Geom

Adam Freidin

May 6, 2002

The mathematical reasoning and spirit of the `geom` library,

This document created with LaTeX 2_ε, XY-pic and `vim` Thanks to everyone for everything.

Contents

1 Introduction

Geom is a library that solves many simple common geometrical problems in 3D. It provides functions which find the intersection, distance, class (in/out), nearest point, and interpretation of 3D objects.
Some of the things this library finds are:

The intersection of a line or ray with a plane.
The distance of a point from a plane, line or another point.
The distance of a point "along" a ray.
The class of a point compared to a plane, (front/back/on).
The closest point on a plane or line to some point.
The plane "interpreted" from a triangle.

1.1 Types and Conventions

1.1.1 Conventions Used in this Document

Output variables will be **bold faced**.

Implementation details will be left to the programmer.

The distinction between `float`s and `double`s is ignored as the library handles both equally well.

1.1.2 Conventions Used in the `geom` Library

All function names are prefixed by `geom`, followed by input types, operation, and output types. Example:

All function names in the actual library are suffixed by either a lowercase `'f'` or `'d'`, to indicate either a `float` or `double` version.

1.1.3 Conventions in Common

Function type labels and variables in general share a 1–to–n corespondence, for example a triangle is passed to a function point by point, not as one big structure. The choice is made for a fine blend of minimum data movement and speed.

Each type is represented by a single mnemonic charcter, some better chosen than others. Mnemonic characters in the following section will be represenented by BIG BLOCK LETTERS.

1.1.4 Types

A Triangle ($\vec{p}_1$, $\vec{p}_2$, $\vec{p}_3$) is represented by its verticies.

A Line ($\vec{o}$, $\vec{d}$) will be represented by its two parts, origin ($\vec{o}$) and direction ($\vec{d}$).

Directions ($\vec{d}$) are unit vectors. Origins lie on the line.

A Ray $(\vec{o}, \vec{d})$ has exactly the same representation as a line $(\vec{o}, \vec{d})$.

In this library a ray is bi–directional, but one direction is considered negative. (e.g. plane-ray intersection does not fail if the ray points away, it just results in a negative distance).

A Segment ($\vec{a}$, $\vec{b}$) is defined by its endpoints.

An eDge is a segment, where $\vec{a} \rightarrow \vec{b}$ is ordered counter-clockwise around a polygon.

A Plane ($\mathcal{P}$) is defined by the equation $\mathcal{P}_a x + \mathcal{P}_b y + \mathcal{P}_c z + \mathcal{P}_d = 0$ where the plane normal $\mathcal{P}_{\vec{n}}$ is $(\mathcal{P}_a, \mathcal{P}_b, \mathcal{P}_c)$ and the distance along the normal from the origin is $-\mathcal{P}_d$.

A twospace line ($\mathcal{Q}$, Q), is likewise defined with the equation $\mathcal{Q}_a x + \mathcal{Q}_b y + \mathcal{Q}_c = 0$. where $\|(\mathcal{Q}_a,\ \mathcal{Q}_b)\| = 1$.

A Vector ($\vec{v}$) is is just an ordered triple, $\vec{v} = (x, y, z)$. With $x, y, z \in \mathbb{R}$ of course.

A point ($\vec{x}$, X) has the same format as a vector but represents a position in space.

A 2–vector ($\vec{w}$, W) is an ordered pair, $\vec{w} = (x, y)$.

A 2–point ($\vec{e}$, E) is an ordered pair, $\vec{e} = (x, y)$.

A scalar (z, Z) is just a real number.

A Class (c) is just an integer, with "magic" significant values.

A (quaternion) rotation of a point $\vec{x}$ around a vector $\vec{v}$ by an angle θ will be written as: $\vec{x} \overset{\theta}{\circlearrowleft} \vec{v}$
A projection of $\vec{v}$ onto $\vec{d}$ will be written as: $\vec{v} \angle \vec{d}$.

1.1.5 Operations

`X` means intersection. In the case of `geomPRxZ` this is the distance along the ray before it intersects the plane. However `geomPLxX` should be used to find the point of intersection, and `geomPRdV` to find the distance vector along the ray.

`d` means distance or difference, again, often it is interpreted in either vector or scalar terms.

`c` means classification, usefull for checking non-fuzzy relationships between objects. Examples include point–plane relationships (front/on/back) and point triangle (interior/exterior) relationships.

`n` means nearest, somewhat similar to projection, these functions find the closest point on a plane or line to a reference point.

`i` means interpret, these function convert segements to lines and triangles to planes.

`p` means project, somewhat similar to nearest, in the sense that the point "projected" onto a plane is the nearest point. But project also has the operations of "projecting" a line/ray onto a plane, and of course, projecting a vector onto a vector.

2 A Couple of Preliminaries

Ignore these, these aren't geometrical.

2.1 `int geomZcC(`z`)`

Classifies a number as big enough.

Although it's not really part of the geometry library, it is potentially the most commonly used function. If checking is enabled, then `geomZcC` will be available. It simply returns true iff $|z| > \epsilon$. Any divisor will be checked through this function, and the dividing function will return true iff all checks pass.

