Padvinder Documentation

Release 0.0.1

Adrian Köring

Feb 21, 2018

Package Structure

CHAPTER 1

Padvinder

1.1 Ray

Rays are stand-ins for lightrays heading from the camera through the scene.

class padvinder.ray.**Ray**(*position=(0, 0, 0)*, *direction=(1, 0, 0)*)

Bases: object

A ray consists of a starting position and a direction. Both are specified as vectors.

The starting position is a point in the scene, the ray begins in. The direction is where the ray is heading in and is always normalized.

Parameters

- **position** (*numpy.ndarray_like*) – an array of three dimension
- **direction** (*numpy.ndarray_like*) – Direction must have the same number of dimension as position. The direction vector will be stored normalized to a length of one and can not initially have lenght zero.

Raises

- ValueError – Raises a ValueError if the input contains NaNs or Infs.
- ZeroDivisionError – Raises a ZeroDivisionError if the direction vector has length zero

Examples

```
>>> Ray((0, 0, 0), (1, 0, 0))
Ray([0.0, 0.0, 0.0], [1.0, 0.0, 0.0])
>>> Ray((3.0, 2.3, -5.1), (-10.0, 34.0, -2.0))
Ray([3.0, 2.3, -5.1], [-0.28171808,  0.95784149, -0.05634362])
```

position
Return the ray's position.

direction
Return the ray's normalized direction.

point (*distance*)
Returns a point lying t-units from the origin on the ray.

Parameters **distance** (*float*) – The number of units along the ray to get the point of

Returns **point on ray** – where the point is calculated as ray_origin + distance*ray_direction

Return type numpy.ndarray_like

Examples

```
>>> Ray((0, 0, 0), (1, 0, 0)).point(10)
[10.0, 0.0, 0.0]
```

1.2 Camera

Cameras produce the initial ray from the camera position through the currently rendered pixel and into the scene.

class padvinder.camera.**Camera** (*position=(0, 0, 0), up=(0, 1, 0), look_at=(0, 0, 1)*)
Bases: object

This Camera Model sets up and contains an orthonormal coordinate system. The cameras position is the starting positon of all rays fired into the scene. The rays through the center of the image will pass through the look_at point. Position and look_at define the optical axis. The up vector provides a vague upwards direction helping to orient the camera.

Parameters

- **position** (*numpy.ndarray_like*) – the position of the camera in the scene, origin of the fired rays
- **up** (*numpy.ndarray_like*) – the general upwards direction of the camera
- **look_at** (*numpy.ndarray_like*) – the position in the scene that the camera will look at

Raises ValueError – if any of position, up or look_at contain Infs or NaNs or if the position and look_at are at the same point

Examples

```
>>> Camera()
>>> Camera((0,0,0), (0,1,0), (0,0,-1), 35)
Camera(position=[0.0, 0.0, -1.0],
       up=[0.0, 1.0, 0.0],
       look_at=[0.0, 0.0, 0.0])
```

position
Return the position of the camera.

up
Return the up vector of the camera.

optical_axis
Return the optical axis of the camera.

ray(*pixel*, *resolutions*, *rand=True*)
Given the pixel and the camera resolution, returns a ray that originates at the camera position and passes through the pixel. If rand is set true, a little random offset (smaller than the distance between two pixels is added to the pixel position. This will together with multiple samples per pixel mitigate aliasing.

Parameters

- **pixel** (*numpy.ndarray_like*) – (x, y) coordinates of the pixel in the image - numpy style. Aka (0, 0) is the upper left hand corner and the x values are iterating downwards while y is iterating horizontally. x must be in the intervall of [0, dimension[0]] and y must be in [0, dimension[1]] The pixel [0,0] is the upper lefthand corner and the pixel [res_x, rex_y] is the lower righthand corner.
- **resolutions** (*numpy.ndarray_like*) – (res_x, resx_y) the resolution of the camera in x and y.
- **rand** (*boolean*) – When False, every ray passes through the exact center of the pixel. When True a random offset smaller than the distance between two pixels is added the the pixel center. The ray then passes through the perturbed pixel center.

