



Raysect Documentation

Release 0.3.1

Dr Alex Meakins

Mar 21, 2017

Table of Contents

1	Quick Installation	3
1.1	Introduction	3
1.2	Downloading and Installation	4
1.3	How it works	5
1.4	Quickstart Guide	6
1.5	Primitives	9
1.6	Materials	15
1.7	Conventions	15
1.8	License	16
1.9	Need Help?	16
1.10	Core Functionality	17
1.11	Examples Gallery	18
1.12	Raysect Core	18
1.13	Primitives Module	46
1.14	Optical Module	53
2	Indices and Tables	63
	Python Module Index	65



Welcome to Raysect, an OOP ray-tracing framework for Python. Raysect has been built with scientific ray-tracing in mind. Some of its features include:

- Fully spectral, high precision. Supports scientific ray-tracing of spectra from physical light sources such as plasmas.
- All core loops are written in cython for speed.
- Easily extensible, written with user customisation of materials and emissive sources in mind.
- Different observer types supported such as Pinhole cameras and optical fibres.

Quick Installation

The easiest way to install Raysect is using `pip`:

```
pip install raysect
```

For more detailed installation instructions see [Downloading and Installation](#).

Introduction

- What is raysect
 - An open-source python framework for geometrical optical simulations.
- **Where to use raysect?**
 - Why instead of povray, etc?
 - Science/engineering perspective.
 - Robustness over speed, philosophy.
 - Designed to be easy to extend by a physicist/engineer.
- **Feature set**
 - Path tracer
 - Full scenegraph for managing geometry and coordinate transforms.
 - Set of geometric primitives, lens types, meshes and CSG.
 - Simulated Physical Observers => CCDs, cameras, fibreoptics.
 - Optical materials, associated material library (BRDFs), metals, glasses.
 - multi-core.
 - geometric optics => lenses, blah.

- **Structure/Architecture**

- OOP framework written in a combination of python and cython. All major functionality is accessible from python. It is possible to extend all components from python, however to gain full speed, the cython api should be used.
- The core of raysect is actually completely generalised and can be used for other ray-tracing applications such as neutron transport, etc. However, at the present time the optical model is the only application which has been implemented.
- The core of Raysect is a generalised kernel for calculating interactions with rays and or volumes onto which physics models that require raytracing (such as geometric optics) can be built.

- **Contributions**

- Welcome, but...

Downloading and Installation

Prerequisites

The Raysect package requires Python 3.3+, numpy, scipy and matplotlib. Scipy version 0.13 or higher is recommended. Raysect has not been tested on Python 2, currently support for Python 2 is not planned. IPython is recommended for interactive use.

Installation

Raysect is available from the python package repository [pypi](#). The easiest way to install Raysect is using [pip](#):

```
pip install raysect
```

If pip is not available, the source files can be downloaded from [pypi](#) or from our [development repository](#). Once you have the source files, locate the folder containing setup.py and run:

```
python setup.py install
```

If all the required dependencies are present (cython, numpy, scipy and matplotlib), this should start the Raysect compilation and installation process.

If you would like to play with the bleeding-edge code or contribute to development, please see the Raysect development repository on [github](#).

Testing

A selection of test scripts can be run with the *nose* testing framework. These are routinely run on the development version. Running `nosetests` at the terminal in the source directory should run all of these tests to completion without errors or failures.

Many of the demos used throughout the Raysect documentation are distributed with the source code in the `demo` folder.

How it works

What is a ray tracer?

An algorithm for simulating light propagation, light is modelled as a bundle of rays that travel through a scene. The paths of the rays follow a straight line unless they interact with objects in the scene. A wide variety of optical interactions can be simulated such as reflection and refraction, scattering, and dispersion. The technique is used in computer graphics to generate photo realistic images of a 3D scene that are ideally indistinguishable from a photo of the same scene.

Some example scientific applications:

- Design of lenses and optical systems, such as cameras, microscopes and telescopes. Image-forming properties of a system to be modeled.
- Simulating optical diagnostics of plasmas through forward modelling, diagnostic design optimisation.

Ray-tracing is typically very computationally expensive and is best suited for applications that don't require real-time calculation.

Key Concepts

Rays

Represent a ray of light

- defines a line with an origin and direction
- wavelength range and number of spectral samples, centre of range used for refraction calculations

Ray objects must implement a tracing algorithm

- `spectrum = ray.trace(world)`
- causes the ray to start sampling the world
- returns a Spectrum object
- samples of spectral radiance: $\text{W/m}^2/\text{str/nm}$

Observers

Represents objects that measure light, e.g. CCDs, cameras, photo diodes, eyes. Observers launch rays and accumulate samples of the scene, which is more convenient than tracing individual rays manually.

- can be placed in the world and moved around
- `observe()` method triggers ray-tracing, i.e `camera.observe()`

Primitives

Scenes in Raysect consist of primitives, which are the basic objects making up the scene. These are objects that rays can interact with, e.g light sources, lenses, mirrors, diffuse surfaces. Types of primitives:

- Mathematically defined surfaces and solids (i.e. sphere, box, cylinder, cone).
- Constructive Solid Geometry Operators (union, intersect, subtract).
- Tri-poly meshes optimised for millions of polygons, support instancing. Importers for STL and OBJ.

Primitive surfaces and volumes can be assigned materials, e.g. glass, metal, emitter properties.

Scene-graph

Primitives and Observers are typically defined in their own local coordinate system but need to be placed into the “world”. There needs to be a system to keep track of the locations/co-ordinate transforms of all objects in the scene.

The Scene-graph is a tree structure consisting of nodes, which can be both primitives and observers. Each node has an associated coordinate space and is translated/rotated relative to its parent node. I.e. a car node may have four wheel nodes as children. Operations applied to a parent are automatically propagated down to all children. The resulting data structure describes the spatial arrangement of primitives throughout a scene.

The World is the root node of the scene-graph - all primitives and observers must be attached to World. When adding nodes to the world, nodes are always parented to another node (e.g. World) and given a transform (e.g. a translation and/or rotation) relative to their parent. Allows us to build hierarchies of objects and manipulate the whole group with one transform.

Process of raytracing

We desire the intensity/spectrum of light reaching an observer e.g. camera pixel. Our strategy is to sample light (radiance) along paths reaching observer and accumulate many samples to obtain intensity. Raysect uses a “Path tracing” algorithm where we trace a ray from the observer through all material interactions until it reaches a light source. Finally, we propagate the resulting spectrum from the light source back through all material/volume interactions.

Rays are fired backwards from the observer towards light sources since this is more computationally efficient than the other way round. The majority of light rays fired from a light source won’t reach the observer resulting in wasted computation.

During ray propagation, rays are tested for intersection with objects in the scene. Once the nearest object has been identified, the material properties of the object are inspected to determine the next step of the algorithm. Materials can alter the path of ray propagation and alter the ray’s spectra through absorption and reflection curves. Some materials will require more rays to be launched to return an accurate spectra.

Quickstart Guide

This example is based on the demo file `demos/quickstart/demo_lambert.py`. It outlines the typical workflow used in raysect.

Create Primitives

Set-up your primitives by defining materials, meshes, etc:

```
# Box defining the ground plane
ground = Box(lower=Point3D(-50, -1.51, -50), upper=Point3D(50, -1.5, 50),
↳material=Lambert())

# checker board wall that acts as emitter
emitter = Box(lower=Point3D(-10, -10, 10), upper=Point3D(10, 10, 10.1),
↳material=Checkerboard(4, d65_white, d65_white, 0.1, 2.0),
↳transform=rotate(45, 0, 0))

# Sphere
# Note that the sphere must be displaced slightly above the ground plane to prevent
↳numerically issues that could
```

```
# cause a light leak at the intersection between the sphere and the ground.
sphere = Sphere(radius=1.5, transform=translate(0, 0.0001, 0), material=schott("N-BK7
↳"))
```

Add Observer

Add an observer and configure its sampling settings. All of these camera settings have sensible defaults, The camera settings will be explained in detail in another section:

```
# processing pipeline (human vision like camera response)
rgb = RGBPipeline2D()

# camera
camera = PinholeCamera(pixels=(512, 512), fov=45, pipeline=[rgb],
↳transform=translate(0, 10, -10) * rotate(0, -45, 0))

# camera - pixel sampling settings
camera.pixel_samples = 250
camera.min_wavelength = 375.0
camera.max_wavelength = 740.0
camera.spectral_bins = 15
camera.spectral_rays = 1
```

Build Scenegraph

Assemble the scene-graph by linking primitives and observers to the World. Set their transforms:

```
world = World()

sphere.parent = world
ground.parent = world
emitter.parent = world
camera.parent = world
```

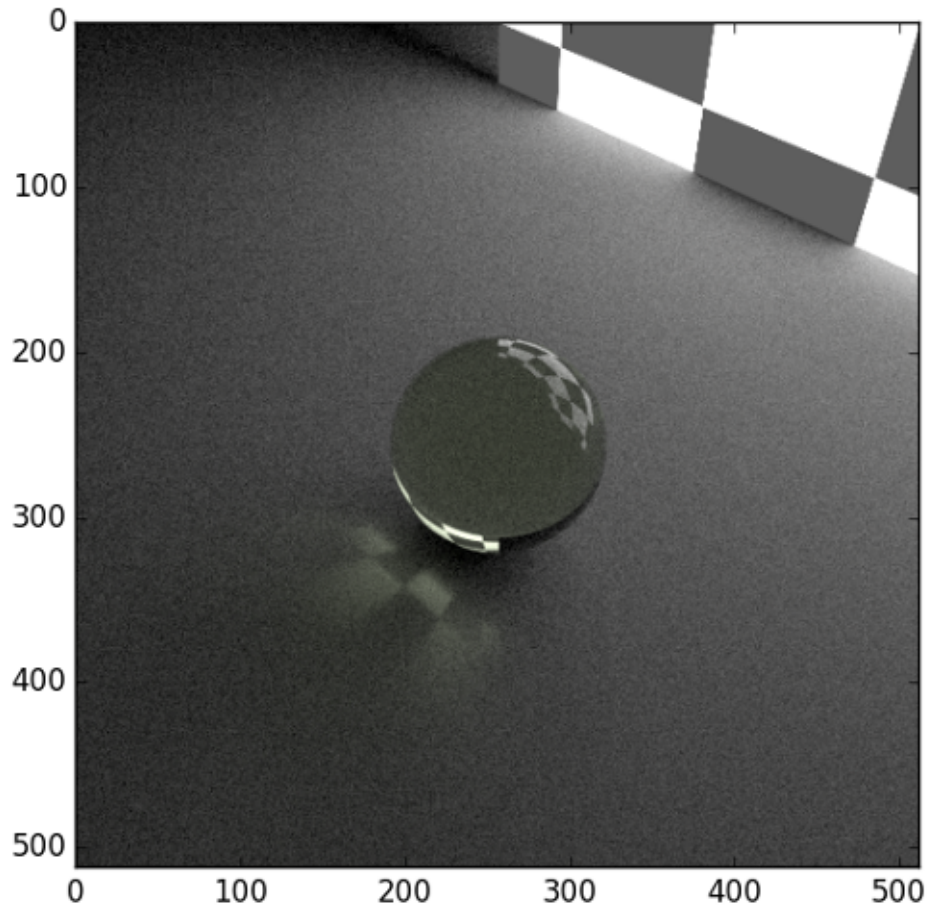
Observe()

Call observe() on an Observer or trace a ray manually:

```
plt.ion()
camera.observe()

plt.ioff()
rgb.save("render.png")
rgb.display()
```

The resulting image should render like this.



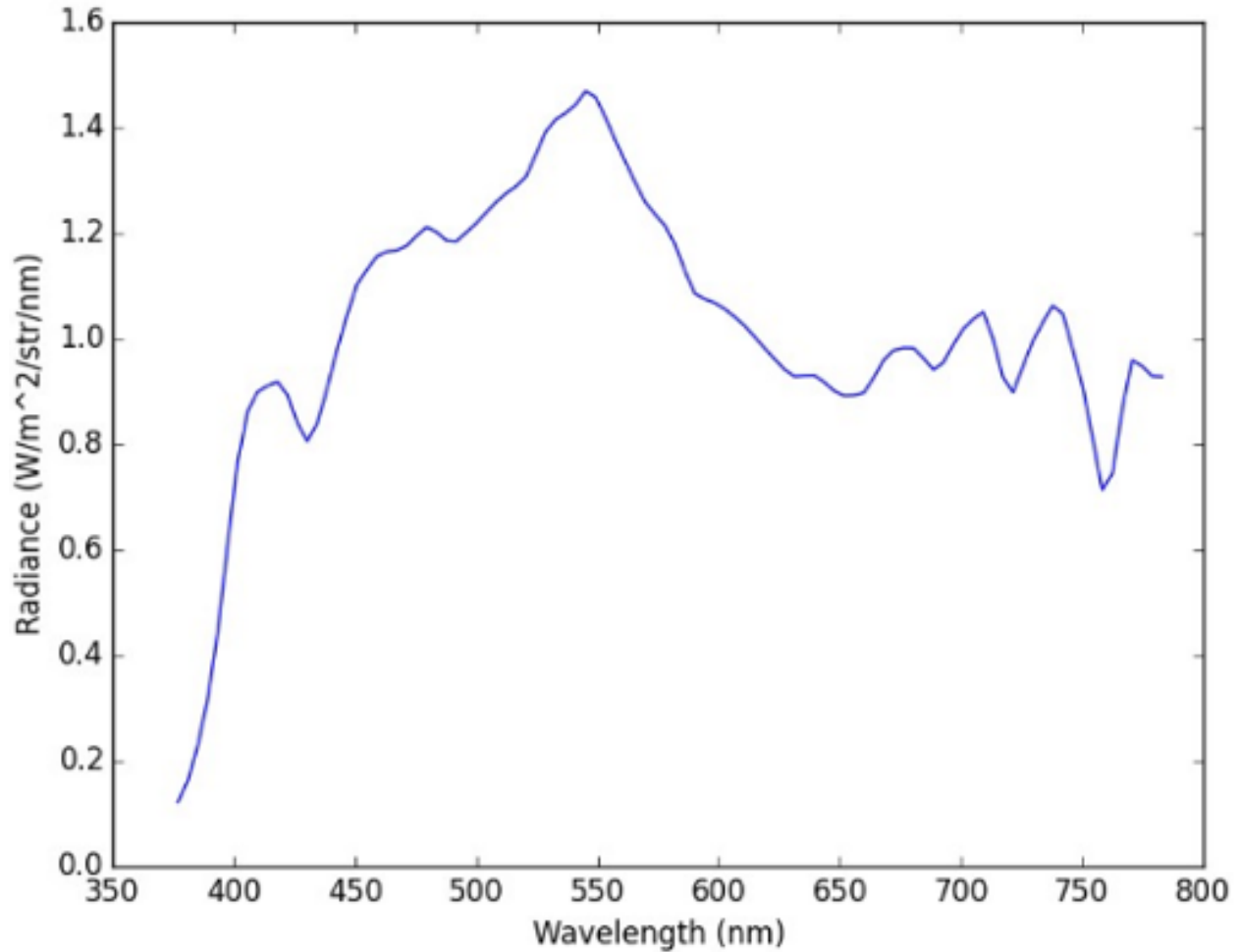
Simulated Spectrum

Lets simulate measuring a spectrum by launching a single ray:

```
ray = Ray(origin=Point3D(0, 0, -5),
          direction=Vector3D(0, 0, 1),
          min_wavelength=375,
          max_wavelength=785,
          bins=100)

ray.trace(world)
```

The resulting plot should look something like this.



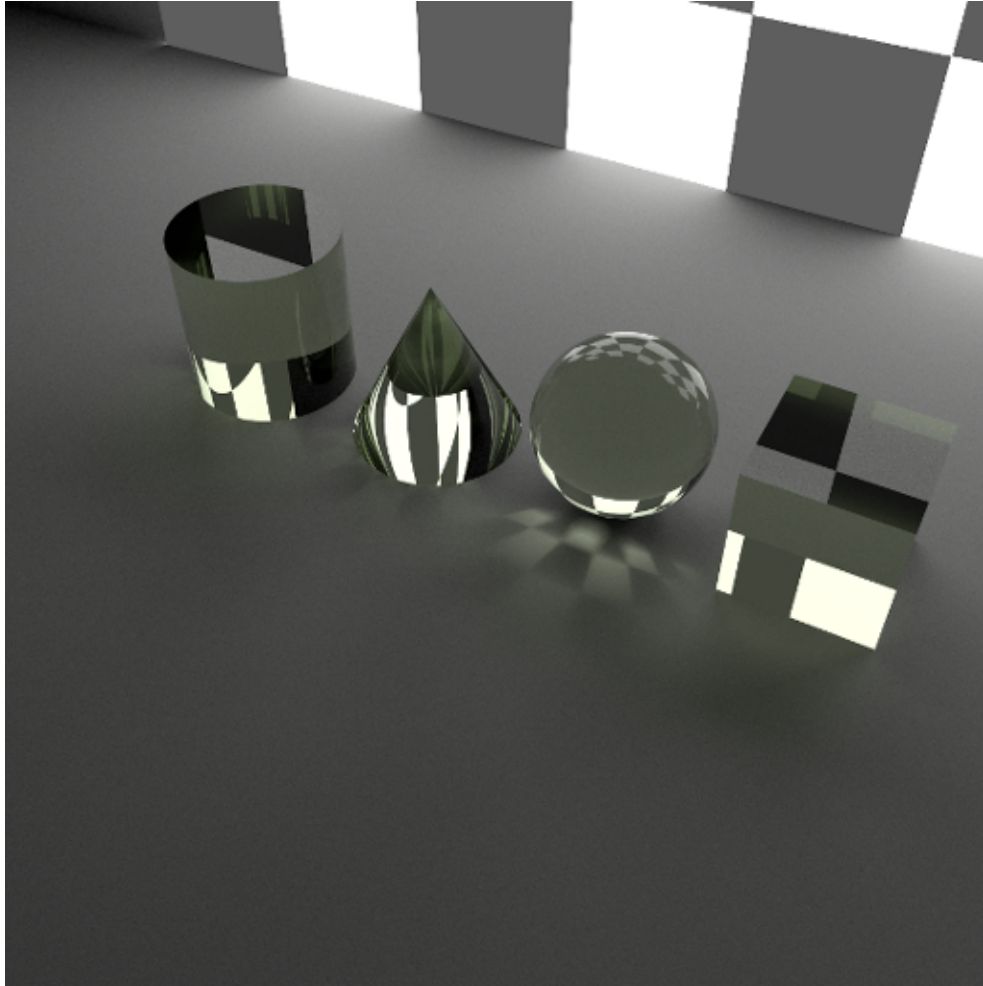
Due to the statistical nature of the path tracer, you may need to run the trace command more than once until you find a path that intersects with the light source.