2.2 $\mathbb{R}$ `geomSetEpsilon(`ϵ`)`

Sets the library's epsilon (ϵ) value (See above).

2.3 GEOM_CHECK_MODE

Using the preprocessor, you can coerce `geom` into one of three modes.

1. Fast mode, no checking is performed. For the self assured.

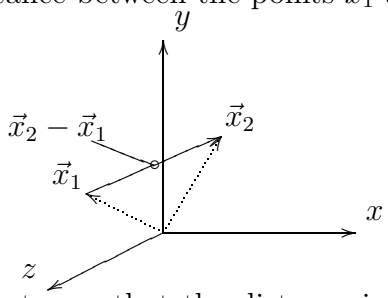
2. Check–cull mode, division checks are performed and calculations abort when failure occurs. Use this mode only if you want the speed and are willing to check return values all the time.

3. Check–continue mode, division checks are performed, but the library continues blithely on anyway. Use this mode if you care about saftey, but want every scrap of (in)stability you can get.

3 Point Functions

3.1 void geomXXdZ(z, $\vec{x}_1$, $\vec{x}_2$)

Returns a scalar distance between the points $\vec{x}_1$ and $\vec{x}_2$.



It should be easy to see that the distance is given by the length of the difference vector $\vec{x}_1 - \vec{x}_2$.

$$z \Leftarrow \|\vec{x}_1 - \vec{x}_2\|$$

3.2 void geomXXdV($\vec{\mathbf{v}}$, $\vec{x}_1$, $\vec{x}_2$)

Returns a vector difference between the points $\vec{x}_1$ and $\vec{x}_2$.

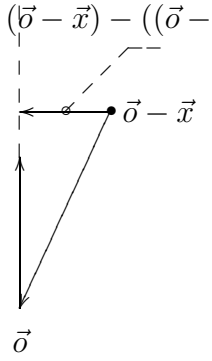
$$\vec{\mathbf{v}} \Leftarrow \vec{x}_1 - \vec{x}_2$$

4 Line Functions

4.1 `void geomLXdV(`$\vec{\mathbf{v}}$`,` $\vec{o}$`,` $\vec{d}$`,` $\vec{x}$`)`

This function calculates the vector from a point to the closest point on a line.

For the vector to be a minimum vector it must be perpendicular to the line. The standard method for doing this is to figure out what part of the vector from $\vec{\mathbf{x}}$ to $\vec{o}$ (the only point–to–line vector we have to work with) points in the direction of the line, and subtract that. What remains is a vector that is perpendicular to the line, and (starting from $\vec{\mathbf{x}}$) lands on it.



$$\vec{\mathbf{v}} \Leftarrow (\vec{o}-\vec{x})-((\vec{o}-\vec{x})\cdot\vec{d})\vec{d}$$

4.2 `void geomLXnX(`$\vec{\mathbf{x}}$`,` $\vec{o}$`,` $\vec{d}$`,` $\vec{x}$`)`

Calculates the closest point on a line to a point.

Just add the point to the distance vector, and we have it:

$$\vec{\mathbf{x}} \Leftarrow (\vec{o}-\vec{x})-\frac{(\vec{o}-\vec{x})\cdot\vec{d}}{\|\vec{d}\|}+\vec{x}$$

4.3 `void geomLXdZ(z,` $\vec{o}$`,` $\vec{d}$`,` $\vec{x}$`)`

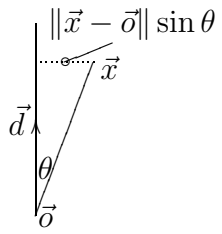
Calculates the distance vector

Although we could reuse our previous result (`geomLXdV`), and just extract its magnitude, there is a faster, slightly more elegent solution for this problem.

Just like the dot product measures how close two vectors are in direction ($\vec{a}\cdot\vec{b}=\|\vec{a}\|\|\vec{b}\|\cos\theta$), the magnitude of the cross product measures how far

away they are in direction[1] ($\|\vec{a} \times \vec{b}\| = \|\vec{a}\|\|\vec{b}\| \sin\theta$).

The distance of a point from a line is exactly $\|(\vec{x} - \vec{o})\| \sin\theta$, where theta measures the angle between $\vec{d}$ and $(\vec{x} - \vec{o})$.



since $\|\vec{d}\| = 1$.

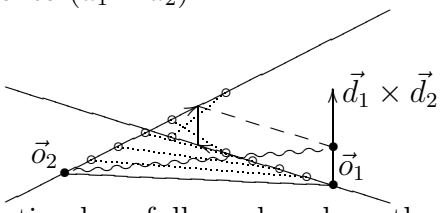
So we can formulate the distance as

$$\mathbf{z} \Leftarrow \sqrt{((\vec{x} - \vec{o}) \times \vec{d}) \cdot ((\vec{x} - \vec{o}) \times \vec{d})}$$

4.4 `checked geomLLdV(`$\vec{\mathbf{v}}$`, `$\vec{o}_1$`, `$\vec{d}_1$`, `$\vec{o}_2$`, `$\vec{d}_2$`)`

Computes the shortest vector that spans the lines (from $\mathcal{L}_1$ to $\mathcal{L}_2$).

