euclid Documentation

Release 0.1

Alex Holkner, Morten Lied Johansen

May 11, 2023

Contents

Abstract

A module providing vector, matrix and quaternion operations for use in 2D and 3D graphics applications.

CHAPTER 1

Introduction

This document describes the `euclid` module, which provides vector, matrix and quaternion classes for 2D and 3D graphics applications. Everything is provided in the `euclid` namespace:

```
>>> from euclid import *
>>>
```

Type checking of arguments is done with assertions. The advantage of this is that in a usual Python session an exception will be raised earlier rather than later, with a message indicating the expected type. When Python is run with the `-O` or `-OO` flags, assertions are removed and the code executes faster.

CHAPTER 2

Vector classes

Two mutable vector types are available: **Vector2** and **Vector3**, for 2D and 3D vectors, respectively. Vectors are assumed to hold floats, but most operations will also work if you use ints or longs instead. Construct a vector in the obvious way:

```
>>> Vector2(1.5, 2.0)
Vector2(1.50, 2.00)
```

```
>>> Vector3(1.0, 2.0, 3.0)
Vector3(1.00, 2.00, 3.00)
```

2.1 Element access

Components may be accessed as attributes (examples that follow use **Vector3**, but all results are similar for **Vector2**, using only the *x* and *y* components):

```
>>> v = Vector3(1, 2, 3)
>>> v.x
1
>>> v.y
2
>>> v.z
3
```

Vectors support the list interface via slicing:

```
>>> v = Vector3(1, 2, 3)
>>> len(v)
3
>>> v[0]
1
```

(continues on next page)

(continued from previous page)

```
>>> v[:]
(1, 2, 3)
```

You can also "swizzle" the components (*a la* GLSL or Cg):

```
>>> v = Vector3(1, 2, 3)
>>> v.xyz
(1, 2, 3)
>>> v.zx
(3, 1)
>>> v.zzzz
(3, 3, 3, 3)
```

All of the above accessors are also mutators[1]:

```
>>> v = Vector3(1, 2, 3)
>>> v.x = 5
>>> v
Vector3(5.00, 2.00, 3.00)
>>> v[1:] = (10, 20)
>>> v
Vector3(5.00, 10.00, 20.00)
```

[1] assignment via a swizzle (e.g., `v.xyz = (1, 2, 3)`) is supported only if the `_enable_swizzle_set` variable is set. This is disabled by default, as it impacts on the performance of ordinary attribute setting, and is slower than setting components sequentially anyway.

2.2 Operators

Addition and subtraction are supported via operator overloading (note that in-place operators perform faster than those that create a new object):

```
>>> v1 = Vector3(1, 2, 3)
>>> v2 = Vector3(4, 5, 6)
>>> v1 + v2
Vector3(5.00, 7.00, 9.00)
>>> v1 -= v2
>>> v1
Vector3(-3.00, -3.00, -3.00)
```

You can also use tuples in place of actual Vectors:

```
>>> v = Vector3(1, 1, 1)
>>> (1, 1, 1) - v
Vector3(0.00, 0.00, 0.00)
```

Multiplication and division can be performed with a scalar only:

```
>>> Vector3(1, 2, 3) * 2
Vector3(2.00, 4.00, 6.00)
>>> v1 = Vector3(1., 2., 3.)
>>> v1 /= 2
>>> v1
Vector3(0.50, 1.00, 1.50)
```

The magnitude of a vector can be found with `abs`:

```
>>> v = Vector3(1., 2., 3.)
>>> abs(v)
3.7416573867739413
```

A vector can be normalized in-place (note that the in-place method also returns `self`, so you can chain it with further operators):

```
>>> v = Vector3(1., 2., 3.)
>>> v.normalize()
Vector3(0.27, 0.53, 0.80)
>>> v
Vector3(0.27, 0.53, 0.80)
```

The following methods do *not* alter the original vector or their arguments:

`copy()` Returns a copy of the vector. `__copy__` is also implemented.

