Geometry

James Scoon

Sep 23, 2020

CONTENTS:

CHAPTER

ONE

FULL API

1.1 Namespaces

1.1.1 Namespace std

1.2 Classes and Structs

1.2.1 Struct Intersection

- Defined in file_include_Geometry.hpp

Struct Documentation

`struct Intersection`

Object that holds information about an intersection.

Public Members

IntersectionType **`type`**

The type of intersection. This may be of type NONE, POINT, LINE or PLANE.

void ***`object`**

The intersection object, held as a void pointer to allow casting to its original form.

1.2.2 Struct SLESolution

- Defined in file_include_Geometry.hpp

Struct Documentation

`struct SLESolution`

Object that holds information about the solution of a system of equations.

Public Members

SLENumSolutions **`numSolutions`**

The number of solutions. This may be of type ZERO, ONE or INFINITE.

vector<long double> **`solutionVector`**

A vector containing the solution if there is a solution.

1.2.3 Class Line

- Defined in file_include_Geometry.hpp

Class Documentation

`class Line`

Lines are represented by a starting point and an ending point, and can also be used as line segments.

Public Functions

`Line`(*Point s*, *Point e*)

Constructs a line from two point classes.

Parameters

- `s`: Starting point.
- `e`: Ending point.

`Line`(long double *sx*, long double *sy*, long double *ex*, long double *ey*)

Constructs a line from the components of two dimensional points.

Parameters

- `sx`: X component of the starting point.
- `sy`: Y component of the starting point.
- `ex`: X component of the ending point.
- `ey`: Y component of the ending point.

`Line`(long double *sx*, long double *sy*, long double *sz*, long double *ex*, long double *ey*, long double *ez*)

Constructs a line from the components of three dimensional points.

Parameters

- `sx`: X component of the starting point.
- `sy`: Y component of the starting point.
- `sz`: Z component of the starting point.
- `ex`: X component of the ending point.

- `ey`: Y component of the ending point.
- `ez`: Z component of the ending point.

long double **`GetLength`**()
Returns the length of the current line segment

Point **`GetNearestPoint`**(*Point p*)
Returns the point on the line closest to another point in space given by the input.

Parameters

- `p`: The input point.

Point **`GetNearestPointInSegment`**(*Point p*)
Returns the point in the line segment closest to another point in space given by the input.

Parameters

- `p`: The input point.

Point **`GetVector`**()
Converts the line into a vector or the difference between the start and end points.

bool **`IsCollinear`**(*Point p*)
Returns true if a point is collinear to the line.

Parameters

- `p`: The point that is checked.

bool **`IsInSegment`**(*Point p*)
Returns true if a point is contained within a line segment.

Parameters

- `p`: The point that is checked.

bool **`IsParallel`**(*Line l*)
Returns true if the line is parallel to another line.

Parameters

- `l`: The line that is checked.

bool **`IsPerpendicular`**(*Line l*)
Returns true if the line is perpendicular to another line.

Parameters

- `l`: The line that is checked.

bool **`Equals`**(*Line l*)
Returns true if the input line and the current line are the same.

Parameters

- `l`: The line that is checked.

bool **`EqualsSegment`**(*Line l*)
Returns true if the input line segment and the current line segment have the exact same place regardless of direction.

Parameters

- `l`: The line that is checked.

Intersection **GetIntersection** (*Line l*)
Checks for intersections with another line and returns an intersection object.

Return An intersection object of type line if both lines are the same, of type point if they intersect at a point or of type none if no intersection could be found.

Parameters

- `l`: The line that the current line may intersect with.

Intersection **GetSegmentIntersection** (*Line l*)
Checks for intersections with another line segment and returns an intersection object.

Return An intersection object of type line segment if both input segments are collinear, of type point if both input segments meet at a point or of type none if no intersection could be found. If the intersection is of type line segment, it only contains the overlapping region of both input segments.

Parameters

- `l`: The line segment that the current line may intersect with.

Point **GetInterpolation** (long double *fraction*)
Returns a point at an intermediate position of the line segment.

Parameters

- `fraction`: A decimal value that corresponds to the location of the interpolation, where zero yields the starting point and one yields the ending point. This value does not necessarily have to be between zero and one.

Public Members

Point **start**
The starting point of the line or line segment.

Point **end**
The ending point of the line or line segment.

