Gillcup Documentation

Release 0.1

Petr Viktorin

2012-05-20

CONTENTS

Contents:

CHAPTER

ONE

INTRODUCTION

Gillcup is a 2D animation library.

It is intended for both scripted (i.e. the entire animation is known ahead of time), and interactive animations.

Gillcup is modular: the core provides a timer and animated objects with nothing relating to graphics. Visualization classes based on Pyglet are provided in the `gillcup.graphics` module.

Gillcup depends on nothing but Pyglet, easing deployment. Pyglet, in turn, depends on Python and OpenGL.

CHAPTER

TWO

THE GILLCUP TUTORIAL

Contents:

2.1 The First Steps: Drawing a Rectangle

This section is for beginner animators (although Python knowledge is required). If you have some experience with object-oriented graphics, skip to the last section and see if you understand the code there.

2.1.1 Running Code

Let's get some results! Create a Python file and paste the following into it:

```python
from gillcup.graphics import mainwindow
from gillcup.graphics.layer import Layer
rootLayer = Layer()


mainwindow.run(rootLayer)
```

When you run it, you will get a blank window.

Let's see what those four lines mean:

```python
from gillcup.graphics import mainwindow
```

This imports the `mainwindow` module, which has convenience functions for displaying animations you might create. You can use them as-is, or look at the code for an example of how to write your own.

Next up:

```python
from gillcup.graphics.layer import Layer
rootLayer = Layer()
```

Here we use the `Layer` class. Think of a Layer as the term is used in image processing application: a transparent sheet on which objects can be drawn. In this case, we initialize an empty Layer and nothing more. That's why our window was empty!

On to the last line:

```python
mainwindow.run(rootLayer)
```

This is where we call a convenience function to "run" our Layer, that is, display it and any animations that are on it. There's not much to see now, but that's about to change in the next section!

2.1.2 Displaying Things

In this section, we will draw a rectangle in the middle of the window.

Modify your Python file to read:

```
from gillcup.graphics import mainwindow
from gillcup.graphics.layer import Layer
from gillcup.graphics.colorrect import ColorRect
rootLayer = Layer()

rect = ColorRect(rootLayer)

mainwindow.run(rootLayer)
```

When you run this, you will see that the window is now grey.

There are two added lines: an import and an instantiation of a ColorRect class. As the name suggests, the ColorRect class represents a colored rectangle. In this case, the rectangle is grey, and completely covers the window. We'll change that later; first we need to explain a debugging call that will be useful later.

2.1.3 Scene Trees

The argument we gave to our ColorRect is the "parent". Every Gillcup graphics object can have a parent layer, which is given as the first argument to the constructor. This makes the object attach to its parent.

Of course, Layers themselves are also such objects, which can lead to complex structures called "scene trees". If you're ever confused about the structure of what is displayer on the screen, you can call:

```
rootLayer.dump()
```

This will print out the scene tree to standard output. In our case, we have:

```
Layer x(768, 576)
  ColorRect
```

This is a very simple scene tree consisting of a Layer scaled to 768x576 (the size of the window), and a ColorRect. The indentation shows the ColorRect is contained in the Layer.

2.1.4 Color

I don't know about you, but I don't like grey too much. It would be much more interesting to color our screen, say, blue. Of course I wouldn't say that if it wasn't ridiculously easy to do; just change the ColorRect call to:

```
rect = ColorRect(rootLayer, color=(0, 0, 1))
```

and run the script again. Looking at the output, maybe this particular hue wasn't such a good idea, but keep with me.

This example shows us two things:

- We can give named arguments to constructors of Gillcup graphics objects to set attributes such as color.
- Colors in Gillcup are given as RGB triples of floats in the 0..1 range.

These are true generally, although I might add that setting attributes also works the Python way, as you can test by adding the following line just above the mainwindow.run() call:

```
rect.color = (1, 1, 0)  # No! Yellow!
```

2.1.5 Position and Scale

There are more attributes that you can set than colors, of course. For animations, we will want to make ourselves familiar with three of these: position, scale, and anchorPoint.

First, an introduction to Gillcup's geometry: The "x" axis points right, and the "y" axis points up (i.e. not down as you may be used to from GUI toolkits). The origin — that is, the (0, 0) point – is in the lower left of the window. This is standard in math, OpenGL and Pyglet.

The mainwindow.run() scales (resizes) its root layer so that the (1, 1) point is in the upper right corner of the window.

