Gillcup Graphics Documentation

Release 0.2.0-alpha.1

Petr Viktorin

2017-03-27

Contents

Gillcup Graphics is a Pyglet-based graphics library that allows you to craft scenes out of trees of objects such as layers and sprites. It uses Gillcup animated properties to allow dynamic changes to scene trees, allowing complex animations or games.

CHAPTER 1

Installation and Dependencies

The short story: `pip install gillcup_graphics`

You need to have Python and OpenGL on your computer to run Gillcup Graphics.

Besides that, Gillcup Graphics depends on two Python libraries: Gillcup and Pyglet. These are pulled in automatically when you install via pip or a similar tool.

If you have no clue what that meant, please go read its instructions, and substitute `gillcup_graphics` for the package you would like to install.

If you have an aversion to pip for some reason, use any other tool that can install from PyPI.

CHAPTER 2

Tutorial

Read this to get started quickly.

The tutorial will assume that you know your Python, and have some idea about how computer graphics work – if you aren't familiar with coordinate systems, you should read a different text.

Gillcup Graphics tries to not hide anything from you, but to make your work easier by providing a higher-level graphics layer atop Pyglet. To do non-trivial things (such as advanced transformations, fast batches, shaders, video), or things that are out of scope for a this library (e.g. sound), you should learn something about the underlying concepts. Gillcup Graphics mixes with Pyglet or "raw" OpenGL calls easily.

Hello, World!

Note: This is not an introduction page for dummies.

Despite the name, this page covers lots of info. It also introduces some problems that *you* (the programmer, not the library), will be expected to solve in your applications, and gives an example of how to solve them. In exchange for what looks like boilerplate code now, you get the freedom to later solve the problems in other ways. Ways which which are more appropriate in your case. Ways which, hopefully, let you do more than the library's author imagined.

You'll have to understand what you're doing, and have an idea of what the library is doing for you. If that rubs you the wrong way, you should pick a different library.

To greet the planet, type this into your favorite editor ad run it:

```
from gillcup_graphics import Layer, Text, Window, run

root_layer = Layer()

hi_world = Text(root_layer, 'Hello, World!',
        position=(0.5, 0.5),
        relative_anchor=(0.5, 0),
        scale=(0.001, 0.001),
```

```
    )

Window(root_layer, width=400, height=400)

run()
```

It's quite a lot to take in at once, so let's go through it one line at a time.

Obviously, you will need to import some objects first. For our first example we'll only need four:

```
from gillcup_graphics import Layer, Text, Window, run
```

Once that's done, we'll create a *Layer*:

```
root_layer = Layer()
```

You probably know about the kind of layers you can find in modern painting applications: potentially semi-transparent "sheets" stacked atop one another, with layers further down showing wherever the higher layers are transparent.

Our kind of layers is a somewhat different take on that sheets idea. They are not just stacked, but arranged in trees: each layer can have one or more children, either other layers or different objects. When you move, rotate, or resize a layer, all its children are moved, rotated or resized with it. Layers themselves are completely transparent; the actual drawing is done by the leaves of the scene tree.

The layer at the root of a scene tree – here, `root_layer` – is typically the only graphics object that doesn't have a parent. For the others, we specify a parent as the first argument when creating them:

```
hi_world = Text(root_layer, 'Hello, World!',
```

Here, we're creating a *Text* object in our `root_layer`. We're also passing a certain well-loved string for the Text object to draw. (Note that the assignment is actually unnecessary. The text gets attached to the layer just by having a parent specified; a reference doesn't need to be kept around.)

Unfortunately, just that won't do. We also need to specify where we want the text to be drawn. Let's put it in the middle:

```
        position=(0.5, 0.5),
```

This brings us to the concept of coordinates and sizing. Since we did not specify a *size* for `root_layer`, it has a default size of 1×1 unit. Later, when we display it, it will be stretched (*scaled*) to cover the entire window (in our case, 400×400 pixels). However, regardless of the scaling, the children of the layer deal with a unit square, and (0.5, 0.5) is right in the middle of that.

There's one more problem: as customary in OpenGL (and Pyglet (and math)), the origin of the coordinate system is in the lower-left corner. The `position` argument moves the origin to the specified point. So if we just set the `position` to `(0.5, 0.5)`, our text wouldn't be centered at all!

The solution to this problem is to set the `anchor` property. The anchor specifies the origin of the coordinate system, in whatever units the object itself uses. Just what we need? Well, almost. The catch is in the units. Unlike Layers, a Text's size is not 1×1 units, instead, it's the pixel size of the underlying glyph textures. In our case, it should be something in the ballpark of 600×100, although can't say for sure, because I don't know what font Pyglet uses on your system by default. So, to really center the object, we'd either need to say something like `anchor = (300, 50)`, and ignore the inaccuracy, or create the object, get its size, and set the anchor to half or that.

