pyglet2d

Release 0.1.0

July 26, 2014

Contents

CHAPTER 1

Contents

1.1 Overview

1.1.1 pyglet2d

Polygon primitives for pyglet.

This package provides a `Shape` object that can be acts as an interface between the libraries polygon and pyglet. The former provides numerical routines for handling shapes, and the latter can process OpenGL bindings. With pyglet2d, you can incorporate 2D shapes into your applications without having to write your own OpenGL calls.

Features

- In addition the standard constructor (from a list or array of points), `Shape``s can be constructed with ``Shape.regular_polygon`, `Shape.circle`, `Shape.rectangle`, and `Shape.from_dict`. The latter is a specification-based constructor that is easy to be use with JSON or YAML.
- `Shape` has two methods that are useful as pyglet callbacks: `Shape.draw` and `Shape.update`. `Shape``s can be given a velocity, and their positions will be updated when ``Shape.update` is called.
- A `Shape` can be scaled and translated using the methods `Shape.scale` and `Shape.translate`, or with in-place arithmetic.
- Alternatively, setting the properties `Shape.center` and `Shape.radius` will translate and scale the shape, respectively.
- Clipping operations provided by polygon are bound to the operators |, &, and ^.
- Additional polygon methods can be accessed directly from the `Shape.poly` attribute, where the `Polygon` object is stored.

- Shortcuts are provided to polygon functions via the boolean methods `Shape.overlaps(other)` and `Shape.covers(other)`.

Example

See `tests/graphics_test.py` for a usage example. This script also serves as a test. Run it to make sure that your graphics pipeline is working correctly:

```
python tests/graphics_test.py
```

Requirements

- Python >= 3.3
- pyglet >= 1.2alpha1. This must be manually installed as it is not on PyPi.
- polygon >= 3
- numpy

Installation

```
pip install pyglet2d --upgrade
```

Documentation

https://pyglet2d.readthedocs.org/

Development

To run the all tests run:

```
tox
```

1.2 Reference

class `pyglet2d.`**`Shape`**(*vertices*, *color=(255, 255, 255)*, *velocity=(0, 0)*, *colors=None*)

classmethod **`circle`**(*center*, *radius*, *n_vertices=50*, ***kwargs*)

Construct a circle.

Parameters

- **center** (*array-like*) –
- **radius** (*float*) –
- **n_vertices** (*int, optional*) – Number of points to draw. Decrease for performance, increase for appearance.
- ****kwargs** – Other keyword arguments are passed to the |Shape| constructor.

covers (*other*)
Check if the shape completely covers another shape.

Returns

Return type bool

distance_to (*point*)
Distance from center to arbitrary point.

point : array-like

Returns

Return type float

draw ()
Draw the shape in the current OpenGL context.

enable (*enabled*)
Set whether the shape should be drawn.

classmethod from_dict (*spec*)
Create a |Shape| from a dictionary specification.

Parameters **spec** (*dict*) – A dictionary with either the fields `'center'` and `'radius'` (for a circle), `'center'`, `'radius'`, and `'n_vertices'` (for a regular polygon), or ‘‘‘vertices’‘. If only two vertices are given, they are assumed to be lower left and top right corners of a rectangle. Other fields are interpreted as keyword arguments.

overlaps (*other*)
Check if two shapes overlap.

Returns

Return type bool

classmethod rectangle (*vertices*, ***kwargs*)
Shortcut for creating a rectangle aligned with the screen axes from only two corners.

Parameters

- **vertices** (*array-like*) – An array containing the `[x, y]` positions of two corners.
- ****kwargs** – Other keyword arguments are passed to the |Shape| constructor.

classmethod regular_polygon (*center*, *radius*, *n_vertices*, *start_angle=0*, ***kwargs*)
Construct a regular polygon.

Parameters

- **center** (*array-like*) –
- **radius** (*float*) –
- **n_vertices** (*int*) –
- **start_angle** (*float, optional*) – Where to put the first point, relative to *center*, in degrees counter-clockwise starting from the horizontal axis.
- ****kwargs** – Other keyword arguments are passed to the |Shape| constructor.

scale (*factor*)
Resize the shape by a proportion (e.g., 1 is unchanged), in-place.

Parameters **factor** (*float*) –

translate (*vector*)
Translate the shape along a vector, in-place.

Parameters **vector** (*array-like*) –

update (*dt*)
Update the shape's position by moving it forward according to its velocity.

Parameters **dt** (*float*) –

1.3 Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

1.3.1 Bug reports

When reporting a bug please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

1.3.2 Documentation improvements

pyglet-polygon could always use more documentation, whether as part of the official pyglet-polygon docs, in docstrings, or even on the web in blog posts, articles, and such.

1.3.3 Feature requests and feedback

The best way to send feedback is to file an issue at https://github.com/hsharrison/pyglet2d/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

1.3.4 Development

To set up *pyglet-polygon* for local development:

1. Fork pyglet-polygon on GitHub.
2. Clone your fork locally:

   ```
   git clone git@github.com:your_name_here/pyglet-polygon.git
   ```

3. Create a branch for local development:

   ```
   git checkout -b name-of-your-bugfix-or-feature
   ```

   Now you can make your changes locally.

4. When you're done making changes, run all the tests with one tox command:

   ```
   tox
   ```

5. Commit your changes and push your branch to GitHub:

   ```
   git add .
   git commit -m "Your detailed description of your changes."
   git push origin name-of-your-bugfix-or-feature
   ```

6. Submit a pull request through the GitHub website.

Pull Request Guidelines

If you need some code review or feedback while you're developing the code just make the pull request.

For merging, you should:

1. Include passing tests (run `tox`) [1].
2. Update documentation when there's new API, functionality etc.
3. Add a note to `CHANGELOG.rst` about the changes.
4. Add yourself to `AUTHORS.rst`.

Tips

To run a subset of tests:

```
tox -e envname -- py.test -k test_myfeature
```

[1] If you don't have all the necessary python versions available locally you can rely on Travis - it will run the tests for each change you add in the pull request.
It will be slower though ...

Authors

- Henry S. Harrison - https://github.com/hsharrison

CHAPTER 3

Changelog

3.1 0.1.0 (2014-07-25)

- First release on PyPI.

CHAPTER 4

Indices and tables

- *genindex*
- *modindex*
- *search*