The ColorRect class introduced earlier has the same geometry as the screen: (0, 0) is in the lower left corner, (1, 1) in the upper right. It's easy to see now why our rectangle covers the screen.

Let's now make the rectangle a bit smaller, so we can see that it is actually a rectangle. Change the ColorRect call to this:

```
rect = ColorRect(rootLayer, scale=(0.5, 0.5))
```

This will resize the rectangle by 1/2 in each direction, so that it rectangle only covers a quarter of the screen. But, the resize is relative to the origin, the lower left corner of the window. Wouldn't it be better to have the rectangle centered?

To change the point a graphic object scales around, we set the anchorPoint property. You can change our call to:

```
rect = ColorRect(rootLayer, scale=(0.5, 0.5), anchorPoint=(0.5, 0.5))
```

As you can see, that didn't work. This is because anchorPoint is not just the central point for scaling, but also for rotations, and, most importantly, for the position of the object.

So, we will also need to set the position attribute. The position specifies where, relative to the parent, an object's anchorPoint is. We would like it to be in the middle of the layer.

Our instantiation line is getting longer and longer, so we may want to split it in seeral pieces. (In real life, if you find out you are reusing the same arguments over and over, you're encouraged to subclass or make a factory function.):

```
rect = ColorRect(rootLayer)
rect.scale = (0.5, 0.5)
rect.anchorPoint = (0.5, 0.5)
rect.position = (0.5, 0.5)
```

And now, we have a nice rectangle centered on the screen!

2.1.6 Rotation and Opacity

The final two attributes I want to cover are rotation and opacity. These are straightforward to use, as they are just numbers. Just keep in mind that the rotation is in degrees. (Radians are, unfortunately, a bit unwieldy for animation use.) So, for a see-through rectangle on its side, add these:

```
rect.rotation = 45
rect.opacity = 0.75
```

What is this?, I hear you say. It's not a rectangle any more! Read the next section for an explanation (or excuse, rather).

2.1.7 A Note on Aspect Ratio

Gillcup does not care about the spect ratio. It is your own responsibility to scale your layers to the correct size, or use a rectangular window (using the width and height arguments to mainwindow.run()).

The reason is that there is no universally right solution to this problem; letting you fix it hovewer you want is better than having you undo Gillcup's fix first and then aply your own.

2.1.8 This Lesson's Code

Here is the complete code we've come up with, commented for those that skipped to here:

```
# Boilerplate
from gillcup.graphics import mainwindow
from gillcup.graphics.layer import Layer
from gillcup.graphics.colorrect import ColorRect
rootLayer = Layer()


# Create a rectangle and set various attributes on it
rect = ColorRect(rootLayer)
rect.scale = (0.5, 0.5)  # Resize to 1/2
rect.anchorPoint = (0.5, 0.5)  # Move origin to the center
rect.position = (0.5, 0.5)  # Move to the center of the screen

rect.rotation = 45  # Rotate 45°
rect.opacity = 0.75  # Make it see-through a bit

rect.color = (1, 1, 0)  # Make it yellow


# Show the result
mainwindow.run(rootLayer)
```

2.2 The Gillcup Magic: Animations

So, you've gone throught the first part of the tutorial, and you still think you can take more? Yes? Then here we go.

2.2.1 Revive the Rectangle

Again, let's start with some simple code:

```
from gillcup.graphics import mainwindow
from gillcup.graphics.layer import Layer
from gillcup.graphics.colorrect import ColorRect
rootLayer = Layer(timer=mainwindow.getMainTimer())

rect = ColorRect(rootLayer, color=(1, 1, 0))
rect.scale = (0.5, 0.5)
rect.anchorPoint = (0.5, 0.5)
rect.position = (0.5, 0.5)

rect.rotateTo(45)

mainwindow.run(rootLayer)
```

This might seem very familiar to you – it's very similar to the code from the first part of the tutorial. What is different now is that it's using animation API for the rotation in the lines:

```
rootLayer = Layer(timer=mainwindow.getMainTimer())
```

```
rect.rotateTo(45)
```

If we want to animate, the layers need to know the current time; the first line says that we are using a normal timer that reflects the system clock's seconds and starts when the main window is displayed. This is great for interactive animations; but other timers are possible: if we wanted to render a movie, we'd use a timer that advances by a fixed amount after we're drawn each frame.

The second line applies a very uninteresting animation to the rectangle, rotating it instantly by 45°. This is about as disappointing as the blank window from the previous part of the tutorial; let's fix that.