There's a more elegant approach:

```
        relative_anchor=(0.5, 0),
```

`relative_anchor` maps the given coordinates from an unit square to the object's size and sets `anchor` to the result. So, with `(0.5, 0)`, anchor will always be in the bottom center of the object (even if the text is later resized). This is done with a bit of `magic` that will be dispelled later: with Gillcup, it's not that hard to code such dynamic properties yourself.

The astute reader has no doubt noticed that we did not align the text to the center of its bounding rectangle, `(0.5, 0.5)`, but to the bottom center, `(0.5, 0)`. This is purely for æsthetic reasons: the composition tends to look better when a centered object is a bit above the actual center.

Now that we know something about our objects' geometry, the last argument to the text should start to make sense:

```
        scale=(0.001, 0.001),
    )
```

If we tried to put a roughly 600×100 text in a 1×1 window, we'd see, in the best case, a small part of a giant white blob. More probably, we'd see nothing, because we'd hit the space between the letters.

To bring the text back to scale, we shrink it to a thousandth of its original size, making it roughly 0.6×0.1. That will fit in a 1×1 square easily, leaving nice wide margins around.

And now that our scene is constructed, we can worry about displaying it:

```
Window(root_layer, width=400, height=400)
```

This creates a special Pyglet window that redirects its draw events to the `root_layer`.[1]

And now the last step:

```
run()
```

The run function simply runs a Pyglet main loop, which handles draw events and window close events and various other kinds of events that we haven't taken advantage of just yet.

Finally, you get to see our piece of text (and notice that it doesn't really look all that impressive[2]). But, at least you understand it now! Later in the tutorial you'll learn about animations, input handling, and other objects to draw (such as *sprites*). There's not that much left until your first Gillcup game!

The Clock

Still here? Good.

In this part of the tutorial, we'll learn about the Gillcup clock, the time source for animations. We'll go through the following piece of code, which draws a blinking square:

```
from gillcup_graphics import Layer, Rectangle, Window, run, RealtimeClock

root_layer = Layer()

rect = Rectangle(root_layer,
        size=(0.5, 0.5),
        position=(0.5, 0.5),
        relative_anchor=(0.5, 0.5),
        rotation=45,
    )
```

[1] It also redirects other events, such as mouse and resize ones.

[2] In a real application, you'll probably want to load a known font, and adjust all the sizes to be pixel-perfect. This requires some Pyglet-fu.

```
Window(root_layer, width=400, height=400)

clock = RealtimeClock()


def blink(hide_flag):
    rect.hidden = hide_flag

    if not rect.dead:
        clock.schedule(lambda: blink(not hide_flag), 0.5)

blink(True)

run()
```

Let's fast-forward through the beginning:

```
from gillcup_graphics import Layer, Rectangle, Window, run, RealtimeClock

root_layer = Layer()

rect = Rectangle(root_layer,
        size=(0.5, 0.5),
        position=(0.5, 0.5),
        relative_anchor=(0.5, 0.5),
        rotation=45,
    )

Window(root_layer, width=400, height=400)
```

The only thing that's different from the previous part of the tutorial is that we're using a Rectangle, and the options are a bit different. As you probably can guess, the `rotation` part rotates the rectangle by 45°.

Next comes the all-important line that creates a clock:

```
clock = RealtimeClock()
```

Animations in Gillcup aren't necessarily tied to the system time. You can use different clocks – for example, to render a movie with a fixed frame rate, you wouldn't want to use the system time. Or for a special effect, you might want to use a Clock that runs at a different speed than your main Clock. But here, we happen to be using the RealtimeClock, which is tied to the system time, in seconds.

Now, we define a simple function that can show or hide our square:

```
def blink(hide_flag):
```

The `hidden` attribute sets the visibility of our Rectangle – either it's there, or it isn't.

The final lines of our `blink` function look like this:

```
    if not rect.dead:
```

Let's start with the `schedule` call. The `schedule()` method schedules a function to be called after the given time elapses. So, here, another instance of `blink`, with the `on` argument inverted, will be called 0.5s from the time this line runs. That `blink` will schedule another one, and so on, as long as the `rect` is "alive".

But no longer. That's what the `if` is for. If we wanted to use this in a larger animation, where the blinking rectangle only appears for a short time, we'd only want to schedule when the rectangle is still on the scene. It's good practice to do this even when our animation only includes this one part. Normally, objects are removed from the scene via the *`die()`* method, which sets the `dead` attribute. So, we can use `dead` to see if it's necessary to schedule the next action.

