Pyrr Documentation

Release 0.10.0

Adam Griffiths

Mar 13, 2019

Contents

Pyrr is a Python mathematical library.

CHAPTER 1

Overview

1.1 Data types

1.1.1 Modules

Each data type resides in its own module named after the specified type.

Where there may be multiple versions of a data type, each version is located in its own module.

Functions which are able to handle all versions of a data type are located in the central module. Otherwise, functions reside in the specific data type's module.

For example:

- **vector3.py** Provides functions for creating and manipulating 3D vectors (x,y,z).
- **vector4.py** Provides functions for creating and manipulating 4D vectors (x,y,z,w).
- **vector.py** Provides functions that work with both 3D and 4D vectors.

1.1.2 Conversion

Data conversion functions are provided in the module of the type being converted to.

For example:

```
# module matrix44.py
def create_from_matrix33(mat) :
    pass

def create_from_eulers(eulers):
    pass

def create_from_quaternion(quat):
    pass
```

1.2 Geometric Tests

1.2.1 Naming Scheme

Geometric Tests, also called Collision Detection, is provided by the geometric_tests module.

Functions match a specific naming scheme. The function name provides information on what types are being checked, and what check is being performed.

Function names adhere to the following format:

```
def <type>_<check type>_<type>( ... ):
    pass
```

The order of parameters matches the order of types given in the function name.

The following are examples of this naming scheme:

```
def point_intersect_line(point, line):
    """This function returns the intersection point as a vector or None if there is␣
↪no intersection.
    """
    pass

def point_closest_point_on_line_segment(point, segment):
    """The function returns the closest on the line segment, to the given point.
    """
    pass
```

1.2.2 Types of Checks

Below are some of the types of checks provided. These are the names used by the functions and their meaning.

- **intersect** Returns the intersection point of the two types or None.
- **does_intersect** Returns True if the types are intersecting.
- **height_above** Returns the height of one type above another.
- **closest_point_on** Returns the closest point on the second type.
- **parallel** Returns True if the types are parallel to each other.
- **coincident** Returns True if the types are not only parallel, but are along the same direction axis.
- **penetration** Returns the penetration distance of one type into the second.

There may be more checks provided by the module than are listed here.

CHAPTER 2

Object Oriented API

2.1 Matrices

2.1.1 Matrix33

Represents a 3x3 Matrix.

The Matrix33 class provides a number of convenient functions and conversions.

```
import numpy as np
from pyrr import Quaternion, Matrix33, Matrix44, Vector3

m = Matrix33()
m = Matrix33([[1.,0.,0.],[0.,1.,0.],[0.,0.,1.]])

# copy constructor
m = Matrix44(Matrix44())

# explicit creation
m = Matrix33.identity()
m = Matrix33.from_matrix44(Matrix44())

# inferred conversions
m = Matrix33(Quaternion())
m = Matrix33(Matrix44())

# multiply matricies together
m = Matrix33() * Matrix33()
m = Matrix33() * Matrix44()

# extract a quaternion from a matrix
q = m.quaternion

# convert from quaternion back to matrix
```

(continues on next page)

(continued from previous page)

```
m = q.matrix33
m = Matrix33(q)

# rotate a matrix by a quaternion
m = Matrix33.identity() * Quaternion()

# rotate a vector 3 by a matrix
v = Matrix33.from_x_rotation(np.pi) * Vector3([1.,2.,3.])

# undo a rotation
m = Matrix33.from_x_rotation(np.pi)
v = m * Vector3([1.,1.,1.])
# ~m is the same as m.inverse
v = ~m * v

# access specific parts of the matrix
# first row
m1 = m.m1
# first element, first row
m11 = m.m11
# third element, third row
m33 = m.m33
# first row, same as m1
r1 = m.r1
# first column
c1 = m.c1
```

class pyrr.objects.matrix33.**Matrix33**

c1

c2

c3

classmethod from_matrix44(*matrix*, *dtype=None*)

Creates a Matrix33 from a Matrix44.

The Matrix44 translation will be lost.

m1

m11

m12

m13

m2

m21

m22

m23

m3

m31

m32

m33

matrix33
Returns the Matrix33.

This can be handy if you're not sure what type of Matrix class you have but require a Matrix33.

matrix44
Returns a Matrix44 representing this matrix.

quaternion
Returns a Quaternion representing this matrix.

r1

r2

r3

2.1.2 Matrix44

Represents a 4x4 Matrix.

The Matrix44 class provides a number of convenient functions and conversions.

```
import numpy as np
from pyrr import Quaternion, Matrix33, Matrix44, Vector4

m = Matrix44()
m = Matrix44([[1.,0.,0.,0.],[0.,1.,0.,0.],[0.,0.,1.,0.],[0.,0.,0.,1.]])

# copy constructor
m = Matrix44(Matrix44())

# explicit creation
m = Matrix44.identity()
m = Matrix44.from_matrix44(Matrix44())

# inferred conversions
m = Matrix44(Quaternion())
m = Matrix44(Matrix33())

# multiply matricies together
m = Matrix44() * Matrix44()

# extract a quaternion from a matrix
q = m.quaternion

# convert from quaternion back to matrix
m = q.matrix44
m = Matrix44(q)

# rotate a matrix by a quaternion
m = Matrix44.identity() * Quaternion()

# rotate a vector 4 by a matrix
v = Matrix44.from_x_rotation(np.pi) * Vector4([1.,2.,3.,1.])

# undo a rotation
```

(continues on next page)

(continued from previous page)

```
m = Matrix44.from_x_rotation(np.pi)
v = m * Vector4([1.,1.,1.,1.])
# ~m is the same as m.inverse
v = ~m * v

# access specific parts of the matrix
# first row
m1 = m.m1
# first element, first row
m11 = m.m11
# fourth element, fourth row
m44 = m.m44
# first row, same as m1
r1 = m.r1
# first column
c1 = m.c1
```

class pyrr.objects.matrix44.**Matrix44**

c1

c2

c3

c4

decompose()

Decomposes an affine transformation matrix into its scale, rotation and translation components.

Parameters **m** (*numpy.array*) – A matrix.

Returns tuple (scale, rotation, translation) Vector3 scale Quaternion rotation Vector3 translation

classmethod from_matrix33(*matrix*, *dtype=None*)

Creates a Matrix44 from a Matrix33.

classmethod from_translation(*translation*, *dtype=None*)

Creates a Matrix44 from the specified translation.

classmethod look_at(*eye*, *target*, *up*, *dtype=None*)

Creates a Matrix44 for use as a lookAt matrix.

m1

m11

m12

m13

m14

m2

m21

m22

m23

m24

m3

m31

m32

m33

m34

m4

m41

m42

m43

m44

matrix33
Returns a Matrix33 representing this matrix.

matrix44
Returns the Matrix44.

This can be handy if you're not sure what type of Matrix class you have but require a Matrix44.

classmethod orthogonal_projection(*left*, *right*, *top*, *bottom*, *near*, *far*, *dtype=None*)
Creates a Matrix44 for use as an orthogonal projection matrix.

classmethod perspective_projection(*fovy*, *aspect*, *near*, *far*, *dtype=None*)
Creates a Matrix44 for use as a perspective projection matrix.

classmethod perspective_projection_bounds(*left*, *right*, *top*, *bottom*, *near*, *far*, *dtype=None*)
Creates a Matrix44 for use as a perspective projection matrix.

quaternion
Returns a Quaternion representing this matrix.

r1

r2

r3

r4

2.2 Quaternion

2.2.1 Quaternion

Represents a Quaternion rotation.

The Quaternion class provides a number of convenient functions and conversions.

```
import numpy as np
from pyrr import Quaternion, Matrix33, Matrix44, Vector3, Vector4

q = Quaternion()
```

(continues on next page)

(continued from previous page)

```
# explicit creation
q = Quaternion.from_x_rotation(np.pi / 2.0)
q = Quaternion.from_matrix(Matrix33.identity())
q = Quaternion.from_matrix(Matrix44.identity())

# inferred conversions
q = Quaternion(Quaternion())
q = Quaternion(Matrix33.identity())
q = Quaternion(Matrix44.identity())

# apply one quaternion to another
q1 = Quaternion.from_y_rotation(np.pi / 2.0)
q2 = Quaternion.from_x_rotation(np.pi / 2.0)
q3 = q1 * q2

# extract a matrix from the quaternion
m33 = q3.matrix33
m44 = q3.matrix44

# convert from matrix back to quaternion
q4 = Quaternion(m44)

# rotate a quaternion by a matrix
q = Quaternion() * Matrix33.identity()
q = Quaternion() * Matrix44.identity()

# apply quaternion to a vector
v3 = Quaternion() * Vector3()
v4 = Quaternion() * Vector4()

# undo a rotation
q = Quaternion.from_x_rotation(np.pi / 2.0)
v = q * Vector3([1.,1.,1.])
# ~q is the same as q.conjugate
original = ~q * v
assert np.allclose(original, v)

# get the dot product of 2 Quaternions
dot = Quaternion() | Quaternion.from_x_rotation(np.pi / 2.0)
```

class pyrr.objects.quaternion.**Quaternion**

angle
Returns the angle around the axis of rotation of this Quaternion as a float.

axis
Returns the axis of rotation of this Quaternion as a Vector3.

conjugate
Returns the conjugate of this Quaternion.

This is a Quaternion with the opposite rotation.

cross(*other*)
Returns the cross of this Quaternion and another.

