pySFML 2 - Cython Documentation

Release 0.2

Bastien Léonard

October 11, 2015

Contents

A new Python 2/3 binding for SFML 2, made with Cython. Most features of SFML are currently available, but this is still a work in progress. Feel free to report any issue you encounter.

You can find the source code, downloads and the issue tracker here: https://github.com/bastienleonard/pysfml2-cython.

There is also a thread on the official forums: http://en.sfml-dev.org/forums/index.php?topic=5311.0. I use it to make announcements and answer questions, but if you want to report an issue, please consider using the Github tracker. I sometimes forget bugs and suggestions that I read on the forums.

The documentation should now be complete, but if you need more detailed information, the SFML 2 documentation may be useful.

If you haved used SFML in the past, you will probably want to read *Learning pySFML from a C++ SFML background*.

Note: Make sure you read the *Caveats* page, so that you know what the most important current limitations are.

Contents:

CHAPTER 1

Introduction

1.1 What is this project about?

This project allows you use to use SFML 2 from Python. As SFML's author puts it, "SFML is a free multimedia C++ API that provides you low and high level access to graphics, input, audio, etc." It's the kind of library you use for writing multimedia applications such as games or video players.

1.2 What isn't this project about?

This binding currently doesn't aim to be used as an OpenGL wrapper, unlike the original SFML library. This is because there are already such wrappers available in Python, such as Pygame, PyOpenGL or pyglet.

1.3 Doesn't SFML already have a Python binding?

It does, but the binding needed to be rewritten, mainly because the current binding is directly written in C++ and is a maintenance nightmare. This new binding is written in Cython, hence the name.

Also, I find that the current binding lacks some features, such as:

- It doesn't follow Python's naming conventions.
- It lacks some fancy features such as properties, exceptions and iterators (for example, my binding allows you to iterate on events with a simple `for` loop).

You should also note that the current PySFML release on SFML's website is buggy (for example, `Image.SetSmooth()` doesn't work). You'd need to compile the latest version yourself to avoid these bugs.

1.4 Why SFML 2?

SFML 1 is now part of the past; it contains some important bugs and apparently won't be updated anymore.

SFML 2 is still a work in progress, but it's stable enough for many projects and it only breaks a few parts of SFML 1's API.

SFML 2 brings in important changes, such as new features, performance improvement and a more consistent API. In my opinion, if you aren't tied to SFML 1, you should stop using it and try SFML 2.

1.5 What does "Cython" mean? Can I use this module with Python 2/3?

I use it in the binding's name to help distinguish it with other bindings. The fact the it's written with Cython means that it's easier to maintain, and as fast as a C or C++ binding (although some parts *might* need optimizations).

Don't worry, the module works with the traditional Python interpreter (CPython), version 2 or 3. (For more information, see *Building the module*.) However, it doesn't work with other interpreters like PyPy.

CHAPTER 2

Caveats

Currently, the binding doesn't work correctly when built straight from the Git repo, see this forum post: http://en.sfml-dev.org/forums/index.php?topic=5311.msg52943#msg52943 If you want to build from the source, you're encouraged to use the latest source release. See *Building without Cython*.

Windows programs crash just before exiting. My guess is that it's related to the destruction of static objects; I'll try to fix it for the next minor release.

A current limitation is that `Texture` objects won't work as expected unless they are created after your `RenderWindow`. It isn't a big problem in practice, but it's something to keep in mind until the issue is fixed. This seems to be related to a bug in SFML: https://github.com/LaurentGomila/SFML/issues/160 It may also be dependent on the platform, but even if it works correctly on your system, you shouldn't rely on it for now.

CHAPTER 3

Frequently Asked Questions

How do I draw a line?

> The general answer is: use *RectangleShape*. If your line has a width of one pixel, you can also use *RenderTarget.draw()* with two vertices and the *LINES* primitive.

CHAPTER 4

Changelog

0.2 (07/20/2012):

- `Keyboard.BACK` has been renamed to *`Keyboard.BACK_SPACE`*, to fit with the C++ SFML change.
- Added support for file streaming: see *`SoundStream`*, *`SoundBuffer.load_from_stream()`*, *`Music.open_from_stream()`*, *`Font.load_from_stream()`*, *`Image.load_from_stream()`*, *`Texture.load_from_stream()`*, *`Shader.load_both_types_from_stream()`* and *`Shader.load_from_stream()`*.
- *`RectangleShape.size`* doesn't raise exceptions for no reason anymore.
- Removed `RenderTexture.create()`, the constructor should be used instead.
- *`RenderTexture.active`* now raises an exception when setting it causes an error.
- Added `copy()` and `__repr__()` methods in *`Vertex`*.
- Removed `View.get_transform()` and `View.get_inverse_transform()`; SFML's documentation says they are meant for internal use only.
- *`View.from_rect()`* and *`View.reset()`* now accept tuples.
- Setting *`Shape.texture`* to `None` now does the right thing at the C++ level (it sets the underlying texture pointer to `NULL`).
- The API reference should now be complete, and it has been reorganized to avoid huge pages. A FAQ page has been started.

0.1.3 (06/19/2012):

- Replaced `Sprite.text_rect` with two *`Sprite.get_texture_rect()`* and *`Sprite.set_texture_rect()`*.
- *`RenderStates`*' constructor now takes a *blend mode* as its first parameter.
- Added missing methods in *`ConvexShape`* (`get_point()`, `get_point_count()`, `set_point()`, `set_point_count()`). The `point_count` attribute has been removed.
- Added *`RenderWindow.height`*, *`RenderWindow.width`*, *`Texture.bind()`*, *`Texture.NORMALIZED`*, *`Texture.PIXELS`*, *`Color.TRANSPARENT`*, *`Image.flip_horizontally()`*, *`Image.flip_vertically()`* and *`RenderWindow.active`*.
- *`Glyph`*'s attributes are now modifiable.
- *`RenderWindow.wait_event()`* now raises *`PySFMLException`* when the underlying C++ method fails. (In the past, the error would be ignored.)
- *`Image.get_pixels()`* now returns None when the image is empty.

- *Image.get_pixel()* and *Image.set_pixel()* now raise IndexError if the pixel coordinates are out of range.
- *Image.save_to_file()* now raises *PySFMLException* when an error occurs.
- The constructors of *Keyboard*, *Mouse* and *Style* now raise NotImplementedError.
- Fixed a bug where SFML would fail to raise an exception. This typically happened when a tuple, a *FloatRect* or an *IntRect* was expected, but another type was passed.
- Added the tests in the source release.
- Completed the documentation of many graphics classes.

0.1.2:

- Added copy() methods in *Transform*, *IntRect*, *FloatRect*, *Time* and *Sprite*.
- *RenderTarget.draw()* now also accepts a tuple of vertices. Also fixed error handling when the objects contained in the list/tuple have the wrong type.
- Added == and != operators in *IntRect* and *FloatRect*.
- *Transform*'s constructor now creates an identity transform when called with no arguments.
- Transform now supports the *=* operator. (It already worked in the past, because Python will automatically use the * operator if *= isn't provided, but it's slower.)
- *SoundBuffer.save_to_file()* now raises an exception in case of failure. (In the past, it didn't report errors in any way.)
- Removed Chunk.sample_count and SoundBuffer.sample_count. Instead, use len(Chunk.samples)``and ``len(SoundBuffer.samples), respectively.
- *SoundBuffer.load_from_samples()* now uses strings/bytes (for Python 2/3, respectively) instead of list.
- Fixed bugs in *Font*, *Image* and *Shader* classmethods that load from strings/bytes objects.
- Added *Joystick.update()*.
- *Transformable* isn't abstract anymore, and can be inherited safely.
- Completed the events and audio documentation, added documentation for some graphics classes.
- Expanded the tutorial for C++ developers.

0.1.1:

- The seconds(), milliseconds() and microseconds() functions are removed. Use the *Time* constructor with keyword arguments instead, e.g. milliseconds(200) becomes Time(milliseconds=200).
- Made Sprite more straightforward to inherit, __cinit__() won't raise errors because it automatically gets passed the constructor arguments anymore.
- Fixed a bug in Time where some arithemtic operators would always raise an exception.
- Fixed a bug in RenderStates where internal attributes and properties got mismatched because they had the same name.
- Added a __repr__() method in *Time* (mostly to have more readable unit test errors, __str__() already existed in the past).
- Documentation: added a "caveats" page, and a new tutorial for people who are coming from a C++ SFML background.

- Added some unit tests.

0.1:

- The module is now called sfml. To keep using the sf prefix, import the module with `import sfml as sf`.
- Python 3 users don't need to use bytes instead of strings anymore. When a C++ method expects a byte string and the user passes a Unicode object, it is encoded to a byte string with *sfml.default_encoding* (UTF-8 by default, you can change it as needed).
- Added the *Listener* class.
- Added audio streaming (still lacking performance-wise).
- Added *Texture.copy_to_image()*.
- Improved examples.
- Fixed various bugs and memory leaks.

CHAPTER 5

Building the module

5.1 Binary releases

If you're on Windows, you can download the current binary release and ignore most of this section.

Official releases are at https://github.com/bastienleonard/pysfml2-cython/downloads. The installer contains the module itself, and the required DLLs (SFML and dependencies). The DLLs are dropped in Python's folder, e.g. `C:\Python27`. If you haven't already, make sure that this folder has been added to the `PATH` environment variable.

Christoph Gohlke also provides installers which are currently more up-to-date, with support for Python 2.6 as well as native 64 bits installers on his website: http://www.lfd.uci.edu/~gohlke/pythonlibs/#pysfml

You should be able to use pySFML 2 without installing anything else. Feedback is welcome.

On other platforms, there may still be easier ways to build the module. Someone has written AUR scripts for Arch Linux users:

- https://aur.archlinux.org/packages.php?ID=50841
- https://aur.archlinux.org/packages.php?ID=50842

5.2 Getting SFML 2

The first thing you should do is get SFML 2 and make sure it works. Please refer to the official tutorial: http://sfml-dev.org/tutorials/2.0/compile-with-cmake.php

Some platforms may make it easier to install it, for example Arch Linux users can get it from the AUR.

If you are on Windows, you will probably want to copy SFML's headers and libraries directories to the corresponding directories of your compiler/IDE, and SFML's DLLs to Windows' DLL directory.

5.3 Building on Windows

If you don't have a C++ compiler installed, I suggest using MinGW.

If you are using a recent version of MinGW, you may encounter this error when building the module:

```
error: unrecognized command line option '-mno-cygwin'
```

The problem is that the `-mno-cygwin` has been dropped in recent MinGW releases. A quick way to fix this is to remove the option from the distutils source. Find the `distutils/cygwinccompiler.py` in your Python installation (it should be something like `C:\Python27\Lib\distutils\cygwinccompiler.py`). Find the `MinGW32CCompiler` class and remove the `-mno-cygwin` options:

```
# class CygwinCCompiler
self.set_executables(compiler='gcc -mno-cygwin -O -Wall',
                     compiler_so='gcc -mno-cygwin -mdll -O -Wall',
                     compiler_cxx='g++ -mno-cygwin -O -Wall',
                     linker_exe='gcc -mno-cygwin',
                     linker_so='%s -mno-cygwin %s %s'
                               % (self.linker_dll, shared_option,
                                  entry_point))
```

If you are using Visual C++, please use the 2008 version. Python was built with this version, and it's apparently difficult to use 2010 because it links to another C or C++ runtime.

5.4 Common build options

You can build the module with the `setup.py` script (or `setup3k.py` for Python 3). This section discusses some common options that you may need or find useful.

`--inplace` means that the module will be dropped in the current directory. I find this more practical, as it makes it easier to test the module once built.

`--compiler=mingw32` obviously means that MinGW will be invoked instead of the default compiler. This is needed when you want to use GCC on Windows. This command will show you the list of compilers you can specify: `python setup.py build_ext --help-compiler`. Visual Studio is the default compiler and should work without using this option.

In the end, the command will look something like this:

```
python setup.py build_ext --inplace --compiler=mingw32
```

5.5 Building without Cython

If you download a source release at the download page, you don't need to install Cython, since the release already contains the files that Cython would generate.

Make sure that `USE_CYTHON` is set to `False` in setup.py (or setup3k.py, if you're building for Python 3). You can then build the module by typing this command:

```
python setup.py build_ext
```

5.6 Building with Cython installed

Warning: Currently, modules built straight from the repo probably won't work (this may depend on your Cython version). Consider using a source release, or follow the indications from this forum post if you still want to build from Git: http://en.sfml-dev.org/forums/index.php?topic=5311.msg52943#msg52943

Warning: Several Ubuntu users reported that they can't build the module because the Cython package is currently outdated. One solution is to install Cython manually, for example with `easy_install cython`.

If you downloaded the source straight from the Git repo or if you have modified the source, you'll need to install Cython to build a module including the changes. Also, make sure that `USE_CYTHON` is set to `True` in setup.py.

When you've done so, you can build the module by typing this command:

```
python setup.py build_ext
```

If you get an error related with `DL_IMPORT`, refer to the end of the *Building a Python 3 module* section.

5.7 Building a Python 3 module

It's possible to build a Python 3 module, but you may encounter a few minor problems.

First of all, on my machine, the Cython class used in `setup3k.py` to automate Cython invocation is only installed for Python 2. It's probably possible to install it for Python 3, but it's not complicated to invoke Cython manually:

```
cython --cplus sfml.pyx
```

The next step is to invoke the `setup3k.py` script to build the module. Since we called Cython already, make sure that `USE_CYTHON` is set to `False` in `setup3k.py`, then invoke this command:

```
python3 setup3k.py build_ext
```

(Note that you may have to type `python` instead of `python3`; typically, GNU/Linux systems provide this as a way to call a specific version of the interpreter, but I'm not sure that's the case for all of them as well as Windows.)

(Also note that on GNU/Linux, the generated file won't be called `sfml.so` but something like `sfml.cpython-32mu.so`. Apparently, on Windows it's still `sfml.pyd`.)

The second problem used to be that you had to use bytes instead of Unicode e.g. when passing a filename or window title to SFML. This is now gone, except possibly in methods that I forgot to fix; make sure to report the issue if you encounter such a case. When you pass a Unicode object to these methods, they now encode it in UTF-8 before passing them to SFML. You can change the encoding by setting the *`default_encoding`* variable at any time.

Finally, compilation may fail because the `src/sfml.h` file generated by Cython uses the deprecated `DL_IMPORT()` macro. At the root of the project, there is a `patch.py` script that will remove the offending macros for you. The trick is that `src/sfml.h` will not exist at first; the setup script will create it, then try to compile it and fail. That's when you need to use `patch.py`, and build the module again.

CHAPTER 6

Tutorials

6.1 pySFML basics

Warning: The module has recently been renamed from `sf` to `sfml`, to be more clear and avoid clashes. However, it's easy to still use `sf` as the namespace in your code; just write `import sfml as sf`. This is the approach that we follow in this tutorial and in the examples. The reference uses `sfml` though, since it's the "official" namespace.

Welcome to pySMFL's official tuturial! You are going to learn how to display an image and move it based on user input. But first, here is the full listing:

```
import sfml as sf


def main():
    window = sf.RenderWindow(sf.VideoMode(640, 480),
                             'Drawing an image with SFML')
    window.framerate_limit = 60
    running = True
    texture = sf.Texture.load_from_file('python-logo.png')
    sprite = sf.Sprite(texture)

    while running:
        for event in window.iter_events():
            if event.type == sf.Event.CLOSED:
                running = False

        if sf.Keyboard.is_key_pressed(sf.Keyboard.RIGHT):
            sprite.move(5, 0)
        elif sf.Keyboard.is_key_pressed(sf.Keyboard.LEFT):
            sprite.move(-5, 0)

        window.clear(sf.Color.WHITE)
        window.draw(sprite)
        window.display()

    window.close()


if __name__ == '__main__':
    main()
```

You can get the `python-logo.png` file here, or use any other image file supported: bmp, dds, jpg, png, tga, or psd.

