

SECTION 2 GEOMETRY	3
2.1 Parametric Geometry	4
2.1.1 Parametric Line	4
2.1.2 Parametric Circle	4
2.1.3 Parametric Aligned Ellipse	5
2.1.4 Rotated Parametric Ellipse	5
2.1.5 Parametric Line Intersection	6
2.2 Coordinate Transforms	7
2.2.1 Rotation & Translation Matrices	8
2.2.2 Scaling Matrices [TBD]	12
2.2.3 Orthographic Projection Matrix [TBD]	12
2.2.4 Perspective Projection Matrix [TBD]	12
2.3 Vectors and Unit Vectors [TBD]	12
2.3.1 Vectors	12
2.3.2 Unit Vectors	12
2.3.3 Vector Rotation	13
2.3.4 Vector Scaling [TBD]	15
2.4 Projective Geometry and Homogeneous Coordinates	15
2.4.1 3D Homogeneous Coordinate Space [TBD]	15
2.4.2 2D Homogeneous Coordinate Space [TBD]	16
2.5 Homogeneous Point Equations	16
2.5.1 Translate Point by a Vector [TBD]	17
2.5.2 Rotate Point about the Origin [TBD]	18
2.5.3 Rotate Point about a Point and a Vector [TBD]	19
2.5.4 Project Point to a Plane [TBD]	20
2.6 Homogeneous (Plücker) Line Equations	21
2.6.2 Line from a Point and a Vector	23
2.6.3 Line from Two Points	24
2.6.4 Principal Point of a Line [TBD]	24
2.6.5 Point of Closest Approach to Line [TBD]	25
2.6.6 Validation Examples	27
2.6.7 Translate Line by a Vector [TBD]	27
2.6.8 Rotate Line about the Origin	28
2.6.9 Rotate a Line about a Point and a Vector [TBD]	29
2.6.10 Project Line to a Plane [TBD]	30
2.7 Homogeneous Plane Equations	30
2.7.2 Plane from a Point and a Normal Vector	31
2.7.3 Plane from Three Points	32
2.7.4 Principal Point of a Plane [TBD]	33
2.7.5 Translate Plane by a Vector [TBD]	33
2.7.6 Rotate Plane about the Origin [TBD]	34
2.7.7 Rotate Plane about a Point and a Vector [TBD]	34
2.8 Geometric Intersections	34
2.8.1 Intersection of a Line and a Plane	34
2.8.2 Intersection of Two Planes [TBD]	35
2.9 Geometric Projections	35
2.9.1 Projection of a Point onto a Plane [TBD]	35
2.9.2 Projection of a Point onto a Plane (2D) [TBD]	35
2.9.3 Projection of a Line onto a Plane [TBD]	36
2.9.4 Projection of a Line onto a Plane (2D) [TBD]	36
2.10 Polygon Routines	36

2.10.1	Point Inside Polygon	36
2.10.2	Create Simple Concave Hull Polygon	38
2.10.3	Create Convex Hull Polygon	39
2.10.4	Boolean Polygon Operations	40
2.11	Point Sorting and Bounding Algorithms [TBD]	44
2.11.1	2D/3D Vector Point Sorting	44
2.11.2	Directional Bounding Box [TBD]	45
2.11.3	Minimal Bounding Box [TBD]	45
2.12	Geometric Reference	45
2.12.1	Area Calculations	45
2.12.2	Volume Calculations [TBD]	46
2.13	New Ideas	46
2.13.1	Minimal Bounding Circle [TBD]	46
2.13.2	Minimal Bounding Sphere [TBD]	47
2.13.3	Circle from Two Points [TBD]	47
2.13.4	Circle from Two Points and a Radius [TBD]	47
2.13.5	Circle from Three Points [TBD]	48
2.13.6	Regress a Circle from N-Points [TBD]	48
2.13.7	Sphere from Two Points [TBD]	49
2.13.8	Sphere from Three Points [TBD]	49

SECTION 2 GEOMETRY

2.1 Parametric Geometry

Most geometric objects are described in terms of an explicit or implicit equation relying on one component being solved for in terms of the other components. For instance the explicit and implicit equations for a 2D line are:

$$\begin{aligned}\text{Explicit: } & y = Mx + b \\ \text{Implicit: } & Ax + By + C = 0\end{aligned}$$

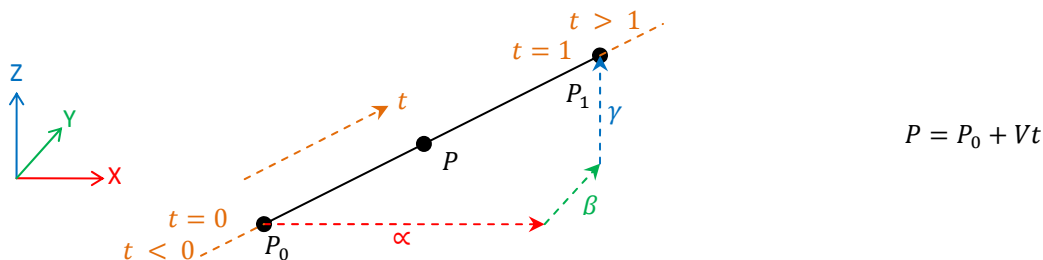
The explicit case solves for y in terms of x , while the implicit case can be solved for x or y in terms of the other. Both equations force one variable to be the independent and one to be the dependent.

Parametric geometry is an alternative way to represent geometric objects. One or more additional parameters are defined that allow both x and y to be solved for as dependent variables. The parametric representation can be quite useful in describing geometric objects.

2.1.1 Parametric Line

Description: The parametric equation of a line describes an n -dimensional line in terms of its vector components and an independent real number t that essentially represents the percentage distance between the two defining points of the line.

The general parametric equation for a line is:



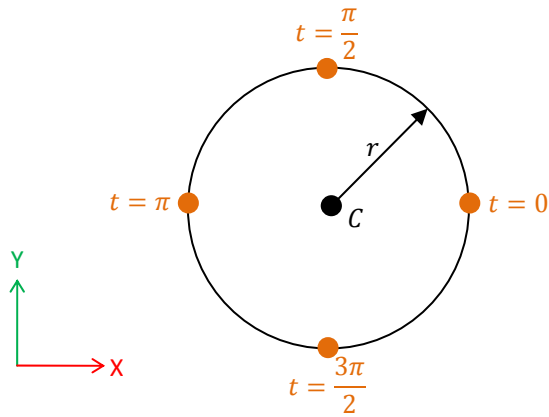
Where P is any point on the line, P_0 is the first of the two endpoints, V is the direction vector of the line, and t is the independent variable for the line. In three dimensions, the general equation is broken into the three equations for the each directional component.

$$\begin{aligned}x &= x_0 + \alpha \cdot t & \rightarrow & \alpha = x_1 - x_0 \\ y &= y_0 + \beta \cdot t & \rightarrow & \beta = y_1 - y_0 \\ z &= z_0 + \gamma \cdot t & \rightarrow & \gamma = z_1 - z_0\end{aligned}$$

The independent variable t ranges from 0 to 1 on the line segment between the points, but can range from $-\infty < t < \infty$. This allows the location of any point P to be found on the line.

2.1.2 Parametric Circle

Description: The parametric circle equations describe a 2D circle in terms of the independent t that represents the angle of the circle at each point.



$$x = x_c + r \cdot \cos(t)$$

$$y = y_c + r \cdot \sin(t)$$

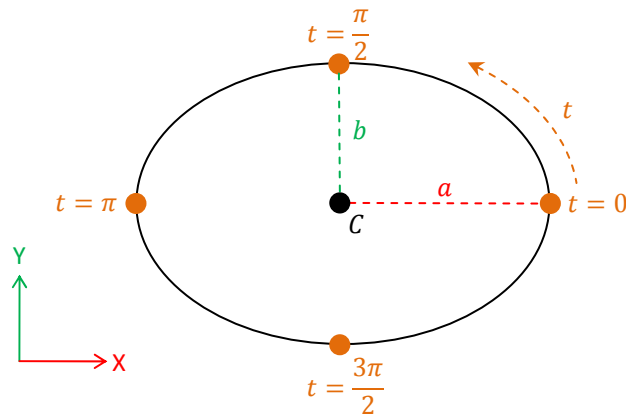
The independent variable t can have values from $-\infty < t < \infty$ but is more commonly restrained to $0 < t < 2\pi$. This allows any circle to be drawn in 2-dimensions.

The parametric computation is less complicated to calculate than solving the standard implicit equation for a circle:

$$(x - a)^2 + (y - b)^2 = r^2$$

2.1.3 Parametric Aligned Ellipse

Description: The parametric ellipse equations describe an ellipse in two dimensions that is aligned with the X and Y axes.



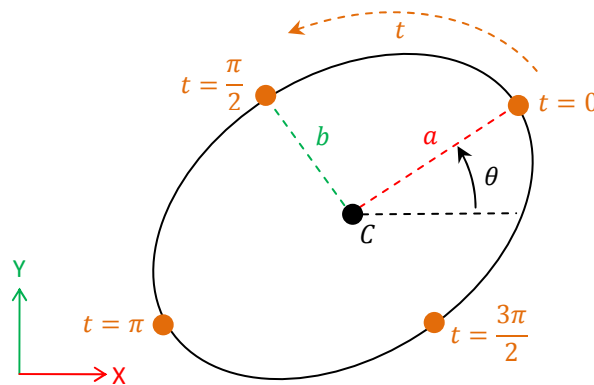
$$x = x_c + a \cdot \cos(t)$$

$$y = y_c + b \cdot \sin(t)$$

The independent variable t can have values from $-\infty < t < \infty$ but is more commonly restrained to $0 < t < 2\pi$. This allows any ellipse with its minor and major axes aligned to the x, y-axes to be drawn in 2-dimensions.

2.1.4 Rotated Parametric Ellipse

Description: The parametric ellipse equations describe an ellipse in two dimensions that is rotated about the z-axis by an angle θ .



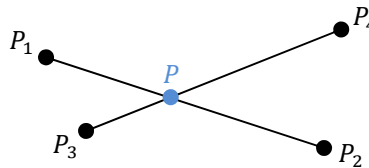
$$x = x_c + a \cdot \cos(t) \cdot \cos(\theta) - b \cdot \sin(t) \cdot \sin(\theta)$$

$$y = y_c + a \cdot \cos(t) \cdot \sin(\theta) + b \cdot \sin(t) \cdot \cos(\theta)$$

The independent variable t can have values from $-\infty < t < \infty$ but is more commonly restrained to $0 < t < 2\pi$. This allows any ellipse with any orientation to be drawn in 2-dimensions.

2.1.5 Parametric Line Intersection

Description: Finds the point of intersection between two 2D parametric lines.



Using the parametric line equations:

$$x = x_0 + \alpha \cdot t$$

$$y = y_0 + \beta \cdot t$$

Setting the parametric equations in terms of the lines above:

$$x = x_1 + (x_2 - x_1) \cdot s \quad \text{and} \quad x = x_3 + (x_4 - x_3) \cdot t$$

$$y = y_1 + (y_2 - y_1) \cdot s \quad \quad \quad y = y_3 + (y_4 - y_3) \cdot t$$

The two pairs of equations can be converted to a linear system of equations by setting the two x equations equal and setting the two y equations equal.

$$x_1 + (x_2 - x_1) \cdot s = x_3 + (x_4 - x_3) \cdot t$$

$$y_1 + (y_2 - y_1) \cdot s = y_3 + (y_4 - y_3) \cdot t$$

The equations can be rearranged to solve in terms of s and t .

$$x_{21} \cdot s - x_{43} \cdot t = x_{31}$$

$$y_{21} \cdot s - y_{43} \cdot t = y_{31}$$

Where:

$$\begin{aligned}x_{21} &= x_2 - x_1 \\y_{21} &= y_2 - y_1 \\x_{31} &= x_3 - x_1 \\&\dots\end{aligned}$$

These equations can now be converted to matrix form.

$$\begin{bmatrix}x_{21} & x_{43} \\y_{21} & y_{43}\end{bmatrix} \cdot \begin{bmatrix}s \\t\end{bmatrix} = \begin{bmatrix}x_{31} \\y_{31}\end{bmatrix}$$

And solved for s and t .