All that's left is calling `blink` for the first time, and running the code:

```
blink(True)
```

The `run` function starts the Pyglet main loop, which, in addition to taking care of drawing our scene, feeds the current time to our `clock`.

Animations

Now that we've got the Clock basics out of the way, let's talk about Animations: a way to smoothly change an object's attribute over time.

First, let's take a look at this piece of code (differences from the previous page are highlighted):

```
from gillcup_graphics import Layer, Rectangle, Window, run, RealtimeClock
from gillcup import Animation

root_layer = Layer()

rect = Rectangle(root_layer,
        size=(0.5, 0.5),
        position=(0.5, 0.5),
        relative_anchor=(0.5, 0.5),
        rotation=45,
    )

Window(root_layer, width=400, height=400)

clock = RealtimeClock()


def blink(on):
    if on:
        clock.schedule(Animation(rect, 'opacity', 1, time=0.3))
    else:
        clock.schedule(Animation(rect, 'opacity', 0, time=0.3))

    if not rect.dead:
        clock.schedule(lambda: blink(not on), 0.5)

blink(True)

clock.schedule(Animation(rect, 'rotation', 0, time=1, timing='infinite'))

clock.schedule(Animation(rect, 'color', 1, 0.5, 0, time=5))

run()
```

Aside from a new import, the first difference is in the blink function:

```
    if on:
```

The `opacity` attribute affects the visibility of our Rectangle – 0 means completely transparent, 1 is fully opaque, anything in between that would make the Rectangle see-through. Opacity, and other properties like `size`, `scale`, `relative_anchor`, `rotation` or `color`, are Gillcup *animated properties*. There are several ways to manipulate them: they can be set when creating a GraphicsObject, as we've seen previously in the tutorial; they become attributes of the object so you can set them normally, as in `rest.opacity = 0.4`, and finally, they can be animated, as we'll see later.

Here, instead of just changing the value, we schedule an `Animation` on our clock. The Animation will smoothly change `rect`'s `opacity` from its current value to `1` in the interval of `0.3` seconds (that is, 0.3 of whatever units of time `clock` uses).

The second changed line is similar – it just animates the value back to zero.

The next added line reads:

```

```

Similar to the above, this smoothly changes the `rotation` from the current value (45) to 0 in the course of 1 second. However, due to the `infinite` timing, it doesn't stop there: it continues past 0° and goes on, through -45° in the second second, -90° in the third, and so on. The final effect is that our square will rotate forever.

The last added line reads:

```

```

Here, we re-color our square from white to orange over the course of 5 seconds. We're animating a *tuple property* – the `color` consists of three values, `red`, `green`, and `blue`. So, we need to pass three values to the Animation. We could also animate the component properties individually, with the same effect. Most tuple properties are like this: `position`'s components are is `x` and `y`, `scale`'s components include `scale_x` and `scale_y`, and so on. Whether you animate the entire tuple or the compoents is up to you.

The `Animation` class has several options and subclasses that let you take control of exactly how the attribute is changed; the `infinite` timing is just one of them.

If you wish to control an animated property in ways unrelated to a clock, take a look in the `gillcup.effect` module. The Effect class provides the control over an attribute; Animation is an Effect subclass that adds time-related stuff. And chaining.

Chaining animations

Animations offer a way to schedule an action to do after they are done. In the following code:

```
from gillcup_graphics import Layer, Rectangle, Window, run, RealtimeClock
from gillcup import Animation

root_layer = Layer()

rect = Rectangle(root_layer,
        size=(0.5, 0.5),
        position=(0.25, 0.25),
        relative_anchor=(0.5, 0.5),
    )
```

```
Window(root_layer, width=400, height=400)

clock = RealtimeClock()


def announce_end():
    print 'done'

animation = Animation(rect, 'x', 0.75, time=1)

clock.schedule(animation)

animation = animation.chain(Animation(rect, 'y', 0.75, time=1))
animation = animation.chain(Animation(rect, 'x', 0.25, time=1))
animation = animation.chain(Animation(rect, 'y', 0.25, time=1))

animation.chain(announce_end)

run()
```

…the `chain()` method does just that: when an Animation is done, `chain`'s argument is scheduled on that Animation's clock. The argument can be another Animation, or just a regular callable.

When doing complex stuff, be aware that each Animation can only be scheduled and run once. If you need to do the same thing several times, you can make an Animation factory (look at `blink` in the first Animation example).