This is the equivalent of combining Quaternion rotations (like Matrix multiplication).

dot(*other*)
Returns the dot of this Quaternion and another.

exp()
Returns a new Quaternion representing the exponentional of this Quaternion

classmethod from_axis(*axis*, *dtype=None*)
Creates a new Quaternion from an axis with angle magnitude.

classmethod from_axis_rotation(*axis*, *theta*, *dtype=None*)
Creates a new Quaternion with a rotation around the specified axis.

classmethod from_eulers(*eulers*, *dtype=None*)
Creates a Quaternion from the specified Euler angles.

classmethod from_inverse_of_eulers(*eulers*, *dtype=None*)
Creates a Quaternion from the inverse of the specified Euler angles.

classmethod from_matrix(*matrix*, *dtype=None*)
Creates a Quaternion from the specified Matrix (Matrix33 or Matrix44).

classmethod from_x_rotation(*theta*, *dtype=None*)
Creates a new Quaternion with a rotation around the X-axis.

classmethod from_y_rotation(*theta*, *dtype=None*)
Creates a new Quaternion with a rotation around the Y-axis.

classmethod from_z_rotation(*theta*, *dtype=None*)
Creates a new Quaternion with a rotation around the Z-axis.

inverse
Returns the inverse of this quaternion.

is_identity
Returns True if the Quaternion has no rotation (0.,0.,0.,1.).

length
Returns the length of this Quaternion.

lerp(*other*, *t*)
Interpolates between quat1 and quat2 by t. The parameter t is clamped to the range [0, 1]

matrix33
Returns a Matrix33 representation of this Quaternion.

matrix44
Returns a Matrix44 representation of this Quaternion.

negative
Returns the negative of the Quaternion.

normalise()
normalizes this Quaternion in-place.

normalised
Returns a normalized version of this Quaternion as a new Quaternion.

normalize()
normalizes this Quaternion in-place.

normalized
Returns a normalized version of this Quaternion as a new Quaternion.

power (*exponent*)
Returns a new Quaternion representing this Quaternion to the power of the exponent.

slerp (*other*, *t*)
Spherically interpolates between quat1 and quat2 by t. The parameter t is clamped to the range [0, 1]

w

x

xw

xy

xyw

xyz

xyzw

xz

xzw

y

z

2.3 Vectors

2.3.1 Vector3

Represents a 3 dimensional Vector.

The Vector3 class provides a number of convenient functions and conversions.

```
import numpy as np
from pyrr import Quaternion, Matrix33, Matrix44, Vector3

v = Vector3()
v = Vector3([1.,2.,3.])

# copy constructor
v = Vector3(Vector3())

# add / subtract vectors
v = Vector3([1.,2.,3.]) + Vector3([4.,5.,6.])

# rotate a vector by a Matrix
v = Matrix33.identity() * Vector3([1.,2.,3.])
v = Matrix44.identity() * Vector3([1.,2.,3.])

# rotate a vector by a Quaternion
v = Quaternion() * Vector3([1.,2.,3.])

# get the dot-product of 2 vectors
d = Vector3([1.,0.,0.]) | Vector3([0.,1.,0.])

# get the cross-product of 2 vectors
x = Vector3([1.,0.,0.]) ^ Vector3([0.,1.,0.])
```

(continues on next page)

(continued from previous page)

```
# access specific parts of the vector
# x value
x,y,z = v.x, v.y, v.z

# access groups of values as np.ndarray's
xy = v.xy
xz = v.xz
xyz = v.xyz
```

class pyrr.objects.vector3.**Vector3**

classmethod from_vector4(*vector*, *dtype=None*)
Create a Vector3 from a Vector4.

Returns the Vector3 and the W component as a tuple.

inverse
Returns the opposite of this vector.

vector3

x

xy

xyz

xz

y

z

2.3.2 Vector4

Represents a 4 dimensional Vector.

The Vector4 class provides a number of convenient functions and conversions.

```
import numpy as np
from pyrr import Quaternion, Matrix33, Matrix44, Vector4

v = Vector4()
v = Vector4([1.,2.,3.])

# explicit creation
v = Vector4.from_vector3(Vector3([1.,2.,3.]), w=1.0)

# copy constructor
v = Vector4(Vector4())

# add / subtract vectors
v = Vector4([1.,2.,3.,4.]) + Vector4([4.,5.,6.,7.])

# rotate a vector by a Matrix
v = Matrix44.identity() * Vector4([1.,2.,3.,4.])
```

(continues on next page)

(continued from previous page)

```
# rotate a vector by a Quaternion
v = Quaternion() * Vector4([1.,2.,3.,4.])

# get the dot-product of 2 vectors
d = Vector4([1.,0.,0.,0.]) | Vector4([0.,1.,0.,0.])

# access specific parts of the vector
# x value
x,y,z,w = v.x, v.y, v.z, v.w

# access groups of values as np.ndarray's
xy = v.xy
xyz = v.xyz
xyzw = v.xyzw
xz = v.xz
xw = v.xw
xyw = v.xyw
xzw = v.xzw
```

class pyrr.objects.vector4.**Vector4**

classmethod from_vector3(*vector*, *w=0.0*, *dtype=None*)
Create a Vector4 from a Vector3.

By default, the W value is 0.0.

inverse
Returns the opposite of this vector.

vector3
Returns a Vector3 and the W component as a tuple.

w

x

xw

xy

xyw

xyz

xyzw

xz

xzw

y

z

CHAPTER 3

Procedural API

3.1 Axis Aligned Bounding Box

3.1.1 AABB

Provides functions to calculate and manipulate Axis-Aligned Bounding Boxes (AABB).

AABB are a simple 3D rectangle with no orientation. It is up to the user to provide translation.

An AABB is represented by an array of 2 x 3D vectors. The first vector represents the minimum extent. The second vector represents the maximum extent.

It should be noted that rotating the object within an AABB will invalidate the AABB. It is up to the user to either:

- recalculate the AABB.
- use an AAMBB instead.

TODO: add transform(matrix)

pyrr.aabb.**add_aabbs**(**args*, ***kwargs*)
Extend an AABB to encompass a list of other AABBs.

pyrr.aabb.**add_points**(**args*, ***kwargs*)
Extends an AABB to encompass a list of points.

pyrr.aabb.**centre_point**(**args*, ***kwargs*)
Returns the centre point of the AABB.

pyrr.aabb.**clamp_points**(**args*, ***kwargs*)
Takes a list of points and modifies them to fit within the AABB.

pyrr.aabb.**create_from_aabbs**(**args*, ***kwargs*)
Creates an AABB from a list of existing AABBs.

AABBs must be a 2D list. Ie::

numpy.array([AABB, AABB,])

pyrr.aabb.**create_from_bounds**(**args*, ***kwargs*)
Creates an AABB using the specified minimum and maximum values.

pyrr.aabb.**create_from_points**(**args*, ***kwargs*)
Creates an AABB from the list of specified points.

Points must be a 2D list. Ie::

numpy.array([[x, y, z], [x, y, z],])

pyrr.aabb.**create_zeros**(*dtype=None*)

class pyrr.aabb.**index**

maximum = 1

minimum = 0

pyrr.aabb.**maximum**(**args*, ***kwargs*)
Returns the maximum point of the AABB.

pyrr.aabb.**minimum**(**args*, ***kwargs*)
Returns the minimum point of the AABB.

3.1.2 AAMBB

Provides functions to calculate and manipulate Axis-Aligned Minimum Bounding Boxes (AAMBB).

AAMBB are a simple 3D rectangle with no orientation. It is up to the user to provide translation. AAMBB differ from AABB in that they allow for the content to rotate freely and still be within the AAMBB.

An AAMBB is represented in the same way an AABB is; a array of 2 x 3D vectors. The first vector represents the minimum extent. The second vector represents the maximum extent.

Note that because the AAMBB set's it's dimensions using the vector length of any points set within it, the user should be careful to avoid adding the AAMBB to itself or the AAMBB will continue to grow.

TODO: add transform(matrix) TODO: add point_within_aabb TODO: use point_within_aabb for unit tests

pyrr.aambb.**add_aabbs**(**args*, ***kwargs*)
Extend an AAMBB to encompass a list of other AABBs or AAMBBs.

It should be noted that this ensures that the encompassed AABBs can rotate freely. Using the AAMBB itself in this calculation will create an event bigger AAMBB.

pyrr.aambb.**add_points**(**args*, ***kwargs*)
Extends an AAMBB to encompass a list of points.

It should be noted that this ensures that the encompassed points can rotate freely. Calling this using the min / max points from the AAMBB will create an even bigger AAMBB.

pyrr.aambb.**centre_point**(*bb*)
Returns the centre point of the AABB. This should always be [0.0, 0.0, 0.0]

pyrr.aambb.**clamp_points**(*bb*, *points*)
Takes a list of points and modifies them to fit within the AABB.

pyrr.aambb.**create_from_aabbs**(*aabbs*, *dtype=None*)
Creates an AAMBB from a list of existing AABBs.

AABBs must be a 2D list. Ie::

numpy.array([AABB, AABB,])

pyrr.aambb.**create_from_bounds**(**args*, ***kwargs*)
Creates an AAMBB using the specified minimum and maximum values.

pyrr.aambb.**create_from_points**(**args*, ***kwargs*)
Creates an AAMBB from the list of specified points.

Points must be a 2D list. Ie::

numpy.array([[x, y, z], [x, y, z],])

pyrr.aambb.**create_zeros**(*dtype=None*)

class pyrr.aambb.**index**

maximum = 1

minimum = 0

pyrr.aambb.**maximum**(*bb*)
Returns the maximum point of the AABB.

pyrr.aambb.**minimum**(*bb*)
Returns the minimum point of the AABB.

3.2 Euler

Provide functions for the creation and manipulation of Euler angles.

Eulers represent 3 rotations: Pitch, Roll and Yaw.

Eulers are represented using a numpy.array of shape (3,).

pyrr.euler.**create**(*roll=0.0*, *pitch=0.0*, *yaw=0.0*, *dtype=None*)
Creates an array storing the specified euler angles.

Input values are in radians.

Parameters

- **pitch** (*float*) – The pitch in radians.
- **roll** (*float*) – The roll in radians.
- **yaw** (*float*) – The yaw in radians.

Return type numpy.array

pyrr.euler.**create_from_x_rotation**(*theta*, *dtype=None*)

pyrr.euler.**create_from_y_rotation**(*theta*, *dtype=None*)

pyrr.euler.**create_from_z_rotation**(*theta*, *dtype=None*)

class pyrr.euler.**index**
Defines the indices used to store the Euler values in the numpy array.

pitch = 1

roll = 0

yaw = 2

pyrr.euler.**pitch**(*eulers*)
Extracts the pitch value from the euler.

Return type float.

pyrr.euler.**roll**(*eulers*)
Extracts the roll value from the euler.

Return type float.

pyrr.euler.**yaw**(*eulers*)
Extracts the yaw value from the euler.

Return type float.

3.3 Geometric Tests

Defines a number of functions to test interactions between various forms data types.

pyrr.geometric_tests.**point_closest_point_on_line**(**args*, ***kwargs*)
Calculates the point on the line that is closest to the specified point.

Parameters

- **point** (*numpy.array*) – The point to check with.
- **line** (*numpy.array*) – The line to check against.

Return type numpy.array

Returns The closest point on the line to the point.

pyrr.geometric_tests.**point_closest_point_on_line_segment**(**args*, ***kwargs*)
Calculates the point on the line segment that is closest to the specified point.