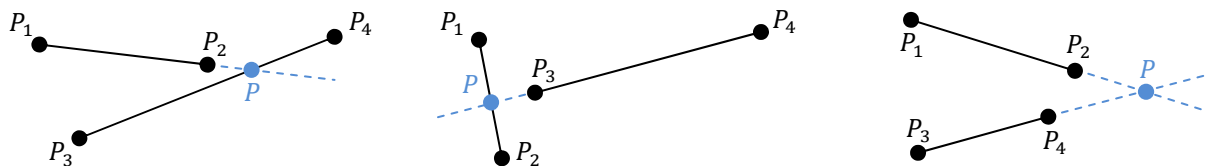
$$\begin{bmatrix}s \\t\end{bmatrix} = \begin{bmatrix}x_{21} & x_{43} \\y_{21} & y_{43}\end{bmatrix}^{-1} \cdot \begin{bmatrix}x_{31} \\y_{31}\end{bmatrix}$$

$$\begin{bmatrix}s \\t\end{bmatrix} = \frac{1}{x_{21}y_{43} - x_{43}y_{21}} \cdot \begin{bmatrix}y_{43} & -x_{43} \\-y_{21} & x_{21}\end{bmatrix} \cdot \begin{bmatrix}x_{31} \\y_{31}\end{bmatrix}$$

$$s = \frac{x_{31}y_{43} - x_{43}y_{31}}{x_{21}y_{43} - x_{43}y_{21}} \quad \text{and} \quad t = \frac{x_{21}y_{31} - x_{31}y_{21}}{x_{21}y_{43} - x_{43}y_{21}}$$

The values of s and t give the percent distance that the intersection is between the endpoints on each line. If the values of s and t are between 0 and 1, then the intersection point lies internal to the two line segments. If s or t are greater than 1 or less than 0, the lines intersect but at some point external point. If the lines are parallel the denominator of both equations will become 0 and cause the value of s and t to become infinite. This is as expected since parallel lines are defined as meeting at infinity.

The physical interpretation of s and t are shown below. The left hand image shows a case where $s > 1$ and $0 \leq t \leq 1$. The point is internal in the second line, but off of the first line. The middle shows the opposite. In this case $0 \leq s \leq 1$ and $t < 0$. The point is internal in the first line but off of the second. The right figure shows the most common case, where the intersection is outside of both lines. In this figure $s > 1$ and $t > 1$.



The s and t values can then be used to calculate the location of the intersection point. Using s or t , plug back into the original parametric line equation and solve for x and y .

$$\begin{aligned}x &= x_1 + (x_2 - x_1) \cdot s \\y &= y_1 + (y_2 - y_1) \cdot s\end{aligned}$$

This method, while more laborious than some other Cartesian intersection calculations, is useful in that it automatically solves for the location of the intersection point on the lines.

2.2 Coordinate Transforms

The coordinate transforms shown in this section allow points to be translated, rotated or projected to other locations using homogeneous coordinate transformations.

2.2.1 Rotation & Translation Matrices

Rotation and translation matrices allow points to easily be moved around in a coordinate system, or moved from one coordinate system to another. All the following transformation matrices are in 3D homogenous coordinate format. For more information on homogeneous coordinates, see *Projective Geometry and Homogeneous Coordinates (Chapter 2.4)*

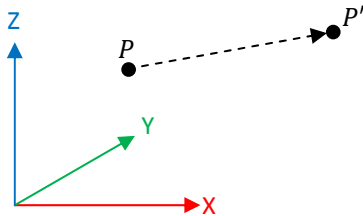
2.2.1.1 Translation

Description: The translation matrix translates a point in 3D Cartesian space to another location specified by an offset vector (dx, dy, dz) .

[TBD: Add mathematical basis]

$$T = \begin{bmatrix} 1 & 0 & 0 & dx \\ 0 & 1 & 0 & dy \\ 0 & 0 & 1 & dz \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

A point P is translated to position P' converting the point P to a homogeneous representation and multiplying by the rotation matrix.



$$\begin{bmatrix} P'_x \\ P'_y \\ P'_z \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & dx \\ 0 & 1 & 0 & dy \\ 0 & 0 & 1 & dz \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} P_x \\ P_y \\ P_z \\ 1 \end{bmatrix}$$

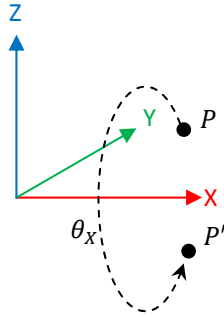
2.2.1.2 Rotation About X-Axis

Description: The x-axis rotation matrix rotates a point in 3D Cartesian space around the x-axis of the coordinate system by an angle θ_x .

[TBD: Add mathematical basis]

$$R_x = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\theta_x) & -\sin(\theta_x) & 0 \\ 0 & \sin(\theta_x) & \cos(\theta_x) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

A point P is rotated about the x-axis to position P' by converting the point P to its homogeneous representation and multiplying by the rotation matrix.



$$\begin{bmatrix} P'_x \\ P'_y \\ P'_z \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\theta_X) & -\sin(\theta_X) & 0 \\ 0 & \sin(\theta_X) & \cos(\theta_X) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} P_x \\ P_y \\ P_z \\ 1 \end{bmatrix}$$

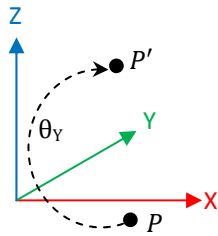
2.2.1.3 Rotation About Y-Axis

Description: The y-axis rotation matrix rotates a point in 3D Cartesian space around the y-axis of the coordinate system by an angle θ_Y .

[TBD: Add mathematical basis]

$$R_Y = \begin{bmatrix} \cos(\theta_Y) & 0 & \sin(\theta_Y) & 0 \\ 0 & 1 & 0 & 0 \\ -\sin(\theta_Y) & 0 & \cos(\theta_Y) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

A point P is rotated about the y-axis to position P' by converting the point P to its homogeneous representation and multiplying by the rotation matrix.



$$\begin{bmatrix} P'_x \\ P'_y \\ P'_z \\ 1 \end{bmatrix} = \begin{bmatrix} \cos(\theta_Y) & 0 & \sin(\theta_Y) & 0 \\ 0 & 1 & 0 & 0 \\ -\sin(\theta_Y) & 0 & \cos(\theta_Y) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} P_x \\ P_y \\ P_z \\ 1 \end{bmatrix}$$

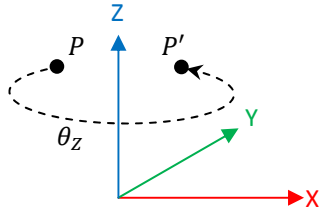
2.2.1.4 Rotation About Z-Axis

Description: The z-axis rotation matrix rotates a point in 3D Cartesian space around the z-axis of the coordinate system by an angle θ_Z .

[TBD: Add mathematical basis]

$$R_Z = \begin{bmatrix} \cos(\theta_Z) & -\sin(\theta_Z) & 0 & 0 \\ \sin(\theta_Z) & \cos(\theta_Z) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

A point P is rotated about the z-axis to position P' by converting the point P to its homogeneous representation and multiplying by the rotation matrix.

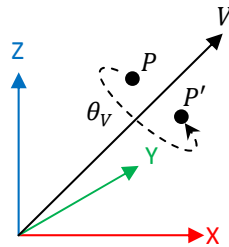


$$\begin{bmatrix} P'_x \\ P'_y \\ P'_z \\ 1 \end{bmatrix} = \begin{bmatrix} \cos(\theta_Z) & -\sin(\theta_Z) & 0 & 0 \\ \sin(\theta_Z) & \cos(\theta_Z) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} P_x \\ P_y \\ P_z \\ 1 \end{bmatrix}$$

2.2.1.5 Rotation About Vector

Description: This rotation matrix rotates a point in 3D Cartesian space around a vector (V) originating at the origin by an angle θ_V .

[TBD: Add mathematical basis]

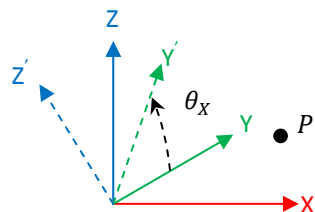


A point P is rotated about the vector V to position P' by converting the point P to its homogeneous representation and multiplying by the rotation matrix.

$$R_V = \begin{bmatrix} 1 + (x^2 - 1) \cdot C & -zS + xyC & yS + xzC & 0 \\ zS + xyC & 1 + (y^2 - 1) \cdot C & -xS + yzC & 0 \\ -yS + xzC & xS + yzC & 1 + (z^2 - 1) \cdot C & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \begin{array}{l} V \equiv (x, y, z) \\ S = \sin(\theta_V) \\ C = (1 - \cos(\theta_V)) \end{array}$$

2.2.1.6 Coordinate System Rotation about X-Axis

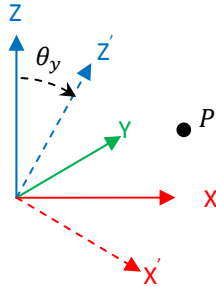
Description: The x-axis coordinate system rotation is similar to the standard x-axis rotation, except this rotation matrix rotates the coordinate system instead of the point about the current x-axis by an angle θ_X .



$$\begin{bmatrix} P'_x \\ P'_y \\ P'_z \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\theta_X) & \sin(\theta_X) & 0 \\ 0 & -\sin(\theta_X) & \cos(\theta_X) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} P_x \\ P_y \\ P_z \\ 1 \end{bmatrix}$$

2.2.1.7 Coordinate System Rotation about Y-Axis

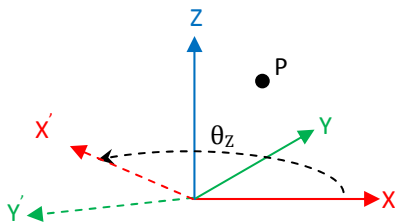
Description: The y-axis coordinate system rotation is similar to the standard y-axis rotation, except this rotation matrix rotates the coordinate system instead of the point about the current y-axis by an angle θ_Y .



$$\begin{bmatrix} P'_x \\ P'_y \\ P'_z \\ 1 \end{bmatrix} = \begin{bmatrix} \cos(\theta_Y) & 0 & -\sin(\theta_Y) & 0 \\ 0 & 1 & 0 & 0 \\ \sin(\theta_Y) & 0 & \cos(\theta_Y) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} P_x \\ P_y \\ P_z \\ 1 \end{bmatrix}$$

2.2.1.8 Coordinate System Rotation about Z-Axis

Description: The z-axis coordinate system rotation is similar to the standard z-axis rotation, except this rotation matrix rotates the coordinate system instead of the point about the current z-axis by an angle θ_z .



$$\begin{bmatrix} P'_x \\ P'_y \\ P'_z \\ 1 \end{bmatrix} = \begin{bmatrix} \cos(\theta_Z) & \sin(\theta_Z) & 0 & 0 \\ -\sin(\theta_Z) & \cos(\theta_Z) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} P_x \\ P_y \\ P_z \\ 1 \end{bmatrix}$$

2.2.1.9 Rotation and Translation Combinations [TBD]

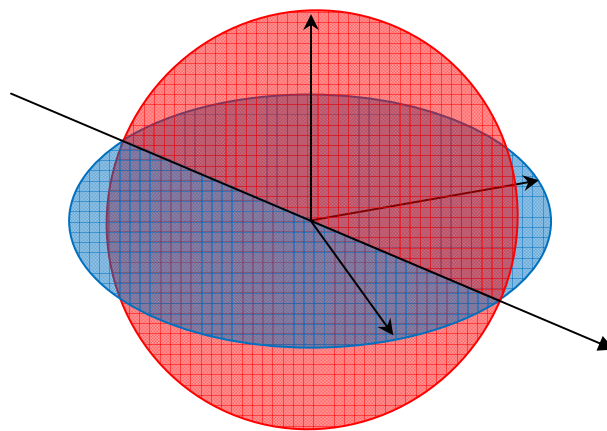
[TBD]

2.2.1.10 Coordinate System to Coordinate System Transform [TBD]

[TBD]

2.2.1.11 Euler Angles [TBD]

[TBD]



2.2.2 Scaling Matrices [TBD]

[TBD]

2.2.3 Orthographic Projection Matrix [TBD]

[TBD]

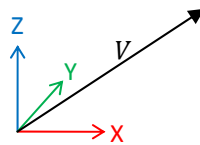
2.2.4 Perspective Projection Matrix [TBD]

[TBD]

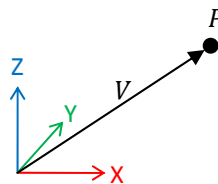
2.3 Vectors and Unit Vectors [TBD]

2.3.1 Vectors

A spatial vector is relative quantity defined by a magnitude and a direction. By themselves, vectors do not have an absolute position in space, but it is convenient to think of a vector emanating from the coordinate system origin.



Vectors are used to determine offsets from one point in space to another. As such, a vector can be thought of as the offsets of a point from the coordinate system origin.



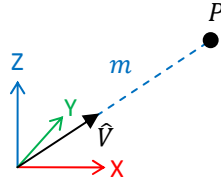
When viewed in this fashion, vectors are mathematically equivalent to points. The coefficients of a vector define the location of a point and vice-versa. The magnitude, or overall length, of a vector is calculated by the RSS of the directional components of the vector.

$$|V| = \sqrt{V_x^2 + V_y^2 + V_z^2}$$

The vector directional components can be any real number: positive, negative, or zero. A vector with zeros for each directional component also has a zero length and is referred to as a null vector.

2.3.2 Unit Vectors

A unit vector is a special case of vector that only indicates direction, not magnitude. Unit vectors always have a magnitude of 1. As such, a null vector cannot be represented by a unit vector. Unit vectors also have a relationship with the representation of a point in space. Instead of a direct correlation, the point, P is described by, the unit vector, \hat{V} , and a magnitude, m .



