traditional approach is to then construct a binary tree (Figure 3 (c)) to represent the remaining fragment of the fourth polygon (Figure 3 (b)—we use the reverse of the convention of a polygon's orientation to facilitate dimension freedom [5]). That binary tree is attached as a subtree at the leaf node N. Yet this seven-node subtree introduces six nodes that represent empty sets. This is the case in a binary tree whenever the key at an interior node matches the key at one of its ancestors. In this case all three interior nodes of the subtree would be present along the path to the root. Storing nodes that represent the empty set does not breach the BSP tree—the empty sets are convex—but it is not optimal.

The conclusion we make is that the recursive splitting of a polygon must maintain for each edge of the polygon

whether the edge is the result of a cut. Only those edges that are not the result of a cut are used to construct the leaf subtree.

As is now well-understood, it is necessary to use ternary logic for the sidedness predicate. Figure 4 illustrates the issue in the present context. If two polygons have sides that coincide with a splitting line (perhaps because they have already been cut by precisely that line), then the act of folding the coincidence of sidedness with either the positive or the negative sides will result in a zero-area quadrangle.

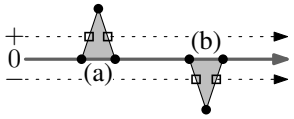

Figure 4: Necessity of handling point-hyperplane incidence: Folding 0 into − means that polygon (a) is unnecessarily split, and likewise for polygon (b) if 0 is folded into +.

But how can we determine reliably whether a given vertex or edge is incident to a splitting line? One approach is to off-load the problem on the number type and assume that we have at our disposal an exotic number type capable of determining without failure the sign of a determinant. Yet a solution is possible that depends on no stronger than the built-in finite precision number types.

The solution we use is to define polygons by their bounding lines rather than their bounding vertices [16]. Sugihara and Iri have also shown how the sidedness predicates can be resolved directly from the coefficients of the (hyper-) planes and without appealing to duality.

In addition, we also store with each bounding line a flag describing whether the line is the result of a split. Storing these flags ensures that the BSP trees we construct contain no nodes representing empty sets.

3.2 Algorithm

Figure 5 shows a new polygon splitter with the following new features. It is generic with respect to the geometry. It avoids slivers by using the input lines to define all new fragments. It is suitable for concave polygons, and it maintains a flag for each polygon boundary to identify whether it is the result of a cut, which makes the resulting fragments suitable for processing in BSP trees.

The main operation in the graphics pipeline is to clip a polygon six times with the boundary of a cube in oriented projective space. In that setting the hyperplane is three-dimensional, but the object clipped is a two-dimensional polygon lying in 3D.

The present algorithm is not identical to the one that appeared in the monograph [5, Page 263]. That work also represented polygons using their bounding hyperplanes, but vertex coordinates were computed to determine sidedness—an operation that is at present resolved by using hyperplane-based predicates [16].

Both preceding works on polygon clipping [5, 1] avert the construction of new vertices at the clip/split locations. They both do so by outputting hyperplanes instead of the classical method of outputting vertices during each iteration. The atomic test in Bernstein and Fussell's algorithm is based on four boundary edges and three vertices, for a total of 27 cases (each vertex may lie on either side or coincide with the splitting plane). In addition to being purely combinatorial or geometry free, the present algorithm (as well as the previous monograph presentation [5]) iterates instead over three hyperplanes and two vertices while handling 9 cases (three outcomes for each vertex). For completeness, we show the algorithm handling both positive and negative fragments. We also handle the coincidence of the polygon with the splitting plane—a case that can arise in 3D.

The implementation was invoked in spherical, Euclidean, and hyperbolic geometries. Examples of splitting a polygon on the sphere, in the Euclidean plane, and in the hyperbolic plane are shown in Figure 6.

4 Future Work

The trajectory we take is to define algorithms and a usable library for vector computer graphics that complements what can be done in raster computer graphics. Rather than represent an image of a 3D object as a uniformly sampled shading value (a raster image), we wish to capture an image as a planar graph on either a sphere or on a subset of the Euclidean plane. The extension of this work to 3D must proceed while satisfying the following two simultaneous objectives. The present splitting routine must be usable when splitting a 3D polygon embedded in a plane in space. But the splitting function must also robustly handle the case of a 3D polytope, which is necessary for dimension freedom in a BSP tree. A splitting routine has also been implemented for 1D in both spherical geometry and in Euclidean geometry [5]. Even though simple, the need arises in practice for the computation of Boolean operations on regular sets in 1D.