Note: If you're new to Python, you may find the last two lines confusing. They're not necessary to make the script run: if you remove them as well as the `def main():` line and adjust the indentation accordingly, the program will still run fine. But it's a good practice to use this pattern in your scripts.

The `main()` function that we defined isn't a "standard" function that gets automatically called, like in C or C++. So we call the function ourself if `__name__ == __main__`, i.e. if our file has been launched by the user, rather than imported by some code. You can find more information here: http://stackoverflow.com/questions/419163/what-does-if-name-main-do

---

6.1.1 Creating a window

Windows in pySFML are created with the *RenderWindow* class. This class provides some useful constructors to create directly our window. The interesting one here is the following:

```
window = sf.RenderWindow(sf.VideoMode(640, 480), 'SFML Window')
```

Here we create a new variable named `window` that will represent our new window. Let's explain the parameters:

- The first parameter is a *VideoMode*, which represents the chosen video mode for the window. Here, the size is 640x480 pixels, with a depth of 32 bits per pixel. Note that the specified size will be the internal area of the window, excluding the titlebar and the borders.
- The second parameter is the window title.

If you want to create your window later, or recreate it with different parameters, you can use its *RenderWindow.create()* method:

```
window.create(sf.VideoMode(640, 480), 'SFML Window');
```

The constructor and the *RenderWindow.create()* method also accept two optional additionnal parameters: the first one to have more control over the window's style, and the second one to set more advanced graphics options; we'll come back to this one in another tutorial, beginners usually don't need to care about it. The style parameter can be a combination of the `sf.Style` flags, which are `NONE`, `TITLEBAR`, `RESIZE`, `CLOSE` and `FULLSCREEN`. The default style is `Style.RESIZE | Style.CLOSE`.

```
# This creates a fullscreen window
window.create(sf.VideoMode(800, 600), 'SFML Window', sf.Style.FULLSCREEN);
```

6.1.2 Video modes

When you create a *VideoMode*, you can choose the bits per pixel with a third argument. If you don't, it is set to 32, which is what we do in our examples, since it's probably the most common value.

In the previous examples, any video mode size works because we run in windowed mode. But if we want to run in fullscreen mode, we have to choose one of the allowed modes. The *VideoMode.get_fullscreen_modes()* class method returns a list of all the valid fullscreen modes. They are sorted from best to worst, so `sf.VideoMode.get_fullscreen_modes()[0]` will always be the highest-quality mode available:

```
window = sf.RenderWindow(sf.VideoMode.get_fullscreen_modes[0], 'SFML Window', sf.Style.FULLSCREEN)
```

If you are getting the video mode from the user, you should check its validity before applying it. This is done with *VideoMode.is_valid()*:

```
mode = get_mode_from_somewhere()

if not mode.is_valid():
    # Error...
```

The current desktop mode can be obtained with the *VideoMode.get_desktop_mode()* class method.

6.1.3 Main loop

Let's write a skeleton of our game loop:

```
# Setup code
window = sf.RenderWindow(sf.VideoMode(640, 480), 'SFML window')
# ...

while True:
    # Handle events
    # ...

    window.clear(sf.Color.WHITE)

    # Draw our stuff
    # ...

    window.display()
```

`RenderWindow.clear()` fills the window with the specified color. (If you don't pass any color, black will be used.) You can create "custom" color objects with the *Color* constructor. For example, if you wanted to a pink background you could write `window.clear(sf.Color(255, 192, 203))`. The call to *RenderWindow.display()* simply updates the content of the window.

This code doesn't look right currently, because we have a loop that doesn't really do anything: it just draws the same background over and over. Don't worry, it will make more sense once we will actually draw stuff.

If you run this program and look at your process manager, you'll see that it is using 100% of one of your processor's time. This isn't surprising, given the busy loop we wrote. A simple fix is to set the *RenderWindow.framerate_limit* attribute:

```
window.framerate_limit = 60
```

This line tells SFML to ensure that the window isn't updated more than 60 times per second. It should to go in the setup code.

6.1.4 Event handling basics

The most common way to handle events in pySFML is to use *RenderWindow.iter_events()*. You can still use *RenderWindow.poll_event()* like in C++ SFML, but it will just make the code look a bit clumsy.

If you're used to C++ SFML, you will need to change your habit: pySFML events only have the attributes that make sense for this particular event; there's no equivalent to the C++ union.

You need to test the `type` attribute to know kind of event you're looking at. Here are the event types:

- `sf.Event.CLOSED`
- `sf.Event.RESIZED`
- `sf.Event.LOST_FOCUS`

- sf.Event.GAINED_FOCUS
- sf.Event.TEXT_ENTERED
- sf.Event.KEY_PRESSED
- sf.Event.KEY_RELEASED
- sf.Event.MOUSE_WHEEL_MOVED
- sf.Event.MOUSE_BUTTON_PRESSED
- sf.Event.MOUSE_BUTTON_RELEASED
- sf.Event.MOUSE_MOVED
- sf.Event.MOUSE_ENTERED
- sf.Event.MOUSE_LEFT
- sf.Event.JOYSTICK_BUTTON_PRESSED
- sf.Event.JOYSTICK_BUTTON_RELEASED
- sf.Event.JOYSTICK_MOVED
- sf.Event.JOYSTICK_CONNECTED
- sf.Event.JOYSTICK_DISCONNECTED

In our case, we just use the "closed" event to stop the program:

```
for event in window.iter_events():
    if event.type == sf.Event.CLOSED:
        running = False
```

Most event objects contain special attributes containing useful values, but CLOSED doesn't, it just tells you that the user want to close your application. KEY_PRESSED is another common event type. Events of this type contain several attributes, but the most important one is code. It's an integer that maps to one of the constants in the *Keyboard* class.

For example, if we wanted to close the window when the user presses the Escape key, our event loop could look like this:

```
while running:
    for event in window.iter_events():
        if event.type == sf.Event.CLOSED:
            running = False
        elif event.type == sf.Event.KEY_PRESSED:
            if event.code == sf.Keyboard.ESCAPE:
                running = False
```

See *Event types reference* for the list of all events and the attributes they contain.

Note: In fullscreen mode, you can't rely on the window manager's controls to send the CLOSED event, so it's a good idea to set a shortcut like we just did to make sure the user is able to close the application.

6.1.5 Drawing the image

You will need to use at least two classes for displaying the image: *Texture* and *Sprite*. It's important to understand the difference between these two:

- Textures contain the actual image that you want to display. They are heavy objects, and you shouldn't have the same image/texture loaded more than once in memory. Textures objects can't be displayed directly; for example there's no way to set the (x, y) position of a texture. You need to use sprites for this purpose.
- Sprites are lightweight objects associated with a texture, either with the constructor or the *Sprite.texture* attribute. They have many visual properties that you can change, such as the (x, y) position, the zoom or the rotation.

In practice, you might have several creatures displayed on screen, all from the same image. The image would be loaded only once into memory, and several sprite objects would be created. They would all have the same texture property, but their position would be set to the creature's position on screen. They could also have a different rotation or other effects, based on the creature's state.

There are two main steps to displaying our image. First, we need to load the image in the setup code and create the sprite:

```
texture = sf.Texture.load_from_file('python-logo.png')
sprite = sf.Sprite(texture)
```

Now, we can display the sprite in the game loop:

```
window.clear(sf.Color.WHITE)
window.draw(sprite)
window.display()
```

6.1.6 Real-time input handling

What if we want to do something as long as the user is pressing a certain key? For example, we want to move our logo as long as the user is pressing the right arrow key, or the left key. In that case, it's not enough to know that the user just pressed the key. We want to know whether he is still holding it or not.

To achieve that, you would need to set a boolean to `True` as soon as the user is pressing the key. When you get the "release" event for that key, you set it back to `False`. And you read the value of that boolean to know whether the right key is pressed or not.

As it turns out, SFML has this kind of feature built in. You can call *Keyboard.is_key_pressed()* with the code the key as an argument; it will return `True` if this key is currently pressed. The key codes are class attributes in *Keyboard*: for example, *Keyboard.LEFT* and *Keyboard.RIGHT* map to the left and right arrow keys. Your event loop would then look something like this:

```
while running:
    for event in window.iter_events():
        if event.type == sf.Event.CLOSED:
            running = False

    if sf.Keyboard.is_key_pressed(sf.Keyboard.RIGHT):
        sprite.move(5, 0)
    elif sf.Keyboard.is_key_pressed(sf.Keyboard.LEFT):
        sprite.move(-5, 0)
```

The *Mouse* class provides a similar class method, *Mouse.is_button_pressed()*, for when you need to know whether a mouse button is pressed.

6.1.7 Images and textures

Another class may be useful for displaying images: *Image*. The difference between a texture and an image is that a texture gets loaded into video memory and can be efficiently displayed. If you want to display an image, you need to

create a texture and call *Texture.load_from_image()*, and then display the texture. On the other hand, you can access and modify the pixels of an image as needed.

The bottom line is: use textures by default, and use images only if it's needed.

6.2 Learning pySFML from a C++ SFML background

6.2.1 Naming convention

This module follows the style guide for Python code as much as possible. To give you an idea, here is a list of attribute naming examples:

- Classes: `RenderWindow`, `Texture`.
- Methods and attributes: `default_view`, `load_from_file()`.
- Constants: `CLOSED`, `KEY_PRESSED`, `BLEND_ALPHA`.

Namespaces normally follow the same nesting as in C++, e.g. `sf::Event::Closed` becomes *sfml.Event.CLOSED*. Events are an exception, see *Events*.

6.2.2 Object initialization with class methods

C++ SFML has a general pattern for creating objects when their initialization may fail:

- Allocate an "empty" object.
- Call a method that will initialize the object, e.g. `loadFromFile()`.
- If this method returned `false`, handle the error.

In pySFML, you typically just have to call a class method, e.g. *Texture.load_from_file()*. If you want to handle possible errors at this point, you write an `except` block (see *Error handling*). Otherwise, the exception will propagate to the next handler.

In some cases, class methods are the only way to initialize an object. In that case, the constructor will raise `NotImplementedError` if you call it. In other cases, the constructors peform some kind of default initialization, while class methods do more specific work.

6.2.3 Properties

Generally speaking, `set*()/get*()` methods are replaced by properties. For example, `RenderWindow.getSize()/RenderWindow.setSize()` becomes a *RenderWindow.size* property which behaves like a normal attribute. I tend to create properties when the user can safely ignore that he's not dealing with an actual attribute, i.e. when the property doesn't do anything non-obvious and is fast to execute.

In some cases, it's not that straightforward. Some properties only have a getter or a setter, even though they should have both (for example, *RenderWindow.key_repeat_enabled*). The reason is that C++ SFML doesn't provide the missing set/get method. This has been pointed out to SFML's author, who is going to fix it someday. I could fix it myself, but it would require to add quite a lot of boilerplate that I will need to remove when SFML gets the missing methods. The reason why these methods are missing in the first place is that's they're not very useful, so I consider that to be a decent trade-off.

I tend to use a method instead of an attribute when I feel like a `get*()` method involves some kind of computation. For example, `View.get_inverse_transform()` is a method instead of a property because I somehow feel like

it involves something heavier than simply looking up an attribute. Admittedly, this is subjective, and it's difficult to be consistent with this kind of choice as well.

6.2.4 Events

pySFML objects only feature the attributes that they actually need. For example, `event.key.code` in C++ becomes `event.code`. Accessing an attribute that doesn't make sense for this event will raise an exception, because the object event doesn't have it at all. As you can see in the *Event types reference*, there is some overlap, so theoretically you could confuse a `MOUSE_WHEEL_MOVED` event for a `MOUSE_MOVED` event, access the `x` or `y` attribute, without raising any exception.

Instead of using *`RenderWindow.poll_event()`*, events are usually retrieved in `for` loop with `RenderWindow.iter_event()`:

```
for event in window.iter_events():
    if event.type == sfml.Event.CLOSED:
        ...
```

6.2.5 Error handling

Unlike C++ SFML, there are no boolean return values to indicate success or failure. Anytime SFML returns `False`, typically, when a file can't be opened, pySFML raises *`PySFMLException`*. Please read the description of this exception for more information.

I'd like to add more specific exceptions, but since SFML only returns `True` or `False`, I can't tell if the source of the failure is a non existant file, an invalid file content, an internal library failure, or anything else. SFML's author wants to improve error handling in a future release. At this point, more specific exceptions will probably be possible to implement.

6.2.6 Creating your own drawables

Unlike in C++ SFML, you don't have to inherit a `Drawable` class. This is covered in *Creating your own drawables*.

6.2.7 Time

Time values are created with *`Time`*'s constructor using keyword arguments, instead of calling a global function. For example, `sf::milliseconds(200)` becomes `sfml.Time(milliseconds=200)`.

6.2.8 "Missing" features

`Vector2f` has been ported, but tuples are used instead of `Vector2i` and `Vector3f`. These classes are used so sparsely that it doesn't seem worth porting them. Note that you can pass tuples instead of *`Vector2f`* objects.

The network and threading parts of SFML aren't ported in this module, since similar features are already provided by the standard library. For UDP and TCP connections, you should look into the `socket` module. `threading` is the general, high-level module for threading stuff. For URL retrieval, `urllib` and `urllib2` are provided.

You may also want to check out non standard libraries such as Twisted or requests.

Most streaming features are also currently missing.

CHAPTER 7

API reference

This reference is splitted in sections for readability only. Every class is available in the same `sfml` namespace.

7.1 Exceptions

exception `sfml.`**`PySFMLException`**

Raised when any important error is encountered. Typically, file loading methods such as *`Texture.load_from_file()`* return the new object if everything went well, and raise this exception otherwise.

A simple example of error handling:

```
try:
    texture = sf.Texture.load_from_file('texture.png')
except sf.PySFMLException as e:
    pass # Handle error: pring message, log it, ...
```

In C++:

```
sf::Texture texture;

if (!texture.LoadFromFile("texture.png"))
{
    // Handle error
}
```

Please understand that you don't *have* to handle exceptions every time you call a method that might throw one; you can handle them at a higher level or even not handle them at all, if the default behavior of stopping the program and printing a traceback is OK. This is an advantage compared to C++ SFML, where ignoring return statuses means that your program will try to keep running normally if an important error is raised.

`message`

A string describing the error. This is the same message that C++ SFML would write in the console.

7.2 System

`sfml.`**`default_encoding`**

Currently, this encoding is used when the user passes a Unicode object to method that will call a SFML method which only supports `std::string` argument. The user-supplised Unicode object will be encoded with this encoding and the resulting bytes will be passed to SFML. This is mostly for Python

3 users, so they don't have to use byte strings all the time. Here is the list of valid encodings: http://docs.python.org/py3k/library/codecs.html#standard-encodings

class `sfml.`**`Clock`**

Utility class that measures the elapsed time.

Its provides the most precise time that the underlying OS can achieve (generally microseconds or nanoseconds). It also ensures monotonicity, which means that the returned time can never go backward, even if the system time is changed.

Usage example:

```
clock = sfml.Clock()
...
time1 = clock.elapsed_time
...
time2 = clock.restart()
```

The *Time* object returned by the clock can then be converted to a number of seconds, milliseconds or even microseconds.

`elapsed_time`

A *Time* object containing the time elapsed since the last call to *restart()*, or the construction of the instance if *restart()* has not been called yet.

`restart``()`

Restart the clock, and return a *Time* object containing the elapsed time since the clock started.

class `sfml.`**`InputStream`**

This abstract class allows users to define their own file-like input sources from which SFML can load resources.

SFML resource classes like *Texture* and *SoundBuffer* provide loadFromFile and loadFromMemory class methods which read data from conventional sources. However, if you have data coming from a different source (over a network, embedded, encrypted, compressed, etc) you can derive your own class from *InputStream* and load SFML resources with their loadFromStream function.