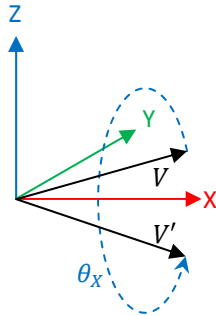
A unit vector is created by dividing a vector by its magnitude.

$$\hat{v} = \frac{V}{|V|}$$

2.3.3 Vector Rotation

Since vectors only have a direction and magnitude, not an absolute position, they cannot be translated, only rotated. Using the transformation matrices described in Section 2.2.1, vectors and unit vectors can be rotated about the origin along any vector direction.

Rotation About the X-Axis

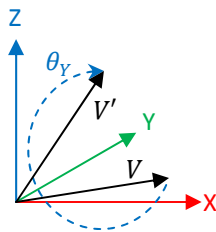


$$\begin{bmatrix} V'_x \\ V'_y \\ V'_z \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\theta_x) & -\sin(\theta_x) & 0 \\ 0 & \sin(\theta_x) & \cos(\theta_x) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} V_x \\ V_y \\ V_z \\ 1 \end{bmatrix}$$

Simplified Computational Form:

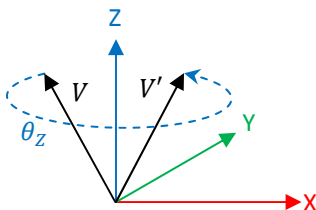
$$\begin{aligned} V'_x &= V_x \\ V'_y &= V_y \cdot \cos(\theta_x) - V_z \cdot \sin(\theta_x) \\ V'_z &= V_y \cdot \sin(\theta_x) + V_z \cdot \cos(\theta_x) \end{aligned}$$

Rotation About the Y-Axis



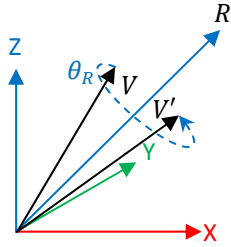
$$\begin{bmatrix} V'_x \\ V'_y \\ V'_z \\ 1 \end{bmatrix} = \begin{bmatrix} \cos(\theta_y) & 0 & \sin(\theta_y) & 0 \\ 0 & 1 & 0 & 0 \\ -\sin(\theta_y) & 0 & \cos(\theta_y) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} V_x \\ V_y \\ V_z \\ 1 \end{bmatrix}$$

Rotation About the Z-Axis



$$\begin{bmatrix} V'_x \\ V'_y \\ V'_z \\ 1 \end{bmatrix} = \begin{bmatrix} \cos(\theta_z) & -\sin(\theta_z) & 0 & 0 \\ \sin(\theta_z) & \cos(\theta_z) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} V_x \\ V_y \\ V_z \\ 1 \end{bmatrix}$$

Rotation About a Reference Vector (R)



$$\begin{bmatrix} V'_x \\ V'_y \\ V'_z \\ 1 \end{bmatrix} = \begin{bmatrix} 1 + (x^2 - 1) \cdot C & -zS + xyC & yS + xzC & 0 \\ zS + xyC & 1 + (y^2 - 1) \cdot C & -xS + yzC & 0 \\ -yS + xzC & xS + yzC & 1 + (z^2 - 1) \cdot C & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} V_x \\ V_y \\ V_z \\ 1 \end{bmatrix}$$

$$\begin{aligned} R &\equiv (x, y, z) \\ S &= \sin(\theta_R) \\ C &= (1 - \cos(\theta_R)) \end{aligned}$$

Simplified Computational Form:

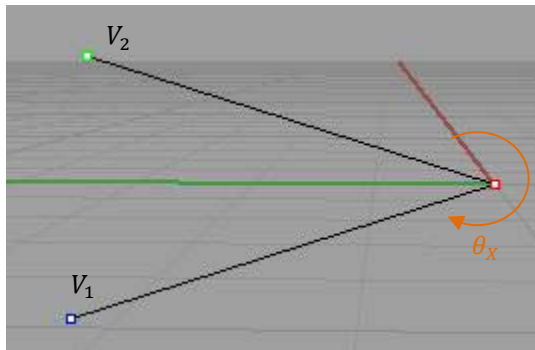
$$\begin{aligned} V'_x &= (1 + (x^2 - 1) \cdot C) \cdot V_x + (-zS + xyC) \cdot V_y + (yS + xzC) \cdot V_z \\ V'_y &= (zS + xyC) \cdot V_x + (1 + (y^2 - 1) \cdot C) \cdot V_y + (-xS + yzC) \cdot V_z \\ V'_z &= (-yS + xzC) \cdot V_x + (xS + yzC) \cdot V_y + (1 + (z^2 - 1) \cdot C) \cdot V_z \end{aligned}$$

As implied by rotation, the magnitude of the vector (or unit vector) does not change with the rotation, therefore these operations can be used on both vectors and unit vectors. Each transform operation accumulates numerical round off error and may cause a unit vector to need to be re-normalized.

2.3.3.1 Validation Examples

The following examples were created in Rhino3d to test the rotations of vectors. Each test vector is defined by a point at the origin and a second point in space. The vector value is the value of this second point.

Example 1 (Rotation About X-Axis)



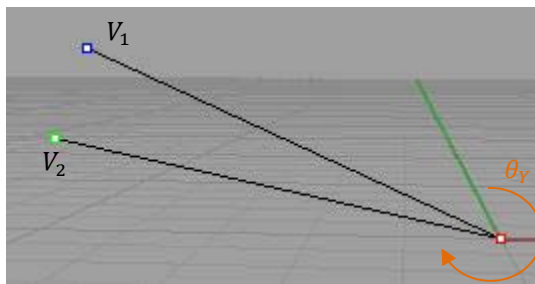
Inputs:

$$\begin{aligned} V_1 &= (-2.2895648, 2.9299575, -0.6279180) \\ \theta_x &= 33^\circ \end{aligned}$$

Output:

$$V_2 = (-2.2895648, 2.7992578, 1.0691528)$$

Example 2 (Rotation About Y-Axis)



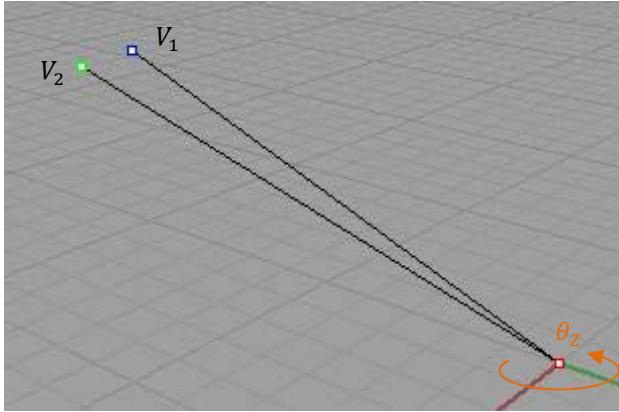
Inputs:

$$\begin{aligned} V_1 &= (-4.0489100, -1.3917385, 1.9754380) \\ \theta_y &= -12^\circ \end{aligned}$$

Output:

$$V_2 = (-4.3711482, -1.3917385, 1.0904542)$$

Example 3 (Rotation About Z-Axis)



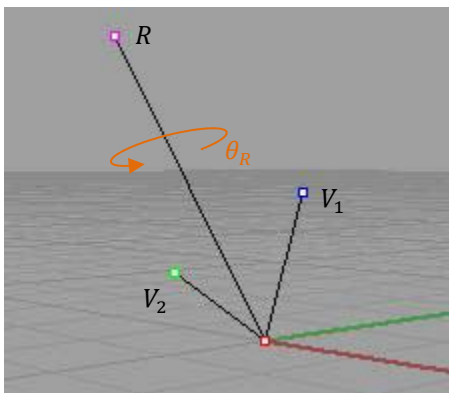
Inputs:

$V_1 = (1.0817985, -9.0228444, 4.1734866)$
 $\theta_z = 10^\circ$

Output:

$V_2 = (2.6321640, -8.6979148, 4.1734866)$

Example 3 (Rotation About Vector)



Inputs:

$V_1 = (-2.2895648, 2.7992578, 1.0691528)$
 $R = (-3.4870730, 1.1343368, 2.7573779)$
 $\theta_R = 45^\circ$

Output:

$V_2 = (-3.3061512, 1.8036369, 0.1931244)$

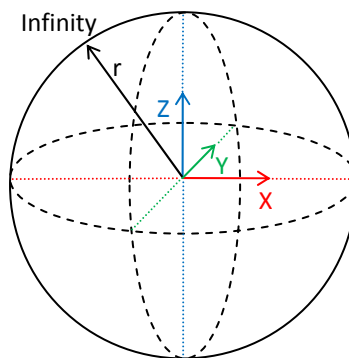
2.3.4 Vector Scaling [TBD]

[TBD]

2.4 Projective Geometry and Homogeneous Coordinates

[TBD]

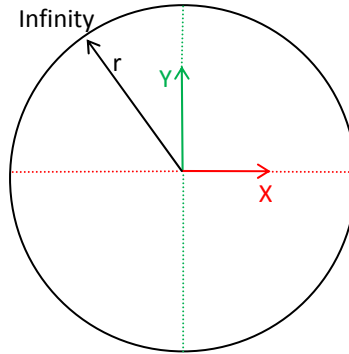
2.4.1 3D Homogeneous Coordinate Space [TBD]



[TBD]

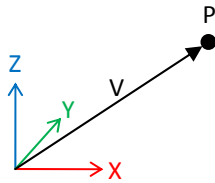
2.4.2 2D Homogeneous Coordinate Space [TBD]

[TBD]



2.5 Homogeneous Point Equations

A standard Cartesian point in 3-space can be represented by a vector (V) from the origin to the point in space.

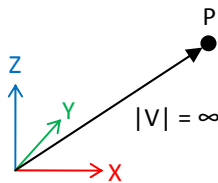


$$V = x\hat{i} + y\hat{j} + z\hat{k}$$

The \hat{i} , \hat{j} , and \hat{k} components represent the unit directional components in the 3 principal axes of the coordinate system and x , y , and z represent the distance along each of these axes to the point. A factor w is introduced into this equation with a default value of 1.

$$V = \frac{x\hat{i} + y\hat{j} + z\hat{k}}{w} \quad \text{if } w = 1$$

For standard Cartesian representation, this does not affect the value of the point. The w component becomes useful when trying to represent a point infinitely far away. If $w = 0$ the point is located on the infinite plane, but unlike standard Cartesian mathematics, the vector direction x , y , z is still meaningful and now represents a direction vector to the point with an infinite magnitude.



$$V = \frac{x\hat{i} + y\hat{j} + z\hat{k}}{0}$$

To eliminate confusion, the w is separated from the x , y , and z variables when writing the equation of a point.

$$(w; x, y, z)$$

There are other benefits to the homogeneous points. Since x , y , and z are divided by w all the values of the point can be multiplied by a scalar without changing the location of the point.

$$(w ; x, y, z) = \alpha \cdot (w ; x, y, z) = (\alpha w ; \alpha x, \alpha y, \alpha z)$$

This allows the point values to be changed to more convenient forms. One of these is the unit vector and inverse magnitude representation. Each component is divided by the magnitude of the vector and the resulting point value looks like this.

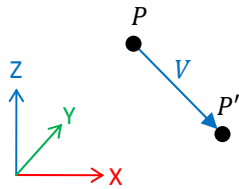
$$(w ; x, y, z) = \left(\frac{1}{|V|} ; \hat{x}, \hat{y}, \hat{z} \right)$$

The homogeneous component w can also be used to scale the point in 3D space. If the value of w is changed and the values of x , y , and z are interpreted without the w context, the values can be viewed as being scaled. This is used frequently in image graphics.

$$(2 ; x, y, z) \rightarrow \begin{matrix} 1/2 x \\ 1/2 y \\ 1/2 z \end{matrix}$$

2.5.1 Translate Point by a Vector [TBD]

A point P can be translated in 3D coordinate space by a vector V to a point P' .



$$P' = P + V \cdot w$$

or

$$P'_x = P_x + V_x \cdot w$$

$$P'_y = P_y + V_y \cdot w$$

$$P'_z = P_z + V_z \cdot w$$

The vector V is multiplied by w to keep its application 1:1 with the current coordinate system. If $w = 0$ the point will be at infinity so the movement of the point by a finite vector should have no effect. Likewise, if $w = 2$ all the values of x , y , and z are scaled by a factor of 2 so the vector needs to be scaled to have an appropriate effect.

Alternatively, the homogeneous translation matrix can be applied directly without the need for applying the scale to the translation vector.

$$\begin{bmatrix} P'_x \\ P'_y \\ P'_z \\ P'_w \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & dx \\ 0 & 1 & 0 & dy \\ 0 & 0 & 1 & dz \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} P_x \\ P_y \\ P_z \\ P_w \end{bmatrix}$$

2.5.1.1 Validation Examples

The following examples were created in Rhino3d to test the rotations of vectors. Each test vector is defined by a point at the origin and a second point in space. The vector value is the value of this second point.