You can ask the ray to trace repeatedly using the sample method instead. This will combine the results of multiple paths:

```
ray.sample(world, 10000)
```

Primitives

The raysect primitives: sphere; box; cylinder; and cone.



Geometric Primitives

Sphere

class raysect.primitive.**Sphere**

A sphere primitive.

The sphere is centered at the origin of the local co-ordinate system.

Parameters

- **radius** (*float*) – Radius of the sphere in meters (default = 0.5).
- **parent** (*Node*) – Scene-graph parent node or None (default = None).
- **transform** (*AffineMatrix3D*) – An *AffineMatrix3D* defining the local co-ordinate system relative to the scene-graph parent (default = identity matrix).
- **material** (*Material*) – A *Material* object defining the sphere's material (default = None).
- **name** (*str*) – A string specifying a user-friendly name for the sphere (default = "").

__init__

Initialize self. See help(type(self)) for accurate signature.

Box

class raysect.primitive.Box

A box primitive.

The box is defined by lower and upper points in the local co-ordinate system.

Parameters

- **lower** (*Point3D*) – Lower point of the box (default = *Point3D*(-0.5, -0.5, -0.5)).
- **upper** (*Point3D*) – Upper point of the box (default = *Point3D*(0.5, 0.5, 0.5)).
- **parent** (*Node*) – Scene-graph parent node or None (default = None).
- **transform** (*AffineMatrix3D*) – An *AffineMatrix3D* defining the local co-ordinate system relative to the scene-graph parent (default = identity matrix).
- **material** (*Material*) – A *Material* object defining the box’s material (default = None).
- **name** (*str*) – A string specifying a user-friendly name for the box (default = “”).

__init__

Initialize self. See help(type(self)) for accurate signature.

Cylinder

class raysect.primitive.Cylinder

A cylinder primitive.

The cylinder is defined by a radius and height. It lies along the z-axis and extends over the z range [0, height]. The ends of the cylinder are capped with disks forming a closed surface.

Parameters

- **radius** (*float*) – Radius of the cylinder in meters (default = 0.5).
- **height** (*float*) – Height of the cylinder in meters (default = 1.0).
- **parent** (*Node*) – Scene-graph parent node or None (default = None).
- **transform** (*AffineMatrix3D*) – An *AffineMatrix3D* defining the local co-ordinate system relative to the scene-graph parent (default = identity matrix).
- **material** (*Material*) – A *Material* object defining the cylinder’s material (default = None).
- **name** (*str*) – A string specifying a user-friendly name for the cylinder (default = “”).

__init__

Initialize self. See help(type(self)) for accurate signature.

Cone

class raysect.primitive.Cone

A cone primitive.

The cone is defined by a radius and height. It lies along the z-axis and extends over the z range [0, height]. The tip of the cone lies at z = height. The base of the cone sits on the x-y plane and is capped with a disk, forming a closed surface.

Parameters

- **radius** (*float*) – Radius of the cone in meters in x-y plane (default = 0.5).
- **height** (*float*) – Height of the cone in meters (default = 1.0).
- **parent** (*Node*) – Scene-graph parent node or None (default = None).
- **transform** (*AffineMatrix3D*) – An AffineMatrix3D defining the local co-ordinate system relative to the scene-graph parent (default = identity matrix).
- **material** (*Material*) – A Material object defining the cone’s material (default = None).
- **name** (*str*) – A string specifying a user-friendly name for the cone (default = “”).

`__init__`

Initialize self. See help(type(self)) for accurate signature.

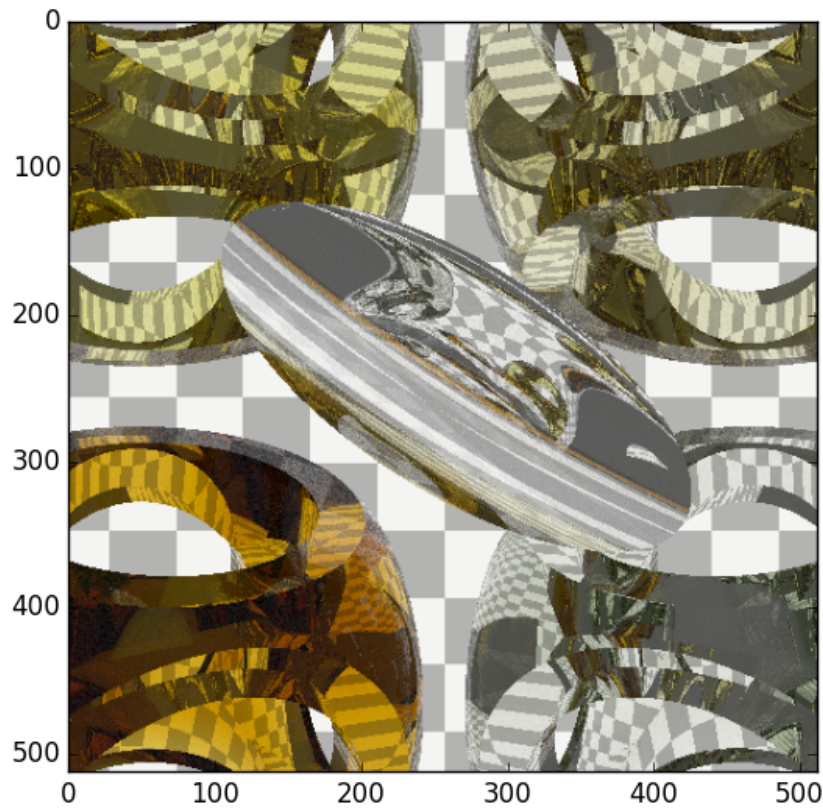
CSG Operations

Operations such as union, subtract, intersect on some basic glass primitives:

```
# Making the lense in the centre
s1 = Sphere(1.0, transform=translate(0, 0, 1.0-0.01))
s2 = Sphere(0.5, transform=translate(0, 0, -0.5+0.01))
Intersect(s1, s2, world, translate(0,0,-3.6)*rotate(50,50,0), glass)

# More complex glass structure
cyl_x = Cylinder(1, 4.2, transform=rotate(90, 0, 0)*translate(0, 0, -2.1))
cyl_y = Cylinder(1, 4.2, transform=rotate(0, 90, 0)*translate(0, 0, -2.1))
cyl_z = Cylinder(1, 4.2, transform=rotate(0, 0, 0)*translate(0, 0, -2.1))
cube = Box(Point3D(-1.5, -1.5, -1.5), Point3D(1.5, 1.5, 1.5))
sphere = Sphere(2.0)

Intersect(sphere, Subtract(cube, Union(Union(cyl_x, cyl_y), cyl_z)), world,
↪translate(-2.1,2.1,2.5)*rotate(30, -20, 0), glass)
```

Meshes

It is easiest to import meshes from existing CAD files in either obj or stl with the helper methods.

`raysect.primitive.mesh.obj.import_obj(cls, filename, scaling=1.0, **kwargs)`
 Create a mesh instance from a Wavefront OBJ mesh file (.obj).

Some engineering meshes are exported in different units (mm for example) whereas Raysect units are in m. Applying a scale factor of 0.001 would convert the mesh into m for use in Raysect.

Parameters

- **filename** (*str*) – Mesh file path.
- **scaling** (*double*) – Scale the mesh by this factor (default=1.0).
- ****kwargs** – Accepts optional keyword arguments from the Mesh class.

Return type *Mesh*

`raysect.primitive.mesh.stl.import_stl(cls, filename, scaling=1.0, mode=0, **kwargs)`
 Create a mesh instance from a STereoLithography (STL) mesh file (.stl).

Some engineering meshes are exported in different units (mm for example) whereas Raysect units are in m. Applying a scale factor of 0.001 would convert the mesh into m for use in Raysect.

Parameters

- **filename** (*str*) – Mesh file path.
- **scaling** (*double*) – Scale the mesh by this factor (default=1.0).
- ****kwargs** – Accepts optional keyword arguments from the Mesh class.

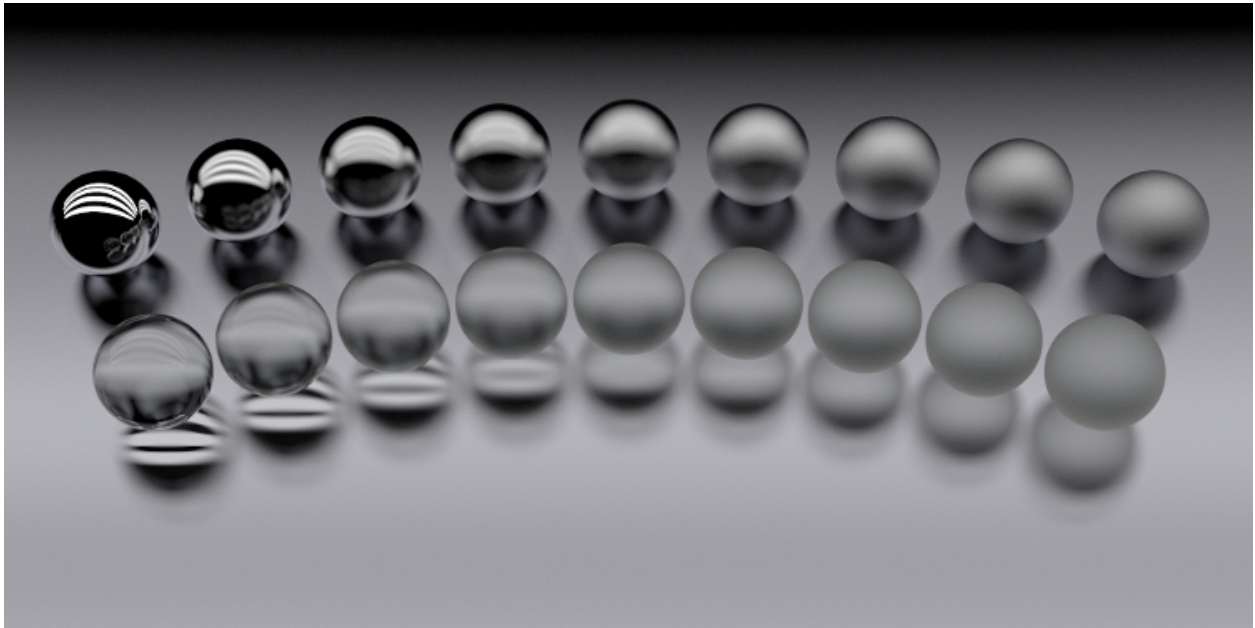
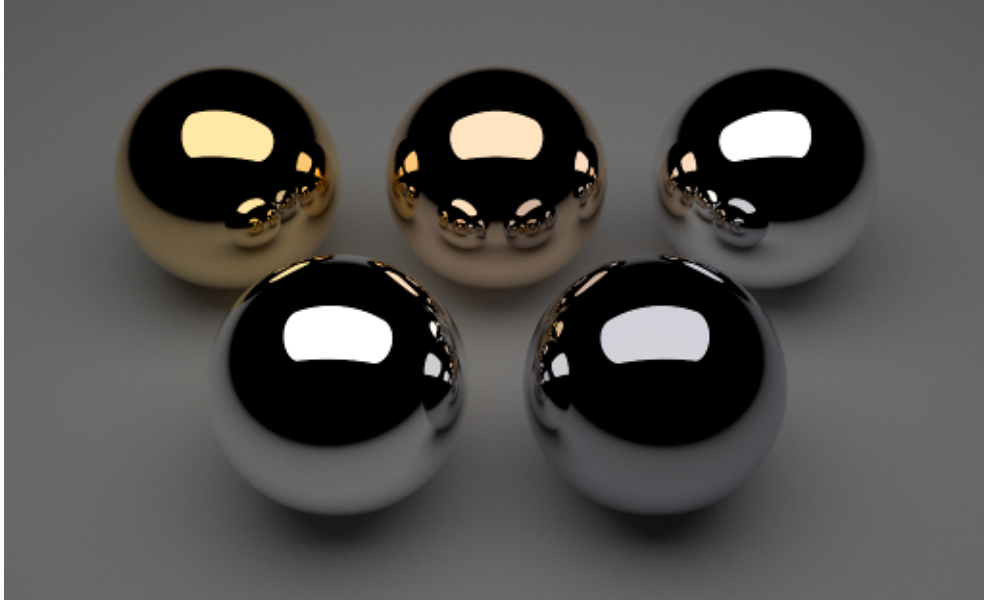
Return type *Mesh*

An example:

```
from raysect.primitive.mesh import import_obj
mesh = import_obj("./resources/stanford_bunny.obj", scaling=1, parent=world,
                  transform=translate(0, 0, 0)*rotate(165, 0, 0), material=gold)
```



Materials



Conventions

In raysect the following conventions apply:

- **the coordinate system is right-handed**
 - the x-axis (-ve, +ve) maps to (right, left)
 - the y-axis (-ve, +ve) maps to (down, up)
 - the z-axis (-ve, +ve) maps to (back, forward)

- **object orientation in local space (for general consistency)**
 - objects with a clear up and forward direction must be aligned such that their forward direction is along the +ve z-axis and their up direction is aligned to point along the +ve y direction
 - where objects have an obvious axis of rotational symmetry (e.g. a cylinder or cone) that axis should be aligned with the Z-axis
 - where objects have an obvious plane of symmetry, that plane should be aligned with the y-z plane

In raysect.optical:

- dimensions are in meters
- angles are in degrees
- solid angles are in steradians
- wavelength is in nanometers
- power is in Watts
- spectral radiance is in $\text{W/m}^2/\text{sr/nm}$

License

Copyright (c) 2014-2017, Dr Alex Meakins, Raysect Project All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. Neither the name of the Raysect Project nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

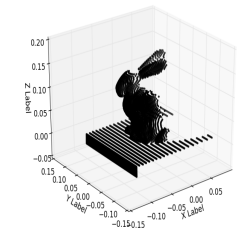
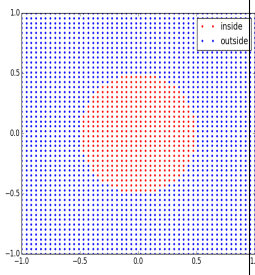
THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS “AS IS” AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Need Help?

Please post a question on the [github issue queue](#).

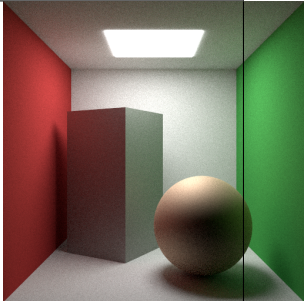
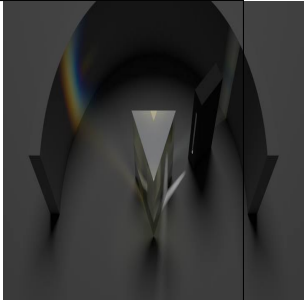
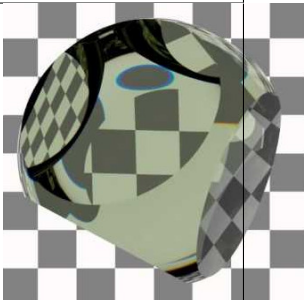
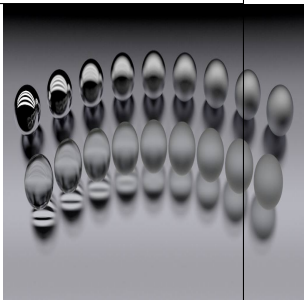
Core Functionality

Table 1.1: Core API examples

Name	Description	Preview
Ray Intersection Points	Tracking of and visualisation of where rays intersect with objects in the scene.	
Point Inside A Material	Finding all primitives which contain a test point.	

