**Slope Stability**

Problem: **Find intersection of two polygons (using Weiler-Atherton algorithm)**

Outcome: **Get** ***ordered* sequence of nodes of polygon or polygons that is/are result of intersection of big polygon with smaller polygons (soil layers).**

1. Problem was narrowed down to finding intersection of big **convex** polygon and arbitrary (concave or convex) polygons (soil layers)

To achieve our goal I implemented **Weiler-Atherton algorithm**, which allows as to get intersection of big convex polygon (representing Failure surface) and an arbitrary soil polygon. This algorithm allows to get as an output multiple polygons if Soil polygon intersects in 2 or more different areas :

https://en.wikipedia.org/wiki/Weiler%E2%80%93Atherton\_clipping\_algorithm

This Algorithm is also described in first two pages of “Computer Graphics” by Professor Richard Eckert (3\_17\_04\_W-A-clip\_polycurves.pdf):

1. **To implement Weiler-Atherton algorithm I use three main variable structures: node, edge (Starting node and ending node), and Polygon (list of nodes and list of edges)**

*struct Node {*

*double x; //x coordinate*

*double y; // y coordinate*

*bool intersection; //is intersection point with another polygon or no*

*bool inward; //is inward intersection or no(only for intersections)*

*int insideClipPoly; //is point inside concave Polygon or not (1-inside, 0 -lies on side, -1 outside)*

*int index; //index of intersection node*

*bool visited; // indicates if intersection node was visited during tracing*

*};*

*//type Edge*

*struct Edge {*

*Node start;*

*Node end;*

*};*

*//type Polygon (consists of list of nodes and list of edges)*

*struct Polygon {*

*std::list<Node> polyNodes;*

*std::list<Edge> polyEdges;*

*};*

1. **There are many auxiliary functions that program uses during its run.**

i)Following functions print out Node and Edge:

*//function "printNode" prints out node*

*void printNode (Node a)*

*{*

*std::cout<<" ("<<a.x<<", "<<a.y<<") is intersection:"<<a.intersection<<" Is inward:"<<a.inward<<" Is inside ClipPolygon:"<<a.insideClipPoly<<" Index:"<<a.index<<" Is Visited:"<<a.visited<<"\n";*

*return;*

*}*

*//Function "printEdge" prints out edge*

*void printEdge (Edge a)*

*{*

*std::cout<<"Start of edge";*

*printNode(a.start);*

*std::cout<<"End of edge";*

*printNode(a.end);*

*return;*

*}*

ii)function “printPolygon” prints out all edges and function function “printPolygonNodes” prints out all nodes in column along with node information.

*//Function "printPolygon" prints edges using "printEdge function"*

*void printPolygon (Polygon poly)*

*{*

*for\_each (poly.polyEdges.begin(), poly.polyEdges.end(), printEdge);*

*std::cout<<"\n";*

*return;*

*}*

*//Function "printPolygonNodes" prints all nodes and also intersection points along the edges*

*void printPolygonNodes (Polygon poly)*

*{*

*for\_each (poly.polyNodes.begin(), poly.polyNodes.end(), printNode);*

*std::cout<<"\n";*

*return;*

*}*

iii) function ”distTwoPoints” calculates distance between two nodes based on regular formula using coordinates:

//function distTwoPoints returns distance between two nodes

double distTwoPoints (Node a, Node b)

{

double dist = sqrt( (b.y-a.y)\*(b.y-a.y) + (b.x-a.x)\*(b.x-a.x) );

return dist;

}

iiii) function “angle between vectors” returns angle between two edges represented as vectors. Return in radians value is calculated through dot product formula <https://en.wikipedia.org/wiki/Dot_product>, and cases when angle is equal 0 or Pi are handled separately:

//between two vectors (represented as edges) and returns angle between two vectors

double angleBetweenVector(Edge firstEdge, Edge secondEdge)

{

//calculating coordinates of first vector(edge)

double firstX = firstEdge.end.x-firstEdge.start.x;

double firstY = firstEdge.end.y-firstEdge.start.y;

//calculating coordinates of second vector (edge)

double secondX = secondEdge.end.x - secondEdge.start.x;

double secondY = secondEdge.end.y - secondEdge.start.y;

//calculating dot product of two vectors

double dotProduct = firstX\*secondX + firstY\*secondY;

//calculating lengths of both vectors

double firstLength = sqrt(firstX\*firstX+firstY\*firstY);

double secondLenght = sqrt(secondX\*secondX+secondY\*secondY);

//checking if dotProduct is almost equal to firstLength\*secondLenght and returning angle zero

if ( fabs(dotProduct - (firstLength\*secondLenght))<MINDouble)

{

std::cout<<"\n angle almost zero \n";

return 0.0;

}

//checking if dotProduct is almost equal to (-firstLength\*secondLenght) and returning angle PI

if ( fabs(dotProduct + (firstLength\*secondLenght))<MINDouble)

{

std::cout<<"\n angle almost 180 degree (PI) \n";

return PI;

}

//checking if dotProduct is by absolute value bigger than firstLength\*secondLenght

//that means cos is bigger than 1 by abs value - EXIT with ERROR

if ( fabs(dotProduct + (firstLength\*secondLenght))<MINDouble)

{std::cout<<"\n absolute value of cos greater than 1 detected in vertice to vertice angle test - ERROR\n"; exit (EXIT\_FAILURE);}

//calculating and returning angle in radians between two vectors

double angleRad = acos(dotProduct / (firstLength\*secondLenght));

return angleRad;

}

iiiii) function “halfPlaneOrient” is auxiliary function for determining of polygons winding (clockwise or counter clockwise). Function returns -1,0,1 depending on where point x is from line formed by AB (left side, on line or right side). This is determined from vector orientation formula: <http://mathworld.wolfram.com/VectorOrientation.html>

//function halfPlaneOrient returns -1 if point x is on left side of line ab (Vector ab)

//returns 0 if x belongs line ab and returns 1 if x is on right side of line ab (Vector ab)

int halfPlaneOrient (Node a, Node b, Node x)

{

double orient = (x.x - a.x)\*(b.y-a.y) - (x.y-a.y)\*(b.x-a.x);

if (orient >0) return 1;

else if (orient < 0) return -1;

else return 0;

}

And function “vectorOrient” determines if vectors formed by 4 nodes form angle less than 180 degree or more than 180 degree (1 less than 180, -1 more than 180, 0 parallel). We use same vector orientation formula: <http://mathworld.wolfram.com/VectorOrientation.html>

//function "vectorOrient" returns 1 if angle between vectors formed by points a1,b1 and points a2,b2 is from 0 to 180 degree

//function returns 0 if vectors are parallel

//function returns -1 if vectors form angle from 180 to 360 degree

int vectorOrient (Node a1, Node b1, Node a2, Node b2)

{

double orient = (b1.x - a1.x)\*(b2.y-a2.y) - (b1.y-a1.y)\*(b2.x-a2.x);

if (orient >0) return -1;

else if (orient < 0) return 1;

else return 0;

}

iiiiii) function “windingTestClipPolygon” checks winding of clip (Failure Plane) polygon based on orientation of following node in respect of previous edge. It returnes (1 – clockwise, -1 - counterclockwise). It also gives ERROR message if Clip Polygon is Concave or has collinear edges

//Function "windingTestClipPolygon" returns polygon winding and checks polygon for concavity

//if polygon is not convex it exits with error message "not convex"

//if polygon has collinear sides it exits with error message

//if polygon is convex it returns +1 if polygon is winded Clockwise

// returns -1 if polygon is winded CounterClockwise

int windingTestClipPolygon(Polygon &poly)

{

int halfPlaneSign = 0, halfPlaneSignNew = 0; //initial value of winding numbers

Node startNode = poly.polyNodes.back(); //stastNode is beginning of current side of polygon - last node in polygon list

Node endNode, nextNode; //endNode is end of current side of polygon, nextNode - next node after current side

for (std::list<Node>::iterator it=poly.polyNodes.begin(); it != poly.polyNodes.end(); ++it)

{

endNode = \*it;

if ( std::next(it,1) == poly.polyNodes.end() ) nextNode = poly.polyNodes.front(); //if nextNode(node after current) is beyond last element in list - then nextNode is first element

else nextNode = \*(std::next(it));

halfPlaneSignNew = halfPlaneOrient( startNode, endNode, nextNode); //calculating sign of current winding

if (halfPlaneSignNew == 0 )

{std::cout<<"\n collinear nodes detected - ERROR\n"; exit (EXIT\_FAILURE); } // Collinear nodes detected

else if (halfPlaneSign\*halfPlaneSignNew < 0)

{std::cout<<"\n Polygon is not convex - ERROR\n"; exit (EXIT\_FAILURE); } // Non-convex polygon detected, product of two heighboring windings less than 0

//winding (angles between neighbor sides) are same sign - switch to next side

halfPlaneSign = halfPlaneSignNew;

startNode = endNode;

}

return halfPlaneSign;

}

function “windingTestSoilPolygon” checks winding of soil polygon. It returns (1 – clockwise, -1 - counterclockwise). Soil polygon accepted to be concave, but collinear edges will cause ERROR message:

//Function "windingTestSoilPolygon" returns polygon winding

//if polygon has collinear sides it exits with error message

//it returns +1 if polygon is winded CounterClockwise

//returns -1 if polygon is winded Clockwise

int windingTestSoilPolygon(Polygon &poly)

{

int halfPlaneSign = 0, halfPlaneSignAcc = 0; //initial value of winding numbers

Node startNode = poly.polyNodes.back(); //stastNode is beginning of current side of polygon - last node in polygon list

Node endNode, nextNode; //endNode is end of current side of polygon, nextNode - next node after current side

for (std::list<Node>::iterator it=poly.polyNodes.begin(); it != poly.polyNodes.end(); ++it)

{

endNode = \*it;

if ( std::next(it,1) == poly.polyNodes.end() ) nextNode = poly.polyNodes.front(); //if nextNode(node after current) is beyond last element in list - then nextNode is first element

else nextNode = \*(std::next(it));

halfPlaneSign = halfPlaneOrient( startNode, endNode, nextNode); //calculating sign of current winding

halfPlaneSignAcc += halfPlaneSign; //accumulating signs of winding in variable halfPlaneSgnAcc

if (halfPlaneSign == 0 )

{std::cout<<"\n collinear nodes detected - ERROR\n"; exit (EXIT\_FAILURE); } // Collinear nodes detected

startNode = endNode; //switch to next node

}

if (halfPlaneSignAcc<0)

{

return -1; //return -1 if winding counter clockwise

}

else if (halfPlaneSignAcc>0)

{

return 1;// return 1 if winding clockwise

}

if (halfPlaneSignAcc=0 ) {std::cout<<"\n soil pollygon is not accepted shape (too many concave areas) - ERROR\n"; exit (EXIT\_FAILURE); }

}

1. **Following function are called directly while performing step-by-step execution of Weiler-Atherton algorithm**

i) Function “WindingReverse” converts order of nodes from counterclockwise to clockwise, because Weiler-Atherton algorithm accepts only clockwise winded polygons. It also reverts order of nodes and order of start-end pair in node!

//Function "windingReverse" inverts order of points and edges (from counterclockwise to clockwise, or opposite)

void windingReverse(Polygon &poly)

{

poly.polyNodes.reverse(); //reverse nodes

poly.polyEdges.reverse(); //reverse edges

//swap nodes in edges

for (std::list<Edge>::iterator it=poly.polyEdges.begin(); it != poly.polyEdges.end(); ++it)

{

Node tempNode;

tempNode = (\*it).start;

(\*it).start = (\*it).end;

(\*it).end = tempNode;

}

return;

}

ii) Function “labelInside” labels Nodes of one polygon’s *inside* parameter – *true*, if they are inside another polygon. (We need this to catch case where one polygon is completely inside other). Its calculated based on orientation of node in respect to another polygons edges.

//function "labelInsede" labels nodes of soilPolygon if they are inside, on the edge

//or outside of convex ClipPolygon

void labelInside (Polygon &convexPoly, Polygon &soilPoly)

{

//iterating through nodes of soilPolygon

for (std::list<Node>::iterator itSoil=soilPoly.polyNodes.begin(); itSoil != soilPoly.polyNodes.end(); ++itSoil)

{

int winding = 1;

bool onSide = false;

std::list<Edge>::iterator itConvex=convexPoly.polyEdges.begin();

//iterating through all edges of convex polygon and checking orientation

//if all orientations =1 thus point is inside polygon

while ( (itConvex != convexPoly.polyEdges.end()) && (winding>=0) )

{

winding = halfPlaneOrient((\*itConvex).start, (\*itConvex).end, (\*itSoil));

if (!onSide) {if (winding == 0) onSide=true;}

itConvex++;

}

//assigning correct value to insideClipPoly variable (1 inside, -1 outside, 0 on the edge)

if (winding == -1) (\*itSoil).insideClipPoly = -1;

else {if (onSide) (\*itSoil).insideClipPoly = 0;

else (\*itSoil).insideClipPoly = 1;}

}

return;}

1. **Next functions are also auxiliary and they aid in calculating intersecting points and putting them in list of nodes in right place.**

i) Function “computeIntersection” gets 2 edges as an input and returns true if they intersect. It also writes intersection if exists in intersectionNode variable and passes back to main algorithm. This function considers edges as vectors given with parametric equation. (see formulas on second page of 3\_17\_04\_W-A-clip\_polycurves.pdf document.) The system of two parametric equations of segments (edges) is solved through determinant formula. Case if determinant = 0 (edges are parallel) is considered separately and intersection is calculated in this case as well.