Line clipping and splitting is an equally important problem. In the context of BSP trees we often need to determine *sub-hyperplanes* [5, 1], the subset of a splitting line lying inside a convex region—an instance of line clipping. Figure 7 suggests that it may be possible to use classical duality to combine an implementation for line and polygon splitting [14]. But note that, as illustrated in the figure, the vertices to be addressed under line and polygon splitting are distinct. The vertices addressed under line splitting are the line's intersections

```
Split(Polygon P, Line L)
    returns positive_polygon, negative_polygon: Polygon
    classify (implicit) vertices of P with respect to L
    if no vertex lies in L⁻
    or no vertex lies in L⁺
        for each bounding line of P // Case (a)
            if line coincides with L
                set the corresponding edge flag
        if no vertex lies in L⁻ // Case (b)
            copy P with new edge flags into positive_polygon
        if no vertex lies in L⁺ // Case (c)
            copy P with new edge flags into negative_polygon
        return
    if all vertices of P lie on L // Case (d)
        return
    vector_of_lines positive_lines, negative_lines
    vector_of_flags flags_of_positive_lines, flags_of_negative_lines
    for each bounding edge e of P
        // e.source is implicitly defined by predecessor(e) and e
        // e.target is implicitly defined by e and successor(e)
        if e.source is not in L⁻ and e.target is not in L⁻
            insert e to positive_lines // Case (e)
            insert flag of e to flags_of_positive_lines
            if e.target lies on L // Case (f)
                insert L to positive_lines
                insert true to flags_of_positive_lines
        else if e.source is not in L⁺ and e.target is not in L⁺
            insert e to negative_lines // Case (g)
            insert flag of e to flags_of_negative_lines
            if e.target lies on L
                insert -L to negative_lines // Case (h)
                insert true to flags_of_negative_lines
        else // segment straddles the splitting line; split
            if e.source lies in L⁺ and e.target lies in L⁻
                insert e to positive_lines // Case (i)
                insert flag of e to flags_of_positive_lines
                insert L to positive_lines
                insert true to flags_of_positive_lines
                insert e to negative_lines
                insert flag of e to flags_of_negative_lines
            // The symmetric next case is included for completeness
            if e.source lies in L⁻ and e.target lies in L⁺
                insert e to negative_lines // Case (j)
                insert flag of e.source to flags_of_negative_lines
                insert -L to negative_lines
                insert true to flags_of_negative_lines
                insert e to positive_lines
                insert flag of e to flags_of_positive_lines
    construct positive_polygon from positive_lines and flags_of_positive_lines
    construct negative_polygon from negative_lines and flags_of_negative_lines
```

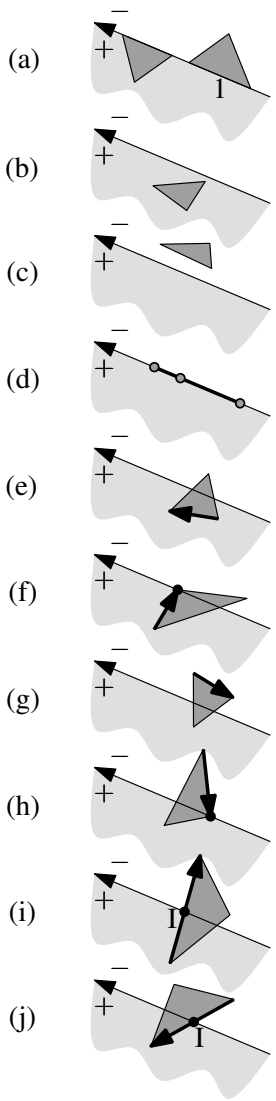

Figure 5: Geometry-free polygon splitting—Cases (f) and (h) in the algorithm handle the incidence of a vertex with the splitting line, case (a) handles the incidence of an edge with the splitting line, and cases (b) and (c) handle the cases when the polygon lies strictly on one side of the splitting line. Cases (f) and (h) are themselves special cases of (e) and (g). Cases (i) and (j) handle a proper (interior) intersection between the polygon's boundary and the splitting line. If, as iterative runs of the algorithm guarantee, the input polygon represents a regular set, case (d) cannot arise in 2D geometry. We include it to be able to handle 3D polygon splitting.