The distance between two lines can be found by simply projecting the vector $(\vec{o}_2 - \vec{o}_1)$ onto $(\vec{d}_1 \times \vec{d}_2)$.



As the illustration hopefully makes clear, the projection of $(\vec{o}_2 - \vec{o}_1)$ onto $(\vec{d}_1 \times \vec{d}_2)$ yields the smallest vector that connects the two lines.

$$\vec{\mathbf{v}} \Leftarrow \frac{(\vec{o}_2 - \vec{o}_1) \cdot (\vec{d}_1 \times \vec{d}_2)}{\|\vec{d}_1 \times \vec{d}_2\|^2} (\vec{d}_1 \times \vec{d}_2)$$

4.5 `checked geomLLdZ(`$\vec{\mathbf{v}}$`, `$\vec{o}_1$`, `$\vec{d}_1$`, `$\vec{o}_2$`, `$\vec{d}_2$`)`

Returns the distance between two lines.

[1] Or you could say that it measures the size of the directed plane spanning $\vec{a}$ and $\vec{b}$, hehe

We could just use the length of the result of `geomLLdV`.

$$\left\| \frac{(\vec{o}_2 - \vec{o}_1) \cdot (\vec{d}_1 \times \vec{d}_2)}{\|\vec{d}_1 \times \vec{d}_2\|^2} (\vec{d}_1 \times \vec{d}_2) \right\|$$

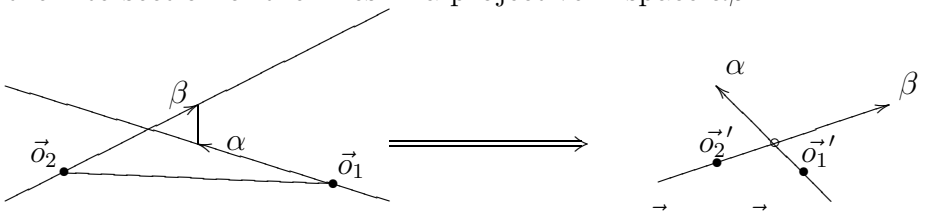
But this simplfies to

$$\mathbf{z} \Leftarrow (\vec{o}_2 - \vec{o}_1) \cdot \frac{\vec{d}_1 \times \vec{d}_2}{\|\vec{d}_1 \times \vec{d}_2\|}$$

4.6 `checked geomLL*(`$\mathcal{OUTPUT}$`,` $\vec{o}_1$`,` $\vec{d}_1$`,` $\vec{o}_2$`,` $\vec{d}_2$`)`

This section describes all of the remaining `geomLL` functions. These functions all compute the things related to the nearest points on a pair of lines. This requires solving a simple linear equation. The system being the intersections of the lines projected into the 2–space plane in which both lie.

In order to find the nearest point on $(\vec{o}_1, \vec{d}_1)$ to $(\vec{o}_2, \vec{d}_2)$ we first solve for the intersection of the lines in a projective 2–space $\alpha\beta$.



We already have some nice $\alpha\beta$ directions, ($\vec{d}_1$ and $\vec{d}_2$).

$$\begin{aligned}
\vec{o}_1{}' &= (\vec{o}_1 \cdot \vec{d}_1, \vec{o}_1 \cdot \vec{d}_2) \\
\vec{o}_2{}' &= (\vec{o}_2 \cdot \vec{d}_1, \vec{o}_2 \cdot \vec{d}_2) \\
\vec{d}_1{}' &= (\vec{d}_1 \cdot \vec{d}_1, \vec{d}_1 \cdot \vec{d}_2) \\
\vec{d}_2{}' &= (\vec{d}_2 \cdot \vec{d}_1, \vec{d}_2 \cdot \vec{d}_2)
\end{aligned}$$

Now we can solve for

$$\vec{o}_1{}' + \vec{d}_1{}' t = \vec{o}_2{}' + \vec{d}_2{}' u$$

in both α and β, resulting in

$$t = \frac{\vec{d}_{2\beta}(\vec{o}_{1\alpha} - \vec{o}_{2\alpha}) - \vec{d}_{2\alpha}(\vec{o}_{1\beta} - \vec{o}_{2\beta})}{\vec{d}_{2\alpha}\vec{d}_{1\beta}{}' - \vec{d}_{1\alpha}\vec{d}_{2\beta}{}'}$$

Sorry it's complicated, but the problem is hard.

Note that $\vec{o}_1 + \vec{d}_1 t$ is nearest point on $(\vec{o}_1, \vec{d}_1)$ to $(\vec{o}_2, \vec{d}_2)$.

All of the other solutions are mearly offsets (of `geomLLdV`) from this point and calculations of distance, etc...