> **Warning:** Exceptions that occur in the implemented methods won't be propagated, but printed on `sys.stderr` (the console, by default). This is because of concerns regarding multithreading and exception propagation. Please keep your methods as simple as possible, and if they don't work, make sure you read the console.

Usage example:

```
class ExampleStream(sfml.InputStream):
    def __init__(self, filename):
        sfml.InputStream.__init__(self)
        self.filename = filename
        self.file = open(filename, 'rb')
        self.file.seek(0, 2)
        self.size = self.file.tell()
        self.file.seek(0)

    def get_size(self):
        print('{0}: get_size()'.format(self.filename))
        return self.size

    def read(self, size):
        print('{0}: read({1})'.format(self.filename, size))

        return self.file.read(size)
```

```
        def seek(self, position):
            print('{0}: seek({1})'.format(self.filename, position))
            self.file.seek(position)

            return self.tell()

        def tell(self):
            print('{0}: tell()'.format(self.filename))

            return self.file.tell()

        def close(self):
            self.file.close()

    # Now you can load textures...
    texture_stream = ExampleStream(some_path)
    texture = sfml.Texture.load_from_stream(texture_stream)

    # Music...
    music_stream = ExampleStream('music.ogg')
    music = sfml.Music.open_from_stream(music_stream)
    music.play()

    # Etc.
```

get_size()
Return the number of bytes available in the stream, or -1 on error.

read(*int size*)

***size* is the desired number of bytes to read. The method should** return a string in Python 2, or a bytes object in Python 3. If needed, its length can be smaller than *size*.

seek(*int position*)
Change the current position to *position*, from the beginning of the streal. This method has to return the actual position sought to, or -1 on error.

tell()
Return the current reading position on the stream, or -1 on error.

class sfml.**Time**(*seconds=-1.0*, *milliseconds=-1*, *microseconds=-1*)
Instead of forcing the user to use a specific time units, SFML uses this class to encapsulate time values. The user can get an actual time value by using the following methods: *as_seconds()*, *as_milliseconds()* and *as_microseconds()*. You can also create your own time objects by calling the constructor with one keyword argument.

Using one keyword argument is equivalent to calling the corresponding function. For example, `sfml.seconds(10) == sfml.Time(seconds=10)`.

This class provides the following special methods:

- Comparison operators: ==, !=, <, >, <=, >=.
- Arithmetic operators: +, −, *, /, unary −.
- `str()` returns a representation of the number of seconds.

ZERO
Predefind "zero" time value (class attribute).

as_seconds()
Return a `float` containing the number of seconds for this time object.

as_milliseconds()
Return an `int` containing the number of milliseconds for this time object.

as_microseconds()
Return an `int` containing the number of microseconds for this time object.

copy()
Return a new Time object with the same value as self.

7.3 Graphics

7.3.1 Misc

Blend modes

sfml.**BLEND_ADD**
Pixel = Source + Dest.

sfml.**BLEND_ALPHA**
Pixel = Source * Source.a + Dest * (1 - Source.a).

sfml.**BLEND_MULTIPLY**
Pixel = Source * Dest.

sfml.**BLEND_NONE**
Pixel = Source.

Primitive types

sfml.**POINTS**
List of individual points.

sfml.**LINES**
List of individual lines.

sfml.**LINES_STRIP**
List of connected lines, a point uses the previous point to form a line.

sfml.**TRIANGLES**
List of individual triangles.

sfml.**TRIANGLES_FAN**
List of connected triangles, a point uses the common center and the previous point to form a triangle.

sfml.**TRIANGLES_STIP**
List of connected triangles, a point uses the two previous points to form a triangle.

sfml.**QUADS**
List of individual quads.

Basic classes

class sfml.**Color**(*int r*, *int g*, *int b*[, *int a=255*])

Represents a color of 4 components:

- red,
- green,
- blue,
- alpha (opacity).

Each component is a public member, an unsigned integer in the range [0, 255]. Thus, colors can be constructed and manipulated very easily:

```
color = sfml.Color(255, 0, 0)  # red; you can also use Color.RED
color.r = 0  # make it black
color.b = 128  # make it dark blue
```

The fourth component of colors, named "alpha", represents the opacity of the color. A color with an alpha value of 255 will be fully opaque, while an alpha value of 0 will make a color fully transparent, whatever the value of the other components is.

This class provides the following special methods:

- Comparison operators: == and !=.
- Arithmetic operators: + and *.

The following colors are available as static attibutes, e.g. you can use *Color.WHITE* to obtain a reference to the white color:

BLACK

BLUE

CYAN

GREEN

MAGENTA

RED

TRANSPARENT
Transparent black color, i.e. this is equal to Color(0, 0, 0, 0).

WHITE

YELLOW

r
Red component.

g
Green component.

b
Blue component.

a
Alpha (opacity) component.

copy()
Return a new Color with the same value as self.

class sfml.**Vector2f**(*float x=0.0; float y=0.0*)

You don't have to use this class; everywhere you can pass a *Vector2f*, you should be able to pass a tuple as well. However, it can be more practical to use it, as it overrides arithmetic and comparison operators, is mutable and requires that you use the *x* and *y* members instead of indexing.

This class provides the following special methods:

- Comparison operators: == and !=.

x

x coordinate for this vector.

y

y coordinate for this vector.

copy()

Return a new *Vector2f* with x and y set to the value of self.

class sfml.**IntRect**(*int left=0*, *int top=0*, *int width=0*, *int height=0*)

A rectangle is defined by its top-left corner and its size.

To keep things simple, *IntRect* doesn't define functions to emulate the properties that are not directly members (such as right, bottom, center, etc.), instead it only provides intersection functions.

IntRect uses the usual rules for its boundaries:

- The left and top edges are included in the rectangle's area.
- The right (left + width) and bottom (top + height) edges are excluded from the rectangle's area.

This means that sfml.IntRect(0, 0, 1, 1) and sfml.IntRect(1, 1, 1, 1) don't intersect.

Usage example:

```
# Define a rectangle, located at (0, 0) with a size of 20x5
r1 = sfml.IntRect(0, 0, 20, 5)

# Define another rectangle, located at (4, 2) with a size of 18x10
r2 = sfml.IntRect(4, 2, 18, 10)

# Test intersections with the point (3, 1)
b1 = r1.contains(3, 1) # True
b2 = r2.contains(3, 1) # False

# Test the intersection between r1 and r2
result = sfml.IntRect()
b3 = r1.intersects(r2, result) # True
# result == (4, 2, 16, 3)
```

Note: You don't have to use this class; everywhere you can pass a *IntRect*, you should be able to pass a tuple as well. However, it can be more practical to use it, as it provides useful methods and is mutable.

This class provides the following special methods:

- Comparison operators: == and !=.

left

Left coordinate of the rectangle.

top

Top coordinate of the rectangle.

width
Width of the rectangle.

height
Height of the rectangle.

contains(*int x*, *int y*)
Return whether or not the rectangle contains the point *(x, y)*.

copy()
Return a new IntRect object with the same value as self.

intersects(*IntRect rect*[, *IntRect intersection*])
Return whether or not the two rectangles intersect. If *intersection* is provided, it will be set to the intersection area.

class sfml.**FloatRect**(*float left=0*, *float top=0*, *float width=0*, *float height=0*)
A rectangle is defined by its top-left corner and its size.

To keep things simple, *FloatRect* doesn't define functions to emulate the properties that are not directly members (such as right, bottom, center, etc.), instead it only provides intersection functions.

FloatRect uses the usual rules for its boundaries:

- The left and top edges are included in the rectangle's area.
- The right (left + width) and bottom (top + height) edges are excluded from the rectangle's area.

This means that `sfml.FloatRect(0, 0, 1, 1)` and `sfml.FloatRect(1, 1, 1, 1)` don't intersect.

See *IntRect* for an example.

Note: You don't have to use this class; everywhere you can pass a *FloatRect*, you should be able to pass a tuple as well. However, it can be more practical to use it, as it provides useful methods and is mutable.

This class provides the following special methods:

- Comparison operators: == and !=.

left
The left coordinate of the rectangle.

top
The top coordinate of the rectangle.

width
The width of the rectangle.

height
The height of the rectangle.

contains(*int x*, *int y*)
Return whether or not the rectangle contains the point *(x, y)*.

copy()
Return a new FloatRect object with the same value as self.

intersects(*FloatRect rect*[, *FloatRect intersection*])
Return whether or not the two rectangles intersect. If *intersection* is provided, it will be set to the intersection area.

7.3.2 Windowing

class `sfml.`**`RenderWindow`**(*[VideoMode mode, title[, style[, ContextSettings settings]]]*)

This class inherits *`RenderTarget`*.

This class represents an OS window that can be painted using the other graphics-related classes, such as *`Sprite`* and *`Text`*.

The constructor creates the window with the size and pixel depth defined in *mode*. If specified, *style* must be a value from the *`Style`* class. *settings* is an optional *`ContextSettings`* specifying advanced OpenGL context settings such as antialiasing, depth-buffer bits, etc. You shouldn't need to use it for a regular usage.

`active`
Write-only. If true, the window is activated as the current target for OpenGL rendering. A window is active only on the current thread, if you want to make it active on another thread you have to deactivate it on the previous thread first if it was active. Only one window can be active on a thread at a time, thus the window previously active (if any) automatically gets deactivated. If an error occurs, *`PySFMLException`* is raised.

`framerate_limit`
Write-only. If set, the window will use a small delay after each call to *`display()`* to ensure that the current frame lasted long enough to match the framerate limit. SFML will try to match the given limit as much as it can, but since the precision depends on the underlying OS, the results may be a little unprecise as well (for example, you can get 65 FPS when requesting 60).

`height`
The height of the rendering region of the window. The height doesn't include the titlebar and borders of the window. Unlike *`RenderTarget.height`*, this property can be modified.

`joystick_threshold`
Write-only. The joystick threshold is the value below which no *`Event.JOYSTICK_MOVED`* event will be generated. Default value: 0.1.

`key_repeat_enabled`
Write-only. If key repeat is enabled, you will receive repeated *`Event.KEY_PRESSED`* events while keeping a key pressed. If it is disabled, you will only get a single event when the key is pressed. Default value: `True`.

`mouse_cursor_visible`
Write-only. Whether or not the mouse cursor is shown. Default value: `True`.

`open`
Read-only. Whether or not the window exists. Note that a hidden window (`visible = False`) is open (so this attribute would be `True`).

`position`
The position of the window on screen. This attribute only works for top-level windows (i.e. it will be ignored for windows created from the *`system_handle`* of a child window/control).

`settings`
Read-only. The settings of the OpenGL context of the window. Note that these settings may be different from what was passed when creating the window, if one or more settings were not supported. In this case, SFML chooses the closest match.

`size`
The size of the rendering region of the window. The size doesn't include the titlebar and borders of the window. Unlike *`RenderTarget.size`*, this property can be modified.

`system_handle`
Return the system handle as a long (or int on Python 3). Windows and Mac users will probably need to

convert this to another type suitable for their system's API. You shouldn't need to use this, unless you have very specific stuff to implement that pySFML doesn't support, or implement a temporary workaround until a bug is fixed. If you need to use it, please contact me and show me your use case to see if I can make the API more user-friendly.

title
Write-only. The title of the window.

vertical_sync_enabled
Write-only. Whether or not the vertical synchronization is enabled. Activating vertical synchronization will limit the number of frames displayed to the refresh rate of the monitor. This can avoid some visual artifacts, and limit the framerate to a good value (but not constant across different computers). Default value: `False`.

visible
Write-only. Whether or not the window is shown. Default value: `True`.

width
The width of the rendering region of the window. The width doesn't include the titlebar and borders of the window. Unlike *RenderTarget.width*, this property can be modified.

classmethod from_window_handle(*long window_handle*[, *ContextSettings settings*])
Construct the window from an existing control. Use this class method if you want to create an SFML rendering area into an already existing control. The fourth parameter is an optional structure specifying advanced OpenGL context settings such as antialiasing, depth-buffer bits, etc. You shouldn't care about these parameters for regular usage.

Equivalent to this C++ constructor:

```
RenderWindow(WindowHandle, ContextSettings=ContextSettings())
```

close()
Close the window and destroy all the attached resources. After calling this function, the instance remains valid and you can call *create()* to recreate the window. All other methods such as *poll_event()* or *display()* will still work (i.e. you don't have to test *open* every time), and will have no effect on closed windows.

create(*VideoMode mode*, *title*[, *int style*[, *ContextSettings settings*]])
Create (or recreate) the window. If the window was already created, it closes it first. If *style* contains *Style.FULLSCREEN*, then *mode* must be a valid video mode.

display()
Display on screen what has been rendered to the window so far. This function is typically called after all the OpenGL rendering has been done for the current frame, in order to show it on screen.

iter_events()
Return an iterator which yields the current pending events. Example:

```
for event in window.iter_events():
    if event.type == sfml.Event.CLOSED:
        pass # ...
```

The traditional *poll_event()* method can be used to achieve the same effect, but using this iterator makes your life easier and is the recommended way to handle events.

poll_event()
Pop the event on top of events stack, if any, and return it. This method is not blocking: if there's no pending event then it will return `None` and leave the event unmodified. Note that more than one event may be present in the events stack, thus you should always call this function in a loop to make sure that you process every pending event.

```
event = sfml.Event()

while window.poll_event(event):
   pass # process event...
```

> **Warning:** In most cases, you should use *iter_events()* instead, as it takes care of creating the event objects for you.

set_icon(*int width*, *int height*, *str pixels*)
Change the window's icon. *pixels* must be a string in Python 2, or a bytes object in Python 3. It should contain width x height pixels in 32-bits RGBA format. The OS default icon is used by default.

wait_event()
Wait for an event and return it. This method is blocking: if there's no pending event, it will wait until an event is received. After this function returns (and no error occured), the event object is always valid and filled properly. This method is typically used when you have a thread that is dedicated to events handling: you want to make this thread sleep as long as no new event is received. If an error occurs, *PySFMLException* is raised.

```
event = sfml.Event()

if window.wait_event(event):
   pass # process event...
```

class sfml.**Style**
This window contains the available window styles, as class attributes. See *RenderWindow*.

Calling the constructor will raise NotImplementedError.

CLOSE
Titlebar + close button.

DEFAULT
Default window style.

FULLSCREEN
Fullscreen mode (this flag and all others are mutually exclusive).

NONE
No border/title bar (this flag and all others are mutually exclusive).

RESIZE
Titlebar + resizable border + maximize button.

TITLEBAR
Title bar + fixed border.

class sfml.**ContextSettings**(*int depth=24*, *int stencil=8*, *int antialiasing=0*, *int major=2*, *int minor=0*)
Class defining the settings of the OpenGL context attached to a window. *ContextSettings* allows to define several advanced settings of the OpenGL context attached to a window.

All these settings have no impact on the regular SFML rendering (graphics module), except the anti-aliasing level, so you may need to use this structure only if you're using SFML as a windowing system for custom OpenGL rendering.

Please note that these values are only a hint. No failure will be reported if one or more of these values are not supported by the system; instead, SFML will try to find the closest valid match. You can then retrieve the settings that the window actually used to create its context, with *RenderWindow.settings*.

antialiasing_level
Number of multisampling levels for antialiasing.

depth_bits
Bits of the depth buffer.

major_version
Major number of the context version to create. Only versions greater or equal to 3.0 are relevant; versions less than 3.0 are all handled the same way (i.e. you can use any version < 3.0 if you don't want an OpenGL 3 context).

minor_version
Minor number of the context version to create. Only versions greater or equal to 3.0 are relevant; versions less than 3.0 are all handled the same way (i.e. you can use any version < 3.0 if you don't want an OpenGL 3 context).

stencil_bits
Bits of the stencil buffer.

class `sfml.`**VideoMode**(*[width, height, bits_per_pixel=32]*)
A video mode is defined by a width and a height (in pixels) and a depth (in bits per pixel). Video modes are used to setup windows (*RenderWindow*) at creation time.