1.2.4 Class Matrix

- Defined in file_include_Geometry.hpp

Class Documentation

class Matrix
Matrices hold a table of values that can be used to apply linear transformations to points.

Public Functions

`Matrix`(int *rows*, int *cols*)

Constructs a zero matrix of a given size.

Parameters

- `rows`: The number of rows in the zero matrix.
- `cols`: The number of columns in the zero matrix.

`Matrix`(vector<vector<long double>> *m*)

Constructs a matrix from a two dimensional C++ vectors.

Parameters

- `m`: A two dimensional C++ vector.

`Matrix`(**`const`** *Matrix* &*m*)

Constructs a copy of another matrix.

Parameters

- `m`: The matrix to copy.

long double **`Get`**(int *row*, int *col*)

Returns the value of the matrix at a given row and column.

Parameters

- `row`: The row to access.
- `col`: The column to access.

void **`Set`**(int *row*, int *col*, long double *value*)

Changes the value of the matrix at a given row and column to a given amount.

Parameters

- `row`: The row to access.
- `col`: The column to access.
- `value`: The new value at that position.

void **`SwapRow`**(int *a*, int *b*)

Swaps the positions of two rows within the matrix.

Parameters

- `a`: The index of the first row to swap.
- `b`: The index of the second row to swap.

void **`SwapColumn`**(int *a*, int *b*)

Swaps the positions of two columns within the matrix.

Parameters

- `a`: The index of the first column to swap.
- `b`: The index of the second column to swap.

void **`DeleteRow`**(int *row*)

Deletes a row from the matrix.

Parameters

- `row`: The index of the row to delete.

void **`DeleteColumn`** (int *col*)
Deletes a column from the matrix.

Parameters

- `col`: The index of the column to delete.

void **`InsertRow`** (int *row*)
Inserts a row of zeroes into the matrix.

Parameters

- `row`: The index at which the row is inserted.

void **`InsertColumn`** (int *col*)
Inserts a column of zeroes into the matrix.

Parameters

- `col`: The index at which the column is inserted.

int **`GetRows`** ()
Returns the number of rows held by a matrix.

int **`GetColumns`** ()
Returns the number of columns held by a matrix.

Matrix **`GetTranspose`** ()
Returns the transpose of the matrix

Matrix **`GetSubmatrix`** (int *rows*, int *cols*, int *startRow* = 0, int *startCol* = 0)
Returns a submatrix of a given size that is offset a certain amount.

Return The resulting submatrix.

Parameters

- `rows`: The number of rows in the submatrix.
- `cols`: The number of columns in the submatrix.
- `startRow`: The row at which the submatrix starts or is offset.
- `startCol`: The column at which the submatrix starts or is offset.

Point **`Multiply`** (*Point Xvec*)
Multiplies or Applies a linear transformation onto an input vector, assuming that the current matrix is the transformation matrix. This can be expressed by the equation AX = Y, where A is the current matrix, X is the input vector and Y is the output vector.

Return The vector Y that is the output of the transformation.

Parameters

- `Xvec`: The vector X that is multiplied by the transformation matrix A.

long double **`GetDeterminant`** ()
Returns the determinant of the current matrix assuming it is square.

SLESolution **`SolveEquations`** (*Point Yvec*)
Solves a system of equations of the form AX = Y for the vector X, assuming that the current matrix A is the transformation matrix.

Return An *SLESolution* object that is of type NONE if the system has no solutions, ONE if there is one solution or INFINITE if there are infinitely many solutions. If there is one solution, the *SLESolution* object also stores this vector.

Parameters

- `Yvec`: The vector Y in the equation.

Public Members

vector<vector<long double>> **`matrix`**
A two dimensional vector that holds all of the coefficients.

1.2.5 Class NSphere

- Defined in file_include_Geometry.hpp

Class Documentation

`class NSphere`
NSpheres are regions or spaces bounded by all points that are a certain radius away from a center point. This means that the *NSphere* is capable of representing circles, spheres and higher dimensional objects such as hyperspheres. The rank of the *NSphere* is determined by the rank of the center point.

Public Functions

`NSphere`(*Point* *c*, long double *r*)
Constructs an *NSphere* from a center point object and a given radius.

Parameters

- `c`: The center point, whose rank determines the rank of the *NSphere*.
- `r`: The radius of the *NSphere*.