Returns **ray** – with the position being the camera position and direction being a vector that starts at the position and passes through the (potentiall offsetted) given pixel

Return type *Ray*

Raises NotImplemeted – because this is an abstract base class

Examples

```
>>> camera = PerspectiveCamera()
>>> camera.ray((0,0), (100, 100), False)
```

class padvinder.camera.**PerspectiveCamera**(*position=(0, 0, 0)*, *up=(0, 1, 0)*, *look_at=(0, 0, 1)*, *focal_length=24*)

Bases: *padvinder.camera.Camera*

The Perspective Camera Model extends the orthonormal coordinate system with a focal length and therefore a concrete image plane. The cameras position is the starting positon of all rays fired into the scene. The rays through the center of the image will pass through the look_at point. Position and look_at define the optical axis. The up vector provides a vague upwards direction helping to orient the camera. The focal length defines how far the 35mm equivalent sized image plane is from the camera position.

Parameters

- **position** (*numpy.ndarray_like*) – the position of the camera in the scene, origin of the fired rays
- **up** (*numpy.ndarray_like*) – the general upwards direction of the camera
- **look_at** (*numpy.ndarray_like*) – the position in the scene that the camera will look at
- **focal_length** (*float*) – the distance in mm between the position and the image plane - must be in the intervall of (0, +inf).

Raises `ValueError` – if any of position, up or look_at contain Infs or NaNs or if the position and look_at are at the same point of ir the focal_length is not positive

Examples

```
>>> PerspectiveCamera()
>>> PerspectiveCamera((0,0,1), (0,1,0), (0,0,0), 24)
Camera(position=[0.0, 0.0, 1.0],
       up=[0.0, 1.0, 0.0],
       look_at=[0.0, 0.0, 0.0],
       focal_length=24)
```

focal_length
Return the focal length of the camera.

ray(*pixel*, *resolutions*, *rand=True*)
Given the pixel and the camera resolution, returns a ray that originates at the camera position and passes through the pixel. If rand is set true, a little random offset (smaller than the distance between two pixels is added to the pixel position. This will together with multiple samples per pixel mitigate aliasing.

Parameters

- **pixel** (*numpy.ndarray_like of shape (2,)*) – (x, y) coordinates of the pixel in the image. Numpy style: aka (0, 0) is the upper left hand corner and the x values are iterating downwards while y is iterating horizontally. x must be in the intervall of [0, dimension[0]] and y must be in [0, dimension[1]] The pixel [0,0] is the upper lefthand corner and the pixel [res_x, rex_y] is the lower righthand corner.
- **resolutions** (*numpy.ndarray_like of shape (2,)*) – the resolution of the camera in x and y.
- **rand** (*boolean*) – When False, every ray passes through the exact center of the pixel. When True a random offset smaller than the distance between two pixels is added the the pixel center. The ray then passes through the perturbed pixel center.

Returns **ray** – with the position being the camera position and direction being a vector that starts at the position and passes through the (potentiall offsetted) given pixel

Return type *Ray*

Examples

```
>>> camera = PerspectiveCamera()
>>> camera.ray((50, 50), (100, 100), False)
Ray(position=[0, 0, 0], direction=[0, 0, 1])
```

1.3 Material

Materials define the surface properties and specify how light rays get coloured and reflected. All materials are callables - they implement the __call__ method and can be used like functions.

class padvinder.material.**Material**(*color=(0.5, 0.5, 0.5)*)
An emission material consists of an emitted colour only. Without gradients, lighting or anything.

Parameters **color** (*numpy.ndarray_like*) – of three dimensions and contains colors as (Red, Green, Blue) where (0,0,0) is black and (1,1,1) is white

Raises ValueError – if the color contains any non-finite (inf, nan) values

Examples

```
>>> Material((0.8, 0.8, 0.8))
Material(color=[.8 .8 .8])
```

color
Returns the color of the material.

__call__(*surface_normal*, *incoming_color*, *incoming_direction*, *outgoing_direction*)
Calculate light reflected from the material toward the outgoing direction. Keep in mind, while pathtracing starts at the camera and heads into the scene, the rays contribution is accumulated 'backwards'. Therefore the incoming direction is further down the path and outgoing_direction is closer towards the camera.