Example 1

V_2

Inputs:

$P_1 = (\quad)$

$V_1 = (\quad)$

Output:

$P_2 = (\quad)$

V_1

Example 2

V_1

Inputs:

$P_1 = (\quad)$

$V_1 = (\quad)$

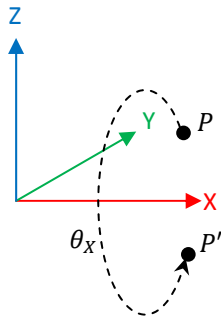
Output:

$P_2 = (\quad)$

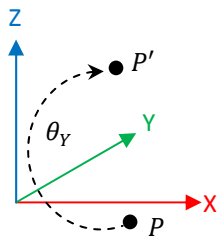
V_2

2.5.2 Rotate Point about the Origin [TBD]

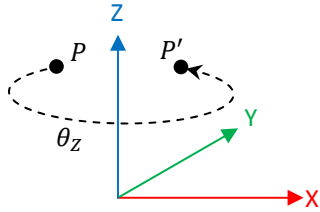
The homogeneous coordinate transformations defined in Section 2.2.1 can be applied directly to homogeneous points as long as the w-component is added to the calculation. To rotate a point around the x, y, and z-axes by θ_x , θ_y , and θ_z the homogeneous transforms are applied as:



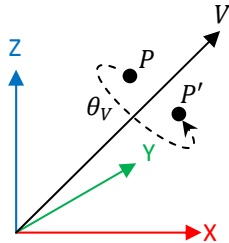
$$\begin{bmatrix} P'_x \\ P'_y \\ P'_z \\ P'_w \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\theta_x) & -\sin(\theta_x) & 0 \\ 0 & \sin(\theta_x) & \cos(\theta_x) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} P_x \\ P_y \\ P_z \\ P_w \end{bmatrix}$$



$$\begin{bmatrix} P'_x \\ P'_y \\ P'_z \\ P'_w \end{bmatrix} = \begin{bmatrix} \cos(\theta_y) & 0 & \sin(\theta_y) & 0 \\ 0 & 1 & 0 & 0 \\ -\sin(\theta_y) & 0 & \cos(\theta_y) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} P_x \\ P_y \\ P_z \\ P_w \end{bmatrix}$$



$$\begin{bmatrix} P'_x \\ P'_y \\ P'_z \\ P'_w \end{bmatrix} = \begin{bmatrix} \cos(\theta_z) & -\sin(\theta_z) & 0 & 0 \\ \sin(\theta_z) & \cos(\theta_z) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} P_x \\ P_y \\ P_z \\ P_w \end{bmatrix}$$



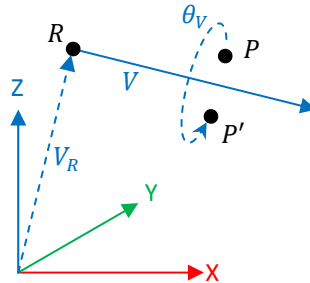
$$\begin{bmatrix} P'_x \\ P'_y \\ P'_z \\ P'_w \end{bmatrix} = \begin{bmatrix} 1 + (x^2 - 1) \cdot C & -zS + xyC & yS + xzC & 0 \\ zS + xyC & 1 + (y^2 - 1) \cdot C & -xS + yzC & 0 \\ -yS + xzC & xS + yzC & 1 + (z^2 - 1) \cdot C & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} P_x \\ P_y \\ P_z \\ P_w \end{bmatrix}$$

$$\begin{aligned} V &\equiv (x, y, z) \\ S &= \sin(\theta_v) \\ C &= (1 - \cos(\theta_v)) \end{aligned}$$

The rotations are relative to the angle, not the actual spatial location of the point so the value of w is unimportant for these calculations.

2.5.3 Rotate Point about a Point and a Vector [TBD]

A point P can be rotated to a position P' about a reference point R and a direction vector V by an angle θ_v using a set of primitive coordinate system operations.



Conceptually, the point must first be translated to the origin, then rotated about the vector V , and then translated back to the correct position in space. The sequence of operations is:

$$\begin{bmatrix} P'_x \\ P'_y \\ P'_z \\ P'_w \end{bmatrix} = [T_{V_R}] \cdot [R_V] \cdot [T_{-V_R}] \cdot \begin{bmatrix} P_x \\ P_y \\ P_z \\ P_w \end{bmatrix}$$

Where T_{V_R} is the translation matrix from the origin to the reference point, R_V is the rotation matrix about vector V , and T_{-V_R} is the translation from the reference point to the coordinate system origin. The full equation expanded out looks like this:

$$\begin{bmatrix} P'_x \\ P'_y \\ P'_z \\ P'_w \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & R_x \\ 0 & 1 & 0 & R_y \\ 0 & 0 & 1 & R_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 + (x^2 - 1) \cdot C & -zS + xyC & yS + xzC & 0 \\ zS + xyC & 1 + (y^2 - 1) \cdot C & -xS + yzC & 0 \\ -yS + xzC & xS + yzC & 1 + (z^2 - 1) \cdot C & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & 0 & -R_x \\ 0 & 1 & 0 & -R_y \\ 0 & 0 & 1 & -R_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} P_x \\ P_y \\ P_z \\ P_w \end{bmatrix}$$

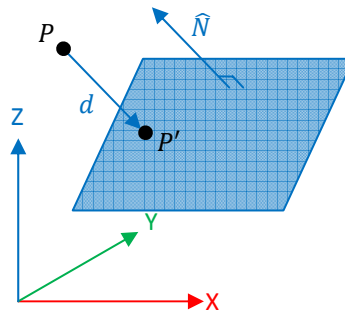
The full form can be combined into a computationally simpler form:

$$\begin{bmatrix} P'_x \\ P'_y \\ P'_z \\ P'_w \end{bmatrix} = \begin{bmatrix} 1 + (x^2 - 1) \cdot C & -zS + xyC & yS + xzC & R_x \cdot P_w \\ zS + xyC & 1 + (y^2 - 1) \cdot C & -xS + yzC & R_y \cdot P_w \\ -yS + xzC & xS + yzC & 1 + (z^2 - 1) \cdot C & R_z \cdot P_w \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} P_x - R_x \cdot P_w \\ P_y - R_y \cdot P_w \\ P_z - R_z \cdot P_w \\ P_w \end{bmatrix} \quad \begin{array}{l} V \equiv (x, y, z) \\ S = \sin(\theta_V) \\ C = (1 - \cos(\theta_V)) \end{array}$$

[TBD: Validate]

2.5.4 Project Point to a Plane [TBD]

A point in P 3D space can be projected to a point P' on a plane with a normal vector \hat{N} .



The standard equation for a plane is:

$$Ax + By + Cz + D = 0$$

Where the normal vector $\hat{N} = (A, B, C)$. If the point P is to be projected onto the plane in the direction normal to the planar surface, then P' can be described in terms of the plane's normal vector.

$$P' = P + \hat{N} \cdot d \rightarrow \begin{bmatrix} P'_x \\ P'_y \\ P'_z \end{bmatrix} = \begin{bmatrix} P_x \\ P_y \\ P_z \end{bmatrix} + \begin{bmatrix} A \\ B \\ C \end{bmatrix} \cdot d$$

Where d is the distance in the normal direction from point P to P' . The distance d can be calculated by first solving the planar equation at point P' .

$$A \cdot P'_x + B \cdot P'_y + C \cdot P'_z + D = 0$$

Expanding the planar equation to contain known quantities:

$$A \cdot (P_x + A \cdot d) + B \cdot (P_y + B \cdot d) + C \cdot (P_z + C \cdot d) + D = 0$$

Rearranging the equation to solve for d :

$$\begin{aligned} A^2 d + B^2 d + C^2 d &= -A \cdot P_x - B \cdot P_y - C \cdot P_z - D \\ d &= \frac{-A \cdot P_x - B \cdot P_y - C \cdot P_z - D}{A^2 + B^2 + C^2} \end{aligned}$$

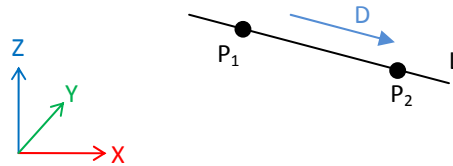
The location of the projected point P' can now be solved for using the projection equation:

$$P' = \begin{bmatrix} P_x \\ P_y \\ P_z \end{bmatrix} + \begin{bmatrix} A \\ B \\ C \end{bmatrix} \cdot d$$

[TBD: Validate, additionally validate that non-standard planar equation form doesn't break the equation (may need to unitize A,B,C to represent normal vector)]

2.6 Homogeneous (Plücker) Line Equations

A Plücker line is the homogenous representation of a Cartesian line. The simplest method of defining a line is with two points.



A Plücker line is completely defined by two vector quantities, the first of which is the unit direction vector of the line. The unit vector (D) can be solved by subtracting the two points and unitizing the results.

$$D = \frac{P_2 - P_1}{|P_2 - P_1|}$$

Where the magnitude of D is:

$$|P_2 - P_1| = \sqrt{(P_{2x} - P_{1x}) \cdot (P_{2y} - P_{1y}) \cdot (P_{2z} - P_{1z})}$$

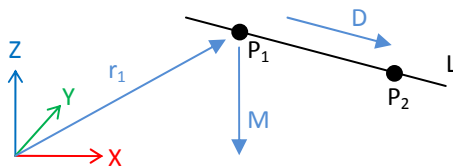
And the unit directional components are:

$$D_x = \frac{P_{2x} - P_{1x}}{|D|}$$

$$D_y = \frac{P_{2y} - P_{1y}}{|D|}$$

$$D_z = \frac{P_{2z} - P_{1z}}{|D|}$$

The second component of a Plücker line is known as the line's moment vector about the origin. This vector is calculated by taking the cross product of the direction vector and a vector from the origin to any point on the line.



$$M = r_1 \times D$$

The component break down for M is:

$$\begin{aligned}M_x &= y_1 D_z - z_1 D_y \\M_y &= z_1 D_x - x_1 D_z \\M_z &= x_1 D_y - y_1 D_x\end{aligned}$$

The two vectors D and M are all that is required to fully specify a Plücker line. The standard representation of a Plücker line is:

$$\{D; M\} \quad \text{or} \quad \{D_x, D_y, D_z; M_x, M_y, M_z\}$$

Like a homogeneous point, the values of a Plücker line can be multiplied by a scalar without changing the specification of the line.

$$\{D; M\} = \alpha \cdot \{D_x, D_y, D_z; M_x, M_y, M_z\} = \{\alpha D_x, \alpha D_y, \alpha D_z; \alpha M_x, \alpha M_y, \alpha M_z\}$$

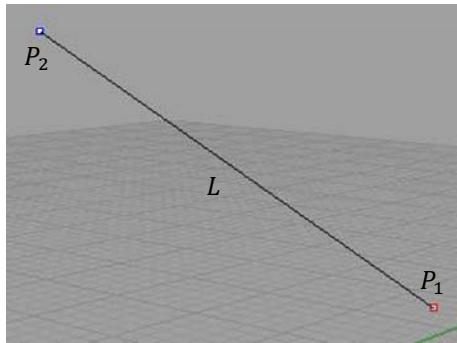
Lines that are on the infinite plane (or infinitely far from the origin) still have directions, but the moment vector becomes meaningless. The vector r_1 becomes infinite making the values for M indeterminate or more conveniently for this case zero. So for a line on the infinite plane the Plücker equation is:

$$\{D_x, D_y, D_z; 0, 0, 0\}$$

2.6.1.1 Validation Examples

These examples were created in Rhino3d to test building a Plücker line from two points. The line is created using P_1 as the starting point and P_2 as the end point.

Example 1



Inputs:

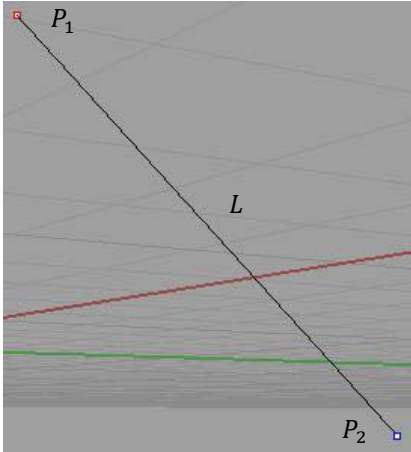
$$P_1 = (6.3711392, 3.2472407, 4.2585484)$$

$$P_2 = (-3.8617264, -3.6792328, 12.7622197)$$

Output:

$$L = \left\{ \begin{array}{l} -0.6821910413457215 \\ -0.46176490091089872 \\ 0.56691142111831561 \\ 3.8073460199005 \\ -6.5170151456317642 \\ -0.72672994674433289 \end{array} \right\}$$

Example 2



Inputs:

$$P_1 = (1.0000000, 2.0000000, 4.2585484)$$

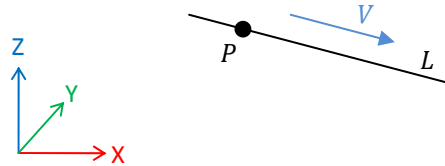
$$P_2 = (7.0000000, 3.0000000, -1.0000000)$$

Output:

$$L = \begin{pmatrix} 0.74620671968744334 \\ 0.12436778661457389 \\ -0.65399402531360884 \\ -1.8376142893262526 \\ 3.8317514575078189 \\ -1.3680456527603129 \end{pmatrix}$$

2.6.2 Line from a Point and a Vector

A Plücker line, L , can be created from a point in space, P , and a direction vector, V .