`NSphere`(long double *x*, long double *y*, long double *r*)
Constructs a two dimensional *NSphere* or a circle given the coordinates of the center and the radius.

Parameters

- `x`: The X coordinate of the center of the circle.
- `y`: The Y coordinate of the center of the circle.
- `r`: The radius of the circle.

`NSphere`(long double *x*, long double *y*, long double *z*, long double *r*)
Constructs a three dimensional *NSphere* or a sphere given the coordinates of the center and the radius.

Parameters

- `x`: The X coordinate of the center of the sphere.
- `y`: The Y coordinate of the center of the sphere.
- `z`: The Z coordinate of the center of the sphere.
- `r`: The radius of the sphere.

bool **`ContainsPoint`**(*Point* *p*)

Returns true if the input point is within or on the surface of the *NSphere*.

Parameters

- `p`: The point that is checked.

bool **`PointOnBorder`**(*Point* *p*)

Returns true if the input point is on the surface of the *NSphere*.

Parameters

- `p`: The point that is checked.

bool **`Intersects`**(*NSphere* *s*)

Returns true if the volume of the input *NSphere* intersects with the current *NSphere*.

Parameters

- `s`: The *NSphere* that is checked.

bool **`FullyContains`**(*NSphere* *s*)

Returns true if the volume of the input *NSphere* is fully contained by the current *NSphere*.

Parameters

- `s`: The *NSphere* that is checked.

long double **`GetSurfaceArea`**()

Returns the surface area of the *NSphere*.

long double **`GetVolume`**()

Returns the volume of the *NSphere*.

Point **`GetNearestPointOnSurface`**(*Point* *p*)

Returns the point on the surface of the *NSphere* closest to another point in space given by the input.

Parameters

- `p`: The input point.

Public Members

Point **`center`**

Center point of the *NSphere*.

long double **`radius`**

Radius of the *NSphere*, which represents the distance between the center point and all boundary points.

1.2.6 Class Plane

- Defined in file_include_Geometry.hpp

Class Documentation

`class Plane`

Planes are infinitely large flat surfaces that are represented by a center position and a normal vector.

Public Functions

`Plane` (*Point c*, *Point a*, *Point b*)

Constructs a plane from three points that represent two vectors CA and CB that lie on the plane.

Parameters

- `c`: Starting point C of both vectors and intersection point of the plane.
- `a`: Ending point A of the vector CA.
- `b`: Ending point B of the vector CB.

`Plane` (*Point c*, *Point n*)

Constructs a plane from an intersecting point and a normal vector to the plane.

Parameters

- `c`: Center point C that intersects the plane.
- `n`: Normal vector to the plane.

bool **`ContainsPoint`** (*Point p*)

Returns true if the input point lies on the plane.

Parameters

- `p`: The point that is checked.

bool **`ContainsLine`** (*Line l*)

Returns true if the input line lies on the plane.

Parameters

- `l`: The line that is checked.

bool **`IsParallel`** (*Line l*)

Returns true if the input line is parallel to the plane.

Parameters

- `l`: The line that is checked.

bool **`IsParallel`** (*Plane p*)

Returns true if the input plane is parallel to the current plane.

Parameters

- `p`: The plane that is checked.

bool **`IsPerpendicular`** (*Line l*)

Returns true if the input line is perpendicular to the plane.

Parameters

- `l`: The line that is checked.

bool **`IsPerpendicular`** (*Plane p*)

Retuns true if the input plane is perpendicular to the current plane.

Parameters

- `p`: The plane that is checked.

Intersection **`GetIntersection`** (*Line l*)

Checks for intersections between the plane and an input line and returns an intersection object.

Return An intersection object of type line if the input line lies on the plane, of type point if the line goes through the plane at a point or of type none if no intersection was found.

Parameters

- `l`: The line that the plane may intersect with.

Intersection **`GetIntersection`** (*Plane p*)

Checks for intersections between the current plane and another input plane and returns an intersection object.

Return An intersection object of type plane if both planes are the same, of type line if they intersect at a line or of type none if no intersection was found.

Parameters

- `p`: The plane that the current plane may intersect with.

Point **`GetNearestPoint`** (*Point p*)

Returns the point on the plane closest to another point in space given by the input.