The main usage of video modes is for fullscreen mode: you have to use one of the valid video modes allowed by the OS (which are defined by what the monitor and the graphics card support), otherwise your window creation will just fail.

VideoMode provides a static method for retrieving the list of all the video modes supported by the system: *get_fullscreen_modes*.

A custom video mode can also be checked directly for fullscreen compatibility with its *is_valid()* method.

Additionnally, VideoMode provides a static method to get the mode currently used by the desktop: *get_desktop_mode()*. This allows to build windows with the same size or pixel depth as the current resolution.

Usage example:

```
# Display the list of all the video modes available for fullscreen
modes = sfml.VideoMode.get_fullscreen_modes()

for mode in modes:
    print(mode)

# Create a window with the same pixel depth as the desktop
desktop_mode = sfml.VideoMode.get_desktop_mode()
window.create(sfml.VideoMode(1024, 768, desktop_mode.bits_per_pixel),
              'SFML window')
```

This class overrides the following special methods:

- Comparison operators (==, !=, <, >, <= and >=).
- `str(mode)` returns a description of the mode in a `widthxheightxbpp` format.
- `repr(mode)` returns a string in a `VideoMode(width, height, bpp)` format.

width
Video mode width, in pixels.

height
Video mode height, in pixels.

bits_per_pixel
Video mode depth, in bits per pixel.

classmethod get_desktop_mode()
Return the current desktop mode.

classmethod get_fullscreen_modes()
Return a list of all the video modes supported in fullscreen mode. It is sorted from best to worst, so that the first element will always give the best mode (higher width, height and bits-per-pixel).

is_valid()
Return a boolean telling whether the mode is valid or not. This is only relevant in fullscreen mode; in other cases all modes are valid.

class sfml.**View**
The constructor creates a default view of (0, 0, 1000, 1000).

2D camera that defines what region is shown on screen. This is a very powerful concept: you can scroll, rotate or zoom the entire scene without altering the way that your drawable objects are drawn.

A view is composed of a source rectangle, which defines what part of the 2D scene is shown, and a target viewport, which defines where the contents of the source rectangle will be displayed on the render target (window or texture).

The viewport allows to map the scene to a custom part of the render target, and can be used for split-screen or for displaying a minimap, for example. If the source rectangle has not the same size as the viewport, its contents will be stretched to fit in.

To apply a view, you have to assign it to the render target. Then, every objects drawn in this render target will be affected by the view until you use another view.

Usage example:

```
window = sfml.RenderWindow(sfml.VideoMode(640, 480), 'Title')

# Initialize the view with a rectangle located at (100, 100) and
# a size of 400x200
view = sfml.View.from_rect(sfml.FloatRect(100, 100, 400, 200))

# Rotate it by 45 degrees
view.rotate(45)

# Set its target viewport to be half of the window
view.view_port = sfml.FloatRect(0.0, 0.0, 0.5, 1.0)

# Apply it
window.view = view

# Render stuff
window.draw(some_sprite)

# Set the default view back
window.view = window.default_view

# Render stuff not affected by the view
window.draw(some_text)
```

center
The center of the view, as a tuple. The value can also be set from a *Vector2f* object.

height
Shortcut for `self.size[1]`.

rotation
The orientation of the view, as a float. Default value: 0.0 degree.

size
The size of the view, as a tuple. The value can also be set from a *Vector2f* object.

viewport
The target viewport. The viewport is the rectangle into which the contents of the view are displayed, expressed as a factor (between 0 and 1) of the size of the *RenderTarget* to which the view is applied. For example, a view which takes the left side of the target would be defined with `View.viewport = sfml.FloatRect(0, 0, 0.5, 1)`. By default, a view has a viewport which covers the entire target.

width
Shortcut for `self.size[0]`.

classmethod from_center_and_size(*center*, *size*)
Return a new view created from a center and a size. *center* and *size* can be either tuples or *Vector2f*.

classmethod from_rect(*rect*)
Return a new view created from a rectangle. *rect* can be a tuple or a *FloatRect*.

move(*float x*, *float y*)
Move the view relatively to its current position.

reset(*rect*)
Reset the view to the given rectangle. *rect* can be a tuple or a *FloatRect*. Note that this function resets the rotation angle to 0.

rotate(*float angle*)
Rotate the view relatively to its current orientation.

zoom(*float factor*)
Resize the view rectangle relatively to its current size. Resizing the view simulates a zoom, as the zone displayed on screen grows or shrinks. *factor* is a multiplier:

- 1 keeps the size unchanged.
- > 1 makes the view bigger (objects appear smaller).
- < 1 makes the view smaller (objects appear bigger).

7.3.3 Drawing

Note: Creating your own drawables

A drawable is an object that can be drawn directly to render target, e.g. you can write `window.draw(a_drawable)`.

In the past, creating a drawable involved inheriting the `Drawable` class and overriding its `render()` method. With the new graphics API, you only have to define a `draw()` method that takes two parameters:

```
def draw(self, target, states):
    target.draw(self.logo)
    target.draw(self.princess)
```

target and *states* are *RenderTarget* and *RenderStates* objects, respectively. See `examples/customdrawable.py` for a working example, which also shows how you can use the low-level API.

The *Transformable* class now contains the operations that can be appied to a drawable. Most drawable (i.e. objects that can be drawn on a target) are transformable as well.

C++ documentation:

- http://www.sfml-dev.org/documentation/2.0/classsf_1_1Drawable.php
- http://www.sfml-dev.org/documentation/2.0/classsf_1_1Transformable.php

---

class sfml.**RenderStates**(*blend_mode=-1*, *shader=None*, *texture=None*, *transform=None*)

The constructor first creates a default RenderStates object, then sets its attributes with respect to the provided arguments. Constructing a default set of render states is equivalent to using *RenderStates.DEFAULT*. The default set defines

- the *BLEND_ALPHA* blend mode,
- the *Transform.IDENTITY* transform,
- no texture (None),
- no shader (None).

Contains the states used for drawing to a *RenderTarget*. There are four global states that can be applied to the drawn objects:

- The blend mode: how pixels of the object are blended with the background.
- The transform: how the object is positioned/rotated/scaled.
- The texture: which image is mapped to the object.
- The shader: which custom effect is applied to the object.

High-level objects such as sprites or text force some of these states when they are drawn. For example, a sprite will set its own texture, so that you don't have to care about it when drawing the sprite.

The transform is a special case: sprites, texts and shapes (and it's a good idea to do it with your own drawable classes too) combine their transform with the one that is passed in the RenderStates structure. So that you can use a "global" transform on top of each object's transform.

Most objects, especially high-level drawables, can be drawn directly without defining render states explicitely — the default set of states is ok in most cases:

```
window.draw(sprite)
```

If you just want to specify a shader, you can pass it directly to the *RenderTarget.draw()* method:

```
window.draw(sprite, shader)
```

Note that unlike in C++ SFML, this only works for shaders and not for other render states. This is because adding other possibilities means writing a lot of boilerplate code in the binding, and shader seemed to be most used state when writing this method.

When you're inside the draw method of a drawable object, you can either pass the render states unmodified, or change some of them. For example, a transformable object will combine the current transform with its own transform. A sprite will set its texture. Etc.

DEFAULT

A RenderStates object with the default values, as a class attribute.

blend_mode

See *Blend modes* for a list of the valid values.

shader
A *Shader* object.

texture
A *Texture* object.

transform
A *Transform* object.

class sfml.**RenderTarget**
Base class for *RenderWindow* and *RenderTexture*. It is abstract; the constructor will raise NotImplementedError if you call it.

RenderTarget defines the common behaviour of all the 2D render targets. It makes it possible to draw 2D entities like sprites, shapes, text without using any OpenGL command directly.

A *RenderTarget* is also able to use views (*View*), which are some kind of 2D cameras. With views you can globally scroll, rotate or zoom everything that is drawn, without having to transform every single entity.

On top of that, render targets are still able to render direct OpenGL stuff. It is even possible to mix together OpenGL calls and regular SFML drawing commands. When doing so, make sure that OpenGL states are not messed up by calling the *push_gl_states()*/*pop_gl_states()* methods.

default_view
Read-only. The default view has the initial size of the render target, and never changes after the target has been created.

height
Read-only. The height of the rendering region of the target.

size
Read-only. The size of the rendering region of the target, as a tuple.

view
The view is like a 2D camera, it controls which part of the 2D scene is visible, and how it is viewed in the render-target. The new view will affect everything that is drawn, until another view is set. The render target keeps its own copy of the view object, so it is not necessary to keep the original one alive after calling this function. To restore the original view of the target, you can pass the result of *default_view* to this function.

width
Read-only. The width of the rendering region of the target.

clear(*[color]*)
Clear the entire target with a single color. This function is usually called once every frame, to clear the previous contents of the target. The default is black.

convert_coords(*int x*, *int y*[, *view=None*])
Convert a point from target coordinates to view coordinates. Initially, a unit of the 2D world matches a pixel of the render target. But if you define a custom view, this assertion is not true anymore, e.g. a point located at (10, 50) in your render target (for example a window) may map to the point (150, 75) in your 2D world — for example if the view is translated by (140, 25). For render windows, this method is typically used to find which point (or object) is located below the mouse cursor.

When the *view* argument isn't provided, the current view of the render target is used.

draw(*drawable*, *...*)
drawable may be:

- A built-in drawable, such as *Sprite* or *Text*, or a user-made drawable (see *Creating your own drawables*). You can pass a second argument of type *Shader* or *RenderStates*. Example:

```
window.draw(sprite, shader)
```

- A list or a tuple of *Vertex* objects. You must pass a *primitive type* as a second argument, and can pass a *Shader* or *RenderStates* as a third argument. Example:

```
window.draw(vertices, sfml.QUADS, shader)
```

See `examples/vertices.py` for a working example.

get_viewport(*view*)
Return the viewport of a view applied to this render target, as an *IntRect*. The viewport is defined in the view as a ratio, this method simply applies this ratio to the current dimensions of the render target to calculate the pixels rectangle that the viewport actually covers in the target.

pop_gl_states()
Restore the previously saved OpenGL render states and matrices. See *push_gl_states()*.

push_gl_states()
Save the current OpenGL render states and matrices. This method can be used when you mix SFML drawing and direct OpenGL rendering. Combined with *pop_gl_states()*, it ensures that:

- SFML's internal states are not messed up by your OpenGL code.
- Your OpenGL states are not modified by a call to a SFML method.

More specifically, it must be used around code that calls `draw()` methods. Example:

```
# OpenGL code here...
window.push_gl_states()
window.draw(...)
window.draw(...)
window.pop_gl_states()
# OpenGL code here...
```

Note that this method is quite expensive: it saves all the possible OpenGL states and matrices, even the ones you don't care about. Therefore it should be used wisely. It is provided for convenience, but the best results will be achieved if you handle OpenGL states yourself (because you know which states have really changed, and need to be saved and restored). Take a look at the *reset_gl_states()* method if you do so.

reset_gl_states()
Reset the internal OpenGL states so that the target is ready for drawing. This function can be used when you mix SFML drawing and direct OpenGL rendering, if you choose not to use *push_gl_states()*/*pop_gl_states()*. It ensures that all OpenGL states needed by SFML are set, so that subsequent draw() calls will work as expected.

Example:

```
# OpenGL code here...
glPushAttrib(...)
window.reset_gl_states()
window.draw(...)
window.draw(...)
glPopAttrib(...)
# OpenGL code here...
```

class `sfml.`**RenderTexture**(*int width*, *int height*[, *bool depth=False*])
This class inherits *RenderTarget*.

Target for off-screen 2D rendering into an texture. *RenderTexture* is the little brother of *RenderWindow*.

It implements the same 2D drawing and OpenGL-related functions (see their base class *RenderTarget* for more details), the difference is that the result is stored in an off-screen texture rather than being show in a window.

Rendering to a texture can be useful in a variety of situations:

- Precomputing a complex static texture (like a level's background from multiple tiles).
- Applying post-effects to the whole scene with shaders.
- Creating a sprite from a 3D object rendered with OpenGL.
- Etc.

Usage example:

```
# Create a new render-window
window = sfml.RenderWindow(sf.VideoMode(800, 600), 'pySFML window')

# Create a new render texture
render_texture = sfml.RenderTexture(500, 500)

# The main loop
while window.open:
    # Event processing
    # ...

    # Clear the whole texture with red color
    render_texture.clear(sfml.Color.RED)

    # Draw stuff to the texture
    render_texture.draw(sprite)  # sprite is a Sprite
    render_texture.draw(shape)   # shape is a Shape
    render_texture.draw(text)    # text is a Text

    # We're done drawing to the texture
    render_texture.display()

    # Now we start rendering to the window, clear it first
    window.clear()

    # Draw the texture
    sprite = sfml.Sprite(render_texture.texture)
    window.draw(sprite);

    # End the current frame and display its contents on screen
    window.display()
```

active
: Write-only. If true, the render texture's context becomes current for future OpenGL rendering operations (so you shouldn't care about it if you're not doing direct OpenGL stuff). Only one context can be current in a thread, so if you want to draw OpenGL geometry to another render target (like a *RenderWindow*), don't forget to activate it again.

  If an error occurs, *PySFMLException* is raised.

texture
: Read-only.The target texture, as a *Texture*. After drawing to the render-texture and calling *display()*, you can retrieve the updated texture using this function, and draw it using a sprite (for example).

> **Warning:** Textures obtained with this property should never be modified. The object itself is a normal *Texture* object, but the underlying C++ object is specified as `const` and a C++ compiler wouldn't let you attempt to modify it.

smooth
Whether the smooth filtering is enabled or not. Default value: `False`.

display()
Update the contents of the target texture. This method updates the target texture with what has been drawn so far. Like for windows, calling this function is mandatory at the end of rendering. Not calling it may leave the texture in an undefined state.

class `sfml.`**Shader**
The constructor will raise `NotImplementedError` if called. Use class methods like *load_from_file()* or *load_from_memory()* instead.

Shaders are programs written using a specific language, executed directly by the graphics card and allowing to apply real-time operations to the rendered entities.

There are two kinds of shaders:

- Vertex shaders, that process vertices.
- Fragment (pixel) shaders, that process pixels.

A shader can be composed of either a vertex shader alone, a fragment shader alone, or both combined (see the variants of the load classmethods).

Shaders are written in GLSL, which is a C-like language dedicated to OpenGL shaders. You'll probably need to learn its basics before writing your own shaders for SFML.

Like any Python program, a shader has its own variables that you can set from your Python. *Shader* handles four different types of variables:

- floats
- vectors (2, 3 or 4 components)
- textures
- transforms (matrices)

The value of the variables can be changed at any time with *set_parameter()*:

```
shader.set_parameter('offset', 2.0)
shader.set_parameter('color', 0.5, 0.8, 0.3)
shader.set_parameter('matrix', transform); # transform is a sfml.Transform
shader.set_parameter('overlay', texture) # texture is a sfml.Texture
shader.set_parameter('texture', sfml.Shader.CURRENT_TEXTURE)
```

The special *Shader.CURRENT_TEXTURE* argument maps the given texture variable to the current texture of the object being drawn (which cannot be known in advance).

To apply a shader to a drawable, you must pass it as an additional parameter to *RenderTarget.draw()*:

```
window.draw(sprite, shader)
```

Which is in fact just a shortcut for this:

```
states = sfml.RenderStates()
states.shader = shader
window.draw(sprite, states)
```

Shaders can be used on any drawable, but some combinations are not interesting. For example, using a vertex shader on a *Sprite* is limited because there are only 4 vertices, the sprite would have to be subdivided in order to apply wave effects. Another bad example is a fragment shader with *Text*: the texture of the text is not the actual text that you see on screen, it is a big texture containing all the characters of the font in an arbitrary order; thus, texture lookups on pixels other than the current one may not give you the expected result.