//Function ""computeIntersection" computes intersection point of two edges and

//returns "true" if they intersect. IntersectionNode has value of intersection

bool computeIntersection(Edge edge1, Edge edge2, Node &intersectionNode, int i)

{

double det = -1\*(edge1.end.x-edge1.start.x)\*(edge2.end.y-edge2.start.y)+(edge2.end.x-edge2.start.x)\*(edge1.end.y-edge1.start.y);

//case if edges are parallel

if (det==0)

{

//lengths of both edges

double lengthOfEdge1 = distTwoPoints (edge1.start, edge1.end);

double lengthOfEdge2 = distTwoPoints (edge2.start, edge2.end);

//if end of edge1 lies on segment (edge2.start, edge2.end] then edge1.end is intersection

if( fabs((distTwoPoints (edge1.end,edge2.start)+distTwoPoints(edge1.end,edge2.end)-lengthOfEdge2)) < MINDouble )

{

//edge1.end should not be same as edge2.start

if (distTwoPoints (edge1.end,edge2.start)>MINDouble){

intersectionNode.x = edge1.end.x;

intersectionNode.y = edge1.end.y;

intersectionNode.index = i;

intersectionNode.insideClipPoly = false;

intersectionNode.intersection = true;

intersectionNode.inward = false;

intersectionNode.visited = false;

return true;

}

}

//if end of edge2 lies on segment (edge1.start, edge1.end] then edge2.end is intersection

if( fabs(distTwoPoints (edge2.end,edge1.start)+distTwoPoints(edge2.end,edge1.end)-lengthOfEdge1) < MINDouble )

{

//edge2.end should not be same as edge1.start

if (distTwoPoints (edge2.end,edge1.start)>MINDouble) {

intersectionNode.x = edge2.end.x;

intersectionNode.y = edge2.end.y;

intersectionNode.index = i;

intersectionNode.insideClipPoly = false;

intersectionNode.intersection = true;

intersectionNode.inward = false;

intersectionNode.visited = false;

return true;

}

}

return false; //if parallel but not lay on each other then return false

}

double detT = -1\*(edge2.start.x-edge1.start.x)\*(edge2.end.y-edge2.start.y)+(edge2.end.x-edge2.start.x)\*(edge2.start.y-edge1.start.y);

double detS = (edge1.end.x-edge1.start.x)\*(edge2.start.y-edge1.start.y)-(edge2.start.x-edge1.start.x)\*(edge1.end.y-edge1.start.y);

//calculating parameter s and t for intersecting point between two edges

//if both parameters are 0<s,t<1 then edges intersect

double s = detS/det;

double t = detT/det;

//if edges don't intersect - return false

//else we calculate coordinates x and y of intersection trough parametric equation of line

if (s<=0 || s>1 || t<=0 || t>1)

return false;

else

{

double x = edge2.start.x +(edge2.end.x-edge2.start.x)\*s;

double y = edge2.start.y +(edge2.end.y-edge2.start.y)\*s;

intersectionNode.x = x;

intersectionNode.y = y;

intersectionNode.index = i;

intersectionNode.insideClipPoly = false;

intersectionNode.intersection = true;

intersectionNode.inward = false;

intersectionNode.visited = false;

return true;

}

}

ii) If two edges intersect then two polygons (Convex and Soil), two edges (which intersect), and intersection Node are passed to function **“insertIntersection”**. Main purpose of this function is to place intersecting node in proper place in the list of Nodes in both polygons. Function matches beginning coordinate of intersecting edge with corresponding Node in Node list and inserts Intersection point after it (same algorithm for both polygons)

//function "insertIntersection" inserts intersection node into both list of nodes

//for convexPolygon and soilPolygon, after start point of convexEdge and soilEdge

void insertIntersection(Polygon &convexPoly, Polygon &soilPoly, Edge convexEdge, Edge soilEdge, Node intersection)

{

bool nodeFoundFlag = false;//flag which indicates that node corresponding to end of edge found

//searching proper place and inserting intersection in convex polygon

std::list<Node>::iterator it=convexPoly.polyNodes.begin();

while ( (it != convexPoly.polyNodes.end()) && !nodeFoundFlag )

{

if ( ((\*it).x == convexEdge.end.x) && ((\*it).y == convexEdge.end.y ))

{

convexPoly.polyNodes.insert(it,intersection);

nodeFoundFlag = true;

}

it++;

}

if (!nodeFoundFlag) {std::cout<<"\n couldn't find where to insert intersection in convexPoly - ERROR\n"; exit (EXIT\_FAILURE);}

//searching proper place and inserting intersection in soil polygon

nodeFoundFlag = false;//flag which indicates that node corresponding to beginning of edge found

std::list<Node>::iterator it2=soilPoly.polyNodes.begin();

while ( (it2 != convexPoly.polyNodes.end()) && !nodeFoundFlag )

{

if ( ((\*it2).x == soilEdge.end.x) && ((\*it2).y == soilEdge.end.y ))

{

soilPoly.polyNodes.insert(it2,intersection);

nodeFoundFlag = true;

}

it2++;

}

if (!nodeFoundFlag) {std::cout<<"\n couldn't find where to insert intersection in SoilPoly - ERROR\n"; exit (EXIT\_FAILURE);}

return;

}

iii) If along one edge there are more than one intersection they can be in wrong order in list because we put them only based on the fact that they belong to that edge. Function **“rearrangeIntersectionByDist”** Rearranges order of intersection points along same edge if they are in wrong order. It does it based on calculating distance from swapping two intersection point if they are on one edge and distance from them to beginning of edge in not in correct order. This way we move along intersecting point of one edge and swap if wrong position till reach end of edge (like in bubble sort algorithm).

//function "rearrangeIntersectionsByDist" arranges intersection point on each edge in correct order

//by distance from beginning of the edge

void rearrangeIntersectionsByDist( Polygon &poly)

{

Node tempNode;

Node begin = poly.polyNodes.back(); //begin with last node of polygon

//last node should not be intersection, otherwise - mistake and exit

if (begin.intersection) {std::cout<<"\n last node in polygon is intersection) - ERROR\n"; exit (EXIT\_FAILURE);}

//for loop goes through list of nodes

for (std::list<Node>::iterator it=poly.polyNodes.begin(); it != poly.polyNodes.end(); ++it)

{

//if intersection found - we go through all consequetive intersections by while loop

if ( (\*it).intersection)

{

//iterator it2 iterates through other intersection of same edge (if exist)

std::list<Node>::iterator it2 = it;

it2++;

while ( (\*it2).intersection )

{

//if distance between intersection (\*it) and beginning of edge is grater

//than distance between intersection (\*it2) and beginning node then swap nodes

if (distTwoPoints(begin, \*it) > distTwoPoints (begin, \*it2 ) )

{

tempNode = \*it;

(\*it)=\*(it2);

\*(it2)=tempNode;

}

it2++;

}

}

//else, if not intersection node becomes beginning node

else begin = \*it;

}

return;

}

iiii)function “**removeIntersection**” removes intersection with index = intersectIndex from both polygons (Convex failure and Soil). This is done when vertex of one polygon only touches but not actualy intersects another

//function "removeIntersection" removes intersection point from both Polygon lists

//if vertice of one polygon only touches but not intersects another polygon

void removeIntersection (Polygon &convexPoly, Polygon &soilPoly, int intersectIndex)

{

//find intersection point on convex polygon

std::list<Node>::iterator itConvexNodes = convexPoly.polyNodes.begin();

while ( (\*itConvexNodes).index != intersectIndex)

{itConvexNodes++;

if (itConvexNodes==convexPoly.polyNodes.end())

{std::cout<<"\n Intersection number "<<intersectIndex<<"was not found in Convex polygon list - ERROR\n"; exit (EXIT\_FAILURE);}

}

//erase intersection from convex polygon list

itConvexNodes = convexPoly.polyNodes.erase(itConvexNodes);

//find intersecting point in soil polygon

std::list<Node>::iterator itSoilNodes = soilPoly.polyNodes.begin();

while ( (\*itSoilNodes).index != intersectIndex)

{itSoilNodes++;

if (itSoilNodes==soilPoly.polyNodes.end())

{std::cout<<"\n Intersection number "<<intersectIndex<<"was not found in Soil polygon list - ERROR\n"; exit (EXIT\_FAILURE);}

}

//erase intersection point from soil polygon list

itSoilNodes = soilPoly.polyNodes.erase(itSoilNodes);

return;

}

iiiii) Function “**markIntersection**” assigns intersection node with index = intersectIndex, value passed in parameter inward (if true then intersection is inward).

//function "markIntersection" assigns intersection with index "intersectIndex" inward parameter value

void markIntersection(Polygon &poly, int intersectIndex, bool inward )

{

//find intersection point on polygon

std::list<Node>::iterator itNodes = poly.polyNodes.begin();

while ( (\*itNodes).index != intersectIndex)

{itNodes++;

if (itNodes==poly.polyNodes.end())

{std::cout<<"\n Intersection number "<<intersectIndex<<"was not found in polygon list - ERROR\n"; exit (EXIT\_FAILURE);}

}

//mark intersection inward parameter

(\*itNodes).inward = inward;

return;

}

iiiiii) Function “**markVisited**” marks intersection with index – intersectIndex as visited in both polygons. This is needed during main iteration of Weiler-Atherton clipping algorithm implementation.

//function "markVisited" marks intersection with index - intersectIndex as visited in both polygons

void markVisited (Polygon &convexPoly, Polygon &soilPoly, int intersectIndex)

{

//find intersection node with index "intersectionIndex" in Convex Polygon

std::list<Node>::iterator itNodes = convexPoly.polyNodes.begin();

while ( (\*itNodes).index != intersectIndex )

{itNodes++;

if (itNodes==convexPoly.polyNodes.end())

{std::cout<<"\n Intersection number "<<intersectIndex<<" was not found in Convex polygon list - ERROR\n"; exit (EXIT\_FAILURE);}

}

//mark intersection visited is Convex Polygon

(\*itNodes).visited = true;

//find intersection node with index "intersectionIndex" in Soil Polygon

itNodes = soilPoly.polyNodes.begin();

while ( (\*itNodes).index != intersectIndex )

{itNodes++;

if (itNodes==soilPoly.polyNodes.end())

{std::cout<<"\n Intersection number "<<intersectIndex<<" was not found in Soil polygon list - ERROR\n"; exit (EXIT\_FAILURE);}

}

//mark intersection visited in Soil Polygon

(\*itNodes).visited = true;

return;

}

iiiiiii) Function “**removeDuplicateVerticeAndMarkIntersection**” removes vertex that is located at the same spot as intersection point with index = intersectIndex (is same point); or is extremely close (closer than MINDouble value 1.0e-9) to it. This function also assigns corresponding inward value to same intersection point.

//function "removeDuplicateVerticeAndMarkIntersection" removes vertice that duplicates intersection node or is very-very close

//function also mark intersection point INWARD as true if inward parameter is true

void removeDuplicateVerticeAndMarkIntersection(Polygon &poly, int intersectIndex, bool inward)

{

//find intersection point on polygon

std::list<Node>::iterator itNodes = poly.polyNodes.begin();

while ( (\*itNodes).index != intersectIndex)

{itNodes++;

if (itNodes==poly.polyNodes.end())

{std::cout<<"\n Intersection number "<<intersectIndex<<"was not found in polygon list - ERROR\n"; exit (EXIT\_FAILURE);}

}

//mark intersection inward parameter

(\*itNodes).inward = inward;

//find duplicating (or very close) vertice and delete from list

Node intersectionNode = (\*itNodes);

itNodes = poly.polyNodes.begin();

//while distance between intersection point and and vertice is not very close - iterate

while ( distTwoPoints( (\*itNodes),intersectionNode )>MINDouble || (\*itNodes).intersection)

{itNodes++;

if (itNodes==poly.polyNodes.end())

{std::cout<<"\n Intersection number "<<intersectIndex<<"was not positioned near any vertice - ERROR\n"; exit (EXIT\_FAILURE);}

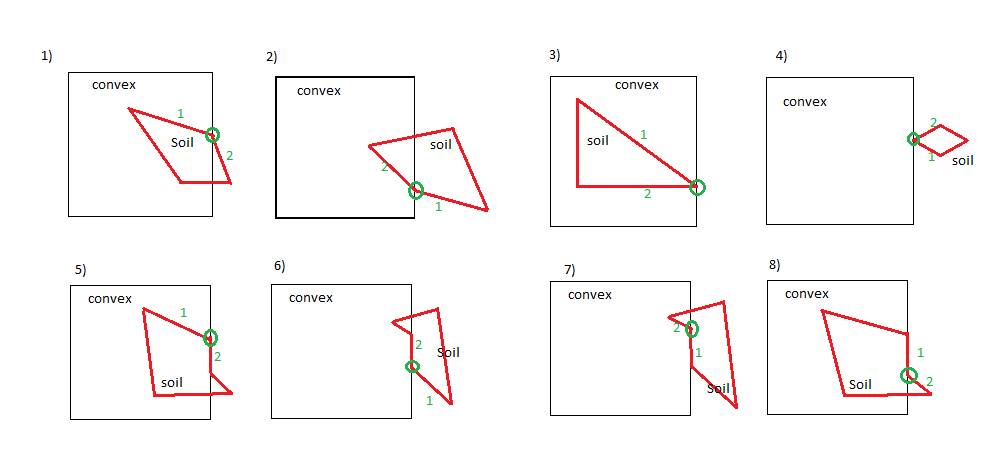
}

itNodes = poly.polyNodes.erase(itNodes);; //delete node duplicate or very close to intersection

return;

}

**6) Next several function represent algorithms developed by me (Dimitri Eiramjani) to handle all non-regular intersection cases. Intersection is considered regular when to edges intersect not at the end point but somewhere between endpoints of edges. All other types of intersections (vertex of one polygon lies on edge of another, two vertices of two polygons located at same spot, or two edges interlay on each other) are considered non-regular.**

i) Function “**verticeSoilMarkIntersection**” deals with case when vertex of soil polygon lies exactly on the Edge of Convex (Failure plane) polygon. There are 8 cases of this type of intersection (see on picture) and all 8 types are handled separately. 

In function “**verticeSoilMarkIntersection**” we first find intersecting point in Soil poligon. Then we find two edges in Soil Polygon first edge which is before intersection point(marked number 1 in drawing) and second edge is edge after intersection point (marked number 2 in drawing). Then we find intersection point in Convex polygon, and points before and after it to construct intersecting edge of Convex polygon. Both polygon’s winding is already clockwise.

After that we use “VectorOrient” function to figure out angles between Soil polygon edges “1” and “2” with intersecting edge of Convex polygon. Based on information about angle we make decision which out of 8 cases we have.