Figure 6: Polygon splitting in Euclidean, spherical, and hyperbolic geometries

with the polygon's bounding lines.

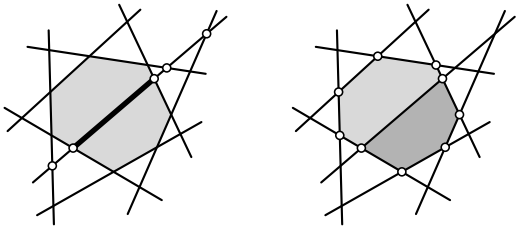

Figure 7: Combining polygon and line clipping

Genericity can also be used in the context of attributes [7]. A polygon's vertex will frequently carry along data such as texture coordinates, which will also need to be clipped along with the geometry. Yet clipping texture coordinates is not as straight-forward as it may seem because of the distinct metrics in each geometry. The appropriate solution is to devise an interpolation module that caters for distances, angles, and areas. Geometry and dimension freedom suggest that the solution should handle interpolants of an arbitrary function, not just linear interpolation, in an arbitrary geometry.

Acknowledgment

I am grateful for the helpful comments made by reviewer #3.

References

[1] G. Bernstein and D. Fussell. Fast, exact, linear booleans. *Comput. Graph. Forum*, 28(5):1269–1278, 2009.

[2] J. Blinn and M. Newell. Clipping using homogeneous coordinates. *Comput. Graph.*, 12(3):245–251, Aug. 1978.

[3] H. Buseman and P. Kelly. *Projective Geometry and Projective Metrics*. Academic Press, 1953.

[4] S. Ghali. Geometry-free geometric computing—towards higher-order genericity through purely combinatorial geometric algorithms. to appear.

[5] S. Ghali. *Introduction to Geometric Computing*. Springer, 2008.

[6] S. Ghali. Sense and sidedness in the graphics pipeline via a passage through a separable space. *The Visual Computer*, 25(4):367–375, Apr. 2009.

[7] P. Heckbert. Generic convex polygon scan conversion and clipping. In A. Glassner, editor, *Graphics Gems I*, pages 84–86. Academic Press, 1990.

[8] M. Henle. *Modern Geometries: Non-Euclidean, Projective, and Discrete*. Prentice-Hall, 2nd edition, 2001.

[9] P. Henrici. *Applied and Computational Complex Analysis*. Wiley, 1974.

[10] Y. Liang and B. Barsky. An analysis and algorithm for polygon clipping. *CACM*, 26(11):868–876, 1983.

[11] S. Mann, N. Litke, and T. DeRose. A coordinate free geometry ADT. Technical Report CS-97-15, University of Waterloo, July 1997.

[12] N. Myers. Traits: A new and useful template technique. *C++ Report*, June 1995.

[13] J. Nievergelt, P. Schorn, M. de Lorenzi, C. Ammann, and A. Brüngger. XYZ : Software for geometric computation. Report 163, ETH, Zürich, July 1991.

[14] V. Skala. A new approach to line and line segment clipping in homogeneous coordinates. *The Visual Computer*, 21:905–914, 2005.

[15] J. Stolfi. *Oriented Projective Geometry: A Framework for Geometric Computations*. Academic Press, 1991.

[16] K. Sugihara and M. Iri. A solid modelling system free from topological inconsistency. *J. Inform. Proc.*, 12(4):380–393, 1989.

[17] I. Sutherland and G. Hodgman. Reentrant polygon clipping. *CACM*, 17:32–42, 1974.

[18] W. Thibault and B. Naylor. Set operations on polyhedra using binary space partitioning trees. *Comput. Graph.*, 21(4):153–162, 1987.