GeoLib Polygon Operations

Document version 1.0

www.geolib.co.uk
support@geolib.co.uk

Contents

Introduction

GeoLib has 4 main types of polygons, which result from them either having curved lines or straight lines only, and containing holes or not. The classes and their base classes which represent these are as shown below.

Basic functions

C2DPolygon

This class represents a polygon with straight edges and no holes. It can be non-convex but it is assumed to be simple i.e. not self-intersecting. This type of polygon can be created from a set of points, which will be ordered in a clockwise orientation as follows:

```
C2DPointSet pts;

pts.AddCopy( 1.0, 1.0 );
pts.AddCopy( 1.0, 2.0 );
pts.AddCopy( 2.0, 2.0 );
pts.AddCopy( 2.0, 1.0);

C2DPolygon Poly1(pts);
bool bTest = Poly1.IsClockwise();
```

This type of polygon can also be created from an array of points as follows:

```
C2DPoint pt[3] = {  C2DPoint(2.4, 3.45),
                    C2DPoint(3.4, 7.45),
                    C2DPoint(8.4, 34.45)};

C2DPolygon Poly1(pt, 3);
```

If the order of the points is unknown, they can be automatically reordered upon creation. The reordering is done to minimise the perimeter and eliminate crossing lines.

```
C2DPointSet pts;

pts.AddCopy( 1.0, 1.0 );
pts.AddCopy( 2.0, 2.0 );    // Will cross
pts.AddCopy( 1.0, 2.0 );
pts.AddCopy( 2.0, 1.0);     // Will cross

C2DPolygon Poly1(pts, true);    // Flag to indicate reorder
bool bTest = Poly1.HasCrossingLines();
```

The above code will reorder the polygon to eliminate the crossing line and will result in a simple square.

The following code shows how to call various other simple functions on the polygon.

```
C2DPoint pt1(1.5, 1.5);
bool bTest = Poly1.Contains( pt1 ); // Point inside?

C2DPoint pt2(1.8, 1.8);
C2DLine line1(pt1, pt2);
bTest = Poly1.Contains( line1 );    // Line inside?

// create an arc of radius 0.5 with centre to right of line
// and the curve to the left of the line (true, false);
C2DArc arc1(line1, 0.5, true, false);
bTest = Poly1.Contains( arc1 );     // Arc inside?

C2DPolygon Poly2(Poly1);            // Copy of poly1
C2DPoint ptCentre = Poly2.GetCentroid(); // Find the centre
Poly2.Grow( 0.5, ptCentre);     // Shrink Poly2 about centre
bTest = Poly1.Contains( Poly2 );    // Poly inside?

bTest = Poly1.Crosses(line1);       // Crosses line
bTest = Poly1.Crosses(arc1);        // Crosses arc

double dDist = Poly1.Distance(pt1); // Distance to point
dDist = Poly1.Distance(line1);      // Distance to line
dDist = Poly1.Distance(arc1);       // Distance to arc
dDist = Poly1.Distance(Poly2);      // Distance to poly
bTest = Poly1.IsWithinDistance( pt1, 2.5); // Point proximity
```

```
double dArea = Poly1.GetArea();      // Gets the area
dArea = Poly1.GetAreaSigned();       // -ve if clockwise

C2DCircle circle1;
Poly1.GetBoundingCircle(circle1);    // Bounding circle

C2DRect rect1;
Poly1.GetBoundingRect(rect1);        // Bounding rect

double dPerimeter = Poly1.GetPerimeter();   // Perimeter

bTest = Poly1.IsConvex();            // Is convex?

Poly1.Move(C2DVector(5,6));          // Move the polygon

CInterval Int1;
Poly1.Project(C2DVector(5, 6), Int1 ); // Vector projection
Poly1.Project(line1, Int1 );             // Line projection

Poly1.Reflect( pt1 );                // Reflect through point
Poly1.Reflect( line1 );              // Reflect through line
```