Case 1: we deleted vertex which is located at intersection point from Soil polygon and mark that intersection as **outward**

Case 2: we delete vertex at intersecting point from Soil polygon and mark intersecting point as **inward**

Case 3 and 4: just delete intersecting point from both polygons (not real intersection)

Case 5: we delete duplicating vertex from Soil polygon node list and mark intersection as **outward**

Case 7: we delete duplicating vertex (located at intersection point) from Soil polygon, and mark intersection as **inward.**

Case 6 and 8: just delete intersecting point from both polygons (because actual intersection was caught at other end of Soil polygon edge which lies on Convex polygon edge)

//function "verticeSoilMarkIntersect" decides what to do with intersection

//which is close to or same as node of soil polygon

//marks it inward or deletes

void verticeSoilMarkIntersect(Polygon &convexPoly, Polygon &soilPoly, int intersectIndex)

{

Edge firstSoilEdge; //edge before intersecting node

Edge secondSoilEdge; //edge after intersecting node

//find intersecting point in soil polygon

std::list<Node>::iterator itSoilNodes = soilPoly.polyNodes.begin();

while ( (\*itSoilNodes).index != intersectIndex)

{itSoilNodes++;

if (itSoilNodes==soilPoly.polyNodes.end())

{std::cout<<"\n Intersection number "<<intersectIndex<<"was not found in Soil polygon list - ERROR\n"; exit (EXIT\_FAILURE);}

}

Node intersectionSoil = \*itSoilNodes; //intersection node in soil polygon

//looking for same two edges before and after intersecting node in soil Polygon

std::list<Edge>::iterator itSoilEdges = soilPoly.polyEdges.begin();

while ( distTwoPoints( intersectionSoil, (\*itSoilEdges).end )>MINDouble )

{itSoilEdges++;

if (itSoilEdges==soilPoly.polyEdges.end())

{std::cout<<"\n Edge before intersection "<<intersectIndex<<"was not found in Soil polygon edge list - ERROR\n"; exit (EXIT\_FAILURE);}

}

firstSoilEdge = \*itSoilEdges;

itSoilEdges++;

//check if firstSoilEdge was last one in list of edges, then second edge is first in list else next will be second

if (itSoilEdges==soilPoly.polyEdges.end())

secondSoilEdge = \*( soilPoly.polyEdges.begin() );

else secondSoilEdge = \* itSoilEdges;

//find intersection point on convex polygon

std::list<Node>::iterator itConvexNodes = convexPoly.polyNodes.begin();

while ( (\*itConvexNodes).index != intersectIndex)

{itConvexNodes++;

if (itConvexNodes==convexPoly.polyNodes.end())

{std::cout<<"\n Intersection number "<<intersectIndex<<"was not found in Convex polygon list - ERROR\n"; exit (EXIT\_FAILURE);}

}

Node intersectionConvex = \*itConvexNodes;

//find points before and after intersecting point in convex polygon

Node intersectionConvexBefore; //node before intersecting node in Convex polygon

Node intersectionConvexAfter; //node after intersecting node in Convex polygon

//if intersection is first in list

if (itConvexNodes==convexPoly.polyNodes.begin())

{

intersectionConvexBefore = convexPoly.polyNodes.back(); //node before intersection

itConvexNodes++;

if (itConvexNodes==convexPoly.polyNodes.end())

{std::cout<<"\n Intersection "<<intersectIndex<<"was last in Convex polygon list - ERROR\n"; exit (EXIT\_FAILURE);}

else intersectionConvexAfter = \*itConvexNodes; //node after intersection

}

else //else if intersection not first in convex polygon list

{

itConvexNodes--;

intersectionConvexBefore = \*itConvexNodes; //node before intersection

itConvexNodes++;

itConvexNodes++;

if (itConvexNodes==convexPoly.polyNodes.end())

{std::cout<<"\n Intersection "<<intersectIndex<<"was last in Convex polygon list - ERROR\n"; exit (EXIT\_FAILURE);}

else intersectionConvexAfter = \*itConvexNodes; //node after intersection

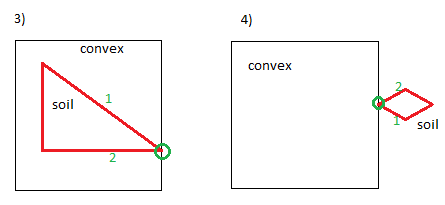
}

//test if orientation between edge before intersection in Soil polygon and Convex polygon segment

//and orientation between edge after intersection in Soil Polygon and Convex Polygon segment is opposite

if (vectorOrient(firstSoilEdge.start, firstSoilEdge.end, intersectionConvexBefore, intersectionConvexAfter)\*vectorOrient(secondSoilEdge.start, secondSoilEdge.end, intersectionConvexBefore, intersectionConvexAfter) == -1)

//then vertice only touches Convex polygon - so REMOVE INTERSECTION from both lists Convex and Soil

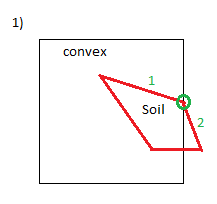


{removeIntersection (convexPoly, soilPoly, intersectIndex);}

//test if orientation between edge before intersection in Soil polygon and Convex polygon segment

//and orientation between edge after intersection in Soil Polygon and Convex Polygon segment is same sign

//then if orientation is +1 angle of both edges is less than 180 degree - delete vertice near intersection

//and mark intersection as OUTWARD in soil polygon. Mark as OUTWARD in Convex Polygon

if ( (vectorOrient(firstSoilEdge.start, firstSoilEdge.end, intersectionConvexBefore, intersectionConvexAfter)==1) && (vectorOrient(secondSoilEdge.start, secondSoilEdge.end, intersectionConvexBefore, intersectionConvexAfter) == 1) )

{

//delete intersection and mark vertice as intersection OUTWARD in soil polygon.

removeDuplicateVerticeAndMarkIntersection (soilPoly, intersectIndex, false);

//Mark as OUTWARD in Convex Polygon

markIntersection(convexPoly,intersectIndex, false);

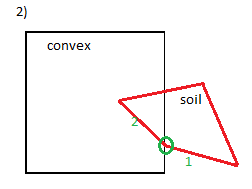
}

//test if orientation between edge before intersection in Soil polygon and Convex polygon segment

//and orientation between edge after intersection in Soil Polygon and Convex Polygon segment is same sign

//if orientation is -1 angle both edges is 180-360 degree - delete vertice near intersection

//and mark intersection as INWARD in soil polygon. Mark as INWARD in Convex Polygon



if ( (vectorOrient(firstSoilEdge.start, firstSoilEdge.end, intersectionConvexBefore, intersectionConvexAfter)==-1) && (vectorOrient(secondSoilEdge.start, secondSoilEdge.end, intersectionConvexBefore, intersectionConvexAfter) == -1) )

{

//delete intersection and mark vertice as intersection INWARD in soil polygon.

removeDuplicateVerticeAndMarkIntersection (soilPoly, intersectIndex, true);

//Mark as INWARD in Convex Polygon

markIntersection(convexPoly,intersectIndex, true);

}

//test if orientation of edge after intersection in soil polygon and Convex polygon segment is 0

//(edge of soil after intersecting vertice is collinear with convex polygon edge)

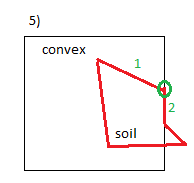
if ( vectorOrient(secondSoilEdge.start, secondSoilEdge.end, intersectionConvexBefore, intersectionConvexAfter) == 0 )

{

//then if orientation between edge before intersection in Soil polygon and Convex polygon segment is

//1 angle is 0-180 degree - delete vertice near intersection in Soil polygon

//mark intersection point as OUTWARD. Mark as OUTWARD in Convex Polygon



if ( vectorOrient(firstSoilEdge.start, firstSoilEdge.end, intersectionConvexBefore, intersectionConvexAfter)==1 )

{

removeDuplicateVerticeAndMarkIntersection (soilPoly, intersectIndex, false);

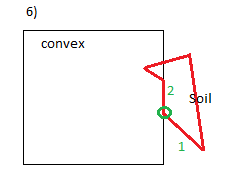
markIntersection(convexPoly,intersectIndex, false);

}

//if orientation between edge before intersection in Soil polygon and Convex polygon segment is

//-1, angle is 180-360 degree (side before intersection comes from outside)

//Then just delete intersection from Soil polygon and from Convex Polygon also



if ( vectorOrient(firstSoilEdge.start, firstSoilEdge.end, intersectionConvexBefore, intersectionConvexAfter)==-1 )

{

removeIntersection(convexPoly, soilPoly, intersectIndex);

}

}

//test if orientation of edge before intersection in Soil and Convex polygon segment is 0

//(edge of soil before intersection vertice is collinear with edge of convex polygon)

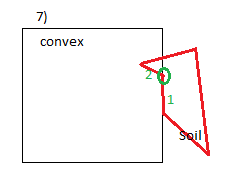
if ( vectorOrient(firstSoilEdge.start, firstSoilEdge.end, intersectionConvexBefore, intersectionConvexAfter)==0 )

{

//then if orientation between edge after intersection in Soil Polygon and Convex Polygon segment is

// -1 angle is 180-360 degree - delete vertice near intersection in Soil polygon

//and mark INWARD in soil polygon. Mark as INWARD in Convex Polygon



if ( vectorOrient(secondSoilEdge.start, secondSoilEdge.end, intersectionConvexBefore, intersectionConvexAfter) == -1 )

{

removeDuplicateVerticeAndMarkIntersection (soilPoly, intersectIndex, true);

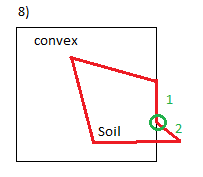
markIntersection(convexPoly,intersectIndex, true);

}

//if orientation between edge after intersection in Soil Polygon and Convex Polygon segment is

// 1 angle is 0-180 degree - just delete intersection from soil polygon (no intersection)

//Delete intersection from Convex Polygon also



if ( vectorOrient(secondSoilEdge.start, secondSoilEdge.end, intersectionConvexBefore, intersectionConvexAfter) == 1 )

{

removeIntersection(convexPoly, soilPoly, intersectIndex);

}

}

//If orientation of both edges before and after intersecting point in Soil polygon with

//Convex polygon segment are equal to 0 - ERROR Collinear Edges detected in Soil Polygon

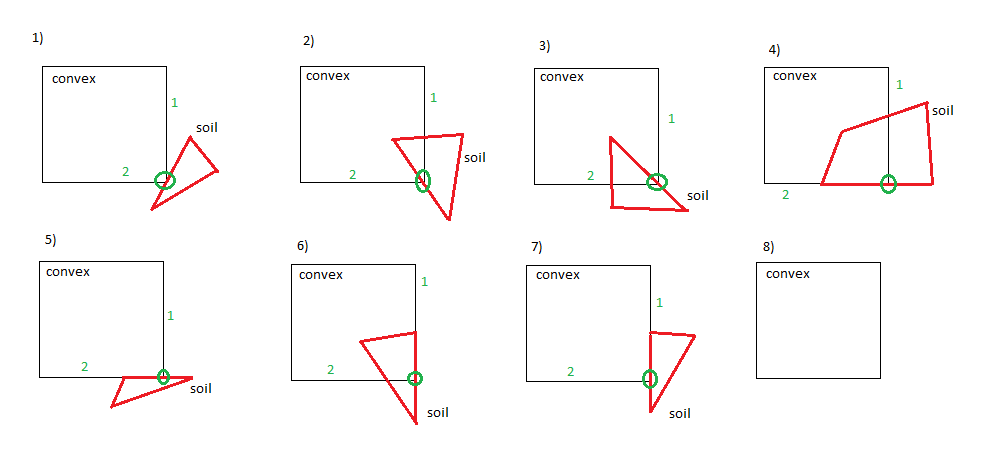
if ( (vectorOrient(firstSoilEdge.start, firstSoilEdge.end, intersectionConvexBefore, intersectionConvexAfter)==0)&& (vectorOrient(secondSoilEdge.start, secondSoilEdge.end, intersectionConvexBefore, intersectionConvexAfter) ==0) )

{ std::cout<<"\n Collinear edges in Soil polygon around intersection number "<<intersectIndex<<" - ERROR\n"; exit (EXIT\_FAILURE);}

return;

}

ii) Function “**verticeConvexMarkIntersection**” deals with case, when two polygons intersect exactly or extremely close to vertex of Convex (failure plane) polygon. There is 7 cases of that type of intersections and all those cases are handled separately in this function. Convex polygon (failure plane is consider to be always convex).



First we find intersecting point with index “intersectIndex” in Convex polygon (this intersecting point coincides with vertex of convex polygon). Then we find two edges of big convex polygon – before and after intersecting point. Those edges marked “1”-before and “2” – after intersecting point. Then we find intersection point in Soil polygon (circled green). Then we find two points on Soil polygon (red): first exactly before and second exactly after intersecting point. Both polygon’s winding is already clockwise.

After that we use “VectorOrient” function to figure out angles between Convex polygon edges “1” and “2” with intersecting edge of Soil polygon separately. Based on information about angles (are they collinear, more or less then 180 degree) we make decision which out of 8 cases we have.

**Case 1**: We **delete** intersection because soil polygon only touches vertex of Convex polygon.

**Case 2**: We delete vertex in Convex polygon which coincides with intersection and mark intersection as **INWARD** in both polygons

**Case3:**  We delete vertex in Convex polygon which coincides with intersection and mark intersection as **OUTWARD** in both polygons

**Case 4:** we just **delete** intersection from both polygons, because outward intersection was caught before intersecting vertex marked green and inward intersection was caught after.

**Case 5 and 7:** Just **remove** intersection from both polygons because Soil polygon only touches Convex polygon.

**Case 6:** just **delete** intersecting point from both polygons because actual outward intersection was caught before (where vertex of Soil polygon lies on convex polygon edge number “1”).

//function "verticeConvexMarkIntersect" decides what to do with intersection

//which is very close to or same as node of Convex polygon

//marks it inward or deletes

void verticeConvexMarkIntersect (Polygon &convexPoly, Polygon &soilPoly, int intersectIndex)

{

Edge firstConvexEdge; //edge before intersecting node

Edge secondConvexEdge; //edge after intersecting node

//find intersecting point in convex polygon

std::list<Node>::iterator itConvexNodes = convexPoly.polyNodes.begin();

while ( (\*itConvexNodes).index != intersectIndex)

{itConvexNodes++;

if (itConvexNodes==convexPoly.polyNodes.end())

{std::cout<<"\n Intersection number "<<intersectIndex<<"was not found in Convex polygon list - ERROR\n"; exit (EXIT\_FAILURE);}

}

Node intersectionConvex = \*itConvexNodes; //intersection node in Convex polygon

//looking for same two edges before and after intersecting node in Convex Polygon

std::list<Edge>::iterator itConvexEdges = convexPoly.polyEdges.begin();

while ( distTwoPoints( intersectionConvex, (\*itConvexEdges).end )>MINDouble )

{itConvexEdges++;

if (itConvexEdges==convexPoly.polyEdges.end())

{std::cout<<"\n Edge before intersection "<<intersectIndex<<" was not found in Convex polygon edge list - ERROR\n"; exit (EXIT\_FAILURE);}

}

firstConvexEdge = \*itConvexEdges;

itConvexEdges++;

//check if firstSoilEdge was last one in list of edges, then second edge is first in list else next will be second

if (itConvexEdges==convexPoly.polyEdges.end())

secondConvexEdge = \*( convexPoly.polyEdges.begin() );

else secondConvexEdge = \* itConvexEdges;

//find intersection point on soil polygon

std::list<Node>::iterator itSoilNodes = soilPoly.polyNodes.begin();

while ( (\*itSoilNodes).index != intersectIndex)

{itSoilNodes++;

if (itSoilNodes==soilPoly.polyNodes.end())

{std::cout<<"\n Intersection number "<<intersectIndex<<" was not found in Soil polygon list - ERROR\n"; exit (EXIT\_FAILURE);}

}

Node intersectionSoil = \*itSoilNodes;

//find points before and after intersecting point in soil polygon

Node intersectionSoilBefore; //node before intersecting node in Soil polygon

Node intersectionSoilAfter; //node after intersecting node in Soil polygon

//if intersection is first in list

if (itSoilNodes==soilPoly.polyNodes.begin())

{

intersectionSoilBefore = soilPoly.polyNodes.back(); //node before intersection

itSoilNodes++;

if (itSoilNodes==soilPoly.polyNodes.end())

{std::cout<<"\n Intersection "<<intersectIndex<<"was last in Soil polygon list - ERROR\n"; exit (EXIT\_FAILURE);}

else intersectionSoilAfter = \*itSoilNodes; //node after intersection

}

else //else if intersection not first in convex polygon list

{

itSoilNodes--;

intersectionSoilBefore = \*itSoilNodes; //node before intersection

itSoilNodes++;

itSoilNodes++;

if (itSoilNodes==soilPoly.polyNodes.end())

{std::cout<<"\n Intersection "<<intersectIndex<<" was last in Soil polygon list - ERROR\n"; exit (EXIT\_FAILURE);}

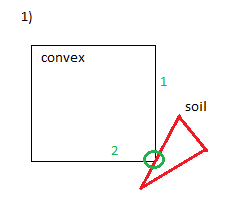
else intersectionSoilAfter = \*itSoilNodes; //node after intersection