Parameters

- **surface_normal** (*numpy.ndarray_like*) – normal vector at the geometries surface
- **incoming_color** (*numpy.ndarray_like*) – the color the ray has accumulated up to this point
- **incoming_direction** (*numpy.ndarray_like*) – the direction from where the 'light shines' onto the surface
- **outgoing_direction** (*numpy.ndarray_like*) – the direction into which the 'light gets reflected' from the surface

Returns **color** – the light color 'getting reflected' from the surface

Return type numpy.ndarray_like

outgoing_direction(*normal*, *incoming_direction*)
Given a surface normal and an incoming direction, determine the direction in which the path continues.

normal [numpy.ndarray_like of shape (3,)] the surface normal at the intersection point

incoming_direction [numpy.ndarray_like of shape (3,)] the direction from which light hits the surface

Returns **outgoing direction** – the direction in which light is reflected from the surface

Return type numpy.ndarray_like of shape (3, 0)

class padvinder.material.**Emission**(*color=(10*, *10*, *10)*)
Emission is equivalent to the abstract base class Material. Due to semantics this class exists and merely inherits without modifications.

Parameters **color** (*numpy.ndarray_like*) – of three dimensions and contains colors as (Red, Green, Blue) where (0,0,0) is black and (1,1,1) is white

Raises ValueError – if the color contains any non-finite (inf, nan) values

Examples

```
>>> Emission()
Emission(color=[10.0, 10.0, 10.0])
```

__call__(*surface_normal*, *incoming_color*, *incoming_direction*, *outgoing_direction*)
Calculate light reflected from the material toward the outgoing direction. Keep in mind, while pathtracing starts at the camera and heads into the scene, the rays contribution is accumulated 'backwards'. Therefore the incoming direction is further down the path and outgoing_direction is closer towards the camera.

Parameters

- **surface_normal** (*numpy.ndarray_like*) – normal vector at the geometries surface
- **incoming_color** (*numpy.ndarray_like*) – the color the ray has accumulated up to this point
- **incoming_direction** (*numpy.ndarray_like*) – the direction from where the 'light shines' onto the surface
- **outgoing_direction** (*numpy.ndarray_like*) – the direction into which the 'light gets reflected' from the surface

Returns **color** – the light color 'getting reflected' from the surface

Return type numpy.ndarray_like

color
Returns the color of the material.

outgoing_direction(*normal*, *incoming_direction*)
Given a surface normal and an incoming direction, determine the direction in which the path continues.

normal [numpy.ndarray_like of shape (3,)] the surface normal at the intersection point

incoming_direction [numpy.ndarray_like of shape (3,)] the direction from which light hits the surface

Returns **outgoing direction** – the direction in which light is reflected from the surface

Return type numpy.ndarray_like of shape (3, 0)

class padvinder.material.**Lambert**(*color=(0.5*, *0.5*, *0.5)*, *diffuse=1*)

diffuse
Returns the diffuse value of the material.

__call__(*surface_normal*, *incoming_color*, *incoming_direction*, *outgoing_direction*)
Calculate light reflected from the material toward the outgoing direction. Keep in mind, while pathtracing starts at the camera and heads into the scene, the rays contribution is accumulated 'backwards'. Therefore the incoming direction is further down the path and outgoing_direction is closer towards the camera.

Parameters

- **surface_normal** (*numpy.ndarray_like*) – normal vector at the geometries surface
- **incoming_color** (*numpy.ndarray_like*) – the color the ray has accumulated up to this point

- **incoming_direction** (*numpy.ndarray_like*) – the direction from where the 'light shines' onto the surface
- **outgoing_direction** (*numpy.ndarray_like*) – the direction into which the 'light gets reflected' from the surface

Returns **color** – the light color 'getting reflected' from the surface

Return type numpy.ndarray_like

color
Returns the color of the material.

1.4 Geometry

Module collecting a number of renderable objects. Geometry is an abstract base class defining the interface and Sphere and Plane are concrete, renderable implementatons.

class padvinder.geometry.**Geometry**(*material=Material(color=[0.5 0.5 0.5])*)
Bases: object

Baseclass for geometry. Either implicitly (eg. spheres and planes) or explicitly via triangles.