Parameters

- `p`: The input point.

Public Members

Point **`center`**

Center point or intersecting point of the plane.

Point **`normal`**

Normal vector to the plane.

1.2.7 Class Point

- Defined in file_include_Geometry.hpp

Class Documentation

`class Point`

Points are objects that can contain any number of components. They can also be considered vectors.

Public Functions

`Point` (long double *x*, long double *y*)

Constructs a point in two dimensions given the X and Y coordinates.

Parameters

- `x`: X coordinate of the point.
- `y`: Y coordinate of the point.

`Point` (long double *x*, long double *y*, long double *z*)
Constructs a point in three dimensions given the X, Y and Z coordinates.

Parameters

- `x`: X coordinate of the point.
- `y`: Y coordinate of the point.
- `z`: Z coordinate of the point.

`Point` (vector<long double> *coords*)
Constructs a point from an arbitrary set of coordinates.

Parameters

- `coords`: An N dimensional array or vector of coordinates.

`Point` (**`const`** *Point* &*p*)
Creates a copy of another point.

Parameters

- `p`: The point to copy.

int **`GetRank`** ()
Returns the rank or the number of dimensions the point contains.

long double **`GetX`** ()
Returns the first component of the point.

long double **`GetY`** ()
Returns the second component of the point.

long double **`GetZ`** ()
Returns the third component of the point.

long double **`Get`** (int *index*)
Returns the value of the component at a certain index.

Parameters

- `index`: Index of the component.

void **`SetX`** (long double *x*)
Sets the first component of the point.

Parameters

- `x`: The value to set the first component to.

void **`SetY`** (long double *y*)
Sets the second component of the point.

Parameters

- `y`: The value to set the second component to.

void **`SetZ`** (long double *z*)
Sets the third component of the point.

Parameters

- `z`: The value to set the third component to.

void **`Set`** (int *index*, long double *value*)
Sets the value of the component at a certain index.

Parameters

- `index`: Index of the component.
- `value`: Value of the component.

long double **`SqrMagnitude`**()
: Returns the square of the magnitude of the vector.

long double **`Magnitude`**()
: Returns the magnitude of the vector.

void **`Add`**(*Point p*)
: Adds the components of another point onto those of the current point.

Parameters

- `p`: The point to add onto the current point.

void **`Subtract`**(*Point p*)
: Subtracts the components of another point from those of the current point.

Parameters

- `p`: The point to subtract from the current point.

void **`Scale`**(long double *factor*)
: Scales the vector by a given factor.

Parameters

- `factor`: The value to scale the point by.

void **`Normalize`**()
: Normalizes or sets the magnitude of the current vector to be equal to one.

void **`Rotate2D`**(long double *angle*, *Point center* = *Point*(0, 0))
: If the current point is two dimensional, the function rotates the point by a certain angle.

Parameters

- `angle`: The angle in radians to rotate the point by. If the angle is positive, the rotation will be counterclockwise.
- `center`: The center of rotation. By default this is the origin.

void **`Rotate3D`**(long double *angle*, *Point normal*, *Point center* = *Point*(0, 0, 0))
: If the current point is three dimensional, the function rotates the point by a certain angle around a normal vector.

Parameters

- `angle`: The angle in radians to rotate the point by. If the angle is positive, the rotation will be counterclockwise.
- `normal`: A vector that represents the axis of rotation.
- `center`: The center of rotation. By default this is the origin.

long double **`DotProduct`**(*Point p*)
: Returns the dot product of the current point and another point.

Parameters

- `p`: The other point in the dot product.

Point **`CrossProduct`** (*Point* *p*)

Returns the cross product of the current point and another point. This function does not work for vectors of more than three dimensions.

Parameters

- `p`: The other point in the cross product.

Point **`operator+`** (*Point* *p*) **`const`**

Returns the addition of the current point and another point.

Parameters

- `p`: The point added to the current point.

Point **`operator-`** (*Point* *p*) **`const`**

Returns the subtraction of the current point and another point.

Parameters

- `p`: The point subtracted from the current point.

Point **`operator*`** (long double *amount*) **`const`**

Multiplies the magnitude of the vector by a given amount.

Parameters

- `amount`: The amount to multiply the vector by.

Point **`operator/`** (long double *amount*) **`const`**

Divides the magnitude of the vector by a given amount.