Building animations

Using the `chain` method can be clumsy at times. Another way to build complex animations from the basic building blocks is provided by the + and | operators:

```
from gillcup_graphics import Layer, Rectangle, Window, run, RealtimeClock
from gillcup import Animation

root_layer = Layer()

rect = Rectangle(root_layer,
        size=(0.5, 0.5),
        position=(0.25, 0.25),
        relative_anchor=(0.5, 0.5),
    )

Window(root_layer, width=400, height=400)

clock = RealtimeClock()


def announce_end():
    print('done')

animation = (
    Animation(rect, 'x', 0.75, time=1) +
    0.5 +  # a half-second delay
    Animation(rect, 'y', 0.75, time=1))
animation += [0.5, Animation(rect, 'x', 0.25, time=1), 0.5]
animation += (
    Animation(rect, 'y', 0.25, time=1) |
```

```
    Animation(rect, 'rotation', -90, time=1))
animation += announce_end

clock.schedule(animation)

run()
```

The operators create a larger animation, which runs its components one after the other (+) or in parallel (|).

To create delays, you can "add" numbers to Animations. You can also use lists (or other iterables) of animations; these work as if all their components were "added" toghether (except iterators are evaluated lazily). However, be sure that you don't accidentally pass a raw number of list to `chain` or `Clock.schedule`.

Since each animation can only be scheduled once, be careful to not accidentally schedule any "parts" of a larger animation individually.

If you wish to look how this is implemented, or how to extend it in different ways, see Gillcup's `actions` module. Actions implement the concept of a chainable event; Animation (an Action subclass) adds the changing of an object's attribute.

Processes

Finally, we can schedule actions via a `Process` generator. This allows you to lay out an animation in a procedural manner, as if you were writing a script for a movie. Things that don't happen in a single instant (of Clock time) are handled by yielding actions. This works best for complex animations that require additional processing or looping, but don't "branch out" too much.

```
from gillcup_graphics import Layer, Rectangle, Window, run, RealtimeClock
from gillcup import Animation
from gillcup.actions import process_generator

root_layer = Layer()

rect = Rectangle(root_layer,
        size=(0.5, 0.5),
        position=(0.25, 0.25),
        relative_anchor=(0.5, 0.5),
    )

Window(root_layer, width=400, height=400)

clock = RealtimeClock()


@process_generator
def process():
    while not rect.dead:
        for prop, value in (
                ('x', 0.75),
                ('y', 0.75),
                ('x', 0.25),
                ('y', 0.25)):
            print('Animating {0} towards {1}'.format(prop, value))
            yield Animation(rect, prop, value, time=0.5, easing='sine.in_out')

clock.schedule(process())
```

```
run()
```

Any action[1] that is `yield`-ed is scheduled on the clock, and the rest of the process is chained to it. Note that, again, the "infinite" animation loop is broken when its object dies.

Input

Now, let us handle users' input. We'll start with mouse handling.

To handle mouse input, you need to define event handlers in your *`GraphicsObject`*. These are methods called `on_pointer_motion`, `on_pointer_press`, and so on. Each one is called with a pointer (cursor) identifier, coordinates, and possibly other keyword arguments. The `pointer` identifier is currently always the string `'mouse'`. If you hook your program up to a different input system than a traditional mouse-based computer, such as a multi-touch screen, you can use different pointer identifiers to differentiate between different pointers[1].

The coordinates are more interesting. These come in three arguments, `x, y, z`[2], and represent the position of the pointer in the object's own local coordinates.

The keyword arguments vary depending on the event type. For example, press and drag events will be called with a "button" argument. The methods need to accept any additional arguments that might be passed to them (for example, if someone connects Gillcup Graphics to a tablet, you might start getting pressure/tilt information in the events).

For `motion` and `press` events, the handler can return a true value to "claim" the event. Once claimed, events don't propagate to any objects that might be below the claimer. For `press` events, the claimer will also exclusively get subsequent `drag` and `release` events.

Enough theory! The following example is a bit less minimal than the previous ones, but it should illustrate typical Gillcup Graphics usage better:

```
from __future__ import division

import gillcup
import gillcup_graphics


class DemoRect(gillcup_graphics.Rectangle):
    """A rectangle that goes purple on hover and blue on click/hold/drag
    """
    def __init__(self, parent, clock, **kwargs):
        super(DemoRect, self).__init__(parent, **kwargs)
        self.clock = clock
        self.pointers_in = set()
        self.pointers_pressed = set()

    def on_pointer_motion(self, pointer, x, y, z, **kwargs):
        if pointer not in self.pointers_in:
            self.clock.schedule(gillcup.Animation(self, 'red', 0, time=0.15))
        self.pointers_in.add(pointer)
        return True

    def on_pointer_leave(self, pointer, x, y, z, **kwargs):
        self.pointers_in.remove(pointer)
```

[1] You can also yield numbers or lists/iterables, as with the + and | operators.