C2DPolyArc

This class inherits from the C2DPolyBase class and provides an interface to it so that curved lines can be added to the polygon. The polygon is created in a different way to the simple straight lined polygon, as each point has to be added with consideration to the curve in the line between it and the proceeding point. It is still perfectly acceptable to add straight lines. To create a polygon, the start point is set then subsequent points are added through the “LineTo” function. If the line is to be curved, the radius of the curve must be provided, whether the curve is to the right of the 2 points, and whether the arc’s centre is to the right of the 2 points. Finally the polygon is closed with either a straight line or an arc. Note that the final point you add must not be the starting point as this is done for you when the polygon is closed.

```
// New PolyArc
C2DPolyArc* pPoly = new C2DPolyArc;
// Set the start point
pPoly->SetStartPoint( C2DPoint( 50, 50));
// Curved line to the next point, radius 100
// Arc to the right but arc centre to left
pPoly->LineTo( C2DPoint( 50, 100), 100, true, false);
// straight line
pPoly->LineTo(C2DPoint( 100, 100));
// Back to start. No point provided.
pPoly->Close();
```

See Table 1 for a full list of functions for this and other polygon classes.

C2DPolyBase

This class forms the base for both the C2DPolygon and the C2DPolyArc and most of the functionality is contained within this. At its core is a set of pointers to the abstract class C2DLineBase. In other words, at the core of GeoLib, a polygon is just a set of connected lines of some sort. GeoLib currently supports straight and arced lines but there is no reason why other lines cannot be used within the polygon base class as long as they can inherit from the abstract line base class. This base class can be used directly if required although in most cases one of the inherited types would be more appropriate.

Holed Polygons

Holed polygons are simply polygons with holes in them and are managed through 3 similar classes; C2DHoledPolyBase, C2DHoledPolygon and C2DHoledPolyArc. The base class contains most of the functionality with the other 2 mainly providing an interface to it. This class has a pointer to a rim and an array of pointers to holes, all of which are held as C2DPolyBase pointers. Through the interface classes C2DHoledPolygon and C2DHoledPolyArc, the user can ensure that, for example, only straight lined polygons are holes or rims, although direct use of the base class and mixed types is also fine. The following is an example of how they can be used. It is important to note that holes and rims can be set directly as pointers or as copies or those provided. ***If pointers are handed over then they will be deleted when the holed polygon is deleted so it is wrong to hand over a pointer to a polygon created on the stack.***

```
// Create new polygon on the heap
C2DPolygon* pPoly1 = new C2DPolygon;
// Give it some pts
pPoly1->Create( pts );
// Create a new Holed Poly on the stack
// in this example
C2DHoledPolygon HoledPoly;
// Set the rim directly and do NOT
// delete the pointer pPoly1
HoledPoly.SetRimDirect( pPoly1 );
// Create a new polygon on the stack
C2DPolygon Poly2(*pPoly1);
Poly2.Grow( 0.5, Poly2.GetCentroid());
// Add a hole as a copy of polygon 2
HoledPoly.AddHole( Poly2 );
// Add a new polygon directly
HoledPoly.AddHoleDirect( new C2DPolygon );
```

As mentioned, the base class can be used as shown in the example below.

```
// Base class this time
C2DHoledPolyBase HoledPoly;
// Straight line rim
HoledPoly.SetRimDirect( pPoly1 );
// Hole with arcs.
HoledPoly.AddHole( pPolyArc );
```