`magnitude()` Returns the magnitude of the vector; equivalent to `abs(v)`. Example:

```
>>> v = Vector3(1., 2., 3.)
>>> v.magnitude()
3.7416573867739413
```

`magnitude_squared()` Returns the sum of the squares of each component. Useful for comparing the length of two vectors without the expensive square root operation. Example:

```
>>> v = Vector3(1., 2., 3.)
>>> v.magnitude_squared()
14.0
```

`normalized()` Return a unit length vector in the same direction. Note that this method differs from `normalize` in that it does not modify the vector in-place. Example:

```
>>> v = Vector3(1., 2., 3.)
>>> v.normalized()
Vector3(0.27, 0.53, 0.80)
>>> v
Vector3(1.00, 2.00, 3.00)
```

`dot(other)` Return the scalar "dot" product of two vectors. Example:

```
>>> v1 = Vector3(1., 2., 3.)
>>> v2 = Vector3(4., 5., 6.)
>>> v1.dot(v2)
32.0
```

`determinant(other)` Return the scalar "determinant" of two 2D vectors. Example:

```
>>> v1 = Vector2(1., 2.)
>>> v2 = Vector2(3., 4.)
>>> v1.determinant(v2)
-2.0
```

`cross()` and `cross(other)` Return the cross product of a vector (for **Vector2**), or the cross product of two vectors (for **Vector3**). The return type is a vector. Example:

```
>>> v1 = Vector3(1., 2., 3.)
>>> v2 = Vector3(4., 5., 6.)
>>> v1.cross(v2)
Vector3(-3.00, 6.00, -3.00)
```

In two dimensions there can be no argument to `cross`:

```
>>> v1 = Vector2(1., 2.)
>>> v1.cross()
Vector2(2.00, -1.00)
```

`reflect(normal)` Return the vector reflected about the given normal. In two dimensions, *normal* is the normal to a line, in three dimensions it is the normal to a plane. The normal must have unit length. Example:

```
>>> v = Vector3(1., 2., 3.)
>>> v.reflect(Vector3(0, 1, 0))
Vector3(1.00, -2.00, 3.00)
>>> v = Vector2(1., 2.)
>>> v.reflect(Vector2(1, 0))
Vector2(-1.00, 2.00)
```

`rotate(theta)` For 2D vectors, return the vector rotated around origin by the angle theta. Example:

```
>>> v = Vector2(1., 2.)
>>> v.rotate(math.pi/2)
Vector2(-2.00, 1.00)
```

`rotate_around(axes, theta)` For 3D vectors, return the vector rotated around axis by the angle theta. Example:

```
>>> v = Vector3(1., 2., 3.)
>>> axes = Vector3(1.,1.,0)
>>> v.rotate_around(axes,math.pi/4)
Vector3(2.65, 0.35, 2.62)
```

`angle(other)` Return the angle between two vectors. Example:

```
>>> v1 = Vector2(1., 1.)
>>> v1.angle(Vector2(1., 0.))
-0.7853981633974483
>>> v1.angle(Vector2(-1., 0.))
2.356194490192345
```

`clockwise(other)` For 2D vectors, return true if other vector is clockwise to this. Example:

```
>>> v = Vector2(1., 1.)
>>> other = Vector2(1., 0.)
>>> v.clockwise(other)
True
>>> other.clockwise(v)
False
```

`project(other)` Return the projection (the component) of the vector on other.

Tests for equality include comparing against other sequences:

```
>>> v2 = Vector2(1, 2)
>>> v2 == Vector2(3, 4)
False
>>> v2 != Vector2(1, 2)
False
>>> v2 == (1, 2)
True
```

```
>>> v3 = Vector3(1, 2, 3)
>>> v3 == Vector3(3, 4, 5)
False
>>> v3 != Vector3(1, 2, 3)
False
>>> v3 == (1, 2, 3)
True
```

Vectors are not hashable, and hence cannot be put in sets nor used as dictionary keys:

```
>>> {Vector2(): 0}
Traceback (most recent call last):
    ...
TypeError: unhashable type: 'Vector2'
```

```
>>> {Vector3(): 0}
Traceback (most recent call last):
    ...
TypeError: unhashable type: 'Vector3'
```

CHAPTER 3

Matrix classes

Two matrix classes are supplied, **Matrix3**, a 3x3 matrix for working with 2D affine transformations, and **Matrix4**, a 4x4 matrix for working with 3D affine transformations.