[1] Of course, you'd need to do such hooking up manually, by calling *`pointer_event()`* of your root layer.

[2] The `z` coordinate will generally be zero.

```
        if not self.pointers_in:
            self.clock.schedule(gillcup.Animation(self, 'red', 1, time=0.15))
        return True

    def on_pointer_press(self, pointer, x, y, z, button, **kwargs):
        if not self.pointers_pressed:
            self.clock.schedule(gillcup.Animation(self, 'green', 0, time=0.15))
        self.pointers_pressed.add((pointer, button))
        return True

    def on_pointer_release(self, pointer, x, y, z, button, **kwargs):
        self.pointers_pressed.remove((pointer, button))
        if not self.pointers_pressed:
            self.clock.schedule(gillcup.Animation(self, 'green', 1, time=0.15))
        return True


class DraggableRect(gillcup_graphics.Rectangle):
    """A rectangle that can be dragged around, but is "transparent" to hover

    N.B.: This simple mechanism will not work with objects that are scaled or
    rotated.
    """
    def __init__(self, parent, **kwargs):
        super(DraggableRect, self).__init__(parent, **kwargs)
        self.drag_starts = {}

    def on_pointer_press(self, pointer, x, y, z, button, **kwargs):
        self.drag_starts[pointer, button] = x, y
        return True

    def on_pointer_drag(self, pointer, x, y, z, button, **kwargs):
        start_x, start_y = self.drag_starts[pointer, button]
        self.x += x - start_x
        self.y += y - start_y
        return True

    def on_pointer_release(self, pointer, x, y, z, button, **kwargs):
        del self.drag_starts[pointer, button]
        return True


class DemoLayer(gillcup_graphics.Layer):
    def __init__(self, clock, *args, **kwargs):
        super(DemoLayer, self).__init__(*args, **kwargs)
        n = 10
        for x in range(n):
            for y in range(n):
                rct = DemoRect(self, clock, scale=(1 / n, 1 / n),
                    position=(x / n, y / n))
                if (x, y) == (5, 5):
                    rct.rotation = 30
                    rct.blue = 0.7
        DraggableRect(self, size=(0.05, 0.05), color=(0, 1, 0),
            position=(1 / 3, 1 / 3))


if __name__ == '__main__':
```

```
    clock = gillcup_graphics.RealtimeClock()
    layer = DemoLayer(clock)
    window = gillcup_graphics.Window(layer, width=500, height=500)
    gillcup_graphics.run()
```

Let's take it one class at a time, this time.

The `DemoRect` class defines a rectangle that changes color depending on mouse input. In the `on_pointer_motion` and `on_pointer_leave`, we keep track of which pointers are "hovering" over. If there are any, it changes color to cyan (or more precisely, becomes "less red"). When the last pointer leaves, it switches back to white (or, "full red"). The `on_pointer_press` and `on_pointer_leave` do a similar thing with button-press-based events, except for each pointer they also remember which buttons are pressed. From the time a button is pressed over the rectangle, to the time the last such button is released, the square is "less green". This nicely illustrates the dragging concept: even if you drag your mouse out of a rectangle, it still counts as dragging that rectangle, and the "release" events will fire even if the mouse is outside at that time.

The `DraggableRect` uses `press`, `drag`, and `release` events to implement an object that can be dragged around: it remembers where, relative to itself, the drag started, and when dragged, moves so that the pointer is at the same place (again, relative to the DraggableRect).

Finally, the `DemoLayer` is just a container for a hundred instances of `DemoRect` and a `DraggableRect`. One `DemoRect` is rotated and discolored to show what happens with transformations and overlapping.

CHAPTER 3

Module Reference

Shiny graphical flashiness for Gillcup

Gillcup Graphics provides a number of modules:

gillcup_graphics.objects

Drawable objects

This module provides basic objects you can draw and animate.

Perhaps the most important of these is the `Layer`, which does not have a graphical representation by itself, but can contain other objects, such as a *`Rectangle`*, *`Sprite`* or another *`Layer`*. Graphics objects are thus arranged in a scene tree. The tree is rooted at a parent-less Layer[1], which will usually be shown in a *`Window`*.

Each object has a lot of AnimatedProperties that control its position on the screen (relative to its parent), and properties like color and opacity.