Shaders can also be used to apply global post-effects to the current contents of the target (like the old `PostFx` class in SFML 1). This can be done in two different ways:

- Draw everything to a *RenderTexture*, then draw it to the main target using the shader.
- Draw everything directly to the main target, then use *Texture.update()* to copy its contents to a texture and draw it to the main target using the shader.

The first technique is more optimized because it doesn't involve retrieving the target's pixels to system memory, but the second one doesn't impact the rendering process and can be easily inserted anywhere without impacting all the code.

Like *Texture* that can be used as a raw OpenGL texture, *Shader* can also be used directly as a raw shader for custom OpenGL geometry:

```
window.active = True
shader.bind()
# render OpenGL geometry ...
shader.unbind()
```

IS_AVAILABLE
True if the system supports shaders. You shoul always test this class attribute before using the shader features. If it is false, then any attempt to use *Shader* will fail.

CURRENT_TEXTURE
Special type/value that can be passed to *set_parameter()*, and that represents the texture of the object being drawn.

FRAGMENT
Fragment (pixel) shader type, as an int class attribute.

VERTEX
Vertex shader type, as an int class attribute.

classmethod load_both_types_from_file(*str vertex_shader_filename*, *str fragment_shader_filename*)
Load both the vertex and the fragment shaders. If one of them fails to load, the shader is left empty (the valid shader is unloaded). The sources must be text files containing valid shaders in GLSL language. GLSL is a C-like language dedicated to OpenGL shaders; you'll probably need to read a good documentation for it before writing your own shaders.

PySFMLException is raised if an error occurs.

classmethod load_both_types_from_memory(*str vertex_shader*, *str fragment_shader*)
Load both the vertex and the fragment shaders. If one of them fails to load, the shader is left empty (the valid shader is unloaded). The sources must be valid shaders in GLSL language. GLSL is a C-like language dedicated to OpenGL shaders; you'll probably need to read a good documentation for it before writing your own shaders.

PySFMLException is raised if an error occurs.

classmethod load_both_types_from_stream(*InputStream vertex_stream*, *InputStream fragment_stream*)
Load both the vertex and fragment shaders from custom streams. If one of them fails to load, the shader is left empty (the valid shader is unloaded). The source codes must be valid shaders in GLSL language. GLSL

is a C-like language dedicated to OpenGL shaders; you'll probably need to read a good documentation for it before writing your own shaders.

PySFMLException is raised if an error occurs.

classmethod load_from_file(*filename*, *int type*)
Load a single shader, either vertex or fragment, identified by the *type* parameter, which must be *Shader.FRAGMENT* or *Shader.VERTEX*. The source must be a text file containing a valid shader in GLSL language. GLSL is a C-like language dedicated to OpenGL shaders; you'll probably need to read a good documentation for it before writing your own shaders.

PySFMLException is raised if an error occurs.

classmethod load_from_memory(*str shader*, *int type*)
Load a single shader, either vertex or fragment, identified by the *type* argument, which must be *Shader.FRAGMENT* or *Shader.VERTEX*. The source code must be a valid shader in GLSL language. GLSL is a C-like language dedicated to OpenGL shaders; you'll probably need to read a good documentation for it before writing your own shaders.

PySFMLException is raised if an error occurs.

classmethod load_from_stream(*InputStream stream*, *int type*)
Load a single shader, either vertex or fragment, identified by the *type* argument, which must be *Shader.FRAGMENT* or *Shader.VERTEX*. GLSL is a C-like language dedicated to OpenGL shaders; you'll probably need to read a good documentation for it before writing your own shaders.

PySFMLException is raised if an error occurs.

bind()
Bind the shader for rendering (activate it). This method is normally for internal use only, unless you want to use the shader with a custom OpenGL rendering instead of a SFML drawable:

```
window.active = True
shader.bind()
# ... render OpenGL geometry ...
shader.unbind()
```

set_parameter(*str name*, ...)
Set a shader parameter.

The first parameter, *name*, is the name of the variable to change in the shader. After *name*, you can pass an argument or several floats, depending on your need:

- 1 float,
- 2 floats,
- 3 floats,
- 4 floats,
- a color,
- a transform,
- a texture.

If you want to pass the texture of the object being drawn, which cannot be known in advance, you can pass the special value *CURRENT_TEXTURE*:

```
shader.set_parameter('the_texture', sfml.Shader.CURRENT_TEXTURE)
```

unbind()
Unbind the shader (deactivate it). This method is normally for internal use only, unless you want to use the shader with a custom OpenGL rendering instead of a SFML drawable.

class sfml.**Transform**([*float a00*, *float a01*, *float a02*, *float a10*, *float a11*, *float a12*, *float a20*, *float a21*, *float a22*])
If called with no arguments, the value is set to the *IDENTITY* transform.

A *Transform* is a 3x3 transform matrix that specifies how to translate, rotate, scale, shear, project, etc. In mathematical terms, it defines how to transform a coordinate system into another.

For example, if you apply a rotation transform to a sprite, the result will be a rotated sprite. And anything that is transformed by this rotation transform will be rotated the same way, according to its initial position.

Transforms are typically used for drawing. But they can also be used for any computation that requires to transform points between the local and global coordinate systems of an entity (like collision detection).

Example:

```
# Define a translation transform
translation = sfml.Transform()
translation.translate(20, 50)

# Define a rotation transform
rotation = sf.Transform()
rotation.rotate(45)

# Combine them
transform = translation * rotation

# Use the result to transform stuff...
point = transform.transform_point(10, 20)
rect = transform.transform_rect(sfml.FloatRect(0, 0, 10, 100))
```

This class provides the following special methods:

- * and *= operators.
- str() returns the content of the matrix in a human-readable format.

IDENTITY
Class attribute containing the identity matrix.

matrix
Read-only. a list of 16 floats containing the transform elements as a 4x4 matrix, which is directly compatible with OpenGL functions.

combine(*transform*)
Combine the current transform with *transform*. The result is a transform that is equivalent to applying this followed by transform. Mathematically, it is equivalent to a matrix multiplication.

copy()
Return a new transform object with the same content as self.

get_inverse()
Return the inverse of the transform. If the inverse cannot be computed, an *IDENTITY* transform is returned.

rotate(*float angle*[, *float center_x*, *float center_y*])
Combine the current transform with a rotation. This method returns self, so calls can be chained:

```
transform = sfml.Transform()
transform.rotate(90).translate(50, 20)
```

The center of rotation can be provided with *center_x* and *center_y*, so that you can build rotations around arbitrary points more easily (and efficiently) than the usual `translate(-center).rotate(angle).translate(center)`.

scale (*float scale_x*, *float scale_y*[, *float*, *center_x*, *float center_y*])
Combine the current transform with a scaling. The center of scaling can be provided with *center_x* and *center_y*, so that you can build scaling around arbitrary points more easily (and efficiently) than the usual `translate(-center).scale(factors).translate(center)`.

This method returns self, so calls can be chained:

```
transform = sfml.Transform()
transform.scale(2, 1, 8, 3).rotate(45)
```

transform_point (*float x*, *float y*)
Transform the point and return it as a tuple.

transform_rect (*FloatRect rectangle*)
Transform a rectangle and return it as a *FloatRect*. Since SFML doesn't provide support for oriented rectangles, the result of this function is always an axis-aligned rectangle. Which means that if the transform contains a rotation, the bounding rectangle of the transformed rectangle is returned.

translate (*float x*, *float y*)
Combine the current transform with a translation. This method returns self, so calls can be chained:

```
transform = sfml.Transform()
transform.translate(100, 200).rotate(45)
```

class sfml.**Transformable**
Decomposed transform defined by a position, a rotation and a scale.

This class is provided for convenience, on top of *Transform*.

Transform, as a low-level class, offers a great level of flexibility but it's not always convenient to manage. One can easily combine any kind of operation, such as a translation followed by a rotation followed by a scaling, but once the result transform is built, there's no way to go backward and, say, change only the rotation without modifying the translation and scaling. The entire transform must be recomputed, which means that you need to retrieve the initial translation and scale factors as well, and combine them the same way you did before updating the rotation. This is a tedious operation, and it requires to store all the individual components of the final transform.

That's exactly what *Transformable* was written for: it hides these variables and the composed transform behind an easy to use interface. You can set or get any of the individual components without worrying about the others. It also provides the composed transform (as a *Transform* object), and keeps it up-to-date.

In addition to the position, rotation and scale, *Transformable* provides an "origin" component, which represents the local origin of the three other components. Let's take an example with a 10x10 pixels sprite. By default, the sprite is positionned/rotated/scaled relatively to its top-left corner, because it is the local point (0, 0). But if we change the origin to be (5, 5), the sprite will be positionned/rotated/scaled around its center instead. And if we set the origin to (10, 10), it will be transformed around its bottom-right corner.

To keep the *Transformable* class simple, there's only one origin for all the components. You cannot position the sprite relatively to its top-left corner while rotating it around its center, for example. To do this kind of thing, use *Transform* directly.

Transformable can be used as a base class. It is often combined with a *draw() method* — that's what SFML's sprites, texts and shapes do:

```
// TODO: port to Python
class MyEntity : public sf::Transformable, public sf::Drawable
{
    virtual void draw(sf::RenderTarget& target, sf::RenderStates states) const
    {
        states.transform *= getTransform();
        target.draw(..., states);
    }
};

MyEntity entity;
entity.setPosition(10, 20);
entity.setRotation(45);
window.draw(entity);
```

It can also be used as a member, if you don't want to use its API directly (because you don't need all its functions, or you have different naming conventions for example):

```
// TODO: port to Python
class MyEntity
{
public :
    void SetPosition(const MyVector& v)
    {
        myTransform.setPosition(v.x(), v.y());
    }

    void Draw(sf::RenderTarget& target) const
    {
        target.draw(..., myTransform.getTransform());
    }

private :
    sf::Transformable myTransform;
};
```

origin
The local origin of the object, as a tuple. When setting the attribute, you can also pass a *Vector2f*. The origin of an object defines the center point for all transformations (position, scale, rotation). The coordinates of this point must be relative to the top-left corner of the object, and ignore all transformations (position, scale, rotation). The default origin of a transformable object is (0, 0).

position
The position of the object, as a tuple. When setting the attribute, you can also pass a *Vector2f*. This method completely overwrites the previous position. See *move()* to apply an offset based on the previous position instead. The default position of a transformable object is (0, 0).

rotation
The orientation of the object, as a float in the range [0, 360]. This method completely overwrites the previous rotation. See *rotate()* to add an angle based on the previous rotation instead. The default rotation of a transformable object is 0.

scale
The scale factors of the object. This method completely overwrites the previous scale. See the *scale()* to add a factor based on the previous scale instead. The default scale of a transformable object is (1, 1).

The object returned by this property will behave like a tuple, but it might be important in some cases to know that its exact type isn't tuple, although its class does inherit tuple. In practice it should behave just like one, except if you write code that checks for exact type using the `type()` function. Instead, use

isinstance():

```
if isinstance(some_object, tuple):
    pass # We now know that some_object is a tuple
```

x
Shortcut for `self.position[0]`.

y
Shortcut for `self.position[1]`.

get_inverse_transform()
Return the inverse of the combined *Transform* of the object.

get_transform()
Return the combined *Transform* of the object.

move(*float x*, *float y*)
Move the object by a given offset. This method adds to the current position of the object, unlike *position()* which overwrites it. So it is equivalent to the following code:

```
object.position = object.position + offset
```

rotate(*float angle*)
Rotate the object. This method adds to the current rotation of the object, unlike *rotation()* which overwrites it. So it is equivalent to the following code:

```
object.rotation = object.rotation + angle
```

scale(*float x*, *float y*)
Scale the object. This method multiplies the current scale of the object, unlike the *scale* attribute which overwrites it. So it is equivalent to the following code:

```
scale = object.scale
object.scale(scale[0] * factor_x, scale[1] * factor_y)
```

class sfml.**Vertex**([*position*[, *color*[, *tex_coords*]]])
A vertex is an improved point. It has a position and other extra attributes that will be used for drawing: a color and a pair of texture coordinates.

The vertex is the building block of drawing. Everything which is visible on screen is made of vertices. They are grouped as 2D primitives (triangles, quads, ... see *Blend modes*), and these primitives are grouped to create even more complex 2D entities such as sprites, texts, etc.

If you use the graphical entities of SFML (*Sprite*, *Text*, *Shape*) you won't have to deal with vertices directly. But if you want to define your own 2D entities, such as tiled maps or particle systems, using vertices will allow you to get maximum performances.

This class provides the following special methods:

- `repr(vertex)` returns a description in format `Vertex(position, color, tex_coords`.

Example:

```
# define a 100x100 square, red, with a 10x10 texture mapped on it
vertices = [sfml.Vertex((0, 0), sfml.Color.RED, (0, 0)),
            sfml.Vertex((0, 100), sfml.Color.RED, (0, 10)),
            sfml.Vertex((100, 100), sfml.Color.RED, (10, 10)),
            sfml.Vertex((100, 0), sfml.Color.RED, (10, 0))]

# draw it
window.draw(vertices, sfml.QUADS)
```

Note: although texture coordinates are supposed to be an integer amount of pixels, their type is float because of some buggy graphics drivers that are not able to process integer coordinates correctly.

color
Color of the vertex.

position
2D position of the vertex. The value is always retrieved as a tuple. It can be set as a tuple or a *Vector2f*.

tex_coords
Coordinates of the texture's pixel map to the vertex. The value is always retrieved as a tuple. It can be set as a tuple or a *Vector2f*.

copy()
Return a new vertex with the same value as self.

Shapes

class sfml.**Shape**
This abstract class inherits *Transformable*.

Shape is a drawable class that allows to define and display a custom convex shape on a render target.

Every shape has the following attributes:

- a texture,
- a texture rectangle,
- a fill color,
- an outline color,
- an outline thickness.

Each feature is optional, and can be disabled easily:

- the texture can be None,
- the fill/outline colors can be *Color.TRANSPARENT*,
- the outline thickness can be zero.

You can write your own derived shape class, there are only two methods to override:

- *get_point_count()* must return the number of points of the shape,
- *get_point()* must return the points of the shape.

A few concrete shapes are provided: *RectangleShape*, *CircleShape* and *ConvexShape*.

fill_color
The fill color of the shape. This color is modulated (multiplied) with the shape's texture if any. It can be used to colorize the shape, or change its global opacity. You can use *Color.TRANSPARENT* to make the inside of the shape transparent, and have the outline alone. By default, the shape's fill color is opaque white.

global_bounds
Read-only. The global bounding rectangle of the entity, as a *FloatRect*. The returned rectangle is in global coordinates, which means that it takes in account the transformations (translation, rotation, scale, ...) that are applied to the entity. In other words, this function returns the bounds of the sprite in the global 2D world's coordinate system.

local_bounds
Read-only. The local bounding rectangle of the entity, as a *FloatRect*. The returned rectangle is in local coordinates, which means that it ignores the transformations (translation, rotation, scale, ...) that are applied to the entity. In other words, this function returns the bounds of the entity in the entity's coordinate system.

texture
The source texture of the shape. Can be `None` to disable texturing. Also see *set_texture()*, which allows you to update *texture_rect* automatically.

texture_rect
The sub-rectangle of the texture that the shape will display. The texture rect is useful when you only want to display a part of the texture. By default, the texture rect covers the entire texture.

outline_color
The outline color of the shape. You can use *Color.TRANSPARENT* to disable the outline. By default, the shape's outline color is opaque white.

outline_thickness
The thickness of the shape's outline, as a float. This number cannot be negative. Using zero disables the outline. By default, the outline thickness is 0.0.

get_point(*int index*)
This method should be overriden to return a tuple or a *Vector2f* containing the coordinates at the position *index*.

get_point_count()
This method should be overriden to return the number of points, as an integer.

set_texture(*texture*[, *reset_rect=False*])
Set the source texture of the shape. *texture* can be `None` to disable texturing. If *reset_rect* is true, the *texture_rect* property of the shape is automatically adjusted to the size of the new texture. If it is false, the texture rect is left unchanged.