}

//test if orientation between edge before intersection in Convex polygon and Soil polygon segment

//and orientation between edge after intersection in Convex Polygon and Soil Polygon segment is opposite

if (vectorOrient(firstConvexEdge.start, firstConvexEdge.end, intersectionSoilBefore, intersectionSoilAfter)\*vectorOrient(secondConvexEdge.start, secondConvexEdge.end, intersectionSoilBefore, intersectionSoilAfter) == -1)



//then vertice only touches Soil polygon - so REMOVE INTERSECTION from both lists Convex and Soil

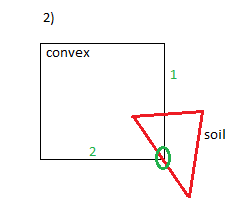
{removeIntersection (convexPoly, soilPoly, intersectIndex);}

//test if orientation between edge before intersection in Convex polygon and Soil polygon segment

//and orientation between edge after intersection in Convex Polygon and Soil Polygon segment is same sign

//then if orientation is +1 angle of both edges is less than 180 degree - delete vertice near intersection

//and mark intersection as INWARD in Convex polygon. Mark as INWARD in Soil Polygon



if ( (vectorOrient(firstConvexEdge.start, firstConvexEdge.end, intersectionSoilBefore, intersectionSoilAfter)==1) && (vectorOrient(secondConvexEdge.start, secondConvexEdge.end, intersectionSoilBefore, intersectionSoilAfter) == 1) )

{

//delete intersection and mark vertice as intersection INWARD in Convex polygon.

removeDuplicateVerticeAndMarkIntersection (convexPoly, intersectIndex, true);

//Mark as INWARD in Soil Polygon

markIntersection(soilPoly,intersectIndex, true);

}

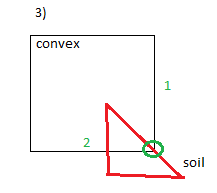
//test if orientation between edge before intersection in Convex polygon and Soil polygon segment

//and orientation between edge after intersection in Convex Polygon and Soil Polygon segment is same sign

//if orientation is -1 angle both edges is 180-360 degree - delete vertice near intersection

//and mark intersection as OUTWARD in Convex polygon. Mark as OUTWARD in Soil Polygon

if ( (vectorOrient(firstConvexEdge.start, firstConvexEdge.end, intersectionSoilBefore, intersectionSoilAfter)==-1) && (vectorOrient(secondConvexEdge.start, secondConvexEdge.end, intersectionSoilBefore, intersectionSoilAfter) == -1) )



{

//delete intersection and mark vertice as intersection OUTWARD in Convex polygon.

removeDuplicateVerticeAndMarkIntersection (convexPoly, intersectIndex, false);

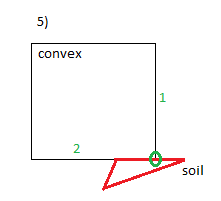
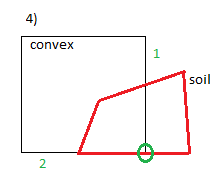
//Mark as OUTWARD in Soil Polygon

markIntersection(soilPoly,intersectIndex, false);

}

//test if orientation of edge after intersection in Convex polygon and Soil polygon segment is 0

//(edge of Convex after intersecting vertice is collinear with soil polygon edge)



if ( vectorOrient(secondConvexEdge.start, secondConvexEdge.end, intersectionSoilBefore, intersectionSoilAfter) == 0 )

{

//Then just delete intersection

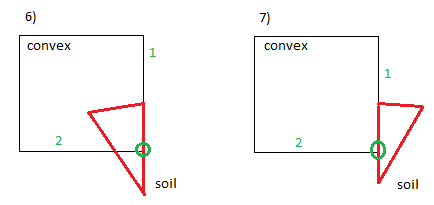
removeIntersection(convexPoly, soilPoly, intersectIndex);

}

//test if orientation of edge before intersection in Convex polygon and Soil polygon segment is 0

//(edge of Convex before intersection vertice is collinear with edge of Soil polygon)

if ( vectorOrient(firstConvexEdge.start, firstConvexEdge.end, intersectionSoilBefore, intersectionSoilAfter)==0 )



{

//Then just delete intersection

removeIntersection(convexPoly, soilPoly, intersectIndex);

}

//If orientation of both edges before and after intersecting point in Convex polygon with

//Soil polygon segment are equal to 0 - ERROR Collinear Edges detected in Convex Polygon

if ( (vectorOrient(firstConvexEdge.start, firstConvexEdge.end, intersectionSoilBefore, intersectionSoilAfter)==0)&& (vectorOrient(secondConvexEdge.start, secondConvexEdge.end, intersectionSoilBefore, intersectionSoilAfter) ==0) )

{ std::cout<<"\n Collinear edges in Convex polygon around intersection number "<<intersectIndex<<" - ERROR\n"; exit (EXIT\_FAILURE);}

return;

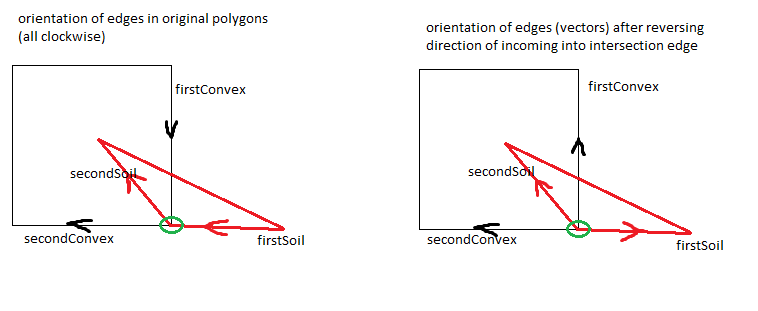
}

iii) Function “**verticeToVerticeMarkInward**” is longest function and deals with case when vertex of soil polygon lies exactly on vertex of Convex (Failure plane) polygon (so called vertex-to-vertex intersection). There are \*\*\*\*\* cases of this type of intersection (see on picture) and all \*\*\*\* types are handled separately.

First, we find intersection point in Convex polygon using its index from “intersectIndex” parameter. Then we find two edges before and after intersecting node in Convex polygon and store those edges as “*firstConvexEdge*” and “*secondConvexEdge*”.

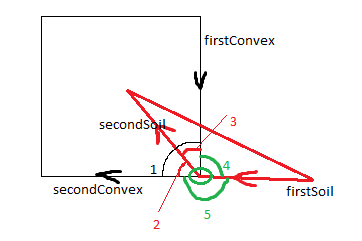
Second, we find intersecting vertex in soil polygon and then find two edges one before intersecting vertex one after intersecting vertex. We store those edges as *“firstSoilEdge*” and *“secondSoilEdge*”.

Third, we reversing direction of edges before intersection in both polygons (“*firstConvexEdge*” and *“secondSoilEdge*”). This is done because we need all vectors to start from intersection point for making comparison of angles later and deciding what kind of intersection it is.



Next we calculate 5 angles in radians. Those angle values will be used in determining which case we have during Vertex-to-Vertex intersection.

Those 5 angles are shown on picture:



1 - “convexAngle”

2 – “seconConvexSeconSoilAngle”

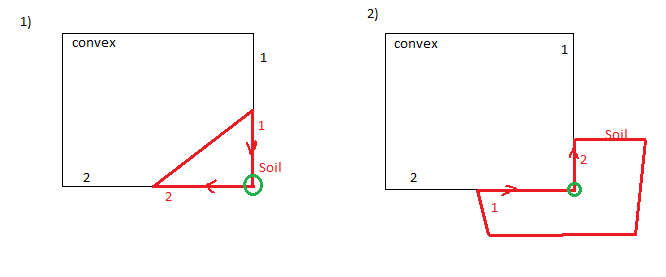
3 – “firstConvexSecondSoilAngle”

4 – “firstConvexFirstSoilAngle”

5 – “secondConvexFirstSoilAngle”

After that using these angles we got 14 cases of different type of intersection and for each type we make appropriate decision about what kind of intersection it is (inward, outward or not intersection at all).

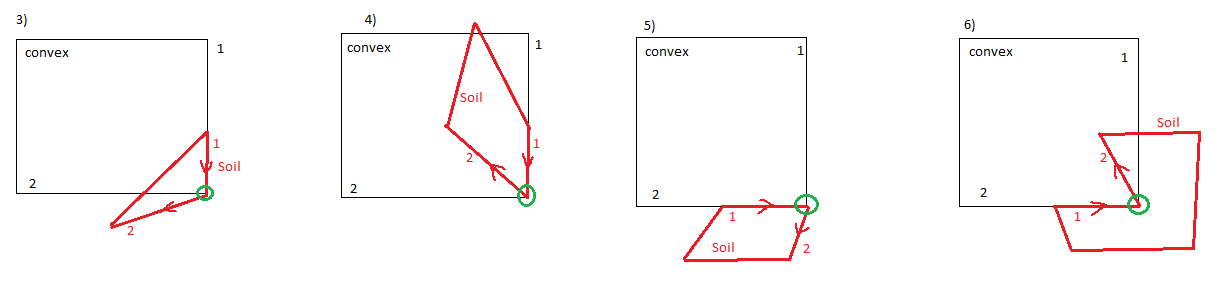
**Case 1 and 2:** First two cases represent situation when edges of Convex and Soil polygons interlay (angle between both of them are zero). In both of these cases we just delete intersection, because there is no actual intersection.



**Case 3, 4, 5 and 6:** In these four cases only first (before intersection) edge of Soil Polygon interlays with edge of Convex polygon. In cases 3 and 4 first soil edge interlays with first (before intersection) edge of Convex polygon and in cases 5, 6 first Soil polygon edge interlays with second (after intersection) edge of Convex polygon. Then we compare angle between first and second edges of big Convex polygon (“convexAngle”) with sum of angles that second edge of Soil polygon makes with both edges of Convex polygon (“firstConvexSecondSoilAngle”+ “seconConvexSeconSoilAngle”).

If they are equal (**cases 4 and 6**) – then second soil edge is entering Convex polygon, so we **remove** duplicating vertices in both polygons and mark intersection as **Inward**.

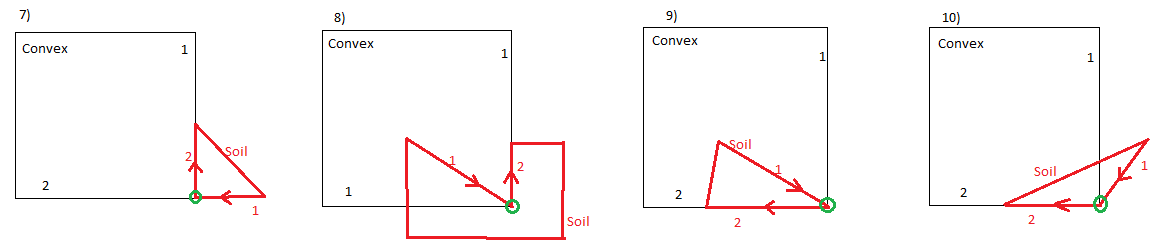
Else in **cases 3 and 5** – there is no actual intersection and we just **REMOVE** intersection.



**Case 7, 8, 9, and 10:** In these four cases only second (after intersection) edge of Soil Polygon interlays with edge of Convex polygon. In cases 7 and 8 second soil edge interlays with first (before intersection) edge of Convex polygon and in cases 9 and 10 second edge of Soil polygon interlays with second (after intersection) edge of Convex polygon. Then we compare angle between first and second edges of big Convex polygon (“convexAngle”) with sum of angles that first edge of Soil polygon makes with both edges of Convex polygon (“firstConvexFirstSoilAngle”+ “seconConvexFirstSoilAngle”).

If “convexAngl” is equal to above mentioned sum of angles (**cases 8 and 9**) – then first soil edge is exiting Convex polygon, so we **remove** duplicating vertices in both polygons and mark intersection as **OUTWARD.**

Else in **cases 7 and 10** there is no actual intersection and we just **REMOVE** intersection.



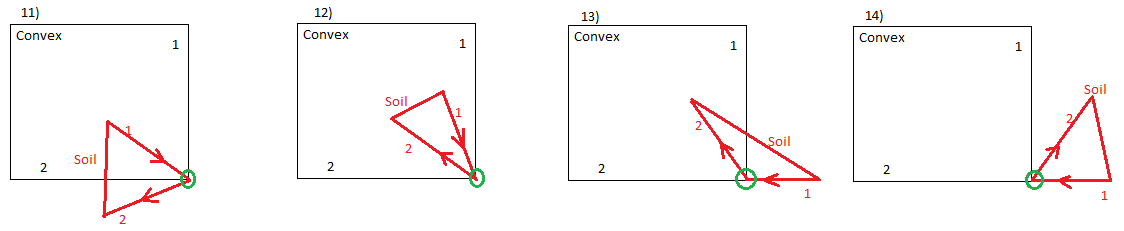
**Case 11, 12, 13, and 14:** In this 4 cases edges of Convex and Soil polygons don’t interlay.

In **case 11** first edge of Soil polygon is located inside Convex polygon angle (“convexAngle” = “firstConvexFirstSoilAngle”+ “seconConvexFirstSoilAngle”) and second edge of Soil is outside of Convex angle(“convexAngle” not equal “firstConvexSecondSoilAngle”+ “seconConvexSecondSoilAngle”) then we delete duplicating vertices from both polygons and mark intersection **OUTWARD.**

In **case 12** first edge of Soil polygon is located inside Convex polygon angle (“convexAngle” = “firstConvexFirstSoilAngle”+ “seconConvexFirstSoilAngle”) and second edge of Soil is also located inside of Convex angle (“convexAngle“ = ”firstConvexSecondSoilAngle”+ “seconConvexSecondSoilAngle”) then Soil polygon is inside Convex and we just **REMOVE** intersection.

In **case 13** first edge of Soil polygon is located outside Convex polygon angle (“convexAngle” < “firstConvexFirstSoilAngle”+ “seconConvexFirstSoilAngle”) and second edge of Soil is inside of Convex angle (“convexAngle” = “firstConvexSecondSoilAngle”+ “seconConvexSecondSoilAngle”) then soil polygon enters Convex polygon from outside and we delete duplicating vertices from both polygons and mark intersection **INWARD.**

In **case 14** first edge of Soil polygon is located outside Convex polygon angle (“convexAngle” < “firstConvexFirstSoilAngle”+ “seconConvexFirstSoilAngle”) and second edge of Soil is also located outside of Convex angle (“convexAngle“ < ”firstConvexSecondSoilAngle”+ “seconConvexSecondSoilAngle”) then Soil polygon is outside Convex and we just **REMOVE** intersection.