From the definition of a Plücker line the first three components, D_x , D_y , and D_z , are the direction vector of the line. Therefore the direction of the line is simply the unit vector of V .

$$D = \frac{V}{|V|}$$

The moment vector of the line can then be calculated.

$$M_x = P_y D_z - P_z D_y$$

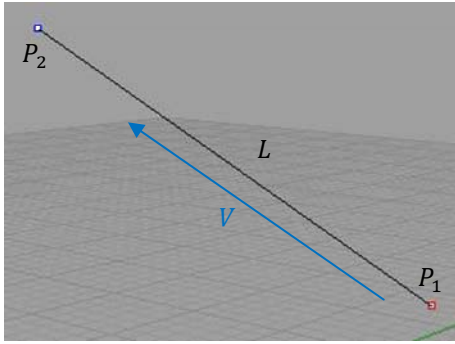
$$M_y = P_z D_x - P_x D_z$$

$$M_z = P_x D_y - P_y D_x$$

2.6.2.1 Validation Examples

These examples were created in Rhino3d to test building a Plücker line from a point and a vector. The line is created using P_1 as the point and $V = P_2 - P_1$ as the vector.

Example 1



Inputs:

$$P_1 = (6.3711392, 3.2472407, 4.2585484)$$

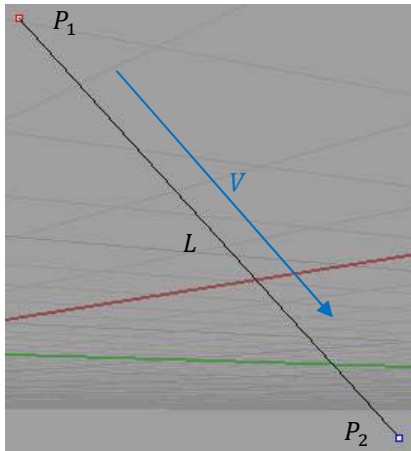
$$P_2 = (-3.8617264, -3.6792328, 12.7622197)$$

$$V = (-10.2328656, -6.9264735, 8.5036713)$$

Output:

$$L = \begin{Bmatrix} -0.6821910413457215 \\ -0.46176490091089872 \\ 0.56691142111831561 \\ 3.8073460199005 \\ -6.5170151456317642 \\ -0.72672994674433289 \end{Bmatrix}$$

Example 2



Inputs:

$$P_1 = (1.0000000, 2.0000000, 4.2585484)$$

$$P_2 = (7.0000000, 3.0000000, -1.0000000)$$

$$V = (6, 1, -5.2585484)$$

Output:

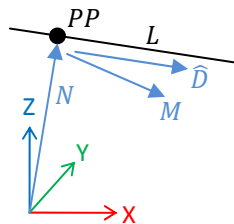
$$L = \begin{Bmatrix} 0.74620671968744334 \\ 0.12436778661457389 \\ -0.65399402531360884 \\ -1.8376142893262526 \\ 3.8317514575078189 \\ -1.3680456527603129 \end{Bmatrix}$$

2.6.3 Line from Two Points

Refer to: *Homogeneous (Plücker) Line Equations (Section 2.6)*

2.6.4 Principal Point of a Line [TBD]

The principal point of a line is the point on the line that is closest to the coordinate system origin. The principal point, PP , is defined by a vector from the coordinate system origin normal to the line, N .



The normal vector is calculated by taking the cross product of the line's direction unit vector, \hat{D} , and the moment vector, M .

$$N = \hat{D} \times M$$

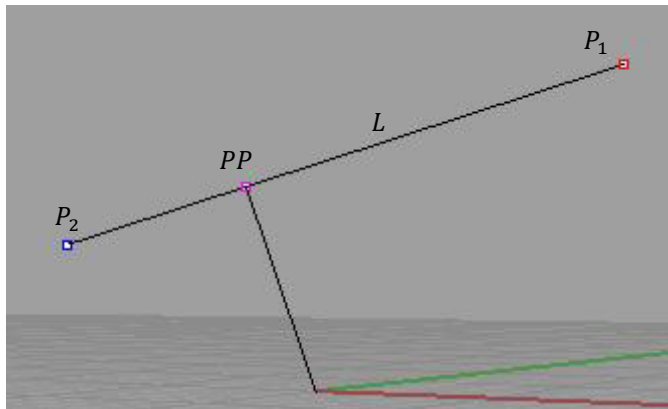
The principal point is simply the point at the location specified by the normal vector.

$$PP = (N_x, N_y, N_z)$$

2.6.4.1 Validation Examples

These examples were run in Rhino3d to test the calculation of the principal point algorithm for a Plücker line. The line is created using P_1 as the starting point and P_2 as the end point. The principal point is defined by PP . Using the input data, the accuracy should be to 7 significant digits.

Example 1



Inputs:

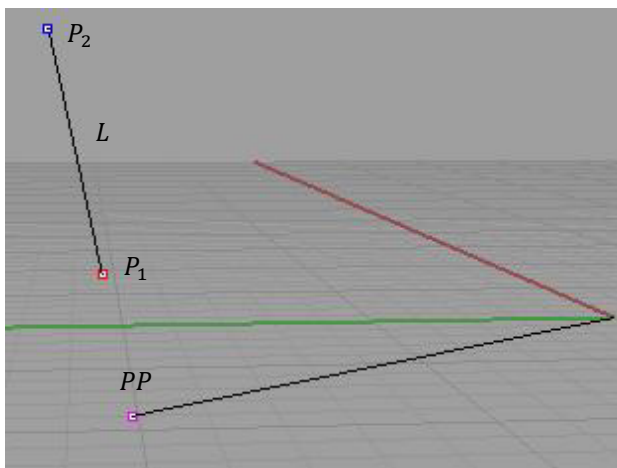
$$P_1 = (7.8556175, 1.6781878, 8.1735906)$$

$$P_2 = (-5.6592172, -3.1409219, 3.8003282)$$

Output:

$$PP = (-1.1542723, -1.5345520, 5.2580823)$$

Example 2



Inputs:

$$P_1 = (0.4122103, 10.3871865, 0.9676981)$$

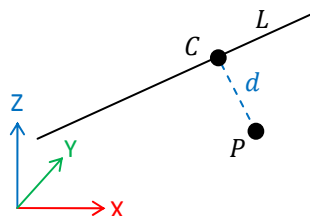
$$P_2 = (-0.1498946, 11.4004480, 5.8315792)$$

Output:

$$PP = (0.7494732, 9.7792295, -1.9506305)$$

2.6.5 Point of Closest Approach to Line [TBD]

Description: Algorithm finds the point of closest approach, C , to a point, P , in space on a line, L . C is the point on the line with the minimum distance, d , from P . Consequently, the line defined by \overline{PC} is also normal to the line L .



The point C can be found using a pair of constraints. First, the line segment \overline{PC} must be normal to the direction of L . Second to fit the definition of the moment vector for a Plücker line, a vector to any point on the line crossed with the direction vector must give the moment vector. The first constraint is set by taking the dot product of the offset vector and the line's direction vector, D .

$$PC \circ D = 0$$

The dot product is expanded out to yield:

$$(C_x - P_x) \cdot D_x + (C_y - P_y) \cdot D_y + (C_z - P_z) \cdot D_z = 0$$

Regrouping this equation in-terms of C gives:

$$D_x C_x + D_y C_y + D_z C_z = P_x D_x + P_y D_y + P_z D_z$$

The second constraint requires the vector from the origin to C to satisfy the moment vector calculation. This can be displayed using the moment vector calculation.

$$D \times C = M$$

The cross product expanded out gives:

$$\begin{aligned} M_x &= C_y D_z - D_y C_z \\ M_y &= C_x D_z - D_x C_z \\ M_z &= C_x D_y - D_x C_y \end{aligned}$$

Each of these equations is then refactored in terms of C .

$$\begin{aligned} D_z C_y - D_y C_z &= M_x \\ D_z C_x - D_x C_z &= M_y \\ D_y C_x - D_x C_y &= M_z \end{aligned}$$

These four equations can be combined and put into matrix form to solve for the location of C .

$$\begin{bmatrix} D_x & D_y & D_z \\ 0 & -D_z & D_y \\ -D_z & 0 & D_x \\ -D_y & D_x & 0 \end{bmatrix} \cdot \begin{bmatrix} C_x \\ C_y \\ C_z \end{bmatrix} = \begin{bmatrix} P_x D_x + P_y D_y + P_z D_z \\ M_x \\ M_y \\ M_z \end{bmatrix}$$

This matrix appears to be overdetermined because there are more equations than unknowns, but the cross product equations are linearly dependent so the overall rank of this system is 3. Since the shape of this matrix is not square, a standard matrix inverse cannot be applied to solve for the location of C . Instead, the matrix is converted to an augmented version.

$$\left[\begin{array}{ccc|c} D_x & D_y & D_z & P_x D_x + P_y D_y + P_z D_z \\ 0 & -D_z & D_y & M_x \\ -D_z & 0 & D_x & M_y \\ -D_y & D_x & 0 & M_z \end{array} \right]$$

This augmented matrix can now be solved by converting it to reduced row echelon form. *Note: The reduced row echelon form algorithm is detailed in the linear algebra section.* Reducing the augmented matrix to reduced row

echelon form will move the linearly dependent row to the bottom of the array and the top three lines will represent the solution for the x, y, and z components of C in the form shown below.

$$\left[\begin{array}{ccc|c} 1 & 0 & 0 & C_x \\ 0 & 1 & 0 & C_y \\ 0 & 0 & 1 & C_z \\ 0 & 0 & 0 & 0 \end{array} \right]$$

2.6.6 Validation Examples

[TBD]

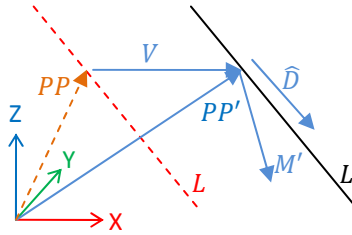
Example 1

[TBD]

Example 2

2.6.7 Translate Line by a Vector [TBD]

A Plücker line, L , can be translated to a new line, L' , by a vector, V , in 3-dimensional space.



By definition, a translation does not involve any rotation, so the direction unit vector, \hat{D} , of the line does not change. Therefore, only the moment vector must be updated. Using the direction and the moment vectors, the principal point can be calculated.

$$PP = \hat{D} \times M$$

The vector movement applies equally to all points on the line, so a point on the new line can be calculated by adding the offset vector to the principal point.

$$PP' = PP + V$$

This new point is not necessarily the principal point of the new line, but can still be used to calculate a new moment vector. The new moment vector is calculated using the new principal point and the direction vector.

$$M' = PP' \times \hat{D}$$

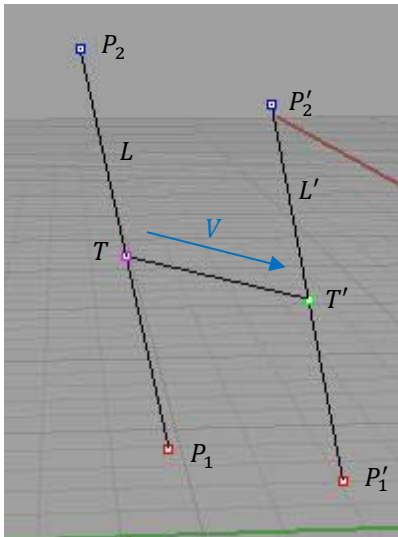
or

$$\begin{aligned} M'_x &= PP'_y \hat{D}_z - PP'_z \hat{D}_y \\ M'_y &= PP'_z \hat{D}_x - PP'_x \hat{D}_z \\ M'_z &= PP'_x \hat{D}_y - PP'_y \hat{D}_x \end{aligned}$$

2.6.7.1 Validation Examples

These examples were run in Rhino3d to test the translation of a Plücker line. The original line, L_1 , is created using P_1 as the starting point and P_2 as the end point. The vector, V , is defined by T and T' . The new line is created using P_1 and P_2 . A translation of V on L should yield L' . The coefficients of L' should be accurate to 7 significant digits given the accuracy of the input values.