The default constructor intializes the matrix to the identity:

```
>>> Matrix3()
Matrix3([    1.00      0.00      0.00
             0.00      1.00      0.00
             0.00      0.00      1.00])
>>> Matrix4()
Matrix4([    1.00      0.00      0.00      0.00
             0.00      1.00      0.00      0.00
             0.00      0.00      1.00      0.00
             0.00      0.00      0.00      1.00])
```

3.1 Element access

Internally each matrix is stored as a set of attributes named `a` to `p`. The layout for Matrix3 is:

    # a b c # e f g # i j k

and for Matrix4:

    # a b c d # e f g h # i j k l # m n o p

If you wish to set or retrieve a number of elements at once, you can do so with a slice:

```
>>> m = Matrix4()
>>> m[:]
[1.0, 0, 0, 0, 0, 1.0, 0, 0, 0, 0, 1.0, 0, 0, 0, 0, 1.0]
>>> m[12:15] = (5, 5, 5)
>>> m
Matrix4([    1.00      0.00      0.00      5.00
```

(continues on next page)

(continued from previous page)

```
            0.00      1.00      0.00      5.00
            0.00      0.00      1.00      5.00
            0.00      0.00      0.00      1.00])
```

Note that slices operate in column-major order, which makes them suitable for working directly with OpenGL's `glLoadMatrix` and `glGetFloatv` functions.

3.2 Class constructors

There are class constructors for the most common types of transform.

`new_identity` Equivalent to the default constructor. Example:

```
>>> m = Matrix4.new_identity()
>>> m
Matrix4([     1.00      0.00      0.00      0.00
              0.00      1.00      0.00      0.00
              0.00      0.00      1.00      0.00
              0.00      0.00      0.00      1.00])
```

`new_scale(x, y)` and `new_scale(x, y, z)` The former is defined on **Matrix3**, the latter on **Matrix4**. Equivalent to the OpenGL call `glScalef`. Example:

```
>>> m = Matrix4.new_scale(2.0, 3.0, 4.0)
>>> m
Matrix4([     2.00      0.00      0.00      0.00
              0.00      3.00      0.00      0.00
              0.00      0.00      4.00      0.00
              0.00      0.00      0.00      1.00])
```

`new_translate(x, y)` and `new_translate(x, y, z)` The former is defined on **Matrix3**, the latter on **Matrix4**. Equivalent to the OpenGL call `glTranslatef`. Example:

```
>>> m = Matrix4.new_translate(3.0, 4.0, 5.0)
>>> m
Matrix4([     1.00      0.00      0.00      3.00
              0.00      1.00      0.00      4.00
              0.00      0.00      1.00      5.00
              0.00      0.00      0.00      1.00])
```

`new_rotate(angle)` Create a **Matrix3** for a rotation around the origin. *angle* is specified in radians, anti-clockwise. This is not implemented in **Matrix4** (see below for equivalent methods). Example:

```
>>> import math
>>> m = Matrix3.new_rotate(math.pi / 2)
>>> m
Matrix3([     0.00     -1.00      0.00
              1.00      0.00      0.00
              0.00      0.00      1.00])
```

The following constructors are defined for **Matrix4** only.

`new` Construct a matrix with 16 values in column-major order.

`new_rotatex(angle)`, `new_rotatey(angle)`, `new_rotatez(angle)` Create a **Matrix4** for a rotation around the X, Y or Z axis, respectively. *angle* is specified in radians. Example:

```
>>> m = Matrix4.new_rotatex(math.pi / 2)
>>> m
Matrix4([    1.00     0.00     0.00     0.00
             0.00     0.00    -1.00     0.00
             0.00     1.00     0.00     0.00
             0.00     0.00     0.00     1.00])
```

new_rotate_axis(angle, axis) Create a **Matrix4** for a rotation around the given axis. *angle* is specified in radians, and *axis* must be an instance of **Vector3**. It is not necessary to normalize the axis. Example:

```
>>> m = Matrix4.new_rotate_axis(math.pi / 2, Vector3(1.0, 0.0, 0.0))
>>> m
Matrix4([    1.00     0.00     0.00     0.00
             0.00     0.00    -1.00     0.00
             0.00     1.00     0.00     0.00
             0.00     0.00     0.00     1.00])
```