This is similar to point_closest_point_on_line, except this is against the line segment of finite length. Whereas point_closest_point_on_line checks against a line of infinite length.

Parameters

- **point** (*numpy.array*) – The point to check with.
- **line_segment** (*numpy.array*) – The finite line segment to check against.

Return type numpy.array

Returns The closest point on the line segment to the point.

pyrr.geometric_tests.**point_closest_point_on_plane**(**args*, ***kwargs*)
Calculates the point on a plane that is closest to a point.

Parameters

- **point** (*numpy.array*) – The point to check with.
- **plane** (*numpy.array*) – The infinite plane to check against.

Return type numpy.array

Returns The closest point on the plane to the point.

pyrr.geometric_tests.**point_closest_point_on_ray**(**args*, ***kwargs*)
Calculates the point on a ray that is closest to a point.

Parameters

- **point** (*numpy.array*) – The point to check with.
- **ray** (*numpy.array*) – The ray to check against.

Return type numpy.array

Returns The closest point on the ray to the point.

pyrr.geometric_tests.**point_height_above_plane**(**args*, ***kwargs*)
Calculates how high a point is above a plane.

Parameters

- **point** (*numpy.array*) – The point to check.
- **plane** (*numpy.array*) – The plane to check.

Return type float

Returns The height above the plane as a float. The value will be negative if the point is behind the plane.

pyrr.geometric_tests.**point_intersect_line**(**args*, ***kwargs*)
Calculates the intersection point of a point and aline.

Performed by checking if the cross-product of the point relative to the line is 0.

pyrr.geometric_tests.**point_intersect_line_segment**(**args*, ***kwargs*)
Calculates the intersection point of a point and a line segment.

Performed by checking if the cross-product of the point relative to the line is 0 and if the dot product of the point relative to the line start AND the end point relative to the line start is less than the segment's squared length.

pyrr.geometric_tests.**point_intersect_rectangle**(**args*, ***kwargs*)
Calculates the intersection point of a point and a 2D rectangle.

For 3D points, the Z axis will be ignored.

Returns Returns True if the point is touching

or within the rectangle.

pyrr.geometric_tests.**ray_coincident_ray**(**args*, ***kwargs*)
Check if rays are coincident.

Rays must not only be parallel to each other, but reside along the same vector.

Parameters **ray1, ray2** (*numpy.array*) – The rays to check.

Return type boolean

Returns Returns True if the two rays are co-incident.

pyrr.geometric_tests.**ray_intersect_aabb**(**args*, ***kwargs*)
Calculates the intersection point of a ray and an AABB

Parameters

- **ray1** (*numpy.array*) – The ray to check.
- **aabb** (*numpy.array*) – The Axis-Aligned Bounding Box to check against.

Return type numpy.array

Returns Returns a vector if an intersection occurs. Returns None if no intersection occurs.

pyrr.geometric_tests.**ray_intersect_plane**(**args*, ***kwargs*)
Calculates the intersection point of a ray and a plane.

Parameters

- **ray** (*numpy.array*) – The ray to test for intersection.

- **plane** (*numpy.array*) – The ray to test for intersection.
- **front_only** (*boolean*) – Specifies if the ray should

only hit the front of the plane. Collisions from the rear of the plane will be ignored.

:return The intersection point, or None if the ray is parallel to the plane. Returns None if the ray intersects the back of the plane and front_only is True.

pyrr.geometric_tests.**ray_parallel_ray**(**args*, ***kwargs*)
Checks if two rays are parallel.

Parameters **ray1, ray2** (*numpy.array*) – The rays to check.

Return type boolean

Returns Returns True if the two rays are parallel.

pyrr.geometric_tests.**sphere_does_intersect_sphere**(**args*, ***kwargs*)
Checks if two spheres overlap.

Note: This will return True if the two spheres are touching perfectly but sphere_penetration_sphere will return 0.0 as the touch but don't penetrate.

This is faster than circle_penetrate_amount_circle as it avoids a square root calculation.

Parameters

- **s1** (*numpy.array*) – The first circle.
- **s2** (*numpy.array*) – The second circle.

Return type boolean

Returns Returns True if the circles overlap. Otherwise, returns False.

pyrr.geometric_tests.**sphere_penetration_sphere**(**args*, ***kwargs*)
Calculates the distance two spheres have penetrated into one another.

Parameters

- **s1** (*numpy.array*) – The first circle.
- **s2** (*numpy.array*) – The second circle.

Return type float

Returns The total overlap of the two spheres. This is essentially: r1 + r2 - distance Where r1 and r2 are the radii of circle 1 and 2 and distance is the length of the vector p2 - p1. Will return 0.0 if the circles do not overlap.

pyrr.geometric_tests.**vector_parallel_vector**(**args*, ***kwargs*)
Checks if two vectors are parallel.

Parameters **v1, v2** (*numpy.array*) – The vectors to check.

Return type boolean

Returns Returns True if the two vectors are parallel.

3.4 Geometry

Geometry functions.

pyrr.geometry.**create_cube**(*scale=(1.0, 1.0, 1.0)*, *st=False*, *rgba=False*, *dtype='float32'*, *type='triangles'*)

Returns a Cube reading for rendering.

Output is a tuple of numpy arrays. The first value is the vertex data, the second is the indices.

The first dimension of the vertex data contains the list of vertices. The second dimension is the vertex data.

Vertex data is always in the following order:

```
[x, y, z, s, t, r, g, b, a]
```

ST and RGBA are optional. If ST is dropped but RGBA is included the format will be:

```
[x, y, z, r, g, b, a]
```

If both ST and RGBA are dropped the format will be:

```
[x, y, z]
```

RGBA can also be of size 3 (RGB) or 4 (RGBA).

Output format is as follows:

```
numpy.array([
    # vertex 1
    [x, y, z, s, t, r, g, b, a],
    # vertex 2
    [x, y, z, s, t, r, g, b, a],
    ...
    # vertex N
    [x, y, z, s, t, r, g, b, a],
], dtype = dtype)
```

Parameters

- **st** (*bool*, *scalar*, *list*, *tuple*, *numpy.ndarray*) – The ST texture co-ordinates.

  Default is False, which means ST will not be included in the array.

  If True is passed, the default ST values will be provided with the bottom-left of the quad being located at ST=(0.0,0.0) and the top-right being located at ST=(1.0,1.0).

  If a 2d list, tuple or numpy array is passed, it must have one of the following shapes:

  ```
  (2,2,), (4,2,), (6,2,),
  ```

  If the shape is (2,2,), the values are interpreted as the minimum and maximum values for ST.

  For example:

  ```
  st=((0.1,0.3),(0.2,0.4))
  ```

  S values will be between 0.1 to 0.2. T values will be between 0.3 to 0.4.

  The bottom left will receive the minimum of both, and the top right will receive the maximum.

  If the shape is (4,2,), the values are interpreted as being the actual ST values for the 4 vertices of each face.

The vertices are in counter-clockwise winding order from the top right:

```
[top-right, top-left, bottom-left, bottom-right,]
```

If the shape is (6,2,), the values are interpreted as being the minimum and maximum values for each face of the cube.

The faces are in the following order:

```
[front, right, back, left, top, bottom,]
```

- **rgba** (*bool*, *scalar*, *list*, *tuple*, *numpy.ndarray*) – The RGBA colour.

  Default is False, which means RGBA will not be included in the array.

  If True is passed, the default RGBA values will be provided with all vertices being RGBA=(1.0, 1.0, 1.0, 1.0).

  If a 2d list, tuple or numpy array is passed, it must have one of the following shapes.:

```
(3,), (4,), (4,3,), (4,4,), (6,3,), (6,4,), (24,3,), (24,4,),
```

  If the shape is (3,), the values are interpreted as being an RGB value (no alpha) to set on all vertices.

  If the shape is (4,), the values are intepreted the same as the shape (3,) except the alpha value is included.

  If the shape is (4,3,), the values are interpreted as being a colour to set on the 4 vertices of each face.

  The vertices are in counter-clockwise winding order from the top right:

```
[top-right, top-left, bottom-left, bottom-right]
```

  If the shape is (4,4,), the values are intepreted the same as the shape (4,3,) except the alpha value is included.

  If the shape is (6,3,), the values are interpreted as being one RGB value (no alpha) for each face.

  The faces are in the following order:

```
[front, right, back, left, top, bottom,]
```

  If the shape is (6,4,), the values are interpreted the same as the shape (6,3,) except the alpha value is included.

  If the shape is (24,3,), the values are interpreted as being an RGB value (no alpha) to set on each vertex of each face (4 * 6).

  The faces are in the following order:

```
[front, right, back, left, top, bottom,]
```

  The vertices are in counter-clockwise winding order from the top right:

```
[top-right, top-left, bottom-left, bottom-right]
```

  If the shape is (24,4,), the values are interpreted the same as the shape (24,3,) except the alpha value is included.

- **type** (*string*) – The type of indices to generate.

  Valid values are:

  ```
  ['triangles', 'triangle_strip', 'triangle_fan', 'quads', 'quad_
  ↪strip',]
  ```

  If you just want the vertices without any index manipulation, use 'quads'.

pyrr.geometry.**create_quad**(*scale=(1.0, 1.0), st=False, rgba=False, dtype='float32', type='triangles'*)

Returns a Quad reading for rendering.

Output is a tuple of numpy arrays. The first value is the vertex data, the second is the indices.

The first dimension of the vertex data contains the list of vertices. The second dimension is the vertex data.

Vertex data is always in the following order:

```
[x, y, z, s, t, r, g, b, a]
```

ST and RGBA are optional. If ST is dropped but RGBA is included the format will be:

```
[x, y, z, r, g, b, a]
```

If both ST and RGBA are dropped the format will be:

```
[x, y, z]
```

RGBA can also be of size 3 (RGB) or 4 (RGBA).

Output format is as follows:

```
numpy.array([
    # vertex 1
    [x, y, z, s, t, r, g, b, a],
    # vertex 2
    [x, y, z, s, t, r, g, b, a],
    ...
    # vertex N
    [x, y, z, s, t, r, g, b, a],
], dtype = dtype)
```

Parameters

- **st** (*bool,scalar,list,tuple,numpy.ndarray*) – The ST texture co-ordinates.

  Default is False, which means ST will not be included in the array.

  If True is passed, the default ST values will be provided with the bottom-left of the quad being located at ST=(0.0,0.0) and the top-right being located at ST=(1.0,1.0).

  If a 2d list, tuple or numpy array is passed, it must have one of the following shapes:

  ```
  (2,2,), (4,2,),
  ```

  If the shape is (2,2,), the values are interpreted as the minimum and maximum values for ST.