Example 1



Inputs:

$$P_1 = (0.4122103, 10.3871865, 0.9676981)$$

$$P_2 = (-0.1498946, 11.4004480, 5.8315792)$$

$$P_1' = (1.0822860, 8.1078862, 0.2925770)$$

$$P_2' = (0.5201811, 9.1211477, 5.1564581)$$

$$T = (0.1311578, 10.8938173, 3.3996387)$$

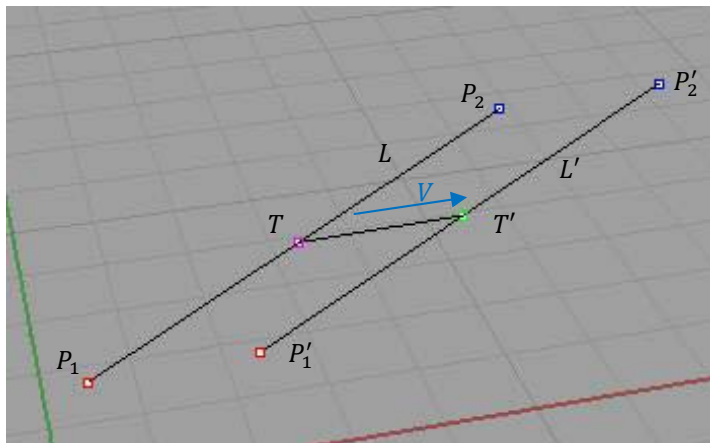
$$T' = (0.8012336, 8.6145169, 2.7245175)$$

$$V = (0.6700758, -2.2793004, -0.6751212)$$

Output:

$$L' = \begin{pmatrix} -0.11242097968314492 \\ 0.20265229942883076 \\ 0.97277621725825991 \\ 7.8278674657864578 \\ -1.0857138740443286 \\ 1.1308242563030826 \end{pmatrix}$$

Example 2



Inputs:

$$P_1 = (0.4122103, 0.3871865, 0.9676981)$$

$$P_2 = (4.9396196, 1.4004480, 2.8320656)$$

$$P_1' = (2.1030877, 0.2475363, 1.0976548)$$

$$P_2' = (6.6304970, 1.2607979, 2.9620222)$$

$$T = (2.6759149, 0.8938173, 1.8998818)$$

$$T' = (4.3667924, 0.7541671, 2.0298385)$$

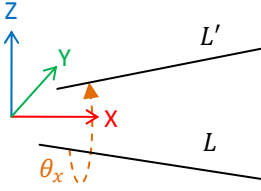
$$V = (1.6908775, -0.1396502, 0.1299567)$$

Output:

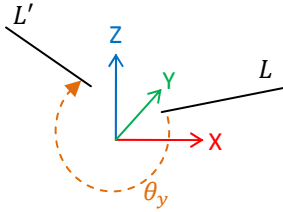
$$L' = \begin{pmatrix} 0.90548186286234023 \\ 0.20265232064060906 \\ 0.37287348117869917 \\ -0.13014257058320877 \\ 0.20972088116068566 \\ 0.20205597286566995 \end{pmatrix}$$

2.6.8 Rotate Line about the Origin

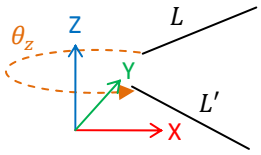
A Plücker line can be rotated about the origin using a series of coordinate system transforms. The direction and moment vectors must simply be rotated in the same fashion.



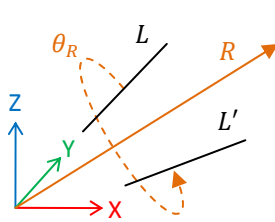
$$\begin{bmatrix} D'_x & M'_x \\ D'_y & M'_y \\ D'_z & M'_z \\ 1 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\theta_x) & -\sin(\theta_x) & 0 \\ 0 & \sin(\theta_x) & \cos(\theta_x) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} D_x & M_x \\ D_y & M_y \\ D_z & M_z \\ 1 & 1 \end{bmatrix}$$



$$\begin{bmatrix} D'_x & M'_x \\ D'_y & M'_y \\ D'_z & M'_z \\ 1 & 1 \end{bmatrix} = \begin{bmatrix} \cos(\theta_y) & 0 & \sin(\theta_y) & 0 \\ 0 & 1 & 0 & 0 \\ -\sin(\theta_y) & 0 & \cos(\theta_y) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} D_x & M_x \\ D_y & M_y \\ D_z & M_z \\ 1 & 1 \end{bmatrix}$$



$$\begin{bmatrix} D'_x & M'_x \\ D'_y & M'_y \\ D'_z & M'_z \\ 1 & 1 \end{bmatrix} = \begin{bmatrix} \cos(\theta_z) & -\sin(\theta_z) & 0 & 0 \\ \sin(\theta_z) & \cos(\theta_z) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} D_x & M_x \\ D_y & M_y \\ D_z & M_z \\ 1 & 1 \end{bmatrix}$$



$$\begin{bmatrix} D'_x & M'_x \\ D'_y & M'_y \\ D'_z & M'_z \\ 1 & 1 \end{bmatrix} = \begin{bmatrix} 1 + (x^2 - 1) \cdot C & -zS + xyC & yS + xzC & 0 \\ zS + xyC & 1 + (y^2 - 1) \cdot C & -xS + yzC & 0 \\ -yS + xzC & xS + yzC & 1 + (z^2 - 1) \cdot C & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} D_x & M_x \\ D_y & M_y \\ D_z & M_z \\ 1 & 1 \end{bmatrix}$$

$$\begin{aligned} V &\equiv (x, y, z) \\ S &= \sin(\theta_V) \\ C &= (1 - \cos(\theta_V)) \end{aligned}$$

2.6.8.1 Validation Examples

Example 1

[TBD]

Example 2

[TBD]

2.6.9 Rotate a Line about a Point and a Vector [TBD]

[TBD]

2.6.9.1 Validation Examples

Example 1

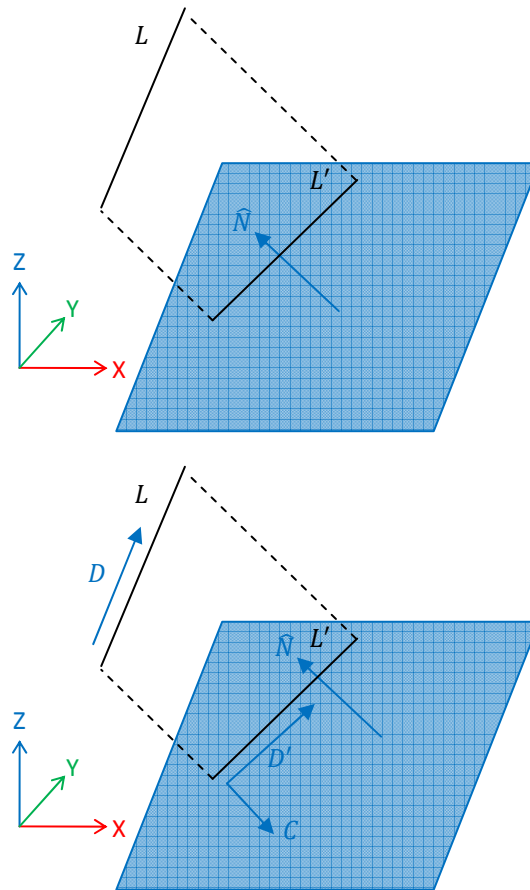
[TBD]

Example 2

[TBD]

2.6.10 Project Line to a Plane [TBD]

[TBD]



2.6.10.1 Validation Examples

Example 1

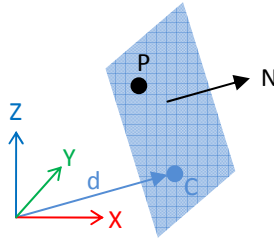
[TBD]

Example 2

[TBD]

2.7 Homogeneous Plane Equations

Homogeneous planes are represented by their normal vector and the distance of closest approach to the coordinate system origin.



Given a normal vector (N) and a point on the plane (P), the full planar equation can be solved with the standard Cartesian formula:

$$Ax + By + Cz + D = 0$$

The direction cosines of the normal vector correspond to the values of A, B, and C and x, y, and z correspond to the value of point P.

$$D = -N_x P_x - N_y P_y - N_z P_z$$

The full planar equation can now be represented in short hand form much like the equation of a point.

$$[D ; N] \quad \text{or} \quad [D ; A , B , C]$$

Like the equations of points and lines, the components of a plane can be multiplied by a scalar without affecting the plane.

$$[D ; A , B , C] = \alpha \cdot [D ; A , B , C] = [\alpha D ; \alpha A , \alpha B , \alpha C]$$

A convenient way to represent a homogeneous plane is to unitize the normal vector which causes D to reduce to the negative of the distance from the origin to the point of closest approach.

$$[D ; A , B , C] = [-d ; N_x , N_y , N_z]$$

2.7.1.1 Validation Examples

Example 1

[TBD]

Example 2

[TBD]

2.7.2 Plane from a Point and a Normal Vector

Description: A homogeneous plane can be constructed from the normal vector and any point on the plane.

Given a normal vector V and a point on the plane P the equation of the homogeneous plane can be solved using the planar equation.

$$Ax + By + Cz + D = 0$$

We know that the normal vector is described by the components A , B , and C in the homogeneous equation of a plane. The only value that remains to be solved for is D . Plugging in the normal vectors directions into the planar equation and solving for D gives the new equation:

$$D = -(V_x x + V_y y + V_z z)$$

The variables x , y , and z represent the location of any point on the plane, so plugging in the point P , the value of D can be solved for.

$$D = -(V_x P_x + V_y P_y + V_z P_z)$$

2.7.2.1 Validation Examples

Example 1

[TBD]

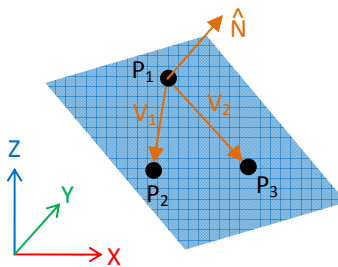
Example 2

[TBD]

2.7.3 Plane from Three Points

Description: Calculates a homogeneous plane from 3 spatial points.

Given the three points P_1 , P_2 , and P_3 , the normal vector of the plane can be solved for by taking the cross product of vectors between each of the points.



The selection of which two pair combinations to use is arbitrary. As such, the normal vector \hat{N} is somewhat arbitrary. Depending on the selection of the points and the order in which the cross product is taken, the normal vector can be either positive or negative. Since the normal vector isn't specified either of these may be correct.

This approach assumes the points are ordered in a right-handed sense such that points in a counterclockwise orientation will give a positive normal vector and points in a clockwise order will give a negative.

First the two intermediate vectors must be calculated.

$$V_1 = (P_{2,x} - P_{1,x}, P_{2,y} - P_{1,y}, P_{2,z} - P_{1,z})$$

$$V_2 = (P_{3,x} - P_{1,x}, P_{3,y} - P_{1,y}, P_{3,z} - P_{1,z})$$

The cross product of these two vectors is then taken and unitized to get the normal vector:

$$\hat{N} = \frac{V_1 \times V_2}{|V_1 \times V_2|}$$

Then D can be solved for using the same method as in *Section 2.7.2* using the normal vector and point P_1 :

$$D = -(\hat{N}_x P_{1,x} + \hat{N}_y P_{1,y} + \hat{N}_z P_{1,z})$$

2.7.3.1 Validation Examples

Example 1

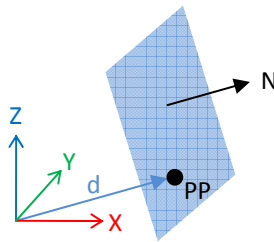
[TBD]

Example 2

[TBD]

2.7.4 Principal Point of a Plane [TBD]

[TBD]



2.7.4.1 Validation Examples

Example 1

[TBD]

Example 2

[TBD]

2.7.5 Translate Plane by a Vector [TBD]

[TBD]

2.7.5.1 Validation Examples

Example 1

[TBD]

Example 2

[TBD]

2.7.6 Rotate Plane about the Origin [TBD]

[TBD]

2.7.6.1 Validation Examples

Example 1

[TBD]

Example 2

[TBD]

2.7.7 Rotate Plane about a Point and a Vector [TBD]

[TBD]

2.7.7.1 Validation Examples

Example 1

[TBD]

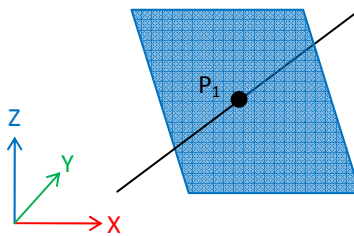
Example 2

[TBD]

2.8 Geometric Intersections

2.8.1 Intersection of a Line and a Plane

[TBD]



2.8.1.1 Validation Examples

Example 1

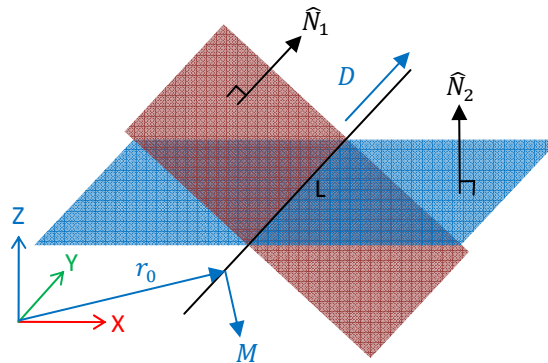
[TBD]

Example 2

[TBD]

2.8.2 Intersection of Two Planes [TBD]

[TBD]



2.8.2.1 Validation Examples

Example 1

[TBD]

Example 2

[TBD]

2.9 Geometric Projections

2.9.1 Projection of a Point onto a Plane [TBD]

[TBD]

2.9.1.1 Validation Examples

Example 1

[TBD]

Example 2

[TBD]

2.9.2 Projection of a Point onto a Plane (2D) [TBD]

[TBD]

2.9.2.1 Validation Examples

Example 1

[TBD]

Example 2

[TBD]

2.9.3 Projection of a Line onto a Plane [TBD]

[TBD]

2.9.3.1 Validation Examples

Example 1

[TBD]

Example 2

[TBD]

2.9.4 Projection of a Line onto a Plane (2D) [TBD]

[TBD]

2.9.4.1 Validation Examples

Example 1

[TBD]

Example 2

[TBD]

2.10 Polygon Routines

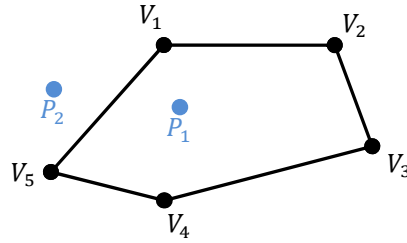
These routines are methods to build and extract information from 2D polygons. These methods are used regularly in both computer aided drafting (CAD) and photogrammetric applications.