Calling this method does the same thing as modifiying the *texture* attribute, except when the *reset_rect* parameter is used.

update()
Recompute the internal geometry of the shape. This method must be called by the derived class every-time the shape's points change (i.e. the result of either *get_point_count()* or *get_point()* is different). This includes when the shape object is created.

If you call this method from a built-in shape, it will raise `NotImplementedError`.

class `sfml.`**RectangleShape**([*size*])
This class inherits *Shape*. *size* can be either a tuple or a *Vector2f*.

Usage example:

```
rectangle = sfml.RectangleShape((100, 50))
rectangle.outline_color = sfml.Color.RED
rectangle.outline_thickness = 5
rectangle.position = (10, 20)
# ...
window.draw(rectangle)
```

size
The size of the rectangle, as a tuple. The value can also be set from a *Vector2f*.

class `sfml.`**CircleShape**([*float radius*[, *int point_count*]])
This class inherits *Shape*.

Usage example:

```
circle = sfml.CircleShape(150)
circle.outline_color = sfml.Color.Red
circle.outline_thickness = 5
circle.position = (10, 20)
# ...
window.draw(circle)
```

Since the graphics card can't draw perfect circles, we have to fake them with multiple triangles connected to each other. The *point_count* property defines how many of these triangles to use, and therefore defines the quality of the circle.

The number of points can also be used for another purpose; with small numbers you can create any regular polygon shape: equilateral triangle, square, pentagon, hexagon, ...

point_count
The number of points in the circle.

radius
The radius of the circle, as a float.

class sfml.**ConvexShape**([*int point_count*])
This class inherits *Shape*.

Specialized shape representing a convex polygon.

It is important to keep in mind that a convex shape must always be... convex, otherwise it may not be drawn correctly. Moreover, the points must be defined in order; using a random order would result in an incorrect shape.

Usage example:

```
polygon = sfml.ConvexShape(3)
polygon.set_point(0, (0, 0))
polygon.set_point(1, (0, 10))
polygon.set_point(2, (25, 5))
polygon.outline_color = sfml.Color.RED
polygon.outline_thickness = 5
polygon.position = (10, 20)
# ...
window.draw(polygon)
```

get_point(*int index*)
Return the position of a point. The result is undefined if *index* is out of the valid range.

get_point_count()
Return the number of points of the polygon.

set_point(*int index*, *point*)
Set the position of a point. Don't forget that the polygon must remain convex, and the points need to stay ordered! *set_point_count()* must be called first in order to set the total number of points. The result is undefined if index is out of the valid range.

point may be either a tuple or a *Vector2f*.

set_point_count(*int count*)
Set the number of points of the polygon. *count* must be greater than 2 to define a valid shape.

Image dislay

class sfml.**Image**(*int width*, *int height*[, *color*])

Image is an abstraction to manipulate images as bidimensional arrays of pixels. It allows you to load, manipulate and save images.

The constructor create images of the specified size, filled with a color. For loading images, you should use one of the class methods. *load_from_file()* is the most common one.

Image can handle a unique internal representation of pixels, which is RGBA 32 bits. This means that a pixel must be composed of 8 bits red, green, blue and alpha channels — just like a *Color*. All the functions that return an array of pixels follow this rule, and all parameters that you pass to *Image* methods (such as *load_from_pixels()*) must use this representation as well.

An image can be copied, but you should note that it's a heavy resource.

Usage example:

```
# Load an image file from a file
background = sfml.Image.load_from_file('background.jpg')

# Create a 20x20 image filled with black color
image = sfml.Image(20, 20, sfml.Color.BLACK)

# Copy image1 on image2 at position (10, 10)
image.copy(background, 10, 10)

# Make the top-left pixel transparent
color = image[0,0]
color.a = 0
image[0,0] = color

# Save the image to a file
image.save_to_file('result.png')
```

This class provides the following special methods:

- image[tuple] returns a pixel from the image, as a *Color* object. Equivalent to *get_pixel()*. Example:

```
print image[0,0]  # Create tuple implicitly
print image[(0,0)]  # Create tuple explicitly
```

- image[tuple] = color sets a pixel of the image to a *Color* object value. Equivalent to *set_pixel()*. Example:

```
image[0,0] = sfml.Color(10, 20, 30)  # Create tuple implicitly
image[(0,0)] = sfml.Color(10, 20, 30)  # Create tuple explicitly
```

height
Read-only. The height of the image.

size
Read-only. The size of the image, as a tuple.

width
Read-only. The width of the image.

classmethod load_from_file(*filename*)
Load the image from *filename* on disk and return a new *Image* object. The supported image formats are bmp, png, tga, jpg, gif, psd, hdr and pic. Some format options are not supported, like progressive jpeg.

PySFMLException is raised if an error occurs.

classmethod load_from_memory(*bytes mem*)
Load the image from a file in memory. The supported image formats are bmp, png, tga, jpg, gif, psd, hdr and pic. Some format options are not supported, like progressive jpeg.

PySFMLException is raised if an error occurs.

classmethod load_from_pixels(*int width*, *int height*, *bytes pixels*)
Return a new image, created from a str/bytes object of pixels. *pixels* is assumed to contain 32-bits RGBA pixels, and have the given *width* and *height*. If not, the behavior is undefined. If *pixels* is `None`, an empty image is created.

classmethod load_from_stream(*InputStream stream*)
Load the image from a custom stream. The supported image formats are bmp, png, tga, jpg, gif, psd, hdr and pic. Some format options are not supported, like progressive jpeg.

PySFMLException is raised if an error occurs.

copy(*Image source*, *int dest_x*, *int dest_y*[, *source_rect*, *apply_alpha*])
Copy pixels from another image onto this one. This method does a slow pixel copy and should not be used intensively. It can be used to prepare a complex static image from several others, but if you need this kind of feature in real-time you'd better use *RenderTexture*.

Without *source_rect*, the whole image is copied. *source_rect* can be either an *IntRect* or a tuple.

If *apply_alpha* is provided, the transparency of *source*'s pixels is applied. If it isn't, the pixels are copied unchanged with their alpha value.

create_mask_from_color(*color*, *int alpha*)
Create a transparency mask from a specified color-key. This method sets the alpha value of every pixel matching the given color to *alpha* (0 by default), so that they become transparent.

flip_horizontally()
Flip the image horizontally (left <-> right).

flip_vertically()
Flip the image vertically (top <-> bottom).

get_pixel(*int x*, *int y*)
Return the color of the pixel at *(x, y)*.

`IndexError` is raised if the pixel is out of range.

get_pixels()
Return a str (in Python 2) or a bytes (Python 3) object to the pixels. The returned value points to an array of RGBA pixels made of 8 bits integers components. The size of the object is *width* * *height* * 4. If the image is empty, `None` is returned.

save_to_file(*filename*)
Save the image to a file on disk. The format of the image is automatically deduced from the extension. The supported image formats are bmp, png, tga and jpg. The destination file is overwritten if it already exists. This method fails if the image is empty.

PySFMLException is raised if saving fails.

set_pixel(*int x*, *int y*, *color*)
Set the color of the pixel at *(x, y)* to *color*. This method doesn't check the validity of the pixel coordinates, using out-of-range values will result in an undefined behaviour.

`IndexError` is raised if the pixel is out of range.

class sfml.**Texture**([*int width*[, *int height*]])

The constructor serves the same purpose as `Texture.create()` in C++ SFML. It raises *PySFMLException* if texture creation fails.

Image living on the graphics card that can be used for drawing. A texture lives in the graphics card memory, therefore it is very fast to draw a texture to a render target, or copy a render target to a texture (the graphics card can access both directly).

Being stored in the graphics card memory has some drawbacks. A texture cannot be manipulated as freely as a *Image*, you need to prepare the pixels first and then upload them to the texture in a single operation (see *update()*).

Texture makes it easy to convert from/to *Image*, but keep in mind that these calls require transfers between the graphics card and the central memory, therefore they are slow operations.

A texture can be loaded from an image, but also directly from a file/memory/stream. The necessary shortcuts are defined so that you don't need an image first for the most common cases. However, if you want to perform some modifications on the pixels before creating the final texture, you can load your file to a *Image*, do whatever you need with the pixels, and then call *load_from_image()*.

Since they live in the graphics card memory, the pixels of a texture cannot be accessed without a slow copy first. And they cannot be accessed individually. Therefore, if you need to read the texture's pixels (like for pixel-perfect collisions), it is recommended to store the collision information separately, for example in an array of booleans.

Like *Image*, Texture can handle a unique internal representation of pixels, which is RGBA 32 bits. This means that a pixel must be composed of 8 bits red, green, blue and alpha channels — just like a *Color*.

Usage example:

```
# This example shows the most common use of Texture:
# drawing a sprite

# Load a texture from a file
texture = sfml.load_from_file('texture.png')

# Assign it to a sprite
sprite = sfml.Sprite(texture)

# Draw the textured sprite
window.draw(sprite)
```

```
# This example shows another common use of Texture:
# streaming real-time data, like video frames

# Create an empty texture
texture = sfml.Texture(640, 480)

# Create a sprite that will display the texture
sprite = sfml.Sprite(texture)

while True:
    # ...

    # Update the texture
    # Get a fresh chunk of pixels (the next frame of a movie, for example)
    # This should be a string object in Python 2, and a bytes object in Python 3
    pixels = get_pixels()
    texture.update(pixels)
```

```
        # draw it
        window.draw(sprite)

        # ...
```

MAXIMUM_SIZE
Read-only. The maximum texture size allowed, as a class attribute. This maximum size is defined by the graphics driver. You can expect a value of 512 pixels for low-end graphics card, and up to 8192 pixels or more for newer hardware.

NORMALIZED
Constant for the type of texture coordinates where the range is [0 .. 1], as a class attribute.

PIXELS
Constant for the type of texture coordinates where the range is [0 .. size], as a class attribute.

height
Read-only. The height of the texture.

repeated
Whether the texture is repeated or not. Repeating is involved when using texture coordinates outside the texture rectangle [0, 0, width, height]. In this case, if repeat mode is enabled, the whole texture will be repeated as many times as needed to reach the coordinate (for example, if the X texture coordinate is 3 * width, the texture will be repeated 3 times). If repeat mode is disabled, the "extra space" will instead be filled with border pixels. Repeating is disabled by default.

> **Warning:** On very old graphics cards, white pixels may appear when the texture is repeated. With such cards, repeat mode can be used reliably only if the texture has power-of-two dimensions (such as 256x128).

size
Read-only. The size of the texture.

smooth
Whether the smooth filter is enabled or not. When the filter is activated, the texture appears smoother so that pixels are less noticeable. However if you want the texture to look exactly the same as its source file, you should leave it disabled. The smooth filter is disabled by default.

width
Read-only. The width of the texture.

classmethod load_from_file(*filename*[, *area*])
Load the texture from a file on disk. This function is a shortcut for the following code:

```
        image = sfml.Image.load_from_file(filename)
        sfml.Texture.load_from_image(image, area)
```

area, if specified, may be either a tuple or an *IntRect*. Then only a sub-rectangle of the whole image will be loaded. If the area rectangle crosses the bounds of the image, it is adjusted to fit the image size.

The maximum size for a texture depends on the graphics driver and can be retrieved with the getMaximumSize function.

PySFMLException is raised if an error occurs.

classmethod load_from_image(*image*[, *area*])
Load the texture from an image.

area, if specified, may be either a tuple or an *IntRect*. Then only a sub-rectangle of the whole image will be loaded. If the area rectangle crosses the bounds of the image, it is adjusted to fit the image size.

The maximum size for a texture depends on the graphics driver and is accessible with the *MAXIMUM_SIZE* class attribute.

PySFMLException is raised if an error occurs.

classmethod load_from_memory(*bytes data*[, *area*])
Load the texture from a file in memory. This function is a shortcut for the following code:

```
image = sfml.Image.load_from_memory(data)
texture = sfml.Texture.load_from_image(image, area)
```

area, if specified, may be either a tuple or an *IntRect*. Then only a sub-rectangle of the whole image will be loaded. If the area rectangle crosses the bounds of the image, it is adjusted to fit the image size.

The maximum size for a texture depends on the graphics driver and is accessible with the *MAXIMUM_SIZE* class attribute.

PySFMLException is raised if an error occurs.

classmethod load_from_stream(*InputStream stream*[, *area*])
Load the texture from a custom stream. This class method is a shortcut for the following code:

```
image = sfml.Image.load_from_stream(stream)
texture = sfml.Texture.load_from_image(image, area)
```

area can a tuple of an *IntRect*, and is used to load only a sub-rectangle of the whole image. If you want the entire image then leave the default value (which is an empty *IntRect*). If the area rectangle crosses the bounds of the image, it is adjusted to fit the image size.

The maximum size for a texture depends on the graphics driver and can be retrieved with the *MAXIMUM_SIZE* class attribute.

PySFMLException is raised if an error occurs.

bind([*int coordinate_type*])
Activate the texture for rendering. This method is mainly used internally by the SFML rendering system. However it can be useful when using *Texture* with OpenGL code (this method is equivalent to `glBindTexture()`).

coordinate_type controls how texture coordinates will be interpreted. If *NORMALIZED* (the default), they must be in range [0 .. 1], which is the default way of handling texture coordinates with OpenGL. If *PIXELS*, they must be given in pixels (range [0 .. size]). This mode is used internally by the graphics classes of SFML, it makes the definition of texture coordinates more intuitive for the high-level API, users don't need to compute normalized values.

copy_to_image()
Copy the texture pixels to an image and return it. This method performs a slow operation that downloads the texture's pixels from the graphics card and copies them to a new image, potentially applying transformations to pixels if necessary (texture may be padded or flipped).

update(*source*, ...)
This method can be called in three ways, to be consistent with the C++ method overloading:

```
update(bytes pixels[, width, height, x, y])
```

Update a part of the texture from an array of pixels. The size of *pixels* must match the width and height arguments, and it must contain 32-bits RGBA pixels. No additional check is performed on the size of the pixel array or the bounds of the area to update, passing invalid arguments will lead to an undefined behaviour.

```
update(image[, x, y])
```

Update the texture from an image. Although the source image can be smaller than the texture, it's more convenient to use the *x* and *y* parameters for updating a sub-area of the texture.

```
update(window[, x, y])
```

Update the texture from the contents of a window. Although the source window can be smaller than the texture, it's more convenient to use the *x* and *y* parameters for updating a sub-area of the texture. No additional check is performed on the size of the window, passing a window bigger than the texture will lead to an undefined behaviour.

class sfml.**Sprite**([*texture*])

This class inherits *Transformable*.

Drawable representation of a texture, with its own transformations, color, etc.

It inherits all the attributes from *Transformable*: position, rotation, scale, origin. It also adds sprite-specific properties such as the texture to use, the part of it to display, and some convenience functions to change the overall color of the sprite, or to get its bounding rectangle.

Sprite works in combination with the *Texture* class, which loads and provides the pixel data of a given texture.

The separation of Sprite and *Texture* allows more flexibility and better performances: indeed a *Texture* is a heavy resource, and any operation on it is slow (often too slow for real-time applications). On the other side, a sf::Sprite is a lightweight object which can use the pixel data of a *Texture* and draw it with its own transformation/color/blending attributes.

Usage example:

```
# Load a texture
texture = sfml.Texture.load_from_file('texture.png')

# Create a sprite
sprite = sfml.Sprite(texture)
sprite.texture_rect = sfml.IntRect(10, 10, 50, 30)
sprite.color = sfml.Color(255, 255, 255, 200)
sprite.position = (100, 25)

# Draw it
window.draw(sprite)
```

color

The global color of the sprite. This color is modulated (multiplied) with the sprite's texture. It can be used to colorize the sprite, or change its global opacity. By default, the sprite's color is opaque white.

global_bounds

Read-only. The global bounding rectangle of the entity, as a *FloatRect*.

