Geometry Utilities (mathutils.geometry)

The Blender geometry module

mathutils.geometry.**area_tri(v1, v2, v3)**

Returns the area size of the 2D or 3D triangle defined.

PARAMETERS:

- **v1** (`mathutils.Vector`) – Point1
- **v2** (`mathutils.Vector`) – Point2
- **v3** (`mathutils.Vector`) – Point3

RETURN TYPE:

float

mathutils.geometry.**barycentric_transform(point, tri_a1, tri_a2, tri_a3, tri_b1, tri_b2, tri_b3)**

Return a transformed point, the transformation is defined by 2 triangles.

PARAMETERS:

- **point** (`mathutils.Vector`) – The point to transform.
- **tri_a1** (`mathutils.Vector`) – source triangle vertex.
- **tri_a2** (`mathutils.Vector`) – source triangle vertex.
- **tri_a3** (`mathutils.Vector`) – source triangle vertex.
- **tri_b1** (`mathutils.Vector`) – target triangle vertex.
- **tri_b2** (`mathutils.Vector`) – target triangle vertex.
- **tri_b3** (`mathutils.Vector`) – target triangle vertex.

RETURNS:

The transformed point

RETURN TYPE:

`mathutils.Vector`

mathutils.geometry.**box_fit_2d(points)**

Returns an angle that best fits the points to an axis aligned rectangle

PARAMETERS:

points (*Sequence[Sequence[float]]*) – Sequence of 2D points.

RETURNS:

angle

RETURN TYPE:

float

mathutils.geometry.**box_pack_2d(boxes)**

Returns a tuple with the width and height of the packed bounding box.

PARAMETERS:

boxes (*list[list[float]]*) – list of boxes, each box is a list where the first 4 items are [X, Y, width, height, …] other items are ignored. The X and Y values in this list are modified to set the packed positions.

RETURNS:

The width and height of the packed bounding box.

RETURN TYPE:

tuple[float, float]

mathutils.geometry.**closest_point_on_tri(pt, tri_p1, tri_p2, tri_p3)**

Takes 4 vectors: one is the point and the next 3 define the triangle.

PARAMETERS:

- **pt** (`mathutils.Vector`) – Point
- **tri_p1** (`mathutils.Vector`) – First point of the triangle
- **tri_p2** (`mathutils.Vector`) – Second point of the triangle
- **tri_p3** (`mathutils.Vector`) – Third point of the triangle

RETURNS:

The closest point of the triangle.

RETURN TYPE:

`mathutils.Vector`

mathutils.geometry.**convex_hull_2d(points)**

Returns a list of indices into the list given

PARAMETERS:

points (*Sequence[Sequence[float]]*) – Sequence of 2D points.

RETURNS:

a list of indices

RETURN TYPE:

list[int]

mathutils.geometry.**delaunay_2d_cdt(vert_coords, edges, faces, output_type, epsilon, need_ids=True)**

Computes the Constrained Delaunay Triangulation of a set of vertices, with edges and faces that must appear in the triangulation. Some triangles ma be eaten away, or combined with other triangles, according to output type. The returned verts may be in a different order from input verts, may be moved slightly, and may be merged with other nearby verts. The three returned orig lists give, for each of verts, edges, and faces, the list of input element indices corresponding to the positionally same output element. For edges, the orig indices start with the input edges and then continue with the edges implied by each of the faces (n of them for an n-gon). If the need_ids argument is supplied, and False, then the code skips the preparatio of the orig arrays, which may save some time.

PARAMETERS:

- **vert_coords** (Sequence[`mathutils.Vector`]) – Vertex coordinates (2d)
- **edges** (*Sequence[Sequence[int, int]]*) – Edges, as pairs of indices in `vert_coords`
- **faces** (*Sequence[Sequence[int]]*) – Faces, each sublist is a face, as indices in *vert_coords* (CCW oriented)
- **output_type** (*int*) – What output looks like. 0 => triangles with convex hull. 1 => triangles inside constraints. 2 => the input constraints, intersected. 3 => like 2 but detect holes and omit them from output. 4 => like 2 but with extra edges to make valid BMesh faces. 5 => like 4 but detect holes and omit them from output.
- **epsilon** (*float*) – For nearness tests; should not be zero
- **need_ids** – are the orig output arrays needed?

RETURNS:

Output tuple, (vert_coords, edges, faces, orig_verts, orig_edges, orig_faces)

RETURN TYPE:

tuple[list[`mathutils.Vector`], list[tuple[int, int]], list[list[int]], list[list[int]], list[list[int]], list[list[int]]]

mathutils.geometry.**distance_point_to_plane(pt, plane_co, plane_no)**

Returns the signed distance between a point and a plane (negative when below the normal).

PARAMETERS:

- **pt** (`mathutils.Vector`) – Point
- **plane_co** (`mathutils.Vector`) – A point on the plane

- **plane_no** (`mathutils.Vector`) – The direction the plane is facing

RETURN TYPE:

float

mathutils.geometry.**interpolate_bezier(knot1, handle1, handle2, knot2, resolution)**

Interpolate a bezier spline segment.