2.10.1 Point Inside Polygon

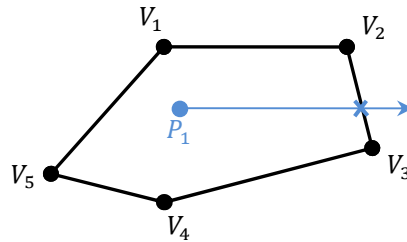
Description: Determines whether a point P is inside a 2D polygon comprised of vertices $V_1 - V_n$. There are several different approaches that can be used according to the type of polygon. These are listed below along with the required checks to determine which method to use.

2.10.1.1 Crossing Method

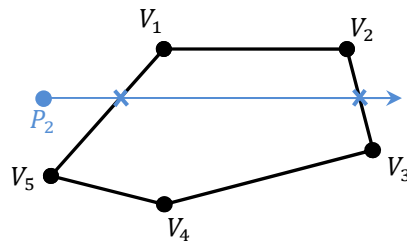
This is the least complicated method for determining if a point is inside a polygon. The crossing method counts the number of times that a line originating from the point crosses the edge of a polygon. The figure below shows two sample points, P_1 inside the polygon, and P_2 outside.



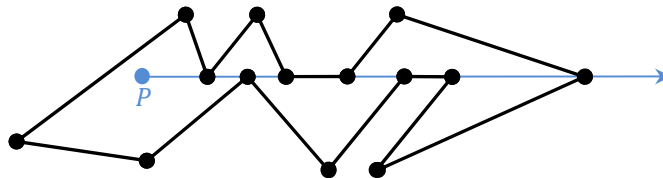
The crossing method determines point inclusion by counting the number of times that the line crosses an edge of the polygon. If the number of crossings is odd, the point is inside the polygon. If the number of crossings is even, the point is outside. The figure below shows the example for the interior point.



When drawing a line to the right from the inside point, it's apparent that the line crosses only one side of the polygon.



Some issues can arise when using this method, especially with concave polygons. The figure below shows some of the issues that can arise when using complicated convex polygons.



The vertices that cross the vector's path along with the horizontal edges that are coincident with the vector can cause problems in the tallying of crossings. To account for these conditions, a few crossing rules are set.

1. An upward edge includes its starting point and excludes its end point.
2. A downward edge excludes its starting point and includes its endpoint.
3. Horizontal edges are excluded
4. Intersection of the ray and an edge must be to the right of the point.

This method accounts for any special conditions that arise because of concave intersection points.

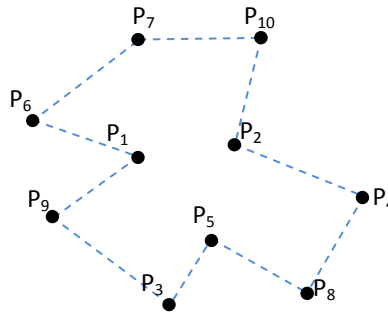
2.10.1.2 Bins Method [TBD]

[TBD]

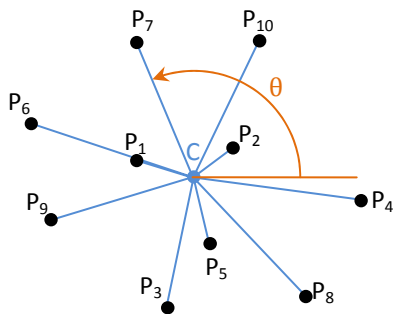
2.10.2 Create Simple Concave Hull Polygon

Description: A simple concave polygon is one where sides do not intersect, but is allowed to have concave regions. This method creates a simple concave polygon from a series of points. This method is used primarily as a first step for more complicated methods, like the convex hull polygon.

Given a set of random points, a simple convex hull polygon can be constructed.

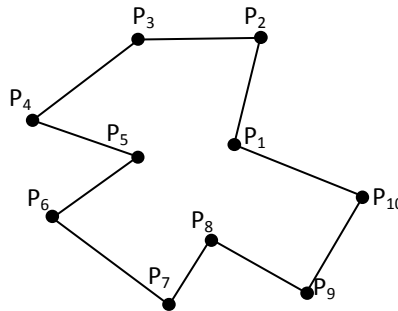


This is accomplished by taking the centroid C of the data set and determining the rotational order of all of the points around the centroid.

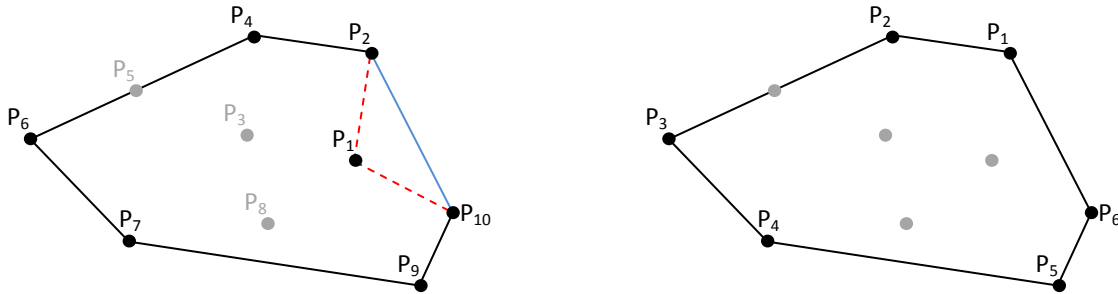


$$\theta = \text{atan2}(C_y - P_{y,i}, C_x - P_{x,i})$$

The points are sorted by their rotational order and a new polygon is built.



This ensures that the polygon will be simple and not have any self-intersecting edges, but doesn't specify any type of general shape. This result can then be used directly to create a convex hull polygon.



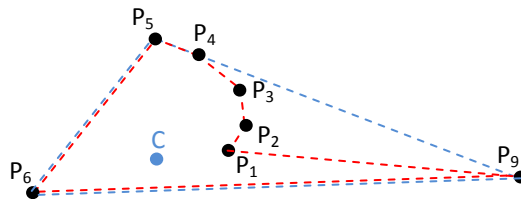
The first block show the removal of a concave vertex P_3 . This vertex is immediately removed from the stack for further calculations. As such, the next vertex check, P_4 , will compare the edges $\overline{P_2P_4}$ and $\overline{P_4P_5}$.

The second block shows the removal of a collinear vertex P_5 . This edge can be accurately described by $\overline{P_4P_6}$ so the extra vertex is not required. Consequently, this vertex could be left in and the polygon would still be considered convex, but it would no longer be considered minimal.

The third block shows the completion of the convex hull by removing the first point, P_1 , from the polygon and creating the closing edge from P_{10} to P_2 .

The final result of the convex hulling is shown in the 4th block. The resulting convex hull for the 10 input points is an irregular hexagon.

The final step of removing the 1st point to create the convex hull leads us to a potential problem. There are point geometries that can make the creation of a convex hull difficult. Take the case below.



The problem with this case is that closing this polygon requires the removal of the first four points. For the algorithm to account for this, the Graham scan must be done twice to ensure there are no special conditions that leave concave vertices.

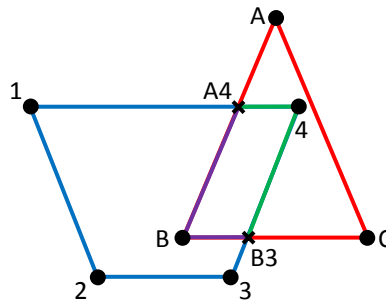
2.10.4 Boolean Polygon Operations

These methods deal with merging a pair of simple polygons (polygons without holes or self-intersection). The polygons are required to be right-handed so that a positive polygon has its vertices ordered in a counterclockwise sense. These methods will work for polygons that are ordered positive or negative, as long as they are consistent with each other.

2.10.4.1 Polygon Segmentation

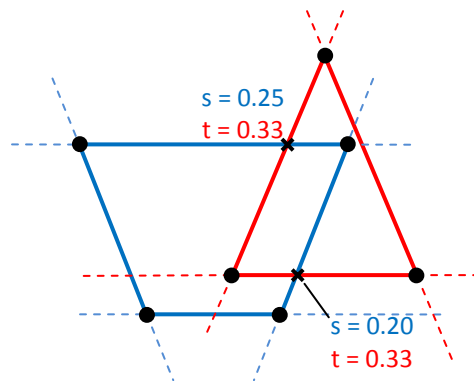
Description: Polygon segmentation is a process of breaking up polygons into multiple polyline segments based on the intersections of two polygons. Each segment starts and ends with a intersection and has a type of either inside or outside.

The figure below shows two intersecting polygons. The left polygon's vertices are marked 1-4. The right polygon's are marked A-C. The intersection points are labeled as a combination of the edges from each polygon. The naming scheme uses the previous vertex from each polygon to create the name of the intersection point.

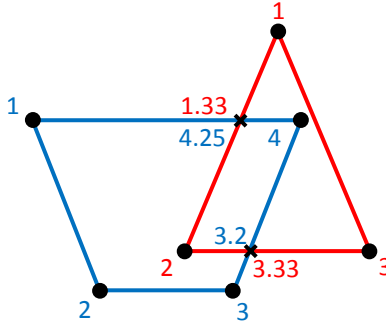


From this figure, each polygon can be seen as a combination of two segments. In this particular case each polygon has an internal and an external segment. An external segment is one that lies outside of the other polygon. The outside segment for the left hand polygon is made up of points A4, 1, 2, 3, and B3. The inside segment, which lies inside of the right hand polygon is made up of points B3, 4, A4. The right hand polygon can be broken down in this same fashion.

To split the polygons into segments the intersection points between the polygons must be found. The most direct way to calculate the intersections is using the *Parametric Line Intersection* method in (Section 2.1.5). The intersection of each pair of edges from the left polygon is found with each edge in the right polygon. Each edge pair is checked for an internal intersection point. Any internal intersection is a valid polygon intersection.



The figure above shows the intersections for the two test polygons. The only intersections that have a t-value between 0 and 1 are shown with black x's. These intersections are inserted into the polygon structure according to the parametric line's t-value. Each vertex is given an integer value corresponding to its order in the polygon. The intersection points are given a value that is equal to the previous actual vertex value plus the t-value for the intersection. This new value is called the t-score. The t-scores for this test set are shown in the next figure.



The intersections along with the original vertices are combined and sorted according to their t-scores to create new polygons. The new polygons are broken into segments at every intersection.

Polygon1	Polygon 2		Polygon1 (Seg 1)	Polygon 1 (Seg 2)	Polygon 2 (Seg 1)	Polygon 2 (Seg 2)
1	1	→	4.25	3.2	1.33	3.33
2	1.33		1	4	2	3
3	2		2	4.25	3.33	1
3.2	3.33		3			1.33
4	3		3.2			
4.25						

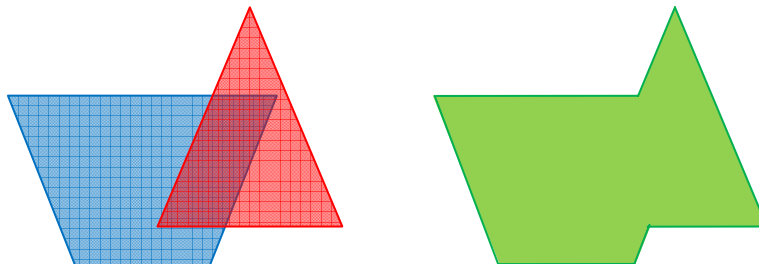
The polygon segments must then be categorized into either internal or external segments. This can be done by checking any of the points in the segment for inclusion in the other polygon using *Point Inside Polygon* method in (Section 2.10.1). For instance, in Polygon 1 – Segment 2, point 4 is found to be inside to polygon 2. This makes Polygon1 – Segment 2 an inside segment.

It is possible to have a segment that consists of only two back-to-back intersections. In this situation, the midpoint between the two intersections is used as a check point.