The solutions are:

4.6.1 `checked geomLLnX(`$\vec{\mathbf{x}}$`,` $\vec{o}_1$`,` $\vec{d}_1$`,` $\vec{o}_2$`,` $\vec{d}_2$`)`

$$\vec{\mathbf{x}} \Leftarrow \vec{o}_1 + t\vec{d}_1$$

4.6.2 `checked geomLLnnXX(`$\vec{\mathbf{x}}$`,` $\vec{\mathbf{x}}_2$`,` $\vec{o}_1$`,` $\vec{d}_1$`,` $\vec{o}_2$`,` $\vec{d}_2$`)`

$$\vec{\mathbf{x}} \Leftarrow \vec{o}_1 + t\vec{d}_1$$
$$\vec{\mathbf{x}}_2 \Leftarrow \vec{\mathbf{x}} + \vec{d}_1 \times \vec{d}_2$$

4.6.3 `checked geomLLndXV(`$\vec{\mathbf{x}}$`,` $\vec{\mathbf{v}}$`,` $\vec{o}_1$`,` $\vec{d}_1$`,` $\vec{o}_2$`,` $\vec{d}_2$`)`

$$\vec{\mathbf{x}} \Leftarrow \vec{o}_1 + t\vec{d}_1$$
$$\vec{\mathbf{v}} \Leftarrow \vec{d}_1 \times \vec{d}_2$$

4.6.4 `checked geomLLndXZ(`$\vec{\mathbf{x}}$`,` $\mathbf{z}$`,` $\vec{o}_1$`,` $\vec{d}_1$`,` $\vec{o}_2$`,` $\vec{d}_2$`)`

$$\vec{\mathbf{x}} \Leftarrow \vec{o}_1 + t\vec{d}_1$$
$$\mathbf{z} \Leftarrow \|\vec{d}_1 \times \vec{d}_2\|$$

5 Plane Functions

5.1 `void geomPXdZ(`$\mathbf{z}$`,` $\mathcal{P}$`,` $\vec{x}$`)`

Finds the distance of a point from a plane.

As mentioned above, our stipulation that $\mathcal{P}_{\vec{n}}$ is a unit vector implies that the distance can be found by evaluating the plane equation

$$\mathbf{z} \Leftarrow \mathcal{P}_a x + \mathcal{P}_b y + \mathcal{P}_c z + \mathcal{P}_d$$

or, more succinctly,

$$\mathbf{z} \Leftarrow \vec{x} \cdot \mathcal{P}_{\vec{n}} + \mathcal{P}_d$$

5.2 `void geomPXdV(`$\vec{\mathbf{v}}$`,` $\mathcal{P}$`,` $\vec{x}$`)`

Finds the vector that points from a point $\vec{\mathbf{x}}$ to the nearest point on a plane.

This vector will be parallel to $\mathcal{P}_{\vec{n}}$ and have the length of the distance from the plane. However, if the point is in front of the plane, then it's distance will be positive, but the direction to the plane is in the opposite direction. A negative solves that problem.

$$\vec{\mathbf{v}} \Leftarrow -(\mathcal{P}_{\vec{n}} \cdot \vec{x} + \mathcal{P}_d)\mathcal{P}_{\vec{n}}$$

5.3 `void geomPXnX(`$\vec{\mathbf{x}}$`,` $\mathcal{P}$`,` $\vec{x}$`)`

Finds the closest point on a plane, given a point.

This point will be the same as the sum of the difference vector to the plane, and the original point.

$$\vec{\mathbf{x}} \Leftarrow \vec{x} - (\mathcal{P}_{\vec{n}} \cdot \vec{x} + \mathcal{P}_d)\mathcal{P}_{\vec{n}}$$

5.4 `checked geomPRxZ(z,` $\mathcal{P}$`,` $\vec{o}$`,` $\vec{d}$`)`

Finds the distance to a plane along a ray (distance to intersection).

Note that the distance along the ray will be the distance from the origin of the ray divided by the component of ray direction towards the plane. The component of ray direction towards the plane is $-\mathcal{P}_{\vec{n}} \cdot \vec{d}$. The distance from the plane is the now familiar $\mathcal{P}_{\vec{n}} \cdot \vec{o} + \mathcal{P}_d$.

Together we have

$$\mathbf{z} \Leftarrow -\frac{\mathcal{P}_{\vec{n}} \cdot \vec{d}}{\mathcal{P}_{\vec{n}} \cdot \vec{o} + \mathcal{P}_d}$$

5.5 `checked geomPRdV(`$\vec{\mathbf{v}}$`,` $\mathcal{P}$`,` $\vec{o}$`,` $\vec{d}$`)`

Finds the distance vector along a ray to a plane.

This is clearly $\vec{d}$ scaled by the result of `geomPRxZ`.

$$\vec{\mathbf{v}} \Leftarrow -\frac{\mathcal{P}_{\vec{n}} \cdot \vec{d}}{\mathcal{P}_{\vec{n}} \cdot \vec{o} + \mathcal{P}_d}\vec{d}$$

5.6 `checked geomPLxX(`$\vec{\mathbf{x}}$`, `$\mathcal{P}$`, `$\vec{o}$`, `$\vec{d}$`)`

Finds the intersection of a line and a plane.