Examples Gallery

Table 1.2: Example scenes

Name	Description	Preview
Cornell Box	An industry standard test scene for benchmarking ray-tracers.	
Prism dispersion	White light is split into its component colours as it passes through a glass prism.	
Making animations	Looping over the observe loop whilst changing the position of primitives generates an animation.	
Surface roughness	Material properties can be varied from smooth to rough with a material roughness modifier.	

Raysect Core

The core module of raysect is made up of math, acceleration, and scenegraph classes.

Core Classes

class raysect.core.ray.**Ray**

Describes a line in space with an origin and direction.

Parameters

- **origin** (*Point3D*) – Point defining ray’s origin (default is *Point3D*(0, 0, 0)).
- **direction** (*Vector3D*) – Vector defining ray’s direction (default is *Vector3D*(0, 0, 1)).
- **max_distance** (*double*) – The terminating distance of the ray.

copy ()

Copy this ray to a new Ray instance.

Parameters

- **origin** (*Point3D*) – Point defining origin (default is *Point3D*(0, 0, 0)).
- **direction** (*Vector3D*) – Vector defining direction (default is *Vector3D*(0, 0, 1)).

Returns A new Ray instance.

Return type *Ray*

point_on ()

Returns the point on the ray at the specified parametric distance from the ray origin.

Positive values correspond to points forward of the ray origin, along the ray direction.

Parameters **t** (*double*) – The distance along the ray.

Returns A point at distance t along the ray direction measured from its origin.

Return type *Point3D*

class raysect.core.intersection.**Intersection**

Describes the result of a ray-primitive intersection.

The inside and outside points are launch points for rays emitted from the hit point on the surface. Rays cannot be launched from the hit point directly as they risk re-intersecting the same surface due to numerical accuracy. The inside and outside points are slightly displaced from the primitive surface at a sufficient distance to prevent re-intersection due to numerical accuracy issues. The *inside_point* is shifted backwards into the surface relative to the surface normal. The *outside_point* is equivalently shifted away from the surface in the direction of the surface normal.

Parameters

- **ray** (*Ray*) – The incident ray object (world space).
- **ray_distance** (*double*) – The distance of the intersection along the ray path.
- **primitive** (*Primitive*) – The intersected primitive object.
- **hit_point** (*Point3D*) – The point of intersection between the ray and the primitive (primitive local space).
- **inside_point** (*Point3D*) – The interior ray launch point (primitive local space).
- **outside_point** (*Point3D*) – The exterior ray launch point (primitive local space).
- **normal** (*Normal3D*) – The surface normal (primitive local space)
- **exiting** (*bool*) – True if the ray is exiting the surface, False otherwise.

- **world_to_primitive** (*AffineMatrix3D*) – A world to primitive local transform matrix.
- **primitive_to_world** (*AffineMatrix3D*) – A primitive local to world transform matrix.

class `raysect.core.boundingBox.BoundingBox2D`

Axis-aligned 2D bounding box.

Parameters

- **lower** (*Point2D*) – (optional) starting point for lower box corner
- **upper** (*Point2D*) – (optional) starting point for upper box corner

contains ()

Returns true if the given 2D point lies inside the bounding box.

Parameters **point** (*Point2D*) – A given test point.

Return type `boolean`

extend ()

Enlarge this bounding box to enclose the given point.

The resulting bounding box will be larger so as to just enclose the existing bounding box and the new point. This class instance will be edited in place to have the new bounding box dimensions.

Parameters

- **point** (*Point2D*) – the point to use for extending the bounding box.
- **padding** (*float*) – optional padding parameter, gives extra margin around the new point.

extent ()

Returns the spatial extent of this bounding box along the given dimension.

Parameters **axis** (*int*) – specifies the axis to return, {0: X axis, 1: Y axis}.

Return type `float`

largest_axis ()

Find the largest axis of this bounding box.

Returns an int specifying the longest axis, {0: X axis, 1: Y axis}.

Return type `int`

largest_extent ()

Find the largest spatial extent across all axes.

Returns distance along the largest bounding box axis.

Return type `float`

lower

The point defining the lower corner of the bounding box.

Return type *Point2D*

pad ()

Makes the bounding box larger by the specified amount of padding.

Every bounding box axis will end up larger by a factor of 2 x padding.

Parameters **padding** (*float*) – distance to use as padding margin

surface_area()

Returns the surface area of the bounding box.

Return type float

union()

Union this bounding box instance with the input bounding box.

The resulting bounding box will be larger so as to just enclose both bounding boxes. This class instance will be edited in place to have the new bounding box dimensions.

Parameters **box** (`BoundingBox2D`) – A bounding box instance to union with this bounding box instance.

upper

The point defining the upper corner of the bounding box.

Return type `Point2D`

vertices()

Get the list of vertices for this bounding box.

Returns A list of `Point2D`'s representing the corners of the bounding box.

Return type list

class raysect.core.boundingBox.BoundingBox3D

Axis-aligned bounding box.

Represents a bounding box around a primitive's surface. The points defining the lower and upper corners of the box must be specified in world space.

Axis aligned bounding box ray intersections are extremely fast to evaluate compared to intersections with more general geometry. Prior to testing a primitives `hit()` method the `hit()` method of the bounding box is called. If the bounding box is not hit, then the expensive primitive `hit()` method is avoided.

Combined with a spatial subdivision acceleration structure, the cost of ray- primitive evaluations can be heavily reduced ($O(n) \rightarrow O(\log n)$).

For optimal speed the bounding box is aligned with the world space axes. As rays are propagated in world space, co-ordinate transforms can be avoided.

Parameters

- **lower** (`Point3D`) – (optional) starting point for lower box corner
- **upper** (`Point3D`) – (optional) starting point for upper box corner

centre

The point defining the geometric centre of the bounding box.

Return type `Point3D`

contains()

Returns true if the given 3D point lies inside the bounding box.

Parameters **point** (`Point3D`) – A given test point.

Return type boolean

enclosing_sphere()

Returns the radius of a sphere guaranteed to enclose the bounding box.

The sphere is centred at the box centre. A small degree of padding is added to avoid numerical accuracy issues.

Returns Radius of sphere.

Return type float

extend()

Enlarge this bounding box to enclose the given point.

The resulting bounding box will be larger so as to just enclose the existing bounding box and the new point. This class instance will be edited in place to have the new bounding box dimensions.

Parameters

- **point** (*Point3D*) – the point to use for extending the bounding box.
- **padding** (*float*) – optional padding parameter, gives extra margin around the new point.

extent()

Returns the spatial extend of this bounding box along the given dimension.

Parameters **axis** (*int*) – specifies the axis to return, {0: X axis, 1: Y axis, 2: Z axis}.

Return type float

full_intersection()

Returns full intersection information for an intersection between a ray and a bounding box.

The first value is a boolean which is true if an intersection has occurred, false otherwise. Each intersection with a bounding box will produce two intersections, one on the front and back of the box. The remaining two tuple parameters are floats representing the distance along the ray path to the respective intersections.

Parameters **ray** – The ray to test for intersection

Returns A tuple of intersection parameters, (hit, front_intersection, back_intersection).

Return type tuple

hit()

Returns true if the ray hits the bounding box.

Parameters **ray** (*Ray*) – The ray to test for intersection.

Return type boolean

largest_axis()

Find the largest axis of this bounding box.

Returns an int specifying the longest axis, {0: X axis, 1: Y axis, 2: Z axis}.

Return type int

largest_extent()

Find the largest spatial extent across all axes.

Returns distance along the largest bounding box axis.

Return type float

lower

The point defining the lower corner of the bounding box.

Return type *Point3D*

pad()

Makes the bounding box larger by the specified amount of padding.

Every bounding box axis will end up larger by a factor of 2 x padding.

Parameters `padding` (*float*) – distance to use as padding margin

surface_area ()

Returns the surface area of the bounding box.

Return type *float*

union ()

Union this bounding box instance with the input bounding box.

The resulting bounding box will be larger so as to just enclose both bounding boxes. This class instance will be edited in place to have the new bounding box dimensions.

Parameters `box` (*BoundingBox3D*) – A bounding box instance to union with this bounding box instance.

upper

The point defining the upper corner of the bounding box.

Return type *Point3D*

vertices ()

Get the list of vertices for this bounding box.

Returns A list of *Point3D*'s representing the corners of the bounding box.

Return type *list*

volume ()

Returns the volume of the bounding box.

Return type *float*

Math Module

Points and Vectors

class `raysect.core.math.point.Point2D`

Represents a point in 2D affine space.

A 2D point is a location in 2D space which is defined by its x and y coordinates in a given coordinate system. *Vector2D* objects can be added/subtracted from *Point2D* yielding another *Vector2D*. You can also find the *Vector2D* and distance between two *Point2D*s, and transform a *Point2D* from one coordinate system to another.

If no initial values are passed, *Point2D* defaults to the origin: *Point2D*(0.0, 0.0)

Parameters

- `x` (*float*) – initial x coordinate, defaults to `x = 0.0`.
- `y` (*float*) – initial y coordinate, defaults to `y = 0.0`.

Variables

- `x` (*float*) – x-coordinate
- `y` (*float*) – y-coordinate

__add__

Addition operator.

```
>>> Point2D(1, 0) + Vector2D(0, 1)
Point2D(1.0, 1.0)
```

__getitem__

Returns the point coordinates by index ([0,1] -> [x,y]).

```
>>> a = Point2D(1, 0)
>>> a[0]
1
```

__getstate__()

Encodes state for pickling.

__iter__

Iterates over the coordinates (x, y)

__setitem__

Sets the point coordinates by index ([0,1] -> [x,y]).

```
>>> a = Point2D(1, 0)
>>> a[1] = 2
>>> a
Point2D(1.0, 2.0)
```

__setstate__()

Decodes state for pickling.

__sub__

Subtraction operator.

```
>>> Point2D(1, 0) - Vector2D(0, 1)
Point2D(1.0, -1.0)
```

copy()

Returns a copy of the point.

Return type *Point2D*

distance_to()

Returns the distance between this point and the passed point.

Parameters **p** (*Point2D*) – the point to which the distance will be calculated

Return type float

vector_to()

Returns a vector from this point to the passed point.

Parameters **p** (*Point2D*) – point to which a vector will be calculated

Return type *Vector2D*

class raysect.core.math.point.Point3D

Represents a point in 3D affine space.

A point is a location in 3D space which is defined by its x, y and z coordinates in a given coordinate system. Vectors can be added/subtracted from Points yielding another Vector3D. You can also find the Vector3D and distance between two Points, and transform a Point3D from one coordinate system to another.

If no initial values are passed, Point3D defaults to the origin: Point3D(0.0, 0.0, 0.0)

Parameters

- **x** (*float*) – initial x coordinate, defaults to x = 0.0.
- **y** (*float*) – initial y coordinate, defaults to y = 0.0.

- **z** (*float*) – initial z coordinate, defaults to $z = 0.0$.

Variables

- **x** (*float*) – x-coordinate
- **y** (*float*) – y-coordinate
- **z** (*float*) – z-coordinate

`__add__`

Addition operator.

```
>>> Point3D(1, 0, 0) + Vector3D(0, 1, 0)
Point3D(1.0, 1.0, 0.0)
```

`__getitem__`

Returns the point coordinates by index ([0,1,2] -> [x,y,z]).

```
>>> a = Point3D(1, 0, 0)
>>> a[0]
1
```

`__getstate__()`

Encodes state for pickling.

`__iter__`

Iterates over the coordinates (x, y, z)

`__mul__`

Multiplication operator.

Parameters

- **x** (*AffineMatrix3D*) – transformation matrix x
- **y** (*Point3D*) – point to transform

Returns Matrix multiplication of a 3D transformation matrix with the input point.

Return type *Point3D*

`__setitem__`

Sets the point coordinates by index ([0,1,2] -> [x,y,z]).

```
>>> a = Point3D(1, 0, 0)
>>> a[1] = 2
>>> a
Point3D(1.0, 2.0, 0.0)
```

`__setstate__()`

Decodes state for pickling.

`__sub__`

Subtraction operator.

```
>>> Point3D(1, 0, 0) - Vector3D(0, 1, 0)
Point3D(1.0, -1.0, 0.0)
```

`copy()`

Returns a copy of the point.

Return type *Point3D*

distance_to()

Returns the distance between this point and the passed point.

Parameters **p** (*Point3D*) – the point to which the distance will be calculated

Return type float

transform()

Transforms the point with the supplied Affine Matrix.

The point is transformed by premultiplying the point by the affine matrix.

For cython code this method is substantially faster than using the multiplication operator of the affine matrix.

This method expects a valid affine transform. For speed reasons, minimal checks are performed on the matrix.

Parameters **m** (*AffineMatrix3D*) – The affine matrix describing the required coordinate transformation.

Returns A new instance of this point that has been transformed with the supplied Affine Matrix.

Return type *Point3D*

vector_to()

Returns a vector from this point to the passed point.

Parameters **p** (*Point3D*) – the point to which a vector will be calculated.

Return type *Vector3D*

class raysect.core.math.vector.Vector2D

Represents a vector in 2D space.

2D vectors are described by their (x, y) coordinates. Standard Vector2D operations are supported such as addition, subtraction, scaling, dot product, cross product and normalisation.

If no initial values are passed, Vector2D defaults to a unit vector aligned with the x-axis: Vector2D(1.0, 0.0)

Parameters

- **x** (*float*) – initial x coordinate, defaults to x = 0.0.
- **y** (*float*) – initial y coordinate, defaults to y = 0.0.

Variables

- **x** (*float*) – x-coordinate
- **y** (*float*) – y-coordinate

copy()

Returns a copy of the vector.

Return type *Vector2D*

cross()

Calculates the 2D cross product analogue between this vector and the supplied vector

$C = A.cross(B) \iff C = A \times B \iff \det(A, B) = A.x \ B.y - A.y \ B.x$

Note that for 2D vectors, the cross product is the equivalent of the determinant of a 2x2 matrix. The result is a scalar.

Parameters **v** (*Vector2D*) – An input vector with which to calculate the cross product.

Return type float

dot ()

Calculates the dot product between this vector and the supplied vector.

Returns a scalar.

length

The vector's length.

Raises a `ZeroDivisionError` if an attempt is made to change the length of a zero length vector. The direction of a zero length vector is undefined hence it can not be lengthened.

normalise ()

Returns a normalised copy of the vector.

The returned vector is normalised to length 1.0 - a unit vector.

Return type *Vector2D*

orthogonal ()

Returns a unit vector that is guaranteed to be orthogonal to the vector.

Return type `vector2D`

class `raysect.core.math.vector.Vector3D`

Represents a vector in 3D affine space.

Vectors are described by their (x, y, z) coordinates in the chosen coordinate system. Standard Vector3D operations are supported such as addition, subtraction, scaling, dot product, cross product, normalisation and coordinate transformations.

If no initial values are passed, Vector3D defaults to a unit vector aligned with the z-axis: `Vector3D(0.0, 0.0, 1.0)`

Parameters

- **x** (*float*) – initial x coordinate, defaults to x = 0.0.
- **y** (*float*) – initial y coordinate, defaults to y = 0.0.
- **z** (*float*) – initial z coordinate, defaults to z = 0.0.

Variables

- **x** (*float*) – x-coordinate
- **y** (*float*) – y-coordinate
- **z** (*float*) – z-coordinate

__add__

Addition operator.

```
>>> Vector3D(1, 0, 0) + Vector3D(0, 1, 0)
Vector3D(1.0, 1.0, 0.0)
```

__getitem__

Returns the vector coordinates by index ([0,1,2] -> [x,y,z]).

```
>>> a = Vector3D(1, 0, 0)
>>> a[0]
1
```

__getstate__ ()

Encodes state for pickling.

__iter__

Iterates over the vector coordinates (x, y, z)

__mul__

Multiplication operator.

3D vectors can be multiplied with both scalars and transformation matrices.

```
>>> 2 * Vector3D(1, 2, 3)
Vector3D(2.0, 4.0, 6.0)
>>> rotate_x(90) * Vector3D(0, 0, 1)
Vector3D(0.0, -1.0, 0.0)
```

__neg__

Returns a vector with the reverse orientation (negation operator).

__setitem__

Sets the vector coordinates by index ([0,1,2] -> [x,y,z]).

```
>>> a = Vector3D(1, 0, 0)
>>> a[1] = 2
>>> a
Vector3D(1.0, 2.0, 0.0)
```

__setstate__()

Decodes state for pickling.

__sub__

Subtraction operator.

```
>>> Vector3D(1, 0, 0) - Vector3D(0, 1, 0)
Vector3D(1.0, -1.0, 0.0)
```

__truediv__

Division operator.

```
>>> Vector3D(1, 1, 1) / 2
Vector3D(0.5, 0.5, 0.5)
```

copy()

Returns a copy of the vector.

Return type *Vector3D*

cross()

Calculates the cross product between this vector and the supplied vector

$$C = A.cross(B) \Leftrightarrow \vec{C} = \vec{A} \times \vec{B}$$

Parameters **v** (*Vector3D*) – An input vector with which to calculate the cross product.

Return type *Vector3D*

dot()

Calculates the dot product between this vector and the supplied vector.

Returns a scalar.

length

The vector's length.

Raises a `ZeroDivisionError` if an attempt is made to change the length of a zero length vector. The direction of a zero length vector is undefined hence it can not be lengthened.

normalise()

Returns a normalised copy of the vector.