2.2.2 Animations

Change the rotation line to:

```
rect.rotateTo(45, time=3)
```

If you can't see anything new, make sure that you're running this as a script, the run() convenience function doesn't like to be called multiple times.

If you saw movement, congratulations! You've made your first Gillcup animation. As you can see, the setting of the rotation took 3 seconds [1] to complete, smoothly transitioning from the previous value (0°) to 45°.

Tuple Animations

Try adding the following line:

```
rect.moveTo(0.25, 0.25, time=2)
```

As you can see, it's not only numbers that can be animated: tuples of numbers can, too.

Be aware, however, that Gillcup's default animations only animate numbers and tuples of numbers. Don't set a property you want to animate to a list, for example.

Going Lower: the Animate Method

Now, replace the animation lines with this:

```
rect.animate('rotation', 45, time=3)
rect.animate('position', (0.25, 0.25), time=3)
```

As you can see, it does the same thing. In fact, the rotateTo and moveTo methods are just a shortcut for "animate".

The animate method can be used on any instance attribute of an AnimatedObject (of which Layer & co. are subclasses). The animation system doesn't know that "rotation" and "position" have some meaning when the object is drawn. What this means for you is that you can animate your own attributes, even on your own objects (as long as you subclass them from gillcup.animatedobject.AnimatedObject). Remember this, for it will be useful later.

Delayed Actions

Now, delete your animation lines once again and put in the following instead:

[1] The default timer's time happens to be in seconds; the actual animations don't care about what the unit of time is.

```
rect.rotateTo(45, time=3, dt=1)
```

This time, the animation starts a second after the window is shown.

2.2.3 Actions and Effects

Once again replace the animation line, this time with:

```
action = rect.rotationTo(45, time=3)
rect.apply(action, dt=1)
```

It should do the same thing as last time.

What we are doing here is first obtaining an animation object, and then applying it later. This approach has the advantage that you can keep the animation around, and use it at a later time.

The animation object is the only thing you need to keep; any AnimatedObject can be used to apply it (as long as it shares the same timer [2]).

Note that the "dt" (delay) argument is passed to apply().

To make this a bit less confusing, let's introduce a bit of terminology. We have been using the term "animation" somewhat vaguely for a combination of two different concepts that Gillcup calls Actions and Effects.

Actions

An Action is something that can be scheduled for the future: think of it as a delayed function call. The object that the rotationTo method returns is such an action. In our case, the action starts an animation when called.

You can use any function as an action. Try putting the following before your mainwindow.run call:

```
def printMessage():
    print "Starting to turn!"
rect.apply(printMessage, dt=1)
```

Of course, an Action can be more than just a packaged function call. EffectAction, which rotationTo and friends return, knows about the Effect it's going to apply, and it can use this knowledge to its advantage.

Effects

Effects are, in essence, attribute modifiers. They change an AnimatedObject's attribute, usually based on the time and the attribute's previous value.

Effects "last" for a longer time, as opposed to Actions which are instantaneous (as far as Gillcup's timer is concerned).

The simplest useful effect, which we have been using, just linearly interpolates between the old value and a new value. There are, of course, lots of other behaviors for effects, which we'll cover later. But even the simplest effects have one useful functionality: chaining.

2.2.4 Chaining Effects and Actions

Just to make sure we're on the same ground, I'll give the whole code for this example:

[2] The timer of the applying object will be used for the animation. You can theoretically use this for interesting results, but generally mixing multiple timers is just confusing.

```
from gillcup.graphics import mainwindow
from gillcup.graphics.layer import Layer
from gillcup.graphics.colorrect import ColorRect
rootLayer = Layer(timer=mainwindow.getMainTimer())

rect = ColorRect(rootLayer, color=(1, 1, 0))
rect.scale = (0.5, 0.5)
rect.anchorPoint = (0.5, 0.5)
rect.position = (0.5, 0.5)

action = rect.movementTo(0, 0, time=1)
action.chain(rect.movementTo(0.5, 0.5, time=1))
rect.apply(action)

mainwindow.run(rootLayer)
```

What happens here? Our yellow friend moves to a corner, and then goes back.

As you can see, we called the chain() method to get this behavior. What an Effect's chain() method does is simple: it schedules the given Action to happen when the Effect is done.

We have, however, been using an Action's chain(). This does pretty much the same: it chains the scheduled actions on the Effect it applies. Or, if it's not an EffectAction, runs them just after it's done.

The chain method will also take a "dt" argument to delay the new Action.