Parameters

- `amount`: The amount to divide the vector by.

bool **`operator==`** (*Point* *p*) **`const`**

Checks if two points are equal to each other, taking into account possible precision errors.

Return True if both points are practially equal, or false if otherwise.

Parameters

- `p`: The point compared to the current point.

bool **`operator<`** (*Point* *p*) **`const`**

Checks if a point is less than another point, taking into account possible precision errors.

Return True if all the components of the first point are less than those of the other point, or false if otherwise.

Parameters

- `p`: The point compared to the current point.

Public Members

vector<long double> **`coordinates`**
Vector of components or coordinates.

1.2.8 Class Polygon

- Defined in file_include_Geometry.hpp

Inheritance Relationships

Derived Type

- `public Triangle` (*Class Triangle*)

Class Documentation

`class Polygon`
Polygons are used to represent two dimensional shapes bounded by a set of points. The vertices should be ordered in a counter clockwise direction, otherwise the class may not work properly.

Subclassed by *Triangle*

Public Functions

`Polygon`(vector<*Point*> *points*)
Constructs a polygon from a vector or array of *Point* objects

Parameters

- `points`: A C++ vector of point objects.

long double **`GetArea`**()
Returns the area bounded by the polygon.

long double **`GetPerimeter`**()
Returns the perimeter of the bounds of the polygon.

bool **`IsConvex`**()
Returns true if the polygon is a convex shape.

bool **`ContainsPoint`**(*Point p*)
Returns true if the input point is contained within the polygon or on the border of the polygon.

Parameters

- `p`: The point that is checked.

bool **`PointOnBorder`**(*Point p*)
Returns true if the input point is on the border of the polygon.

Parameters

- `p`: The point that is checked.

void **`RemoveDuplicates`**()
Removes duplicate points from the polygon, those of which may cause certain functions to fail.

Line **`GetEdge`**(int *index*)
Returns a line segment which represents one of the edges of the polygon.

Parameters

- `index`: The index of the edge the function should return.

Public Members

vector<*Point*> **`vertices`**
The vertices of the polygon.

1.2.9 Class Quaternion

- Defined in file_include_Geometry.hpp

Class Documentation

`class Quaternion`
Quaternions are used to accurately represent rotations in three dimensions.

Public Functions

`Quaternion`(long double *qr*, long double *qx*, long double *qy*, long double *qz*)
Constructs a quaternion given its four components.

Parameters

- `qr`: The real component of the quaternion.
- `qx`: The X component of the quaternion.
- `qy`: The Y component of the quaternion.
- `qz`: The Z component of the quaternion.

`Quaternion`(long double *r*, *Point* *v*)
Constructs a quaternion from the real component and a three dimensional vector.

Parameters

- `r`: The real component of the quaternion.
- `v`: A three dimensional vector containing the X, Y and Z components.

`Quaternion`(**`const`** *Quaternion* &*q*)
Constructs a copy of another quaternion

Parameters

- `q`: The quaternion to copy.

long double **`GetReal`**()
Returns the real component of the quaternion.

Point **`GetImaginary`**()
Returns a vector or point containing the X, Y and Z components of the quaternion.

long double **`GetX`**()
Returns the X component of the quaternion.

long double **`GetY`**`()`
    Returns the Y component of the quaternion.

long double **`GetZ`**`()`
    Returns the Z component of the quaternion.

long double **`SqrMagnitude`**`()`
    Returns the square of the magnitude of the quaternion.

long double **`Magnitude`**`()`
    Returns the magnitude of the quaternion.

void **`Normalize`**`()`
    Normalizes or scales the magnitude of the quaternion to one.

Quaternion **`Conjugate`**`()`
    Returns the conjugate of the quaternion, where the imaginary vector is inverted.

Quaternion **`Inverse`**`()`
    Inverts the current quaternion.

Quaternion **`operator+`**`(`*Quaternion q*`)` **`const`**
    Returns the addition of two quaternions.

Parameters

- `q`: The quaternion to add to the current quaternion.

Quaternion **`operator-`**`(`*Quaternion q*`)` **`const`**
    Returns the subtraction of two quaternions.

Parameters

- `q`: The quaternion to subtract from the current quaternion.

Quaternion **`operator*`**`(`*Quaternion q*`)` **`const`**
    Returns the product of two quaternions.