The objects are compatible with 3D transformations, but anything outside the *z=0* plane needs custom OpenGL camera setup. Refer to Pyglet documentation for details.

class `gillcup_graphics.`**`GraphicsObject`**(*parent=None*, *to_back=False*, *name=None*, ***kwargs*)

Base class for gillcup_graphics scene objects

Parameters

- **`parent`** – The parent `Layer` in the scene tree
- **`to_back`** – If false (default), the object is inserted at the end of the parent's children list, and thus is drawn after (in front of) its existing siblings. If true, it will be is drawn before (behind) them.
- **`name`** – An optional name of the object. It is stored in the `name` attribute.
- **`kwargs`** – Any animated property (including those from subclasses) can be initialized by passing a value as a keyword argument to `__init__`.

[1] To be precise, the root may be any object. It's just that non-Layers aren't terribly useful here.

draw(***kwargs*)
Draw this object. Overridden in subclasses.

Parameters

- **transformation** – A *GlTransformation* object controlling the current OpenGL matrix.
- **window** – A *Window* for which the drawing is done.

Additional keyword arguments might be present. Unknown ones should be passed to child objects unchanged.

reparent(*new_parent*, *to_back=False*)
Set a new parent

Remove this object from the current parent (if there is one) and attech to a new one (if new_parent is not `None`. The *to_back* argument is the same as for `__init__()`.

Beware that reparenting may throw off the pointer tracking mechanism. Specifically, 'leave' and 'release' events might not fire properly.

Animated Properties:

position
The object's position in space

This is an offset between the parent's anchor and this object's own anchor, in the parent's coordinates.

The individual components are in the `x`, `y`, `z` attributes.

anchor
A point that represents this object for positioning.

The individual components are in the `anchor_x`, `anchor_y`, `anchor_z` attributes.

relative_anchor
Anchor of the object relative to the object's size

When `relative_anchor` is (1, 1), the `anchor` is in the object's upper right corner. When `relative_anchor` is (0.5, 0.5), `anchor` will me in the middle.

This property is only effective if `anchor` is not set by other means.

The individual components are in the `relative_anchor_x`, `relative_anchor_y`, `relative_anchor_z` attributes.

scale
The object's scale.

The individual components are in the `scale_x`, `scale_y`, `scale_z` attributes.

size
The object's natural size

The individual components are in the `width` and `height` attributes.

rotation
Rotation about the object's anchor

Event handlers:

pointer_event(*event_type*, *pointer*, *x*, *y*, *z*, ***kwargs*)
Handle a pointer (mouse) event

Dispatches to on_pointer_<event> methods. See `Layer` for the available handlers.

on_pointer_motion(*pointer*, *x*, *y*, *z*, ***kwargs*)
Handle pointer (mouse) movement

Called when a pointer moves to point (x, y, z) of the object. The coordinates are in the object's own coordinate space.

Return a true value to stop the event from propagating to objects further down.

Remember to override the *hit_test* method so the object's shape is known to the pointer handling machinery.

on_pointer_leave(*pointer*, *x*, *y*, *z*, ***kwargs*)
Handle pointer (mouse) leaving the object

Called when a pointer moves to point (x, y, z), which is outside the object. The coordinates are in the object's own coordinate space.

If the pointer left without known coordinates (this can happen, for example, when the object's transformation matrix becomes singular), all of x, y, z will be set to False (which is equal to 0).

All objects that recieved a 'motion' event for a pointer will recieve a 'leave' event for that pointer, unless they (or their parent chains) are destroyed first.

on_pointer_press(*pointer*, *x*, *y*, *z*, *button*, ***kwargs*)
Handle a pointer (mouse) button press on this object

Called when a pointer button is pressed on point (x, y, z) of the object. The coordinates are in the object's own coordinate space.

Return a true value to "claim" the resulting drag operation. The claiming object will receive 'drag' and 'release' pointer events. Returning true also stops the event's propagation to objects further below.

Subsequent drag and release events follow the pointer even outside the object, including outside the window itself.

on_pointer_drag(*pointer*, *x*, *y*, *z*, *button*, ***kwargs*)
Handle a pointer (mouse) drag on this object

Called when a pointer is dragged (with a button pressed) on point (x, y, z) of the object. The coordinates are in the object's own coordinate space, and may be outside the object (or even the window).

Drag events are only triggered for objects that "claimed" a press event.

on_pointer_release(*pointer*, *x*, *y*, *z*, *button*, ***kwargs*)
Handle a pointer (mouse) button release on this object

Called when a pointer button is released on point (x, y, z) of the object. The coordinates are in the object's own coordinate space, and may be outside the object (or even the window).