The returned vector is normalised to length 1.0 - a unit vector.

Return type *Vector3D*

orthogonal()

Returns a unit vector that is guaranteed to be orthogonal to the vector.

Return type `vector3D`

transform()

Transforms the vector with the supplied `AffineMatrix3D`.

The vector is transformed by pre-multiplying the vector by the affine matrix.

$$\vec{C} = \mathbf{A} \times \vec{B}$$

This method is substantially faster than using the multiplication operator of `AffineMatrix3D` when called from cython code.

Parameters `m` (`AffineMatrix3D`) – The affine matrix describing the required coordinate transformation.

Returns A new instance of this vector that has been transformed with the supplied Affine Matrix.

Return type *Vector3D*

Affine Matrices

class `raysect.core.math.affinematrix.AffineMatrix3D`

A 4x4 affine matrix.

These matrices are used for transforming between coordinate systems. Every primitive in Raysect works in its own local coordinate system, so it is common to need to transform 3D points from local to world space and vice versa. Even though the vectors themselves are 3D, a 4x4 matrix is needed to completely specify a transformation from one 3D space to another.

The coordinate transformation is applied by multiplying the column vector for the desired `Point3D/Vector3D` against the transformation matrix. For example, if the original vector \vec{V}_a is in space A and the transformation matrix \mathbf{T}_{AB} describes the position and orientation of Space A relative to Space B, then the multiplication

$$\vec{V}_b = \mathbf{T}_{AB} \times \vec{V}_a$$

yields the same vector transformed into coordinate Space B, \vec{V}_b .

The individual terms of the transformation matrix can be visualised in terms of the way they change the underlying basis vectors.

$$\mathbf{T}_{AB} = \begin{pmatrix} \vec{x}_b.x & \vec{y}_b.x & \vec{z}_b.x & \vec{t}.x \\ \vec{x}_b.y & \vec{y}_b.y & \vec{z}_b.y & \vec{t}.y \\ \vec{x}_b.z & \vec{y}_b.z & \vec{z}_b.z & \vec{t}.z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Here the unit x-axis vector in space A, $\vec{x}_a = (1, 0, 0)$, has been transformed into space B, \vec{x}_b . The same applies to \vec{y}_b and \vec{z}_b for the \vec{y}_a and \vec{z}_a unit vectors respectively. Together the new basis vectors describe a rotation of the original coordinate system.

The vector \vec{t} in the last column corresponds to a translation vector between the origin's of space A and space B. Strictly speaking, the new rotation vectors don't have to be normalised which corresponds to a scaling in addition to the rotation. For example, a scaling matrix would look like the following.

$$\mathbf{T}_{\text{scale}} = \begin{pmatrix} \vec{s}.x & 0 & 0 & 0 \\ 0 & \vec{s}.y & 0 & 0 \\ 0 & 0 & \vec{s}.z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Multiple transformations can be chained together by multiplying the matrices together, the resulting matrix will encode the full transformation. The order in which transformations are applied is very important. The operation $\mathbf{M}_{\text{translate}} \times \mathbf{M}_{\text{rotate}}$ is different to $\mathbf{M}_{\text{rotate}} \times \mathbf{M}_{\text{translate}}$ because matrices don't commute, and physically these are different operations.

Warning: Because we are using column vectors, transformations should be applied **right to left**.

An an example operation, let us consider the case of moving and rotating a camera in our scene. Suppose we want to rotate our camera at an angle of $\theta_x = 45$ around the x-axis and translate the camera to position $p = (0, 0, 3.5)$. This set of operations would be equivalent to:

$$\mathbf{T} = \mathbf{T}_{\text{translate}} \times \mathbf{T}_{\text{rotate}}$$

In code this would be equivalent to:

```
>>> transform = translate(0, 0, -3.5) * rotate_x(45)
```

If no initial values are passed to the matrix, it defaults to an identity matrix.

Parameters *m* (*object*) – Any 4 x 4 indexable or 16 element object can be used to initialise the matrix. 16 element objects must be specified in row-major format.

inverse()

Calculates the inverse of the affine matrix.

Returns an AffineMatrix3D containing the inverse.

Raises a ValueError if the matrix is singular and the inverse can not be calculated. All valid affine transforms should be invertible.

`raysect.core.math.transform.translate()`

Returns an affine matrix representing a translation of the coordinate space.

Equivalent to the transform matrix, \mathbf{T}_{AB} , where \vec{t} is the vector from the origin of space A to space B.

$$\mathbf{T}_{AB} = \begin{pmatrix} 1 & 0 & 0 & \vec{t}.x \\ 0 & 1 & 0 & \vec{t}.y \\ 0 & 0 & 1 & \vec{t}.z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Parameters

- **x** (*float*) – x-coordinate
- **y** (*float*) – y-coordinate
- **z** (*float*) – z-coordinate

Return type *AffineMatrix3D*

`raysect.core.math.transform.rotate_x()`

Returns an affine matrix representing the rotation of the coordinate space about the X axis by the supplied angle.

The rotation direction is clockwise when looking along the x-axis.

$$\mathbf{T}_{AB} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos \theta & -\sin \theta & 0 \\ 0 & \sin \theta & \cos \theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Parameters `angle` (*float*) – The angle θ specified in degrees.

Return type *AffineMatrix3D*

`raysect.core.math.transform.rotate_y()`

Returns an affine matrix representing the rotation of the coordinate space about the Y axis by the supplied angle.

The rotation direction is clockwise when looking along the y-axis.

$$\mathbf{T}_{AB} = \begin{pmatrix} \cos \theta & 0 & \sin \theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin \theta & 0 & \cos \theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Parameters `angle` (*float*) – The angle θ specified in degrees.

Return type *AffineMatrix3D*

`raysect.core.math.transform.rotate_z()`

Returns an affine matrix representing the rotation of the coordinate space about the Z axis by the supplied angle.

The rotation direction is clockwise when looking along the z-axis.

$$\mathbf{T}_{AB} = \begin{pmatrix} \cos \theta & -\sin \theta & 0 & 0 \\ \sin \theta & \cos \theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Parameters `angle` (*float*) – The angle θ specified in degrees.

Return type *AffineMatrix3D*

`raysect.core.math.transform.rotate()`

Returns an affine transform matrix representing an intrinsic rotation with an axis order (-Y)(-X)'Z'.

For an object aligned such that forward is the +ve Z-axis, left is the +ve X-axis and up is the +ve Y-axis then this rotation operation corresponds to the yaw, pitch and roll of the object.

Parameters

- `yaw` (*float*) – Yaw angle in degrees.
- `pitch` (*float*) – Pitch angle in degrees.
- `roll` (*float*) – Roll angle in degrees.

Return type *AffineMatrix3D*

`raysect.core.math.transform.rotate_vector()`

Returns an affine matrix representing the rotation of the coordinate space about the supplied vector by the specified angle.

Parameters

- `angle` (*float*) – The angle specified in degrees.

- **v** (`Vector3D`) – The vector about which to rotate.

Return type `AffineMatrix3D`

`raysect.core.math.transform.rotate_basis()`

Returns a rotation matrix defined by forward and up vectors.

The +ve Z-axis of the resulting coordinate space will be aligned with the forward vector. The +ve Y-axis will be aligned to lie in the plane defined the forward and up vectors, along the projection of the up vector that lies orthogonal to the forward vector. The X-axis will lie perpendicular to the plane.

The forward and upwards vectors need not be orthogonal. The up vector will be rotated in the plane defined by the two vectors until it is orthogonal.

Parameters

- **forward** (`Vector3D`) – A `Vector3D` object defining the forward direction.
- **up** (`Vector3D`) – A `Vector3D` object defining the up direction.

Return type `AffineMatrix3D`

Functions and Interpolators

class `raysect.core.math.function.function1d.Function1D`

Cython optimised class for representing an arbitrary 1D function.

Using `__call__()` in cython is slow. This class provides an overloadable cython cdef `evaluate()` method which has much less overhead than a python function call.

For use in cython code only, this class cannot be extended via python.

To create a new function object, inherit this class and implement the `evaluate()` method. The new function object can then be used with any code that accepts a function object.

`__call__`

Evaluate the function `f(x)`

Parameters **x** (`float`) – function parameter `x`

Return type `float`

class `raysect.core.math.function.function1d.PythonFunction1D`

Bases: `raysect.core.math.function.function1d.Function1D`

Wraps a python callable object with a `Function1D` object.

This class allows a python object to interact with cython code that requires a `Function1D` object. The python object must implement `__call__()` expecting one argument.

This class is intended to be used to transparently wrap python objects that are passed via constructors or methods into cython optimised code. It is not intended that the users should need to directly interact with these wrapping objects. Constructors and methods expecting a `Function1D` object should be designed to accept a generic python object and then test that object to determine if it is an instance of `Function1D`. If the object is not a `Function1D` object it should be wrapped using this class for internal use.

See also: `autowrap_function1d()`

Parameters **function** (`object`) – the python function to wrap, `__call__()` function must be implemented on the object.

class `raysect.core.math.function.function2d.Function2D`

Cython optimised class for representing an arbitrary 2D function.

Using `__call__()` in cython is slow. This class provides an overloadable cython cdef `evaluate()` method which has much less overhead than a python function call.

For use in cython code only, this class cannot be extended via python.

To create a new function object, inherit this class and implement the `evaluate()` method. The new function object can then be used with any code that accepts a function object.

`__call__`

Evaluate the function `f(x, y)`

Parameters

- `x (float)` – function parameter `x`
- `y (float)` – function parameter `y`

Return type `float`

class `raysect.core.math.function.function2d.PythonFunction2D`

Bases: `raysect.core.math.function.function2d.Function2D`

Wraps a python callable object with a `Function2D` object.

This class allows a python object to interact with cython code that requires a `Function2D` object. The python object must implement `__call__()` expecting two arguments.

This class is intended to be used to transparently wrap python objects that are passed via constructors or methods into cython optimised code. It is not intended that the users should need to directly interact with these wrapping objects. Constructors and methods expecting a `Function2D` object should be designed to accept a generic python object and then test that object to determine if it is an instance of `Function2D`. If the object is not a `Function2D` object it should be wrapped using this class for internal use.

See also: `autowrap_function2d()`

Parameters `function (object)` – the python function to wrap, `__call__()` function must

be implemented on the object.

class `raysect.core.math.function.function3d.Function3D`

Cython optimised class for representing an arbitrary 3D function.

Using `__call__()` in cython is slow. This class provides an overloadable cython cdef `evaluate()` method which has much less overhead than a python function call.

For use in cython code only, this class cannot be extended via python.

To create a new function object, inherit this class and implement the `evaluate()` method. The new function object can then be used with any code that accepts a function object.

`__call__`

Evaluate the function `f(x, y, z)`

Parameters

- `x (float)` – function parameter `x`
- `y (float)` – function parameter `y`
- `y` – function parameter `z`

Return type `float`

class `raysect.core.math.function.function3d.PythonFunction3D`

Bases: `raysect.core.math.function.function3d.Function3D`

Wraps a python callable object with a Function3D object.

This class allows a python object to interact with cython code that requires a Function3D object. The python object must implement `__call__()` expecting three arguments.

This class is intended to be used to transparently wrap python objects that are passed via constructors or methods into cython optimised code. It is not intended that the users should need to directly interact with these wrapping objects. Constructors and methods expecting a Function3D object should be designed to accept a generic python object and then test that object to determine if it is an instance of Function3D. If the object is not a Function3D object it should be wrapped using this class for internal use.

See also: `autowrap_function3d()`

class `raysect.core.math.interpolators.discrete2dmesh.Discrete2DMesh`

Bases: `raysect.core.math.function.function2d.Function2D`

Discrete interpolator for data on a 2d ungridded tri-poly mesh.

The mesh is specified as a set of 2D vertices supplied as an Nx2 numpy array or a suitably sized sequence that can be converted to a numpy array.

The mesh triangles are defined with a Mx3 array where the three values are indices into the vertex array that specify the triangle vertices. The mesh must not contain overlapping triangles. Supplying a mesh with overlapping triangles will result in undefined behaviour.

A data array of length M, containing a value for each triangle, holds the data to be interpolated across the mesh.

By default, requesting a point outside the bounds of the mesh will cause a `ValueError` exception to be raised. If this is not desired the `limit` attribute (default `True`) can be set to `False`. When set to `False`, a default value will be returned for any point lying outside the mesh. The value return can be specified by setting the `default_value` attribute (default is 0.0).

To optimise the lookup of triangles, the interpolator builds an acceleration structure (a KD-Tree) from the specified mesh data. Depending on the size of the mesh, this can be quite slow to construct. If the user wishes to interpolate a number of different data sets across the same mesh - for example: temperature and density data that are both defined on the same mesh - then the user can use the `instance()` method on an existing interpolator to create a new interpolator. The new interpolator will share a copy of the internal acceleration data. The `triangle_data`, `limit` and `default_value` can be customised for the new instance. See `instance()`. This will avoid the cost in memory and time of rebuilding an identical acceleration structure.

Parameters

- **vertex_coords** (*ndarray*) – An array of vertex coordinates (x, y) with shape Nx2.
- **triangles** (*ndarray*) – An array of vertex indices defining the mesh triangles, with shape Mx3.
- **triangle_data** (*ndarray*) – An array containing data for each triangle of shape Mx1.
- **limit** (*bool*) – Raise an exception outside mesh limits - `True` (default) or `False`.
- **default_value** (*float*) – The value to return outside the mesh limits if `limit` is set to `False`.

instance()

Creates a new interpolator instance from an existing interpolator instance.

The new interpolator instance will share the same internal acceleration data as the original interpolator. The `triangle_data`, `limit` and `default_value` settings of the new instance can be redefined by setting the

appropriate attributes. If any of the attributes are set to None (default) then the value from the original interpolator will be copied.

This method should be used if the user has multiple sets of triangle_data that lie on the same mesh geometry. Using this methods avoids the repeated rebuilding of the mesh acceleration structures by sharing the geometry data between multiple interpolator objects.

Parameters

- **instance** (*Discrete2DMesh*) – Discrete2DMesh object.
- **triangle_data** (*ndarray*) – An array containing data for each triangle of shape Mx1 (default None).
- **limit** (*bool*) – Raise an exception outside mesh limits - True (default) or False (default None).
- **default_value** (*float*) – The value to return outside the mesh limits if limit is set to False (default None).

Returns An Discrete2DMesh object.

Return type *Discrete2DMesh*

class raysect.core.math.interpolators.interpolator2dmesh.**Interpolator2DMesh**

Bases: *raysect.core.math.function.function2d.Function2D*

Linear interpolator for data on a 2d ungridded tri-poly mesh.

The mesh is specified as a set of 2D vertices supplied as an Nx2 numpy array or a suitably sized sequence that can be converted to a numpy array.

The mesh triangles are defined with a Mx3 array where the three values are indices into the vertex array that specify the triangle vertices. The mesh must not contain overlapping triangles. Supplying a mesh with overlapping triangles will result in undefined behaviour.

A data array of length N, containing a value for each vertex, holds the data to be interpolated across the mesh.

By default, requesting a point outside the bounds of the mesh will cause a ValueError exception to be raised. If this is not desired the limit attribute (default True) can be set to False. When set to False, a default value will be returned for any point lying outside the mesh. The value return can be specified by setting the default_value attribute (default is 0.0).

To optimise the lookup of triangles, the interpolator builds an acceleration structure (a KD-Tree) from the specified mesh data. Depending on the size of the mesh, this can be quite slow to construct. If the user wishes to interpolate a number of different data sets across the same mesh - for example: temperature and density data that are both defined on the same mesh - then the user can use the instance() method on an existing interpolator to create a new interpolator. The new interpolator will shares a copy of the internal acceleration data. The vertex_data, limit and default_value can be customised for the new instance. See instance(). This will avoid the cost in memory and time of rebuilding an identical acceleration structure.

Parameters

- **vertex_coords** (*ndarray*) – An array of vertex coordinates (x, y) with shape Nx2.
- **vertex_data** (*ndarray*) – An array containing data for each vertex of shape Nx1.
- **triangles** (*ndarray*) – An array of vertex indices defining the mesh triangles, with shape Mx3.
- **limit** (*bool*) – Raise an exception outside mesh limits - True (default) or False.
- **default_value** (*float*) – The value to return outside the mesh limits if limit is set to False.

instance()

Creates a new interpolator instance from an existing interpolator instance.

The new interpolator instance will share the same internal acceleration data as the original interpolator. The `vertex_data`, `limit` and `default_value` settings of the new instance can be redefined by setting the appropriate attributes. If any of the attributes are set to `None` (default) then the value from the original interpolator will be copied.

This method should be used if the user has multiple sets of `vertex_data` that lie on the same mesh geometry. Using this methods avoids the repeated rebuilding of the mesh acceleration structures by sharing the geometry data between multiple interpolator objects.

Parameters

- **instance** (*Interpolator2DMesh*) – Interpolator2DMesh object.
- **vertex_data** (*ndarray*) – An array containing data for each vertex of shape `Nx1` (default `None`).
- **limit** (*bool*) – Raise an exception outside mesh limits - `True` (default) or `False` (default `None`).
- **default_value** (*float*) – The value to return outside the mesh limits if `limit` is set to `False` (default `None`).