****

//function "verticeToVerticeMarkInvard" decides what to do with vertice to vertice intersection

//if intersection exists it is marked inward or outward. If just tach each other - intersection is deleted

//function returns false if intersection in Soil Polygon is deleted

void verticeToVerticeMarkInvard(Polygon &convexPoly, Polygon &soilPoly, int intersectIndex)

{

Edge firstConvexEdge; //edge before intersecting node in Convex polygon

Edge secondConvexEdge; //edge after intersecting node in Convex polygon

Edge firstSoilEdge; //edge before intersecting node in Soil polygon

Edge secondSoilEdge; //edge after intersecting node in Soil polygon

//find intersecting point in convex polygon

std::list<Node>::iterator itConvexNodes = convexPoly.polyNodes.begin();

while ( (\*itConvexNodes).index != intersectIndex)

{itConvexNodes++;

if (itConvexNodes==convexPoly.polyNodes.end())

{std::cout<<"\n Intersection number "<<intersectIndex<<" was not found in Convex polygon list - ERROR\n"; exit (EXIT\_FAILURE);}

}

Node intersectionConvex = \*itConvexNodes; //intersection node in Convex polygon

//looking for same two edges before and after intersecting node in Convex Polygon

std::list<Edge>::iterator itConvexEdges = convexPoly.polyEdges.begin();

while ( distTwoPoints( intersectionConvex, (\*itConvexEdges).end )>MINDouble )

{itConvexEdges++;

if (itConvexEdges==convexPoly.polyEdges.end())

{std::cout<<"\n Edge before intersection "<<intersectIndex<<" was not found in Convex polygon edge list - ERROR\n"; exit (EXIT\_FAILURE);}

}

firstConvexEdge = \*itConvexEdges;

itConvexEdges++;

//check if firstConvexEdge was last one in list of edges, then second edge is first in list else next will be second

if (itConvexEdges==convexPoly.polyEdges.end())

secondConvexEdge = \*( convexPoly.polyEdges.begin() );

else secondConvexEdge = \*itConvexEdges;

//find intersection point on soil polygon

std::list<Node>::iterator itSoilNodes = soilPoly.polyNodes.begin();

while ( (\*itSoilNodes).index != intersectIndex)

{itSoilNodes++;

if (itSoilNodes==soilPoly.polyNodes.end())

{std::cout<<"\n Intersection number "<<intersectIndex<<" was not found in Soil polygon list - ERROR\n"; exit (EXIT\_FAILURE);}

}

Node intersectionSoil = \*itSoilNodes;

//looking for two edges before and after intersecting node in soil Polygon

std::list<Edge>::iterator itSoilEdges = soilPoly.polyEdges.begin();

while ( distTwoPoints( intersectionSoil, (\*itSoilEdges).end )>MINDouble )

{itSoilEdges++;

if (itSoilEdges==soilPoly.polyEdges.end())

{std::cout<<"\n Edge before intersection "<<intersectIndex<<" was not found in Soil polygon edge list - ERROR\n"; exit (EXIT\_FAILURE);}

}

firstSoilEdge = \*itSoilEdges;

itSoilEdges++;

//check if firstSoilEdge was last one in list of edges, then second edge is first in list else next will be second

if (itSoilEdges==soilPoly.polyEdges.end())

secondSoilEdge = \*( soilPoly.polyEdges.begin() );

else secondSoilEdge = \*itSoilEdges;

//reversing first edges (incoming in intersection point) in both polygons

//to get all four vectors outgoing from intersection point

Edge tempEdge;

//reverse in convex polygon

tempEdge.start = firstConvexEdge.end;

tempEdge.end = firstConvexEdge.start;

firstConvexEdge = tempEdge;

//reverse in Soil polygon

tempEdge.start = firstSoilEdge.end;

tempEdge.end = firstSoilEdge.start;

firstSoilEdge = tempEdge;

//calculate angle between first and second edges of Convex Polygon

double convexAngle = angleBetweenVector(firstConvexEdge, secondConvexEdge);

//angle between edges of soil and convex Polygons

double firstConvexFirstSoilAngle = angleBetweenVector(firstConvexEdge, firstSoilEdge);

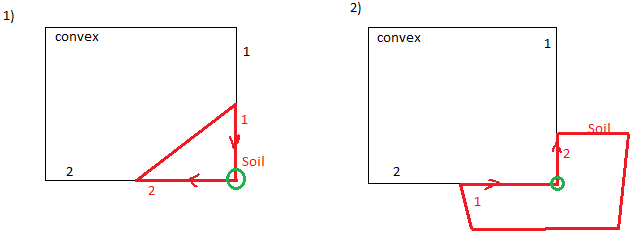
double firstConvexSecondSoilAngle = angleBetweenVector(firstConvexEdge, secondSoilEdge);

double secondConvexFirstSoilAngle = angleBetweenVector(secondConvexEdge, firstSoilEdge);

double secondConvexSecondSoilAngle = angleBetweenVector(secondConvexEdge, secondSoilEdge);

//case where both first and second edges of soil Polygon lay on edges of Convex polygon

//then delete intersection



if ( (firstConvexFirstSoilAngle <MINDouble)&&(secondConvexSecondSoilAngle<MINDouble) )

{

removeIntersection (convexPoly, soilPoly, intersectIndex);

return ;

}

if ( (firstConvexSecondSoilAngle <MINDouble)&&(secondConvexFirstSoilAngle<MINDouble) )

{

removeIntersection (convexPoly, soilPoly, intersectIndex);

return ;

}

//case where only first edge of Soil polygon lays on one of the edges of Convex polygon

if ( (firstConvexFirstSoilAngle <MINDouble)||(secondConvexFirstSoilAngle<MINDouble) )

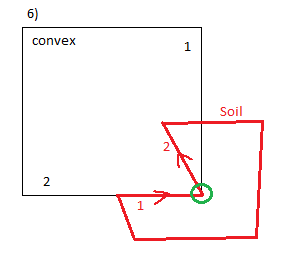
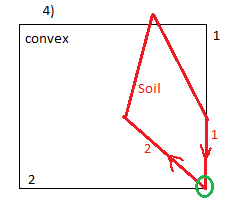
{

//if angle between first-second edges of Convex polygon is equal

//sum of angles secondSoil-firstConvex plus seconSoil-secondConvex

//then secondSoil edge enters Convex polygon.

//Delete duplicating vertices in both polygons and mark intersection INWARD



if ( fabs(convexAngle-firstConvexSecondSoilAngle-secondConvexSecondSoilAngle)<MINDouble )

{

removeDuplicateVerticeAndMarkIntersection(convexPoly, intersectIndex, true);

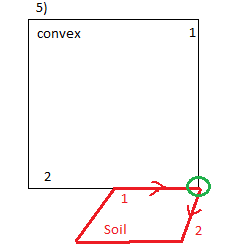
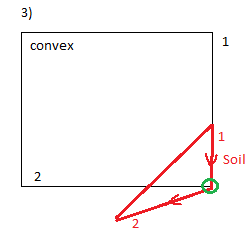
removeDuplicateVerticeAndMarkIntersection(soilPoly, intersectIndex,true);

}

//if angle between first-second edges of Convex polygon is not equal

//sum of angles secondSoil-firstConvex plus seconSoil-secondConvex

//then secondSoil edge goes out Convex polygon. Just delete intersection



else

{

removeIntersection(convexPoly, soilPoly, intersectIndex);

}

return;

}

//case where only second edge of Soil polygon lays on one of the edges of Convex polygon

if ( (firstConvexSecondSoilAngle <MINDouble)||(secondConvexSecondSoilAngle<MINDouble) )

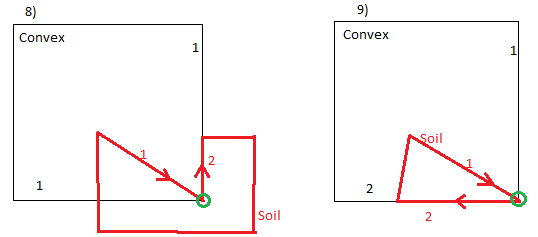
{

//if angle between first-second edges of Convex polygon is equal

//sum of angles firstSoil-firstConvex plus firstSoil-secondConvex

//then firstSoil edge comes from inside of Convex polygon.

//Delete duplicating vertices in both polygons and mark intersection OUTWARD



if ( fabs(convexAngle-firstConvexFirstSoilAngle-secondConvexFirstSoilAngle)<MINDouble )

{

removeDuplicateVerticeAndMarkIntersection(convexPoly, intersectIndex, false);

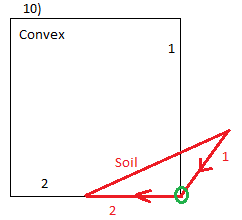
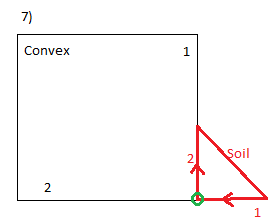
removeDuplicateVerticeAndMarkIntersection(soilPoly, intersectIndex,false);

}

//if angle between first-second edges of Convex polygon is not equal

//sum of angles firstSoil-firstConvex plus firstSoil-secondConvex

//then firstSoil edge comes from outside of Convex polygon. Just delete intersection



else

{

removeIntersection(convexPoly, soilPoly, intersectIndex);

}

return;

}

//case where first edge (before intersection) of Soil polygon comes strictly from inside out of Convex polygon

if ( fabs(convexAngle-firstConvexFirstSoilAngle-secondConvexFirstSoilAngle)<MINDouble )

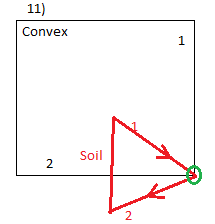
{

//if angle between first-second edges of Convex polygon is not equal

//sum of angles secondSoil-firstConvex plus secondSoil-secondConvex

//then secondSoil goes outside of Convex polygon.

//Delete duplicating vertices in both polygons and mark intersection OUTWARD



if ( fabs(convexAngle-firstConvexSecondSoilAngle-secondConvexSecondSoilAngle)>=MINDouble )

{

removeDuplicateVerticeAndMarkIntersection(convexPoly, intersectIndex, false);

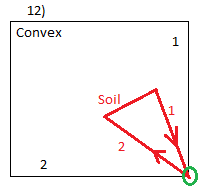
removeDuplicateVerticeAndMarkIntersection(soilPoly, intersectIndex,false);

}

//if angle between first-second edges of Convex polygon is equal

//sum of angles secondSoil-firstConvex plus secondSoil-secondConvex

//then secondSoil returns inside Convex polygon. Just delete intersection



else

{

removeIntersection(convexPoly, soilPoly, intersectIndex);

}

return;

}

//case where first edge (before intersection) of Soil polygon comes strictly from outside out of Convex polygon

if ( fabs(convexAngle-firstConvexFirstSoilAngle-secondConvexFirstSoilAngle)>=MINDouble )

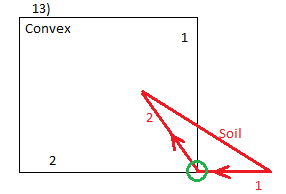
{

//if angle between first-second edges of Convex polygon is equal

//sum of angles secondSoil-firstConvex plus secondSoil-secondConvex

//then secondSoil goes inside Convex polygon.

//Delete duplicating vertices in both polygons and mark intersection INWARD



if ( fabs(convexAngle-firstConvexSecondSoilAngle-secondConvexSecondSoilAngle)<MINDouble )

{

removeDuplicateVerticeAndMarkIntersection(convexPoly, intersectIndex, true);

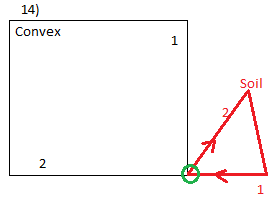
removeDuplicateVerticeAndMarkIntersection(soilPoly, intersectIndex,true);

}

//if angle between first-second edges of Convex polygon is not equal

//sum of angles secondSoil-firstConvex plus secondSoil-secondConvex

//then secondSoil returns outside of Convex polygon. Just delete intersection



else

{

removeIntersection(convexPoly, soilPoly, intersectIndex);

}

return;

}

}

**7)** Function **“regularMarkIntersection”** decides what to do with regular intersection point between edges of two polygons. Main purpose of the function is to decide if intersection is **inward** or **outward.**

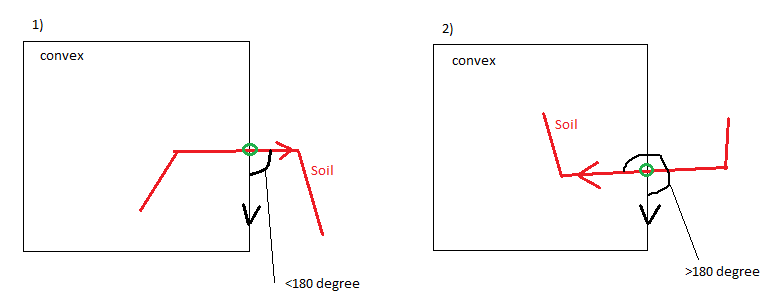
First, we find intersection point in Convex polygon from its index (intersectIndex) and store in “intersectionConvex” variable. Then we find vertex or intersection which is next in Convex Polygon list after intersection node and store in variable “intersectionConvexAfter”.

Second, we find intersection point in Soil polygon from its index (intersectIndex) and store in “intersectionSoil” variable. Then we find vertex or intersection which is next in Soil Polygon list after intersection node and store in variable “intersectionSoilAfter”.

Then, using “vectorOrient” function, we figure out if angle between segments of Soil polygon and Convex polygon coming out of intersection point form angle more or less than 180.

In **Case 1:** returning value of “vector orient” = 1 and angle mentioned above is less than 180 degree, so intersection is **OUTWARD**

In **Case 2:** returning value of “vector orient” = -1 and angle mentioned above is less than 180 degree, so intersection is **INWARD**



//function "regularMarkIntersection" decides what to do with regular intersection between two edges

//it just marks INWARD or OUTWARD

void regularMarkIntersect(Polygon &convexPoly, Polygon &soilPoly, int intersectIndex)

{

//find intersecting point in convex polygon

std::list<Node>::iterator itConvexNodes = convexPoly.polyNodes.begin();

while ( (\*itConvexNodes).index != intersectIndex)

{itConvexNodes++;

if (itConvexNodes==convexPoly.polyNodes.end())

{std::cout<<"\n Intersection number "<<intersectIndex<<"was not found in Convex polygon list - ERROR\n"; exit (EXIT\_FAILURE);}

}

Node intersectionConvex = \*itConvexNodes; //intersection node in Convex polygon

//find vertice or another intersection which is next after intersection in Convex nodes list

Node intersectionConvexAfter; //node in list after intersection node in Convex Polygon

itConvexNodes++;

if (itConvexNodes==convexPoly.polyNodes.end())

{std::cout<<"\n Intersection "<<intersectIndex<<" was last in Convex polygon list - ERROR\n"; exit (EXIT\_FAILURE);}

else intersectionConvexAfter = \*itConvexNodes; //node after intersection

//find intersection point on soil polygon

std::list<Node>::iterator itSoilNodes = soilPoly.polyNodes.begin();

while ( (\*itSoilNodes).index != intersectIndex)

{itSoilNodes++;

if (itSoilNodes==soilPoly.polyNodes.end())

{std::cout<<"\n Intersection number "<<intersectIndex<<" was not found in Soil polygon list - ERROR\n"; exit (EXIT\_FAILURE);}

}

Node intersectionSoil = \*itSoilNodes;

//find vertice or another intersection which is next after intersection in Soil nodes list

Node intersectionSoilAfter; //node in list after intersection node in Soil Polygon

itSoilNodes++;

if (itSoilNodes==soilPoly.polyNodes.end())

{std::cout<<"\n Intersection "<<intersectIndex<<" was last in Soil polygon list - ERROR\n"; exit (EXIT\_FAILURE);}

else intersectionSoilAfter = \*itSoilNodes; //node after intersection

//if orientation between segment of Soil polygon formed by intersection point and next in list of nodes point

//with same segment of Convex polygon

//is 1 - then angle is less than 180 degree (Soil vector goes outside) and intersection is OUTWARD

//is -1 - then angle is more than 180 degree (Soil vector goes inside) and intersection is INWARD

//is 0 - ERROR

if ( vectorOrient(intersectionSoil, intersectionSoilAfter, intersectionConvex, intersectionConvexAfter) == 1)

{

markIntersection(soilPoly, intersectIndex, false);

markIntersection(convexPoly, intersectIndex, false);

}

else if (vectorOrient(intersectionSoil, intersectionSoilAfter, intersectionConvex, intersectionConvexAfter) == -1)

{

markIntersection(soilPoly, intersectIndex, true);

markIntersection(convexPoly, intersectIndex, true);

}

else {std::cout<<"\n Edges with intersection number "<<intersectIndex<<" are collinear - LOGICAL ERROR\n"; exit (EXIT\_FAILURE);}

return;

}

**8)** Function **“markInvard”** get as a parameter two polygons “Convex” and “Soil”. And by looping through all intersections it decides what kind of intersection it is and calls appropriate function to deal with intersection point and mark it inward or outward.