The returned rectangle is in global coordinates, which means that it takes into account the transformations (translation, rotation, scale, ...) that are applied to the entity. In other words, this function returns the bounds of the sprite in the global 2D world's coordinate system.

local_bounds

Read-only. The local bounding rectangle of the entity, as a *FloatRect*.

The returned rectangle is in local coordinates, which means that it ignores the transformations (translation, rotation, scale, ...) that are applied to the entity. In other words, this function returns the bounds of the entity in the entity's coordinate system.

texture

The source *Texture* of the sprite, or None if no texture has been set. Also see *set_texture()*, which lets you provide another argument.

copy()
Return a new Sprite object with the same value. The new sprite's texture is the same as the current one (no new texture is created).

get_texture_rect()
Return the sub-rectangle of the texture displayed by the sprite, as an *IntRect*. The texture rect is useful when you only want to display a part of the texture. By default, the texture rect covers the entire texture.

Warning: This method returns a copy of the rectangle, so code like this won't work as expected:

`sprite.get_texture_rect().top = 10`
`# Or this:`
`rect = sprite.get_texture_rect()`
`rect.top = 10`

Instead, you need to call *set_texture_rect()* with the desired rect:

`rect = sprite.get_texture_rect()`
`rect.top = 10`
`sprite.set_texture_rect(rect)`

set_texture(*texture*[, *reset_rect=False*])
Set the source *Texture* of the sprite. If *reset_rect* is `True`, the texture rect of the sprite is automatically adjusted to the size of the new texture. If it is `False`, the texture rect is left unchanged.

set_texture_rect(*rect*)
Set the sub-rectangle of the texture displayed by the sprite, as an *IntRect*. The texture rect is useful when you only want to display a part of the texture. By default, the texture rect covers the entire texture. *rect* may be an *IntRect* or a tuple.

7.3.4 Text

class `sfml.`**Font**
The constructor will raise `NotImplementedError` if called. Use class methods like *load_from_file()* or *load_from_memory()* instead.

The following types of fonts are supported: TrueType, Type 1, CFF, OpenType, SFNT, X11 PCF, Windows FNT, BDF, PFR and Type 42.

Once it's loaded, you can retrieve three types of information about the font:

- Global metrics, such as the line spacing.
- Per-glyph metrics, such as bounding box or kerning.
- Pixel representation of glyphs.

Fonts alone are not very useful: they hold the font data but cannot make anything useful of it. To do so you need to use the *Text* class, which is able to properly output text with several options such as character size, style, color, position, rotation, etc. This separation allows more flexibility and better performances: a font is a heavy resource, and any operation on it is slow (often too slow for real-time applications). On the other hand, a *Text* is a lightweight object which can combine the glyphs data and metrics of a font to display any text on a render target. Note that it is also possible to bind several text instances to the same font.

Usage example:

```
# Load a font from a file, catch PySFMLException
# if you want to handle the error
```

```
font = sfml.Font.load_from_file('arial.ttf')

# Create a text which uses our font
text1 = sfml.Text()
text1.font = font
text1.character_size = 30
text1.style = sfml.Text.REGULAR

# Create another text using the same font, but with different parameters
text2 = sfml.Text()
text2.font = font
text2.character_size = 50
text1.style = sfml.Text.ITALIC
```

Apart from loading font files, and passing them to instances of *Text*, you should normally not have to deal directly with this class. However, it may be useful to access the font metrics or rasterized glyphs for advanced usage.

DEFAULT_FONT
The default font (Arial), as a class attribute:

```
print sfml.Font.DEFAULT_FONT
```

This font is provided for convenience, it is used by text instances by default. It is provided so that users don't have to provide and load a font file in order to display text on screen.

classmethod load_from_file(*filename*)
Load the font from *filename*, and return a new font object.

Note that this class method knows nothing about the standard fonts installed on the user's system, so you can't load them directly.

PySFMLException is raised if an error occurs.

classmethod load_from_memory(*bytes data*)
Load the font from the string/bytes object (for Python 2/3, respectively) and return a new font object.

> **Warning:** SFML cannot preload all the font data in this function, so you should keep a reference to the *data* object as long as the font is used.

classmethod load_from_stream(*InputStream stream*)
Load the font from a custom stream.

get_glyph(*int code_point*, *int character_size*, *bool bold*)
Return a glyph corresponding to *code_point* and *character_size*.

get_texture(*int character_size*)
Retrieve the texture containing the loaded glyphs of a certain size.

The contents of the returned texture changes as more glyphs are requested, thus it is not very relevant. It is mainly used internally by *Text*.

get_kerning(*int first*, *int second*, *int character_size*)
Return the kerning offset of two glyphs.

The kerning is an extra offset (negative) to apply between two glyphs when rendering them, to make the pair look more "natural". For example, the pair "AV" have a special kerning to make them closer than other characters. Most of the glyphs pairs have a kerning offset of zero, though.

get_line_spacing(*int character_size*)
Get the line spacing.

Line spacing is the vertical offset to apply between two consecutive lines of text.

class sfml.**Glyph**

A glyph is the visual representation of a character. *Glyph* structure provides the information needed to handle the glyph:

- its coordinates in the font's texture,
- its bounding rectangle,
- the offset to apply to get the starting position of the next glyph.

advance

Offset to move horizontically to the next character.

bounds

Bounding rectangle of the glyph as an *IntRect*, in coordinates relative to the baseline.

texture_rect

Texture coordinates of the glyph inside the font's texture, as an *IntRect*.

class sfml.**Text**([*string*, *font*, *character_size=0*])

This class inherits *Transformable*.

string can be a bytes/str/unicode object. SFML will internally store characters as 32-bit integers. A bytes object (str in Python 2) will end up being interpreted by SFML as an "ANSI string" (cp1252 encoding). A unicode object (str in Python 3) will be interpreted as 32-bit code points.

Text is a drawable class that allows to easily display some text with custom style and color on a render target.

It inherits all the functions from *Transformable*: position, rotation, scale, origin. It also adds text-specific properties such as the font to use, the character size, the font style (bold, italic, underlined), the global color and the text to display of course. It also provides convenience functions to calculate the graphical size of the text, or to get the global position of a given character.

Text works in combination with the *Font* class, which loads and provides the glyphs (visual characters) of a given font. The separation of *Font* and *Text* allows more flexibility and better performances: a *Font* is a heavy resource, and any operation on it is slow (often too slow for real-time applications). On the other hand, a *Text* is a lightweight object which can combine the glyphs data and metrics of a *Font* to display any text on a render target.

Usage example:

```
# Declare and load a font
font = sfml.Font.loadFromFile('arial.ttf')

# Create a text
text = sfml.Text('hello')
text.font = font
text.character_size = 30
text.style = sfml.Text.BOLD
text.color = sfml.Color.RED

# Draw it
window.draw(text)
```

Note that you don't need to load a font to draw text, SFML comes with a built-in font that is implicitely used by default.

character_size

The size of the characters, pixels. The default size is 30.

color
The global color of the text. The default color is opaque white.

font
The text's font. The default font is *Font.DEFAULT_FONT*.

global_bounds
Read-only. The global bounding rectangle of the entity, as a *FloatRect*. The returned rectangle is in global coordinates, which means that it takes in account the transformations (translation, rotation, scale, ...) that are applied to the entity. In other words, this function returns the bounds of the sprite in the global 2D world's coordinate system.

local_bounds
Read-only. The local bounding rectangle of the entity, as a *FloatRect*. The returned rectangle is in local coordinates, which means that it ignores the transformations (translation, rotation, scale, ...) that are applied to the entity. In other words, this function returns the bounds of the entity in the entity's coordinate system.

string
This attribute can be set as either a `str` or `unicode` object. The value retrieved will be either `str` or `unicode` as well, depending on what type has been set before. See *Text* for more information.

style
Can be one or more of the following:

- `sfml.Text.REGULAR`
- `sfml.Text.BOLD`
- `sfml.Text.ITALIC`
- `sfml.Text.UNDERLINED`

Example:

```
text.style = sfml.Text.BOLD | sfml.Text.ITALIC
```

find_character_pos(*int index*)
Return the position of the *index*-th character. This method computes the visual position of a character from its index in the string. The returned position is in global coordinates (translation, rotation, scale and origin are applied). If *index* is out of range, the position of the end of the string is returned.

7.4 Events

7.4.1 Event types reference

Type	Attributes	Remarks
Event.CLOSED		In fullscreen, Alt + F4 won't send the `CLOSED` event (on GNU/Linux, at least).
Event.RESIZED	`width`, `height`	
Event.LOST_FOCUS		
Event.GAINED_FOCUS		
Event.TEXT_ENTERED	`unicode`	The attribute lets you retrieve the character entered by the user, as a Unicode string.
Event.KEY_PRESSED, *Event.KEY_RELEASED*	`code`, `alt`, `control`, `shift`, `system`	`code` is the code of the key that was pressed/released, the other attributes are booleans and tell you if the alt/control/shit/system modifier was pressed.
Event.MOUSE_WHEEL_MOVED	`delta`, `x`, `y`	The attribute contains the mouse wheel move (positive if forward, negative if backward).
Event.MOUSE_BUTTON_PRESSED, *Event.MOUSE_BUTTON_RELEASED*	`button`, `x`, `y`	See the *Mouse* class for the button codes.
Event.MOUSE_MOVED	`x`, `y`	
Event.MOUSE_ENTERED		
Event.MOUSE_LEFT		
Event.JOYSTICK_BUTTON_PRESSED, *Event.JOYSTICK_BUTTON_RELEASED*	`joystick_id`, `button`	`button` is a number between 0 and *Joystick.BUTTON_COUNT*- 1.
Event.JOYSTICK_MOVED	`joystick_id`, `axis`, `position`	See the *Joystick* class for the axis codes.
Event.JOYSTICK_CONNECTED, *Event.JOYSTICK_DISCONNECTED*	`joystick_id`	

class `sfml.`**`Event`**

This class behaves differently from the C++ `sf::Event` class. Every Event object will always only feature the attributes that actually make sense regarding the event type. This means that there is no need for the C++ union; you just access whatever attribute you want.

For example, this is the kind of code you'd write in C++:

```
if (event.Type == sf::Event::KeyPressed &&
    event.Key.Code == sf::Keyboard::Escape)
{
    // ...
}
```

In Python, it becomes:

```
if event.type == sfml.Event.KEY_PRESSED and event.code == sfml.Keyboard.ESCAPE:
    # ...
```

Note: All the events have *Event* type. There are no specific subtypes like `KeyPressedEvent` or `MouseEnteredEvent`. Instead, events are common Python objects in the sense that their attributes can be modified at runtime, unlike other pySFML objects. This is how their specific attributes are set.

This class provides the following special methods:

- `str(event)` returns a description of the event with its name and its attributes.

NAMES
A class attribute that maps event codes to a short description:

```
>>> sfml.Event.NAMES[sfml.Event.CLOSED]
'Closed'
>>> sfml.Event.NAMES[sfml.Event.KEY_PRESSED]
'Key pressed'
```

If you want to print this information about a specific object, you can simply use `print`; `Event.__str__()` will look up the description for you.

Event types:

CLOSED
The window requested to be closed.

RESIZED
The window was resized.

LOST_FOCUS
The window lost focus.

GAINED_FOCUS
The window gained focus.

TEXT_ENTERED
A character was entered.

KEY_PRESSED
A key was pressed.

KEY_RELEASED
A key was released.

MOUSE_WHEEL_MOVED
The mouse wheel was scrolled.

MOUSE_BUTTON_PRESSED
A mouse button was pressed.

MOUSE_BUTTON_RELEASED
A mouse button was released.

MOUSE_MOVED
The mouse cursors moved.

MOUSE_ENTERED
The mouse cursor entered the area of the window.

MOUSE_LEFT
The mouse cursor entered the area of the window.

JOYSTICK_BUTTON_PRESSED
A joystick button was pressed.

JOYSTICK_BUTTON_RELEASED
A joystick button was released.

JOYSTICK_MOVED
The joystick moved along an axis.

JOYSTICK_CONNECTED
A joystick was connected.

JOYSTICK_DISCONNECTED
A joystick was disconnected.

class sfml.**Joystick**
This class gives access to the real-time state of the joysticks.

It only contains static functions, so it's not meant to be instanciated. Instead, each joystick is identified by an index that is passed to the functions of this class. Calling the constructor will raise `NotImplementedError`.

This class allows users to query the state of joysticks at any time and directly, without having to deal with a window and its events. Compared to the *Event.JOYSTICK_MOVED*, *Event.JOYSTICK_BUTTON_PRESSED* and *Event.JOYSTICK_BUTTON_RELEASED* events, this class can retrieve the state of axes and buttons of joysticks at any time (you don't need to store and update a boolean on your side in order to know if a button is pressed or released), and you always get the real state of joysticks, even if they are moved, pressed or released when your window is out of focus and no event is triggered.

SFML supports:

- 8 joysticks (*COUNT*)
- 32 buttons per joystick (*BUTTON_COUNT*)
- 8 axes per joystick (*AXIS_COUNT*)

Unlike the keyboard or mouse, the state of joysticks is sometimes not directly available (depending on the OS), so the *update()* method must be called in order to update the current state of joysticks. When you have a window with event handling, this is done automatically, you don't need to call anything. But if you have no window, or if you want to check joysticks state before creating one, you must call *update()* explicitely.

Usage example:

```
# Is joystick #0 connected?
connected = sfml.Joystick.is_connected(0)

# How many buttons does joystick #0 support?
buttons = sfml.Joystick.get_button_count(0)

# Does joystick #0 define a X axis?
has_x = sfml.Joystick.has_axis(0, sfml.Joystick.X)

# Is button #2 pressed on joystick #0?
pressed = sfml.Joystick.is_button_pressed(0, 2)

# What's the current position of the Y axis on joystick #0?
position = sfml.Joystick.get_axis_position(0, sfml.Joystick.Y)
```

COUNT
The maximum number of supported joysticks.

BUTTON_COUNT
The maximum number of supported buttons.

AXIS_COUNT
The maximum number of supported axes.

Axes codes:

X
The *x* axis.

Y
The *y* axis.

Z
The *z* axis.

R
The *r* axis.

U
The *u* axis.

V
The *v* axis.

POV_X
The *x* axis of the point-of-view hat.

POV_Y
The *y* axis of the point-of-view hat.

classmethod is_connected(*int joystick*)
Return `True` is *joystick* is connected, otherwise `False` is returned.

classmethod get_button_count(*int joystick*)
Return the number of buttons supported by *joystick*. If the joystick is not connected, return 0.

classmethod has_axis(*int joystick*, *int axis*)
Return whether *joystick* supports the given *axis*. If the joystick isn't connected, `False` is returned. *axis* should be an *axis code*.

classmethod is_button_pressed(*int joystick*, *int button*)
Return whether *button* is pressed on *joystick*. If the joystick isn't connected, `False` is returned.

classmethod get_axis_position(*int joystick*, *int axis*)
Return the current position along *axis* as a float. If the joystick is not connected, 0.0 is returned. *axis* should be an *axis code*.

classmethod update()
Update the state of all the joysticks. You don't need to call this method yourself in most cases. If you haven't created any window, however, you will need to call it to update the joystick state.

class `sfml.`**Keyboard**
This class provides an interface to the state of the keyboard. It only contains static methods (a single keyboard is assumed), so it's not meant to be instanciated.

This class allows users to query the keyboard state at any time and directly, without having to deal with a window and its events. Compared to the *Event.KEY_PRESSED* and *Event.KEY_RELEASED* events, Keyboard can retrieve the state of a key at any time (you don't need to store and update a boolean on your side in order to know if a key is pressed or released), and you always get the real state of the keyboard, even if keys are pressed or released when your window is out of focus and no event is triggered.