Returns An Interpolator2DMesh object.

Return type *Interpolator2DMesh*

Random

Random samplers

`raysect.core.math.random.seed()`

Seeds the random number generator with the specified integer.

If a seed is not specified the generator is automatically re-seed using the system cryptographic random number generator (`urandom`).

Parameters `d` – Integer seed.

`raysect.core.math.random.uniform()`

Generate random doubles in range `[0, 1)`.

Values are uniformly distributed.

Returns Random double.

`raysect.core.math.random.normal()`

Generates a normally distributed random number.

The mean and standard deviation of the distribution must be specified.

Parameters

- **mean** (*float*) – The distribution mean.
- **stddev** (*float*) – The distribution standard deviation.

Returns Random double.

`raysect.core.math.random.probability()`

Samples from the Bernoulli distribution where $P(\text{True}) = \text{prob}$.

For example, if probability is 0.8, this function will return True 80% of the time and False 20% of the time.

Values of prob outside the [0, 1] range of probabilities will be clamped to the nearest end of the range [0, 1].

Parameters `prob` (*double*) – A probability from [0, 1].

Returns True or False.

Return type bool

`raysect.core.math.random.point_disk()`

Returns a random point on a disk of unit radius.

Return type *Point2D*

`raysect.core.math.random.point_square()`

Returns a random point on a square of unit radius.

Return type *Point2D*

`raysect.core.math.random.vector_sphere()`

Generates a random vector on a unit sphere.

Return type *Vector3D*

`raysect.core.math.random.vector_hemisphere_uniform()`

Generates a random vector on a unit hemisphere.

The hemisphere is aligned along the z-axis - the plane that forms the hemisphere base lies in the x-y plane.

Return type *Vector3D*

`raysect.core.math.random.vector_hemisphere_cosine()`

Generates a cosine-weighted random vector on a unit hemisphere.

The hemisphere is aligned along the z-axis - the plane that forms the hemisphere base lies in the x-y plane.

Return type *Vector3D*

`raysect.core.math.random.vector_cone()`

Generates a random vector in a cone along the z-axis.

The angle of the cone is specified with the theta parameter. For speed, no checks are performed on the theta parameter, it is up to user to ensure the angle is sensible.

Parameters `theta` (*float*) – An angle between 0 and 90 degrees.

Returns A random Vector3D in the cone defined by theta.

Return type *Vector3D*

Bulk sampling

class `raysect.core.math.sampler.PointSampler`

Base class for an object that generates a list of Point3D objects.

`__call__`

Parameters `samples` (*int*) – Number of points to generate.

Return type list

`sample()`

Parameters `samples` (*int*) – Number of points to generate.

Return type list

class `raysect.core.math.sampler.DiskSampler`

Bases: `raysect.core.math.sampler.PointSampler`

Generates a random Point3D on a disk.

Parameters `radius` (*double*) – The radius of the disk.

class `raysect.core.math.sampler.RectangleSampler`

Bases: `raysect.core.math.sampler.PointSampler`

Generates a random Point3D on a rectangle.

Parameters

- **width** (*double*) – The width of the rectangular sampling area of this observer.
- **height** (*double*) – The height of the rectangular sampling area of this observer.

class `raysect.core.math.sampler.VectorSampler`

Base class for an object that generates a list of Vector3D objects.

`__call__`

Parameters `samples` (*int*) – Number of vectors to generate.

Return type list

`sample()`

Parameters `samples` (*int*) – Number of vectors to generate.

Return type list

class `raysect.core.math.sampler.ConeSampler`

Bases: `raysect.core.math.sampler.VectorSampler`

Generates a list of random unit Vector3D objects inside a cone.

The cone is aligned along the z-axis.

Parameters `angle` – Angle of the cone in degrees.

class `raysect.core.math.sampler.SphereSampler`

Bases: `raysect.core.math.sampler.VectorSampler`

Generates a random vector on a unit sphere.

class `raysect.core.math.sampler.HemisphereUniformSampler`

Bases: `raysect.core.math.sampler.VectorSampler`

Generates a random vector on a unit hemisphere.

The hemisphere is aligned along the z-axis - the plane that forms the hemisphere base lies in the x-y plane.

class `raysect.core.math.sampler.HemisphereCosineSampler`

Bases: `raysect.core.math.sampler.VectorSampler`

Generates a cosine-weighted random vector on a unit hemisphere.

The hemisphere is aligned along the z-axis - the plane that forms the hemisphere base lies in the x-y plane.

Unit Conversions

`raysect.core.math.units.cm()`

Converts centimeters to meters.

Parameters *v* (*float*) – Length in centimeters.

Returns Length in meters.

`raysect.core.math.units.foot()`

Converts feet to meters.

Parameters *v* (*float*) – Length in feet.

Returns Length in meters.

`raysect.core.math.units.inch()`

Converts inches to meters.

Parameters *v* (*float*) – Length in inches.

Returns Length in meters.

`raysect.core.math.units.km()`

Converts kilometers to meters.

Parameters *v* (*float*) – Length in kilometers.

Returns Length in meters.

`raysect.core.math.units.mil()`

Converts mils (thousandths of an inch) to meters.

Parameters *v* (*float*) – Length in mils.

Returns Length in meters.

`raysect.core.math.units.mile()`

Converts miles to meters.

Parameters *v* (*float*) – Length in miles.

Returns Length in meters.

`raysect.core.math.units.mm()`

Converts millimeters to meters.

Parameters *v* (*float*) – Length in millimeters.

Returns Length in meters.

`raysect.core.math.units.nm()`

Converts nanometers to meters.

Parameters *v* (*float*) – Length in nanometers.

Returns Length in meters.

`raysect.core.math.units.radian()`

Converts radians to degrees.

Parameters *v* (*float*) – Angle in radians.

Returns Angle in degrees.

`raysect.core.math.units.um()`

Converts micrometers to meters.

Parameters \mathbf{v} (*float*) – Length in micrometers.

Returns Length in meters.

`raysect.core.math.units.yard()`

Converts yards to meters.

Parameters \mathbf{v} (*float*) – Length in yards.

Returns Length in meters.

Scenegraph Module

class `raysect.core.scenegraph.node.Node`

The scene-graph node class.

The basic constituent of a scene-graph tree. Nodes can be linked together by parenting one Node to another to form a tree structure. Each node in a scene-graph represents a distinct co-ordinate system. An affine transform associated with each node describes the relationship between a node and its parent's coordinate system. By combining the transforms (and inverse transforms) along the path between two nodes in the tree, the direct transform between any two arbitrary nodes, and thus their co-ordinate systems, can be calculated. Using this transform it is then possible to transform vectors and points between the two co-ordinate systems.

Parameters

- **parent** (*_NodeBase*) – Assigns the Node's parent to the specified scene-graph object.
- **transform** (*AffineMatrix3D*) – Sets the affine transform associated with the Node.
- **name** (*str*) – A string defining the node name.

name

The name of this node.

Getter Returns this node's name.

Setter Sets this node's name.

Return type *str*

parent

The parent of this node in the scenegraph.

Getter Returns this node's parent node.

Setter Sets this node's parent.

Return type *Node*

to()

Returns an affine transform that, when applied to a vector or point, transforms the vector or point from the co-ordinate space of the calling node to the co-ordinate space of the target node.

For example, if space B is translated +100 in x compared to space A and `A.to(B)` is called then the matrix returned would represent a translation of -100 in x. Applied to point (0,0,0) in A, this would produce the point (-100,0,0) in B as B is translated +100 in x compared to A.

Parameters **node** (*_NodeBase*) – The target node.

Returns An *AffineMatrix3D* describing the coordinate transform.

Rtype *AffineMatrix3D*

to_local()

Returns an affine transform from world space into this nodes local coordinate space.

Return type *AffineMatrix3D*

to_root()

Returns an affine transform from local space into the parent node's coordinate space.

Return type *AffineMatrix3D*

transform

The transform for this node's coordinate system in relation to the parent node.

Getter Returns this node's affine transform matrix.

Setter Sets this node's affine transform matrix.

Return type *AffineMatrix3D*

class raysect.core.scenegraph.observer.**Observer**

A scene-graph class for observing the world.

An observer class is intended to launch rays and sample the world. This is a base class and the observe function must be implemented by a deriving class. This object is the fundamental abstraction for items such as cameras, fibre optics and other sampling objects.

observe()

Virtual method - to be implemented by derived classes.

Triggers the exploration of the scene by emitting rays according to the model defined by the derived class implementing the method.

class raysect.core.scenegraph.primitive.**Primitive**

A scene-graph object representing a ray-intersectable surface/volume.

A primitive class defines an open surface or closed surface (volume) that can be intersected by a ray. For example, this could be a geometric primitive such as a sphere, or more complicated surface such as a polyhedral mesh. The primitive class is the only class in the scene-graph with which a ray can interact.

This is a base class, its functionality must be implemented fully by the deriving class.

Parameters

- **parent** (*_NodeBase*) – Assigns the Node's parent to the specified scene-graph object.
- **transform** (*AffineMatrix3D*) – Sets the affine transform associated with the Node.
- **material** (*Material*) – An object representing the material properties of the primitive.
- **name** (*str*) – A string defining the node name.

bounding_box()

Virtual method - to be implemented by derived classes.

When the primitive is connected to a scene-graph containing a World object at its root, this method should return a bounding box that fully encloses the primitive's surface (plus a small margin to avoid numerical accuracy problems). The bounding box must be defined in the world's coordinate space.

If this method is called when the primitive is not connected to a scene-graph with a World object at its root, it must throw a `TypeError` exception.

Returns A world space `BoundingBox3D` object.

Return type *BoundingBox3D*

contains()

Virtual method - to be implemented by derived classes.

Must returns True if the Point3D lies within the boundary of the surface defined by the Primitive. False is returned otherwise.

Parameters **p** ([Point3D](#)) – The Point3D to test.

Returns True if the Point3D is enclosed by the primitive surface, False otherwise.

Return type bool

hit()

Virtual method - to be implemented by derived classes.

Calculates the closest intersection of the Ray with the Primitive surface, if such an intersection exists.

If a hit occurs an Intersection object must be returned, otherwise None is returned. The intersection object holds the details of the intersection including the point of intersection, surface normal and the objects involved in the intersection.

Parameters **ray** ([Ray](#)) – The ray to test for intersection.

Returns An Intersection object or None if no intersection occurs.

Return type [Intersection](#)

material

The material class for this primitive.

Getter Returns this primitive's material.

Setter Sets this primitive's material.

Return type [Material](#)

next_intersection()

Virtual method - to be implemented by derived classes.

Returns the next intersection of the ray with the primitive along the ray path.

This method may only be called following a call to hit(). If the ray has further intersections with the primitive, these may be obtained by repeatedly calling the next_intersection() method. Each call to next_intersection() will return the next ray-primitive intersection along the ray's path. If no further intersections are found or intersections lie outside the ray parameters then next_intersection() will return None.

If any geometric elements of the primitive, ray and/or scene-graph are altered between a call to hit() and calls to next_intersection() the data returned by next_intersection() may be invalid. Primitives may cache data to accelerate next_intersection() calls which will be invalidated by geometric alterations to the scene. If the scene is altered the data returned by next_intersection() is undefined.

Return type [Intersection](#)

notify_geometry_change()

Notifies the scene-graph root of a change to the primitive's geometry.

This method must be called by primitives when their geometry changes. The notification informs the root node that any caching structures used to accelerate ray-tracing calculations are now potentially invalid and must be recalculated, taking the new geometry into account.

notify_material_change()

Notifies the scene-graph root of a change to the primitive's material.

This method must be called by primitives when their material changes. The notification informs the root node that any caching structures used to accelerate ray-tracing calculations are now potentially invalid and must be recalculated, taking the new material into account.

class raysect.core.scenegraph.world.**World**

The root node of the scene-graph.

The world node tracks all primitives and observers in the world. It maintains acceleration structures to speed up the ray-tracing calculations. The particular acceleration algorithm used is selectable. The default acceleration structure is a kd-tree.

Parameters **name** – A string defining the node name.

accelerator

The acceleration structure used for this world's scene-graph.

Getter Returns this world node's acceleration structure.

Setter Sets this world node's acceleration structure.

build_accelerator ()

This method manually triggers a rebuild of the Acceleration object.

If the Acceleration object is already in a consistent state this method will do nothing unless the force keyword option is set to True.

The Acceleration object is used to accelerate hit() and contains() calculations, typically using a spatial subdivision method. If changes are made to the scene-graph structure, transforms or to a primitive's geometry the acceleration structures may no longer represent the geometry of the scene and hence must be rebuilt. This process is usually performed automatically as part of the first call to hit() or contains() following a change in the scene-graph. As calculating these structures can take some time, this method provides the option of triggering a rebuild outside of hit() and contains() in case the user wants to be able to perform a benchmark without including the overhead of the Acceleration object rebuild.

Parameters **force** (*bool*) – If set to True, forces rebuilding of acceleration structure.

contains ()

Returns a list of Primitives that contain the specified point within their surface.

An empty list is returned if no Primitives contain the Point3D.

This method automatically rebuilds the Acceleration object that is used to optimise the contains calculation - if a Primitive's geometry or a transform affecting a primitive has changed since the last call to hit() or contains(), the Acceleration structure used to optimise the contains calculation is rebuilt to represent the new scene-graph state.

Parameters **point** (*Point3D*) – The point to test.

Returns A list containing all Primitives that enclose the Point3D.

Return type list

hit ()

Calculates the closest intersection of the Ray with the Primitives in the scene-graph, if such an intersection exists.

If a hit occurs an Intersection object is returned which contains the mathematical details of the intersection. None is returned if the ray does not intersect any primitive.

This method automatically rebuilds the Acceleration object that is used to optimise hit calculations - if a Primitive's geometry or a transform affecting a primitive has changed since the last call to hit() or contains(), the Acceleration structure used to optimise hit calculations is rebuilt to represent the new scene-graph state.

Parameters **ray** (*Ray*) – The ray to test.

Returns An Intersection object or None if no intersection occurs.

Return type *Intersection*

name

The name for this world node.

Getter Returns this world node's name.

Setter Sets this world node's name.

Return type str

observers

The list of observers in this scene-graph.

Getter Returns this world node's observers.

Return type list

primitives

The list of primitives maintained in this scene-graph.

Getter Returns this world node's primitive list.

Return type list

to()

Returns an affine transform that, when applied to a vector or point, transforms the vector or point from the co-ordinate space of the calling node to the co-ordinate space of the target node.

For example, if space B is translated +100 in x compared to space A and A.to(B) is called then the matrix returned would represent a translation of -100 in x. Applied to point (0,0,0) in A, this would produce the point (-100,0,0) in B as B is translated +100 in x compared to A.

Parameters **node** (*_NodeBase*) – The target node.

Returns An AffineMatrix3D describing the coordinate transform.

Return type *AffineMatrix3D*

`raysect.core.scenegraph.utility.print_scenegraph()`

Pretty-prints a scene-graph.

This function will print the scene-graph that contains the specified node. The specified node will be highlighted in the tree by post-fixing the node with the string: "[referring node]".

Parameters **node** (*_NodeBase*) – The target node.

Utilities

Containers

Raysect has a number of container classes available for fast operations in cython. These are mainly intended for use by developers.

class `raysect.core.containers.LinkedList`

Basic implementation of a Linked List for fast container operations in cython.

Parameters **initial_items** (*object*) – Optional iterable for initialising container.

Variables

- **length** (*int*) – number of items in the container
- **first** – starting element of container

- **last** – final element of container

add()

Add an item to the end of the container.

Parameters **value** (*object*) – The item to add to the end of the container.

add_items()

Extend this container with another iterable container.

Parameters **iterable** (*object*) – Iterable object such as a list or ndarray with which to extend this container.

get_index()

Get the item from the container at specified index.

Parameters **index** (*int*) – requested item index

insert()

Insert an item at the specified index.

Parameters

- **value** (*object*) – item to insert
- **index** (*int*) – index at which to insert this item

is_empty()

Returns True if the container is empty.

remove()

Remove and return the specified item from the container.

Parameters **index** (*int*) – Index at which an item will be removed.

Returns The object at the specified index position.

class raysect.core.containers.**Stack**

Bases: [raysect.core.containers.LinkedList](#)

Basic implementation of a Stack container for fast container operations in cython. Inherits attributes and methods from LinkedList.

pop()

Removes and returns the most recently added item from the stack

Return type object

push()

Adds an item to the top of the stack

Parameters **value** (*object*) – Object that will be pushed to top of the stack

class raysect.core.containers.**Queue**

Bases: [raysect.core.containers.LinkedList](#)

Basic implementation of a Queue container for fast container operations in cython. Inherits attributes and methods from LinkedList.

next_in_queue()

Returns the next object in the queue

Return type object

Primitives Module

Geometric Primitives

class raysect.primitive.Box

Bases: *raysect.core.scenegraph.primitive.Primitive*

A box primitive.

The box is defined by lower and upper points in the local co-ordinate system.