Parameters **material** (padvinder.material.Material) – A material specifies how the geometry surface interacts with light rays

Examples

```
>>> Geometry()
Geometry(Material(color=[0.5, 0.5, 0.5]))
```

material
Returns the material of this geometry instance.

intersect(*ray*)
Given a ray, intersect it with this geometry instance and returns the distance t of ray position to intersection point (so that ray.point(t) is the intersection point) If no intersection occurs +inf is returned.

Parameters **ray** (Ray) – the ray to be tested for intersection

Returns in (0, +inf]

Return type float

Raises NotImplemented – because this is an abstract base class

Examples

```
>>> a = Sphere()
>>> r = Ray()
>>> a.intersect(r)
1.0
```

normal(*x*)
Given a point on the surface of the geometry instance and returns the surface normal at that point.

Parameters **x** (*numpy.ndarray_like*) – point on the geometry instance

Returns **n** – normal vector of the geometry surface at this point

Return type numpy.ndarray_like

Raises `NotImplemented` – because this is an abstract base class

class padvinder.geometry.**Sphere**(*material=Material(color=[0.5 0.5 0.5])*, *position=(0, 0, 0)*, *radius=1*)

Bases: *padvinder.geometry.Geometry*

An implicitly modeled sphere is given by: LA.norm(position - x) - r = 0, where position is the center of the sphere, x is a point on the surface of the sphere and r is the radius.

Parameters

- **material** (padvinder.material.Material) – A material specifies how the geometry surface interacts with light rays
- **position** (*numpy.ndarray_like*) – position of the sphere's center in world coordinates
- **radius** (*number*) – radius of the sphere

Examples

```
>>> Sphere() #unitsphere
Sphere(Material(color=[0.5, 0.5, 0.5]),
               position=[0.0, 0.0, 0.0],
               radius=1)
```

position

Returns the position of the center of the sphere.

radius

Returns the radius of the sphere.

intersect(*ray*)

Given a ray, intersect it with this sphere instance and returns the distance t of ray position to intersection point (so that ray.get_point(t) is the intersection point) If no intersection occurs +inf is returned.

Parameters **ray** (Ray) – the ray to be tested for intersections

Returns number in (0, +inf]

Return type float

normal(*x*)

Given a point on the surface of the sphere instance and returns the surface normal at that point.

Parameters **x** (*numpy.ndarray_like*) – point on the geometry instance

Returns **normal** – normal vector of the geometry surface at this point

Return type numpy.ndarray_like

class padvinder.geometry.**Plane**(*material=Material(color=[0.5 0.5 0.5])*, *position=(0, 0, 0)*, *normal=(0, 1, 0)*)

Bases: *padvinder.geometry.Geometry*

An implicitly modelled plane is given by n * x - d = 0, where n is the normal vector, x is a point in world coordinates, d is a number and n * x is the dot product of two vectors.

Parameters

- **material** (padvinder.material.Material) – material instance
- **position** (*numpy.ndarray_like*) – the 'origin' of the plane - any point in the world the plane passes through
- **normal** (*numpy.ndarray_like*) – the normalised vector thats orthogonal to the plane

Examples

```
>>> Plane()    # equivalent to ...
>>> Plane(Material(), (0, 0, 0), (1, 0, 0))
Plane(Material(color=[1., 1., 1.]), position=(0, 0, 0), normal=(0, 1, 0))
```

position
Returns the position of the plane.

intersect(*ray*)
Given a ray Returns the value t so that ray.get_point(t) is the closest intersection point or +inf if the plane is not hit.

Parameters **ray** (Ray) – the ray to be tested for intersections

Returns

Return type number in (0, +inf]

Examples

```
>>> a = plane()
>>> r = ray()
>>> a.intersect(r)
1.0
```

normal(*x*)
Given a point on the surface of the plane, returns the surface normal at that point.

Parameters **x** (*numpy.ndarray_like*) – point on the plane instance

Returns **normal** – normal vector of the plane surface at this point

Return type numpy.ndarray_like

1.5 Scene

A Scene is a collection of renderables and performs intersection tests on every contained object.

class padvinder.scene.**Scene**(**renderable*)
Bases: object

A scene contains a collection of renderable objects and performs ray intersections on them. Eventually the distance to the intersection point and the intersected object are returned. If no renderable was intersected (np.inf, None) is returned.