Release events are only triggered for objects that "claimed" a press event. The object that claimed a 'press' event for a pointer will recieve a 'release' event for that pointer/button combination, unless it (or its parent chain) is destroyed first.

Other subclassable methods:

is_hidden()
Return true if this object (and any children) aren't active

Hidden objects are not shown and do not handle events.

If the `hidden` attribute of self is true, this method will return true. Otherwise, it checks `self.dead` and other properties that can hide the object, such as zero scale.

hit_test(*_x*, *_y*, *_z*)
Perform a hit test on this object

Return false if the given point (in local coordinates) is "outside" the object.

transform(*transformation*)
Set up the transformation matrix for object
Parameters **transformation** – The *BaseTransformation* object to modify in-place.
No *push()* or *pop()* calls should be made, only transformations.

die()
Destroy this object

Sets up to detach from the parent on the next frame, and calls die() on all children.

Internal methods:

set_animated_properties(*kwargs*)
Initializes animated properties with keyword arguments

Raises an error if any extra arguments are found.

die()
Destroy this object

Sets up to detach from the parent on the next frame, and calls die() on all children.

class gillcup_graphics.**Layer**(*parent=None*, ***kwargs*)
A container for GraphicsObjects

The Layer is unique in that it can contain child objects.

Init arguments are the same as for *GraphicsObject*.

draw(*transformation*, ***kwargs*)
Draw all of the layer's children

class gillcup_graphics.**DecorationLayer**(*parent=None*, ***kwargs*)
A Layer that does not respond to hit tests

Objects in this layer will not be interactive.

class gillcup_graphics.**Rectangle**(*parent=None*, *to_back=False*, *name=None*, ***kwargs*)
A box of color

hit_test(*x*, *y*, *_z*)
Perform a hit test on the rectangle

Animated Properties:

color
Color or tint of the object

The individual components are in the `red`, `green`, `blue` attributes.

opacity
Opacity of the object

class gillcup_graphics.**Sprite**(*parent*, *texture*, ***kwargs*)
An image

Parameters **texture** – A Pyglet image to show in this sprite. You can use the `pyglet.image.load()` to obtain one.

Other init arguments are the same as for *GraphicsObject*.

hit_test(*x*, *y*, *_z*)
Perform a hit test on this object. Uses the sprite size.

Does not take e.g. alpha into account

Animated Properties:

color
Color or tint of the object

The individual components are in the `red`, `green`, `blue` attributes.

opacity
Opacity of the object

class `gillcup_graphics.`**Text**(*parent*, *text*, *font_name=None*, ***kwargs*)
A text label

Parameters

- **text** – A string to display n this label
- **font_name** – Name of the font to use. See Pyglet documentation for more info on using fonts.

Other init arguments are the same as for *GraphicsObject*.

Note: The API regarding font size is experimental.

---

hit_test(*x*, *y*, *_z*)
Perform a hit test on this object. Uses the bounding rectangle.

size
The natural size of the text

This property is not animated, and cannot be changed.

Returns the size of the entire text, i.e. doesn't take `characters_displayed` into account.

The `width` and `height` attributes contain the size's individual components.

Animated Properties:

color
Color or tint of the object

The individual components are in the `red`, `green`, `blue` attributes.

opacity
Opacity of the object

font_size
The size of the font

characters_displayed
The maximum number of characters displayed

gillcup_graphics.mainwindow

Utilities for interfacing gillcup_graphics with the rest of the world

class gillcup_graphics.**Window**(*layer*, **args*, ***kwargs*)
A main window

A convenience subclass of pyglet.window.Window that shows a *Layer*

Parameters **layer** – The layer to show. Its *scale* will be automatically adjusted to fit the window (and re-adjusted when the window's size changes).

Other arguments are passed to Pyglet's window constructor. The most common arguments to pass are:

Parameters

- **width** – Width of the window
- **height** – Height of the window
- **caption** – Caption that appears in the title, taskbar, etc.
- **fullscreen** – If true, the window will cover the screen
- **resizable** – If true, the user can resize the window

Other arguments are explained in the Pyglet documentation.

manual_draw()
Draw the contents outside of the main loop

class gillcup_graphics.**RealtimeClock**
A gillcup.Clock tied to the system time

Note that the Pyglet main loop must be running (or the Pyglet clock must be ticked otherwise) for this to work.

gillcup_graphics.**run**()
Runs the Pyglet main loop. Alias for pyglet.app.run().

gillcup_graphics.transformation

Transformation objects

The Transformation interface is implemented by several classes that have used whenever info about graphics objects' position, scale, rotation, etc. are needed.