No.	Function	C2DPolyBase	C2DPolygon	C2DPolyArc	C2DholedPolyBase	C2DholedPolygon	C2DholedPolyArc
1	Contains(C2DPoint)	✓	✓	✓	✓	✓	✓
2	Contains(C2DLineBase)	✓	✓	✓	✓	✓	✓
3	Contains(C2DPolyBase)	✓	✓	✓	✓	✓	✓
4	Contains(C2DHoledPolyBase)	✓	✓	✓	✓	✓	✓
5	HasCrossingLines	✓	✓	✓	✓	✓	✓
6	Distance(C2DPoint)	✓	✓	✓	✓	✓	✓
7	Distance(C2DLineBase)	✓	✓	✓	✓	✓	✓
8	Distance(C2DPolyBase)	✓	✓	✓	✓	✓	✓
9	IsWithinDistance(C2DPoint)	✓	✓	✓	✓	✓	✓
10	GetBoundingRect	✓	✓	✓	✓	✓	✓
11	GetPerimeter	✓	✓	✓	✓	✓	✓
12	Draw(Outline)	✓	✓	✓	✓	✓	✓
13	Draw(Filled)	✓	✓	✓	✓	✓	✓
14	Move	✓	✓	✓	✓	✓	✓
15	RotateToRight	✓	✓	✓	✓	✓	✓
16	Grow	✓	✓	✓	✓	✓	✓
17	Reflect(though point)	✓	✓	✓	✓	✓	✓
18	Reflect(through line)	✓	✓	✓	✓	✓	✓
19	Crosses(C2DLineBase)	✓	✓	✓	✓	✓	✓
20	Crosses(C2DPolyBase)	✓	✓	✓	✓	✓	✓
21	Overlaps(C2DPolyBase)	✓	✓	✓	✓	✓	✓
22	Overlaps(C2DHoledPolyBase)	✓	✓	✓	✓	✓	✓
23	Overlaps(returns avoidance vector)		✓				
24	Avoid(C2DPolygon)		✓				
25	SnapToGrid	✓	✓	✓	✓	✓	✓
26	GetNonOverlaps	✓	✓	✓	✓	✓	✓
27	GetUnions	✓	✓	✓	✓	✓	✓
28	GetOverlaps	✓	✓	✓	✓	✓	✓
29	GetBoolean	✓	✓	✓	✓	✓	✓
30	Project(on line)	✓	✓	✓	✓	✓	✓
31	Project(on vector)	✓	✓	✓	✓	✓	✓
32	CreateConvexHull		✓				
33	CreateMorph		✓				
34	CreateRegular		✓				
35	CreateRandom		✓	✓			
36	CreateConvexSubAreas		✓				
37	GetConvexSubAreas		✓				
38	IsConvex		✓				
39	IsClockwise		✓				
40	GetCentroid		✓	✓		✓	✓
41	GetArea		✓	✓		✓	✓
42	GetAreaSigned		✓				
43	GetBoundingCircle		✓				

Table 1 Polygon function summary

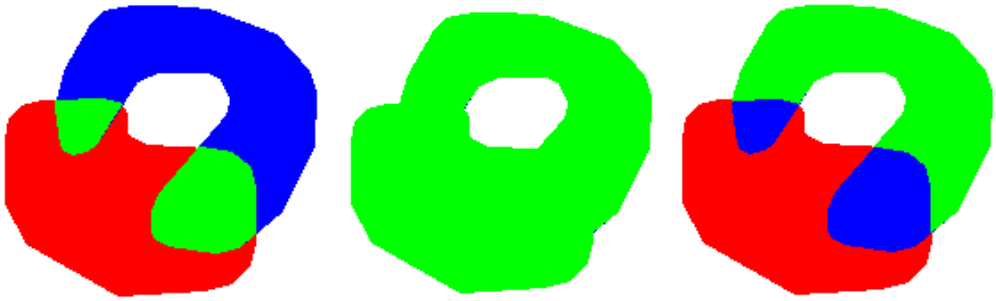
Table 1 shows the functions that each type of polygon supports. Some functions not supported by the holed polygons are identical to those for the rim of the polygon. Holed polygons are assumed to have contained holes that do not intersect. Validation checks for all polygons can be called to ensure this.

The “Overlaps” function for the C2DPolygon can be used to return true if the polygon overlaps another and also return the smallest vector required to move the polygon off the other one. For concave polygons, convex sub areas need to be created first for an accurate result; otherwise the result is for the corresponding convex hull.

Polygon Boolean Operations

Basic function calls

All types of polygons are capable of performing Boolean Operations on all others as this functionality is held within the base class C2DPolyBase. These functions are used to find the overlap, union or difference (non-overlap) between 2 polygons as illustrated.