This is clearly the sum of $\vec{o}$ and the result of `geomPRdV`.

$$\vec{\mathbf{x}} \Leftarrow \vec{o} - \frac{\mathcal{P}_{\vec{n}} \cdot \vec{d}}{\mathcal{P}_{\vec{n}} \cdot \vec{o} + \mathcal{P}_d} \vec{d}$$

5.7 `checked geomPPxL(`$\vec{\mathbf{o}}$`, `$\vec{\mathbf{d}}$`, `$\mathcal{P}$`, `$\mathcal{Q}$`)`

Finds the line of intersection of two planes.

Sounds hard doesn't it? Well, it comes down to two problems:

1. Find $\vec{\mathbf{o}}$, a point on the line of intersection
2. Find $\vec{\mathbf{d}}$, the direction of the line of intersecton

Problem 1 is easy. $\vec{\mathbf{d}}$ is perpendicular to the normals of the planes.

$$\vec{\mathbf{d}} \Leftarrow \frac{\mathcal{P}_{\vec{n}} \times \mathcal{Q}_{\vec{n}}}{\|\mathcal{P}_{\vec{n}} \times \mathcal{Q}_{\vec{n}}\|}$$

Problem 2 is solved by realizing that $\vec{\mathbf{o}}$ is an intersection of a line on $\mathcal{P}$ with $\mathcal{Q}$. Unfortunately the math kind of explodes here, but if you stare at this equation and the equation for `geomPLxX` you'll se that they share the same form, (with an extra correction term to account for $\mathcal{P}_{\vec{n}} \times \mathcal{Q}_{\vec{n}} \times \mathcal{P}_{\vec{n}}$ not being normalized, which, by the way, is my method for creating a vector in the "direction" of the center term, but perpendicular to the outer term).

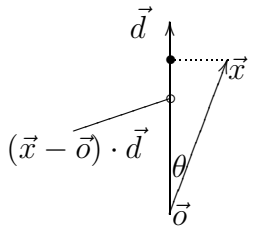
$$\vec{\mathbf{o}} \Leftarrow (-\mathcal{P}_d \mathcal{P}_{\vec{n}}) - \frac{\mathcal{Q}_{\vec{n}} \cdot (\mathcal{P}_{\vec{n}} \times \mathcal{Q}_{\vec{n}} \times \mathcal{P}_{\vec{n}})}{\mathcal{Q}_{\vec{n}} \cdot (-\mathcal{P}_d \mathcal{P}_{\vec{n}}) + \mathcal{Q}_d} \frac{\mathcal{P}_{\vec{n}} \times \mathcal{Q}_{\vec{n}} \times \mathcal{P}_{\vec{n}}}{\|\mathcal{P}_{\vec{n}} \times \mathcal{Q}_{\vec{n}} \times \mathcal{P}_{\vec{n}}\|^2}$$

There's an optimization that lets you remove the $\|\mathcal{P}_{\vec{n}} \times \mathcal{Q}_{\vec{n}} \times \mathcal{P}_{\vec{n}}\|^2$ term by doing a little extra work (in code only), can you spot it?

5.8 `void geomRXpZ(z, `$\vec{o}$`, `$\vec{d}$`, `$\vec{x}$`)`

Finds the projected distance of a point along a ray.

This is the length of the vector from the origin of the ray to point along the ray. The dot product does this nicely...



So

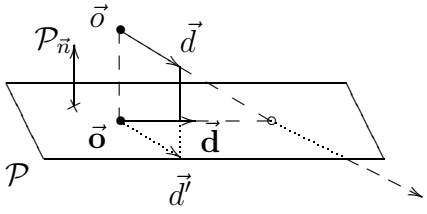
$$\mathbf{z} \Leftarrow (\vec{x}-\vec{o})\cdot\vec{d}$$

5.9 `checked geomPRpR(`$\vec{\mathbf{o}}$`,` $\vec{\mathbf{d}}$`,` $\mathcal{P}$`,` $\vec{o}$`,` $\vec{d}$`)`

Projects a ray onto a plane.

The origin of the new ray is the closest point on the plane to the origin of the old ray (see `geomPXnX`):

We can do this by projecting the origin onto the plane, and removing the component of d in the direction of the plane normal.



Thus we have

$$\vec{\mathbf{o}} \Leftarrow \vec{x} - (\mathcal{P}_{\vec{n}}\cdot\vec{x} + \mathcal{P}_d)\mathcal{P}_{\vec{n}}$$

from `geomPXnX`, and

$$\vec{\mathbf{d}} \Leftarrow \frac{\vec{d} - \frac{\mathcal{P}_{\vec{n}}\cdot\vec{d}}{\|\mathcal{P}_{\vec{n}}\|}\mathcal{P}_{\vec{n}}}{\left\|\vec{d} - \frac{\mathcal{P}_{\vec{n}}\cdot\vec{d}}{\|\mathcal{P}_{\vec{n}}\|}\mathcal{P}_{\vec{n}}\right\|}$$

6 Interpretive Functions

6.1 `checked geomViU(`$\vec{\mathbf{u}}$`,` $\vec{v}$`)`

Normalizes a vector, (a normalized vector has a length of 1).

