Drawing Shapes

The `shapes` module is an easy to use option for creating and manipulating colored shapes, such as rectangles, circles, and lines. Shapes can be resized, positioned, and rotated where applicable, and their color and opacity can be changed. All shapes are implemented using OpenGL primitives, so they can be drawn efficiently with Batched rendering. In the following examples *Batch* will be ommitted for brevity, but in general you always want to use Batched rendering for performance.

For drawing more complex shapes, see the Shaders and Rendering module.

Creating a Shape

Various shapes can be constructed with a specific position, size, and color:

```
circle = shapes.Circle(x=100, y=150, radius=100, color=(50, 225, 30))
square = shapes.Rectangle(x=200, y=200, width=200, height=200, color=(55, 55, 255))
```

You can also change the color, or set the opacity after creation. The opacity can be set on a scale of 0-255, for various levels of transparency:

```
circle.opacity = 120
```

The size of Shapes can also be adjusted after creation:

```
square.width = 200
circle.radius = 99
```

Anchor Points

Similar to images in pyglet, the “anchor point” of a Shape can be set. This relates the shape on the x and y axis. For Circles, the default anchor point is the center of the circle. For Rectangles, it is the bottom left corner. Depending on how you need to position your Shapes,

this can be changed. For Rectangles this is especially useful if you will rotate it, since Shapes will rotate around the anchor point. In this example, a Rectangle is created, and the anchor point is then set to the center:

```
rectangle = shapes.Rectangle(x=400, y=400, width=100, height=50)
rectangle.anchor_x = 50
rectangle.anchor_y = 25
# or, set at the same time:
rectangle.anchor_position = 50, 25

# The rectangle is then rotated around its anchor point:
rectangle.rotation = 45
```

If you plan to create a large number of shapes, you can optionally set the default anchor points:

```
shapes.Rectangle._anchor_x = 100
shapes.Rectangle._anchor_y = 50
```

Advanced Operation

Shape Bounding Checks

Some shapes can check whether an instance contains a 2D point via Python's `in` operator.

The point can be any `tuple` of two numbers:

```
from pyglet.shapes import Circle

# Create a circle
circle = shapes.Circle(x=100, y=100, radius=50)

# Turn the circle red if (200, 200) is inside it
if (200, 200) in circle:
    circle.color = (255, 0, 0)
```

Bounding Checks Are In World Space

If you move or alter your viewport, you must account for this when checking coordinates from mouse events. You may find `Vec2` helpful for this since it is a subclass of `tuple`.

Which Shapes Support Bounding Checks?

Instances of the following shape classes support the `in` operator:

- `Circle`
- `Ellipse`
- `Sector`
- `Line`
- `Rectangle`
- `BorderedRectangle`
- `Triangle`
- `Polygon`
- `Star` (acts like a `Circle` with a radius of `(outer_radius + inner_radius) / 2`).

How Does It Work?

Each of these classes implements a `__contains__()` method which accounts for an instance's:

- `rotation` angle
- `anchor_position`

Custom Shapes

> **Tip**
>
> For complex features and performance, please see Shaders and Rendering.

You can import and subclass `ShapeBase` to create custom shapes:

```
from pyglet.shapes import ShapeBase


class MyCustomShape(ShapeBase):
    """A custom shape class. Implementation details omitted."""
    ...
```

This is the base class for shapes in the `shapes` module, which means you can use the other shapes in the module as reference. Please note that this class has the following caveats:

- The `in` operator requires you to add a `__contains__()` method which:
  - accounts for the following values on an instance:
    - `rotation` angle
    - `anchor_position`
  - returns a `bool`

- It will conflict with other metaclass-based helpers due to using `abc.ABC`