The code required to do this is shown below.

```
C2DHoledPolygonSet PolySet;          // Create a set of polys
Poly1.GetUnion( Poly2, PolySet);
bool bTest = PolySet.size() == 1;    // Will result in 0 or 1
```

In all cases, the set of polygons to receive the result is a set of holed polygons even though it is impossible in some cases to output polygons with holes. In this case the result is a set of holed polygons which simply don’t have any holes. Other functions are “GetOverlaps” and “GetNonOverlaps” and work in exactly the same way. In all cases, if there is no overlap between the 2 polygons, no polygon is added to the set. For example, the union of 2 non-intersecting polygons is both of them but nothing is added to the set. Should the result still be required, the following code could be added.

```
// Add copies of the 2 polygons to the set
// Will convert to holed polygons automatically
if (PolySet.size() == 0)
{
    PolySet << new C2DPolygon(Poly1);
    PolySet << new C2DPolygon(Poly2);
}
```

Time

At the heart of the Boolean operations is the functionality to compute, very quickly, the intersections between 2 sets of lines. The function that does this is held within the C2DLineBaseSet class and the calling procedure is shown below.

```
C2DLineBaseSet Lines1;  // set 1
C2DLineBaseSet Lines2;  // set 2
IndexSet Indexes1;      // Intersection indexes
IndexSet Indexes2;      // Intersection indexes
C2DPointSet Pts1;       // Intersection points
// The function call
Lines1.GetIntersections( Lines2, &Pts1, &Indexes1, &Indexes2);
```

This function is **O(n log(n + k))**, where n is the number of lines in the sets and k is the number if intersections in the x-axis. This, together with extensive use of bounding boxes makes execution time very quick.

GeoLib also includes an extremely efficient function to unify multiple polygons. This time taken to do this is dependant upon the time to unify 2 and also the order in which the polygons are combined.

```
C2DHoledPolygonSet Polys1;
Polys1.AddCopy( Poly1 );    // Add polygons
Polys1.AddCopy( Poly2 );    // Add polygons
Polys1.UnifyProgressive();  // Quick method
```

Degenerate Handling

Polygon Boolean operations can be performed on what are known as degenerate cases with potentially disastrous results. Typical examples of this are when a point from one polygon coincides exactly with that of another, when a line crosses a point or when a line overlaps another line. The reason these cases cause problems for computers is that the execution of polygon Boolean operations relies on the detection of intersections between polygons. In these cases, it is hard to determine whether there is an intersection or not.

There are 2 general ways of dealing with this problem; one is to try to detect them and to deal with every possibility (of which there are many), another is to avoid the problem completely by ensuring it never happens (or almost never happens). GeoLib takes the second approach and within this there are 2 optional ways of handling degenerates. The fist uses a random permutation to move one polygon so that

coincident points and lines are extremely unlikely. The second is similar but uses the concept of a grid with one polygon having points on the grid and another off the grid. Both methods do involve a very slight distortion of the result but this is minute and the grid-based technique has the advantage that it is known and manageable. In both cases, although extremely unlikely, problems can occur in which case they are reported so that the operation could be repeated with different settings.

Point Equality

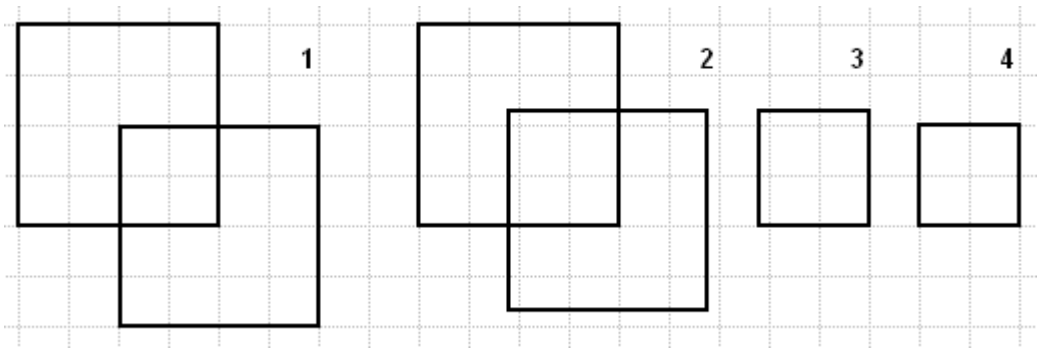
In order to understand how GeoLib handles degenerates, it is first necessary to understand what make 2 points equal. GeoLib uses double precision floating point numbers to represent the x and y values for each point. In order to get around computer rounding errors, 2 points are considered equal when they are very close to each other. How close they need to be to each other also depends upon how close they are to the origin (0, 0). The exact amount is 0.0000000001 (equality tolerance) multiplied by the x or y value. For example, the point 100, 200 has a box around it which is 0.00000002 and 0.00000004 wide and high respectively. All other points within this box are considered equal. To put this into perspective, the proximity is about 0.0001 millimetres in one kilometre.