All you have to do to ensure that a vector has a length of 1 is divide it by its length.

$$\vec{\mathbf{u}} \Leftarrow \frac{\vec{v}}{\|\vec{v}\|}$$

6.2 checked geomSiR($\vec{\mathbf{o}}$, $\vec{\mathbf{d}}$, $\vec{a}$, $\vec{b}$)

Interprets a line segment as a ray.

Rays are a combination of a point and a (normalized) direction.

$$\vec{\mathbf{o}} \Leftarrow \vec{a}$$

$$\vec{\mathbf{d}} \Leftarrow \frac{\vec{b} - \vec{a}}{\|\vec{b} - \vec{a}\|}$$

6.3 checked geomTiP($\mathcal{P}$, $\vec{p}_1$, $\vec{p}_2$, $\vec{p}_3$)

Calculates the plane in which the triangle lies.

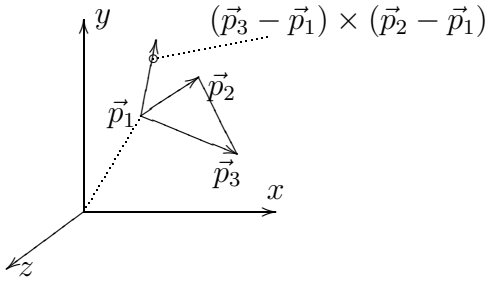
Planes are described by the equation

$$\mathcal{P}_a x + \mathcal{P}_b y + \mathcal{P}_c z + \mathcal{P}_d = 0$$

This is a rather meaningless definition, except that on further inspection, the vector $(\mathcal{P}_a, \mathcal{P}_b, \mathcal{P}_c)$ is normal to the plane and $\mathcal{P}_d$ describes a negative distance from the origin.

A plane $\mathcal{P}$ may then be described by the pair $(\vec{\mathbf{n}}, -\mathbf{d})$. If $\|\vec{\mathbf{n}}\| = 1$ then $-\mathbf{d}$ is indeed the distance from the origin along $\vec{\mathbf{n}}$. In addition, the plane equation $\mathcal{P}_a x + \mathcal{P}_b y + \mathcal{P}_c z + \mathcal{P}_d$ is then the (signed) distance from the plane.

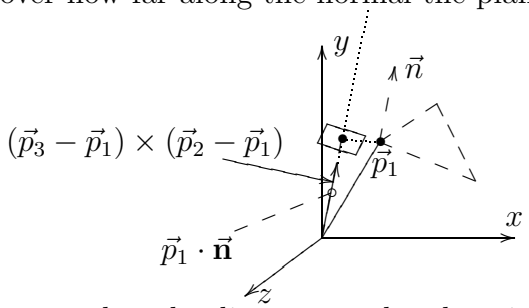
Everyone knows that the cross product is great for generating normals, so $\vec{n}$ will be in the direction of any two edges crossed $(\vec{p}_2 - \vec{p}_1) \times (\vec{p}_3 - \vec{p}_1)$.



But even though it is a normal, it is not in general a unit (normalized) vector. So we must normalize it:

$$\vec{\mathbf{n}} \Leftarrow \frac{(\vec{p}_2 - \vec{p}_1) \times (\vec{p}_3 - \vec{p}_1)}{\|(\vec{p}_2 - \vec{p}_1) \times (\vec{p}_3 - \vec{p}_1)\|}$$

To find $\mathbf{d}$, we need to figure out how far away from the origin for some point on the plane, so we take one of the triangles points and dot it with the normal to discover how far along the normal the plane lies.



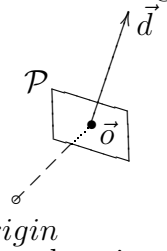
From this we see that the distance to the plane is $\vec{p}_1 \cdot \vec{\mathbf{n}}$ and so

$$\mathbf{d} \Leftarrow -\vec{p}_1 \cdot \vec{\mathbf{n}}$$

The negative accounts for the fact that d itself is the negative distance along $\vec{n}$ from the origin.

6.4 `void geomRiP(`$\mathcal{P}$`,` $\vec{o}$`,` $\vec{d}$`)`

Computes the plane normal to $\vec{d}$ and containing the ray origin $\vec{o}$.



Since we have already defined the plane in normal-distance form

$$\mathcal{P}_{\vec{n}} \Leftarrow \vec{d}$$

The distance $\mathcal{P}_d$ along the normal is just the algebraic opposite of the length of $\vec{o}$ projected onto $\vec{d}$, but since $\|\vec{d}\| = 1$ this is simply

$$\mathcal{P}_d \Leftarrow -\vec{o} \cdot \vec{d}$$

6.5 checked geomVVXiE($\vec{\mathbf{e}}$, $\vec{v}_1$, $\vec{v}_2$, $\vec{x}$)

Locates a point in a 2–space given by two basis vectors.

Linear algebra tells us that

$$\vec{\mathbf{e}} \Leftarrow (\vec{x} \cdot \vec{v}_1, \vec{x} \cdot \vec{v}_2)$$

Which is the amount of $\vec{x}$ in the $\vec{v}_1$ and $\vec{v}_2$ directions.