new_rotate_euler(heading, attitude, bank) Create a **Matrix4** for the given Euler rotation. *heading* is a rotation around the Y axis, *attitude* around the X axis and *bank* around the Z axis. All rotations are performed simultaneously, so this method avoids "gimbal lock" and is the usual method for implemented 3D rotations in a game. Example:

```
>>> m = Matrix4.new_rotate_euler(math.pi / 2, math.pi / 2, 0.0)
>>> m
Matrix4([    0.00    -0.00     1.00     0.00
             1.00     0.00    -0.00     0.00
            -0.00     1.00     0.00     0.00
             0.00     0.00     0.00     1.00])
```

new_perspective(fov_y, aspect, near, far) Create a **Matrix4** for projection onto the 2D viewing plane. This method is equivalent to the OpenGL call `gluPerspective`. *fov_y* is the view angle in the Y direction, in radians. *aspect* is the aspect ration *width* / *height* of the viewing plane. *near* and *far* are the distance to the near and far clipping planes. They must be positive and non-zero. Example:

```
>>> m = Matrix4.new_perspective(math.pi / 2, 1024.0 / 768, 1.0, 100.0)
>>> m
Matrix4([    0.75     0.00     0.00     0.00
             0.00     1.00     0.00     0.00
             0.00     0.00    -1.02    -2.02
             0.00     0.00    -1.00     0.00])
```

3.3 Operators

Matrices of the same dimension may be multiplied to give a new matrix. For example, to create a transform which translates and scales:

```
>>> m1 = Matrix3.new_translate(5.0, 6.0)
>>> m2 = Matrix3.new_scale(1.0, 2.0)
>>> m1 * m2
Matrix3([    1.00     0.00     5.00
             0.00     2.00     6.00
             0.00     0.00     1.00])
```

Note that multiplication is not commutative (the order that you apply transforms matters):

```
>>> m2 * m1
Matrix3([    1.00     0.00     5.00
             0.00     2.00    12.00
             0.00     0.00     1.00])
```

In-place multiplication is also permitted (and optimised):

```
>>> m1 *= m2
>>> m1
Matrix3([    1.00     0.00     5.00
             0.00     2.00     6.00
             0.00     0.00     1.00])
```

Multiplying a matrix by a vector returns a vector, and is used to transform a vector:

```
>>> m1 = Matrix3.new_rotate(math.pi / 2)
>>> m1 * Vector2(1.0, 1.0)
Vector2(-1.00, 1.00)
```

Note that translations have no effect on vectors. They do affect points, however:

```
>>> m1 = Matrix3.new_translate(5.0, 6.0)
>>> m1 * Vector2(1.0, 2.0)
Vector2(1.00, 2.00)
>>> m1 * Point2(1.0, 2.0)
Point2(6.00, 8.00)
```

Multiplication is currently incorrect between matrices and vectors – the projection component is ignored. Use the **Matrix4.transform** method instead.

Matrix4 also defines **transpose** (in-place), **transposed** (functional), **determinant** and **inverse** (functional) methods.

A **Matrix3** can be multiplied with a **Vector2** or any of the 2D geometry objects (**Point2**, **Line2**, **Circle**, etc).

A **Matrix4** can be multiplied with a **Vector3** or any of the 3D geometry objects (**Point3**, **Line3**, **Sphere**, etc).

For convenience, each of the matrix constructors are also available as in-place operators. For example, instead of writing:

```
>>> m1 = Matrix3.new_translate(5.0, 6.0)
>>> m2 = Matrix3.new_scale(1.0, 2.0)
>>> m1 *= m2
```

you can apply the scale directly to *m1*:

```
>>> m1 = Matrix3.new_translate(5.0, 6.0)
>>> m1.scale(1.0, 2.0)
Matrix3([    1.00     0.00     5.00
             0.00     2.00     6.00
             0.00     0.00     1.00])
>>> m1
Matrix3([    1.00     0.00     5.00
             0.00     2.00     6.00
             0.00     0.00     1.00])
```