Random Perturbation

This method simple takes a copy of one of the polygons and moves it a very small amount before calculating the Boolean operation. The movement is calculated to be up to 100 times more than the maximum possible equality value of a given polygon. For example, the 1 kilometre sized polygon would be moved up to 0.01 millimetres.

Grid Based

This method is very similar in principle to the random perturbation method but is based on a conceptual grid. All geometric entities, including polygons can be "Snapped" to the grid, which means that all points are placed on vertices of the grid. This involves minor movement to the shape is allows it to be managed. For example, if dealing in kilometres, the grid can be set to 0.01 millimetres and the accuracy known in advance. Boolean operations using grid methods involve snapping polygons (or copies of them) to the grid, moving one of them off the grid by an amount calculated to minimise problems, performing the operation and snapping the result back to the grid.



Intersection of 2 Squares using grid based degenerate handling

The choice of Grid size is important as if it is too small, the chances of coincident lines and points becomes high. Generally though, it should be possible to set the grid

to a very small value. To help manage the grid, a class is provided called CGrid that allows the user to set the grid value and also help find a minimum recommended setting given a bounding rectangle for the operation to be carried out.

There are 3 grid settings to use when performing Boolean operations as follows:

- Dynamic grid where the grid is automatically set to the minimum recommended setting.
- Predefined grid where the user has already set a desired grid size.
- Pre defined grid pre snapped polygons, where, in addition, the polygons in question are already snapped to the grid.

The following shows code examples for this.

```
C2DHoledPolygonSet PolySet;     // To recieve the result
// Get the union with random perturbation
Poly1.GetUnion( Poly2, PolySet, CGrid::RandomPerturbation );
// Clear the set
PolySet.DeleteAll();
// Get the union with Dynamic Grid
Poly1.GetUnion( Poly2, PolySet, CGrid::DynamicGrid );
// Clear the set
PolySet.DeleteAll();
// Set the grid manually
CGrid::SetGridSize( 0.0001 );
// Get the union with Pre-defined Grid
Poly1.GetUnion( Poly2, PolySet, CGrid::PreDefinedGrid );
// Clear the set
PolySet.DeleteAll();
// Snap both to the grid
Poly1.SnapToGrid();
Poly2.SnapToGrid();
// Get the union with Pre-defined Grid
Poly1.GetUnion( Poly2, PolySet, CGrid::PreDefinedGridPreSnapped );
```

As mentioned, there is still a very small chance of errors occurring but this can be checked for using the following code.

```
CGrid::ResetDegenerateErrors();
// Boolean operation of some sort
if ( CGrid::GetDegenerateErrors() != 0 )
{
    // Handle the error or change settings
    // and repeat.
}
```

Coding Concepts

All geometric entities are derived from the class C2Dbase, an abstract class with pure virtual functions such as Move that all geometric entities must override.

At the heart of all polygon functionality is the C2DPolyBase, which contains a collection of lines, their corresponding bounding rectangles and the bounding rectangle for the whole polygon. The extensive use of bounding rectangles is one reason why GeoLib is so fast.

The lines that form the polygon are held as pointers to the abstract base class C2DLineBase. This allows for a great deal of flexibility to create other types of polygons. In other words, GeoLib's concept of a polygon is just that it is a set of ordered lines of some sort that are joined up. The set of functions that a line must have in order to derive from the C2DLineBase are as follows:

- "Crosses" – find the intersection of itself with any other type of line.
- "Distance" – return the minimum distance from itself to a point or to another line.
- "GetPointFrom" – return the starting point.
- "GetPointTo" – return the end point.
- "GetLength" – return its length.
- "Reverse Direction" – reverses its direction.
- "GetSubLines" – given a set of points on the line, return the set of sub lines given by breaking the line on those points.