Parameters **renderable** (padvinder.geometry.Geometry) – and subclasses. A renderable has to implement the intersect(ray) method

intersect(*ray*)
Performs intersection tests with every renderable in the scene.

Parameters **ray** (Ray) – the light ray to trace through the scene has to support ray.position and ray.direction

Returns **(number, renderable)** – Number is a float in the intervall of [0, np.inf] and corresponds to the distance along the ray to the intersection point on the renderable surface. The renderable is an object previously passed into the scene that was intersected by the ray. If multiple renderables are intersected in the Scene, the one with the shortest distance between intersection point and ray position is returned. If no intersection occured **(np.inf, None)** is returned.

Return type (float, *padvinder.geometry.Geometry*)

1.6 Main

1.7 Utilities

Utilities contains a number of frequently used functions.

padvinder.util.**normalize**(*array*)
Returns a normalized version of the provided array.

Parameters **array** (*numpy.ndarray_like*) – The array to be normalized. Afterwards np.linalg.norm(normalize(array)) is approximately equal to 1.

Returns **normalized array** – the array normalized to unit length: np.linalg.norm(normalize(array)) ~= 1.

Return type numpy.ndarray_like

Raises

- ValueError – if the input array contains Inf's or Nan's
- ZeroDivisionError – if the length of the provided array has length ofzero the division will cause a ZeroDivisionError to be raised

Examples

```
>>> normalize([1, 0, 0])
[1.0, 0.0, 0.0]
>>> normalize([2, 4, 4])
[0.33333333, 0.66666667, 0.66666667]
>>> normalize(np.array((3,4,5)))
[0.42426407, 0.56568542, 0.70710678]
### np.linalg.norm(normalize((x, y, z))) ~= 1
```

padvinder.util.**check_finite**(**args*)
Validate the input parameters and raise ValueErrors if any contains incompatible values (Infs or NaNs) are present.

Parameters **args** (*numpy.ndarray_like*) – a list of lists or arrays

Raises ValueError – if any passed in element is Inf or NaN.

CHAPTER 2

Padvinder Test

2.1 Test Ray

class padvinder.test.test_ray.**TestRay**(*methodName='runTest'*)
Bases: unittest.case.TestCase

test_default_construction()
Test if the ray is constructed with the expected default parameters.

test_input()
Test if the input values are checked correctly. Values that do not validate the checks are omitted because they are covered by the remaining tests.

test_point()
Test if a point with given distance along the ray is calculated correctly.

test_string()
Test if the string representation of the ray is correct. Because testing against a concrete string is tough if numpy changes how they print arrays - we will just test if the call succedes.

test_hypothesis() → None
Test the ray's invariants with 'random' input from hypothesis

2.2 Test Camera

class padvinder.test.test_camera.**TestCamera**(*methodName='runTest'*)
Bases: unittest.case.TestCase

test_default_construction()
Test if the camera is constructed with the expected default parameters.

test_custom_construction()
Test if the camera is constructed correctly with the non-default parameters.

test_invalid_construction()
Test if the camera construction fails as expected on invalid input

test_representation()
Test if the camera class is capable of printing itself.

class padvinder.test.test_camera.**TestPerspectiveCamera**(*methodName='runTest'*)
Bases: unittest.case.TestCase

test_default_construction()
Test if the perspective camera is constructed with the expected default parameters.

test_custom_construction()
Test if the perspective camera is constructed correctly with the non-default parameters.

test_invalid_construction()
Test if the camera construction fails as expected on invalid input

test_ray()
Test if the initial rays are calculated correctly - Beware that the indexing is following numpy's convention: x is vertical & y is horizontal.

test_representation()
Test if the perspective camera class is capable of printing itself.