PARAMETERS:

- **knot1** (`mathutils.Vector`) – First bezier spline point.
- **handle1** (`mathutils.Vector`) – First bezier spline handle.
- **handle2** (`mathutils.Vector`) – Second bezier spline handle.
- **knot2** (`mathutils.Vector`) – Second bezier spline point.
- **resolution** (*int*) – Number of points to return.

RETURNS:

The interpolated points.

RETURN TYPE:

list[`mathutils.Vector`]

mathutils.geometry.**intersect_line_line(v1, v2, v3, v4)**

Returns a tuple with the points on each line respectively closest to the other.

PARAMETERS:

- **v1** (`mathutils.Vector`) – First point of the first line
- **v2** (`mathutils.Vector`) – Second point of the first line
- **v3** (`mathutils.Vector`) – First point of the second line
- **v4** (`mathutils.Vector`) – Second point of the second line

RETURNS:

The intersection on each line or None when the lines are co-linear.

RETURN TYPE:

tuple[`mathutils.Vector`, `mathutils.Vector`] | None

mathutils.geometry.**intersect_line_line_2d(lineA_p1, lineA_p2, lineB_p1, lineB_p2)**

Takes 2 segments (defined by 4 vectors) and returns a vector for their point of intersection or None.

> Warning
>
> Despite its name, this function works on segments, and not on lines.

PARAMETERS:

- **lineA_p1** (`mathutils.Vector`) – First point of the first line
- **lineA_p2** (`mathutils.Vector`) – Second point of the first line
- **lineB_p1** (`mathutils.Vector`) – First point of the second line
- **lineB_p2** (`mathutils.Vector`) – Second point of the second line

RETURNS:

The point of intersection or None when not found

RETURN TYPE:

`mathutils.Vector` | None

mathutils.geometry.**intersect_line_plane(line_a, line_b, plane_co, plane_no, no_flip=False)**

Calculate the intersection between a line (as 2 vectors) and a plane. Returns a vector for the intersection or None.

PARAMETERS:

PARAMETERS:

- **line_a** (mathutils.Vector) – First point of the first line
- **line_b** (mathutils.Vector) – Second point of the first line
- **plane_co** (mathutils.Vector) – A point on the plane
- **plane_no** (mathutils.Vector) – The direction the plane is facing

RETURNS:

The point of intersection or None when not found

RETURN TYPE:

mathutils.Vector | None

mathutils.geometry.**intersect_line_sphere(line_a, line_b, sphere_co, sphere_radius, clip=True)**

Takes a line (as 2 points) and a sphere (as a point and a radius) and returns the intersection

PARAMETERS:

- **line_a** (mathutils.Vector) – First point of the line
- **line_b** (mathutils.Vector) – Second point of the line
- **sphere_co** (mathutils.Vector) – The center of the sphere
- **sphere_radius** (*float*) – Radius of the sphere

RETURNS:

The intersection points as a pair of vectors or None when there is no intersection

RETURN TYPE:

tuple[mathutils.Vector | None, mathutils.Vector | None]

mathutils.geometry.**intersect_line_sphere_2d(line_a, line_b, sphere_co, sphere_radius, clip=True)**

Takes a line (as 2 points) and a sphere (as a point and a radius) and returns the intersection

PARAMETERS:

- **line_a** (mathutils.Vector) – First point of the line
- **line_b** (mathutils.Vector) – Second point of the line
- **sphere_co** (mathutils.Vector) – The center of the sphere
- **sphere_radius** (*float*) – Radius of the sphere

RETURNS:

The intersection points as a pair of vectors or None when there is no intersection

RETURN TYPE:

tuple[mathutils.Vector | None, mathutils.Vector | None]

mathutils.geometry.**intersect_plane_plane(plane_a_co, plane_a_no, plane_b_co, plane_b_no)**

Return the intersection between two planes

PARAMETERS:

- **plane_a_co** (mathutils.Vector) – Point on the first plane
- **plane_a_no** (mathutils.Vector) – Normal of the first plane
- **plane_b_co** (mathutils.Vector) – Point on the second plane
- **plane_b_no** (mathutils.Vector) – Normal of the second plane

RETURNS:

The line of the intersection represented as a point and a vector or None if the intersection can't be calculated

RETURN TYPE:

tuple[mathutils.Vector, mathutils.Vector] | tuple[None, None]

mathutils.geometry.**intersect_point_line(pt, line_p1, line_p2)**

Takes a point and a line and returns a tuple with the closest point on the line and its distance from the first point of the line as a percentage of the length of the line.

PARAMETERS:

- **pt** (`mathutils.Vector`) – Point
- **line_p1** (`mathutils.Vector`) – First point of the line
- **line_p1** – Second point of the line

RETURN TYPE:

tuple[`mathutils.Vector`, float]

mathutils.geometry.**intersect_point_quad_2d(pt, quad_p1, quad_p2, quad_p3, quad_p4)**

Takes 5 vectors (using only the x and y coordinates): one is the point and the next 4 define the quad, only the x and y are used from the vectors. Returns 1 if the point is within the quad, otherwise 0. Works only with convex quads without singular edges.