A graphic object's transform method takes a Transformation object and calls its translate, scale, rotate or premultiply methods. While premultiply is the most general, the other methods are more straightforward to use and often much faster.

For drawing, a GlTransformation object, which will update the OpenGL state directly, is passed to the method. For hit tests and mouse events, a PointTransformation is used.

Each transformation object implements a stack modeled on the OpenGL matrix stack: any state can be saved with push, and the last-pushed state restored with pop. The state context manager simplifies working with the stack.

class gillcup_graphics.transformation.**BaseTransformation**
Base for all transformations: contains common functionality

Transformations are based on a 3D affine transformation matrix (a 4x4 matrix where the last column is [0 0 0 1])

state
Context manager wrapping push() and pop()

push()
Push the matrix state: the corresponding pop() will return here

You probably want to use `state` instead.

pop()
Restore matrix saved by the corresponding push() call

reset()
Reset the matrix to identity

translate(*x=0*, *y=0*, *z=0*)
Change the transformatin to represent moving an object

The object is moved, without rotating, along the vector [x y z].

rotate(*angle*, *x=0*, *y=0*, *z=1*)
Change the transformatin to represent rotating an object

The object is rotated *angle* degrees along the axis specified by the vector [x y z]. This must be an unit vector (i.e. $x^2 + y^2 + z^2 = 1$)

scale(*x=1*, *y=1*, *z=1*)
Change the transformatin to represent scaling an object

The object is rotated *angle* degrees along the axis specified by the vector [x y z]. This must be an unit vector (i.e. $x^2 + y^2 + z^2 = 1$)

premultiply(*values*)
Premultiply the given matrix to self, in situ

Parameters **values** – An iterable of 16 matrix elements in row-major (C) order

class gillcup_graphics.transformation.**GlTransformation**
OpenGL implementation: affects the OpenGL state directly

class gillcup_graphics.transformation.**PointTransformation**(*x*, *y*, *z*)
Transformation for a single point

The `point` attribute corresponds to the vector given to the constructor transformed by whatever transformation was applied to this object.

class gillcup_graphics.transformation.**MatrixTransformation**
A Transformation with a full, queryable result matrix.

__len__()
Return *16*, the size of the matrix.

__getitem__(*item*)
Get a number from the matrix. Supports (x, y) pairs, or single ints.

Note that __len__ and __getitem__ are one variant of the iteration protocol: MatrixTransformation supports iter() as well.

transform_point(*x=0*, *y=0*, *z=0*)
Return the given vector multiplied by this matrix

Returns a 3-element iterable

gillcup_graphics.effectlayer

A Layer that provides some bling

With EffectLayer, it is possible to colorize, fade, or pixelate groups of graphics objects.

This wors by rendering the layer's contents to a texture, and then drawing the texture with effects applied. Currently, it needs the Framebuffer Object (FBO) OpenGL extension to work.

Note: This code is experimental. Expect it to not work.

class gillcup_graphics.**EffectLayer**(*parent=None*, ***kwargs*)
A Layer that can colorize, fade, or pixelate its contents as a whole

Animated Properties:

color
Color or tint of the object

The individual components are in the `red`, `green`, `blue` attributes.

opacity
Opacity of the object

mosaic
Pixelation of this layer

For example, if mosaic=(2, 4), the layer will be drawn using 2×4 blocks

need_offscreen()
Return true if off-screen rendering is needed

Off-screen rendering is only done if needed, i.e. if `color`, `opacity` or `mosaic` don't have their default values.

Subclasses should extend this method if they need off-screen rendering in more circumstances.

class gillcup_graphics.effectlayer.**RecordingLayer**(*parent=None*, ***kwargs*)
A layer that records its contents as an image

After this layer is drawn, the picture is available in the `last_image` attribute as a pyglet ImageData object.

get_image(*width*, *height*)
Draw this layer in a new invisible window and return the ImageData

The most interesting classes of each module are exported directly from the gillcup_graphics package:

- *GraphicsObject* (from *gillcup_graphics.objects*)
- *Layer* (from *gillcup_graphics.objects*)
- *DecorationLayer* (from *gillcup_graphics.objects*)
- *Rectangle* (from *gillcup_graphics.objects*)
- *Sprite* (from *gillcup_graphics.objects*)
- *Text* (from *gillcup_graphics.objects*)
- *Window* (from *gillcup_graphics.mainwindow*)
- *RealtimeClock* (from *gillcup_graphics.mainwindow*)
- `run()` (from *gillcup_graphics.mainwindow*)

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

Python Module Index

g

Index

Symbols

A

B

C

D

E

F

G

H

I

L

M

N

O

P

R

S

T

W