We have two nested loops to go through Soil polygon. While loop is nested in for loop and both going through vertices of Soil Polygon. Outer for loop breaks if variable “nodeDeleted” is false. That means inner while loop went through all vertices smoothly and marked them inward or outward without any deletion of duplicating nodes. If inside while loop the while marking intersection occurred deletion of any node in Soil Polygon (case of duplicating of vertex and intersection at same spot), then while loop breaks and should start again (“nodeDeleted” is assigned to true and outer for loop restarts while loop). We need to do this because after deletion of node from Soil polygon order of vertices is messed and for correct work of “iterator” variable we need to restart loop.

Inside while loop we firs find intersections which are not checked before, then we marked them as “visited”. After that we find same intersection point in Convex Polygon. Then we get next elements (vertices) after intersecting points in lists both polygons. If those next elements are nodes and **both** are at same position or very close to intersection point then we call function “”verticeToVerticeMarkInvard” to deal with intersection. If only in Convex polygon next node after intersection is at same position or very close to intersection node we call “verticeConvexMarkIntersect”. Else if only next node in Soil polygon is at same spot or very close we call function “verticeSoilMarkIntersect”. And else if exclude all this cases we have regular intersection and call function “regularMarkIntersect”.

And Finally we nullify “isVisited” parameter of soil polygon node list becase we will use it later

//function "markInward" marks intersections inward - true if Edge of soil polygon

//with this intersection enters clip polygon (from outside to inside)

void markInward(Polygon &convexPoly, Polygon &soilPoly)

{

int intersectIndex; //index of intersection

int size = soilPoly.polyNodes.size(); //size of soilPolygon before marking and possible deleting duplicates

//for loop iterates iterates as many times as many deletion of vertices are in soil polygon

//if inner while loop iterates through all nodes without deletion for loop breaks

for(int i=1; i<=size; i++)

{

bool nodeDeleted = false; //indicator of deleted node

std::list<Node>::iterator itSoil=soilPoly.polyNodes.begin(); // iterator through soil polygon

//loop through Soil Polygon vertices

while ( itSoil != soilPoly.polyNodes.end())

{

//finding intersection and checking if it was visited and marked before

if ( (\*itSoil).intersection && (!(\*itSoil).visited) )

{

(\*itSoil).visited = true; //mark intersection point as visited

intersectIndex = ((\*itSoil).index);

//looking for same node in convex Polygon

std::list<Node>::iterator it2Convex = convexPoly.polyNodes.begin();

while ( (\*it2Convex).index !=intersectIndex )

{it2Convex++;

if (it2Convex==convexPoly.polyNodes.end())

{std::cout<<"\n Intersection number "<<intersectIndex<<"was found in Soil Polygon, but wasn't found in Convex Polygon - ERROR\n"; exit (EXIT\_FAILURE);}

}

//getting next elements of convex and soil polygons after intersection points

std::list<Node>::iterator itConvexNext = it2Convex;

std::list<Node>::iterator itSoilNext = itSoil;

itConvexNext++;

itSoilNext++;

//if after both intersections are vertices and are very close or same as nodes call function " verticeToVerticeMarkInvard"

if ( (!(\*itConvexNext).intersection) && (!(\*itSoilNext).intersection) && (distTwoPoints(\*itSoilNext, \*itSoil)<NODEAccuracy) && (distTwoPoints(\*itConvexNext, \*it2Convex)<NODEAccuracy) )

{

verticeToVerticeMarkInvard(convexPoly, soilPoly, intersectIndex);

nodeDeleted = true; //in case of vertice to vertice intersection node is always deleted

}

//intersection is close or same as convex polygon node call function " verticeConvexMarkIntersection"

else if ( (!(\*itConvexNext).intersection) && (distTwoPoints(\*itConvexNext, \*it2Convex)<NODEAccuracy))

{

verticeConvexMarkIntersect (convexPoly, soilPoly, intersectIndex);

nodeDeleted = true; //in this case intersection node is delted or vertice near intersection is deleted from convex polygon

}

//intersection is close or same as node of soil polygon call function "verticeSoilMarkIntersect"

else if ( (!(\*itSoilNext).intersection) && (distTwoPoints(\*itSoilNext, \*itSoil)<NODEAccuracy) )

{

verticeSoilMarkIntersect(convexPoly, soilPoly, intersectIndex);

nodeDeleted = true;

}

//if it is regular intersection(not at nodes of polygon) call function "regularMarkIntersect"

else

{

regularMarkIntersect(convexPoly, soilPoly, intersectIndex);

}

}

itSoil++;

if (nodeDeleted)

{ break;} //break the while loop if some nodes are deleted

}

if(!nodeDeleted) break;

}

//nullifying isVisited parameter in soil polygon nodes

for(std::list<Node>::iterator itSoil=soilPoly.polyNodes.begin(); itSoil!=soilPoly.polyNodes.end(); ++itSoil)

{(\*itSoil).visited=false; }

return;

}

9) function “**createIntersection**” Finds all intersections between Convex and Soil polygons, then inserts intersection points in proper place and finally arranges intersections (if more than 1 between two neighboring vertices) in correct order.

All this is done by calling appropriate functions.

//function "createIntersections" finds all intersections of two polygons

// and inserts intersections in lists of nodes

//also function rearranges intersection point of each edge by distance

void createIntersections (Polygon &convexPoly, Polygon &soilPoly)

{

Node intersectionNode; //temp intersection node

int index = 1; //index of intersection node

//loop through edges of both polygons

for (std::list<Edge>::iterator it=convexPoly.polyEdges.begin(); it != convexPoly.polyEdges.end(); ++it)

for (std::list<Edge>::iterator it2=soilPoly.polyEdges.begin(); it2 != soilPoly.polyEdges.end(); ++it2)

{

if (computeIntersection(\*it, \*it2, intersectionNode, index) ) // if has intersection

{index++; //increase index and

insertIntersection (convexPoly, soilPoly, \*it, \*it2, intersectionNode);//insert intersection point in proper place

}

}

rearrangeIntersectionsByDist(convexPoly); //rearrange intersection points of one edge by distance

rearrangeIntersectionsByDist(soilPoly); //rearrange intersection points of one edge by distance

//function "markInward" marks intersections inward - true if Edge of soil polygon

//with this intersection enters clip polygon (from outside to inside)

markInward(convexPoly, soilPoly);

return;

}

10) Function **“isSoilPolyInsideConvex”** is called from main clipping function in case when there are no intersecting areas. The main purpose of the function thus, is to determine why there are no intersection areas between Convex and Soil Polygons. This can happen only in two cases Soil polygon is totally inside or totally outside of Convex Polygon. Function **returns true** if all vertices of Soil polygon are inside of Convex polygon. Function **returns false** if all vertices of Soil polygon are outside of Convex polygon. In case if some of the vertices of Soil Polygon are inside and some outside – function exits with ERROR because this function only intended to be called if Two polygons doesn’t intersect.

//function "isSoilPolyInsideConvex" checks and returns TRUE if all vertices of soil polygon are inside convex polygon

//Else if outside returns FALSE

bool isSoilPolyInsideConvex(Polygon &soilPoly)

{

bool isInside = true;

//check if all vertices of Soil polygon are inside or on the edge of Convex Polygon

for (std::list<Node>::iterator it=soilPoly.polyNodes.begin(); it != soilPoly.polyNodes.end(); ++it)

{ //if at least one outside - mark isInside "false and break for loop"

if ((\*it).insideClipPoly <0)

{

isInside = false;

break;

}

}

//if all vertices of Soil polygon are inside convex - job is done(RETURN)

if (isInside) {return isInside;}

//check if all vertices of Soil polygon are outside or on the edge of Convex Polygon

for (std::list<Node>::iterator it=soilPoly.polyNodes.begin(); it != soilPoly.polyNodes.end(); ++it)

{ //if at least one is inside - mark isInside "true and break for loop"

if ((\*it).insideClipPoly > 0)

{

isInside = true;

break;

}

}

//if all vertices of Soil polygon are outside convex - job is done(RETURN)

if (!isInside) {return isInside;}

//else - neither totally inside nor totally outside - ERROR

else {std::cout<<"\n Some of vertices of Soil Polygon are inside and some outside of Convex. \n Irrelevant case for function - ERROR\n"; exit (EXIT\_FAILURE);}

return true;

}

11) Function **“clipPolygons”** is main function in program that implements Weiler-Atherton Clipping algorithm. Function gets three parameters Convex Polygon, Soil Polygon and third parameter **outputPolyArray** is pointer to Polygon structure where is possible to store array of polygons, because it’s possible to have more than 1 intersecting area. Function returns amount of intersecting areas between polygons.

There are two Boolean variables “secondLapSoil” and “secondLapConvex”. We need these variables to make sure that at least one time through all vertices to reach our starting point – inward intersection in Convex polygon. This is done because if inward intersection is in the middle of polygon list we need to continue up to the end of list and go to second loop to reach starting point.

Main “for” loop goes through Soil Polygon node list to find previously non visited inward intersections. Then we mark “secondLapSoil” and “secondLapConvex” as false because we only start tracing and its first lap. We mark this inward intersection as visited, increment amount of intersecting polygons by one, and store intersection index in “currentTraceIntersectIndex”.

Then we start big do-while loop which is designed to output intersecting polygon. It iterates till we don’t reach starting point – inward intersection in Soil polygon and when we reach it we set variable “endOfOutputList” to true and exit do-while loop. Inside this big do-while loop we have two nested do-while loops. First is to iterate along vertices of Soil polygon and when polygons intersect switch to Convex polygon and continue in second do-while loop to iterate along Convex polygon.

So in main do-while loop we do following operations:

a) We find recorded inward intersection is Soil polygon by “currentTraceIntersectIndex”.

b) Start first inner do-while loop which will **put all nodes of Soil polygon in output list** (in “outputPolyArray”) until we meet intersection with Convex Polygon.

* in this first inner do while loop we check if in two laps we didn’t find intersection with Convex polygon – fire ERROR message
* if we meet visited node before intersection with Convex Polygon – ERROR message

c) After we found intersection with Convex polygon we update “currentTraceIntersectIndex” and mark newly fond intersection as visited.

d) We check if new intersection is INWARD. If it is – we fire ERROR message, because it can’t be two consequent INWARD intersections.

e) Then we find new intersection in Convex polygon list by new “currentTraceIntersectIndex”

f) We start second inner do-while loop which will **put all nodes of Convex polygon in output list** (in “outputPolyArray”) until we meet intersection with Soil Polygon.

* In this loop we check as well if intersection withSoil polygon hasn’t been found in two laps – then ERROR message
* If we bump into visited node then there is two options:

1. Visited node is the node is INWARD intersection from which we started our path (with index “currentIntersectIndex”) then we are finished with current intersection area and mark variable “endOfOutputList” true to exit outer do-while loop.
2. If node was visited but it’s not starting point with index “currentIntersectIndex” – then ERROR message

g) After we exit second do-while loop we check if we are finished tracing current intersection area (“endOfOutputList” is true). If not we assign “currentTraceIntersectIndex” to newly found intersection with Soil polygon in order to continue tracing

h) Lastly we mark newly fond intersection as visited

So “for” loop catches intersection areas and outer do-while loop inside for loop creates each intersection area. After finishing for loop we check how many intersecting areas between two polygons are created (stored in “numberOfIntersectareas”). If zero then there are two possibilities:

1. Soil polygon is totally inside Convex polygon. We check this possibility by calling “**isSoilPolyInsideconvex**” function. If function returns true then we assign whole Soil polygon to **outputPolyArray,** and increase number of intersect areas.
2. If Soil polygon is not inside Convex, so they just don’t intersect and we do nothing.