Parameters

- **lower** (*Point3D*) – Lower point of the box (default = *Point3D*(-0.5, -0.5, -0.5)).
- **upper** (*Point3D*) – Upper point of the box (default = *Point3D*(0.5, 0.5, 0.5)).
- **parent** (*Node*) – Scene-graph parent node or *None* (default = *None*).
- **transform** (*AffineMatrix3D*) – An *AffineMatrix3D* defining the local co-ordinate system relative to the scene-graph parent (default = identity matrix).
- **material** (*Material*) – A *Material* object defining the box’s material (default = *None*).
- **name** (*str*) – A string specifying a user-friendly name for the box (default = “”).

lower

Lower 3D coordinate of the box in primitive’s local coordinates.

Return type *Point3D*

upper

Upper 3D coordinate of the box in primitive’s local coordinates.

Return type *Point3D*

class raysect.primitive.Sphere

Bases: *raysect.core.scenegraph.primitive.Primitive*

A sphere primitive.

The sphere is centered at the origin of the local co-ordinate system.

Parameters

- **radius** (*float*) – Radius of the sphere in meters (default = 0.5).
- **parent** (*Node*) – Scene-graph parent node or *None* (default = *None*).
- **transform** (*AffineMatrix3D*) – An *AffineMatrix3D* defining the local co-ordinate system relative to the scene-graph parent (default = identity matrix).
- **material** (*Material*) – A *Material* object defining the sphere’s material (default = *None*).
- **name** (*str*) – A string specifying a user-friendly name for the sphere (default = “”).

radius

The radius of this sphere.

Return type *float*

class raysect.primitive.Cylinder

Bases: *raysect.core.scenegraph.primitive.Primitive*

A cylinder primitive.

The cylinder is defined by a radius and height. It lies along the z-axis and extends over the z range [0, height]. The ends of the cylinder are capped with disks forming a closed surface.

Parameters

- **radius** (*float*) – Radius of the cylinder in meters (default = 0.5).
- **height** (*float*) – Height of the cylinder in meters (default = 1.0).
- **parent** (*Node*) – Scene-graph parent node or None (default = None).
- **transform** (*AffineMatrix3D*) – An *AffineMatrix3D* defining the local co-ordinate system relative to the scene-graph parent (default = identity matrix).
- **material** (*Material*) – A *Material* object defining the cylinder’s material (default = None).
- **name** (*str*) – A string specifying a user-friendly name for the cylinder (default = “”).

height

Extent of the cylinder along the z-axis.

radius

Radius of the cylinder in x-y plane.

Return type float

class raysect.primitive.Cone

Bases: *raysect.core.scenegraph.primitive.Primitive*

A cone primitive.

The cone is defined by a radius and height. It lies along the z-axis and extends over the z range [0, height]. The tip of the cone lies at z = height. The base of the cone sits on the x-y plane and is capped with a disk, forming a closed surface.

Parameters

- **radius** (*float*) – Radius of the cone in meters in x-y plane (default = 0.5).
- **height** (*float*) – Height of the cone in meters (default = 1.0).
- **parent** (*Node*) – Scene-graph parent node or None (default = None).
- **transform** (*AffineMatrix3D*) – An *AffineMatrix3D* defining the local co-ordinate system relative to the scene-graph parent (default = identity matrix).
- **material** (*Material*) – A *Material* object defining the cone’s material (default = None).
- **name** (*str*) – A string specifying a user-friendly name for the cone (default = “”).

height

The extend of the cone along the z-axis

Return type float

radius

The radius of the cone base in the x-y plane

Return type float

class raysect.primitive.Parabola

Bases: *raysect.core.scenegraph.primitive.Primitive*

A parabola primitive.

The parabola is defined by a radius and height. It lies along the z-axis and extends over the z range [0, height]. The base of the parabola is capped with a disk forming a closed surface. The base of the parabola lies on the x-y plane, the parabola vertex (tip) lies at z=height.

Parameters

- **radius** (*float*) – Radius of the parabola in meters (default = 0.5).
- **height** (*float*) – Height of the parabola in meters (default = 1.0).
- **parent** (*Node*) – Scene-graph parent node or None (default = None).
- **transform** (*AffineMatrix3D*) – An AffineMatrix3D defining the local co-ordinate system relative to the scene-graph parent (default = identity matrix).
- **material** (*Material*) – A Material object defining the parabola’s material (default = None).
- **name** (*str*) – A string specifying a user-friendly name for the parabola (default = “”).

height

The parabola’s extent along the z-axis [0, height].

Return type float

radius

Radius of the parabola base in x-y plane.

Return type float

Meshes

class raysect.primitive.mesh.mesh.**Mesh**

Bases: *raysect.core.scenegraph.primitive.Primitive*

This primitive defines a polyhedral surface with triangular faces.

To define a new mesh, a list of vertices and triangles must be supplied. A set of vertex normals, used for smoothing calculations may also be provided.

The mesh vertices are supplied as an Nx3 list/array of floating point values. For each Vertex, x, y and z coordinates must be supplied. e.g.

```
vertices = [[0.0, 0.0, 1.0], [1.0, 0.0, 0.0], ...]
```

Vertex normals are similarly defined. Note that vertex normals must be correctly normalised.

The triangle array is either Mx3 or Mx6 - Mx3 if only vertices are defined or Mx6 if both vertices and vertex normals are defined. Triangles are defined by indexing into the vertex and vertex normal arrays. i.e:

```
triangles = [[v1, v2, v3, n1, n2, n3], ...]
```

where v1, v2, v3 are the vertex array indices specifying the triangle’s vertices and n1, n2, n3 are the normal array indices specifying the triangle’s surface normals at each vertex location. Where normals are not defined, n1, n2 and n3 are omitted.

The mesh may be an open surface (which does not enclose a volume) or a closed surface (which defines a volume). The nature of the mesh must be specified using the closed argument. If closed is True (default) then the mesh must be watertight and the face normals must be facing so they point out of the volume. If the mesh is open then closed must be set to False. Incorrectly setting the closed argument may result in undefined behaviour, depending on the application of the ray-tracer.

If vertex normals are defined for some or all of the triangles of the mesh then normal interpolation may be enabled for the mesh. For optical models this will result in a (suitably defined) mesh appearing smooth rather than faceted. If the triangles do not have vertex normals defined, the smoothing argument is ignored.

An alternate option for creating a new mesh is to create an instance of an existing mesh. An instance is a “clone” of the original mesh. Instances hold references to the internal data of the target mesh, they are therefore very memory efficient (particularly for detailed meshes) compared to creating a new mesh from scratch. If instance is set, it takes precedence over any other mesh creation settings.

If a mesh contains degenerate triangles (common for meshes generated from CAD models), enable tolerant mode to automatically remove them during mesh initialisation. A degenerate triangle is one where two or more vertices are coincident or all the vertices lie on the same line. Degenerate triangles will produce rendering error if encountered even though they are “infinitesimally” thin. A ray can still intersect them if they perfectly align as the triangle edges are treated as part of the triangle surface).

The `kdtree_*` arguments are tuning parameters for the kd-tree construction. For more information see the documentation of `KDTree3D`. The default values should result in efficient construction of the mesh’s internal kd-tree. Generally there is no need to modify these parameters unless the memory used by the kd-tree must be controlled. This may occur if very large meshes are used.

Parameters

- **vertices** (*object*) – An $N \times 3$ list of vertices.
- **triangles** (*object*) – An $M \times 3$ or $N \times 6$ list of vertex/normal indices defining the mesh triangles.
- **normals** (*object*) – An $K \times 3$ list of vertex normals or `None` (default=`None`).
- **smoothing** (*bool*) – True to enable normal interpolation (default=`True`).
- **closed** (*bool*) – True is the mesh defines a closed volume (default=`True`).
- **tolerant** (*bool*) – Mesh will automatically correct meshes with degenerate triangles if set to `True` (default=`True`).
- **instance** (*Mesh*) – The Mesh to become an instance of (default=`None`).
- **kdtree_max_depth** (*int*) – The maximum tree depth (automatic if set to 0, default=0).
- **kdtree_min_items** (*int*) – The item count threshold for forcing creation of a new leaf node (default=1).
- **kdtree_hit_cost** (*double*) – The relative computational cost of item hit evaluations vs kd-tree traversal (default=20.0).
- **kdtree_empty_bonus** (*double*) – The bonus applied to node splits that generate empty leaves (default=0.2).
- **parent** (*Node*) – Attaches the mesh to the specified scene-graph node (default=`None`).
- **transform** (*AffineMatrix3D*) – The co-ordinate transform between the mesh and its parent (default=unity matrix).
- **material** (*Material*) – The surface/volume material (default=`Material()` instance).
- **name** (*str*) – A human friendly name to identity the mesh in the scene-graph (default=“”).

bounding_box ()

Returns a world space bounding box that encloses the mesh.

The box is padded by a small margin to reduce the risk of numerical accuracy problems between the mesh and box representations following coordinate transforms.

Returns A `BoundingBox3D` object.

contains()

Identifies if the point lies in the volume defined by the mesh.

If a mesh is open, this method will always return False.

This method will fail if the face normals of the mesh triangles are not oriented to be pointing out of the volume surface.

Parameters **p** – The point to test.

Returns True if the point lies in the volume, False otherwise.

from_file()

Instances a new Mesh using data from a file object or filename.

The mesh must be stored in a RaySect Mesh (RSM) format file. RSM files are created with the Mesh save() method.

Parameters

- **file** (*object*) – File object or string path.
- **parent** (*Node*) – Attaches the mesh to the specified scene-graph node.
- **transform** (*AffineMatrix3D*) – The co-ordinate transform between the mesh and its parent.
- **material** (*Material*) – The surface/volume material.
- **name** (*str*) – A human friendly name to identity the mesh in the scene-graph.

hit()

Returns the first intersection with the mesh surface.

If an intersection occurs this method will return an Intersection object. The Intersection object will contain the details of the ray-surface intersection, such as the surface normal and intersection point.

If no intersection occurs None is returned.

Parameters **ray** – A world-space ray.

Returns An Intersection or None.

load()

Loads the mesh specified by a file object or filename.

The mesh must be stored in a RaySect Mesh (RSM) format file. RSM files are created with the Mesh save() method.

Parameters **file** – File object or string path.

next_intersection()

Returns the next intersection of the ray with the mesh along the ray path.

This method may only be called following a call to hit(). If the ray has further intersections with the mesh, these may be obtained by repeatedly calling the next_intersection() method. Each call to next_intersection() will return the next ray-mesh intersection along the ray's path. If no further intersections are found or intersections lie outside the ray parameters then next_intersection() will return None.

Returns An Intersection or None.

save()

Saves the mesh to the specified file object or filename.

The mesh is written in RaySect Mesh (RSM) format. The RSM format contains the mesh geometry and the mesh acceleration structures.

Parameters **file** – File object or string path.

```
raysect.primitive.mesh.obj.import_obj(cls, filename, scaling=1.0, **kwargs)
```

Create a mesh instance from a Wavefront OBJ mesh file (.obj).

Some engineering meshes are exported in different units (mm for example) whereas Raysect units are in m. Applying a scale factor of 0.001 would convert the mesh into m for use in Raysect.

Parameters

- **filename** (*str*) – Mesh file path.
- **scaling** (*double*) – Scale the mesh by this factor (default=1.0).
- ****kwargs** – Accepts optional keyword arguments from the Mesh class.

Return type *Mesh*

```
raysect.primitive.mesh.stl.import_stl(cls, filename, scaling=1.0, mode=0, **kwargs)
```

Create a mesh instance from a STereoLithography (STL) mesh file (.stl).

Some engineering meshes are exported in different units (mm for example) whereas Raysect units are in m. Applying a scale factor of 0.001 would convert the mesh into m for use in Raysect.

Parameters

- **filename** (*str*) – Mesh file path.
- **scaling** (*double*) – Scale the mesh by this factor (default=1.0).
- ****kwargs** – Accepts optional keyword arguments from the Mesh class.

Return type *Mesh*

CSG Operations

```
class raysect.primitive.csg.CSGPrimitive
```

Bases: *raysect.core.scenegraph.primitive.Primitive*

Constructive Solid Geometry (CSG) Primitive base class.

This is an abstract base class and can not be used directly.

CSG is a modeling technique that uses Boolean operations like union and intersection to combine 3D solids. For example, the volumes of a sphere and box could be unified with the ‘union’ operation to create a primitive with the combined volume of the underlying primitives.

Parameters

- **primitive_a** (*Primitive*) – Component primitive A of the compound primitive.
- **primitive_b** (*Primitive*) – Component primitive B of the compound primitive.
- **parent** (*Node*) – Scene-graph parent node or None (default = None).
- **transform** (*AffineMatrix3D*) – An *AffineMatrix3D* defining the local co-ordinate system relative to the scene-graph parent (default = identity matrix).
- **material** (*Material*) – A *Material* object defining the CSG primitive’s material (default = None).

primitive_a

Component primitive A of the compound CSG primitive.

Return type *Primitive*

primitive_b

Component primitive B of the compound CSG primitive.

Return type *Primitive*

class raysect.primitive.csg.**Union**

Bases: *raysect.primitive.csg.CSGPrimitive*

CSGPrimitive that is the volumetric union of primitives A and B.

All of the original volume from A and B will be in the new primitive.

Parameters

- **primitive_a** (*Primitive*) – Component primitive A of the union operation.
- **primitive_b** (*Primitive*) – Component primitive B of the union operation.
- **parent** (*Node*) – Scene-graph parent node or None (default = None).
- **transform** (*AffineMatrix3D*) – An *AffineMatrix3D* defining the local co-ordinate system relative to the scene-graph parent (default = identity matrix).
- **material** (*Material*) – A *Material* object defining the new CSG primitive's material (default = None).

class raysect.primitive.csg.**Intersect**

Bases: *raysect.primitive.csg.CSGPrimitive*

CSGPrimitive that is the volumetric intersection of primitives A and B.

Only volumes that are present in both primitives will be present in the new CSG primitive.

Parameters

- **primitive_a** (*Primitive*) – Component primitive A of the intersection operation.
- **primitive_b** (*Primitive*) – Component primitive B of the intersection operation.
- **parent** (*Node*) – Scene-graph parent node or None (default = None).
- **transform** (*AffineMatrix3D*) – An *AffineMatrix3D* defining the local co-ordinate system relative to the scene-graph parent (default = identity matrix).
- **material** (*Material*) – A *Material* object defining the new CSG primitive's material (default = None).

class raysect.primitive.csg.**Subtract**

Bases: *raysect.primitive.csg.CSGPrimitive*

CSGPrimitive that is the remaining volume of primitive A minus volume B.

Only volumes that are unique to primitive A and don't overlap with primitive B will be in the new CSG primitive.

Parameters

- **primitive_a** (*Primitive*) – Component primitive A of the intersection operation.
- **primitive_b** (*Primitive*) – Component primitive B of the intersection operation.
- **parent** (*Node*) – Scene-graph parent node or None (default = None).
- **transform** (*AffineMatrix3D*) – An *AffineMatrix3D* defining the local co-ordinate system relative to the scene-graph parent (default = identity matrix).
- **material** (*Material*) – A *Material* object defining the new CSG primitive's material (default = None).

Optical Module

Main Optical Classes

class raysect.optical.ray.**Ray**

Bases: `raysect.core.ray.Ray`

Optical Ray class for optical applications, inherits from core Ray class.

Parameters

- **origin** (`Point3D`) – Point defining ray’s origin (default=`Point3D(0, 0, 0)`)
- **direction** (`Vector3D`) – Vector defining ray’s direction (default=`Vector3D(0, 0, 1)`)
- **min_wavelength** (`float`) – Lower wavelength bound for observed spectrum
- **max_wavelength** (`float`) – Upper wavelength bound for observed spectrum
- **bins** (`int`) – Number of samples to use over the spectral range
- **max_distance** (`float`) – The terminating distance of the ray
- **extinction_prob** (`float`) – Probability of path extinction at every material surface interaction (default=0.1)
- **extinction_min_depth** (`int`) – Minimum number of paths before triggering extinction probability (default=3)
- **max_depth** (`int`) – Maximum number of material interactions before terminating ray trajectory.
- **importance_sampling** (`bool`) – Toggles use of importance sampling for important primitives. See help documentation on importance sampling, (default=True).
- **important_path_weight** (`float`) – Weight to use for important paths when using importance sampling.

bins

Number of spectral bins across wavelength range.

Return type `int`

copy()

Obtain a new Ray object with the same configuration settings.

Parameters

- **origin** (`Point3D`) – New Ray’s origin position.
- **direction** (`Vector3D`) – New Ray’s direction.

Return type `Ray`

extinction_min_depth

Minimum number of paths before triggering extinction probability.

Return type `int`

extinction_prob

Probability of path extinction at every material surface interaction.

Return type `float`

important_path_weight

Weight to use for important paths when using importance sampling.

Return type float

max_depth

Maximum number of material interactions before terminating ray trajectory.

Return type int

max_wavelength

Upper bound on wavelength range.

Return type float

min_wavelength

Lower bound on wavelength range.

Return type float

new_spectrum()

Returns a new Spectrum compatible with the ray spectral settings.

Return type *Spectrum*

sample()

Samples the radiance directed along the ray direction.

This methods calls trace repeatedly to obtain a statistical sample of the radiance directed along the ray direction from the world. The count parameter specifies the number of samples to obtain. The mean spectrum accumulated from these samples is returned.