  For example:

```
st=((0.1,0.3),(0.2,0.4))
```

S values will be between 0.1 to 0.2. T values will be between 0.3 to 0.4.

The bottom left will receive the minimum of both, and the top right will receive the maximum.

If the shape is (4,2,), the values are interpreted as being the actual ST values for the 4 vertices of the Quad.

The vertices are in counter-clockwise winding order from the top right:

```
[top-right, top-left, bottom-left, bottom-right,]
```

- **rgba** (*bool*, *scalar*, *list*, *tuple*, *numpy.ndarray*) – The RGBA colour.

  Default is False, which means RGBA will not be included in the array.

  If True is passed, the default RGBA values will be provided with all vertices being RGBA=(1.0, 1.0, 1.0, 1.0)

  If a 2d list, tuple or numpy array is passed, it must have one of the following shapes:

```
(3,), (4,), (4,3,), (4,4,),
```

  If the shape is (3,), the values are interpreted as being an RGB value (no alpha) to set on all vertices.

  If the shape is (4,), the values are intepreted the same as the shape (3,) except the alpha value is included.

  If the shape is (4,3,), the values are interpreted as being a colour to set on the 4 vertices of the Quad.

  The vertices are in counter-clockwise winding order from the top right:

```
[top-right, top-left, bottom-left, bottom-right]
```

  If the shape is (4,4,), the values are intepreted the same as the shape (4,3,) except the alpha value is included.

- **type** (*string*) – The type of indices to generate.

  Valid values are:

```
['triangles', 'triangle_strip', 'triangle_fan', 'quads', 'quad_
↪strip',]
```

  If you just want the vertices without any index manipulation, use 'quads'.

3.5 Integer

Provide functions for the manipulation of integers.

pyrr.integer.**count_bits**(*value*)
Counts the number of bits set to 1 in an integer.

For example:

```
>>> count_bits(0b101111)
5
>>> count_bits(0xf)
4
>>> count_bits(8)
1
>>> count_bits(3)
2
```

Parameters **value** (*int*) – An integer.

Return type integer

Returns The count of bits set to 1.

See also:

http://wiki.python.org/moin/BitManipulation

3.6 Line

Provide functions for the creation and manipulation of Lines.

A Line data structure is simply a numpy.array with 2 vectors:

```
start = numpy.array( [ -1.0, 0.0, 0.0 ] )
end = numpy.array( [ 1.0, 0.0, 0.0 ] )
line = numpy.array( [ start, end ] )
```

Both Lines and Line Segments are defined using the same data structure. The only difference is how the data is interpreted.

A line is defined by two points but extends infinitely.

A line segment only exists between two points. It does not extend forever.

The choice to interprete a line as a line or line segment is up to the function being called. Check the function signature of documentation to determine how a line will be interpreted.

pyrr.line.**create_from_points**(*v1*, *v2*, *dtype=None*)

Creates a line from 2 vectors.

The 2 vectors represent the start and end point of the line.

Parameters

- **v1** (*numpy.array*) – Start point.
- **v2** (*numpy.array*) – End point.

Return type numpy.array

Returns A line extending from v1 to v2.

pyrr.line.**create_from_ray**(**args*, ***kwargs*)

Converts a ray to a line.

The line will extend from 'ray origin -> ray origin + ray direction'.

Parameters **ray** (*numpy.array*) – The ray to convert.

Return type numpy.array

Returns A line beginning at the ray start and extending for 1 unit in the direction of the ray.

pyrr.line.**create_zeros**(*dtype=None*)
Creates a line with the start and end at the origin.

Return type numpy.array

Returns A line with both start and end points at (0,0,0).

pyrr.line.**end**(**args*, ***kwargs*)
Extracts the end point of the line.

Parameters **line** (*numpy.array*) – The line to extract the end from.

Return type numpy.array

Returns The ending point of the line.

class pyrr.line.**index**

end = 1

start = 0

pyrr.line.**start**(**args*, ***kwargs*)
Extracts the start point of the line.

Parameters **line** (*numpy.array*) – The line to extract the start from.

Return type numpy.array

Returns The starting point of the line.

3.7 Matrix functions

3.7.1 Matrix33

3x3 Matrix which supports rotation, translation, scale and skew.

Matrices are laid out in row-major format and can be loaded directly into OpenGL. To convert to column-major format, transpose the array using the numpy.array.T method.

pyrr.matrix33.**apply_to_vector**(**args*, ***kwargs*)
Apply a matrix to a vector.

The matrix's rotation are applied to the vector. Supports multiple matrices and vectors.

Parameters

- **mat** (*numpy.array*) – The rotation / translation matrix. Can be a list of matrices.
- **vec** (*numpy.array*) – The vector to modify. Can be a list of vectors.

Return type numpy.array

Returns The vectors rotated by the specified matrix.

pyrr.matrix33.**create_direction_scale**(*direction*, *scale*)
Creates a matrix which can apply a directional scaling to a set of vectors.

An example usage for this is to flatten a mesh against a single plane.

Parameters

- **direction** (*numpy.array*) – a numpy.array of shape (3,) of the direction to scale.
- **scale** (*float*) – a float value for the scaling along the specified direction. A scale of 0.0 will flatten the vertices into a single plane with the direction being the plane's normal.

Return type numpy.array

Returns The scaling matrix.

pyrr.matrix33.**create_from_axis_rotation**(**args*, ***kwargs*)
Creates a matrix from the specified theta rotation around an axis.

Parameters

- **axis** (*numpy.array*) – A (3,) vector specifying the axis of rotation.
- **theta** (*float*) – A rotation speicified in radians.

Return type numpy.array

Returns A matrix with shape (3,3).

pyrr.matrix33.**create_from_eulers**(**args*, ***kwargs*)
Creates a matrix from the specified Euler rotations.

Parameters **eulers** (*numpy.array*) – A set of euler rotations in the format specified by the euler modules.

Return type numpy.array

Returns A matrix with shape (3,3) with the euler's rotation.

pyrr.matrix33.**create_from_inverse_of_quaternion**(**args*, ***kwargs*)
Creates a matrix with the inverse rotation of a quaternion.

Parameters **quat** (*numpy.array*) – The quaternion to make the matrix from (shape 4).

Return type numpy.array

Returns A matrix with shape (3,3) that respresents the inverse of the quaternion.

pyrr.matrix33.**create_from_matrix44**(*mat*, *dtype=None*)
Creates a Matrix33 from a Matrix44.

Return type numpy.array

Returns A matrix with shape (3,3) with the input matrix rotation.

pyrr.matrix33.**create_from_quaternion**(**args*, ***kwargs*)
Creates a matrix with the same rotation as a quaternion.

Parameters **quat** – The quaternion to create the matrix from.

Return type numpy.array

Returns A matrix with shape (3,3) with the quaternion's rotation.

pyrr.matrix33.**create_from_scale**(*scale*, *dtype=None*)
Creates an identity matrix with the scale set.

Parameters **scale** (*numpy.array*) – The scale to apply as a vector (shape 3).

Return type numpy.array

Returns A matrix with shape (3,3) with the scale set to the specified vector.

pyrr.matrix33.**create_from_x_rotation**(*theta*, *dtype=None*)
Creates a matrix with the specified rotation about the X axis.

Parameters **theta** (*float*) – The rotation, in radians, about the X-axis.

Return type numpy.array

Returns A matrix with the shape (3,3) with the specified rotation about the X-axis.

See also:

http://en.wikipedia.org/wiki/Rotation_matrix#In_three_dimensions

pyrr.matrix33.**create_from_y_rotation**(*theta*, *dtype=None*)
Creates a matrix with the specified rotation about the Y axis.

Parameters **theta** (*float*) – The rotation, in radians, about the Y-axis.

Return type numpy.array

Returns A matrix with the shape (3,3) with the specified rotation about the Y-axis.

See also:

http://en.wikipedia.org/wiki/Rotation_matrix#In_three_dimensions

pyrr.matrix33.**create_from_z_rotation**(*theta*, *dtype=None*)
Creates a matrix with the specified rotation about the Z axis.

Parameters **theta** (*float*) – The rotation, in radians, about the Z-axis.

Return type numpy.array

Returns A matrix with the shape (3,3) with the specified rotation about the Z-axis.

See also:

http://en.wikipedia.org/wiki/Rotation_matrix#In_three_dimensions

pyrr.matrix33.**create_identity**(*dtype=None*)
Creates a new matrix33 and sets it to an identity matrix.

Return type numpy.array

Returns A matrix representing an identity matrix with shape (3,3).

pyrr.matrix33.**inverse**(*mat*)
Returns the inverse of the matrix.

This is essentially a wrapper around numpy.linalg.inv.

Parameters **m** (*numpy.array*) – A matrix.

Return type numpy.array

Returns The inverse of the specified matrix.

See also:

http://docs.scipy.org/doc/numpy/reference/generated/numpy.linalg.inv.html

pyrr.matrix33.**multiply**(*m1*, *m2*)
Multiply two matricies, m1 . m2.

This is essentially a wrapper around numpy.dot(m1, m2)

Parameters

- **m1** (*numpy.array*) – The first matrix. Can be a list of matrices.

- **m2** (*numpy.array*) – The second matrix. Can be a list of matrices.

Return type numpy.array

Returns A matrix that results from multiplying m1 by m2.

3.7.2 Matrix44

4x4 Matrix which supports rotation, translation, scale and skew.

Matrices are laid out in row-major format and can be loaded directly into OpenGL. To convert to column-major format, transpose the array using the numpy.array.T method.

pyrr.matrix44.**apply_to_vector**(**args*, ***kwargs*)
Apply a matrix to a vector.

The matrix's rotation and translation are applied to the vector. Supports multiple matrices and vectors.

Parameters

- **mat** (*numpy.array*) – The rotation / translation matrix. Can be a list of matrices.
- **vec** (*numpy.array*) – The vector to modify. Can be a list of vectors.

Return type numpy.array

Returns The vectors rotated by the specified matrix.

pyrr.matrix44.**create_from_axis_rotation**(**args*, ***kwargs*)
Creates a matrix from the specified rotation theta around an axis.

Parameters

- **axis** (*numpy.array*) – A (3,) vector.
- **theta** (*float*) – A rotation in radians.

Return type numpy.array

Returns A matrix with shape (4,4).

pyrr.matrix44.**create_from_eulers**(**args*, ***kwargs*)
Creates a matrix from the specified Euler rotations.