If you are using plain functions, you can wrap them in gillcup.action.FunctionAction to get the chain() method. Or, just schedule whatever you're chaining for the same time as your function (scheduling is stable: if two things are scheduled for the same time, they will happen in the order they were scheduled).

2.2.5 The Rainbow Cycle

Disclaimer: Sit in a well-lit room, a good distance from the screen. If you fear epileptic seizures, stop reading and forget about making animations.

Please note that Actions and Effects are intended for one-time use. Don't schedule the same Action, or apply the same Effect more times. If you need to, create an equivalent Action and Effect.

This doesn't apply to plain functions, since when they're scheduled, a new Action is always made. So, you can do the following for an infinite loop:

```
def rainbow():
    # Cycle through the colors...
    action = rect.animate('color', (1, 0, 0), time=0.2)
    action = action.chain(rect.animation('color', (1, 1, 0), time=0.2))
    action = action.chain(rect.animation('color', (0, 1, 0), time=0.2))
    action = action.chain(rect.animation('color', (0, 1, 1), time=0.2))
    action = action.chain(rect.animation('color', (0, 0, 1), time=0.2))
    action = action.chain(rect.animation('color', (1, 0, 1), time=0.2))
    # ... then go one more time
    action.chain(rainbow)

rect.apply(rainbow)
```

Try it! If you haven't deleted your movement animation, you get to see that the color cycle and the movement co-exist with each other peacefully.

You also get to see that you have to be a bit careful when using Gillcup's methods: there's "animate", which makes an animation and applies it immediately, and "animation", which creates an animation and gives you an Action that starts it. The graphic object convenience functions also come in such pairs: rotateTo/rotationTo, moveTo/movementTo, and so on. Be sure you know which one you're using.

Another thing you might have noticed is that both flavors of animation methods and chain() all return an Action object. Notice the above pattern of chaining and setting the chain's end; it may useful to you.

2.2.6 Infinite Effects

Replace your animation by the following:

```
rect.rotateTo(90, time=1, infinite=True)
```

This shows how you can make an infinite effect. It rotates out rectangle by 90° in 1 second, then instead of ending, it goes on rotating.

It doesn't make much sense to chain anything to such an animation, but if you do, the chained Action will run at the time specified by the "time" argument, not when the effect is done.

2.2.7 What Was Before Us

This section's animation code will look like this:

```
rect.rotateTo(90, time=1, infinite=True)
rect.rotateTo(0, time=5, dt=2)
rect.animate('color', (1, 0, 0), dt=2)
```

What happens here? The rectangle is rotating happily at the speed of 90°/s, and 2 seconds later it changes to red and starts rotating back to its original position.

You might notice, though, that when the rectangle turns red, it doesn't suddenly start rotating back. The transition is smooth. Why is that?

When I said earlier that a simple Effect interpolates between an old value and a new value, I was only telling half of the truth. The "old value" includes any effect that was on the attribute before. That is, by default an Effect interpolates between a *dynamic value* and the given endpoint.

2.2.8 Dynamic Attributes

Replace you animation code by this:

```
import math
def sinOfTime():
    return math.sin(rect.timer.time) * 90
rect.setDynamicAttribute('rotation', sinOfTime)
```

As you can see, you can set any function you want to work as an attribute getter for AnimatedObjects. It will play along nicely with other effects, too.

Also, the rect.timer.time construction is new. I hope it doesn't need much explanation, though. You can use mainwindow.getMainTimer().time for the same effect.

2.2.9 Effects Can Be Animated

Now, try this:

```
import math
def blueCyan():
    sinOfTime = math.sin(rect.timer.time * 5)
    return 0, 0.5 + sinOfTime / 2, 1
def redYellow():
    sinOfTime = math.sin(rect.timer.time * 10)
    return 1, 0.5 + sinOfTime / 2, 0
rect.setDynamicAttribute('color', blueCyan)
```

Try both color schemes, but then put blueCyan back and add:

```
action = rect.animate('color', (1, 1, 0), time=2, dt=1, keep=True)
action.effect.setDynamicAttribute('value', redYellow)
```

Effects are AnimatedObjects, and can be themselves animated. You just have to know what attributes to look for. One useful attribute, "value", represents the effect's "goal"; it is the value set by the animation method that created the effect.

What we did above is animate this "goal", thus making the effect interpolate towards an animation. And since there was an animation in the beginning too, we interpolated between two animations!

You can build arbitrarily complex animations by using this scheme.