//main function "clipPolygons" calculates intersection of convex polygon and soil polygon and creates array of intersecting areas

//function returns number of intersecting polygons (maybe more than 1)

int clipPolygons(Polygon &convexPoly, Polygon &soilPoly, Polygon \*outputPolyArray)

{

int numberOfIntersectAreas = 0; //number of intersecting areas for current

bool secondLapSoil; //checks if tracing along soil polygon goes second lap

bool secondLapConvex; //checks if tracing along convex polygon goes second lap

int currentIntersectIndex; //index of inward intersection index which is starting point of current tracing

bool endOfOutputList; //flag for ending main do-while loop that creates output list of current intersection area

//main for loop that goes through Soil Polygon nodes searching INWARD intersections

for (std::list<Node>::iterator itSoilMain=soilPoly.polyNodes.begin(); itSoilMain != soilPoly.polyNodes.end(); ++itSoilMain)

{

//finding inward intersection which wasn't visited previously

if ( (\*itSoilMain).intersection && (\*itSoilMain).inward && !(\*itSoilMain).visited )

{

endOfOutputList = false; //flag for ending main do-while loop that creates output list of current intersection area is false at first

secondLapSoil=false; secondLapConvex=false; //making "second lap" flag false in both polygons

currentIntersectIndex = (\*itSoilMain).index; //storing current intersection index

markVisited(convexPoly, soilPoly, currentIntersectIndex); //marking intersection visited in both polygons

numberOfIntersectAreas++; //incrementing amount of intersecting areas

std::list<Node>::iterator itSoilTrace; //iterator "itSoilTrace" iterates through nodes of soil polygon until meets intersection

std::list<Node>::iterator itConvexTrace; //iterator "itConvexTrace" iterates through nodes of soil polygon until meets intersection

int currentTraceIntersectIndex = currentIntersectIndex; //currentTraceIntersectIndex stores intersection indexes during tracing process

do{

//finding intersecting node in soil polygon corresponding to "currentTraceIntersectIndex"

itSoilTrace = soilPoly.polyNodes.begin();

while ( (\*itSoilTrace).index != currentTraceIntersectIndex )

{

itSoilTrace++;

if (itSoilTrace==soilPoly.polyNodes.end())

{std::cout<<"\n Intersection number "<<currentTraceIntersectIndex<<" was not found in Soil polygon list - ERROR\n"; exit (EXIT\_FAILURE);}

}

/////////tracing SOIL polygon////////////////////////

//putting all nodes from Soil polygon into output list until intersection with Convex met

do{

outputPolyArray[numberOfIntersectAreas-1].polyNodes.push\_back((\*itSoilTrace));

itSoilTrace++;

//check if end of list is reached in Soil polygon

//if end of list is reached on first lap switch to first element

//if in second lap - ERROR Intersection with Convex is not found within two loops through Soil list

if (itSoilTrace ==soilPoly.polyNodes.end())

if(!secondLapSoil)

{

itSoilTrace =soilPoly.polyNodes.begin();

secondLapSoil=true;

}

else {std::cout<<"\n Intersection with Convex Polygon is not found within two loops through Soil list - ERROR\n"; exit (EXIT\_FAILURE);}

//if bump into visited node - ERROR

if ((\*itSoilTrace).visited)

{std::cout<<"\n Visited node was not found on tracing way in Soil polygon list - ERROR\n"; printNode ((\*itSoilTrace)); exit (EXIT\_FAILURE);}

}while ( !(\*itSoilTrace).intersection );

//////////end of tracing SOIL polygon/////////////////

//updating currentTraceIntersectIndex with index of newly fond intersection

currentTraceIntersectIndex = (\*itSoilTrace).index;

//Mark newly fond intersection as visited

markVisited(convexPoly, soilPoly, currentTraceIntersectIndex); //marking intersection visited in both polygons

//Checking if intersection with Convex polygon is inward

//Then ERROR - two consequent INWARD intersections in Soil Polygon

if ((\*itSoilTrace).inward)

{std::cout<<"\n Intersection number "<<currentTraceIntersectIndex<<" is inward in Soil polygon list, but should be outward (because follows inward intersection ) - ERROR\n";

exit (EXIT\_FAILURE);}

//else finding intersecting node in Convex polygon corresponding to "currentTraceIntersectIndex"

else {

itConvexTrace = convexPoly.polyNodes.begin();

while ( (\*itConvexTrace).index != currentTraceIntersectIndex )

{

itConvexTrace++;

if (itConvexTrace==convexPoly.polyNodes.end())

{std::cout<<"\n Intersection number "<<currentTraceIntersectIndex<<" was not found in Convex polygon list - ERROR\n"; exit (EXIT\_FAILURE);}

}

}

/////////tracing CONVEX polygon////////////////////////

//putting all nodes from Convex polygon into output list until intersection with Soil met

do{

//push into output and move to next node

outputPolyArray[numberOfIntersectAreas-1].polyNodes.push\_back((\*itConvexTrace));

itConvexTrace++;

//check if end of list is reached in Convex polygon

//if end of list is reached on first lap switch to first element

//if in second lap - ERROR Intersection with Convex is not found within two loops through Soil list

if (itConvexTrace == convexPoly.polyNodes.end())

if(!secondLapConvex)

{

itConvexTrace =convexPoly.polyNodes.begin();

secondLapConvex=true;

}

else {std::cout<<"\n Intersection with Convex Polygon is not found within two loops through Soil list - ERROR\n"; exit (EXIT\_FAILURE);}

//if bump into visited node - Check if It is starting inward intersection with index "currentIntersectIndex"

//if yes - Tracing of current area is done (mark flag "endOfOutoyutList" true to exit main do-while loop).

//If not - ERROR

if ((\*itConvexTrace).visited )

{ if ( ((\*itConvexTrace).index == currentIntersectIndex) && (\*itConvexTrace).intersection)

endOfOutputList=true;

else {std::cout<<"\n Visited node was not found on tracing way in Soil polygon list - ERROR\n"; printNode ((\*itSoilTrace)); exit (EXIT\_FAILURE);}

}

}while ( !(\*itConvexTrace).intersection );

//////////end of tracing CONVEX polygon/////////////////

//checking if it is not "endOfOutputList"

if (!endOfOutputList)

{

//updating currentTraceIntersectIndex with index of newly fond intersection

currentTraceIntersectIndex = (\*itConvexTrace).index;

//Mark newly fond intersection as visited

markVisited(convexPoly, soilPoly, currentTraceIntersectIndex); //marking intersection visited in both polygons

}

} while(!endOfOutputList);

}

}

//catching case when soil polygon is totally inside or totally outside of soil polygon, if no intersection was fond

//if is totally inside - copy Soil polygon in output polygon

if (numberOfIntersectAreas ==0)

{

if (isSoilPolyInsideConvex(soilPoly))

{

outputPolyArray[0].polyNodes.assign (soilPoly.polyNodes.begin(),soilPoly.polyNodes.end());

numberOfIntersectAreas++;

}

}

return numberOfIntersectAreas;

}

/////////\*\*\*End of structure and functions needed for clipping (By Dimitri)\*\*//////////

12) Next functions are designed to incorporate polygon clipping function with “Regions” GUI designed by Robert Weber.

i) Function **“soilPolygonsFromTPolygons”** translates soil polygons from “TPolygon” structures (user input in “Regions” program) to “Polygon” structure. It has soil “TPolygon” and integer size of Tpolygon as a parameter, and returns same soil polygon as “Polygon” structure for use in clipping algorithm.

First we make sure that polygon has more than 2 nodes (otherwise – ERROR message). Then we have for loop which loops through soil “TPolygon” nodes. Inside for loop we first create node (entity for “Polygon” structure) using **“makeNode”** function. Then we push created node in “tempPolygon” variable which will be returned. Along with that in same for loop we create edges from neighboring nodes and also push created edges in returning “tempPolygon”.

//Function "soilPolygonsFromTPolygons" makes Polygon from TPolygon array assigns Array coordinates to list

Polygon soilPolygonsFromTPolygons(TPolygon soilArray, int size )

{

Polygon tempPolygon;

//making sure that soil polygon has more than 2 nodes, else - ERROR

if (size<=2) {std::cout<<"\n non-valid soil polygon(less than 3 nodes) - ERROR\n"; exit (EXIT\_FAILURE);}

Node tempNode; //temporary node for list of nodes

Node edgeStartNode, edgeEndNode; //temporary nodes for list of edges

Edge tempEdge; //temporary edge for list of edges

//constructing beginning node for edge list

edgeStartNode = makeNode(soilArray.points[size-1].x, soilArray.points[size-1].y, false, false, 0, -1, false);

tempEdge.start = edgeStartNode;

for (int i=0; i<size; i++)

{

//making temporary node from array of points

tempNode = makeNode(soilArray.points[i].x, soilArray.points[i].y, false, false, 0, -1, false);

//pushing temporary node in polygon structure in list of nodes

tempPolygon.polyNodes.push\_back(tempNode);

//making temporary edge from nodes

edgeEndNode = tempNode;

tempEdge.end = edgeEndNode;

//pushing temporary edge in polygon struct in list of edges

tempPolygon.polyEdges.push\_back(tempEdge);

tempEdge.start = tempEdge.end;

}

return tempPolygon;

}

ii) Auxiliary function **“highestY”** returns the highest (largest) Y coordinate **among nodes of all Soil polygons.** This function will be used in “assignFailurePlanePolygon” function.

//Function "highestY" returns highest coordinate Y from all soil polygons

double highestY( TPolygon \* polygons, int numberOfPolygons)

{ double high = 0.0;

for (int i=0; i<numberOfPolygons; i++)

{

for (int j=0; j<polygons[i].numberOfPoints; j++)

{

if (polygons[i].points[j].y>high) high = polygons[i].points[j].y;

}

}

return high;

}

iii) Auxiliary function **“leftMostX”** returns leftmost (smallest) X coordinate **among all of nodes of all Soil polygons.** This function will be used later in “assignFailurePlanePolygon” function.

//function "leftMostX" returns leftmost (smallest) X coordinate of all soil polygons

double leftMostX( TPolygon \* polygons, int numberOfPolygons)

{

double left = 1000000.0;

for (int i=0; i<numberOfPolygons; i++)

{

for (int j=0; j<polygons[i].numberOfPoints; j++)

{

if (polygons[i].points[j].x<left) left = polygons[i].points[j].x;

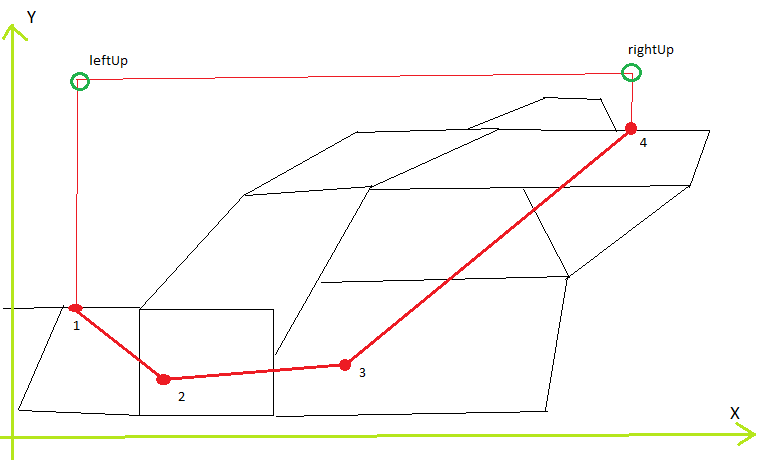
}

}

return left;

}

iiii) Function **“assignFailurePlanePolygon”** makes Convex polygon (“Polygon” type used in clipping algorithm) from failure plane (“TFailurePlane” type used in “Regions” GUI). Function accepts as a parameters: failure array, size of failure array, largest Y coordinate among all Soil polygon nodes (black color on picture), and smallest X coordinate among all nodes of soil polygons (black on picture). Function returns Convex polygon. The main difficulty while writing this function was the fact that **failure plane given us from user of “Regions” program was not closed polygon** but only broken line (bold red in picture) consisted from points (1,2,3,4). While for clipping algorithm **we need closed, convex polygon**. So the main purpose of **“assignFailurePlanePolygon”** is to **make two additional nodes “leftUp” and “rightUp” and construct closed convex polygon** (red color on picture).



First we make sure that polygon has more than 2 nodes (otherwise – ERROR message).

Then, in for loop, we push all existing failure plane vertices (1,2,3,4) in “tempPolygon” variable.

We check if Failure plane is oriented from left to right (point #1 in picture is first in array). If it is then we check if leftmost point (point #1) is below highest soil node. If it is not – then ERROR.

We choose coordinates of “leftUp” vertex of Convex polygon as (leftMostX, highestY+1.0). **Thus “leftUp” point is created straight above leftmost point in failure plane (point #1) and 1.0 unit higher than highest of Soil polygon nodes (black color).** However to avoid collinear edges in Convex polygon **we check if line segments of convex polygon: 1-‘”leftUp” and 2-1 are collinear**. If they are we slightly offset “leftUp” vertex by 0.1 to the left.

Then we check if we need “rightUp” point at all. If Y coordinate of last point of Failure plane (point #4) is less than Y coordinate of highest node of Soil polygons (“highestY” value) as in picture, then we assign variable “needRightUp” to true. At the same time we print warning for user that highest point of Failure plane is below highest Soil level and suggest to extend failure plane. However, in this case program doesn’t throw Error and continues to work by creating “rightUp” vertex.

To create “rightUp” vertex, we **firstly check if last segment of failure plane (3-4 on picture) is vertical.** If it’s not we create **“rightUp” vertex straight above rightmost point in failure plane (point #4) with same X coordinate as point #4 and with Y coordinate 1.0 higher than highest of Soil polygon nodes (highestY+1.0).** If last segment in failure plane is vertical we create “rightUp” point not straight vertical above point #4, but offset it to right by 0.1 value.

After that we push “leftUp” node in front of existing points in Convex polygon, and if we needed and created “rightUp” node, we push it at the back of existing list in Convex polygon.

If failure plane is oriented from right to left (point #1 is located at rightmost and point #4 is leftmost) then use same steps and same logic to construct Convex polygon (stored in “tempPolygon”) from failure plane.

After that we construct edges of Convex polygon from list of its nodes and return it.

//Function "assignFailurePlanePolygon" makes convex polygon list from failurePlane array

Polygon assignFailurePlanePolygon( TFailurePlane failureArray, int size, double highestY, double leftMostX)

{

Polygon tempPolygon;

Node tempNode;

//making sure that polygon has more than 2 nodes, else - ERROR

if (size<=1) {std::cout<<"\n non-valid failure plane(less than 2 nodes) - ERROR\n"; exit (EXIT\_FAILURE);}

//two additional points needed to make polygon from failure plane array

Node leftUp, rightUp;

//variable for checking if we need to extend right side of failure plane UP

bool needRightUp = false;

//\*\*\*\*Push Failure array into tempPolygon\*\*\*\*\*

for (int i=0; i<size; i++)

{

//making temporary node from array of points

tempNode = makeNode(failureArray.points[i].x, failureArray.points[i].y, false, false, 0, -1, false);

//pushing temporary node in polygon structure in list of nodes

tempPolygon.polyNodes.push\_back(tempNode);

}

//\*\*Crate Additional points (or point) and push them in TempPolygon\*\*

//if Failure plane is oriented - from LEFT to RIGHT

if (failureArray.points[0].x<failureArray.points[size - 1].x)

{

//two or one additional points are created depending on orientation of failure surface

//check if first (leftMost point of failure surface is below highest soil level - else ERROR )

if (failureArray.points[0].y >= highestY)

{std::cout<<"\n LeftMost Point in Failure plane should be below Highest Soil level - ERROR\n"; exit (EXIT\_FAILURE);}

//check to avoid collinear edges on left side of failure plane with additional leftUp point

double rize1 = failureArray.points[0].y - (highestY+1.0);

double run1 = failureArray.points[0].x - leftMostX;

double rize2 = failureArray.points[1].y - failureArray.points[0].y;

double run2 = failureArray.points[1].x - failureArray.points[0].x;

//if LeftUp point is on line created by first and second points of failure plane

//move leftUp point by -0.1 left to avoid collinear edges

if (fabs(rize1/run1 - rize2/run2) < MINDouble)

{leftUp = makeNode(leftMostX-0.1, highestY+1.0, false, false, 0, -1, false);}

else {leftUp = makeNode(leftMostX, highestY+1.0, false, false, 0, -1, false);}

//check if we need to extend Right side of fracture plane

if (failureArray.points[size - 1].y<highestY)

{

needRightUp = true;

std::cout<<"\n WARNING - Highest point of failure plane is below Highest soil level.\n";

std::cout<<"Better start over and extend failure plane - below results are for failure plane extended vertically or almost vertically.\n\n";

//check if last segment in failure plane is vertical

if (fabs(failureArray.points[size - 1].x - failureArray.points[size - 2].x) < MINDouble)

//if vertical move rightUp additional point by 0.1 left to avoid collinear edges

{rightUp = makeNode(failureArray.points[size - 1].x-0.1, highestY+1.0, false, false, 0, -1, false);}

else {rightUp = makeNode(failureArray.points[size - 1].x, highestY+1.0, false, false, 0, -1, false);}