Note that these methods operate in-place (they modify the original matrix), and they also return themselves as a result. This allows you to chain transforms together directly:

```
>>> Matrix3().translate(1.0, 2.0).rotate(math.pi / 2).scale(4.0, 4.0)
Matrix3([    0.00    -4.00     1.00
             4.00     0.00     2.00
             0.00     0.00     1.00])
```

All constructors have an equivalent in-place method. For **Matrix3**, they are `identity`, `translate`, `scale` and `rotate`. For **Matrix4**, they are `identity`, `translate`, `scale`, `rotatex`, `rotatey`, `rotatez`, `rotate_axis` and `rotate_euler`. Both **Matrix3** and **Matrix4** also have an in-place `transpose` method.

The `copy` method is also implemented in both matrix classes and behaves in the obvious way.

CHAPTER 4

Quaternions

A quaternion represents a three-dimensional rotation or reflection transformation. They are the preferred way to store and manipulate rotations in 3D applications, as they do not suffer the same numerical degredation that matrices do.

The quaternion constructor initializes to the identity transform:

```
>>> q = Quaternion()
>>> q
Quaternion(real=1.00, imag=<0.00, 0.00, 0.00>)
```

4.1 Element access

Internally, the quaternion is stored as four attributes: `x`, `y` and `z` forming the imaginary vector, and `w` the real component.

4.2 Constructors

Rotations can be formed using the constructors:

`new_identity()` Equivalent to the default constructor.

`new_rotate_axis(angle, axis)` Equivalent to the Matrix4 constructor of the same name. *angle* is specified in radians, *axis* is an instance of **Vector3**. It is not necessary to normalize the axis. Example:

```
>>> q = Quaternion.new_rotate_axis(math.pi / 2, Vector3(1, 0, 0))
>>> q
Quaternion(real=0.71, imag=<0.71, 0.00, 0.00>)
```

`new_rotate_euler(heading, attitude, bank)` Equivalent to the Matrix4 constructor of the same name. *heading* is a rotation around the Y axis, *attitude* around the X axis and *bank* around the Z axis. All angles are given in radians. Example:

```
>>> q = Quaternion.new_rotate_euler(math.pi / 2, math.pi / 2, 0)
>>> q
Quaternion(real=0.50, imag=<0.50, 0.50, 0.50>)
```

new_interpolate(q1, q2, t) Create a quaternion which gives a (SLERP) interpolated rotation between *q1* and *q2*. *q1* and *q2* are instances of **Quaternion**, and *t* is a value between 0.0 and 1.0. For example:

```
>>> q1 = Quaternion.new_rotate_axis(math.pi / 2, Vector3(1, 0, 0))
>>> q2 = Quaternion.new_rotate_axis(math.pi / 2, Vector3(0, 1, 0))
>>> for i in range(11):
...     print Quaternion.new_interpolate(q1, q2, i / 10.0)
...
Quaternion(real=0.71, imag=<0.71, 0.00, 0.00>)
Quaternion(real=0.75, imag=<0.66, 0.09, 0.00>)
Quaternion(real=0.78, imag=<0.61, 0.17, 0.00>)
Quaternion(real=0.80, imag=<0.55, 0.25, 0.00>)
Quaternion(real=0.81, imag=<0.48, 0.33, 0.00>)
Quaternion(real=0.82, imag=<0.41, 0.41, 0.00>)
Quaternion(real=0.81, imag=<0.33, 0.48, 0.00>)
Quaternion(real=0.80, imag=<0.25, 0.55, 0.00>)
Quaternion(real=0.78, imag=<0.17, 0.61, 0.00>)
Quaternion(real=0.75, imag=<0.09, 0.66, 0.00>)
Quaternion(real=0.71, imag=<0.00, 0.71, 0.00>)
```

4.3 Operators

Quaternions may be multiplied to compound rotations. For example, to rotate 90 degrees around the X axis and then 90 degrees around the Y axis:

```
>>> q1 = Quaternion.new_rotate_axis(math.pi / 2, Vector3(1, 0, 0))
>>> q2 = Quaternion.new_rotate_axis(math.pi / 2, Vector3(0, 1, 0))
>>> q1 * q2
Quaternion(real=0.50, imag=<0.50, 0.50, 0.50>)
```