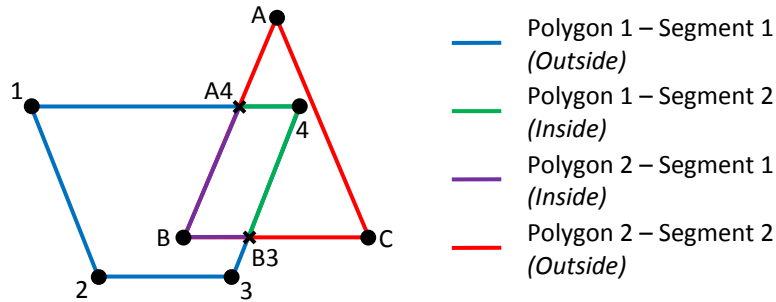
Once the polygons are broken into segments, they are easily manipulated into to form the Boolean combinations discussed in the following sections.

2.10.4.2 Polygon Union

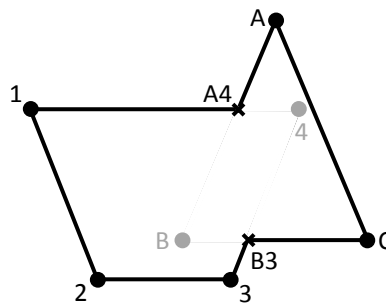
Description: This method takes the union of the two polygons (also known as a Boolean OR). As long as the polygons overlap, the resulting product will be a single polygon.



The union of two polygons can be found from the polygon segments calculated in section 2.10.4.1. Stringing together the outside segments of both polygons, gives the union of the two.



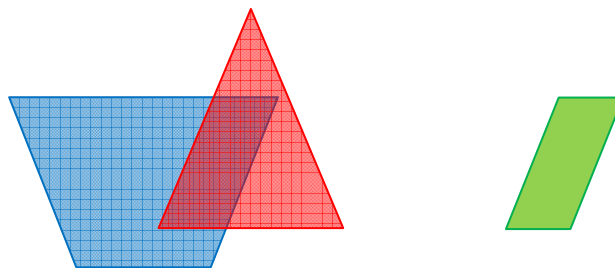
The segments are strung together at their ends by matching intersection points between the two polygons. For this example, looking at only the outside segments, Poly 1 – Seg 1 starts at point A4 and ends at B3. The next segment is looked for in the other polygon. Poly2 – Seg 2 starts at point B3 and ends at A4, so this segment continues the original path and closes the polygon by ending at the start point.



This process works with much more complicated polygons as well. The segments are strung together by interleaving segments from the two polygons matching ends until the start point is reached. This method will always return one polygon object as long as the original polygons overlap.

2.10.4.3 Polygon Intersection [TBD]

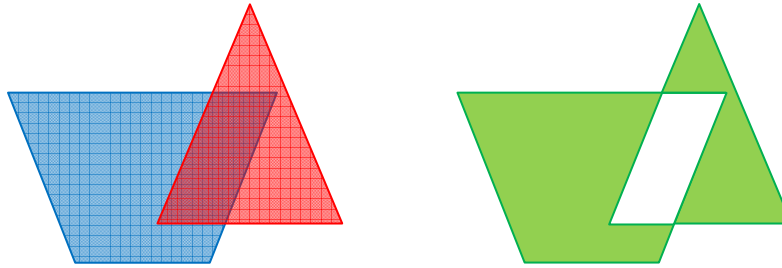
Description: This method takes the intersection of the two polygons (also known as a Boolean AND). If the polygons overlap, the result will be one or more smaller polygons that represent any overlap in the two originals.



[TBD]

2.10.4.4 Polygon Difference [TBD]

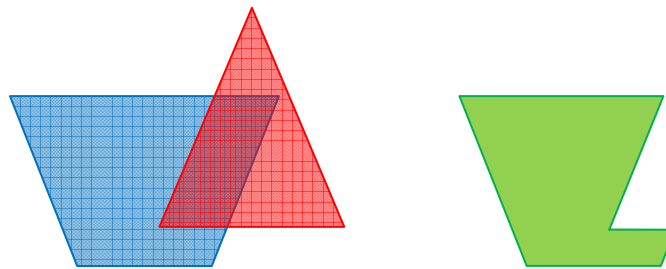
Description: This method returns the polygon difference between two polygons (also known as a Boolean XOR). The resulting polygon(s) will represent the regions of the original two polygons that did not overlap, the opposite of polygon intersection.



[TBD]

2.10.4.5 Polygon Subtraction [TBD]

Description: This method subtracts the area of one polygon from the other. The result is one or more polygons that represent the area of the first polygon that was not overlapped by the second.



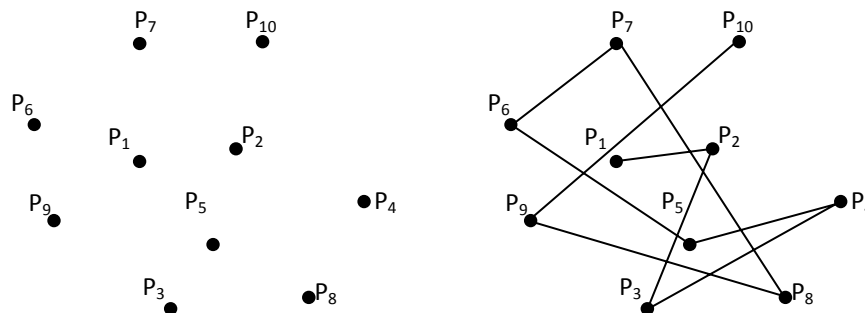
[TBD]

2.11 Point Sorting and Bounding Algorithms [TBD]

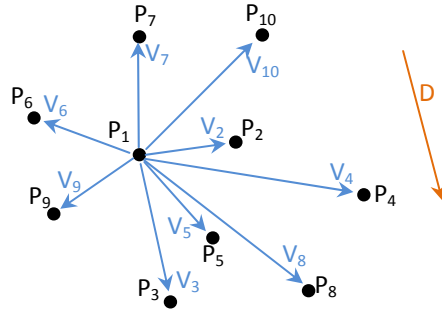
[TBD]

2.11.1 2D/3D Vector Point Sorting

Description: Sorts a series of points in 2D or 3D space according to the direction of a vector.



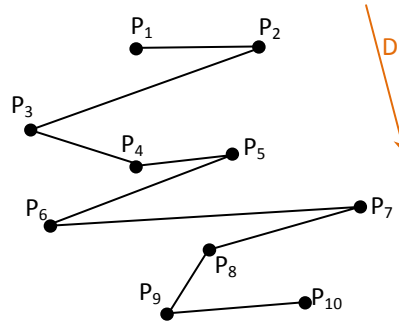
The points shown above are in random order. The mapping from point to point is shown in the right hand image. To sort this list of points into the direction of a vector V vectors between each of the points and a common location must be determined. The first point P_1 is used as the anchor for this system.



Each of the vectors V_2 through V_{10} represent the offset from P_1 to that point. Notice that there is no V_1 since the offset from P_1 to itself is 0. The direction vector D represents the direction the points are to be ordered in. The distance (d) of each point along D is calculated by taking the dot product of the direction vector with each of the offset vectors.

$$d_i = D \cdot V_i$$

The points are then sorted by their associated d_i giving a sorted point list in the direction of D .



2.11.2 Directional Bounding Box [TBD]

[TBD]

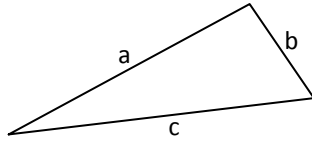
2.11.3 Minimal Bounding Box [TBD]

[TBD]

2.12 Geometric Reference

2.12.1 Area Calculations

Irregular Triangle (Heron's Formula)



$$A = \sqrt{s \cdot (s - a) \cdot (s - b) \cdot (s - c)}$$

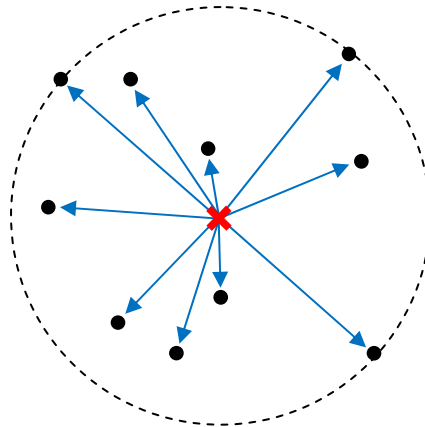
$$\text{where } s = \frac{1}{2}(a + b + c)$$

2.12.2 Volume Calculations [TBD]

[TBD]

2.13 New Ideas

2.13.1 Minimal Bounding Circle [TBD]



[TBD]

<http://forums.wolfram.com/mathgroup/archive/1999/Mar/msg00486.html>

For any practical purpose your suggestions (realized with a numerical minimization) is probably the best. However, we found an algorithm for an "exact" solution. The problem is in fact almost the same in two dimensions; therefore, I will describe the algorithm in two dimensions first:

- 1) Take all pairs of points and calculate the minimal circle covering each pair. (The radius of such a circle is just half the distance between the pair of points, the center is the midpoint between them.)

2) Take all triples of points and calculate the minimal circle covering each triple. (That is just the circum circle.)

3) Throw away all circles which do not cover all points.

4) From the remaining circles take the one with the minimal radius.

This algorithm guarantees to give the correct result.
(Problems might arise from numerics, however.)

Notes:

Can be expanded to sphere

Runs in 2^n time

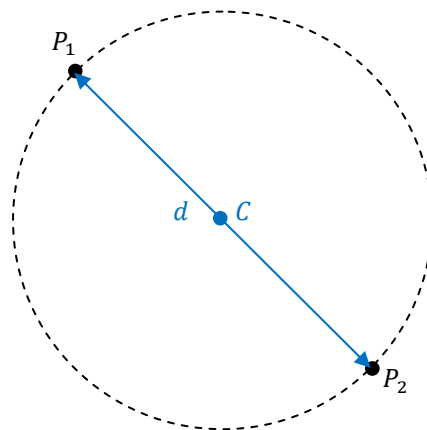
Replace with convex hull to speed up

2.13.2 Minimal Bounding Sphere [TBD]

[TBD]

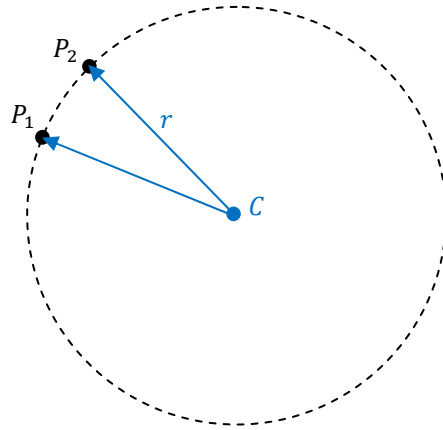
2.13.3 Circle from Two Points [TBD]

[TBD]



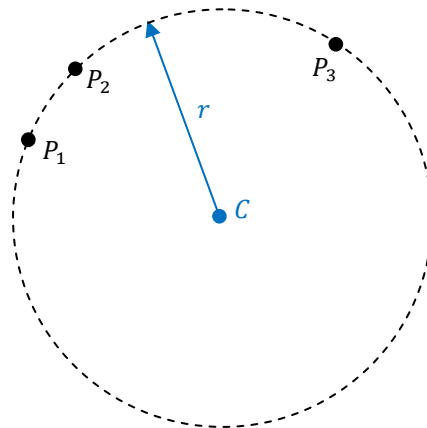
2.13.4 Circle from Two Points and a Radius [TBD]

[TBD]



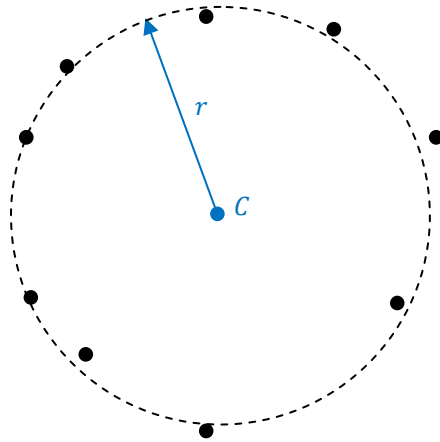
2.13.5 Circle from Three Points [TBD]

[TBD]



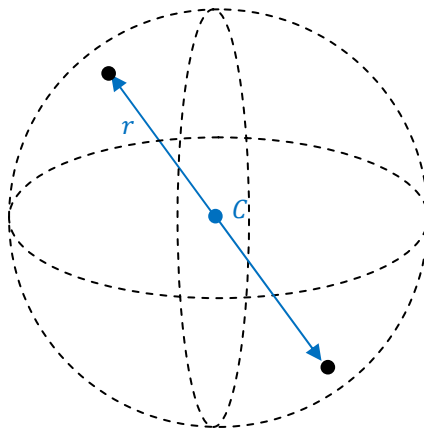
2.13.6 Regress a Circle from N-Points [TBD]

[TBD]



2.13.7 Sphere from Two Points [TBD]

[TBD]



2.13.8 Sphere from Three Points [TBD]

[TBD]