PARAMETERS:

- **pt** (`mathutils.Vector`) – Point
- **quad_p1** (`mathutils.Vector`) – First point of the quad
- **quad_p2** (`mathutils.Vector`) – Second point of the quad
- **quad_p3** (`mathutils.Vector`) – Third point of the quad
- **quad_p4** (`mathutils.Vector`) – Fourth point of the quad

RETURN TYPE:

int

mathutils.geometry.**intersect_point_tri(pt, tri_p1, tri_p2, tri_p3)**

Takes 4 vectors: one is the point and the next 3 define the triangle. Projects the point onto the triangle plane and checks if it is within the triangle.

PARAMETERS:

- **pt** (`mathutils.Vector`) – Point
- **tri_p1** (`mathutils.Vector`) – First point of the triangle
- **tri_p2** (`mathutils.Vector`) – Second point of the triangle
- **tri_p3** (`mathutils.Vector`) – Third point of the triangle

RETURNS:

Point on the triangles plane or None if its outside the triangle

RETURN TYPE:

`mathutils.Vector` | None

mathutils.geometry.**intersect_point_tri_2d(pt, tri_p1, tri_p2, tri_p3)**

Takes 4 vectors (using only the x and y coordinates): one is the point and the next 3 define the triangle. Returns 1 if the point is within the triangle, otherwise 0.

PARAMETERS:

- **pt** (`mathutils.Vector`) – Point
- **tri_p1** (`mathutils.Vector`) – First point of the triangle
- **tri_p2** (`mathutils.Vector`) – Second point of the triangle
- **tri_p3** (`mathutils.Vector`) – Third point of the triangle

RETURN TYPE:

int

mathutils.geometry.**intersect_ray_tri(v1, v2, v3, ray, orig, clip=True)**

Returns the intersection between a ray and a triangle, if possible, returns None otherwise.

PARAMETERS:

- **v1** (mathutils.Vector) – Point1
- **v2** (mathutils.Vector) – Point2
- **v3** (mathutils.Vector) – Point3
- **ray** (mathutils.Vector) – Direction of the projection
- **orig** (mathutils.Vector) – Origin
- **clip** (*bool*) – When False, don't restrict the intersection to the area of the triangle, use the infinite plane defined by the triangle.

RETURNS:

The point of intersection or None if no intersection is found

RETURN TYPE:

mathutils.Vector | None

mathutils.geometry.**intersect_sphere_sphere_2d(p_a, radius_a, p_b, radius_b)**

Returns 2 points on between intersecting circles.

PARAMETERS:

- **p_a** (mathutils.Vector) – Center of the first circle
- **radius_a** (*float*) – Radius of the first circle
- **p_b** (mathutils.Vector) – Center of the second circle
- **radius_b** (*float*) – Radius of the second circle

RETURNS:

2 points on between intersecting circles or None when there is no intersection.

RETURN TYPE:

tuple[mathutils.Vector, mathutils.Vector] | tuple[None, None]

mathutils.geometry.**intersect_tri_tri_2d(tri_a1, tri_a2, tri_a3, tri_b1, tri_b2, tri_b3)**

Check if two 2D triangles intersect.

RETURN TYPE:

bool

mathutils.geometry.**normal(vectors)**

Returns the normal of a 3D polygon.

PARAMETERS:

vectors (*Sequence[Sequence[float]]*) – 3 or more vectors to calculate normals.

RETURN TYPE:

mathutils.Vector

mathutils.geometry.**points_in_planes(planes, epsilon_coplanar=1e-4, epsilon_isect=1e-6)**

Returns a list of points inside all planes given and a list of index values for the planes used.

PARAMETERS:

- **planes** (list[mathutils.Vector]) – List of planes (4D vectors).
- **epsilon_coplanar** (*float*) – Epsilon value for interpreting plane pairs as co-plannar.
- **epsilon_isect** (*float*) – Epsilon value for intersection.

RETURNS:

Two lists, once containing the 3D coordinates inside the planes, another containing the plane indices used.

RETURN TYPE:

tuple[list[mathutils.Vector], list[int]]

mathutils.geometry.**tessellate_polygon(polylines)**

Takes a list of polylines (each point a pair or triplet of numbers) and returns the point indices for a polyline filled with triangles. Does not handle degenerate geometry (such as zero-length lines due to consecutive identical points).

PARAMETERS:

polylines (*Sequence[Sequence[Sequence[float]]] :return: A list of triangles.*) – Polygons where each polygon is a sequence of 2D or 3D points.

RETURN TYPE:

list[tuple[int, int, int]]

mathutils.geometry.**volume_tetrahedron(v1, v2, v3, v4)**

Return the volume formed by a tetrahedron (points can be in any order).

PARAMETERS:

- **v1** (`mathutils.Vector`) – Point1
- **v2** (`mathutils.Vector`) – Point2
- **v3** (`mathutils.Vector`) – Point3
- **v4** (`mathutils.Vector`) – Point4

RETURN TYPE:

float