Multiplying a quaternion by a vector gives a vector, transformed appropriately:

```
>>> q = Quaternion.new_rotate_axis(math.pi / 2, Vector3(0, 1, 0))
>>> q * Vector3(1.0, 0, 0)
Vector3(0.00, 0.00, -1.00)
```

Similarly, any 3D object can be multiplied (e.g., **Point3**, **Line3**, **Sphere**, etc):

```
>>> q * Ray3(Point3(1., 1., 1.), Vector3(1., 1., 1.))
Ray3(<1.00, 1.00, -1.00> + u<1.00, 1.00, -1.00>)
```

As with the matrix classes, the constructors are also available as in-place operators. These are named `identity`, `rotate_euler` and `rotate_axis`. For example:

```
>>> q1 = Quaternion()
>>> q1.rotate_euler(math.pi / 2, math.pi / 2, 0)
Quaternion(real=0.50, imag=<0.50, 0.50, 0.50>)
>>> q1
Quaternion(real=0.50, imag=<0.50, 0.50, 0.50>)
```

Quaternions are usually unit length, but you may wish to use sized quaternions. In this case, you can find the magnitude using `abs`, `magnitude` and `magnitude_squared`, as with the vector classes. Example:

```
>>> q1 = Quaternion()
>>> abs(q1)
1.0
>>> q1.magnitude()
1.0
```

Similarly, the class implements `normalize` and `normalized` in the same way as the vectors.

The following methods do not alter the quaternion:

`conjugated()` Returns a quaternion that is the conjugate of the instance. For example:

```
>>> q1 = Quaternion.new_rotate_axis(math.pi / 2, Vector3(1, 0, 0))
>>> q1.conjugated()
Quaternion(real=0.71, imag=<-0.71, -0.00, -0.00>)
>>> q1
Quaternion(real=0.71, imag=<0.71, 0.00, 0.00>)
```

`get_angle_axis()` Returns a tuple (angle, axis), giving the angle to rotate around an axis equivalent to the quaternion. For example:

```
>>> q1 = Quaternion.new_rotate_axis(math.pi / 2, Vector3(1, 0, 0))
>>> q1.get_angle_axis()
(1.5707963267948966, Vector3(1.00, 0.00, 0.00))
```

`get_matrix()` Returns a **Matrix4** implementing the transformation of the quaternion. For example:

```
>>> q1 = Quaternion.new_rotate_axis(math.pi / 2, Vector3(1, 0, 0))
>>> q1.get_matrix()
Matrix4([    1.00     0.00     0.00     0.00
             0.00     0.00    -1.00     0.00
             0.00     1.00     0.00     0.00
             0.00     0.00     0.00     1.00])
```

CHAPTER 5

2D Geometry

The following classes are available for dealing with simple 2D geometry. The interface to each shape is similar; in particular, the `connect` and `distance` methods are defined identically for each.

For example, to find the closest point on a line to a circle:

```
>>> circ = Circle(Point2(3., 2.), 2.)
>>> line = Line2(Point2(0., 0.), Point2(-1., 1.))
>>> line.connect(circ).p1
Point2(0.50, -0.50)
```

To find the corresponding closest point on the circle to the line:

```
>>> line.connect(circ).p2
Point2(1.59, 0.59)
```

5.1 Point2

A point on a 2D plane. Construct in the obvious way:

```
>>> p = Point2(1.0, 2.0)
>>> p
Point2(1.00, 2.00)
```

Point2 subclasses **Vector2**, so all of **Vector2** operators and methods apply. In particular, subtracting two points gives a vector:

```
>>> Point2(2.0, 3.0) - Point2(1.0, 0.0)
Vector2(1.00, 3.00)
```

The following methods are also defined:

`connect(other)` Returns a **LineSegment2** which is the minimum length line segment that can connect the two shapes. *other* may be a **Point2**, **Line2**, **Ray2**, **LineSegment2** or **Circle**.

distance(other) Returns the absolute minimum distance to *other*. Internally this simply returns the length of the result of `connect`.

5.2 Line2, Ray2, LineSegment2

A **Line2** is a line on a 2D plane extending to infinity in both directions; a **Ray2** has a finite end-point and extends to infinity in a single direction; a **LineSegment2** joins two points.