If you are not dead tired by now, you might have noticed the "keep" attribute above. Read on to know what it does.

When Effects end

When a simple effect ends, it is replaced by a much simpler effect that always gives a constant value. This is done to prevent long "chains" of effects from using up memory and the processor, because, as shown above, effects are not replaced when animating.

The animation functions try to be smart and detect when you are doing advanced stuff and disable this behavior if you are not applying just a simple effect. For example, the infinite rotation above is not killed in this way.

However, it is not always possible to detect when you're going to need the effect after it ends, so to be on the safe side add a keep=True argument to the animation method when you manipulate the Effect later.

2.2.10 Dummy effects

[XXX]

2.3 What needs to be written

[XXX] As you can see, this section is still unfinished.

2.3.1 Scene Tree Dumps

[XXX]

Reparenting

2.3.2 Easing

[XXX]

2.3.3 Sprites

[XXX]

2.3.4 Text

[XXX]

2.3.5 Alpha Layers and Pixelization

[XXX]

2.3.6 The Demo

[XXX]

CHAPTER

THREE

MODULE INDEX

Contents:

3.1 gillcup.timer

In Gillcup, animation means two things: running code at specified times and changing object properties with time.

You will notice that the preceding sentence mentions time quite a lot. But what is this time?

You could determine time by looking at the computer's clock, but that would only work with real-time animations. When you'd want to render a movie, where each frame takes 2 seconds to draw and there are 25 frames per second, you'd be stuck.

That's why Gillcup introduces a flexible source of time: Timers. These are objects with three attributes:

- time, which gives the current time ("now") on the timer,
- advance(dt), which advances the timer by "dt" units, and
- schedule(dt, action), which schedules an "action" to happen "dt" time units from "now"

3.1.1 Obtaining timers

The run() function from graphics.mainwindow automatically creates a timer that is tied to the system clock (or, rather, the Pyglet clock; this means it won't run if the Pyglet main loop is not running); this is a singleton that can be obtained through the mainwindow.getMainTimer() function.

New timers can be created by instantiating the gillcup.timer.Timer class; of course, you must call advance() on these manually.

3.1.2 Timers and layers

Each graphics object can have its timer, which is used for animations. It can be set as the "timer" argument to __init__; if left out, the object inherits its parent timer. It can be changed by setting the "timer" attribute.

If an object has no timer and an animation is requested on it, it searches up through its parent and its descendants before it dies with an error, and the mainwindow convenience functions automatically set the root layer's timer if one's not set.

An object doesn't actually need the timer itself; it is only used when creating animations on it. Even then, a different timer can be specified. However, having the timer available is very convenient.

3.1.3 Module Contents

class `gillcup.timer.`**`Timer`**(*time=0*)

Keeps track of time.

Use advance() to push time forward. This is done manually to allow non-realtime simulations/renders. Time can only be moved forward, because events can change state.

Use schedule() to schedule an event for the future.

`advance`(*dt*)

Call to advance the timer

Steps the timer dt units to the future, running any scheduled Actions.

`schedule`(*dt*, **actions*)

Schedule actions to run "dt" time units from the current time

Scheduling is stable: if two things are scheduled for the same time, they will be called in the order they were scheduled.

Returns the first action scheduled

3.2 gillcup.action

[XXX]

class `gillcup.action.`**`Action`**

Something that can be scheduled: a discrete event.

Also, other Actions can be chained to it. These will be run when the "parent" Action, or an effect applied by it, finishes.

Actions may not be callable. If they are, they won't be scheduled as actions.

`chain`(*action*, **others*, ***kwargs*)

Schedule an Action (or more) at the end of this Action

The dt argument can be given to delay the runnin of the execution by the specified time.

For EffectAction, the actions are scheduled after the applied effect ends.

If this action has already finished, the chained ones are scheduled immediately.

`delay`(*dt*)

Schedule a null action at time dt (useful in chaining)

`run`(*timer*)

Run this action.