Parameters **eulers** (*numpy.array*) – A set of euler rotations in the format specified by the euler modules.

Return type numpy.array

Returns A matrix with shape (4,4) with the euler's rotation.

pyrr.matrix44.**create_from_inverse_of_quaternion**(**args*, ***kwargs*)
Creates a matrix with the inverse rotation of a quaternion.

This can be used to go from object space to intertial space.

Parameters **quat** (*numpy.array*) – The quaternion to make the matrix from (shape 4).

Return type numpy.array

Returns A matrix with shape (4,4) that respresents the inverse of the quaternion.

pyrr.matrix44.**create_from_matrix33**(*mat*, *dtype=None*)
Creates a Matrix44 from a Matrix33.

The translation will be 0,0,0.

Return type numpy.array

Returns A matrix with shape (4,4) with the input matrix rotation.

pyrr.matrix44.**create_from_quaternion**(**args*, ***kwargs*)
Creates a matrix with the same rotation as a quaternion.

Parameters **quat** – The quaternion to create the matrix from.

Return type numpy.array

Returns A matrix with shape (4,4) with the quaternion's rotation.

pyrr.matrix44.**create_from_scale**(*scale*, *dtype=None*)
Creates an identity matrix with the scale set.

Parameters **scale** (*numpy.array*) – The scale to apply as a vector (shape 3).

Return type numpy.array

Returns A matrix with shape (4,4) with the scale set to the specified vector.

pyrr.matrix44.**create_from_translation**(**args*, ***kwargs*)
Creates an identity matrix with the translation set.

Parameters **vec** (*numpy.array*) – The translation vector (shape 3 or 4).

Return type numpy.array

Returns A matrix with shape (4,4) that represents a matrix with the translation set to the specified vector.

pyrr.matrix44.**create_from_x_rotation**(*theta*, *dtype=None*)
Creates a matrix with the specified rotation about the X axis.

Parameters **theta** (*float*) – The rotation, in radians, about the X-axis.

Return type numpy.array

Returns A matrix with the shape (4,4) with the specified rotation about the X-axis.

See also:

http://en.wikipedia.org/wiki/Rotation_matrix#In_three_dimensions

pyrr.matrix44.**create_from_y_rotation**(*theta*, *dtype=None*)
Creates a matrix with the specified rotation about the Y axis.

Parameters **theta** (*float*) – The rotation, in radians, about the Y-axis.

Return type numpy.array

Returns A matrix with the shape (4,4) with the specified rotation about the Y-axis.

See also:

http://en.wikipedia.org/wiki/Rotation_matrix#In_three_dimensions

pyrr.matrix44.**create_from_z_rotation**(*theta*, *dtype=None*)
Creates a matrix with the specified rotation about the Z axis.

Parameters **theta** (*float*) – The rotation, in radians, about the Z-axis.

Return type numpy.array

Returns A matrix with the shape (4,4) with the specified rotation about the Z-axis.

See also:

http://en.wikipedia.org/wiki/Rotation_matrix#In_three_dimensions

pyrr.matrix44.**create_identity**(*dtype=None*)
Creates a new matrix44 and sets it to an identity matrix.

Return type numpy.array

Returns A matrix representing an identity matrix with shape (4,4).

pyrr.matrix44.**create_look_at**(*eye*, *target*, *up*, *dtype=None*)
Creates a look at matrix according to OpenGL standards.

Parameters

- **eye** (*numpy.array*) – Position of the camera in world coordinates.
- **target** (*numpy.array*) – The position in world coordinates that the camera is looking at.
- **up** (*numpy.array*) – The up vector of the camera.

Return type numpy.array

Returns A look at matrix that can be used as a viewMatrix

pyrr.matrix44.**create_matrix33_view**(*mat*)
Returns a view into the matrix in Matrix33 format.

This is different from matrix33.create_from_matrix44, in that changes to the returned matrix will also alter the original matrix.

Return type numpy.array

Returns A view into the matrix in the format of a matrix33 (shape (3,3)).

pyrr.matrix44.**create_orthogonal_projection**(*left*, *right*, *bottom*, *top*, *near*, *far*, *dtype=None*)
Creates an orthogonal projection matrix.

Parameters

- **left** (*float*) – The left of the near plane relative to the plane's centre.
- **right** (*float*) – The right of the near plane relative to the plane's centre.
- **top** (*float*) – The top of the near plane relative to the plane's centre.
- **bottom** (*float*) – The bottom of the near plane relative to the plane's centre.
- **near** (*float*) – The distance of the near plane from the camera's origin. It is recommended that the near plane is set to 1.0 or above to avoid rendering issues at close range.
- **far** (*float*) – The distance of the far plane from the camera's origin.

Return type numpy.array

Returns A projection matrix representing the specified orthogonal perspective.

See also:

http://msdn.microsoft.com/en-us/library/dd373965(v=vs.85).aspx

pyrr.matrix44.**create_orthogonal_projection_matrix**(*left*, *right*, *bottom*, *top*, *near*, *far*, *dtype=None*)
Creates an orthogonal projection matrix.

Parameters

- **left** (*float*) – The left of the near plane relative to the plane's centre.
- **right** (*float*) – The right of the near plane relative to the plane's centre.
- **top** (*float*) – The top of the near plane relative to the plane's centre.
- **bottom** (*float*) – The bottom of the near plane relative to the plane's centre.
- **near** (*float*) – The distance of the near plane from the camera's origin. It is recommended that the near plane is set to 1.0 or above to avoid rendering issues at close range.
- **far** (*float*) – The distance of the far plane from the camera's origin.

Return type numpy.array

Returns A projection matrix representing the specified orthogonal perspective.

See also:

http://msdn.microsoft.com/en-us/library/dd373965(v=vs.85).aspx

pyrr.matrix44.**create_perspective_projection**(*fovy*, *aspect*, *near*, *far*, *dtype=None*)

Creates perspective projection matrix.

See also:

http://www.opengl.org/sdk/docs/man2/xhtml/gluPerspective.xml

See also:

http://www.geeks3d.com/20090729/howto-perspective-projection-matrix-in-opengl/

Parameters

- **fovy** (*float*) – field of view in y direction in degrees
- **aspect** (*float*) – aspect ratio of the view (width / height)
- **near** (*float*) – distance from the viewer to the near clipping plane (only positive)
- **far** (*float*) – distance from the viewer to the far clipping plane (only positive)

Return type numpy.array

Returns A projection matrix representing the specified perpective.

pyrr.matrix44.**create_perspective_projection_from_bounds**(*left*, *right*, *bottom*, *top*, *near*, *far*, *dtype=None*)

Creates a perspective projection matrix using the specified near plane dimensions.

Parameters

- **left** (*float*) – The left of the near plane relative to the plane's centre.
- **right** (*float*) – The right of the near plane relative to the plane's centre.
- **top** (*float*) – The top of the near plane relative to the plane's centre.
- **bottom** (*float*) – The bottom of the near plane relative to the plane's centre.
- **near** (*float*) – The distance of the near plane from the camera's origin. It is recommended that the near plane is set to 1.0 or above to avoid rendering issues at close range.
- **far** (*float*) – The distance of the far plane from the camera's origin.

Return type numpy.array

Returns A projection matrix representing the specified perspective.

See also:

http://www.gamedev.net/topic/264248-building-a-projection-matrix-without-api/

See also:

http://www.glprogramming.com/red/chapter03.html

pyrr.matrix44.**create_perspective_projection_matrix**(*fovy*, *aspect*, *near*, *far*, *dtype=None*)

Creates perspective projection matrix.

See also:

http://www.opengl.org/sdk/docs/man2/xhtml/gluPerspective.xml

See also:

http://www.geeks3d.com/20090729/howto-perspective-projection-matrix-in-opengl/

Parameters

- **fovy** (*float*) – field of view in y direction in degrees
- **aspect** (*float*) – aspect ratio of the view (width / height)
- **near** (*float*) – distance from the viewer to the near clipping plane (only positive)
- **far** (*float*) – distance from the viewer to the far clipping plane (only positive)

Return type numpy.array

Returns A projection matrix representing the specified perpective.

pyrr.matrix44.**create_perspective_projection_matrix_from_bounds**(*left*, *right*, *bottom*, *top*, *near*, *far*, *dtype=None*)

Creates a perspective projection matrix using the specified near plane dimensions.

Parameters

- **left** (*float*) – The left of the near plane relative to the plane's centre.
- **right** (*float*) – The right of the near plane relative to the plane's centre.
- **top** (*float*) – The top of the near plane relative to the plane's centre.
- **bottom** (*float*) – The bottom of the near plane relative to the plane's centre.
- **near** (*float*) – The distance of the near plane from the camera's origin. It is recommended that the near plane is set to 1.0 or above to avoid rendering issues at close range.
- **far** (*float*) – The distance of the far plane from the camera's origin.

Return type numpy.array

Returns A projection matrix representing the specified perspective.

See also:

http://www.gamedev.net/topic/264248-building-a-projection-matrix-without-api/

See also:

http://www.glprogramming.com/red/chapter03.html

pyrr.matrix44.**decompose**(*m*)
Decomposes an affine transformation matrix into its scale, rotation and translation components.

Parameters **m** (*numpy.array*) – A matrix.

Returns tuple (scale, rotation, translation) numpy.array scale vector3 numpy.array rotation quaternion numpy.array translation vector3

pyrr.matrix44.**inverse**(*m*)
Returns the inverse of the matrix.

This is essentially a wrapper around numpy.linalg.inv.

Parameters **m** (*numpy.array*) – A matrix.

Return type numpy.array

Returns The inverse of the specified matrix.

See also:

http://docs.scipy.org/doc/numpy/reference/generated/numpy.linalg.inv.html

pyrr.matrix44.**multiply**(*m1*, *m2*)
Multiply two matricies, m1 . m2.

This is essentially a wrapper around numpy.dot(m1, m2)

Parameters

- **m1** (*numpy.array*) – The first matrix. Can be a list of matrices.
- **m2** (*numpy.array*) – The second matrix. Can be a list of matrices.

Return type numpy.array

Returns A matrix that results from multiplying m1 by m2.

3.8 Plane

Provide functions for the creation and manipulation of Planes.

Planes are represented using a numpy.array of shape (4,). The values represent the plane equation using the values A,B,C,D.

The first three values are the normal vector. The fourth value is the distance of the plane from the origin, down the normal.

pyrr.plane.**create**(*normal=None*, *distance=0.0*, *dtype=None*)
Creates a plane that runs along the X,Y plane.

