/\*

\* File: main.cpp

\* Author: p302450 (Dimitri Eiramjani)

\*

\* Created on February 28, 2018, 2:15 PM

\*/

#include <cstdlib>

#include <iostream>

#include <limits>

#include <math.h>

#include <stdlib.h>

#include <list>

#include <algorithm>

#include <vector>

//Chunk of code with definition from Robert's program

#define MAXIMUM\_NUMBER\_OF\_POLYGONS 10

#define MAXIMUM\_NUMBER\_OF\_POINTS\_PER\_POLYGON 10

#define MAXIMUM\_NUMBER\_OF\_FAILURE\_PLANES 5

#define MAXIMUM\_NUMBER\_OF\_POINTS\_PER\_FAILURE\_PLANE 10

#define MAXIMUM\_NUMBER\_OF\_INTERSECTION\_POINTS 20

#define Accuracy 0.01 // accuracy for == or != of TpointF

//my define constants

#define MINDouble 1.0e-9

#define PI 3.141592653589793

#define MAXIntersectAreas 5

#define NODEAccuracy 0.001

//structures fromRobert's program

struct TPointF

{

double x;

double y;

bool operator==(TPointF Position)

{

/\*

if (fabs(x-Position.x)<=Accuracy && fabs(y-Position.y)<=Accuracy)

// if (x == Position.x && y == Position.y)

{return true;}

else

{return false;}

\*/

double deltaX = x-Position.x;

double deltaY = y-Position.y;

// double distanceXY = sqrt(deltaX\*deltaX+deltaY\*deltaY);

double distanceXY = sqrt(pow(deltaX,2.0)+pow(deltaY,2.0));

if (distanceXY <= Accuracy)

{

return true;

}

else

{

return false;

}

}

bool operator!=(TPointF Position)

{

/\*

if (fabs(x-Position.x)>Accuracy || fabs(y-Position.y)>Accuracy)

// if (x != Position.x || y != Position.y)

{return true;}

else

{return false;}

\*/

double deltaX = x-Position.x;

double deltaY = y-Position.y;

// double distanceXY = sqrt(deltaX\*deltaX+deltaY\*deltaY);

double distanceXY = sqrt(pow(deltaX,2.0)+pow(deltaY,2.0));

if (distanceXY > Accuracy)

{

return true;

}

else

{

return false;

}

}

};

struct TPolygon

{

int numberOfPoints;

TPointF points[MAXIMUM\_NUMBER\_OF\_POINTS\_PER\_POLYGON];

};

struct TFailurePlane

{

int numberOfPoints;

TPointF points[MAXIMUM\_NUMBER\_OF\_POINTS\_PER\_FAILURE\_PLANE];

};

//------------------------------------------------------------------------------------------------------------

// Public variables and functions.

//

// The GUI will fill up the Input Variables and then pass control to TRegionsProcessor by calling the

// ProcessData function. The ProcessData function will call all the routines necessary to process the

// Input Variables and fill up the Output Variables. The GUI will then read these Output Variables and

// display the output.

//------------------------------------------------------------------------------------------------------------

//------------------------------------------------------------------------------------------------------------

// Input Variables

//------------------------------------------------------------------------------------------------------------

int numberOfPolygons;

TPolygon polygons[MAXIMUM\_NUMBER\_OF\_POLYGONS];

int numberOfFailurePlanes;

TFailurePlane failurePlanes[MAXIMUM\_NUMBER\_OF\_FAILURE\_PLANES];

int useFailurePlane;

//------------------------------------------------------------------------------------------------------------

// Output Variables

//------------------------------------------------------------------------------------------------------------

int numberOfIntersectionPoints;

TPointF intersectionPoints[MAXIMUM\_NUMBER\_OF\_INTERSECTION\_POINTS];

//------------------------------------------------------------------------------------------------------------

// Calculation Process Function

//------------------------------------------------------------------------------------------------------------

bool ProcessData();

//Original Structures and functions for clip polygon (by Dimitri Eiramjani)

//type Node

struct Node {

double x; //x coordinate

double y; // y coordinate

bool intersection; //is intersection point 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 Point

struct Point {

double x; //x coordinate

double y; // y coordinate

};

//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;

};

//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 "makeNode" returns node with given parameters assigned to node fields

Node makeNode(double x, double y, bool intersection, bool inward, int insideClipPoly, int index, bool visited)

{

Node tempNode;

tempNode.x = x;

tempNode.y = y;

tempNode.intersection = intersection;

tempNode.inward = inward;

tempNode.insideClipPoly = insideClipPoly;

tempNode.index = index;

tempNode.visited = visited;

return tempNode;

}

//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;

}

//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;

}

//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;

}

//function "angleBetweenVector" calculates angle in radians

//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;

}

//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 >MINDouble) return 1;

else if (orient < -MINDouble) return -1;

else return 0;

}

//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 >MINDouble) return -1;

else if (orient < -MINDouble) return 1;

else return 0;

}

//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" 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); }

}

//Function "windingReverse" inverts order of points and edges

//in Polygon structure (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;

}

//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;

}

//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)>NODEAccuracy){

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)>NODEAccuracy) {

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;

}

}

//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;

}

//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;

}

//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;

}

//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;

}

//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;

}

//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;

}

//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;

}

//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;

}

//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;

}

}

//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;

}

//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;

}

//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;

}

//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;

}

//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)\*\*//////////

//Functions to incorporate polygon program with GUI----------------

//----------------------------------------------------------------

//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;

}

//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;

}

//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;

}

//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;

}

//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;

}

//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);

}

//----------------------------------------------------------------

//End of functions which incorporate program structure with GUI

/\*

\*

\*/

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;

}