All three classes support the same constructors, operators and methods, but may behave differently when calculating intersections etc.

You may construct a line, ray or line segment using any of:

- another line, ray or line segment
- two points
- a point and a vector
- a point, a vector and a length

For example:

```
>>> Line2(Point2(1.0, 1.0), Point2(2.0, 3.0))
Line2(<1.00, 1.00> + u<1.00, 2.00>)
>>> Line2(Point2(1.0, 1.0), Vector2(1.0, 2.0))
Line2(<1.00, 1.00> + u<1.00, 2.00>)
>>> Ray2(Point2(1.0, 1.0), Vector2(1.0, 2.0), 1.0)
Ray2(<1.00, 1.00> + u<0.45, 0.89>)
```

Internally, lines, rays and line segments store a Point2 *p* and a Vector2 *v*. You can also access (but not set) the two endpoints *p1* and *p2*. These may or may not be meaningful for all types of lines.

The following methods are supported by all three classes:

intersect(other) If *other* is a **Line2**, **Ray2** or **LineSegment2**, returns a **Point2** of intersection, or None if the lines are parallel.

If *other* is a **Circle**, returns a **LineSegment2** or **Point2** giving the part of the line that intersects the circle, or None if there is no intersection.

```
>>> c = Circle(Point2(0.0, 0.0), 1.0)
>>> s = LineSegment2(Point2(-4.0, 0.0), Point2(-3.0, 0.0))
>>> s.intersect(c)
```

```
>>> c = Circle(Point2(4,5), 1.0)
>>> r = Ray2(Point2(13.0, 5.0), Vector2(1.0, 0.0))
>>> r.intersect(c)
```

connect(other) Returns a **LineSegment2** which is the minimum length line segment that can connect the two shapes. For two parallel lines, this line segment may be in an arbitrary position. *other* may be a **Point2**, **Line2**, **Ray2**, **LineSegment2** or **Circle**.

distance(other) Returns the absolute minimum distance to *other*. Internally this simply returns the length of the result of `connect`.

LineSegment2 also has a *length* property which is read-only.

5.3 Circle

Circles are constructed with a center **Point2** and a radius:

```
>>> c = Circle(Point2(1.0, 1.0), 0.5)
>>> c
Circle(<1.00, 1.00>, radius=0.50)
```

Internally there are two attributes: *c*, giving the center point and *r*, giving the radius.

The following methods are supported:

`intersect(other)` If *other* is a **Line2**, **Ray2** or **LineSegment2**, returns a **LineSegment2** giving the part of the line that intersects the circle, or None if there is no intersection.

`connect(other)` Returns a **LineSegment2** which is the minimum length line segment that can connect the two shapes. *other* may be a **Point2**, **Line2**, **Ray2**, **LineSegment2** or **Circle**.

`distance(other)` Returns the absolute minimum distance to *other*. Internally this simply returns the length of the result of `connect`.

CHAPTER 6

3D Geometry

The following classes are available for dealing with simple 3D geometry. The interfaces are very similar to the 2D classes (but note that you cannot mix and match 2D and 3D operations).

For example, to find the closest point on a line to a sphere:

```
>>> sphere = Sphere(Point3(1., 2., 3.,), 2.)
>>> line = Line3(Point3(0., 0., 0.), Point3(-1., -1., 0.))
>>> line.connect(sphere).p1
Point3(1.50, 1.50, 0.00)
```

To find the corresponding closest point on the sphere to the line:

```
>>> line.connect(sphere).p2
Point3(1.32, 1.68, 1.05)
```

XXX I have not checked if these are correct.

6.1 Point3

A point on a 3D plane. Construct in the obvious way:

```
>>> p = Point3(1.0, 2.0, 3.0)
>>> p
Point3(1.00, 2.00, 3.00)
```

Point3 subclasses **Vector3**, so all of **Vector3** operators and methods apply. In particular, subtracting two points gives a vector:

```
>>> Point3(1.0, 2.0, 3.0) - Point3(1.0, 0.0, -2.0)
Vector3(0.00, 2.00, 5.00)
```

The following methods are also defined:

intersect(other) If *other* is a **Sphere**, returns `True` iff the point lies within the sphere.

connect(other) Returns a **LineSegment3** which is the minimum length line segment that can connect the two shapes. *other* may be a **Point3**, **Line3**, **Ray3**, **LineSegment3**, **Sphere** or **Plane**.

distance(other) Returns the absolute minimum distance to *other*. Internally this simply returns the length of the result of `connect`.