7 Special Functions

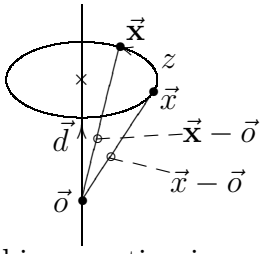
This section contains descriptions of the special functions, the include things like rotation and classification, and other more esoteric operations.

7.1 void geomLXZrX($\vec{\mathbf{x}}$, $\vec{o}$, $\vec{d}$, $\vec{x}$, z)

Rotates point $\vec{x}$ around line $(\vec{o}, \vec{d})$ by angle z in radians.

This function uses the quaternion rotation of `vector.c`, which rotates a point around some direction/vector with origin at the origin $(0, 0, 0)$.

In order to rotate $\vec{x}$ around our line, we rotate $\vec{x} - \vec{o}$ around $\vec{d}$. And then add $\vec{o}$ back.



The equation for this operation is

$$\vec{\mathbf{x}} \Leftarrow (\vec{x} - \vec{o}) \overset{z}{\circlearrowleft} \vec{d} + \vec{o}$$

7.2 void geomLRZrR($\vec{\mathbf{o}}$, $\vec{\mathbf{d}}$, $\vec{o}_l$, $\vec{d}_l$, $\vec{o}_r$, $\vec{d}_r$)

Rotates a ray $(\vec{o}_r, \vec{d}_r)$ around a line $(\vec{o}_l, \vec{d}_l)$.

The way to rotate a ray is to both rotate it's origin and direction, thus we have

$$\vec{\mathbf{o}} \Leftarrow (\vec{o}_r - \vec{o}_l) \overset{z}{\circlearrowleft} \vec{d}_l + \vec{o}$$

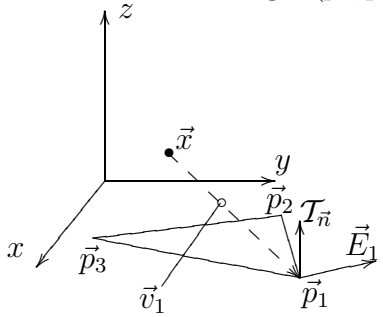
and

$$\vec{\mathbf{d}} \Leftarrow \vec{d}_r \overset{z}{\circlearrowleft} \vec{d}_l$$

7.3 `void geomTXcC(c, $\vec{p}_1$, $\vec{p}_2$, $\vec{p}_3$, $\vec{x}$)`

Classifies a point as inside or outside a triangle.

We do this by checking the relation of the point to the vector normal to the edge in the plane of the triangle. In other words, we look at each $\vec{v}_n = \vec{p}_n - \vec{x}$ and compare it to the edge normal $\vec{E}_n = (\vec{p}_{n+1} - \vec{p}_n) \times \mathcal{T}_{\vec{n}}$. Where $\mathcal{T}_{\vec{n}}$ is $\mathcal{P}_{\vec{n}}$ of the plane in which the triangle $(\vec{p}_1, \vec{p}_2, \vec{p}_3)$ lies.



If $\vec{x}$ is behind the plane which lies along $\mathcal{T}_{\vec{n}}$ and through edge $(\vec{p}_n, \vec{p}_{n+1})$ then $\vec{p}_n - \vec{x}$ will be in the same direction of $\vec{E}_n$. And iff $\vec{x}$ lies behind all of these planes, then it is "inside" the triangle. The dot product again comes in handy to check the relative directionality of $\vec{p} - \vec{x}_n$ to $\vec{E}_n$.

Let $\vec{E}_n$ be the exterior planar normal to edge e_n, or $\mathcal{T}_{\vec{n}} \times (\vec{p}_{n+1} - \vec{p}_n)$, and let $\vec{v}_n$ be the difference vector $\vec{p} - \vec{x}_n$, then

$$\mathbf{c} \Leftarrow \begin{cases} 1 \text{ if } \forall n & \vec{v}_n \cdot \vec{E}_n \geq 0 \\ 0 \text{ if } \exists n & \vec{v}_n \cdot \vec{E}_n < 0 \end{cases}$$

7.4 `void geomTVXcC(c, $\vec{p}_1$, $\vec{p}_2$, $\vec{p}_3$, $\vec{v}$, $\vec{x}$)`

This classifies a point as inside a triangle like before, but rather than using the triangle normal $\mathcal{T}_{\vec{n}}$, it uses some passed in vector $\vec{v}$. The affect this has is to check if the point is inside the triangle *along* vector $\vec{v}$.

If we redefine $\vec{E}$ to $\vec{v} \times (\vec{p}_{n+1} - \vec{p}_n)$, then the same algorithm holds

$$\mathbf{c} \Leftarrow \begin{cases} 1 \text{ if } \forall n & \vec{v}_n \cdot \vec{E}_n \geq 0 \\ 0 \text{ if } \exists n & \vec{v}_n \cdot \vec{E}_n < 0 \end{cases}$$

7.5 `void geomTRcC(c, $\vec{p}_1$, $\vec{p}_2$, $\vec{p}_3$, $\vec{o}$, $\vec{d}$)`

This function classifies a ray as inside a triangle or not.