2.3 Test Material

class padvinder.test.test_material.**TestMaterial**(*methodName='runTest'*)
Bases: unittest.case.TestCase

test_default_construction()
Test if the material is constructed with the expected default parameters.

test_non_default_construction()
Test if the material is constructed correctly with non-default parameters.

test_outgoing_direction() → None

test_representation()
Test if the string representation of the material is correct. Because testing against a concrete string is tough if numpy changes how they print arrays, we will just test if the call succedes.

class padvinder.test.test_material.**TestEmission**(*methodName='runTest'*)
Bases: unittest.case.TestCase

test_default_construction()
Test if the material is constructed with the expected default parameters.

test_non_default_construction()
Test if the material is constructed correctly with non-default parameters.

test_representation()
Test if the string representation of the emission is correct. Because testing against a concrete string is tough if numpy changes how they print arrays, we will just test if the call succedes.

class padvinder.test.test_material.**TestLambert**(*methodName='runTest'*)
Bases: unittest.case.TestCase

test_default_construction()
Test if the material is constructed with the expected default parameters.

test_non_default_construction()
Test if the material is constructed correctly with non-default parameters.

test_representation()
Test if the string representation of the material is correct. Because testing against a concrete string is tough if numpy changes how they print arrays, we will just test if the call succedes.

2.4 Test Geometry

class padvinder.test.test_geometry.**TestGeometry**(*methodName='runTest'*)
Bases: unittest.case.TestCase

test_default_construction()
Test if the geometry is constructed with the expected default parameters.

test_non_default_construction()
Test if the geometry is constructed correctly with the non-default parameters.

test_representation()
Test if the geometry class is capable of printing itself.

class padvinder.test.test_geometry.**TestSphere**(*methodName='runTest'*)
Bases: unittest.case.TestCase

test_default_construction()
Test if the sphere is constructed correctly with non-default parameters.

test_non_default_construction()
Test if the sphere is constructed correctly with the expected default parameters.

test_intersect()
Test if the ray-sphere intersection works as expected.

test_normal()
Test if the sphere normal is calculated correctly.

test_representation()
Test if the sphere class is capable of printing itself.

class padvinder.test.test_geometry.**TestPlane**(*material=Material(color=[0.5 0.5 0.5])*)
Bases: *padvinder.geometry.Geometry*

test_default_construction()
Test if the plane is constructed correctly with non-default parameters.

test_non_default_construction()
Test if the plane is constructed correctly with the expected default parameters.

test_intersect()
Test if the ray-plane intersection works as expected.

test_normal()
Test if the plane normal calculated correctly.

test_representation()
Test if the plane class is capable of printing itself.

2.5 Test Scene

class padvinder.test.test_scene.**TestScene**(*methodName='runTest'*)
Bases: unittest.case.TestCase

setUp()

test_iterator()
Test if one can iterate over all contained objects given a scene

test_intersection()
Test if a scene performs the ray-object intersection with all contained renderables correctly.

2.6 Test Utilities

class padvinder.test.test_util.**TestNormalize**(*methodName='runTest'*)
Bases: unittest.case.TestCase

test_invalid_examples()
Test if normalize rejects invalid parameters as expected.

test_hypothesis() → None
Test normalize's invariants with 'random' input from hypothesis.

class padvinder.test.test_util.**TestCheckFinite**(*methodName='runTest'*)
Bases: unittest.case.TestCase

test_with_finite_values()

test_with_infinite_values()

CHAPTER 3

Padvinder

Padvinder is a little pathtracing renderer written in Python. Originally I was keen to implement my own pathtracer. The focus has always been on quick coding and learning rather than on code performance. Over the time it developed into an exercise for good coding practices like test-driven-development, continuous integration and documentation.

Padvinder is meant to be a dutchly flavoured word for pathtracing; encoding the algorithm used for rendering with Guido van Rossum's dutch origins.

CHAPTER 4

Example Usage

Run an example rendering from the root of the repository via:

```
python -m padvinder
```

I find this to be cleaner and shorter than having a main.py file. It makes use of Python's ability to execute modules. Under the hood __main__.py is executed.

CHAPTER 5

Tests

Run the tests similarly to running the example via:

```
python -m padvinder.test
```

Again, this executes the __main__.py file in the test module while keeping the command line concise.

CHAPTER 6

Continuous Integration

Thanks to Travis CI . Find the build status at Padvinder on Travis.

CHAPTER 7

Documentation

Thanks to Read the Docs . Find the documentation at Padvinder on Read the Docs.

CHAPTER 8

Indices and tables

- genindex
- modindex
- search

Python Module Index

p

Index

S

T

U