Usage example:

```
if sfml.Keyboard.is_key_pressed(sfml.Keyboard.LEFT):
    pass # move left...
elif sfml.Keyboard.is_key_pressed(sfml.Keyboard.RIGHT):
    pass # move right...
elif sfml.Keyboard.is_key_pressed(sfml.Keyboard.ESCAPE):
    pass # quit...
```

Key codes:

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z

NUM0
The 0 key.

NUM1
The 1 key.

NUM2
The 2 key.

NUM3
The 3 key.

NUM4
The 4 key.

NUM5
The 5 key.

NUM6
The 6 key.

NUM7
The 7 key.

NUM8
The 8 key.

NUM9
The 9 key.

ESCAPE

L_CONTROL
The left control key.

L_SHIFT
The left shift key.

L_ALT
The left alt key.

L_SYSTEM
The left OS-specific key, e.g. window, apple or home key.

R_CONTROL
The right control key.

R_SHIFT
The right shift key.

R_ALT
The right alt key.

R_SYSTEM
The right OS-specific key, e.g. window, apple or home key.

MENU
The menu key.

L_BRACKET
The [key.

R_BRACKET
The] key.

SEMI_COLON
The ; key.

COMMA
The , key.

PERIOD
The . key.

QUOTE
The ' key.

SLASH
The / key.

BACK_SLASH
The \ key.

TILDE
The ~ key.

EQUAL
The = key.

DASH
The − key.

SPACE

RETURN

BACK_SPACE
The back space key.

TAB
The tabulation key.

PAGE_UP

PAGE_DOWN

END

HOME

INSERT

DELETE

ADD
The + key.

SUBTRACT
The − key.

MULTIPLY
The ⋆ key.

DIVIDE
The / key.

LEFT
The left arrow.

RIGHT
The right arrow.

UP
The up arrow.

DOWN
The down arrow.

NUMPAD0
The numpad 0 key.

NUMPAD1
The numpad 1 key.

NUMPAD2
The numpad 2 key.

NUMPAD3
The numpad 3 key.

NUMPAD4
The numpad 4 key.

NUMPAD5
The numpad 5 key.

NUMPAD6
The numpad 6 key.

NUMPAD7
The numpad 7 key.

NUMPAD8
The numpad 8 key.

NUMPAD9
The numpad 9 key.

F1

F2

F3

F4

F5

F6

F7

F8

F9

F10

F11

F12

F13

F14

F15

PAUSE

KEY_COUNT
The total number of keyboard keys.

classmethod is_key_pressed(*int key*)
Return `True` if *key* is pressed, otherwise `False` is returned. *key* should a value from the *key codes*.

class `sfml`**.Mouse**
This class gives access to the real-time state of the mouse. It only contains static functions (a single mouse is assumed), so it's not meant to be instanciated. Calling the constructor will raise `NotImplementedError`.

This class allows users to query the mouse state at any time and directly, without having to deal with a window and its events. Compared to the *Event.MOUSE_MOVED*, *Event.MOUSE_BUTTON_PRESSED* and *Event.MOUSE_BUTTON_RELEASED* events, this class can retrieve the state of the cursor and the buttons at any time (you don't need to store and update a boolean on your side in order to know if a button is pressed or released), and you always get the real state of the mouse, even if it is moved, pressed or released when your window is out of focus and no event is triggered.

The *set_position()* and *get_position()* methods can be used to change or retrieve the current position of the mouse pointer. There are two versions: one that operates in global coordinates (relative to the desktop) and one that operates in window coordinates (relative to a specific window).

Usage example:

```
if sfml.Mouse.is_button_pressed(sfml.Mouse.LEFT):
    pass # left click...

# Get global mouse position
position = sfml.Mouse.get_position()

# Set mouse position relative to a window
sfml.Mouse.set_position((100, 200), window)
```

Mouse buttons codes:

LEFT
The left mouse button.

RIGHT
The right mouse button.

MIDDLE
The middle (wheel) mouse button.

X_BUTTON1
The first extra mouse button.

X_BUTTON2
The second extra mouse button.

BUTTON_COUNT
The total number of mouse buttons.

classmethod is_button_pressed(*int button*)
Return `True` if *button* is pressed, otherwise returns `False`. *button* should be a *mouse button code*.

classmethod get_position(*window*)
Return a tuple with the current position of the cursor. With no arguments, the global position on the desktop is returned. If a *window* argument is provided, the position relative to the window is returned.

classmethod set_position(*tuple position*[, *window*])
Set the current position of the cursor. With only one argument, *position* is considered a as global desktop position. If a *window* argument is provided, the position is considered as relative to the window.

7.5 Audio

class sfml.**Chunk**
A chunk of audio data to stream. See *SoundStream*.

samples
Should be a string in Python 2, and bytes in Python 3.

class sfml.**Listener**
The audio listener is the point in the scene from where all the sounds are heard. The audio listener defines the global properties of the audio environment: where and how sounds and musics are heard.

If *View* is the eyes of the user, then *Listener* is his ears (they are often linked together – same position, orientation, etc.).

Because the listener is unique in the scene, this class only contains static functions and doesn't have to be instanciated. Calling the constructor will raise `NotImplementedError`.

Usage example:

```
# Move the listener to the position (1, 0, -5)
sfml.Listener.set_position(1, 0, -5)

# Make it face the right axis (1, 0, 0)
sfml.Listener.set_direction(1, 0, 0)

# Reduce the global volume
sfml.Listener.set_global_volume(50)
```

classmethod get_direction()
Get the current direction of the listener in the scene, as a tuple of three floats.

classmethod get_global_volume()
Get the current value of the global volume, as a `float`.

classmethod get_position()
Get the current position of the listener in the scene, as a tuple of three floats.

classmethod set_global_volume(*float volume*)
Change the global volume of all the sounds and musics.

The volume is a number between 0 and 100; it is combined with the individual volume of each sound / music. The default value for the volume is 100 (maximum).

classmethod set_direction(*float x*, *float y*, *float z*)
Set the orientation of the listener in the scene.

The orientation defines the 3D axes of the listener (left, up, front) in the scene. The orientation vector doesn't have to be normalized. The default listener's orientation is (0, 0, -1).

classmethod set_position(*float x*, *float y*, *float z*)
Set the position of the listener in the scene.

The default listener's position is (0, 0, 0).

class `sfml`.**Music**
This class inherits *SoundStream*. Will raise `NotImplementedError` if the constructor is called. Use class methods instead.

Streamed music played from an audio file. Musics are sounds that are streamed rather than completely loaded in memory.

This is especially useful for compressed musics that usually take hundreds of MB when they are uncompressed: by streaming it instead of loading it entirely, you avoid saturating the memory and have almost no loading delay.

Apart from that, a Music object has almost the same features as the *SoundBuffer*/*Sound* pair: you can play/pause/stop it, request its parameters (channels, sample rate), change the way it is played (pitch, volume, 3D position, ...), etc.

As a sound stream, a music is played in its own thread in order not to block the rest of the program. This means that you can leave the music alone after calling `play()`, it will manage itself very well.

Here is a list of all the supported formats: ogg, wav, flac, aiff, au, raw, paf, svx, nist, voc, ircam, w64, mat4, mat5 pvf, htk, sds, avr, sd2, caf, wve, mpc2k, and rf64.

Usage example:

```
# Create a new music object
music = sfml.Music.open_from_file('music.ogg')

# Change some parameters
music.position = (0, 1, 10) # change its 3D position
music.pitch = 2             # increase the pitch
music.volume = 50           # reduce the volume
music.loop = true           # make it loop

# Play it
music.play()
```

duration
Read-only. The total duration of the music, as a *Time* object.

classmethod open_from_file(*filename*)
Open a music from an audio file. This function doesn't start playing the music (call `play()` to do so).

PySFMLException is raised if an error occurs.

classmethod open_from_memory(*str data*)
Open a music from an audio file in memory. This function doesn't start playing the music (call `play()` to do so).

PySFMLException is raised if an error occurs.

classmethod open_from_stream(*InputStream stream*)
Open a music from an audio file in a custom stream. This class method doesn't start playing the music (call play() to do so).

PySFMLException is raised if an error occurs.

class `sfml.`**Sound**(*[SoundBuffer buffer]*)
Sound is the class to use to play sounds. It provides:

- Control (play, pause, stop)
- Ability to modify output parameters in real-time (pitch, volume, ...)
- 3D spatial features (position, attenuation, ...).

Sound is perfect for playing short sounds that can fit in memory and require no latency, like foot steps or gun shots. For longer sounds, like background musics or long speeches, see *Music*, which is based on streaming.

In order to work, a sound must be given a buffer of audio data to play. Audio data (samples) is stored in a *SoundBuffer*, and attached to a sound with the *buffer* attribute, or as a constructor argument. Note that multiple sounds can use the same sound buffer at the same time.

Usage example:

```
buf = sfml.SoundBuffer.load_from_file('sound.wav')
sound = sfml.Sound()
sound.buffer = buf
sound.play()
```

attenuation
The attenuation factor of the sound.

buffer
The audio buffer attached to the sound.

loop
Whether or not the sound is in loop mode.

min_distance
The minimum distance of the sound.

pitch
The pitch of the sound.

playing_offset
The current playing position of the sound, as a *Time* object.

position
The 3D position of the sound in the audio scene, as a three elements tuple.

relative_to_listener
Whether the sound's position is relative to the listener or absolute.

status
Read-only. Can be one of:

- `sfml.Sound.STOPPED`
- `sfml.Sound.PAUSED`
- `sfml.Sound.PLAYING`

volume
A value between 0 (muted) and 100 (full volume and default value).

pause()
Pause the sound. This method has no effect if the sound isn't playing.

play()
Start or resume playing the sound. This method restarts the sound from its beginning if it's already playing. It uses its own thread so that it doesn't block the rest of the program while the sound is played.

stop()
Stop playing the sound and reset the playing position. This method has no effect is the sound is already stopped.

class `sfml.`**SoundBuffer**
The constructor will raise `NotImplementedError`. Use one of the class methods instead.

Storage for audio samples defining a sound.

A sound buffer holds the data of a sound, which is an array of audio samples.

A sample is a 16 bits signed integer that defines the amplitude of the sound at a given time. The sound is then restituted by playing these samples at a high rate (for example, 44100 samples per second is the standard rate used for playing CDs). In short, audio samples are like texture pixels, and a SoundBuffer is similar to a *Texture*.

A sound buffer can be loaded from a file (see *load_from_file()* for the complete list of supported formats), from memory or directly from a list of samples. It can also be saved back to a file.

Here is the list of all the supported formats: ogg, wav, flac, aiff, au, raw, paf, svx, nist, voc, ircam, w64, mat4, mat5 pvf, htk, sds, avr, sd2, caf, wve, mpc2k, and rf64. (Note that mp3 isn't supported.)

Sound buffers alone are not very useful: they hold the audio data but cannot be played. To do so, you need to use the *Sound* class, which provides functions to play/pause/stop the sound as well as changing the way it is outputted (volume, pitch, 3D position, ...). This separation allows more flexibility and better performances: a SoundBuffer is a heavy resource, and any operation on it is slow (often too slow for real-time applications). On the other hand, a *Sound* is a lightweight object, which can use the audio data of a sound buffer and change the way it is played without actually modifying that data. Note that it is also possible to bind several *Sound* instances to the same SoundBuffer.

Usage example:

```
# Create a new sound buffer
buf = sfml.SoundBuffer.load_from_file('sound.wav')

# Create a sound source and bind it to the buffer
sound1 = sfml.Sound()
sound1.buffer = buf

# Play the sound
sound1.play()

# Create another sound source bound to the same buffer, this time
# passing it to the constructor instead of using the buffer property
sound2 = sfml.Sound(buf)

# Play it with a higher pitch -- the first sound remains unchanged
sound2.pitch = 2
sound2.play()
```

channel_count
Read-only. The number of channels used by the sound (1 for mono, 2 for stereo, etc.).

duration
The total duration of the sound, as a *Time* object.

sample_rate
The sample rate of the sound. This is the number of samples played per second. The higher, the better the quality (for example, 44100 samples/s is CD quality).

samples
The samples stored in the buffer, as a byte string (`str` in Python 2, `bytes` in Python 3). Use `len()` to get the number of samples.

classmethod load_from_file(*filename*)
Load the sound buffer from a file.

PySFMLException is an error occurs.

classmethod load_from_memory(*bytes data*)
Load the sound buffer from a file in memory. *data* should be `str` object in Python 2, and a `bytes` object in Python 3.

PySFMLException is raised if an error occurs.

classmethod load_from_samples(*list samples*, *int channel_count*, *int sample_rate*)
Load the sound buffer from a list of audio samples. *samples* should be a `bytes` object in Python 3, and a string in Python 2. Each sample must be stored on two bytes (`Int16` in C++ SFML).

PySFMLException is raised if an error occurs.

load_from_stream(*InputStream stream*)
Load the sound buffer from a custom stream.

PySFMLException is an error occurs.

save_to_file(*filename*)
Save the sound buffer to an audio file.

PySFMLException is raised if an error occurs.

class `sfml.`**SoundStream**
Abstract class for streamed audio sources.

Unlike audio buffers such as *SoundBuffer*, audio streams are never completely loaded in memory. Instead, the audio data is acquired continuously while the stream is playing. This behaviour allows to play a sound with no loading delay, and keeps the memory consumption very low.

To create your own sound stream, you must inherit this class and at least define a `on_get_data()` method that receives a *Chunk* parameter. `on_seek(Time)` may be implemented as well. Any exception raised in these two methods will be printed to `sys.stdout` and swallowed. This is because it doesn't seem possible to catch an exception raised in another thread, or at least it doesn't seem reliable. So try to keep them as short as possible, and if they don't work, check the console. See `examples/soundstream.py` for an example.

My streaming tests show that this class is still too slow. I optimized it as much as I could, and I'm not sure how to improve it now. Also, `on_seek()` seems to hang the program when seeking is used.

attenuation
The attenuation factor of the sound.

channel_count
Read-only. The number of channels used by the sound (1 for mono, 2 for stereo, etc.).

loop
Whether or not the stream is in loop mode.

min_distance
The minimum distance of the sound.

pitch
The pitch of the sound.

playing_offset
The current position of the stream, as a *Time* object.

position
The 3D position of the sound the audio scene, as a three elements tuple.

relative_to_listener
Whether the sound's position is relative to the listener or absolute.

sample_rate
Read-only. The sample rate of the stream. This is the number of audio samples played per second. The higher, the better the quality.

status
Read-only. Can be one of:

- `sfml.SoundStream.STOPPED`
- `sfml.SoundStream.PAUSED`
- `sfml.SoundStream.PLAYING`

volume
A value between 0 (muted) and 100 (full volume and default value).

initialize(*int channel_count*, *int sample_rate*)
This method must be called by user-defined streams. It's not available from built-in sound streams such as *Music*.

pause()
Pause the stream. This method has no effect if the stream isn't playing.

play()
Start or resume playing the stream. This method restarts the stream from its beginning if it's already playing. It uses its own thread so that it doesn't block the rest of the program while the stream is played.

stop()

Stop playing the stream and reset the playing position. This method has no effect is the stream is already stopped.

CHAPTER 8

Licenses

8.1 Project license

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY BASTIEN LÉONARD "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL BASTIEN LÉONARD OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

8.2 Documentation license

Redistribution and use in source (reStructuredText) and 'compiled' forms (HTML, PDF, PostScript, RTF and so forth) with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code (reStructuredText) must retain the above copyright notice, this list of conditions and the following disclaimer as the first lines of this file unmodified.
2. Redistributions in compiled form (converted to HTML, PDF, PostScript, RTF and other formats) must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

THIS DOCUMENTATION IS PROVIDED BY BASTIEN LÉONARD "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL BASTIEN

LÉONARD BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS DOCUMENTATION, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

CHAPTER 9

Indices and tables

- genindex
- modindex
- search

S

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z