6.2 Line3, Ray3, LineSegment3

A **Line3** is a line on a 3D plane extending to infinity in both directions; a **Ray3** has a finite end-point and extends to infinity in a single direction; a **LineSegment3** joins two points.

All three classes support the same constructors, operators and methods, but may behave differently when calculating intersections etc.

You may construct a line, ray or line segment using any of:

- another line, ray or line segment
- two points
- a point and a vector
- a point, a vector and a length

For example:

```
>>> Line3(Point3(1.0, 1.0, 1.0), Point3(1.0, 2.0, 3.0))
Line3(<1.00, 1.00, 1.00> + u<0.00, 1.00, 2.00>)
>>> Line3(Point3(0.0, 1.0, 1.0), Vector3(1.0, 1.0, 2.0))
Line3(<0.00, 1.00, 1.00> + u<1.00, 1.00, 2.00>)
>>> Ray3(Point3(1.0, 1.0, 1.0), Vector3(1.0, 1.0, 2.0), 1.0)
Ray3(<1.00, 1.00, 1.00> + u<0.41, 0.41, 0.82>)
```

Internally, lines, rays and line segments store a Point3 *p* and a Vector3 *v*. You can also access (but not set) the two endpoints *p1* and *p2*. These may or may not be meaningful for all types of lines.

The following methods are supported by all three classes:

intersect(other) If *other* is a **Sphere**, returns a **LineSegment3** which is the intersection of the sphere and line, or `None` if there is no intersection.

If *other* is a **Plane**, returns a **Point3** of intersection, or `None`.

connect(other) Returns a **LineSegment3** which is the minimum length line segment that can connect the two shapes. For two parallel lines, this line segment may be in an arbitrary position. *other* may be a **Point3**, **Line3**, **Ray3**, **LineSegment3**, **Sphere** or **Plane**.

distance(other) Returns the absolute minimum distance to *other*. Internally this simply returns the length of the result of `connect`.

LineSegment3 also has a *length* property which is read-only.

6.3 Sphere

Spheres are constructed with a center **Point3** and a radius:

```
>>> s = Sphere(Point3(1.0, 1.0, 1.0), 0.5)
>>> s
Sphere(<1.00, 1.00, 1.00>, radius=0.50)
```

Internally there are two attributes: *c*, giving the center point and *r*, giving the radius.

The following methods are supported:

`intersect(other):` If *other* is a **Point3**, returns `True` iff the point lies within the sphere.

If *other* is a **Line3**, **Ray3** or **LineSegment3**, returns a **LineSegment3** giving the intersection, or `None` if the line does not intersect the sphere.

`connect(other)` Returns a **LineSegment3** which is the minimum length line segment that can connect the two shapes. *other* may be a **Point3**, **Line3**, **Ray3**, **LineSegment3**, **Sphere** or **Plane**.

`distance(other)` Returns the absolute minimum distance to *other*. Internally this simply returns the length of the result of `connect`.

6.4 Plane

Planes can be constructed with any of:

- three **Point3**'s lying on the plane
- a **Point3** on the plane and the **Vector3** normal
- a **Vector3** normal and *k*, described below.

Internally, planes are stored with the normal *n* and constant *k* such that *n.p* = *k* for any point on the plane *p*.

The following methods are supported:

`intersect(other)` If *other* is a **Line3**, **Ray3** or **LineSegment3**, returns a **Point3** of intersection, or `None` if there is no intersection.

If *other* is a **Plane**, returns the **Line3** of intersection.

`connect(other)` Returns a **LineSegment3** which is the minimum length line segment that can connect the two shapes. *other* may be a **Point3**, **Line3**, **Ray3**, **LineSegment3**, **Sphere** or **Plane**.

`distance(other)` Returns the absolute minimum distance to *other*. Internally this simply returns the length of the result of `connect`.