It crosses the origin with a normal of 0,0,1 (+Z).

Return type numpy.array

Returns A plane that runs along the X,Y plane.

pyrr.plane.**create_from_points**(**args*, ***kwargs*)
Create a plane from 3 co-planar vectors.

The vectors must all lie on the same plane or an exception will be thrown.

The vectors must not all be in a single line or the plane is undefined.

The order the vertices are passed in will determine the normal of the plane.

Parameters

- **vector1** (*numpy.array*) – a vector that lies on the desired plane.
- **vector2** (*numpy.array*) – a vector that lies on the desired plane.
- **vector3** (*numpy.array*) – a vector that lies on the desired plane.

Raises **ValueError** – raised if the vectors are co-incident (in a single line).

Return type numpy.array

Returns A plane that contains the 3 specified vectors.

pyrr.plane.**create_from_position**(**args*, ***kwargs*)
Creates a plane at position with the normal being above the plane and up being the rotation of the plane.

Parameters

- **position** (*numpy.array*) – The position of the plane.
- **normal** (*numpy.array*) – The normal of the plane. Will be normalized during construction.

Return type numpy.array

Returns A plane that crosses the specified position with the specified normal.

pyrr.plane.**create_xy**(*invert=False*, *distance=0.0*, *dtype=None*)
Create a plane on the XY plane, starting at the origin with +Z being the up vector.

The distance is the distance along the normal (-Z if inverted, otherwise +Z).

pyrr.plane.**create_xz**(*invert=False*, *distance=0.0*, *dtype=None*)
Create a plane on the XZ plane, starting at the origin with +Y being the up vector.

The distance is the distance along the normal (-Y if inverted, otherwise +Y).

pyrr.plane.**create_yz**(*invert=False*, *distance=0.0*, *dtype=None*)
Create a plane on the YZ plane, starting at the origin with +X being the up vector.

The distance is the distance along the normal (-X if inverted, otherwise +X).

pyrr.plane.**invert_normal**(*plane*)
Flips the normal of the plane.

The plane is **not** changed in place.

Return type numpy.array

Returns The plane with the normal inverted.

pyrr.plane.**normal**(*plane*)
Extracts the normal vector from a plane.

Parameters **plane** (*numpy.array*) – The plane.

Return type numpy.array

Returns The normal vector of the plane.

pyrr.plane.**position**(*plane*)
Extracts the position vector from a plane.

This will be a vector co-incident with the plane's normal.

Parameters **plane** (*numpy.array*) – The plane.

Return type numpy.array

Returns A valid position that lies on the plane.

3.9 Quaternion

Provide functions for the creation and manipulation of Quaternions.

pyrr.quaternion.**apply_to_vector**(**args*, ***kwargs*)
Rotates a vector by a quaternion.

Parameters

- **quat** (*numpy.array*) – The quaternion.
- **vec** (*numpy.array*) – The vector.

Return type numpy.array

Returns The vector rotated by the quaternion.

Raises **ValueError** – raised if the vector is an unsupported size

pyrr.quaternion.**conjugate**(**args*, ***kwargs*)
Calculates a quaternion with the opposite rotation.

Parameters **quat** (*numpy.array*) – The quaternion.

Return type numpy.array.

Returns A quaternion representing the conjugate.

pyrr.quaternion.**create**(*x=0.0*, *y=0.0*, *z=0.0*, *w=1.0*, *dtype=None*)

pyrr.quaternion.**create_from_axis**(**args*, ***kwargs*)

pyrr.quaternion.**create_from_axis_rotation**(**args*, ***kwargs*)

pyrr.quaternion.**create_from_eulers**(**args*, ***kwargs*)
Creates a quaternion from a set of Euler angles.

Eulers are an array of length 3 in the following order: [roll, pitch, yaw]

pyrr.quaternion.**create_from_inverse_of_eulers**(**args*, ***kwargs*)
Creates a quaternion from the inverse of a set of Euler angles.

Eulers are an array of length 3 in the following order: [roll, pitch, yaw]

pyrr.quaternion.**create_from_matrix**(**args*, ***kwargs*)

pyrr.quaternion.**create_from_x_rotation**(*theta*, *dtype=None*)

pyrr.quaternion.**create_from_y_rotation**(*theta*, *dtype=None*)

pyrr.quaternion.**create_from_z_rotation**(*theta*, *dtype=None*)

pyrr.quaternion.**cross**(**args*, ***kwargs*)
Returns the cross-product of the two quaternions.

Quaternions are **not** communicative. Therefore, order is important.

This is NOT the same as a vector cross-product. Quaternion cross-product is the equivalent of matrix multiplication.

pyrr.quaternion.**dot**(*quat1*, *quat2*)
Calculate the dot product of quaternions.

This is the same as a vector dot product.

Parameters

- **quat1** (*numpy.array*) – The first quaternion(s).
- **quat2** (*numpy.array*) – The second quaternion(s).

Return type float, numpy.array

Returns If a 1d array was passed, it will be a scalar. Otherwise the result will be an array of scalars with shape vec.ndim with the last dimension being size 1.

pyrr.quaternion.**exp**(**args*, ***kwargs*)
Calculate the exponential of the quaternion

Parameters **quat** (*numpy.array*) – The quaternion.

Return type numpy.array.

Returns The exponential of the quaternion

class pyrr.quaternion.**index**

w = 3

x = 0

y = 1

z = 2

pyrr.quaternion.**inverse**(*quat*)
Calculates the inverse quaternion.

The inverse of a quaternion is defined as the conjugate of the quaternion divided by the magnitude of the original quaternion.

Parameters **quat** (*numpy.array*) – The quaternion to invert.

Return type numpy.array.

Returns The inverse of the quaternion.

pyrr.quaternion.**is_identity**(*quat*)

pyrr.quaternion.**is_non_zero_length**(*quat*)
Checks if a quaternion is not zero length.

This is the opposite to 'is_zero_length'. This is provided for readability.

Parameters **quat** (*numpy.array*) – The quaternion to check.

Return type boolean

Returns False if the quaternion is zero length, otherwise True.

See also:

is_zero_length

pyrr.quaternion.**is_zero_length**(*quat*)
Checks if a quaternion is zero length.

Parameters **quat** (*numpy.array*) – The quaternion to check.

Return type boolean.

Returns True if the quaternion is zero length, otherwise False.

pyrr.quaternion.**length**(*quat*)
Calculates the length of a quaternion.

Parameters **quat** (*numpy.array*) – The quaternion to measure.

Return type float, numpy.array

Returns If a 1d array was passed, it will be a scalar. Otherwise the result will be an array of scalars with shape vec.ndim with the last dimension being size 1.

pyrr.quaternion.**lerp**(*quat1*, *quat2*, *t*)
Interpolates between quat1 and quat2 by t. The parameter t is clamped to the range [0, 1]

pyrr.quaternion.**negate**(**args*, ***kwargs*)
Calculates the negated quaternion.

This is essentially the quaternion * -1.0.

Parameters **quat** (*numpy.array*) – The quaternion.

Return type numpy.array

Returns The negated quaternion.

pyrr.quaternion.**normalise**(*quat*)
Ensure a quaternion is unit length (length ~= 1.0).

The quaternion is **not** changed in place.

Parameters **quat** (*numpy.array*) – The quaternion to normalize.

Return type numpy.array

Returns The normalized quaternion(s).

pyrr.quaternion.**normalize**(*quat*)
Ensure a quaternion is unit length (length ~= 1.0).

The quaternion is **not** changed in place.

Parameters **quat** (*numpy.array*) – The quaternion to normalize.

Return type numpy.array

Returns The normalized quaternion(s).

pyrr.quaternion.**power**(**args*, ***kwargs*)
Multiplies the quaternion by the exponent.

The quaternion is **not** changed in place.

Parameters

- **quat** (*numpy.array*) – The quaternion.
- **scalar** (*float*) – The exponent.

Return type numpy.array.

Returns A quaternion representing the original quaternion to the specified power.

pyrr.quaternion.**rotation_angle**(*quat*)
Calculates the rotation around the quaternion's axis.

Parameters **quat** (*numpy.array*) – The quaternion.

Return type float.

Returns The quaternion's rotation about the its axis in radians.

pyrr.quaternion.**rotation_axis**(**args*, ***kwargs*)
Calculates the axis of the quaternion's rotation.

Parameters **quat** (*numpy.array*) – The quaternion.

Return type numpy.array.

Returns The quaternion's rotation axis.

pyrr.quaternion.**slerp**(*quat1*, *quat2*, *t*)
Spherically interpolates between quat1 and quat2 by t. The parameter t is clamped to the range [0, 1]

pyrr.quaternion.**squared_length**(*quat*)
Calculates the squared length of a quaternion.

Useful for avoiding the performanc penalty of the square root function.

Parameters **quat** (*numpy.array*) – The quaternion to measure.

Return type float, numpy.array

Returns If a 1d array was passed, it will be a scalar. Otherwise the result will be an array of scalars with shape vec.ndim with the last dimension being size 1.

3.10 Ray

Provide functions for the creation and manipulation of Rays.

A ray begins as a single point and extends infinitely in a direction.

The first vector is the origin of the ray. The second vector is the direction of the ray relative to the origin.

The following functions will normalize the ray direction to unit length. Some functions may work correctly with directions that are not unit length, but this may vary from function to function.

pyrr.ray.**create**(**args*, ***kwargs*)

pyrr.ray.**create_from_line**(**args*, ***kwargs*)
Converts a line or line segment to a ray.

pyrr.ray.**direction**(**args*, ***kwargs*)

class pyrr.ray.**index**

direction = 1

position = 0

pyrr.ray.**invert**(**args*, ***kwargs*)

pyrr.ray.**position**(**args*, ***kwargs*)

3.11 Rectangle

Provide functions for the creation and manipulation of 2D Rectangles.

Rectangles are represented using a numpy.array of shape (2,2,).

The first value is a vector of x, y position of the rectangle. The second value is a vector with the width, height of the rectangle.

pyrr.rectangle.**abs_height**(*rect*)
Returns the absolute height of the rectangle.

This caters for rectangles with a negative height.

Return type float

Returns The absolute height of the rectangle.

pyrr.rectangle.**abs_size**(*rect*)
Returns the absolute size of the rectangle.

Return type numpy.array

Returns The absolute size of the rectangle.

pyrr.rectangle.**abs_width**(*rect*)
Returns the absolute width of the rectangle.

This caters for rectangles with a negative width.