Parameters

- **world** (*World*) – World object defining the scene.
- **count** (*int*) – Number of samples to take.

Returns The accumulated spectrum collected by the ray.

Return type *Spectrum*

spawn_daughter()

Spawns a new daughter of the ray.

A daughter ray has the same spectral configuration as the source ray, however the ray depth is increased by 1.

Parameters

- **origin** (*Point3D*) – A Point3D defining the ray origin.
- **direction** (*Vector3D*) – A vector defining the ray direction.

Returns A daughter Ray object.

Return type *Ray*

trace()

Traces a single ray path through the world.

Parameters

- **world** (*World*) – World object defining the scene.
- **keep_alive** (*bool*) – If true, disables Russian roulette termination of the ray.

Returns The resulting Spectrum object collected by the ray.

Return type *Spectrum*

wavelength_range

Upper and lower wavelength range.

Return type tuple

class raysect.optical.spectralfunction.**SpectralFunction**

SpectralFunction abstract base class.

A common interface for representing optical properties that are a function of wavelength. It provides methods for sampling, integrating and averaging a spectral function over specified wavelength ranges. The optical package uses SpectralFunctions to represent a number of different wavelength dependent optical properties, for example emission spectra, refractive indices and attenuation curves.

Deriving classes must implement the integrate method.

It is also recommended that subclasses implement `__call__()`. This should accept a single argument - wavelength - and return a single sample of the function at that wavelength. The units of wavelength are nanometers.

A number of utility sub-classes exist to simplify SpectralFunction development.

see also: NumericallyIntegratedSF, InterpolatedSF, ConstantSF, Spectrum

average ()

Average radiance over the requested spectral range ($\text{W/m}^2/\text{sr/nm}$).

Parameters

- **min_wavelength** (*float*) – lower wavelength for calculation
- **max_wavelength** (*float*) – upper wavelength for calculation

Return type float

sample ()

Re-sample the spectral function with a new wavelength range and resolution.

Parameters

- **min_wavelength** (*float*) – lower wavelength for calculation
- **max_wavelength** (*float*) – upper wavelength for calculation
- **bins** (*int*) – The number of spectral bins

Return type ndarray

class raysect.optical.spectralfunction.**NumericallyIntegratedSF**

Bases: *raysect.optical.spectralfunction.SpectralFunction*

Numerically integrates a supplied function.

This abstract class provides an implementation of the integrate method that numerically integrates a supplied function (typically a non-integrable analytical function). The function to numerically integrate is supplied by sub-classing this class and implementing the function() method.

The function is numerically sampled at regular intervals. A sampling resolution may be specified in the class constructor (default: 1 sample/nm).

Parameters **sample_resolution** (*double*) – The numerical sampling resolution in nanometers.

function ()

Function to numerically integrate.

This is a virtual method and must be implemented through sub-classing.

Parameters **wavelength** (*double*) – Wavelength in nanometers.

Returns Function value at the specified wavelength.

integrate ()

Calculates the integrated radiance over the specified spectral range.

Parameters

- **min_wavelength** (*float*) – The minimum wavelength in nanometers
- **max_wavelength** (*float*) – The maximum wavelength in nanometers

Returns Integrated radiance in W/m²/str

Return type float

class raysect.optical.spectralfunction.**InterpolatedSF**

Bases: *raysect.optical.spectralfunction.SpectralFunction*

Linearly interpolated spectral function.

Spectral function defined by samples of regular or irregular spacing, ends are extrapolated. You must set the ends to zero if you want the function to go to zero at the edges!

wavelengths and samples will be sorted during initialisation.

If normalise is set to True the data is rescaled so the integrated area of the spectral function over the full range of the input data is normalised to 1.0.

Parameters

- **wavelengths** (*object*) – 1D array of wavelengths in nanometers.
- **samples** (*object*) – 1D array of spectral samples.
- **normalise** (*bool*) – True/false toggle for whether to normalise the spectral function so its integral equals 1.

integrate ()

Calculates the integrated radiance over the specified spectral range.

Parameters

- **min_wavelength** (*float*) – The minimum wavelength in nanometers
- **max_wavelength** (*float*) – The maximum wavelength in nanometers

Returns Integrated radiance in W/m²/str

Return type float

class raysect.optical.spectralfunction.**ConstantSF**

Bases: *raysect.optical.spectralfunction.SpectralFunction*

Constant value spectral function

Parameters **value** (*float*) – Constant radiance value

average ()

Average radiance over the requested spectral range (W/m²/sr/nm).

Parameters

- **min_wavelength** (*float*) – lower wavelength for calculation
- **max_wavelength** (*float*) – upper wavelength for calculation

Return type float

integrate()

Calculates the integrated radiance over the specified spectral range.

Parameters

- **min_wavelength** (*float*) – The minimum wavelength in nanometers
- **max_wavelength** (*float*) – The maximum wavelength in nanometers

Returns Integrated radiance in W/m²/str

Return type float

sample()

Re-sample the spectral function with a new wavelength range and resolution.

Parameters

- **min_wavelength** (*float*) – lower wavelength for calculation
- **max_wavelength** (*float*) – upper wavelength for calculation
- **bins** (*int*) – The number of spectral bins

Return type ndarray

class raysect.optical.spectrum.Spectrum

Bases: *raysect.optical.spectralfunction.SpectralFunction*

A class for working with spectra.

Describes the distribution of light at each wavelength in units of radiance (W/m²/str/nm). Spectral samples are regularly spaced over the wavelength range and lie in the centre of the wavelength bins.

Parameters

- **min_wavelength** (*float*) – Lower wavelength bound for this spectrum
- **max_wavelength** (*float*) – Upper wavelength bound for this spectrum
- **bins** (*int*) – Number of samples to use over the spectral range

average()

Finds the average number of spectral samples over the specified wavelength range.

Parameters

- **min_wavelength** (*float*) – The minimum wavelength in nanometers
- **max_wavelength** (*float*) – The maximum wavelength in nanometers

Returns Average radiance in W/m²/str/nm

Return type float

copy()

Returns a copy of the spectrum.

Return type *Spectrum*

integrate()

Calculates the integrated radiance over the specified spectral range.

Parameters

- **min_wavelength** (*float*) – The minimum wavelength in nanometers
- **max_wavelength** (*float*) – The maximum wavelength in nanometers

Returns Integrated radiance in $\text{W/m}^2/\text{str}$

Return type float

is_compatible()

Returns True if the stored samples are consistent with the specified wavelength range and sample size.

Parameters

- **min_wavelength** (*float*) – The minimum wavelength in nanometers
- **max_wavelength** (*float*) – The maximum wavelength in nanometers
- **bins** (*int*) – The number of bins.

Returns True if the samples are compatible with the range/samples, False otherwise.

Return type boolean

is_zero()

Can be used to determine if all the samples are zero.

True if the spectrum is zero, False otherwise.

Return type bool

new_spectrum()

Returns a new Spectrum compatible with the same spectral settings.

Return type *Spectrum*

sample()

Re-sample this spectrum over a new spectral range.

Parameters

- **min_wavelength** (*float*) – The minimum wavelength in nanometers
- **max_wavelength** (*float*) – The maximum wavelength in nanometers
- **bins** (*int*) – The number of spectral bins.

Return type ndarray

to_photons()

Converts the spectrum sample array from radiance $\text{W/m}^2/\text{str}/\text{nm}$ to Photons/ $\text{s/m}^2/\text{str}/\text{nm}$ and returns the data in a numpy array.

Return type ndarray

total()

Calculates the total radiance integrated over the whole spectral range.

Returns radiance in $\text{W/m}^2/\text{str}$

Return type float

wavelengths

Wavelength array in nm

Return type ndarray

`raysect.optical.spectrum.photon_energy()`

Returns the energy of a photon with the specified wavelength.

Parameters **wavelength** (*float*) – Photon wavelength in nanometers.

Returns Photon energy in Joules.

Return type float

`raysect.optical.colour.ciexyz_to_srgb()`
sRGB specified as per IEC 61966-2-1:1999.

x, y, z in range [0, 1] r, g, b in range [0, 1]

`raysect.optical.colour.srgb_to_ciexyz()`
sRGB specified as per IEC 61966-2-1:1999.

r, g, b in range [0, 1] x, y, z in range [0, 1]

class `raysect.optical.scenegraph.world.World`
The root node of the optical scene-graph.

The world node tracks all primitives and observers in the world. It maintains acceleration structures to speed up the ray-tracing calculations. The particular acceleration algorithm used is selectable. The default acceleration structure is a kd-tree.

Parameters `name` – A string defining the node name.

Observers

class `raysect.optical.observer.imaging.pinhole.PinholeCamera`
An observer that models an idealised pinhole camera.

A simple camera that launches rays from the observer's origin point over a specified field of view.

Parameters

- **fov** (*double*) – The field of view of the camera in degrees (default: 45 degrees).
- **etendue** (*double*) – The etendue of each pixel (default: 1.0)

class `raysect.optical.observer.imaging.orthographic.OrthographicCamera`
A camera observing an orthogonal (orthographic) projection of the scene, avoiding perspective effects.

Arguments and attributes are inherited from the base Imaging sensor class.

Parameters **width** (*double*) – width of the orthographic area to observe in meters, the height is deduced from the 'pixels' attribute.

class `raysect.optical.observer.imaging.ccd.CCDArray`
An observer that models an idealised CCD-like imaging sensor.

The CCD is a regular array of square pixels. Each pixel samples red, green and blue channels (behaves like a Foveon imaging sensor). The CCD sensor width is specified with the width parameter. The CCD height is calculated from the width and the number of vertical and horizontal pixels. The default width and sensor ratio approximates a 35mm camera sensor.

Arguments and attributes are inherited from the base Imaging sensor class.

Parameters **width** (*double*) – The width in metres of the sensor (default is 0.035m).

class `raysect.optical.observer.imaging.vector.VectorCamera`
An observer that uses a specified set of pixel vectors.

A simple camera that uses calibrated vectors for each pixel to sample the scene. Arguments and attributes are inherited from the base Observer2D sensor class.

Parameters **fov** (*double*) – The field of view of the camera in degrees (default is 90 degrees).

class raysect.optical.observer.nonimaging.fibreoptic.**FibreOptic**

An optical fibre observer that samples rays from an acceptance cone and circular area at the fibre tip. Inherits arguments and attributes from the base NonImaging sensor class. Rays are sampled over a circular area at the fibre tip and a conical solid angle defined by the acceptance_angle parameter. :param float acceptance_angle: The angle in degrees between the z axis and the cone surface which defines the fibres

soild angle sampling area.

Parameters **radius** (*float*) – The radius of the fibre tip in metres. This radius defines a circular area at the fibre tip which will be sampled over.

class raysect.optical.observer.nonimaging.pixel.**Pixel**

A pixel observer that samples rays from a hemisphere and rectangular area.

Inherits arguments and attributes from the base NonImaging sensor class.

Parameters

- **x_width** (*float*) – The rectangular collection area’s width along the x-axis in local co-ordinates.
- **y_width** (*float*) – The rectangular collection area’s width along the y-axis in local co-ordinates.

class raysect.optical.observer.nonimaging.sightline.**SightLine**

An observer that fires rays along the observers z axis. Inherits arguments and attributes from the base NonImaging sensor class. Fires a single ray oriented along the observer’s z axis in world space.

Optical Materials

class raysect.optical.material.material.**ContinuousBSDF**

Surface space

to simplify maths: normal aligned (flipped) to sit on same side of surface as incoming ray incoming ray vector is aligned to point out of the surface surface space normal is aligned to lie along +ve Z-axis i.e. Normal3D(0, 0, 1)

The w_reflection_origin and w_transmission_origin points are provided as ray launch points. These points are guaranteed to prevent same-surface re-intersections. The reflection origin lies on the same side of the surface as the incoming ray, the transmission origin lies on the opposite side of the surface.

back_face is true if the ray is on the back side of the primitive surface, true if on the front side (ie on the side of the primitive surface normal)

class raysect.optical.material.material.**DiscreteBSDF**

Surface space

to simplify maths: normal aligned (flipped) to sit on same side of surface as incoming ray incoming ray vector is aligned to point out of the surface surface space normal is aligned to lie along +ve Z-axis i.e. Normal3D(0, 0, 1)

The w_reflection_origin and w_transmission_origin points are provided as ray launch points. These points are guaranteed to prevent same-surface re-intersections. The reflection origin lies on the same side of the surface as the incoming ray, the transmission origin lies on the opposite side of the surface.

back_face is true if the ray is on the back side of the primitive surface, true if on the front side (ie on the side of the primitive surface normal)

class raysect.optical.material.absorber.**AbsorbingSurface**

A perfectly absorbing surface material.

class raysect.optical.material.conductor.**Conductor**
Conductor material.

The conductor material simulates the interaction of light with a homogeneous conducting material, such as, gold, silver or aluminium.

This material implements the Fresnel equations for a conducting surface. To use the material, the complex refractive index of the conductor must be supplied.

Parameters

- **index** ([SpectralFunction](#)) – Real component of refractive index - $n(\lambda)$.
- **extinction** – Imaginary component of refractive index (extinction) - $k(\lambda)$.

class raysect.optical.material.conductor.**RoughConductor**
This is implementing Cook-Torrance with conducting fresnel microfacets.

Smith shadowing and GGX facet distribution used to model roughness.

This module contains materials to aid with debugging.

class raysect.optical.material.debug.**Light**
A Lambertian surface material illuminated by a distant light source.

This debug material lights the primitive from the world direction specified by a vector passed to the `light_direction` parameter. An optional intensity and emission spectrum may be supplied. By default the light spectrum is the D65 white point spectrum.

Parameters

- **light_direction** – A world space Vector3D defining the light direction.
- **intensity** – The light intensity (default is 1.0).
- **spectrum** – A [SpectralFunction](#) defining the light spectrum (default is D65 white).

class raysect.optical.material.debug.**PerfectReflectingSurface**
A material that is perfectly reflecting.

class raysect.optical.material.modifiers.**Roughen**
Modifies the surface normal to approximate a rough surface.

This is a modifier material, it takes another material (the base material) as an argument.

The roughen modifier works by randomly deflecting the surface normal about its true position before passing the intersection parameters on to the base material.

The deflection is calculated by interpolating between the existing normal and a vector sampled from a cosine weighted hemisphere. The strength of the interpolation, and hence the roughness of the surface, is controlled by the roughness argument. The roughness argument takes a value in the range [0, 1] where 1 is a fully rough, lambert-like surface and 0 is a smooth, untainted surface.

Parameters

- **material** – The base material.
- **roughness** – A double value in the range [0, 1].

CHAPTER 2

Indices and Tables

- `genindex`
- `modindex`
- `search`

r

- `raysect.core.boundingBox`, 20
- `raysect.core.intersection`, 19
- `raysect.core.material`, 19
- `raysect.core.math.affinematrix`, 29
- `raysect.core.math.interpolators.discrete2dmesh`, 34
- `raysect.core.math.interpolators.interpolator2dmesh`, 35
- `raysect.core.math.units`, 39
- `raysect.core.ray`, 19
- `raysect.core.scenegraph.node`, 40
- `raysect.core.scenegraph.observer`, 41
- `raysect.core.scenegraph.primitive`, 41
- `raysect.core.scenegraph.world`, 42
- `raysect.optical.colour`, 59
- `raysect.optical.material.absorber`, 60
- `raysect.optical.material.conductor`, 60
- `raysect.optical.material.debug`, 61
- `raysect.optical.material.dielectric`, 61
- `raysect.optical.material.emitter`, 60
- `raysect.optical.material.lambert`, 61
- `raysect.optical.material.material`, 60
- `raysect.optical.material.modifiers`, 61
- `raysect.optical.scenegraph.world`, 59

Symbols

- `__add__` (raysect.core.math.point.Point2D attribute), 23
- `__add__` (raysect.core.math.point.Point3D attribute), 25
- `__add__` (raysect.core.math.vector.Vector3D attribute), 27
- `__call__` (raysect.core.math.function.function1d.Function1D attribute), 32
- `__call__` (raysect.core.math.function.function2d.Function2D attribute), 33
- `__call__` (raysect.core.math.function.function3d.Function3D attribute), 33
- `__call__` (raysect.core.math.sampler.PointSampler attribute), 37
- `__call__` (raysect.core.math.sampler.VectorSampler attribute), 38
- `__getitem__` (raysect.core.math.point.Point2D attribute), 23
- `__getitem__` (raysect.core.math.point.Point3D attribute), 25
- `__getitem__` (raysect.core.math.vector.Vector3D attribute), 27
- `__getstate__` () (raysect.core.math.point.Point2D method), 24
- `__getstate__` () (raysect.core.math.point.Point3D method), 25
- `__getstate__` () (raysect.core.math.vector.Vector3D method), 27
- `__init__` (raysect.primitive.Box attribute), 11
- `__init__` (raysect.primitive.Cone attribute), 12
- `__init__` (raysect.primitive.Cylinder attribute), 11
- `__init__` (raysect.primitive.Sphere attribute), 10
- `__iter__` (raysect.core.math.point.Point2D attribute), 24
- `__iter__` (raysect.core.math.point.Point3D attribute), 25
- `__iter__` (raysect.core.math.vector.Vector3D attribute), 27
- `__mul__` (raysect.core.math.point.Point3D attribute), 25
- `__mul__` (raysect.core.math.vector.Vector3D attribute), 28
- `__neg__` (raysect.core.math.vector.Vector3D attribute), 28
- `__setitem__` (raysect.core.math.point.Point2D attribute), 24
- `__setitem__` (raysect.core.math.point.Point3D attribute), 25
- `__setitem__` (raysect.core.math.vector.Vector3D attribute), 28
- `__setstate__` () (raysect.core.math.point.Point2D method), 24
- `__setstate__` () (raysect.core.math.point.Point3D method), 25
- `__setstate__` () (raysect.core.math.vector.Vector3D method), 28
- `__sub__` (raysect.core.math.point.Point2D attribute), 24
- `__sub__` (raysect.core.math.point.Point3D attribute), 25
- `__sub__` (raysect.core.math.vector.Vector3D attribute), 28
- `__truediv__` (raysect.core.math.vector.Vector3D attribute), 28