It is nothing more than a reformat of `geomTVXcC`. The `geom` library does nothing more than call `geomTVXcC(`**c**`,` $\vec{p}_1$`,` $\vec{p}_2$`,` $\vec{p}_3$`,` $\vec{d}$`,` $\vec{o}$`)`

7.6 `void geomPDXcC(`**c**`,` $\mathcal{P}$`,` $\vec{a}$`,` $\vec{b}$`,` $\vec{x}$`)`

This classifies a point to being interior to a polygon/plane and an edge of that polygon using the same algorithm as before.

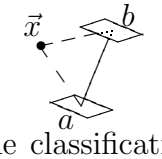
This function generalizes interior checking for an arbitrary convex polygon. To use it, simply check each side, if it classifies the point as inside for every side, then the point is inside. Although `geom` takes a plane $\mathcal{P}$, it only uses the first three elements, so a vector $\vec{v}$ could be used instead (for `geomDVXcC` style functionality).

$$\mathbf{c} \Leftarrow \begin{cases} 1 & \text{if } \mathcal{P}_{\vec{n}} \cdot \vec{E}_n \geq 0 \\ 0 & \text{if } \mathcal{P}_{\vec{n}} \cdot \vec{E}_n < 0 \end{cases}$$

7.7 `checked geomSXcC(`**c**`,` $\vec{a}$`,` $\vec{b}$`,` $\vec{x}$`)`

Determines whether a point is "inside" a segment.

Inside a segment means between planes normal to the segment at the endpoints.



So we just use a modified plane classification formulas, first we set up our plane direction vector, $\vec{d} = (\vec{b} - \vec{a})/\|\vec{b} - \vec{a}\|$, and our distances from the origin, $s = -\vec{d} \cdot \vec{a}$, and $t = -\vec{d} \cdot \vec{b}$.

$$\mathbf{c} \Leftarrow \begin{cases} 1 & \text{if } \vec{x} \cdot \vec{d} + s > 0 \text{ and } \vec{x} \cdot \vec{d} - t < 0 \\ 0 & \text{if } \vec{x} \cdot \vec{d} + s \leq 0 \text{ or } \vec{x} \cdot \vec{d} - t \geq 0 \end{cases}$$

7.8 `checked geomVVpV(`$\vec{\mathbf{v}}$`,` $\vec{v}$`,` $\vec{r}$`)`

Finds the projection of $\vec{v}$ on $\vec{r}$. Although this function is entirely trivial, it is here for completeness and to be checked in the same framework as the rest of the library.

$$\vec{\mathbf{v}} \Leftarrow \frac{\vec{v} \cdot \vec{r}}{\|\vec{r}\|^2} \vec{r}$$

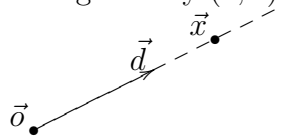
7.9 `void geomVUpV(`$\vec{\mathbf{v}}$`,` $\vec{v}$`,` $\vec{r}$`)`

Finds the projection of $\vec{v}$ on a unit vector $\vec{r}$. Again, it is here for completeness only.

$$\vec{\mathbf{v}} \Leftarrow (\vec{v} \cdot \vec{r})\vec{r}$$

7.10 `void geomRZiX(`$\vec{\mathbf{x}}$`,` $\vec{o}$`,` $\vec{d}$`,` z`)`

Locates a point z distance along the ray $(\vec{o}, \vec{d})$.



This is simply

$$\vec{\mathbf{x}} \Leftarrow \vec{o} + z\vec{d}$$

A Extras

A.1 Types and operations as explained in geom.h

The types and naming conventions as explained in the C source file `geom.h`

```
/*--------------------\
| <<Naming>>
| geom{SrcTypes}{Operation(s)}{DstTypes}[fd](DstList..., SrcList...);
| Sources and destinations are listed
|  1. in size order, larger first, (see list below)
|  2. "operated on" before "environment" (project(dst, a, l))
|  3. as noted.
+---------------------+
| <<Types>>
| T:triangle     : p1[3], p2[3], p3[3]
| L:line         : origin[3], direction[3]
| R:ray          : origin[3], direction[3]
| S:segement     : a[3], b[3]
| D:edge         : a[3], b[3], counter clockwise around polygon
| P:plane        : p[4] = {A,B,C,D},
|                         defined by A*x + B*y + C*z + D == 0
| Q:2-space line : q[3] = {A,B,C},
|                         defined by A*x + B*y + C == 0
| V:vector       : v[3]
| U:unit vector  : v[3], norm3(v) == 1.0
| X:point        : x[3]
| W:2-direction  : w[2]
| E:2-point      : e[2]
| Z:scalar       : z
| C:class int    : i, bool or enum
+---------------------+
| <<Operations>>
| x:intersection
| d:distance
| c:classification
| n:nearest
| i:interpret
| p:project
\--------------------*/
```