Return type float

Returns The absolute width of the rectangle.

pyrr.rectangle.**aspect_ratio**(*rect*)

pyrr.rectangle.**bottom**(*rect*)
Returns the bottom most Y value of the rectangle.

This caters for rectangles with a negative height.

Return type float

Returns The smallest Y value.

pyrr.rectangle.**bounds**(**args*, ***kwargs*)
Returns the absolute boundaries of the rectangle.

This caters for rectangles with a negative width.

Return type Tuple of 4 floats

Returns The absolute left, right, bottom and top of the rectangle.

pyrr.rectangle.**create**(*x=0.0*, *y=0.0*, *width=1.0*, *height=1.0*, *dtype=None*)
Creates a rectangle from the specified position and sizes.

This function will interpret the values literally. A negative width or height will be represented by the returned value.

Return type numpy.array

Returns Returns a rectangle with the specified values.

pyrr.rectangle.**create_from_bounds**(*left*, *right*, *bottom*, *top*, *dtype=None*)
Creates a rectangle from the specified boundaries.

This caters for the left and right, and for the top and bottom being swapped.

Return type numpy.array

Returns Returns a rectangle with the specified values. The rectangle will have a positive width and height regardless of the values passed in.

pyrr.rectangle.**create_zeros**(*dtype=None*)

pyrr.rectangle.**height**(*rect*)
Returns the literal height of the rectangle.

Return type float

Returns The height of the rectangle. This can be a negative value.

class pyrr.rectangle.**index**

position = 0

size = 1

pyrr.rectangle.**left**(*rect*)
Returns the left most X value of the rectangle.

This caters for rectangles with a negative width.

Return type float

Returns The smallest X value.

pyrr.rectangle.**position**(**args*, ***kwargs*)
Returns the literal position of the rectangle.

This is the bottom-left point of the rectangle for rectangles with positive width and height

Return type numpy.array

Returns The position of the rectangle.

pyrr.rectangle.**right**(*rect*)
Returns the right most X value of the rectangle.

This caters for rectangles with a negative width.

Return type float

Returns The biggest X value.

pyrr.rectangle.**scale_by_vector**(**args*, ***kwargs*)
Scales a rectangle by a 2D vector.

Note that this will also scale the X,Y value of the rectangle, which will cause the rectangle to move, not just increase in size.

Parameters

- **rect** (*numpy.array*) – the rectangle to scale. Both x,y and width,height will be scaled.
- **vec** – A 2D vector to scale the rect by.

Return type numpy.array.

pyrr.rectangle.**size**(**args*, ***kwargs*)
Returns the literal size of the rectangle.

These values may be negative.

Return type numpy.array

Returns The size of the rectangle.

pyrr.rectangle.**top**(*rect*)
Returns the top most Y value of the rectangle.

This caters for rectangles with a negative height.

Return type float

Returns The biggest Y value.

pyrr.rectangle.**width**(*rect*)
Returns the literal width of the rectangle.

Return type float

Returns The width of the rectangle. This can be a negative value.

pyrr.rectangle.**x**(*rect*)
Returns the X position of the rectangle.

This will be the left for rectangles with positive height values.

Return type float

Returns The X position of the rectangle. This value will be further right than the 'right' if the width is negative.

pyrr.rectangle.**y**(*rect*)
Returns the Y position of the rectangle.

This will be the bottom for rectangles with positive height values.

Return type float

Returns The Y position of the rectangle. This value will be above the bottom if the height is negative.

3.12 Sphere

Provide functions for the creation and manipulation of 3D Spheres.

Sphere are represented using a numpy.array of shape (4,).

The first three values are the sphere's position. The fourth value is the sphere's radius.

pyrr.sphere.**create**(**args*, ***kwargs*)

pyrr.sphere.**create_from_points**(**args*, ***kwargs*)
Creates a sphere centred around 0,0,0 that encompasses the furthest point in the provided list.

Parameters **points** (*numpy.array*) – An Nd array of vectors.

Return type A sphere as a two value tuple.

pyrr.sphere.**position**(**args*, ***kwargs*)
Returns the position of the sphere.

Parameters **sphere** (*numpy.array*) – The sphere to extract the position from.

Return type numpy.array

Returns The centre of the sphere.

pyrr.sphere.**radius**(**args*, ***kwargs*)
Returns the radius of the sphere.

Parameters **sphere** (*numpy.array*) – The sphere to extract the radius from.

Return type float

Returns The radius of the sphere.

3.13 Trigonometry

Provide functions for the trigonometric functions.

pyrr.trig.**aspect_ratio**(*width*, *height*)

pyrr.trig.**calculate_fov**(*zoom*, *height=1.0*)

Calculates the required FOV to set the view frustrum to have a view with the specified height at the specified distance.

Parameters

- **zoom** (*float*) – The distance to calculate the FOV for.
- **height** (*float*) – The desired view height at the specified distance. The default is 1.0.

Return type A float representing the FOV to use in degrees.

pyrr.trig.**calculate_height**(*fov*, *zoom*)

Performs the opposite of calculate_fov. Used to find the current height at a specific distance.

Parameters

- **fov** (*float*) – The current FOV.
- **zoom** (*float*) – The distance to calculate the height for.

Return type A float representing the height at the specified distance for the specified FOV.

pyrr.trig.**calculate_plane_size**(*aspect_ratio*, *fov*, *distance*)

Calculates the width and height of a plane at the specified distance using the FOV of the frustrum and aspect ratio of the viewport.

Parameters

- **aspect_ratio** (*float*) – The aspect ratio of the viewport.
- **fov** (*float*) – The FOV of the frustrum.
- **distance** (*float*) – The distance from the origin/camera of the plane to calculate.

Return type A tuple of two floats: width and height: The width and height of the plane.

pyrr.trig.**calculate_zoom**(*fov*, *height=1.0*)

Calculates the zoom (distance) from the camera with the specified FOV and height of image.

Parameters

- **fov** (*float*) – The FOV to use.
- **height** (*float*) – The height of the image at the desired distance.

Return type A float representing the zoom (distance) from the camera for the desired height at the specified FOV.

Raises **ZeroDivisionError** – Raised if the fov is 0.0.

3.14 Utilities

Provides common utility functions.

pyrr.utils.**all_parameters_as_numpy_arrays**(*fn*)
Converts all of a function's arguments to numpy arrays.

Used as a decorator to reduce duplicate code.

pyrr.utils.**parameters_as_numpy_arrays**(**args_to_convert*)
Converts specific arguments to numpy arrays.

Used as a decorator to reduce duplicate code.

Arguments are specified by their argument name. For example

```
@parameters_as_numpy_arrays('a', 'b', 'optional')
def myfunc(a, b, *args, **kwargs):
    pass

myfunc(1, [2,2], optional=[3,3,3])
```

3.15 Vector functions

3.15.1 Common Vector Operations

Common Vector manipulation functions.

pyrr.vector.**dot**(**args*, ***kwargs*)
Calculates the dot product of two vectors.

Parameters

- **v1** (*numpy.array*) – an Nd array with the final dimension being size 3. (a vector)
- **v2** (*numpy.array*) – an Nd array with the final dimension being size 3 (a vector)

Return type If a 1d array was passed, it will be a scalar. Otherwise the result will be an array of scalars with shape vec.ndim with the last dimension being size 1.

pyrr.vector.**interpolate**(**args*, ***kwargs*)
Interpolates between 2 arrays of vectors (shape = N,3) by the specified delta (0.0 <= delta <= 1.0).

Parameters

- **v1** (*numpy.array*) – an Nd array with the final dimension being size 3. (a vector)
- **v2** (*numpy.array*) – an Nd array with the final dimension being size 3. (a vector)
- **delta** (*float*) – The interpolation percentage to apply, where 0.0 <= delta <= 1.0. When delta is 0.0, the result will be v1. When delta is 1.0, the result will be v2. Values inbetween will be an interpolation.

Return type A numpy.array with shape v1.shape.

pyrr.vector.**length**(**args*, ***kwargs*)
Returns the length of an Nd list of vectors or a single vector.

Parameters **vec** (*numpy.array*) – an Nd array with the final dimension being size 3 (a vector).

Single vector:

```
numpy.array([ x, y, z ])
```

Nd array:

```
numpy.array([
    [x1, y1, z1],
    [x2, y2, z2]
]).
```

Return type If a 1d array was passed, it will be a scalar. Otherwise the result will be an array of scalars with shape vec.ndim with the last dimension being size 1.

pyrr.vector.**normalise**(**args*, ***kwargs*)

normalizes an Nd list of vectors or a single vector to unit length.

The vector is **not** changed in place.

For zero-length vectors, the result will be np.nan.

Parameters **vec** (*numpy.array*) – an Nd array with the final dimension being vectors

```
numpy.array([ x, y, z ])
```

Or an NxM array:

```
numpy.array([
    [x1, y1, z1],
    [x2, y2, z2]
]).
```

Return type A numpy.array the normalized value

pyrr.vector.**normalize**(**args*, ***kwargs*)

normalizes an Nd list of vectors or a single vector to unit length.

The vector is **not** changed in place.

For zero-length vectors, the result will be np.nan.

Parameters **vec** (*numpy.array*) – an Nd array with the final dimension being vectors

```
numpy.array([ x, y, z ])
```

Or an NxM array:

```
numpy.array([
    [x1, y1, z1],
    [x2, y2, z2]
]).
```

Return type A numpy.array the normalized value

pyrr.vector.**set_length**(**args*, ***kwargs*)

Resizes an Nd list of vectors or a single vector to 'length'.

The vector is **not** changed in place.

Parameters **vec** (*numpy.array*) – an Nd array with the final dimension being size 3 (a vector).

Single vector:: numpy.array([x, y, z])

Nd array::

numpy.array([[x1, y1, z1], [x2, y2, z2]

]).

Return type A numpy.array of shape vec.shape.

pyrr.vector.**squared_length**(**args*, ***kwargs*)
Calculates the squared length of a vector.

Useful when trying to avoid the performance penalty of a square root operation.

Parameters **vec** (*numpy.array*) – An Nd numpy.array.

Return type If one vector is supplied, the result with be a scalar. Otherwise the result will be an array of scalars with shape vec.ndim with the last dimension being size 1.