}

else //this else case assignment should never be used - is done for insure that rightUp node is not "null"

{rightUp = makeNode(failureArray.points[size - 1].x, highestY+1.0, false, false, 0, -1, false);}

//\*\*push additional points in TempPolygon for Convex Polygon\*\*

tempPolygon.polyNodes.push\_front(leftUp); //push left upper node

if(needRightUp) tempPolygon.polyNodes.push\_back(rightUp); //push right upper node if needed

}

//CASE if Failure plane is oriented - from RIGHT to LEFT

else

{ //two or one additional points are created depending on orientation of failure surface

//check if last (leftMost point of failure surface is below highest soil level - else ERROR )

if (failureArray.points[size-1].y >= highestY)

{std::cout<<"\n LeftMost Point in Failure plane should be below Highest Soil level - ERROR\n"; exit (EXIT\_FAILURE);}

//check to avoid collinear edges on left side of failure plane with additional leftUp point

double rize1 = failureArray.points[size-1].y - (highestY+1.0);

double run1 = failureArray.points[size-1].x - leftMostX;

double rize2 = failureArray.points[size-2].y - failureArray.points[size-1].y;

double run2 = failureArray.points[size-2].x - failureArray.points[size-1].x;

//if LeftUp point is on line created by last and before points of failure plane

//move leftUp point by -0.1 left to avoid collinear edges

if (fabs(rize1/run1 - rize2/run2) < MINDouble)

//if vertical move left additional point by -0.1 left to avoid collinear edges

{leftUp = makeNode(leftMostX-0.1, highestY+1.0, false, false, 0, -1, false);}

else {leftUp = makeNode(leftMostX, highestY+1.0, false, false, 0, -1, false);}

//check if we need to extend Right side of fracture plane

if (failureArray.points[0].y<highestY)

{

needRightUp = true;

std::cout<<"\n WARNING - Highest point of failure plane is below Highest soil level.\n";

std::cout<<"\n Better start over and extend failure plane - below results are for failure plane extended vertically or almost vertically.\n";

//check if last segment in failure plane is vertical

if (fabs(failureArray.points[0].x - failureArray.points[1].x) < NODEAccuracy)

//if vertical move rightUp additional point by 0.1 left to avoid collinear edges

{rightUp = makeNode(failureArray.points[0].x-0.1, highestY+1.0, false, false, 0, -1, false);}

else {rightUp = makeNode(failureArray.points[0].x, highestY+1.0, false, false, 0, -1, false);}

}

else //this else case assignment should never be used - is done for insure that rightUp node is not "null"

{rightUp = makeNode(failureArray.points[0].x, highestY+1.0, false, false, 0, -1, false);}

//\*\*push additional points in TempPolygon\*\*

tempPolygon.polyNodes.push\_back(leftUp); //push left upper node

if (needRightUp) tempPolygon.polyNodes.push\_front(rightUp); //push right upper node if needed

}

//\*\*Create list of TempPolygon Edges from list of it's Nodes\*\*

Edge tempEdge; //temporary edge for list of edges

//constructing beginning node for first edge from last node in Node list

tempEdge.start = tempPolygon.polyNodes.back();

//looping through list of Nodes

for (std::list<Node>::iterator itNodes=tempPolygon.polyNodes.begin(); itNodes != tempPolygon.polyNodes.end(); ++itNodes)

{

tempEdge.end = \*itNodes; //making last element of current Edge

//pushing temporary edge in polygon struct in list of edges

tempPolygon.polyEdges.push\_back(tempEdge);

tempEdge.start = tempEdge.end; //making first node of current Edge for next iteration of loop

}

return tempPolygon;

}

iiiii) Function **“assignIntersectionPolygon”** makes opposite transition. It accepts as a parameter list of nodes (nodes of resulting intersection polygons) and returns “TPolygon” structure which is standard for “Regions” GUI to show intersecting polygon on screen.

If size of ”intersectPolylist” is bigger than 0, then we just loop through the list from beginning to end and assign coordinates of vertices to “returnPolygon” variable of type “Tpolygon”. Finally we return “returnPolygon” .

//function "assignIntersectionPolygon" makes and returns TPolygon from list of intersections

TPolygon assignItersectionPolygon (std::list<Node> & intersectPolyList)

{

TPolygon returnPolygon;

int size = intersectPolyList.size();

returnPolygon.numberOfPoints = size;

if (size>0)

{

int i=0;

for (std::list<Node>::iterator it=intersectPolyList.begin(); it != intersectPolyList.end(); ++it)

{

returnPolygon.points[i].x = (\*it).x;

returnPolygon.points[i].y = (\*it).y;

i++;

}

}

return returnPolygon;

}

13) Function “**polyArea**” calculates area of intersection polygons. For calculating area of polygon we use well known and easy to implement “Shoelace formula” - <https://en.wikipedia.org/wiki/Shoelace_formula>

Implementation can be found here: <https://www.geeksforgeeks.org/area-of-a-polygon-with-given-n-ordered-vertices/>

//function "polyArea" calculates area of polygon polygon is presented as an array

double polyArea(TPolygon myTPolygon)

{

double area = 0.0;

// Calculate value of shoelace formula

int j = myTPolygon.numberOfPoints-1;

for (int i = 0; i < myTPolygon.numberOfPoints; i++)

{

area += (myTPolygon.points[j].x + myTPolygon.points[i].x) \* (myTPolygon.points[i].y - myTPolygon.points[j].y);

j = i; // j is previous vertex to i

}

// Return absolute value

return abs(area / 2.0);

}

14) **“main”** function.

In main function we perform all steps needed to call appropriate functions for calculating intersection polygons between Failure Surface (Convex polygon) and all Soil polygons.

**“mySoilPolygons”** – array of all Soil polygons

**“myFailurePlanePolygons” –** array of Convex polygons created from all failure planes

**myIntersections [numberOfFailurePlanes] [numberOfPolygons] [MAXIntersectAreas]** – 3 dimensional array of intersection polygons. First index indicates belonging to failure plane. Second index indicates which Soil polygon intersection is with. And third index indicates how many intersecting regions we have (one Soil polygon can intersect with failure plane in more than 1 region).

**indexArray2D [numberOfFailurePlanes] [numberOfPolygons]** – is two dimensional integer array that stores amount of intersecting areas for each pair of Failure plane and Soil polygon. Values stored in this array are used as third index (**[MAXIntersectAreas]**) of **“myIntersections”** array.

**intersectionPolygons[numberOfFailurePlanes][numberOfPolygons][MAXIntersectAreas]** – 3 dimensional array of intersecting polygons of “TPolygon” type. They correspond to polygons in “myIntersection” variable and are used as output in “Regions” program.

**areas [numberOfFailurePlanes] [numberOfPolygons]** – 2 dimensional array. Values of this array are Areas of intersection of Failure plane (first index of array) and Soil Polygon (second index of array).

First we populate array “mySoilPolygons” from “TPolygon” type arrays accepted from user in “Regions”. In same “for” loop we check winding of polygon and change it to clockwise if it is counterclockwise.

Then we store value of Y coordinate of highest among all nodes of Soil polygons in “yHigh”, and X coordinate of leftmost among all nodes of Soil polygons in “xLeft”.

Then we populate array of Convex polygons “myFailurePlanePolygons” from failure plane arrays of type “TFailurePlane” accepted from user. In same “for” loop, while creating Convex polygons we check them for correct orientation and change orientation if it isn’t correct.

After that we have the most important nested “for” loop that calls functions which calculate intersections along with printing out nodes of intersection polygons. We also assign intersection polygons to “**intersectionPolygons”** array for GUI output.

a) Outer “for” loop loops through Failure planes (Convex polygons).

b) In outer loop we print out nodes of Convex polygons.

c) Then we have inner “for” loop that loops through all Soil polygons

d) Assign currently tested Convex (failure plane) and Soil polygons to temporary variables in order to keep original structure intact

e) Call **“labelInside”** function for currently testing Soil and Convex polygons in order to mark their vertices if they are inside or outside of other polygon

f) Call “**createIntersection**” in order to find, arrange, etc. intersecting points between current pair of Convex and Soil polygons

g) Call “**clipPolygons**” function to calculate intersections between current Convex and Soil polygons and store intersecting polygons in **myIntersection[i][j]**. function returns amount of intersecting polygons and this integer value is stored in **indexArray2D.**

h) After that we print out current Soil polygon

i) Then we have “for” loop in which we print resulting intersection polygons. In this for loop we also assign intersecting polygon lists to “TPolygon” type **intersectionPolygons[i][j][k]** array for GUI “Regions” output.

After that we have auxiliary test output that (3 nested “for” loops) that prints out all intersecting polygon node coordinates from converted “TPolygon” type **intersectionPolygons** array.

And lastly we have nested “for” loop for calculating areas of intersections. All areas of intersections between Convex polygon and Soil polygon are stored then in **areas[i][j]** array(first index – Failure plane number, second index – Soil polygon number). In same for loop areas are printed out.

int main(int argc, char\*\* argv) {

//Declaring Polygon type arrays for clipping algorithm

Polygon mySoilPolygons [numberOfPolygons]; //array of soil polygons

Polygon myFailurePlanePolygons [numberOfFailurePlanes]; //array of failure plane polygons

Polygon tempSoilPolygon, tempConvexPolygon; //polygons to use in main polygon intersection algorithm (in order to leave Polygons in arrays intact)

Polygon myIntersections [numberOfFailurePlanes] [numberOfPolygons][MAXIntersectAreas]; //3-dimensional array of Polygons of intersections (failure planes with soil polygons)

int indexArray2D [numberOfFailurePlanes] [numberOfPolygons]; //2-dimensional array that stores amount of intersecting areas for each Convex-Soil pair of polygons

TPolygon intersectionPolygons[numberOfFailurePlanes][numberOfPolygons][MAXIntersectAreas]; //3-dimensional array of TPolygons with intersections for output

//table of areas for intersections for output (first index number of planes, second index number of polygons)

double areas [numberOfFailurePlanes] [numberOfPolygons];

//Making array "mySoilPolygons" of Soil Polygons (lists) from TPolygon arrays

for (int i=0; i<numberOfPolygons; i++)

{

//making Soil Polygon from Array

mySoilPolygons[i] = soilPolygonsFromTPolygons(polygons[i],polygons[i].numberOfPoints);

//change winding if is counter clockwise

if ( windingTestSoilPolygon(mySoilPolygons[i]) == -1) windingReverse(mySoilPolygons[i]);

}

//\*\*Making array "myFailurePlanePolygons" of Convex Polygons (lists) from failure plane TPolygon arrays\*\*

//Calculating additional point needed for creating polygons from Failure Plane points

double yHigh = highestY(polygons, numberOfPolygons);//calculating biggest y coordinate among all coordinates of soil polygons

double xLeft = leftMostX(polygons, numberOfPolygons);//calculating smallest (leftMost) x coordinate among all coordinates of soil polygons

//std::cout << "\nThe highest Y among polygon coordinates " <<yHigh<<" The leftmost X among polygon coordinates "<<xLeft;

for (int i=0; i<numberOfFailurePlanes; i++)

{

//making Convex failure Polygon from array

myFailurePlanePolygons[i] = assignFailurePlanePolygon(failurePlanes[i], failurePlanes[i].numberOfPoints, yHigh, xLeft);

//change winding if is counter clockwise

if ( windingTestClipPolygon(myFailurePlanePolygons[i])== -1) windingReverse(myFailurePlanePolygons[i]);

}

//main for loop that calls most important intersection calculating functions

//calculating and printing out intersection polygon points

//also assigning intersection lists to Intersection array table for GUI output

//"i" - loops through failure (Convex) polygons

for (int i=0; i<numberOfFailurePlanes; i++)

{

//printing out failure plane

std::cout << "\n----------------------------------------------------\n";

std::cout << "\nThe intersections for failure plane # " <<i+1<<", (polygon nodes)\n";

printPolygonNodes( myFailurePlanePolygons[i]);

//"j" - loops through SOil Polygons

for (int j=0; j<numberOfPolygons; j++)

{

tempConvexPolygon = myFailurePlanePolygons[i]; //assigning failure (convex) polygon to temporary polygon for using in intersection algorithm

tempSoilPolygon = mySoilPolygons[j]; //assigning soil polygon to temporary polygon for using in intersection algorithm

//label inside points of soilPolygon

labelInside(tempConvexPolygon, tempSoilPolygon);

//label inside points of convexPolygon

labelInside(tempSoilPolygon, tempConvexPolygon);

//creating and marking intersection points

createIntersections(tempConvexPolygon, tempSoilPolygon);

//calculating intersection of two polygons and assigning number of intersecting areas to index array

indexArray2D [i][j] = clipPolygons(tempConvexPolygon, tempSoilPolygon, myIntersections [i][j]);

//printing out soil polygon

std::cout << "\n\*\*\* The SoilPolygon # " <<j+1<<". \*\*\*\n";

printPolygonNodes(tempSoilPolygon);

//printing out intersection polygons

std::cout << "\n The intersection polygon(s): \n";

std::cout << "\n The soil polygon # "<<j+1<<" has: "<<indexArray2D [i][j]<<" intersecting areas with failure surface #"<<i+1<<" \n";

//assigning intersection polygon lists to TPolygon arrays for GUI output and printout all intersecting ares (if exist)

for(int k=0; k< indexArray2D [i][j]; k++)

{

intersectionPolygons [i][j][k] = assignItersectionPolygon(myIntersections[i][j][k].polyNodes);

std::cout << " Intersecting area (polygon) # "<<k+1<<" \n";

printPolygonNodes(myIntersections[i][j][k]);

}

}

}

//testing purpose

std::cout<<"\nChecking assignment mechanism of assignItersectionPolygon function by printing out intersecting TPolygon arrays\n";

for (int i=0; i<numberOfFailurePlanes; i++)

{

std::cout << "\nFailure plane #"<<i+1<<"\n";

for(int j=0; j<numberOfPolygons; j++)

{

std::cout << "\n Soil Polygon #"<<j+1<<"\n";

std::cout << " amount of intersecting areas: "<<indexArray2D [i][j]<<"\n";

for(int k=0; k< indexArray2D [i][j]; k++)

{

std::cout << " intersection Polygon #"<<k+1<<"\n";

for (int index=0; index<intersectionPolygons[i][j][k].numberOfPoints; index++)

{

std::cout << " ("<<intersectionPolygons[i][j][k].points[index].x<<", "<<intersectionPolygons[i][j][k].points[index].y<<")";

}

}

}

}

//end Test

//Calculating areas of intersection polygons

for (int i=0; i<numberOfFailurePlanes; i++)

{

for (int j=0; j<numberOfPolygons; j++)

{

areas[i][j] =0;

for(int k=0; k< indexArray2D [i][j]; k++)

{

areas[i][j]= areas[i][j] + polyArea(intersectionPolygons [i][j][k]);

}

std::cout << "\nThe intersection area: Failure plane: "<<i<<" Polygon soil: "<<j<<"\n";

std::cout <<" Area: "<<areas[i][j]<<"\n";

}

}

return 0;

}