Parameters

- `q`: The quaternion to multiply by the current quaternion.

Quaternion **`operator/`**`(`*Quaternion q*`)` **`const`**
    Returns the division of two quaternions.

Parameters

- `q`: The quaternion to divide by the current quaternion.

bool **`operator==`**`(`*Quaternion q*`)` **`const`**
    Checks if two quaternions are equal, taking into account possible precision errors.

Parameters

- `q`: The quaternion that is compared to the current quaternion.

Public Members

long double **`real`**
Real or first component of the quaternion.

long double **`x`**
X or second component of the quaternion.

long double **`y`**
Y or third component of the quaternion.

long double **`z`**
Z or fourth component of the quaternion.

1.2.10 Class Triangle

- Defined in file_include_Geometry.hpp

Inheritance Relationships

Base Type

- `public Polygon` (*Class Polygon*)

Class Documentation

`class Triangle` : **`public`** *Polygon*
Triangles are subclasses of polygons that only contain three points.

Public Functions

`Triangle``(`*Point a*, *Point b*, *Point c*`)`
Constructs a triangle from three points.

Parameters

- `a`: The first point of the triangle.
- `b`: The second point of the triangle.
- `c`: The third point of the triangle.

bool **`HasRightAngle`**`()`
Returns true if the triangle has a right or 90° angle.

int **`GetEqualSides`**`()`
Returns a value which indicates the number of sides that have equal length.

NSphere **`GetCircumcircle`**`()`
Returns the smallest circle that fully contains the triangle.

NSphere **`GetIncircle`**`()`
Returns the largest circle that is fully contained by the triangle.

1.3 Enums

1.3.1 Enum IntersectionType

- Defined in file_include_Geometry.hpp

Enum Documentation

`enum IntersectionType`

Used to tell the type of object in an *Intersection*.

Values:

`enumerator NONE`

`enumerator POINT`

`enumerator LINE`

`enumerator PLANE`

1.3.2 Enum SLENumSolutions

- Defined in file_include_Geometry.hpp

Enum Documentation

`enum SLENumSolutions`

Used to classify the number of solutions that a system of equations yields.

Values:

`enumerator ZERO`

`enumerator ONE`

`enumerator INFINITE`

1.4 Functions

1.4.1 Function Angle

- Defined in file_include_Geometry.hpp

Function Documentation

long double **`Angle`** (*Point a*, *Point b*, *Point c*)

Returns the angle between two vectors BA and BC. These two vectors have a common starting point B.

Return Angle in radians.

Parameters

- `a`: The point A which corresponds to the ending point of the first vector.
- `b`: The starting point B of both vectors.
- `c`: The point C which corresponds to the ending point of the second vector.

1.4.2 Function CCW

- Defined in file_include_Geometry.hpp

Function Documentation

bool **`CCW`** (*Point a*, *Point b*, *Point c*)

Checks if two vectors BA and BC are closer in the counterclockwise direction. These two vectors have a common starting point B.

Return True if BA and BC are closer in the counterclockwise direction, or false if otherwise.

Parameters

- `a`: The point A which corresponds to the ending point of the first vector.
- `b`: The starting point B of both vectors.
- `c`: The point C which corresponds to the ending point of the second vector.

1.4.3 Function DegToRad

- Defined in file_include_Geometry.hpp

Function Documentation

long double **`DegToRad`** (long double *deg*)

Converts degrees into radians.

Return Angle in radians.

Parameters

- `deg`: Angle in degrees.

1.4.4 Function RadToDeg

- Defined in file_include_Geometry.hpp

Function Documentation

long double **`RadToDeg`** (long double *rad*)

Converts radians into degrees.

Return Angle in degrees.

Parameters

- `rad`: Angle in radians.

1.4.5 Function Zero

- Defined in file_include_Geometry.hpp

Function Documentation

Point **`Zero`** (int *rank*)

Returns a point corresponding to the origin or a zero vector with a given rank.

Parameters

- `rank`: The rank or the number of dimensions of the point.

1.5 Defines

1.5.1 Define EPS

- Defined in file_include_Geometry.hpp

Define Documentation

`EPS`

Precision error constant.

1.5.2 Define PI

- Defined in file_include_Geometry.hpp

Define Documentation

`PI`

The mathematical constant Pi.

INDEX