3.15.2 Vector3

Provides functions for creating and manipulating 3D vectors.

pyrr.vector3.**create**(*x=0.0*, *y=0.0*, *z=0.0*, *dtype=None*)

pyrr.vector3.**create_from_matrix44_translation**(**args*, ***kwargs*)

pyrr.vector3.**create_from_vector4**(**args*, ***kwargs*)
Returns a vector3 and the W component as a tuple.

pyrr.vector3.**create_unit_length_x**(*dtype=None*)

pyrr.vector3.**create_unit_length_y**(*dtype=None*)

pyrr.vector3.**create_unit_length_z**(*dtype=None*)

pyrr.vector3.**cross**(*v1*, *v2*)
Calculates the cross-product of two vectors.

Parameters

- **v1** (*numpy.array*) – an Nd array with the final dimension being size 3. (a vector)
- **v2** (*numpy.array*) – an Nd array with the final dimension being size 3. (a vector)

Return type A np.array with shape v1.shape.

pyrr.vector3.**generate_normals**(*v1*, *v2*, *v3*, *normalize_result=True*)
Generates a normal vector for 3 vertices.

The result is a normalized vector.

It is assumed the ordering is counter-clockwise starting at v1, v2 then v3:

```
v1       v3
  \     /
    v2
```

The vertices are Nd arrays and may be 1d or Nd. As long as the final axis is of size 3.

For 1d arrays::

```
>>> v1 = numpy.array( [ 1.0, 0.0, 0.0 ] )
>>> v2 = numpy.array( [ 0.0, 0.0, 0.0 ] )
>>> v3 = numpy.array( [ 0.0, 1.0, 0.0 ] )
>>> vector.generate_normals( v1, v2, v3 )
array([ 0.,  0., -1.])
```

For Nd arrays::

```
>>> v1 = numpy.array( [ [ 1.0, 0.0, 0.0 ], [ 1.0, 0.0, 0.0 ] ] )
>>> v2 = numpy.array( [ [ 0.0, 0.0, 0.0 ], [ 0.0, 0.0, 0.0 ] ] )
>>> v3 = numpy.array( [ [ 0.0, 1.0, 0.0 ], [ 0.0, 1.0, 0.0 ] ] )
>>> vector.generate_normals( v1, v2, v3 )
array([[ 0.,  0., -1.],
       [ 0.,  0., -1.]])
```

Parameters

- **v1** (*numpy.array*) – an Nd array with the final dimension being size 3. (a vector)
- **v2** (*numpy.array*) – an Nd array with the final dimension being size 3. (a vector)
- **v3** (*numpy.array*) – an Nd array with the final dimension being size 3. (a vector)
- **normalize_result** (*boolean*) – Specifies if the result should be normalized before being returned.

pyrr.vector3.**generate_vertex_normals**(*vertices*, *index*, *normalize_result=True*)

Generates a normal vector for each vertex.

The result is a normalized vector.

The index array should list the faces by indexing into the vertices array. It is assumed the ordering in index is counter-clockwise.

The vertices and index arrays are Nd arrays and must be 2d, where the final axis is of size 3.

An example::

```
>>> vertices = numpy.array( [ [ 1.0, 0.0, 0.0 ], [ 0.0, 0.0, 0.0 ], [ 0.0, 1.
↪0, 0.0 ] ] )
>>> index = numpy.array( [ [ 0, 2, 1 ] ] )
>>> vector.generate_vertex_normals( vertices, index )
array([[ 0.,  0., 1.], [ 0.,  0., 1.], [ 0.,  0., 1.]])
```

Parameters

- **vertices** (*numpy.array*) – an 2d array with the final dimension being size 3. (a vector)
- **index** (*numpy.array*) – an Nd array with the final dimension being size 3. (a vector)
- **normalize_result** (*boolean*) – Specifies if the result should be normalized before being returned.

class pyrr.vector3.**index**

x = 0

y = 1

z = 2

class pyrr.vector3.**unit**

x = array([1., 0., 0.])

y = array([0., 1., 0.])

z = array([0., 0., 1.])

3.15.3 Vector4

Provides functions for creating and manipulating 4D vectors.

pyrr.vector4.**create**(*x=0.0*, *y=0.0*, *z=0.0*, *w=0.0*, *dtype=None*)

pyrr.vector4.**create_from_matrix44_translation**(**args*, ***kwargs*)

pyrr.vector4.**create_from_vector3**(**args*, ***kwargs*)

pyrr.vector4.**create_unit_length_w**(*dtype=None*)

pyrr.vector4.**create_unit_length_x**(*dtype=None*)

pyrr.vector4.**create_unit_length_y**(*dtype=None*)

pyrr.vector4.**create_unit_length_z**(*dtype=None*)

class pyrr.vector4.**index**

    w = 3

    x = 0

    y = 1

    z = 2

class pyrr.vector4.**unit**

    x = array([1., 0., 0., 0.])

    y = array([0., 1., 0., 0.])

    z = array([0., 0., 1., 0.])

CHAPTER 4

Development

4.1 Coding Standard

- Follow PEP-8

4.1.1 Modules

Each value type is given its own module.

Functions that reside in these modules include:

- Creation
- Conversion
- Manipulation

4.1.2 Functions

- Existing numpy operations shall be wrapped where it may not be obvious how to perform the operation.

A good example:

```
def multiply(m1, m2):
    # it may not be obvious that the 'dot' operator is the
    # equivalent of multiplication when using arrays as matrices
    # so this is good to help point users in the right direction
    return numpy.dot(m1, m2)
```

A bad example:

```
def add(v1, v2):
    # this functionality is too obvious and too basic
    return v1 + v2
```

- Functions shall not modify data in-place.
- Procedural and Class implementations shall remain in lock-step.

4.1.3 Function names

- Each type shall provide convenience *create* functions for conversions and other initialisations.
- Each type shall have a basic *create* function which returns a zero-ed type, where it makes sense.

A good example:

```
# vector3.py
def create(x=0., y=0., z=0., dtype=None):
    if isinstance(x, (list, np.ndarray)):
        raise ValueError('Function requires non-list arguments')
    return np.array([x,y,z], dtype=dtype)

def create_unit_length_x(dtype=None):
    return np.array([1.0, 0.0, 0.0], dtype=dtype)

def create_unit_length_y(dtype=None):
    return np.array([0.0, 1.0, 0.0], dtype=dtype)

def create_unit_length_z(dtype=None):
    return np.array([0.0, 0.0, 1.0], dtype=dtype)
```

A bad example:

```
# matrix33.py
def create():
    # matrices aren't initialised manually
    # so this isn't ok
    pass
```

- Conversion functions shall be prefixed with *create_from_* followed by the type being converted from:

```
def create_from_matrix(matrix):
    # converts from one type to another
    # this would have to support both matrix33 and matrix44
    pass

def create_from_matrix33(matrix):
    # converts from a very specific type
    # only has to support matrix33
    pass
```

4.1.4 Supplimentary Data

When dealing with arrays and other complex data types, it is helpful to provide methods to identify which array index relates to what data.

A good method to do this is to provide a class definition which contains these values:

```
class index:
    x = 0
```

(continues on next page)

(continued from previous page)

```
    y = 1
    z = 2
    w = 3
```

4.1.5 Tests

- A test class for each module shall be provided in the *pyrr/test* directory.
- A test class shall be the only class in the test module.
- Each source file shall have its own test file.
- Each test function shall have a test case associated with it
- All test cases for a function shall be in a single test function

4.1.6 Layout

These are not strict rules, but are merely suggestions to keep the layout of code in Pyrr consistent.

- Code shall be spaced vertically where doing so helps the readability of complex mathematical functions. Vertical spacing shall be performed at variable or data type boundaries.

A good example:

```
# laying out over multiple lines helps improve readability.
# brackets and parenthesis are laid out to more clearly indicate
# the end of an array / type.
# where appropriate, values are still laid out horizontally.
# provide links where appropriate
#  http://www.example.com/a/link/to/a/relevant/explanation/of/this/code
my_value = numpy.array([
    # X = some comment about how X is calculated
    (0.0, 0.0, 0.0),
    # Y = some comment about how Y is calculated
    (1.0, 1.0, 1.0)
], dtype=[('position', 'float32', (3,))]
)

# laying out parameters vertically can improve readability.
# we'll be less likely to accidently pass an invalid value
# and we can more easily, and more clearly, add logic to the parameters.
some_complex_function_call(
    param_one,
    param_two,
    param_three,
    param_four,
    True if param_five else False,
)
```

A more complicated example:

```
return np.array(
    [
        # m1
        [
```

(continues on next page)

(continued from previous page)

```
            # m11 = 1.0 - 2.0 * (q.y * q.y + q.z * q.z)
            1.0 - 2.0 * (y2 + z2),
            # m21 = 2.0 * (q.x * q.y + q.w * q.z)
            2.0 * (xy + wz),
            # m31 = 2.0 * (q.x * q.z - q.w * q.y)
            2.0 * (xz - wy),
        ],
        # m2
        [
            # m12 = 2.0 * (q.x * q.y - q.w * q.z)
            2.0 * (xy - wz),
            # m22 = 1.0 - 2.0 * (q.x * q.x + q.z * q.z)
            1.0 - 2.0 * (x2 + z2),
            # m32 = 2.0 * (q.y * q.z + q.w * q.x)
            2.0 * (yz + wx),
        ],
        # m3
        [
            # m13 = 2.0 * ( q.x * q.z + q.w * q.y)
            2.0 * (xz + wy),
            # m23 = 2.0 * (q.y * q.z - q.w * q.x)
            2.0 * (yz - wx),
            # m33 = 1.0 - 2.0 * (q.x * q.x + q.y * q.y)
            1.0 - 2.0 * (x2 + y2),
        ]
    ],
    dtype=dtype
)
```

A bad example:

```
# leaving this on a single line would not compromise readability
my_value = numpy.empty(
    (3,)
)
```

The same applies to function definitions:

```
def some_function(that_takes, many_parameters, and_is, hard_to_read, because, its_so,␣
↪big):
    pass
```

Should become:

```
def some_function(
    that_takes,
    many_parameters,
    and_is,
    hard_to_read,
    because,
    its_so,
    big
):
    pass
```

- Code may extend beyond 80 columns, where appropriate.

4.2 Contributing

Pyrr is an Open Source project and contributions are very much welcomed.

If you wish to contribute to Pyrr, do the following:

- Fork Pyrr
- Make your changes
- Send a Pull request

Repository access may be granted on request.

4.2.1 Developers

Pyrr was initially developed by Adam Griffiths of Twisted Pair Development.

Developers and contributors include:

- Adam Griffiths
- Jakub Stasiak
- Korijn van Golen

Is your name left out? Post an issue in Pyrr's bug tracker =)

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

Python Module Index

p

Index

D

E

F

G

H

I

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z