Called from a Timer.

class `gillcup.action.`**`EffectAction`**(*effect*, **args*, ***kwargs*)

An Action that applies an effect when run

effect is applied when this Action is run; the timer, args and kwargs are passed to it.

args should be the object and attribute to apply the Effect to.

class `gillcup.action.`**`FunctionAction`**(*func*, **args*, ***options*)

An Action that executes a function when run

func is called when this Action is run; args are passed to it

Additional options:

- kwargs: a dict of named arguments to pass to the function
- passTimer: if True, the timer will be passed as an additional named argument

class `gillcup.action.`**`WaitForAll`**`(`**actions*`)`
An Action that waits for other actions, and runs when they all are run

class `gillcup.action.`**`WaitForAny`**`(`**actions*`)`
An Action that waits for other actions, and runs when any of them is run

3.3 gillcup.animatedobject

[XXX]

class `gillcup.animatedobject.`**`AnimatedObject`**`(`*timer=None*`)`
An objects whose attributes can be animated

Animated attribute must be instance (not class) attribute in order to work. Every animated attribute must be assigned a normal value before it can be animated. Usually this is done in the constructor.

Each AnimatedObject needs a timer to animate. This can be either through an argument to the animate method, or in an instance attribute. The constructor takes a timer value to set the instance attribute to.

`animate``(`*attribute*, *value*, *dt=0*, *timer=None*, ***options*`)`
Animate the given attribute

Calls self.apply(self.animation(...), dt=dt). Returns the resulting Action.

`animation``(`*attribute*, *value*, ***options*`)`
Return an animation Action for the given attribute.

When this Action is run, the given attribute will be gradually set to the new value. The style of the animation is given by options.

See `gillcup.effect.animation()` for what options are available.

`apply``(`*action*, *dt=0*, *timer=None*`)`
Schedule action on this object's timer

dt is the time in which the Action is to be executed (measured from the timer's current time). timer can be given to specify the timer to use; if None, self's timer will be used

`dynamicAttributeSetter``(`*attribute*, *getter*`)`
Returns an Action to set an attribute getter

After the returned Action runs, the given getter function will be used to provide values for the given attribute.

`setDynamicAttribute``(`*attribute*, *getter*, *dt=0*, *timer=None*`)`
Set a getter for an attribute

If dt==0, sets getter as the attribute getter for the given attribute.

Otherwise, calls self.apply(self.dynamicAttributeSetter(attribute, getter), dt=dt).

3.4 gillcup.easing

[XXX] Adapded partially from Robert Penner's Easing Equations, as they appear in the Qt library. The original license follows:

`gillcup.easing.`**`bounce`**(*amplitude*)
  Bounce easing

`gillcup.easing.`**`circ`**(*t*)
  Circular easing

`gillcup.easing.`**`cubic`**(*t*)
  Cubic easing

  t → t**3

`gillcup.easing.`**`elastic`**(*period*, *amplitude=1*)
  Elastic easing

`gillcup.easing.`**`exp`**(*t*)
  Exponential easing

`gillcup.easing.`**`linear`**(*t*)
  Linear interpolation

  t → t

`gillcup.easing.`**`normalized`**(*func*)
  Decorator to normalize another easing function

  Normalizing is done so that f(0) == 0 and f(1) == 1.

`gillcup.easing.`**`overshoot`**`(`*amount*`)`
Overshoot easing

`gillcup.easing.`**`quad`**`(`*t*`)`
Quadratic easing

t → t**2

`gillcup.easing.`**`quart`**`(`*t*`)`
Quartic easing

t → t**4

`gillcup.easing.`**`quint`**`(`*t*`)`
Quintic easing

t → t**5

`gillcup.easing.`**`showcase`**`(`*items=['(poly)', 'sin', 'exp', 'circ', 'elastic_example', 'overshoot_example', 'bounce_example']*`)`
Show graphs of the easing functions in this module

`gillcup.easing.`**`sin`**`(`*t*`)`
Sinusoidal easing

Quarter of a cosine wave

3.5 gillcup.graphics

Contents:

3.5.1 gillcup.graphics.mainwindow

3.5.2 gillcup.graphics.layer

3.5.3 gillcup.graphics.colorrect

[XXX]

3.5.4 gillcup.graphics.sprite

[XXX]

3.5.5 gillcup.graphics.baselayer

[XXX] The convenience animation methods need manual docs

3.5.6 gillcup.graphics.helpers

[XXX]

`gillcup.graphics.helpers.`**`extend_tuple`**`(`*args*, *default=0*`)`
Extend the given tuple to a triple, padding by the given value

`gillcup.graphics.helpers.`**`extend_tuple_copy`**(*args*)

Extend the given tuple to a triple, copying the last value

`gillcup.graphics.helpers.`**`nullContextManager`**(**args*, ***kwds*)

A context manager that does nothing

3.5.7 gillcup.graphics.text

[XXX]

CHAPTER

FOUR

INDICES AND TABLES

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

PYTHON MODULE INDEX