A

- AbsorbingSurface (class in raysect.optical.material.absorber), 60
- accelerator (raysect.core.scenegraph.world.World attribute), 43
- add() (raysect.core.containers.LinkedList method), 45
- add_items() (raysect.core.containers.LinkedList method), 45
- AffineMatrix3D (class in raysect.core.math.affinematrix), 29
- average() (raysect.optical.spectralfunction.ConstantSF method), 56
- average() (raysect.optical.spectralfunction.SpectralFunction method), 55
- average() (raysect.optical.spectrum.Spectrum method), 57

B

- bins (raysect.optical.ray.Ray attribute), 53
- bounding_box() (raysect.core.scenegraph.primitive.Primitive method), 41

`bounding_box()` (raysect.primitive.mesh.mesh.Mesh method), 49
`BoundingBox2D` (class in raysect.core.boundingBox), 20
`BoundingBox3D` (class in raysect.core.boundingBox), 21
`Box` (class in raysect.primitive), 11, 46
`build_accelerator()` (raysect.core.scenegraph.world.World method), 43

C

`CCDArray` (class in raysect.optical.observer.imaging.ccd), 59
`centre` (raysect.core.boundingBox.BoundingBox3D attribute), 21
`ciexyz_to_srgb()` (in module raysect.optical.colour), 59
`cm()` (in module raysect.core.math.units), 39
`Conductor` (class in raysect.optical.material.conductor), 60
`Cone` (class in raysect.primitive), 11, 47
`ConeSampler` (class in raysect.core.math.sampler), 38
`ConstantSF` (class in raysect.optical.spectralfunction), 56
`contains()` (raysect.core.boundingBox.BoundingBox2D method), 20
`contains()` (raysect.core.boundingBox.BoundingBox3D method), 21
`contains()` (raysect.core.scenegraph.primitive.Primitive method), 41
`contains()` (raysect.core.scenegraph.world.World method), 43
`contains()` (raysect.primitive.mesh.mesh.Mesh method), 49
`ContinuousBSDF` (class in raysect.optical.material.material), 60
`copy()` (raysect.core.math.point.Point2D method), 24
`copy()` (raysect.core.math.point.Point3D method), 25
`copy()` (raysect.core.math.vector.Vector2D method), 26
`copy()` (raysect.core.math.vector.Vector3D method), 28
`copy()` (raysect.core.ray.Ray method), 19
`copy()` (raysect.optical.ray.Ray method), 53
`copy()` (raysect.optical.spectrum.Spectrum method), 57
`cross()` (raysect.core.math.vector.Vector2D method), 26
`cross()` (raysect.core.math.vector.Vector3D method), 28
`CSGPrimitive` (class in raysect.primitive.csg), 51
`Cylinder` (class in raysect.primitive), 11, 46

D

`Discrete2DMesh` (class in raysect.core.math.interpolators.discrete2dmesh), 34
`DiscreteBSDF` (class in raysect.optical.material.material), 60
`DiskSampler` (class in raysect.core.math.sampler), 38
`distance_to()` (raysect.core.math.point.Point2D method), 24

`distance_to()` (raysect.core.math.point.Point3D method), 25
`dot()` (raysect.core.math.vector.Vector2D method), 26
`dot()` (raysect.core.math.vector.Vector3D method), 28

E

`enclosing_sphere()` (raysect.core.boundingBox.BoundingBox3D method), 21
`extend()` (raysect.core.boundingBox.BoundingBox2D method), 20
`extend()` (raysect.core.boundingBox.BoundingBox3D method), 22
`extent()` (raysect.core.boundingBox.BoundingBox2D method), 20
`extent()` (raysect.core.boundingBox.BoundingBox3D method), 22
`extinction_min_depth` (raysect.optical.ray.Ray attribute), 53
`extinction_prob` (raysect.optical.ray.Ray attribute), 53

F

`FibreOptic` (class in raysect.optical.observer.nonimaging.fibreoptic), 59
`foot()` (in module raysect.core.math.units), 39
`from_file()` (raysect.primitive.mesh.mesh.Mesh method), 50
`full_intersection()` (raysect.core.boundingBox.BoundingBox3D method), 22
`function()` (raysect.optical.spectralfunction.NumericallyIntegratedSF method), 55
`Function1D` (class in raysect.core.math.function.function1d), 32
`Function2D` (class in raysect.core.math.function.function2d), 32
`Function3D` (class in raysect.core.math.function.function3d), 33

G

`get_index()` (raysect.core.containers.LinkedList method), 45

H

`height` (raysect.primitive.Cone attribute), 47
`height` (raysect.primitive.Cylinder attribute), 47
`height` (raysect.primitive.Parabola attribute), 48
`HemisphereCosineSampler` (class in raysect.core.math.sampler), 38
`HemisphereUniformSampler` (class in raysect.core.math.sampler), 38
`hit()` (raysect.core.boundingBox.BoundingBox3D method), 22

- hit() (raysect.core.scenegraph.primitive.Primitive method), 42
- hit() (raysect.core.scenegraph.world.World method), 43
- hit() (raysect.primitive.mesh.mesh.Mesh method), 50
- ## I
- import_obj() (in module raysect.primitive.mesh.obj), 13, 51
- import_stl() (in module raysect.primitive.mesh.stl), 13, 51
- important_path_weight (raysect.optical.ray.Ray attribute), 53
- inch() (in module raysect.core.math.units), 39
- insert() (raysect.core.containers.LinkedList method), 45
- instance() (raysect.core.math.interpolators.discrete2dmesh.Discrete2DMesh method), 34
- instance() (raysect.core.math.interpolators.interpolator2dmesh.Interpolator2DMesh method), 35
- integrate() (raysect.optical.spectralfunction.ConstantSF method), 56
- integrate() (raysect.optical.spectralfunction.InterpolatedSF method), 56
- integrate() (raysect.optical.spectralfunction.NumericallyIntegratedSF method), 56
- integrate() (raysect.optical.spectrum.Spectrum method), 57
- InterpolatedSF (class in raysect.optical.spectralfunction), 56
- Interpolator2DMesh (class in raysect.core.math.interpolators.interpolator2dmesh), 35
- Intersect (class in raysect.primitive.csg), 52
- Intersection (class in raysect.core.intersection), 19
- inverse() (raysect.core.math.affinematrix.AffineMatrix3D method), 30
- is_compatible() (raysect.optical.spectrum.Spectrum method), 58
- is_empty() (raysect.core.containers.LinkedList method), 45
- is_zero() (raysect.optical.spectrum.Spectrum method), 58
- ## K
- km() (in module raysect.core.math.units), 39
- ## L
- largest_axis() (raysect.core.boundingBox.BoundingBox2D method), 20
- largest_axis() (raysect.core.boundingBox.BoundingBox3D method), 22
- largest_extent() (raysect.core.boundingBox.BoundingBox2D method), 20
- largest_extent() (raysect.core.boundingBox.BoundingBox3D method), 22
- length (raysect.core.math.vector.Vector2D attribute), 27
- length (raysect.core.math.vector.Vector3D attribute), 28
- Light (class in raysect.optical.material.debug), 61
- LinkedList (class in raysect.core.containers), 44
- load() (raysect.primitive.mesh.mesh.Mesh method), 50
- lower (raysect.core.boundingBox.BoundingBox2D attribute), 20
- lower (raysect.core.boundingBox.BoundingBox3D attribute), 22
- lower (raysect.primitive.Box attribute), 46
- ## M
- material (raysect.core.scenegraph.primitive.Primitive attribute), 42
- max_depth (raysect.optical.ray.Ray attribute), 54
- max_wavelength (raysect.optical.ray.Ray attribute), 54
- Mesh (class in raysect.primitive.mesh.mesh), 48
- mi.Interpolator2DMesh (class in raysect.core.math.interpolators.interpolator2dmesh), 39
- mile() (in module raysect.core.math.units), 39
- min_wavelength (raysect.optical.ray.Ray attribute), 54
- mm() (in module raysect.core.math.units), 39
- ## N
- name (raysect.core.scenegraph.node.Node attribute), 40
- name (raysect.core.scenegraph.world.World attribute), 44
- new_spectrum() (raysect.optical.ray.Ray method), 54
- new_spectrum() (raysect.optical.spectrum.Spectrum method), 58
- next_in_queue() (raysect.core.containers.Queue method), 45
- next_intersection() (raysect.core.scenegraph.primitive.Primitive method), 42
- next_intersection() (raysect.primitive.mesh.mesh.Mesh method), 50
- nm() (in module raysect.core.math.units), 39
- Node (class in raysect.core.scenegraph.node), 40
- normal() (in module raysect.core.math.random), 36
- normalise() (raysect.core.math.vector.Vector2D method), 27
- normalise() (raysect.core.math.vector.Vector3D method), 29
- notify_geometry_change() (raysect.core.scenegraph.primitive.Primitive method), 42
- notify_material_change() (raysect.core.scenegraph.primitive.Primitive method), 42
- NumericallyIntegratedSF (class in raysect.optical.spectralfunction), 55
- ## O
- observe() (raysect.core.scenegraph.observer.Observer method), 41
- Observer (class in raysect.core.scenegraph.observer), 41

observers (raysect.core.scenegraph.world.World attribute), 44
orthogonal() (raysect.core.math.vector.Vector2D method), 27
orthogonal() (raysect.core.math.vector.Vector3D method), 29
OrthographicCamera (class in raysect.optical.observer.imaging.orthographic), 59

P

pad() (raysect.core.boundingBox.BoundingBox2D method), 20
pad() (raysect.core.boundingBox.BoundingBox3D method), 22
Parabola (class in raysect.primitive), 47
parent (raysect.core.scenegraph.node.Node attribute), 40
PerfectReflectingSurface (class in raysect.optical.material.debug), 61
photon_energy() (in module raysect.optical.spectrum), 58
PinholeCamera (class in raysect.optical.observer.imaging.pinhole), 59
Pixel (class in raysect.optical.observer.nonimaging.pixel), 60
Point2D (class in raysect.core.math.point), 23
Point3D (class in raysect.core.math.point), 24
point_disk() (in module raysect.core.math.random), 37
point_on() (raysect.core.ray.Ray method), 19
point_square() (in module raysect.core.math.random), 37
PointSampler (class in raysect.core.math.sampler), 37
pop() (raysect.core.containers.Stack method), 45
Primitive (class in raysect.core.scenegraph.primitive), 41
primitive_a (raysect.primitive.csg.CSGPrimitive attribute), 51
primitive_b (raysect.primitive.csg.CSGPrimitive attribute), 51
primitives (raysect.core.scenegraph.world.World attribute), 44
print_scenegraph() (in module raysect.core.scenegraph.utility), 44
probability() (in module raysect.core.math.random), 36
push() (raysect.core.containers.Stack method), 45
PythonFunction1D (class in raysect.core.math.function.function1d), 32
PythonFunction2D (class in raysect.core.math.function.function2d), 33
PythonFunction3D (class in raysect.core.math.function.function3d), 33

Q

Queue (class in raysect.core.containers), 45

R

radian() (in module raysect.core.math.units), 39

radius (raysect.primitive.Cone attribute), 47
radius (raysect.primitive.Cylinder attribute), 47
radius (raysect.primitive.Parabola attribute), 48
radius (raysect.primitive.Sphere attribute), 46
Ray (class in raysect.core.ray), 19
Ray (class in raysect.optical.ray), 53
raysect.core.boundingBox (module), 20
raysect.core.intersection (module), 19
raysect.core.material (module), 19
raysect.core.math.affinematrix (module), 29
raysect.core.math.interpolators.discrete2dmesh (module), 34
raysect.core.math.interpolators.interpolator2dmesh (module), 35
raysect.core.math.units (module), 39
raysect.core.ray (module), 19
raysect.core.scenegraph.node (module), 40
raysect.core.scenegraph.observer (module), 41
raysect.core.scenegraph.primitive (module), 41
raysect.core.scenegraph.world (module), 42
raysect.optical.colour (module), 59
raysect.optical.material.absorber (module), 60
raysect.optical.material.conductor (module), 60
raysect.optical.material.debug (module), 61
raysect.optical.material.dielectric (module), 61
raysect.optical.material.emitter (module), 60
raysect.optical.material.lambert (module), 61
raysect.optical.material.material (module), 60
raysect.optical.material.modifiers (module), 61
raysect.optical.scenegraph.world (module), 59
RectangleSampler (class in raysect.core.math.sampler), 38
remove() (raysect.core.containers.LinkedList method), 45
rotate() (in module raysect.core.math.transform), 31
rotate_basis() (in module raysect.core.math.transform), 32
rotate_vector() (in module raysect.core.math.transform), 31
rotate_x() (in module raysect.core.math.transform), 30
rotate_y() (in module raysect.core.math.transform), 31
rotate_z() (in module raysect.core.math.transform), 31
RoughConductor (class in raysect.optical.material.conductor), 61
Roughen (class in raysect.optical.material.modifiers), 61

S

sample() (raysect.core.math.sampler.PointSampler method), 37
sample() (raysect.core.math.sampler.VectorSampler method), 38
sample() (raysect.optical.ray.Ray method), 54
sample() (raysect.optical.spectralfunction.ConstantSF method), 57

sample() (raysect.optical.spectralfunction.SpectralFunction method), 55

sample() (raysect.optical.spectrum.Spectrum method), 58

save() (raysect.primitive.mesh.mesh.Mesh method), 50

seed() (in module raysect.core.math.random), 36

SightLine (class in raysect.optical.observer.nonimaging.sightline), 60

spawn_daughter() (raysect.optical.ray.Ray method), 54

SpectralFunction (class in raysect.optical.spectralfunction), 55

Spectrum (class in raysect.optical.spectrum), 57

Sphere (class in raysect.primitive), 10, 46

SphereSampler (class in raysect.core.math.sampler), 38

srgb_to_ciexyz() (in module raysect.optical.colour), 59

Stack (class in raysect.core.containers), 45

Subtract (class in raysect.primitive.csg), 52

surface_area() (raysect.core.boundingBox.BoundingBox2D method), 20

surface_area() (raysect.core.boundingBox.BoundingBox3D method), 23

T

to() (raysect.core.scenegraph.node.Node method), 40

to() (raysect.core.scenegraph.world.World method), 44

to_local() (raysect.core.scenegraph.node.Node method), 40

to_photons() (raysect.optical.spectrum.Spectrum method), 58

to_root() (raysect.core.scenegraph.node.Node method), 41

total() (raysect.optical.spectrum.Spectrum method), 58

trace() (raysect.optical.ray.Ray method), 54

transform (raysect.core.scenegraph.node.Node attribute), 41

transform() (raysect.core.math.point.Point3D method), 26

transform() (raysect.core.math.vector.Vector3D method), 29

translate() (in module raysect.core.math.transform), 30

U

um() (in module raysect.core.math.units), 39

uniform() (in module raysect.core.math.random), 36

Union (class in raysect.primitive.csg), 52

union() (raysect.core.boundingBox.BoundingBox2D method), 21

union() (raysect.core.boundingBox.BoundingBox3D method), 23

upper (raysect.core.boundingBox.BoundingBox2D attribute), 21

upper (raysect.core.boundingBox.BoundingBox3D attribute), 23

upper (raysect.primitive.Box attribute), 46

V

Vector2D (class in raysect.core.math.vector), 26

Vector3D (class in raysect.core.math.vector), 27

vector_cone() (in module raysect.core.math.random), 37

vector_hemisphere_cosine() (in module raysect.core.math.random), 37

vector_hemisphere_uniform() (in module raysect.core.math.random), 37

vector_sphere() (in module raysect.core.math.random), 37

vector_to() (raysect.core.math.point.Point2D method), 24

vector_to() (raysect.core.math.point.Point3D method), 26

VectorCamera (class in raysect.optical.observer.imaging.vector), 59

VectorSampler (class in raysect.core.math.sampler), 38

vertices() (raysect.core.boundingBox.BoundingBox2D method), 21

vertices() (raysect.core.boundingBox.BoundingBox3D method), 23

volume() (raysect.core.boundingBox.BoundingBox3D method), 23

W

wavelength_range (raysect.optical.ray.Ray attribute), 55

wavelengths (raysect.optical.spectrum.Spectrum attribute), 58

World (class in raysect.core.scenegraph.world), 42

World (class in raysect.optical.scenegraph.world), 59

Y

yard() (in module raysect.core.math.units), 40